

CN101

Lecture 6-7

Repetition Structures

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

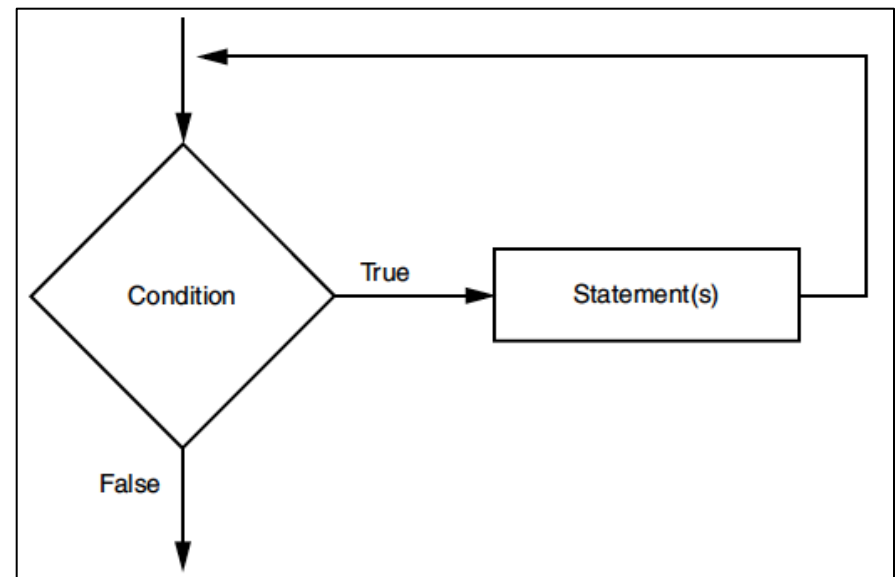
Introduction to Repetition Structures

- 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
- Repetition structure: 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:  
    statement  
    statement  
    etc.
```



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

Program Output (with input shown in bold)

Enter the amount of sales: **10000.00**

Enter the commission rate: **0.10**

The commission is \$1,000.00

Do you want to calculate another commission (Enter y for yes): **y**

Enter the amount of sales: **20000.00**

Enter the commission rate: **0.15**

The commission is \$3,000.00

Do you want to calculate another commission (Enter y for yes): **y**

Enter the amount of sales: **12000.00**

Enter the commission rate: **0.10**

The commission is \$1,200.00

Do you want to calculate another commission (Enter y for yes): **n**

This condition is tested.

```
while keep_going == 'y':
```

If the condition is true,
these statements are
executed, and then the
loop starts over.

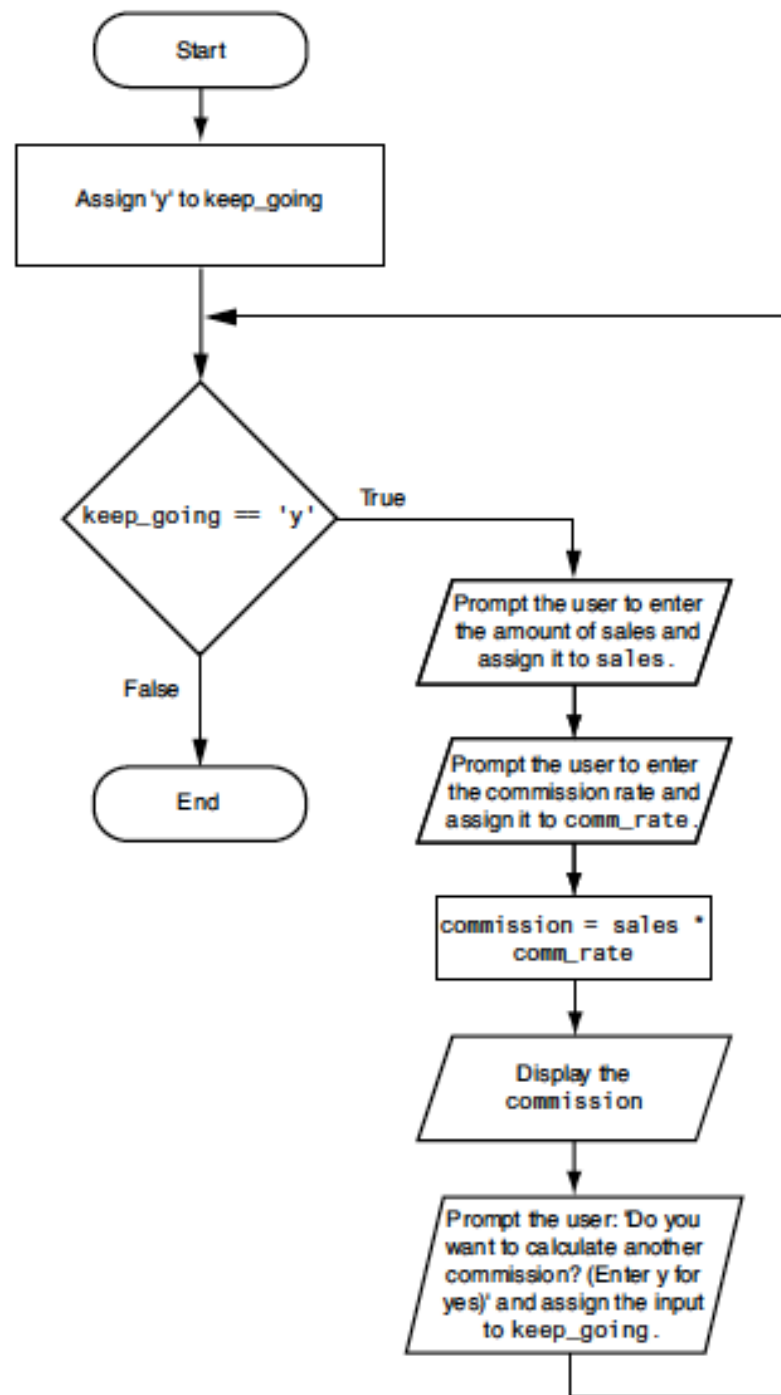
If the condition is false,
these statements are
skipped, and the
program exits the loop.

```
# 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): ')
```

Program 4-2 (temperature.py)

10

```
1  # This program assists a technician in the process
2  # of checking a substance's temperature.
3
4  # Named constant to represent the maximum
5  # temperature.
6  MAX_TEMP = 102.5
7
8  # Get the substance's temperature.
9  temperature = float(input("Enter the substance's Celsius temperature: "))
10
11 # As long as necessary, instruct the user to
12 # adjust the thermostat.
13 while temperature > MAX_TEMP:
14     print('The temperature is too high.')
15     print('Turn the thermostat down and wait')
16     print('5 minutes. Then take the temperature')
17     print('again and enter it.')
18     temperature = float(input('Enter the new Celsius temperature: '))
19
20 # Remind the user to check the temperature again
21 # in 15 minutes.
22 print('The temperature is acceptable.')
23 print('Check it again in 15 minutes.')
```

Program Output (with input shown in bold)

Enter the substance's Celsius temperature: **104.7**

The temperature is too high.

Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.

Enter the new Celsius temperature: **103.2**

The temperature is too high.

Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.

Enter the new Celsius temperature: **102.1**

The temperature is acceptable.

Check it again in 15 minutes.

Infinite Loops

- Loops must contain within themselves a way to terminate
 - Something inside a `while` loop must eventually make the condition false
- Infinite loop: 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

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

Program 4-4 (simple_loop1.py)


```
1  # This program demonstrates a simple for loop
2  # that uses a list of numbers.
3
4  print('I will display the numbers 1 through 5.')
5  for num in [1, 2, 3, 4, 5]:
6      print(num)
```

Program Output

I will display the numbers 1 through 5.


1
2
3
4
5

1st iteration:




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

2nd iteration:




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

3rd iteration:




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

4th iteration:



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

5th iteration:



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

Program 4-5 (simple_loop2.py)

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of numbers.
3
4  print('I will display the odd numbers 1 through 9.')
5  for num in [1, 3, 5, 7, 9]:
6      print(num)
```

Program Output

I will display the odd numbers 1 through 9.

1
3
5
7
9

Program 4-6 (simple_loop3.py)

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of strings.
3
4  for name in ['Winken', 'Blinken', 'Nod']:
5      print(name)
```

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
 - Two arguments: starting value and ending limit
 - Three arguments: third argument is step value

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

0
1
2
3
4

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

1
2
3
4

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

1
3
5
7
9

Program 4-7 (simple_loop4.py)

```
1  # This program demonstrates how the range
2  # function can be used with a for loop.
3
4  # Print a message five times.
5  for x in range(5):
6      print('Hello world')
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

Using the Target Variable Inside the Loop

- Purpose of target variable is to reference each item in a sequence as the loop iterates
- Target variable can be used in calculations or tasks in the body of the loop
 - 🟡 Example: calculate square of each number in a range

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

Program 4-8 (squares.py)

```
1  # This program uses a loop to display a
2  # table showing the numbers 1 through 10
3  # and their squares.
4
5  # Print the table headings.
6  print('Number\tSquare')
7  print('-----')
8
9  # Print the numbers 1 through 10
10 # and their squares.
11 for number in range(1, 11):
12     square = number**2
13     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

Letting the User Control the Loop Iterations

- Sometimes the programmer does not know exactly how many times the loop will execute
- Can receive range inputs from the user, place them in variables, and call the `range` function in the for clause using these variables
 - 🟡 Be sure to consider the end cases: `range` does not include the ending limit

Program 4-10 (user_squares1.py)

```
1  # This program uses a loop to display a
2  # table of numbers and their squares.
3
4  # Get the ending limit.
5  print('This program displays a list of numbers')
6  print('(starting at 1) and their squares.')
7  end = int(input('How high should I go? '))
8
9  # Print the table headings.
10 print()
11 print('Number\tSquare')
12 print('-----')
13
14 # Print the numbers and their squares.
15 for number in range(1, end + 1):
16     square = number**2
17     print(number, '\t', square)
```


Program Output (with input shown in bold)

This program displays a list of numbers
(starting at 1) and their squares.

How high should I go? **5**

Number	Square
1	1
2	4
3	9
4	16
5	25

```
1  # This program uses a loop to display a
2  # table of numbers and their squares.
3
4  # Get the starting value.
5  print('This program displays a list of numbers')
6  print('and their squares.')
7  start = int(input('Enter the starting number: '))
8
9  # Get the ending limit.
10 end = int(input('How high should I go? '))
11
12 # Print the table headings.
13 print()
14 print('Number\tSquare')
15 print('-----')
16
17 # Print the numbers and their squares.
18 for number in range(start, end + 1):
19     square = number**2
20     print(number, '\t', square)
```

Program Output (with input shown in bold)

This program displays a list of numbers and their squares.

Enter the starting number: **5**

How high should I go? **10**

Number	Square
--------	--------

5	25
---	----

6	36
---	----

7	49
---	----

8	64
---	----

9	81
---	----

10	100
----	-----

Generating an Iterable Sequence that Ranges from Highest to Lowest

- The `range` function can be used to generate a sequence with numbers in descending order
 - Make sure starting number is larger than end limit, and step value is negative
 - Example: `range (5, 0, -1)`

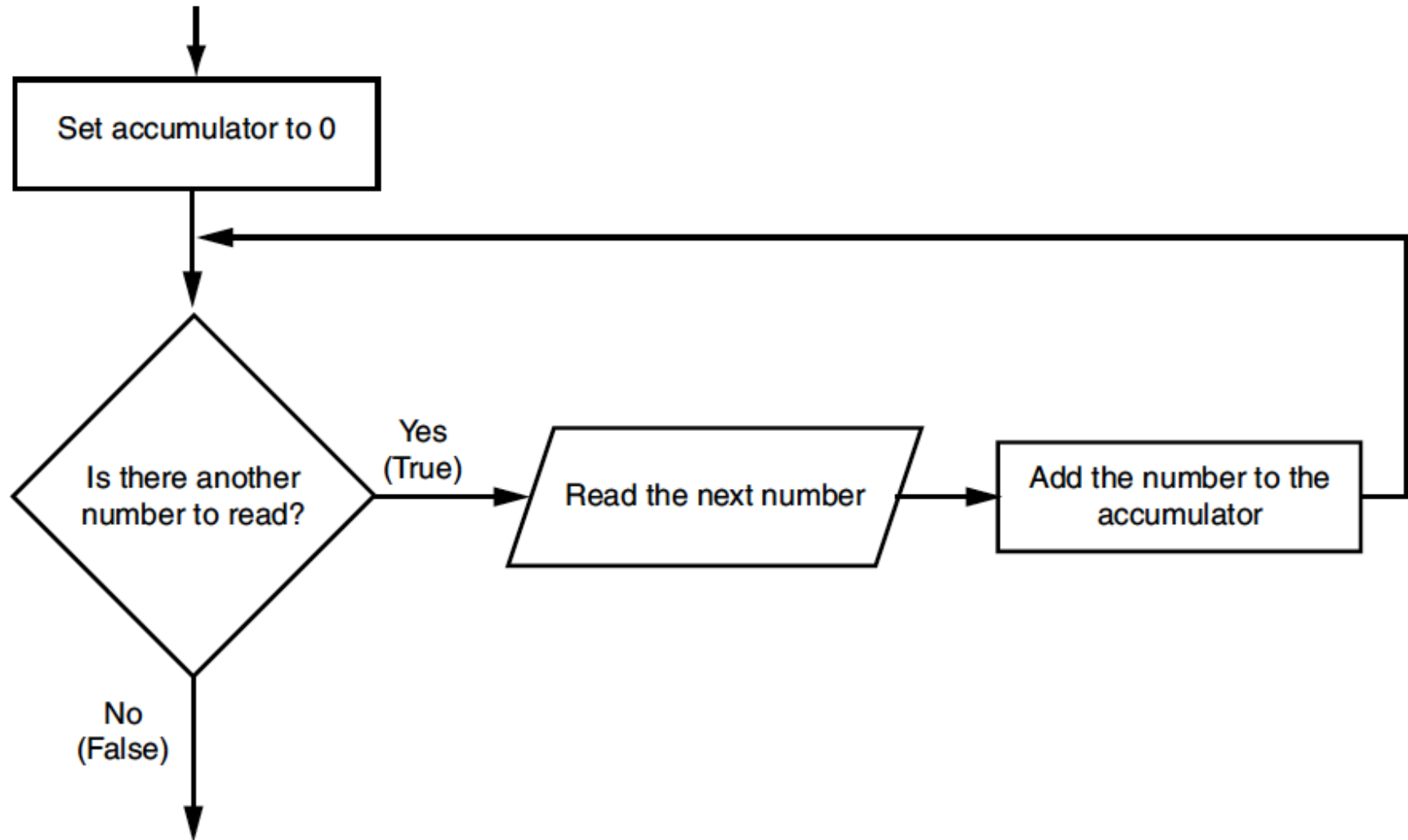
```
>>> for num in range(5, 0, -1):  
        print(num)
```

```
5  
4  
3  
2  
1
```

Calculating a Running Total

- Programs often need to calculate a total of a series of numbers
 - Typically include two elements:
 - A loop that reads each number in series
 - An *accumulator* variable
 - Known as program that keeps a running total: accumulates total and reads in series
 - At end of loop, accumulator will reference the total

Calculating a Running Total (cont'd.)



Program 4-12 (sum_numbers.py)

31

```
1  # This program calculates the sum of a series
2  # of numbers entered by the user.
3
4  MAX = 5 # The maximum number
5
6  # Initialize an accumulator variable.
7  total = 0.0
8
9  # Explain what we are doing.
10 print('This program calculates the sum of')
11 print(MAX, 'numbers you will enter.')
12
13 # Get the numbers and accumulate them.
14 for counter in range(MAX):
15     number = int(input('Enter a number: '))
16     total = total + number
17
18 # Display the total of the numbers.
19 print('The total is', total)
```

Program Output (with input shown in bold)

This program calculates the sum of
5 numbers you will enter.

Enter a number: **1**

Enter a number: **2**

Enter a number: **3**

Enter a number: **4**

Enter a number: **5**

The total is 15.0

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.)

Statement	What It Does	Value of x after the Statement
<code>x = x + 4</code>	Add 4 to x	10
<code>x = x - 3</code>	Subtracts 3 from x	3
<code>x = x * 10</code>	Multiplies x by 10	60
<code>x = x / 2</code>	Divides x by 2	3
<code>x = x % 4</code>	Assigns the remainder of x / 4 to x	2

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

Sentinels

- Sentinel: special value that marks the end of a sequence of items
 - When program reaches a sentinel, it knows that the end of the sequence of items was reached, and the loop terminates
 - Must be distinctive enough so as not to be mistaken for a regular value in the sequence
 - Example: when reading an input file, empty line can be used as a sentinel

Program 4-13 (property_tax.py)

```
1  # This program displays property taxes.
2
3  TAX_FACTOR = 0.0065 # Represents the tax factor.
4
5  # Get the first lot number.
6  print('Enter the property lot number')
7  print('or enter 0 to end.')
8  lot = int(input('Lot number: '))
9
10 # Continue processing as long as the user
11 # does not enter lot number 0.
12 while lot != 0:
13     # Get the property value.
14     value = float(input('Enter the property value: '))
15
16     # Calculate the property's tax.
17     tax = value * TAX_FACTOR
18
19     # Display the tax.
20     print('Property tax: $', format(tax, ',.2f'), sep='')
21
22     # Get the next lot number.
23     print('Enter the next lot number or')
24     print('enter 0 to end.')
25     lot = int(input('Lot number: '))
```

Program Output (with input shown in bold)

Enter the property lot number
or enter 0 to end.

Lot number: **100**

Enter the property value: **100000.00**

Property tax: \$650.00.

Enter the next lot number or
enter 0 to end.

Lot number: **200**

Enter the property value: **5000.00**

Property tax: \$32.50.

Enter the next lot number or
enter 0 to end.

Lot number: **0**

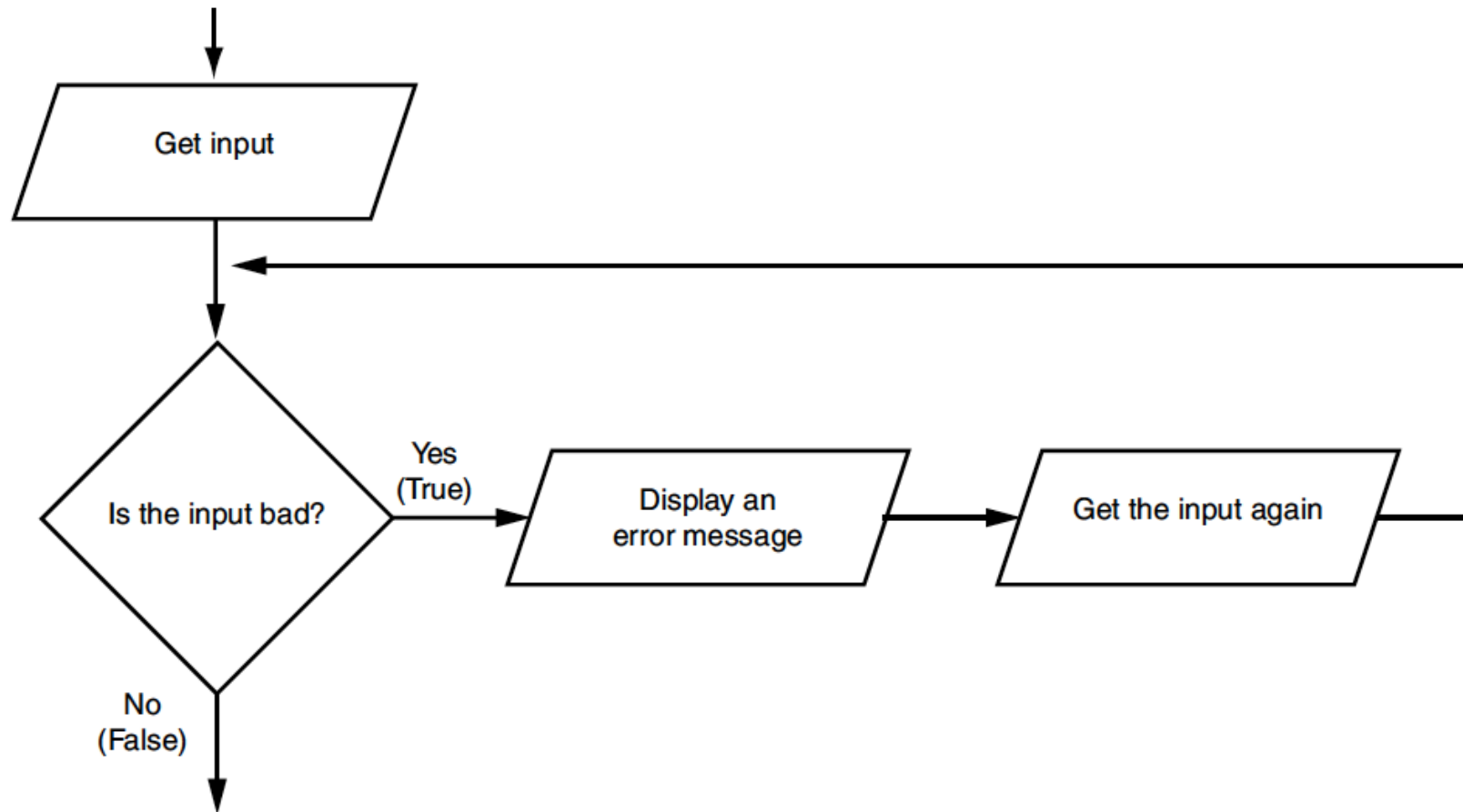
Input Validation Loops

- Computer cannot tell the difference between good data and bad data
 - If user provides bad input, program will produce bad output
 - GIGO: garbage in, garbage out
 - It is important to design program such that bad input is never accepted

Input Validation Loops (cont'd.)

- Input validation: inspecting input before it is processed by the program
 - If input is invalid, prompt user to enter correct data
 - Commonly accomplished using a `while` loop which repeats as long as the input is bad
 - If input is bad, display error message and receive another set of data
 - If input is good, continue to process the input

Input Validation Loops (cont'd.)




```
1  # This program calculates retail prices.
2
3  MARK_UP = 2.5 # The markup percentage
4  another = 'y' # Variable to control the loop.
5
6  # Process one or more items.
7  while another == 'y' or another == 'Y':
8      # Get the item's wholesale cost.
9      wholesale = float(input("Enter the item's " +
10                             "wholesale cost: "))
11
12     # Validate the wholesale cost.
13     while wholesale < 0:
14         print('ERROR: the cost cannot be negative.')
15         wholesale = float(input('Enter the correct ' +
16                                 'wholesale cost: '))
17
18     # Calculate the retail price.
19     retail = wholesale * MARK_UP
20
21     # Display the retail price.
22     print('Retail price: $', format(retail, ',.2f'), sep='')
23
24
25     # Do this again?
26     another = input('Do you have another item? ' +
27                     '(Enter y for yes): ')
```

Program Output (with input shown in bold)

Enter the item's wholesale cost: **-.50**

ERROR: the cost cannot be negative.

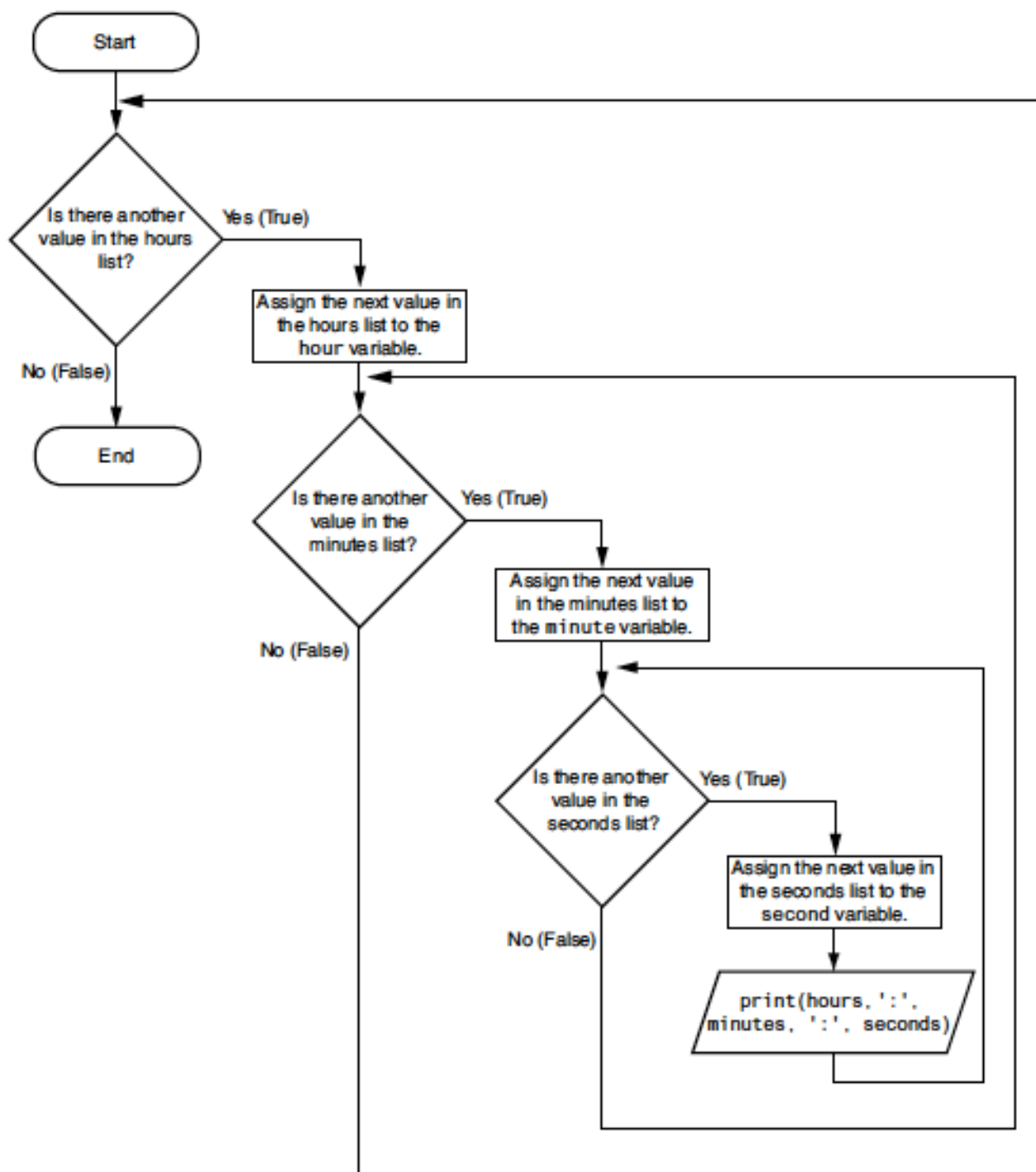
Enter the correct wholesale cost: **0.50**

Retail price: \$1.25.

Do you have another item? (Enter y for yes): **n**

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 (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:
$$\text{number_iterations_inner} \times \text{number_iterations_outer}$$

```
1  # This program averages test scores. It asks the user for the
2  # number of students and the number of test scores per student.
3
4  # Get the number of students.
5  num_students = int(input('How many students do you have? '))
6
7  # Get the number of test scores per student.
8  num_test_scores = int(input('How many test scores per student? '))
9
10 # Determine each student's average test score.
11 for student in range(num_students):
12     # Initialize an accumulator for test scores.
13     total = 0.0
14     # Get a student's test scores.
15     print('Student number', student + 1)
16     print('-----')
17     for test_num in range(num_test_scores):
18         print('Test number', test_num + 1, end='')
19         score = float(input(': '))
20         # Add the score to the accumulator.
21         total += score
22
23     # Calculate the average test score for this student.
24     average = total / num_test_scores
25
26     # Display the average.
27     print('The average for student number', student + 1,
28           'is:', average)
29     print()
```

Program Output (with input shown in bold)

How many students do you have? **3**

How many test scores per student? **3**

Student number 1

Test number 1: **100**

Test number 2: **95**

Test number 3: **90**

The average for student number 1 is: 95.0

Student number 2

Test number 1: **80**

Test number 2: **81**

Test number 3: **82**

The average for student number 2 is: 81.0

Student number 3

Test number 1: **75**

Test number 2: **85**

Test number 3: **80**

The average for student number 3 is: 80.0

Program 4-18 (rectangular_pattern.py)

48

```
1 # This program displays a rectangular pattern
2 # of asterisks.
3 rows = int(input('How many rows? '))
4 cols = int(input('How many columns? '))
5
6 for r in range(rows):
7     for c in range(cols):
8         print('*', end='')
9     print()
```

Program Output (with input shown in bold)

How many rows? **5**

How many columns? **10**

Program 4-19 (triangle_pattern.py)

49

```
1  # This program displays a triangle pattern.  
2  BASE_SIZE = 8  
3  
4  for r in range(BASE_SIZE):  
5      for c in range(r + 1):  
6          print('*', end='')  
7      print()
```

Program Output

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

Program 4-20 (stair_step_pattern.py)

```
1  # This program displays a stair-step pattern.  
2  NUM_STEPS = 6  
3  
4  for r in range(NUM_STEPS):  
5      for c in range(r):  
6          print(' ', end='')  
7          print('#')
```

Program Output

```
#  
 #  
  #  
   #  
    #  
     #
```

Summary

- This chapter covered:
 - Repetition structures, including:
 - Condition-controlled loops
 - Count-controlled loops
 - Nested loops
 - Infinite loops and how they can be avoided
 - `range` function as used in `for` loops
 - Calculating a running total and augmented assignment operators
 - Use of sentinels to terminate loops