Chapter 4: Loops

Chapter Goals

- To implement while and for loops
- To hand-trace the execution of a program
- To become familiar with common loop algorithms
- To understand nested loops
- To implement programs that read and process data sets
- To use a computer for simulations

In this chapter, you will learn about loop statements in Python, as well as techniques for writing programs that simulate activities in the real world.

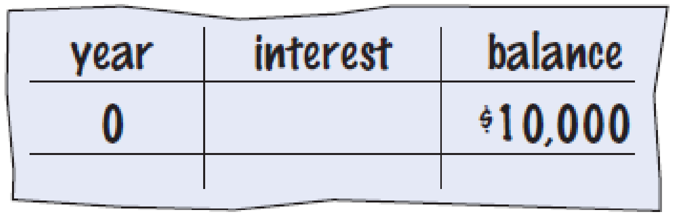
Contents

The **while** Loop

The **while** Loop

- Examples of loop applications
  - Calculating compound interest
  - Simulations, event driven programs
  - Drawing tiles…
- Compound interest algorithm (Chapter 1)

Start with a year value of 0, a column for the interest, and a balance of $10,000.



year	interest	balance
0		$10,000

Repeat the following steps while the balance is less than $20,000.

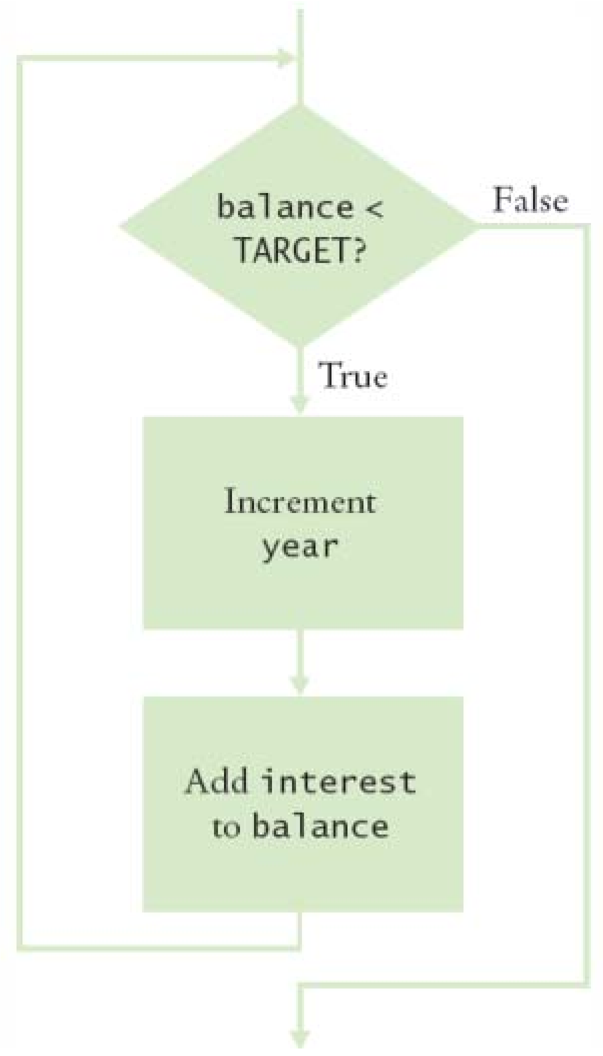
Steps

```
Repeat the following steps while the balance is less than $20,000.
    Add 1 to the year value.
    Compute the interest as balance x 0.05 (i.e. 5 percent interest).
    Add the interest to the balance.
Report the final year value as the answer.
```

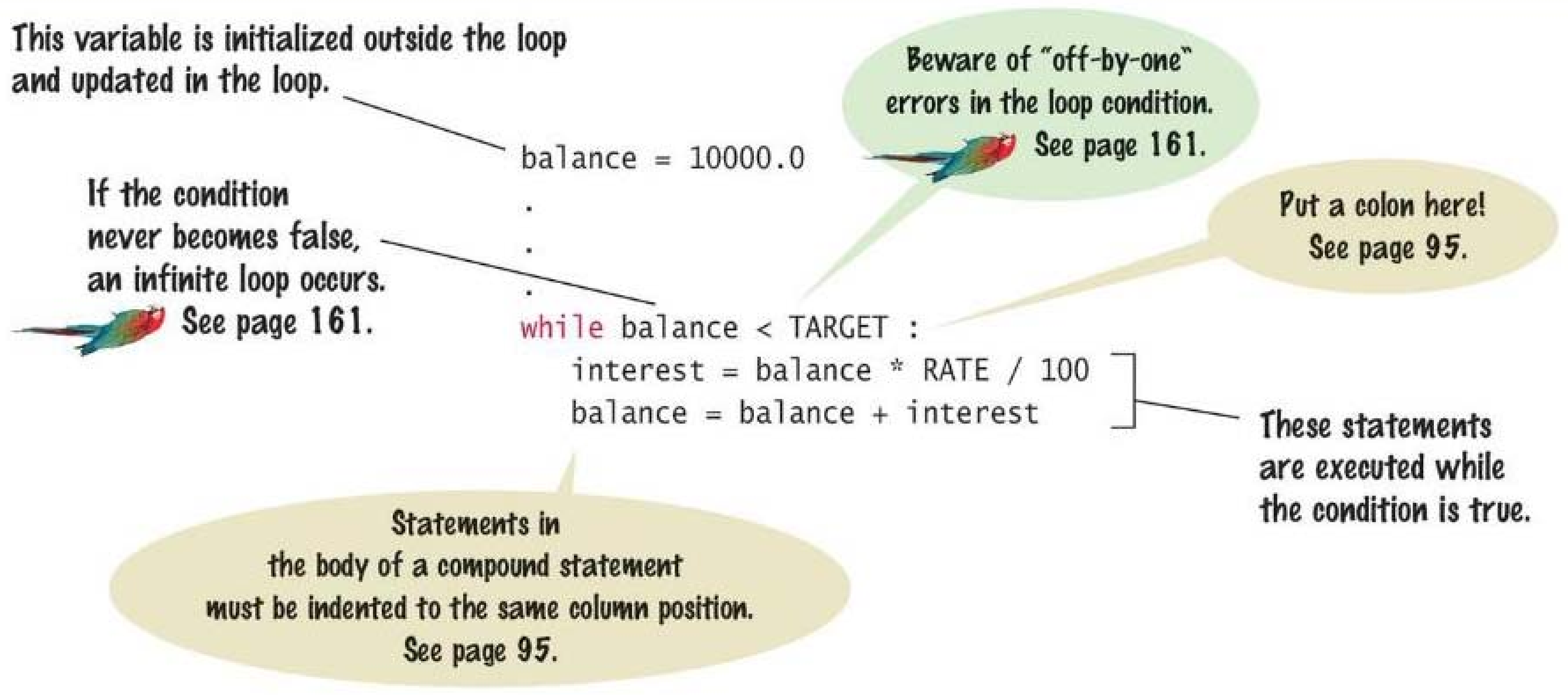
Planning the **while** Loop

```
balance = 10.0
target = 100.0
year = 0
rate = 0.025
while balance < TARGET :
   year = year + 1
   interest = balance * RATE/100
   balance = balance + interest
```

A loop executes instructions repeatedly while a condition is True.



Syntax: `while` Statement



```
balance = 10000.0
.
.
.
while balance < TARGET :
    interest = balance * RATE / 100
    balance = balance + interest
```

This variable is initialized outside the loop and updated in the loop.
Beware of "off-by-one" errors in the loop condition. See page 161.
If the condition never becomes false, an infinite loop occurs. See page 161.
Put a colon here! See page 95.
These statements are executed while the condition is true.
Statements in the body of a compound statement must be indented to the same column position. See page 95.

Count-Controlled Loops

- A **while** loop that is controlled by a counter

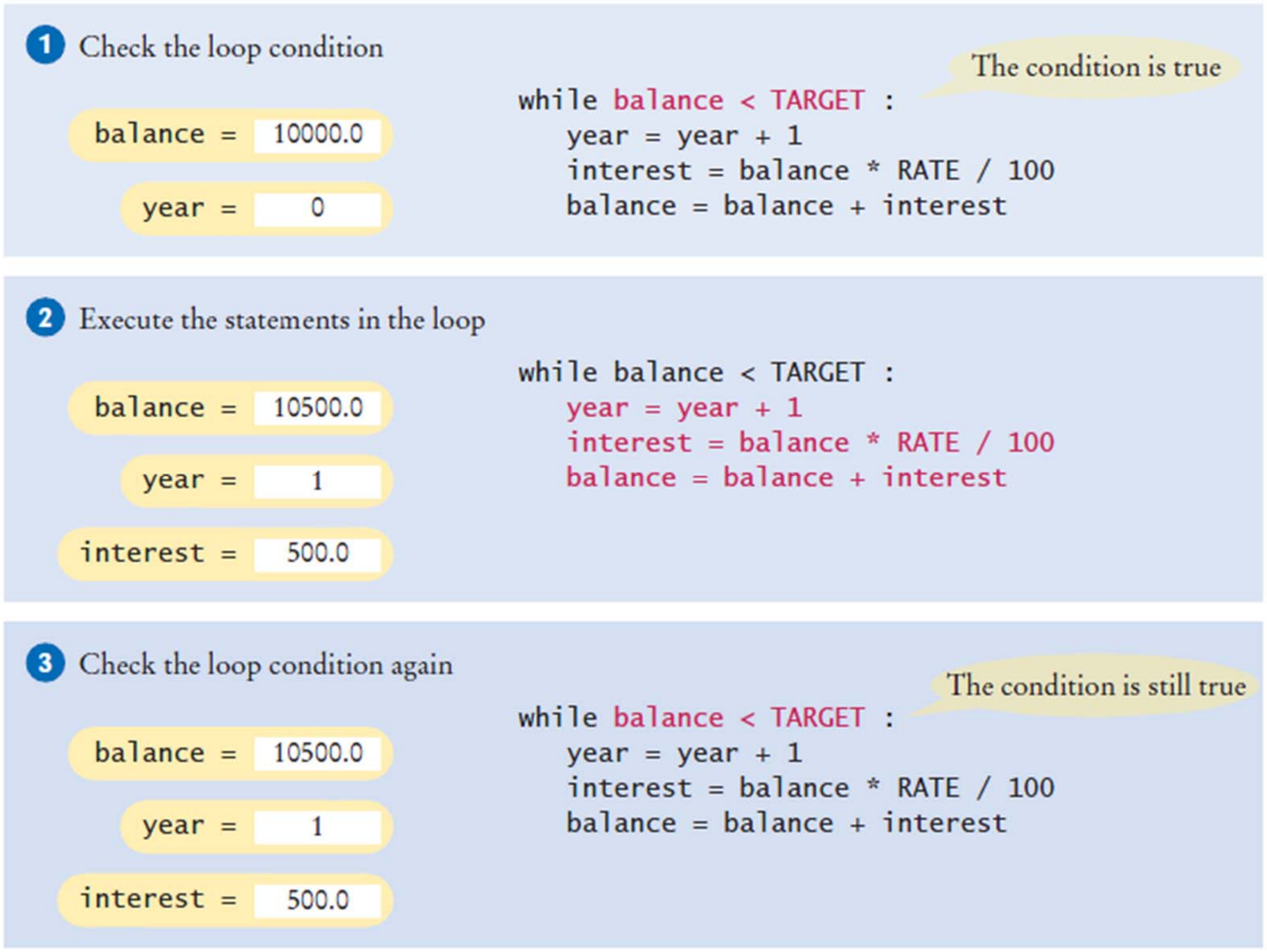
```
counter = 1                                   # Initialize
the counter

while counter <= 10 :                 # Check the counter
  print(counter)
  counter = counter + 1           # Update the loop
  variable
```

Event-Controlled Loops

- A **while** loop that is controlled by a counter

```
balance = INITIAL_BALANCE        # Initialize the loop
variable

while balance <= TARGET:           # Check the loop
variable
  year – year + 1
  balance = balance * 2                        # Update
  the loop variable
```

Execution of the Loop



1 Check the loop condition

```
while balance < TARGET :
    year = year + 1
    interest = balance * RATE / 100
    balance = balance + interest
```

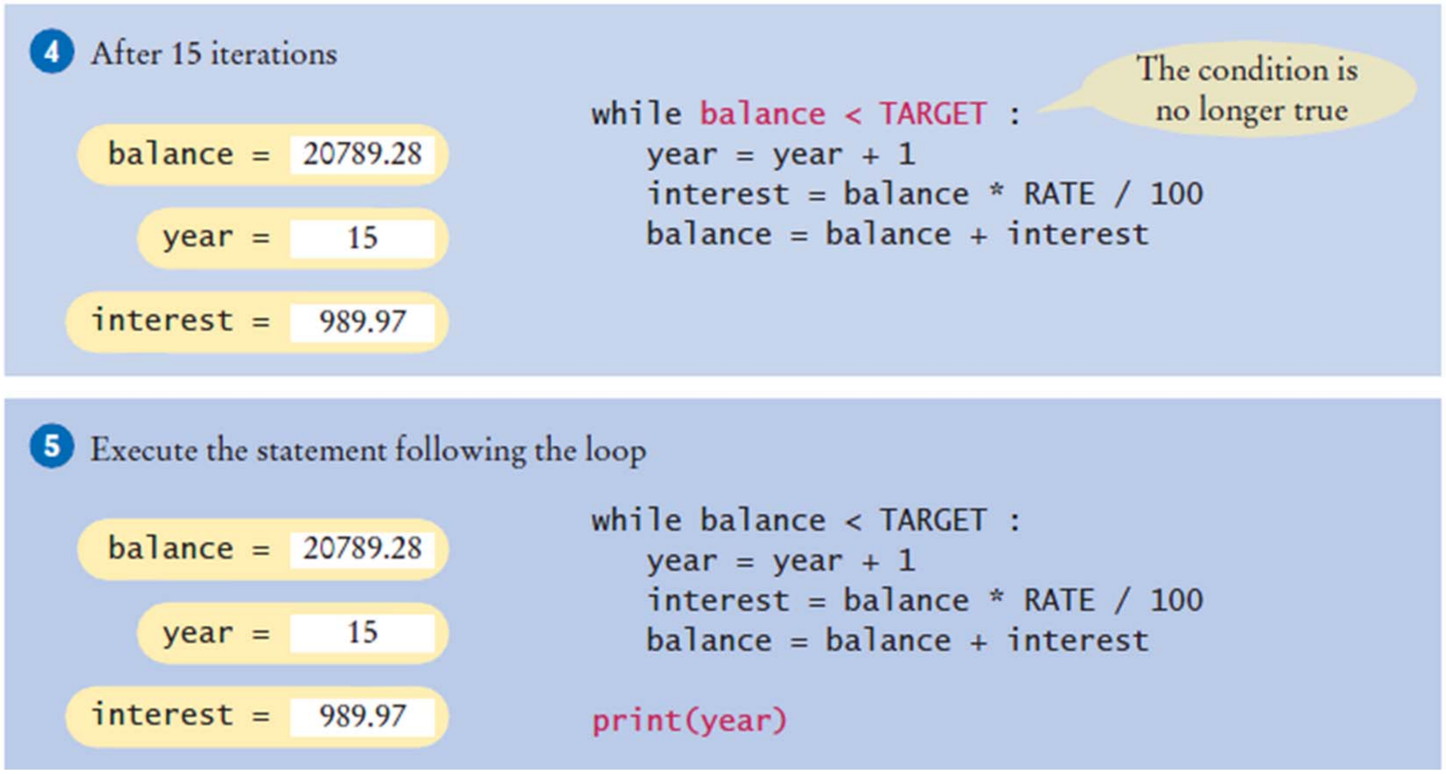
2 Execute the statements in the loop

```
while balance < TARGET :
    year = year + 1
    interest = balance * RATE / 100
    balance = balance + interest
```

3 Check the loop condition again

```
while balance < TARGET :
    year = year + 1
    interest = balance * RATE / 100
    balance = balance + interest
```

Execution of the Loop (2)



4 After 15 iterations

```
balance  = 20789.28
year     = 15
interest = 989.97
```

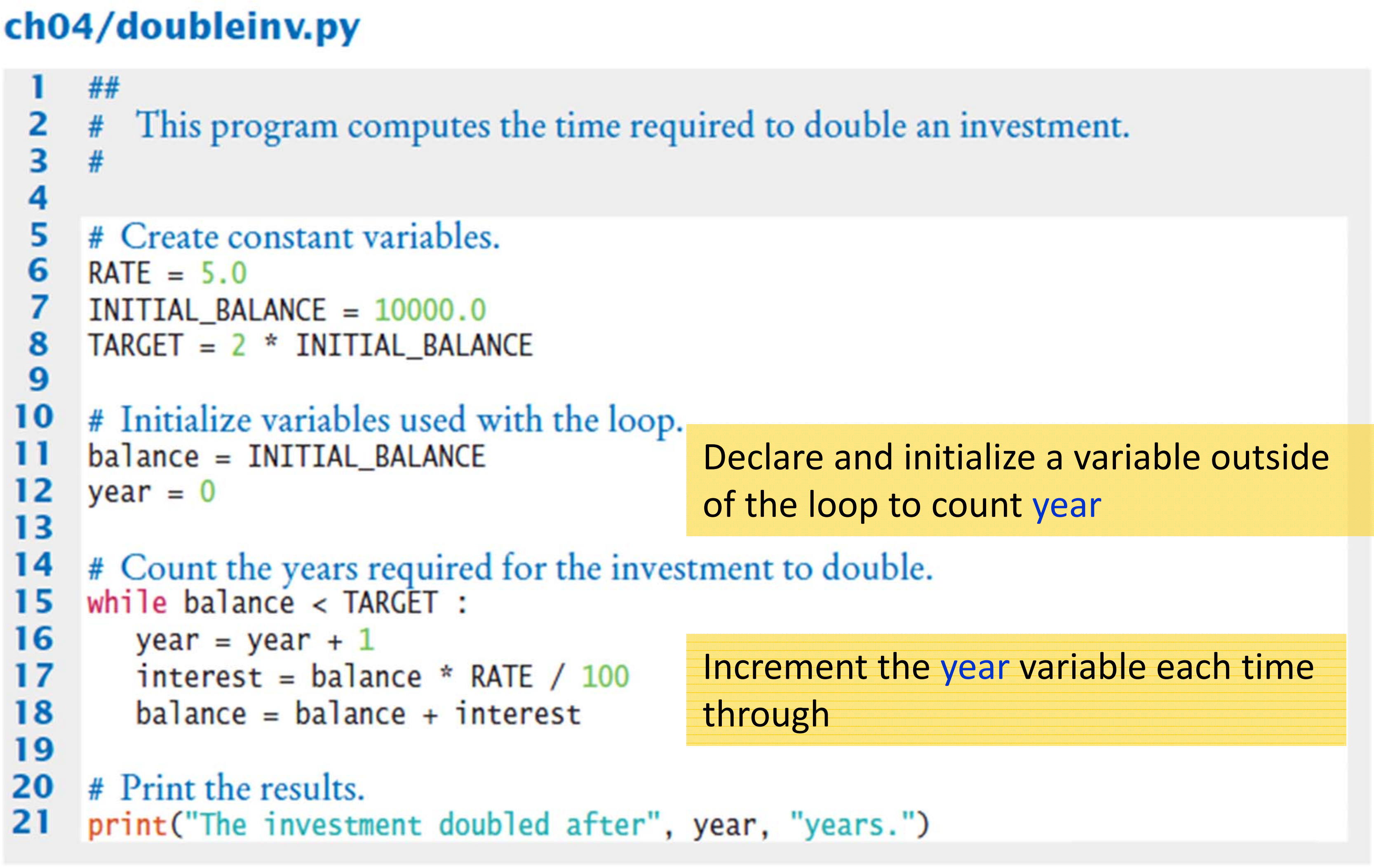
```
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest
```

5 Execute the statement following the loop

```
balance  = 20789.28
year     = 15
interest = 989.97
```

```
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest

print(year)
```

Doubleinv.py



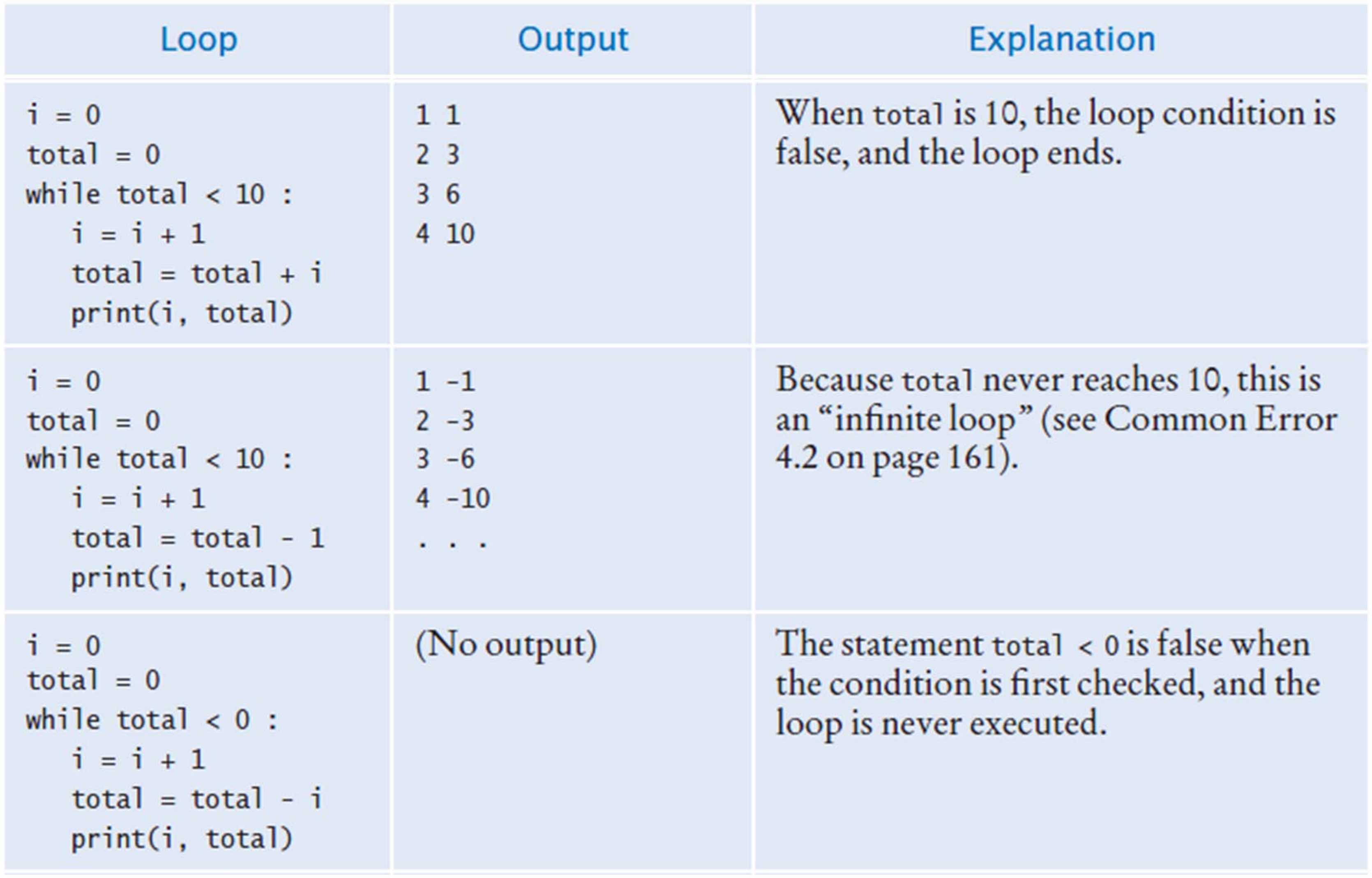
ch04/doubleinv.py

```python
##
#  This program computes the time required to double an investment.
#

# Create constant variables.
RATE = 5.0
INITIAL_BALANCE = 10000.0
TARGET = 2 * INITIAL_BALANCE

# Initialize variables used with the loop.
balance = INITIAL_BALANCE
year = 0

# Count the years required for the investment to double.
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest

# Print the results.
print("The investment doubled after", year, "years.")
```

Declare and initialize a variable outside of the loop to count year

Increment the year variable each time through

`while` Loop Examples



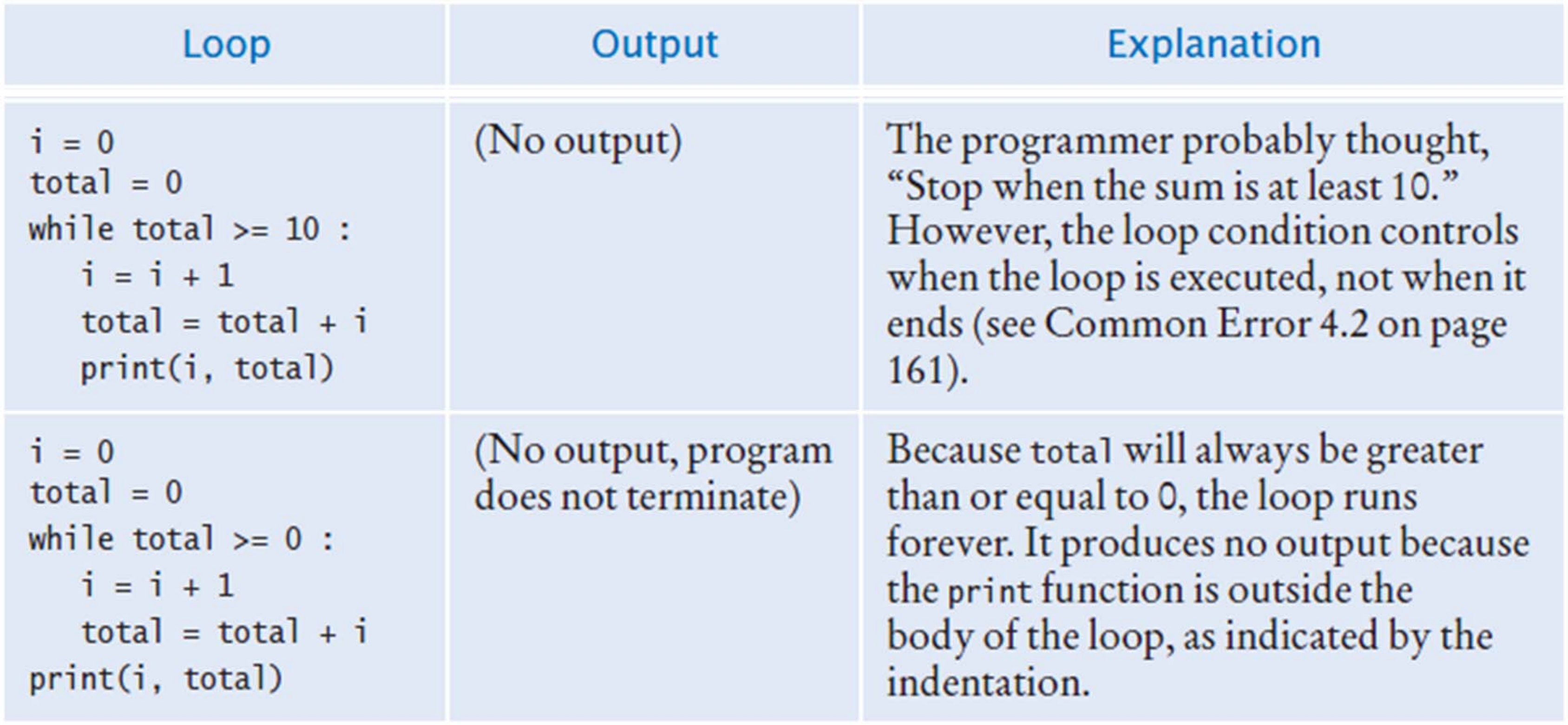
Loop	Output	Explanation

`i = 0`
`total = 0`
`while total < 10 :`
`    i = i + 1`
`    total = total + i`
`    print(i, total)`
	`1 1`
	`2 3`
	`3 6`
	`4 10`
		When `total` is 10, the loop condition is false, and the loop ends.

`i = 0`
`total = 0`
`while total < 10 :`
`    i = i + 1`
`    total = total - 1`
`    print(i, total)`
	`1 -1`
	`2 -3`
	`3 -6`
	`4 -10`
	`. . .`
		Because `total` never reaches 10, this is an "infinite loop" (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total < 0 :`
`    i = i + 1`
`    total = total - i`
`    print(i, total)`
	(No output)	The statement `total < 0` is false when the condition is first checked, and the loop is never executed.

`while` Loop Examples (2)



Loop	Output	Explanation

`i = 0`
`total = 0`
`while total >= 10 :`
`    i = i + 1`
`    total = total + i`
`    print(i, total)`
	(No output)	The programmer probably thought, "Stop when the sum is at least 10." However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total >= 0 :`
`    i = i + 1`
`    total = total + i`
`print(i, total)`
	(No output, program does not terminate)	Because `total` will always be greater than or equal to `0`, the loop runs forever. It produces no output because the `print` function is outside the body of the loop, as indicated by the indentation.

Common Error: Incorrect Test Condition

- The loop body will only execute if the test condition is True.
- If bal is initialized as less than the TARGET and should grow until it reaches TARGET
  - Which version will execute the loop body?

```
while bal >= TARGET :
    year = year + 1
    interest = bal * RATE
    bal = bal + interest
```

```
while bal < TARGET :
    year = year + 1
    interest = bal * RATE
    bal = bal + interest
```

Common Error: **Infinite Loops**

- The loop body will execute until the test condition becomes False.
- What if you forget to update the test variable?
  - bal is the test variable (TARGET doesn’t change)
  - You will loop forever! (or until you stop the program)

```
while bal < TARGET :
    year = year + 1
    interest = bal * RATE
    bal = bal + interest
```

Common Error: **Off-by-One Errors**

- A ‘counter’ variable is often used in the test condition
- Your counter can start at 0 or 1, but programmers often start a counter at 0
- If I want to paint all 5 fingers on one hand, when I am done?
  - If you start at 0, use “<“ If you start at 1, use “<=“
  - 0, 1, 2, 3, 4 1, 2, 3, 4, 5

```
finger = 0
FINGERS = 5
while finger < FINGERS :
   # paint finger
   finger = finger + 1
```

```
finger = 1
FINGERS = 5
while finger <= FINGERS :
   # paint finger
   finger = finger + 1
```

Summary of the **while** Loop

- `while` loops are very common
- Initialize variables before you test
  - The condition is tested BEFORE the loop body
    - This is called pre-test
    - The condition often uses a counter variable
  - Something inside the loop should change one of the variables used in the test
- Watch out for infinite loops!

Sentinel Values

Processing Sentinel Values

- Sentinel values are often used:
  - When you don’t know how many items are in a list, use a ‘special’ character or value to signal the “last” item
  - For numeric input of positive numbers, it is common to use the value -1

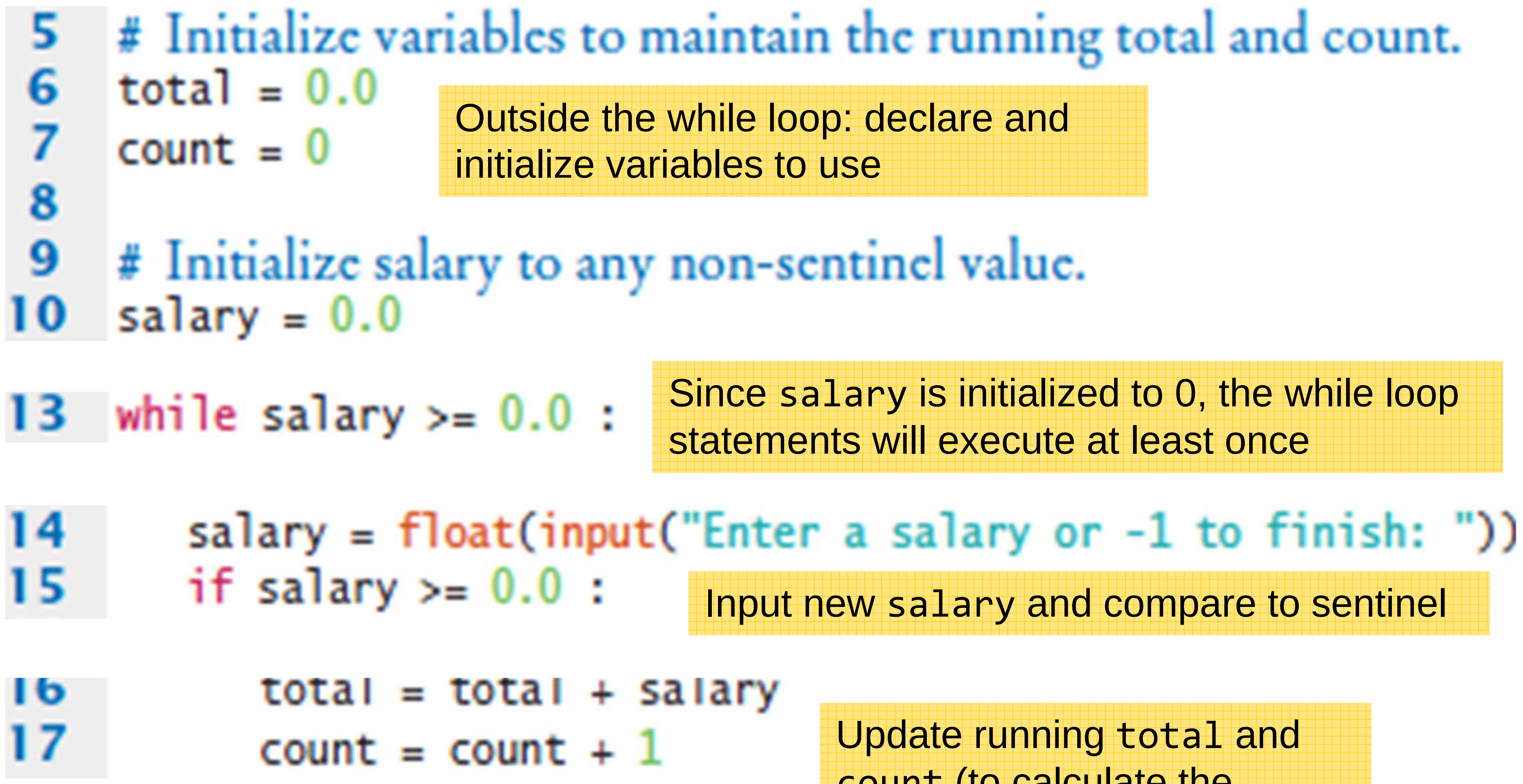
A sentinel value denotes the end of a data set, but it is not part of the data.

```
salary = 0.0
while salary >= 0 :
   salary = float(input())
      if salary >= 0.0 :
         total = total + salary
         count = count + 1
```

Averaging a Set of Values

- Declare and initialize a ‘total’ variable to 0
- Declare and initialize a ‘count’ variable to 0
- Declare and initialize a ‘salary’ variable to 0
- Prompt user with instructions
- Loop until sentinel value is entered
  - Save entered value to input variable (‘salary’)
  - If salary is not -1 or less (sentinel value)
    - Add salary variable to total variable
    - Add 1 to count variable
- Make sure you have at least one entry before you divide!
  - Divide total by count and output.
  - Done!

Sentinel.py (1)



```
# Initialize variables to maintain the running total and count.
total = 0.0
count = 0

# Initialize salary to any non-sentinel value.
salary = 0.0

while salary >= 0.0 :

    salary = float(input("Enter a salary or -1 to finish: "))
    if salary >= 0.0 :

        total = total + salary
        count = count + 1
```

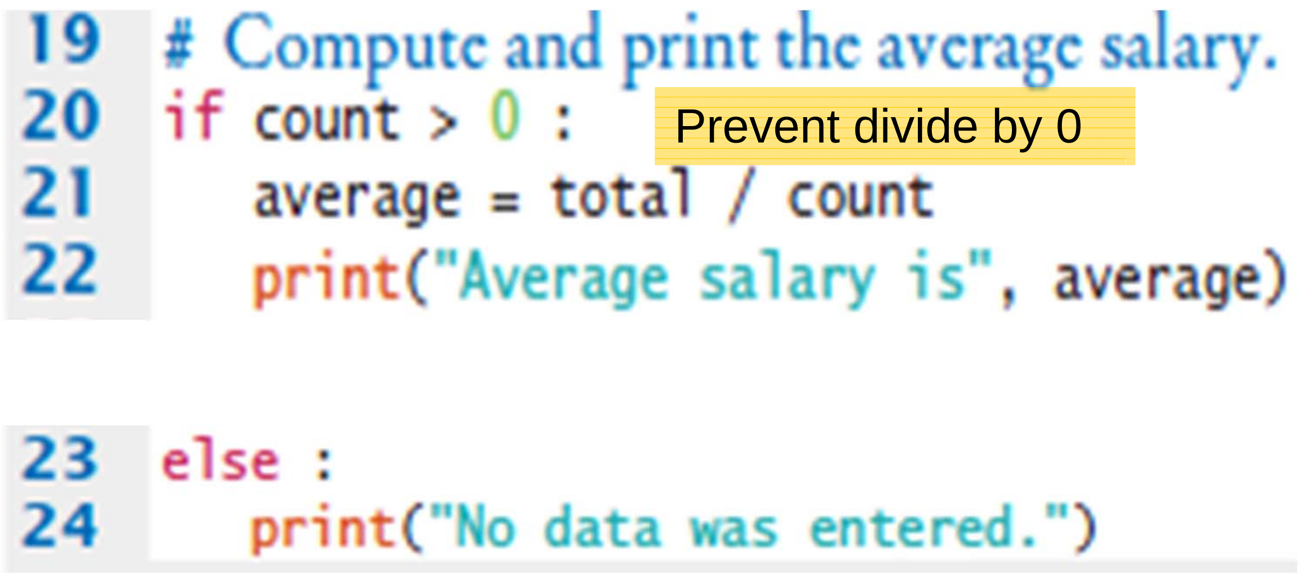
Outside the while loop: declare and initialize variables to use

Since `salary` is initialized to 0, the while loop statements will execute at least once

Input new `salary` and compare to sentinel

Update running `total` and `count` (to calculate the average later)

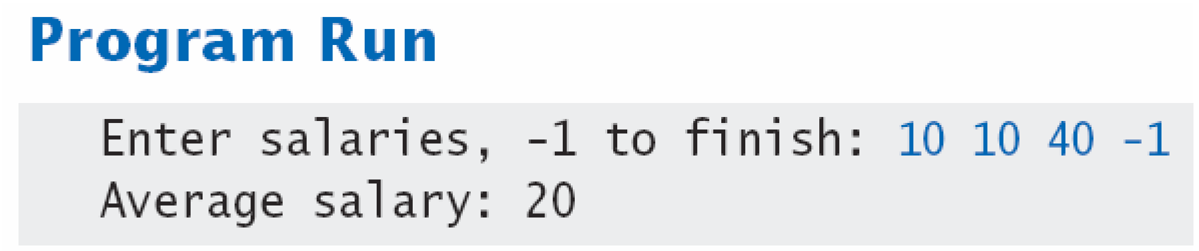
Sentinel.py (2)



```
# Compute and print the average salary.
if count > 0 :
    average = total / count
    print("Average salary is", average)
else :
    print("No data was entered.")
```

Prevent divide by 0

Calculate and output the average salary using the `total` and `count` variables



Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Priming Read

- Some programmers don’t like the “trick” of initializing the input variable with a value other than a sentinel.

```
# Set salary to a value to ensure that the loop
# executes at least once.
salary = 0.0
while salary >= 0 :
```

- An alternative is to change the variable with a read before the loop.

```
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0 :
```

Modification Read

- The input operation at the bottom of the loop is used to obtain the next input.

```
# Priming read
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0.0 :
   total = total + salary
   count = count + 1
   # Modification read
   salary = float(input("Enter a salary or -1 to finish: "))
```

Boolean Variables and Sentinels

- A boolean variable can be used to control a loop
  - Sometimes called a ‘flag’ variable

```
done = False
while not done :
    value = float(input("Enter a salary or -1 to
   finish: "))
    if value < 0.0:
        done = True
    else :
        # Process value
```

Initialize done so that the loop will execute

Set done ‘flag’ to True if sentinel value is found

Common Loop Algorithms

EXAMPLES AND SELF-STUDY

Common Loop Algorithms

1. Sum and Average Value
2. Counting Matches
3. Prompting until a Match Is Found
4. Maximum and Minimum
5. Comparing Adjacent Values

Average Example

Average of Values

- First total the values
- Initialize count to 0
  - Increment per input
- Check for count 0
  - Before divide!

```
total = 0.0
count = 0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = float(inputStr)
   total = total + value
   count = count + 1
   inputStr = input("Enter value: ")

if count > 0 :
   average = total / count
else :
   average = 0.0
```

Sum Example

- Sum of Values
  - Initialize total to 0
  - Use while loop with sentinel

```
total = 0.0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = float(inputStr)
   total = total + value
   inputStr = input("Enter value: ")
```

Counting Matches (e.g., Negative Numbers)

- Counting Matches
  - Initialize negatives to 0
  - Use a while loop
  - Add to negatives per match

```
negatives = 0
inputStr = input("Enter value: ")
while inputStr != "“ :
   value = int(inputStr)
   if value < 0 :
      negatives = negatives + 1
   inputStr = input("Enter value: ")

print("There were", negatives,
"negative values.")
```

Prompt Until a Match is Found

- Initialize boolean flag to False
- Test sentinel in while loop
  - Get input, and compare to range
    - If input is in range, change flag to True
    - Loop will stop executing

```
valid = False
while not valid :
   value = int(input("Please enter a positive value < 100: "))
   if value > 0 and value < 100 :
      valid = True
   else :
      print("Invalid input.")
```

This is an excellent way to validate use provided inputs

Maximum

- Get first input value
  - By definition, this is the largest that you have seen so far
- Loop while you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update largest if necessary

```
largest = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "“ :
   value = int(inputStr)
   if value > largest :
      largest = value
   inputStr = input("Enter a value: ")
```

Minimum

- Get first input value
  - This is the smallest that you have seen so far!
- Loop while you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update smallest if necessary

```
smallest = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "“ :
   value = int(inputStr)
   if value < smallest :
      smallest = value
   inputStr = input("Enter a value: ")
```

Comparing Adjacent Values

- Get first input value
- Use `while` to determine if there are more to check
  - Copy input to previous variable
  - Get next value into input variable
  - Compare input to previous, and output if same

```
value = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "“ :
   previous = value
   value = int(inputStr)
   if value == previous :
      print("Duplicate input")
   inputStr = input("Enter a value: ")
```

The **for** Loop

The `for` Loop

- Uses of a **`for`** loop:
  - The **`for`** loop can be used to iterate over the contents of any **container**.
  - A **container** is is an object (Like a **string**) that contains or stores a collection of elements
  - A **string** is a container that stores the collection of characters in the string

An Example of a `for` Loop

- Note an important difference between the while loop and the for loop.
- In the while loop, the *index variable* i is assigned 0, 1, and so on.
- In the for loop, the *element variable* is assigned stateName[0], stateName[1], and so on.

```
stateName = "Virginia"
i = 0
while i < len(stateName) :
   letter = stateName[i]
   print(letter)
   i = i + 1
```

while version

```
stateName = "Virginia"
for letter in stateName :
   print(letter)
```

for version

The for Loop (2)

- Uses of a for loop:
  - A for loop can also be used as a count-controlled loop that iterates over a range of integer values.

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

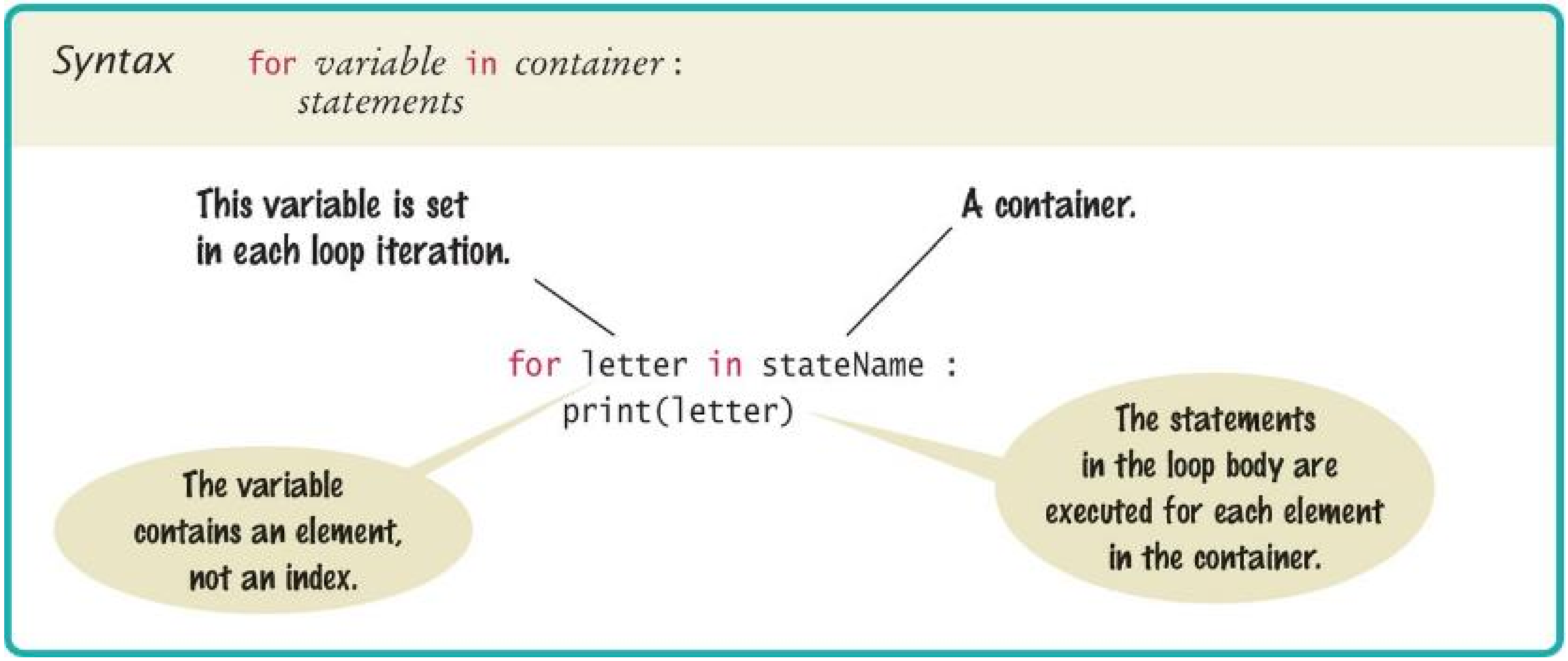
while version

```
for i in range(1, 10) :
   print(i)
```

for version

Syntax of a **for** Statement (Container)

- Using a for loop to iterate over the contents of a container, an element at a time.



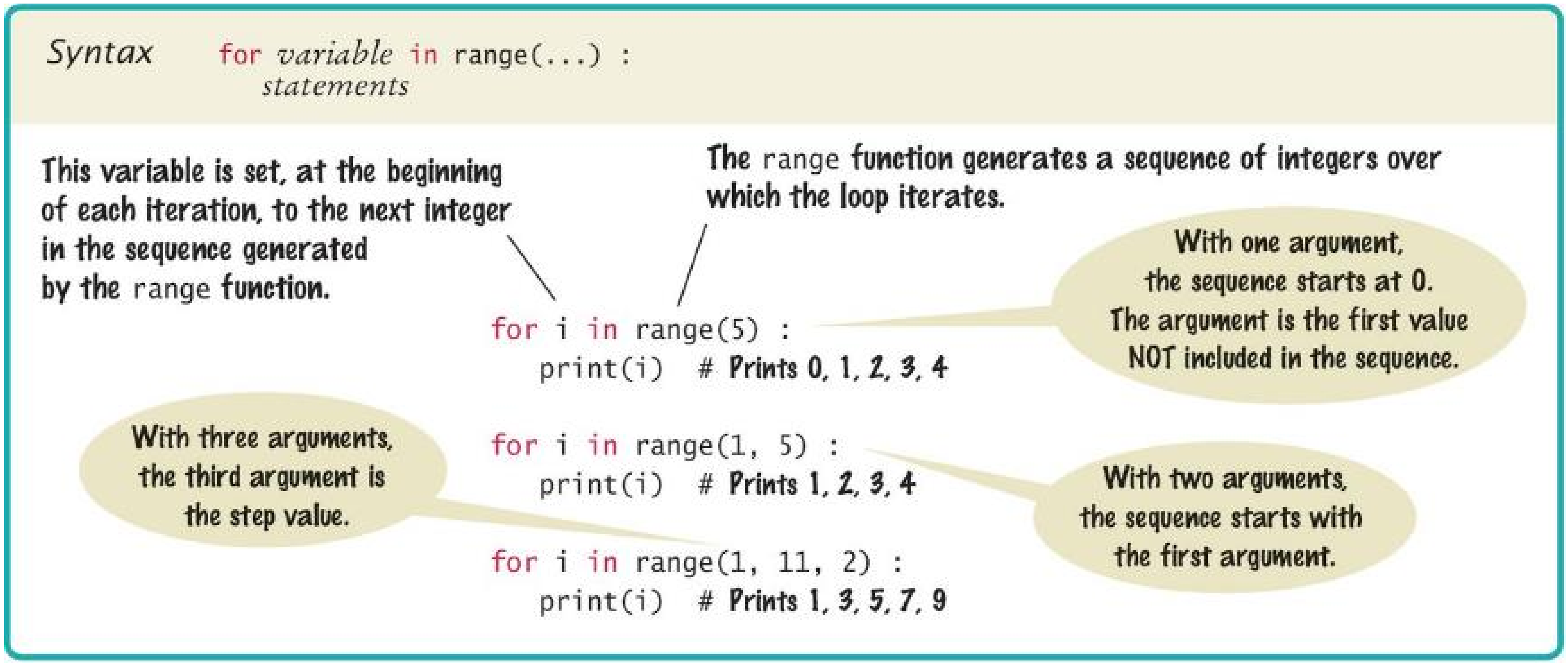
Syntax
```
for variable in container :
    statements
```

```
for letter in stateName :
    print(letter)
```

This variable is set in each loop iteration.
A container.
The variable contains an element, not an index.
The statements in the loop body are executed for each element in the container.

Syntax of a **for** Statement (Range)

- You can use a for loop as a count-controlled loop to iterate over a range of integer values
- We use the range function for generating a sequence of integers that less than the argument that can be used with the for loop



Syntax

```
for variable in range(...) :
    statements
```

```
for i in range(5) :
    print(i)   # Prints 0, 1, 2, 3, 4

for i in range(1, 5) :
    print(i)   # Prints 1, 2, 3, 4

for i in range(1, 11, 2) :
    print(i)   # Prints 1, 3, 5, 7, 9
```

This variable is set, at the beginning of each iteration, to the next integer in the sequence generated by the range function.

The range function generates a sequence of integers over which the loop iterates.

With one argument, the sequence starts at 0. The argument is the first value NOT included in the sequence.

With two arguments, the sequence starts with the first argument.

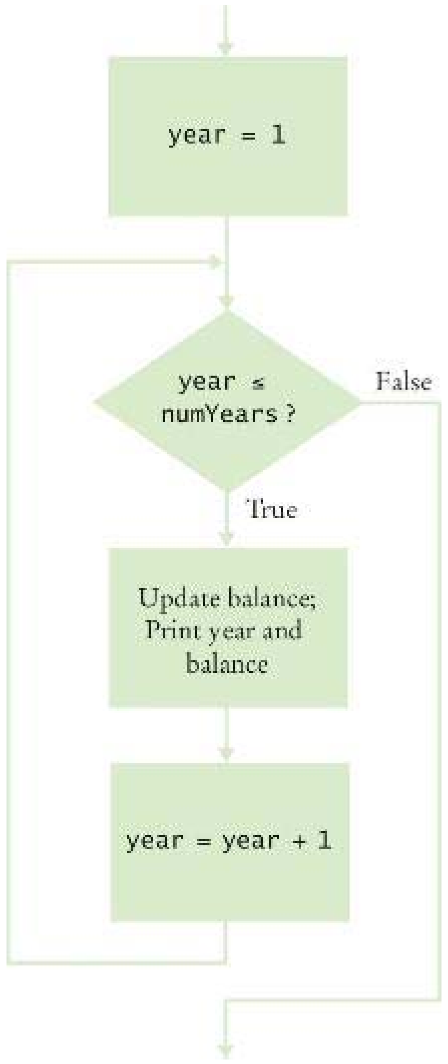
With three arguments, the third argument is the step value.

Planning a **for** Loop

- Print the balance at the end of each year for a number of years

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

```
for year in range(1, numYears + 1) :
    Update balance.
    Print year and balance.
```



Good Examples of **for** Loops

- Keep the loops simple!

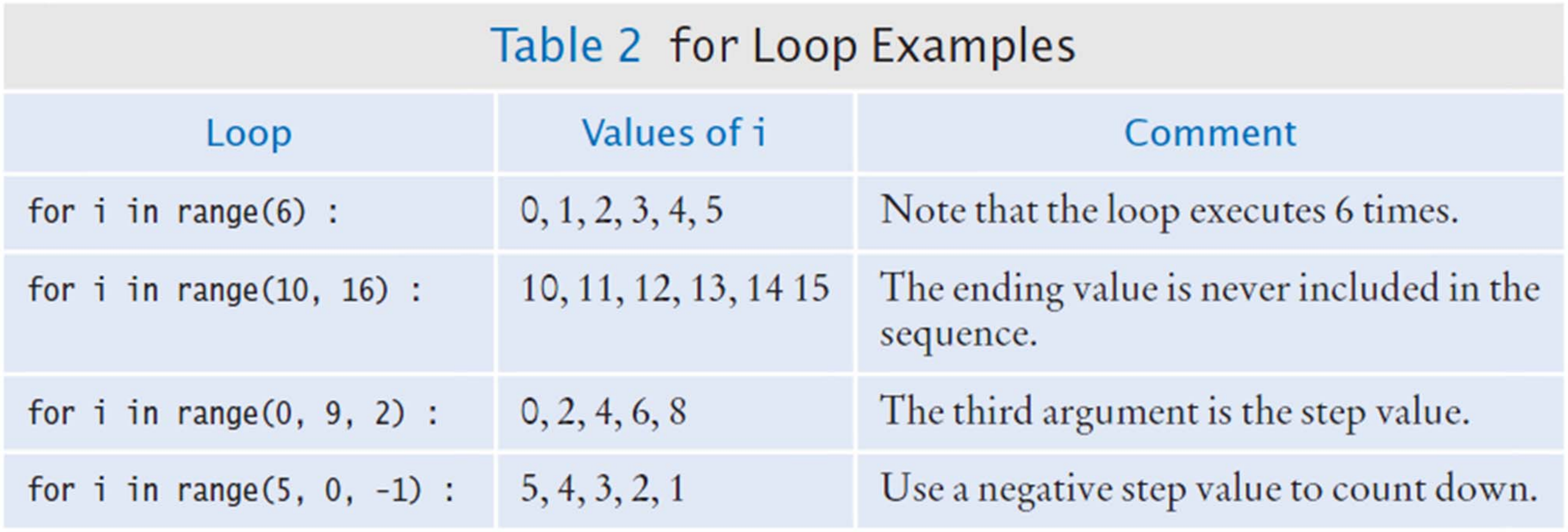
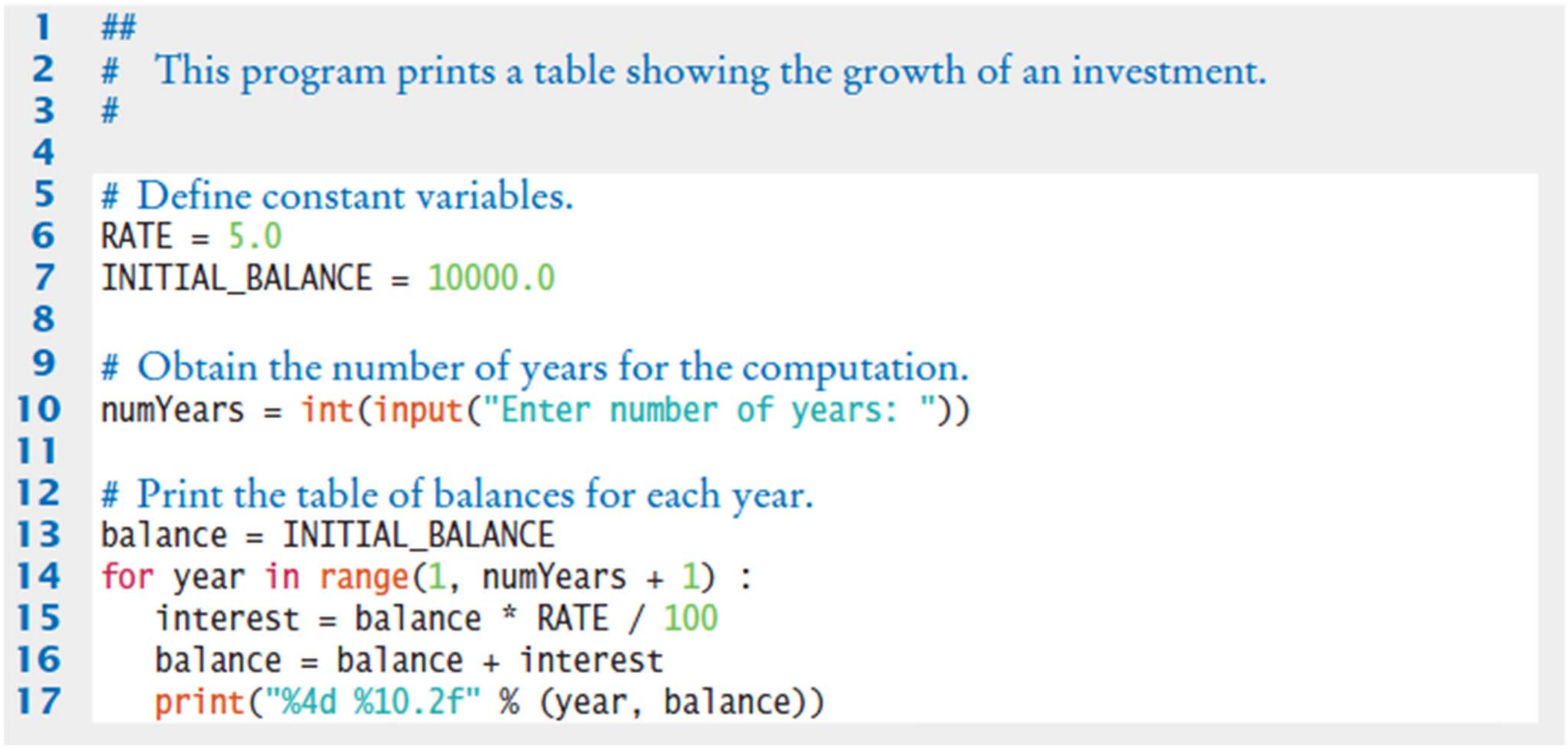


Table 2 for Loop Examples

Loop	Values of i	Comment
`for i in range(6) :`	0, 1, 2, 3, 4, 5	Note that the loop executes 6 times.
`for i in range(10, 16) :`	10, 11, 12, 13, 14 15	The ending value is never included in the sequence.
`for i in range(0, 9, 2) :`	0, 2, 4, 6, 8	The third argument is the step value.
`for i in range(5, 0, -1) :`	5, 4, 3, 2, 1	Use a negative step value to count down.

Investment Example



```
##
#  This program prints a table showing the growth of an investment.
#

# Define constant variables.
RATE = 5.0
INITIAL_BALANCE = 10000.0

# Obtain the number of years for the computation.
numYears = int(input("Enter number of years: "))

# Print the table of balances for each year.
balance = INITIAL_BALANCE
for year in range(1, numYears + 1) :
   interest = balance * RATE / 100
   balance = balance + interest
   print("%4d %10.2f" % (year, balance))
```

Programming Tip

- Finding the correct lower and upper bounds for a loop can be confusing.
  - Should you start at 0 or at 1?
  - Should you use <= b or < b as a termination condition?
- Counting is easier for loops with asymmetric bounds.
  - The following loops are executed b - a times.

```
int i = a
while i < b :
    . . .
    i = i + 1
```

```
for i in range(a, b) :
    . . .
```

Programming Tip

- The loop with symmetric bounds (“<=”, is executed b - a + 1 times.
  - That “+1” is the source of many programming errors.

```
i = a
while i <= b :
   . . .
   i = i + 1
```

```
# For  this version of the loop the
‘+1’ is very noticeable!
for year in range(1, numYears + 1) :
```

Summary of the **for** Loop

- **for** loops are very powerful
- The **for** loop can be used to iterate over the contents of any container, which is an object that contains or stores a collection of elements
  - a string is a container that stores the collection of characters in the string.
- A **for** loop can also be used as a count-controlled loop that iterates over a range of integer values.

Programming Activity

- Create a program that asks the user for a positive number and sums up all even numbers between 0 and that number.
- For instance if a user enters 5, your program should print
  The sum of even numbers between 0 and 5 is 6

Programming Activity

Create a program that finds the average of a number of student grades. First ask the instructor how many grades they plan to add and then get all the grades from the instructor and output the average.

Nested Loops

Loops Inside of Loops

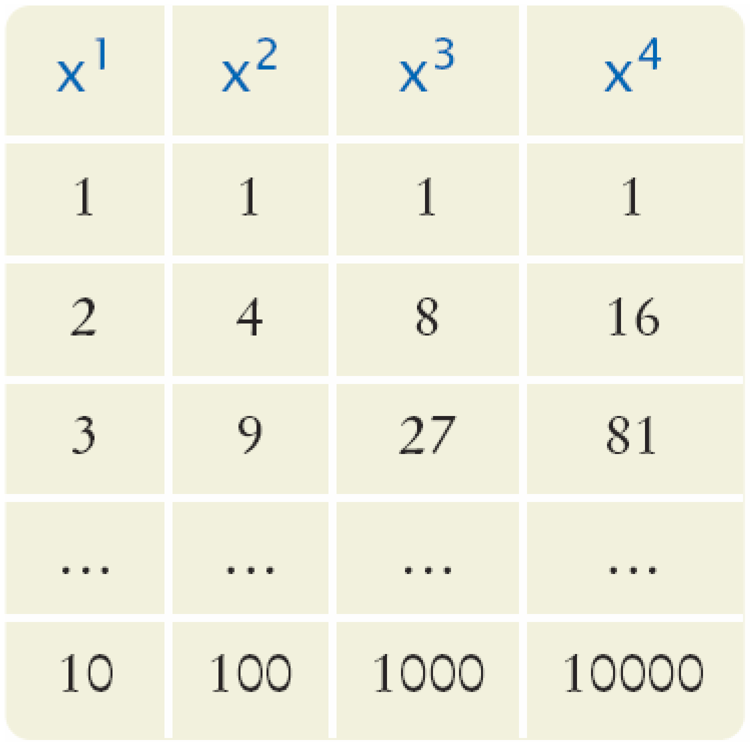
- In Chapter Three we learned how to nest **if** statements to allow us to make complex decisions
  - Remember that to nest the **if** statements we need to indent the code block

- Complex problems sometimes require a nested loop, one loop nested inside another loop
  - The nested loop will be indented inside the code block of the first loop

- A good example of using nested loops is when you are processing cells in a table
  - The outer loop iterates over all of the rows in the table
  - The inner loop processes the columns in the current row

Our Example Problem Statement

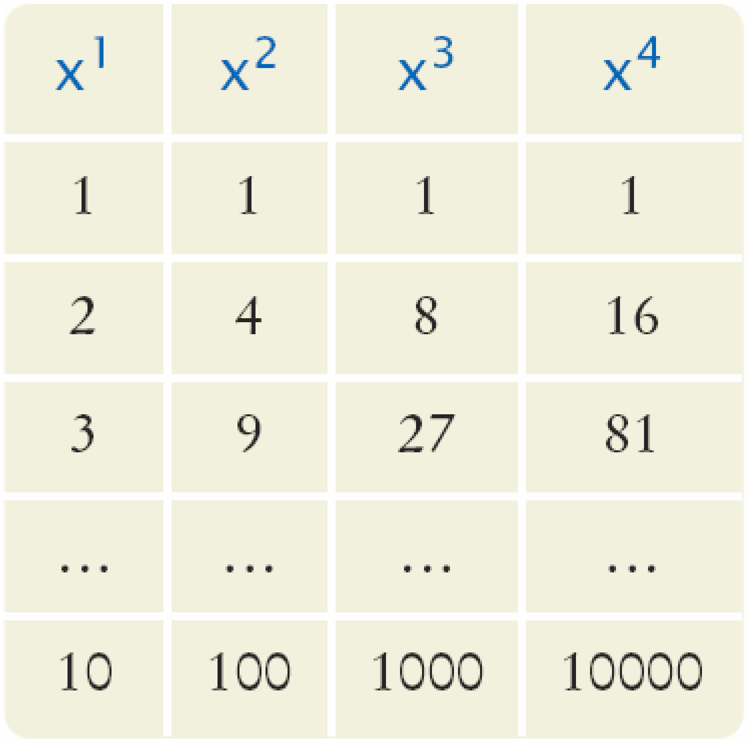
- Print a Table Header that contains x^1, x^2, x^3, and x^4
- Print a Table with four columns and ten rows that contain the powers of x^1, x^2, x^3, and x^4 for x = 1 to 10



x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Applying Nested Loops

- How would you print a table with rows and columns?
  - Print top line (header)
    - Use a for loop
  - Print table body…
    - How many rows are in the table?
    - How many columns in the table?
  - Loop per row
    - Loop per column
- In our example there are:
  - Four columns in the table
  - Ten rows in the table



x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Pseudocode to Print the Table

Print the table header

```
for x from 1 to 10
  print a new table row
  print a new line
```

- How do we print a table row?

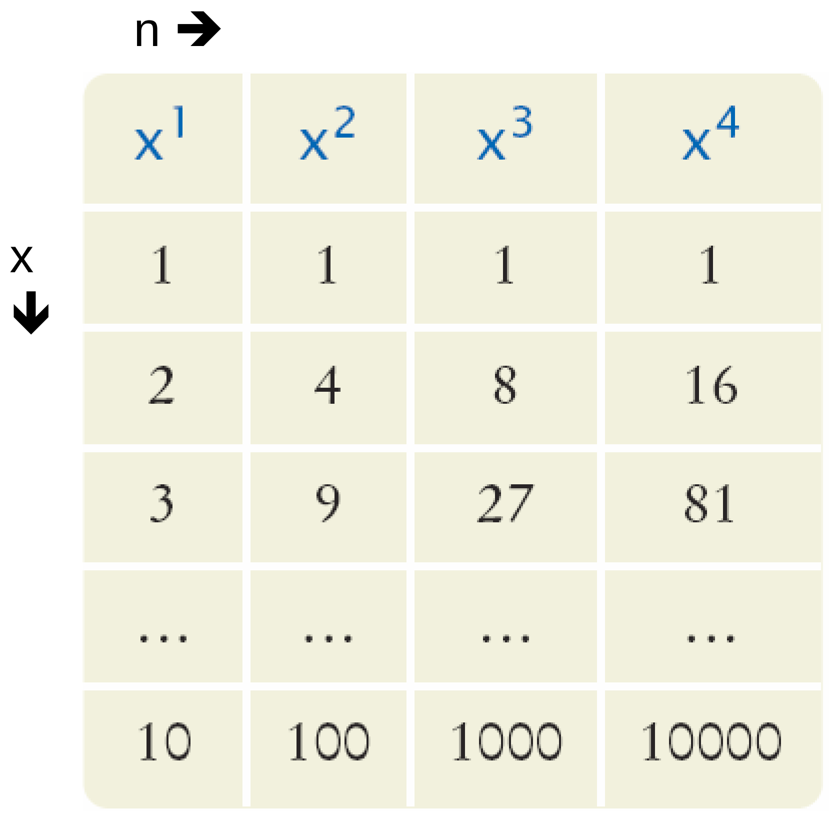
```
For n from 1 to 4
  print $x^n$
```

- We have to place this loop inside the preceding loop
  - The inner loop is *"nested"* inside the outer loop

Pseudocode to Print the Table

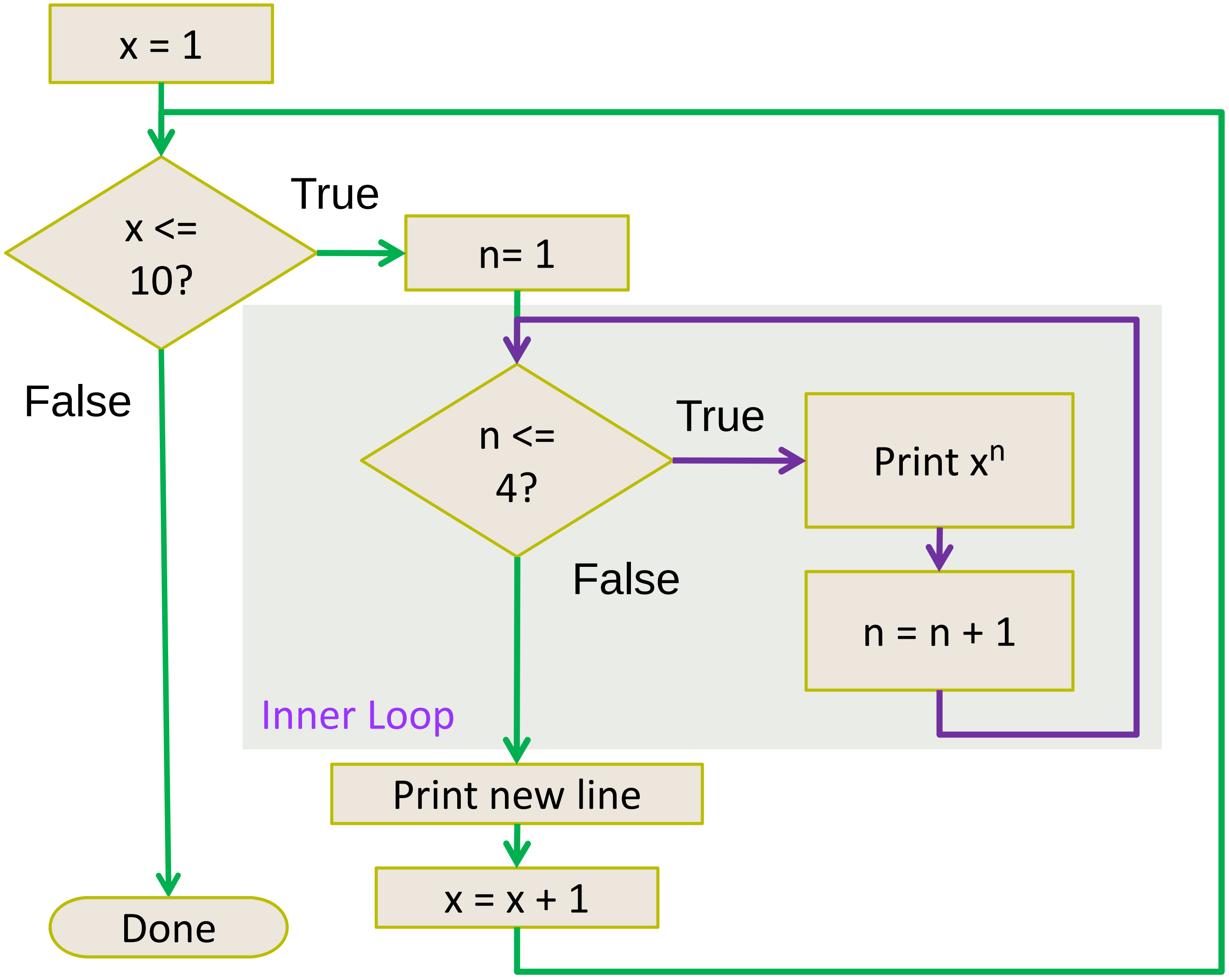
Print the table header:

```
for x from 1 to 10
  for n from 1 to 4
    print $X^n$
  print a new line
```

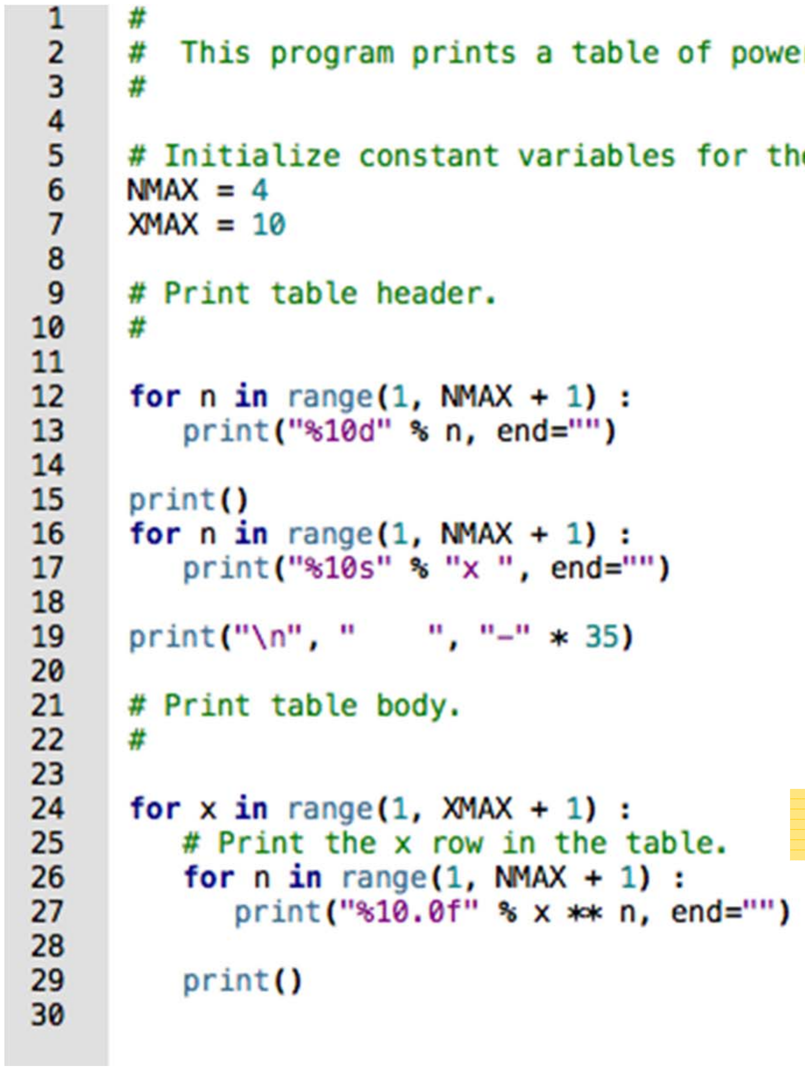


x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
...	...	...	...
10	100	1000	10000

Flowchart of a Nested Loop



Powertable.py



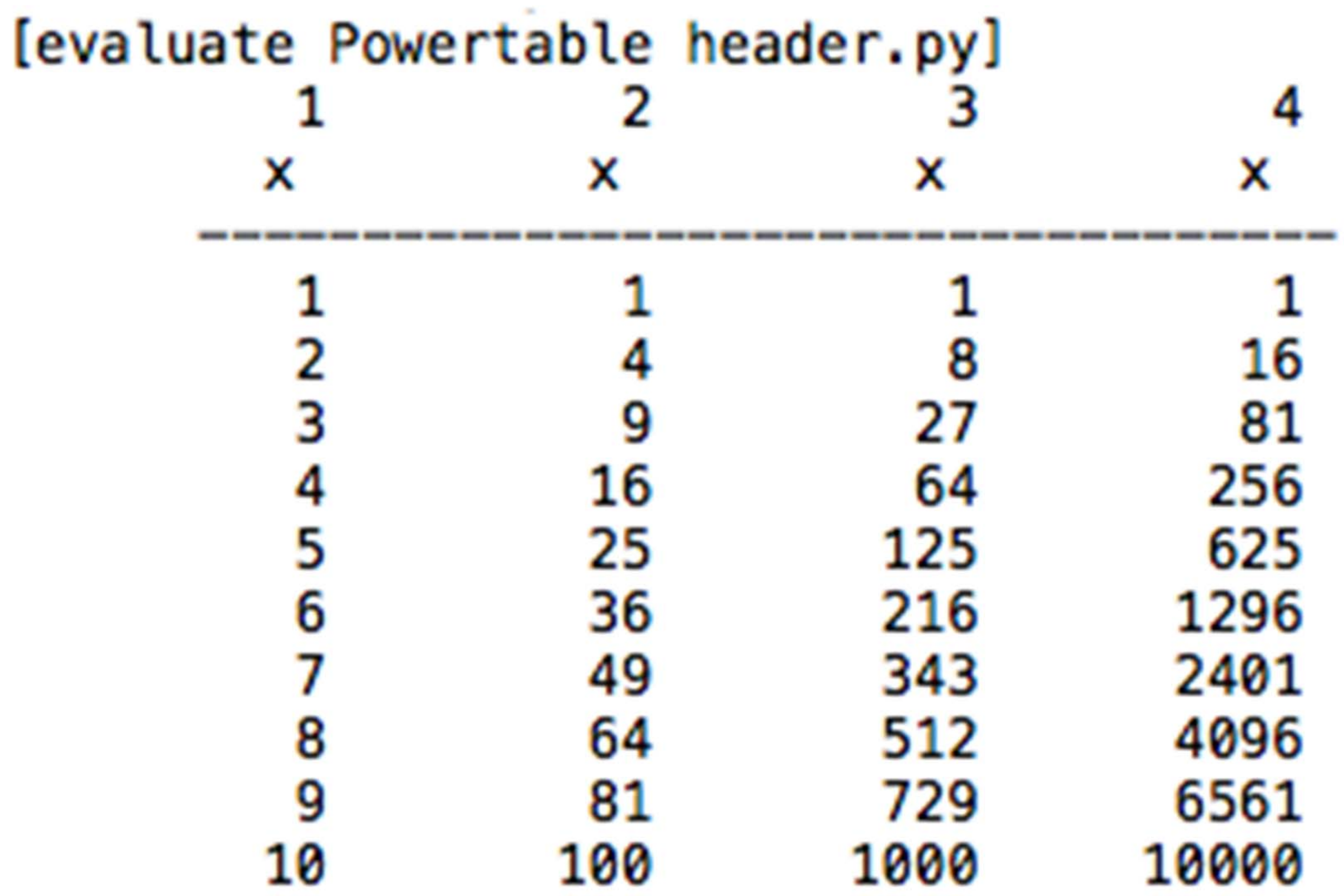
```python
#
#  This program prints a table of powers of x.
#

# Initialize constant variables for the max ranges.
NMAX = 4
XMAX = 10

# Print table header.
#

for n in range(1, NMAX + 1) :
   print("%10d" % n, end="")

print()
for n in range(1, NMAX + 1) :
   print("%10s" % "x ", end="")

print("\n", "    ", "-" * 35)

# Print table body.
#

for x in range(1, XMAX + 1) :
   # Print the x row in the table.
   for n in range(1, NMAX + 1) :
      print("%10.0f" % x ** n, end="")

   print()
```

The **end=“”** suppresses the new line, so the numbers are all printed on the same line

Body of outer loop, x = 1 ➔ 10

Body of inner loop, n = 1 ➔ 4

The Results



```
[evaluate Powertable header.py]
         1         2         3         4
        x         x         x         x 
      -----------------------------------
         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

Nested Loop Examples

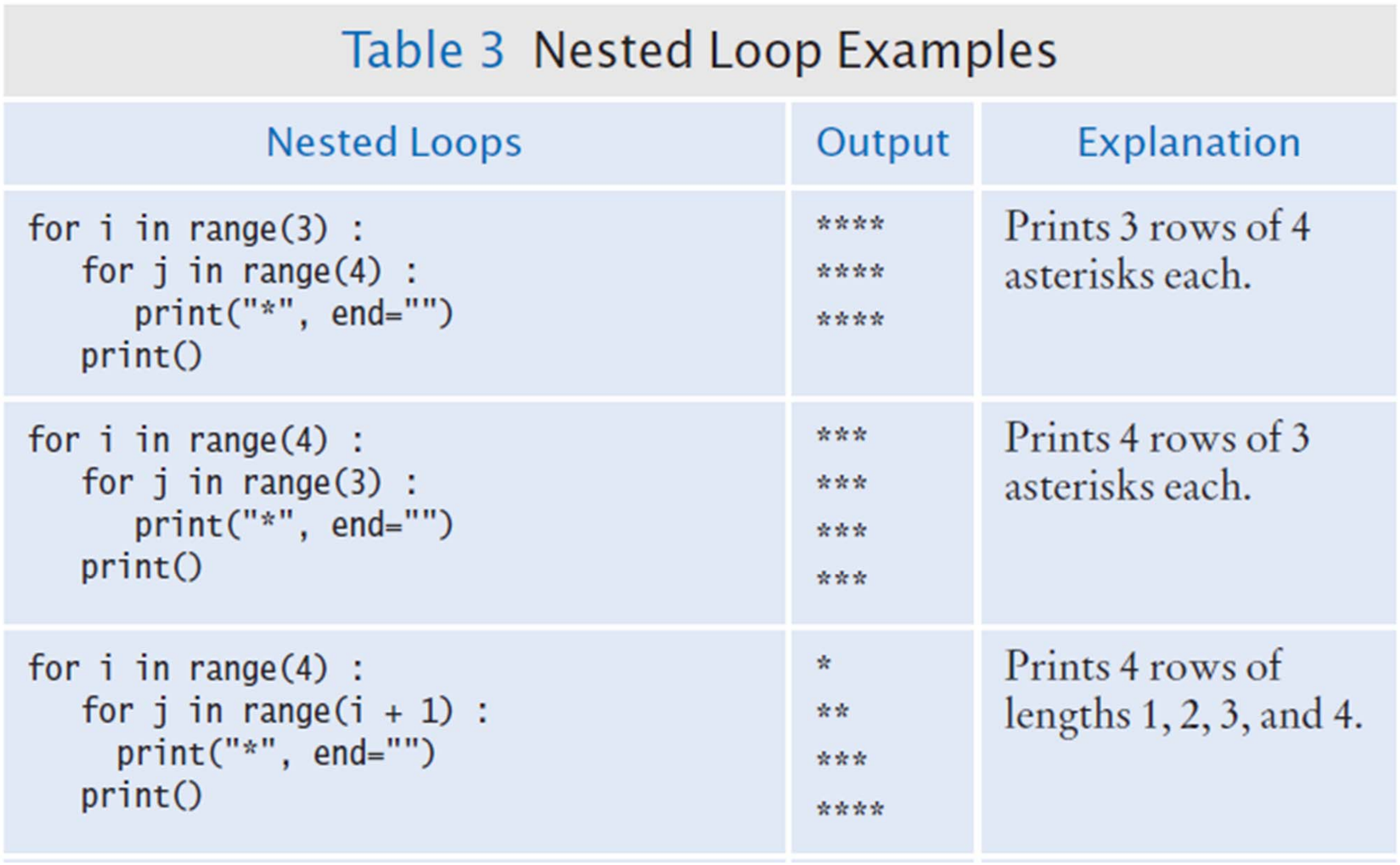


Table 3 Nested Loop Examples

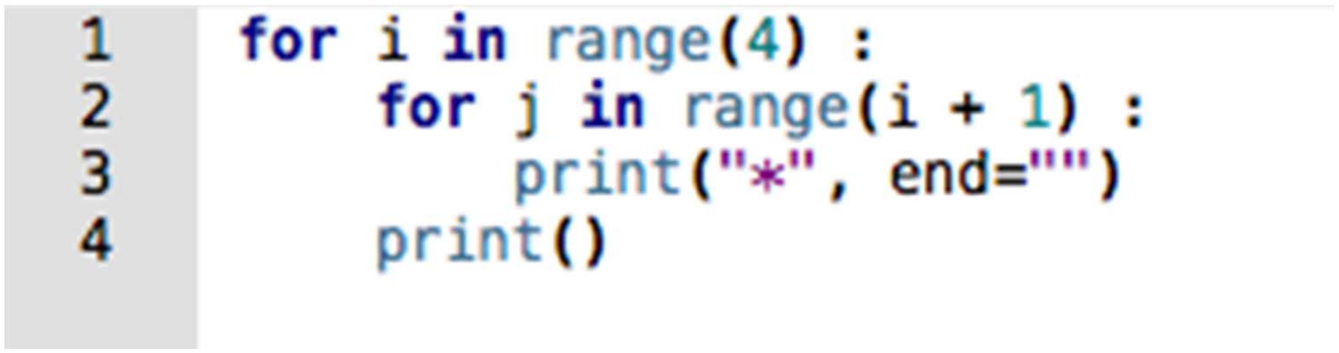
Nested Loops	Output	Explanation

`for i in range(3) :`
`   for j in range(4) :`
`      print("*", end="")`
`   print()`
	`****`
	`****`
	`****`
		Prints 3 rows of 4 asterisks each.

`for i in range(4) :`
`   for j in range(3) :`
`      print("*", end="")`
`   print()`
	`***`
	`***`
	`***`
	`***`
		Prints 4 rows of 3 asterisks each.

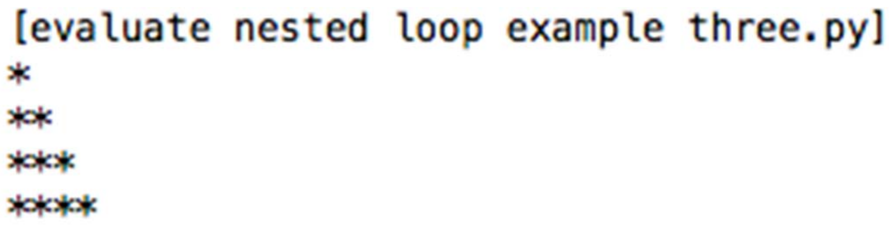
`for i in range(4) :`
`   for j in range(i + 1) :`
`     print("*", end="")`
`   print()`
	`*`
	`**`
	`***`
	`****`
		Prints 4 rows of lengths 1, 2, 3, and 4.

Hand Tracing the Loop



```
for i in range(4) :
    for j in range(i + 1) :
        print("*", end="")
    print()
```

- i will have the values:
  - 0, 1, 2, 3 – So we will have four lines of stars
- j will have the values
  - 0 - So we will have one star
  - 0, 1 - So we will have two stars
  - 0, 1, 2 - So we will have three stars
  - 0, 1, 2, 3 - So we will have four stars



```
[evaluate nested loop example three.py]
*
**
***
****
```

Nested Loop Examples (2)

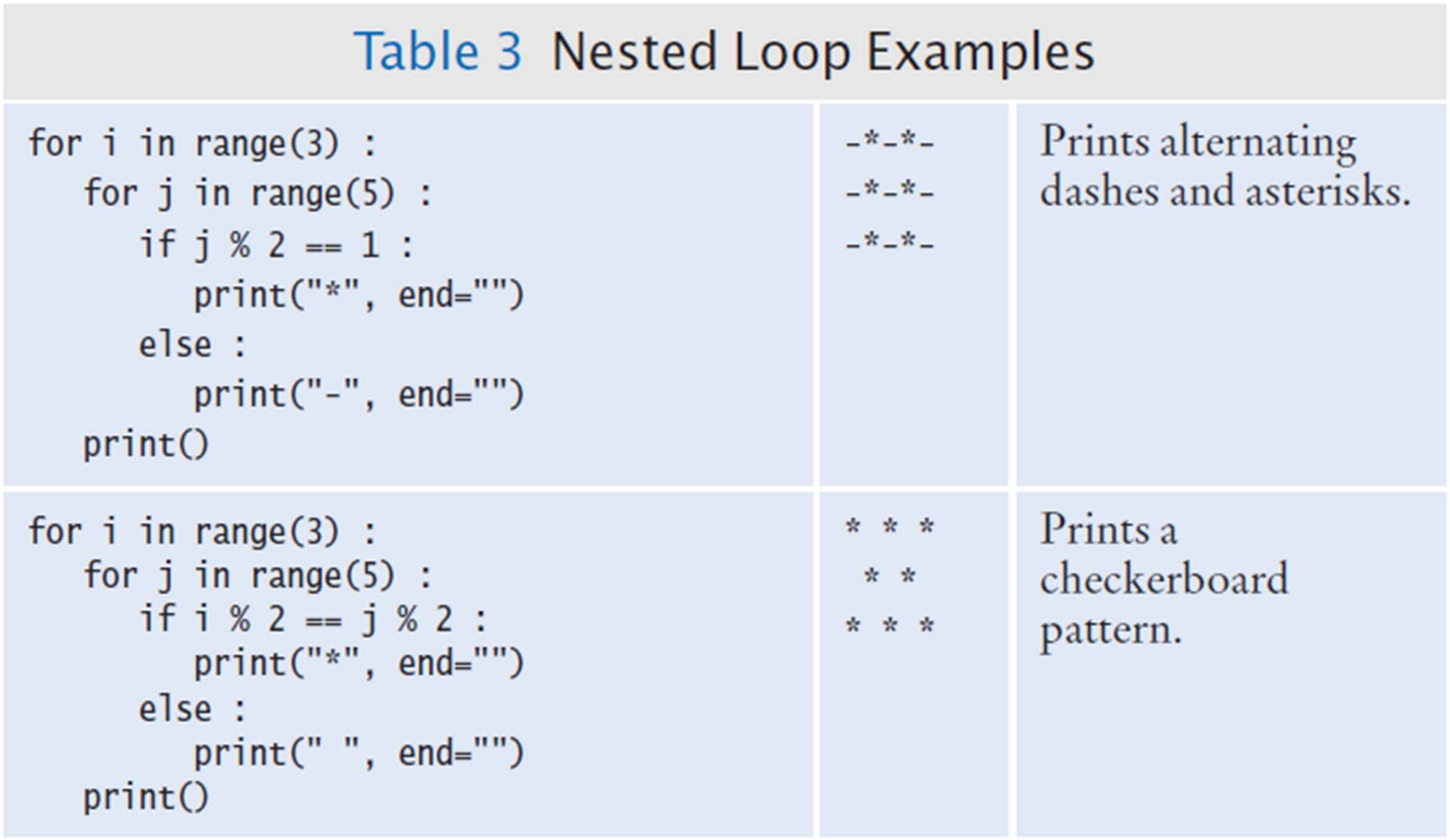


Table 3 Nested Loop Examples

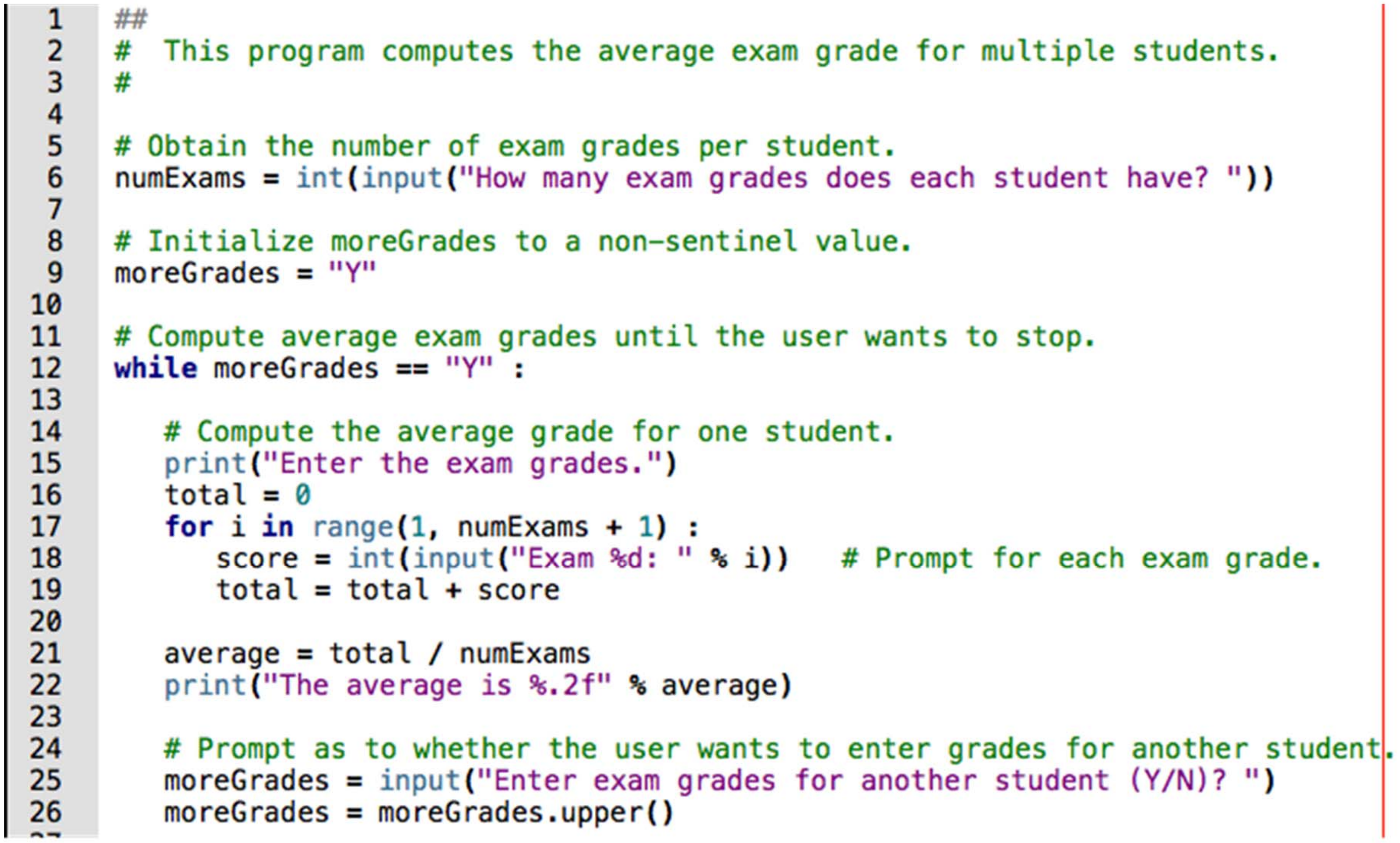
for i in range(3) :
   for j in range(5) :
      if j % 2 == 1 :
         print("*", end="")
      else :
         print("-", end="")
   print()
	-*-*-
	-*-*-
	-*-*-
		Prints alternating dashes and asterisks.

for i in range(3) :
   for j in range(5) :
      if i % 2 == j % 2 :
         print("*", end="")
      else :
         print(" ", end="")
   print()
	* * *
	 * *
	* * *
		Prints a checkerboard pattern.

Exam Averages Problem Statement

- It is common to repeatedly read and process multiple groups of values:
  - Write a program that can compute the average exam grade for multiple students.
  - Each student has the same number of exam grades
  - Prompt the user for the number of exams
  - When you finish a student prompt the user to see if there are more students to process
- What do we know?
- What do we need to compute?
- What is our algorithm / approach?

Translate to Python



```
##
#  This program computes the average exam grade for multiple students.
#

# Obtain the number of exam grades per student.
numExams = int(input("How many exam grades does each student have? "))

# Initialize moreGrades to a non-sentinel value.
moreGrades = "Y"

# Compute average exam grades until the user wants to stop.
while moreGrades == "Y" :

    # Compute the average grade for one student.
    print("Enter the exam grades.")
    total = 0
    for i in range(1, numExams + 1) :
        score = int(input("Exam %d: " % i))    # Prompt for each exam grade.
        total = total + score

    average = total / numExams
    print("The average is %.2f" % average)

    # Prompt as to whether the user wants to enter grades for another student.
    moreGrades = input("Enter exam grades for another student (Y/N)? ")
    moreGrades = moreGrades.upper()
```

Processing Strings

Processing Strings

- A common use of loops is to process or evaluate strings.
- For example, you may need to count the number of occurrences of one or more characters in a string or verify that the contents of a string meet certain criteria.

String Processing Examples & Self-Study

- Counting Matches
- Finding All Matches
- Finding the First or Last Match
- Validating a String

Counting Matches

- Suppose you need to count the number of uppercase letters contained in a string.
- We can use a for loop to check each character in the string to see if it is upper case
- The loop below sets the variable **char** equal to each successive character in the string
- Each pass through the loop tests the next character in the string to see if it is uppercase

```
uppercase = 0
for char in string :
   if char.isupper() :
      uppercase = uppercase + 1
```

Counting Vowels

- Suppose you need to count the vowels within a string
- We can use a for loop to check each character in the string to see if it is in the string of vowels “aeiuo”
- The loop below sets the variable **char** equal to each successive character in the string
- Each pass through the loop tests the lower case of the next character in the string to see if it is in the string “aeiou”

```
vowels = 0
for char in word :
   if char.lower() in "aeiou" :
      vowels = vowels + 1
```

Finding All Matches Example

- When you need to examine every character in a string, independent of its position we can use a for statement to examine each character
- If we need to print the position of each uppercase letter in a sentence we can test each character in the string and print the position of all uppercase characters
- We set the range to be the length of the string
  - We test each character
  - If it is uppercase we print I, its position in the string

```
sentence = input("Enter a sentence: ")
for i in range(len(sentence)) :
   if sentence[i].isupper() :
      print(i)
```

Finding the First Match

- This example finds the position of the first digit in a string.

```
found = False
position = 0
while not found and position < len(string) :
   if string[position].isdigit() :
      found = True
   else :
      position = position + 1

if found :
   print("First digit occurs at position", position)
else :
   print("The string does not contain a digit.")
```

Finding the Last Match

- Here is a loop that finds the position of the last digit in the string.
- This approach uses a while loop to start at the last character in a string and test each value moving from the end of the string to the start of the string
  - Position is set to the length of the string - 1
  - If the character is not a digit, we decrease position by 1
  - Until we find a digit, or process all the characters

```
found = False
position = len(string) - 1
while not found and position >= 0 :
   if string[position].isdigit() :
      found = True
   else :
      position = position - 1
```

Validating a String

- In the United States, telephone numbers consist of three parts—area code exchange, and line number—which are commonly specified in the form (###)###-####.

Validating a String (code)

- We can examine a string to ensure that it contains a correctly formatted phone number. (e.g., (703)321-6753)
- The loop test each character to see it it is correct for its position, or a number

```
valid = len(string) == 13
position = 0
while valid and position < len(string) :
   valid = ((position == 0 and string[position] != "(")
       or (position == 4 and string[position] != ")")
       or (position == 8 and string[position] != "-")
       or (position != 0 and position != 4 and position != 8
          and string[position].isdigit())) :
   position = position + 1
```

Application: Random Numbers and Simulations

Random Numbers/Simulations

- Games often use random numbers to make things interesting
  - Rolling Dice
  - Spinning a wheel
  - Pick a card
- A simulation usually involves looping through a sequence of events
  - Days
  - Events

Generating Random Numbers

- The Python library has a *random number generator* that produces numbers that appear to be random
  - The numbers are not completely random. The numbers are drawn from a sequence of numbers that does not repeat for a long time
  - random() returns a number that is >= 0 and < 1

Simulating Die Tosses

- Goal:
  - To generate a random integer in a given range we use the randint() function
  - Randint has two parameters, the range (inclusive) of numbers generated

ch04/dice.py

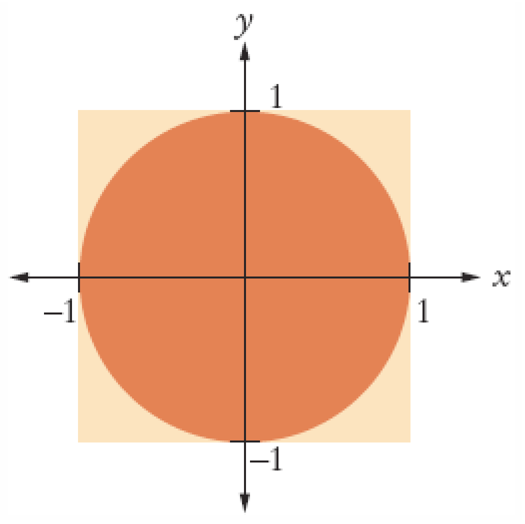
```python
##
#  This program simulates tosses of a pair of dice.
#

from random import randint

for i in range(10) :
    # Generate two random numbers between 1 and 6, inclusive.
    d1 = randint(1, 6)
    d2 = randint(1, 6)

    # Print the two values.
    print(d1, d2)
```

Program Run

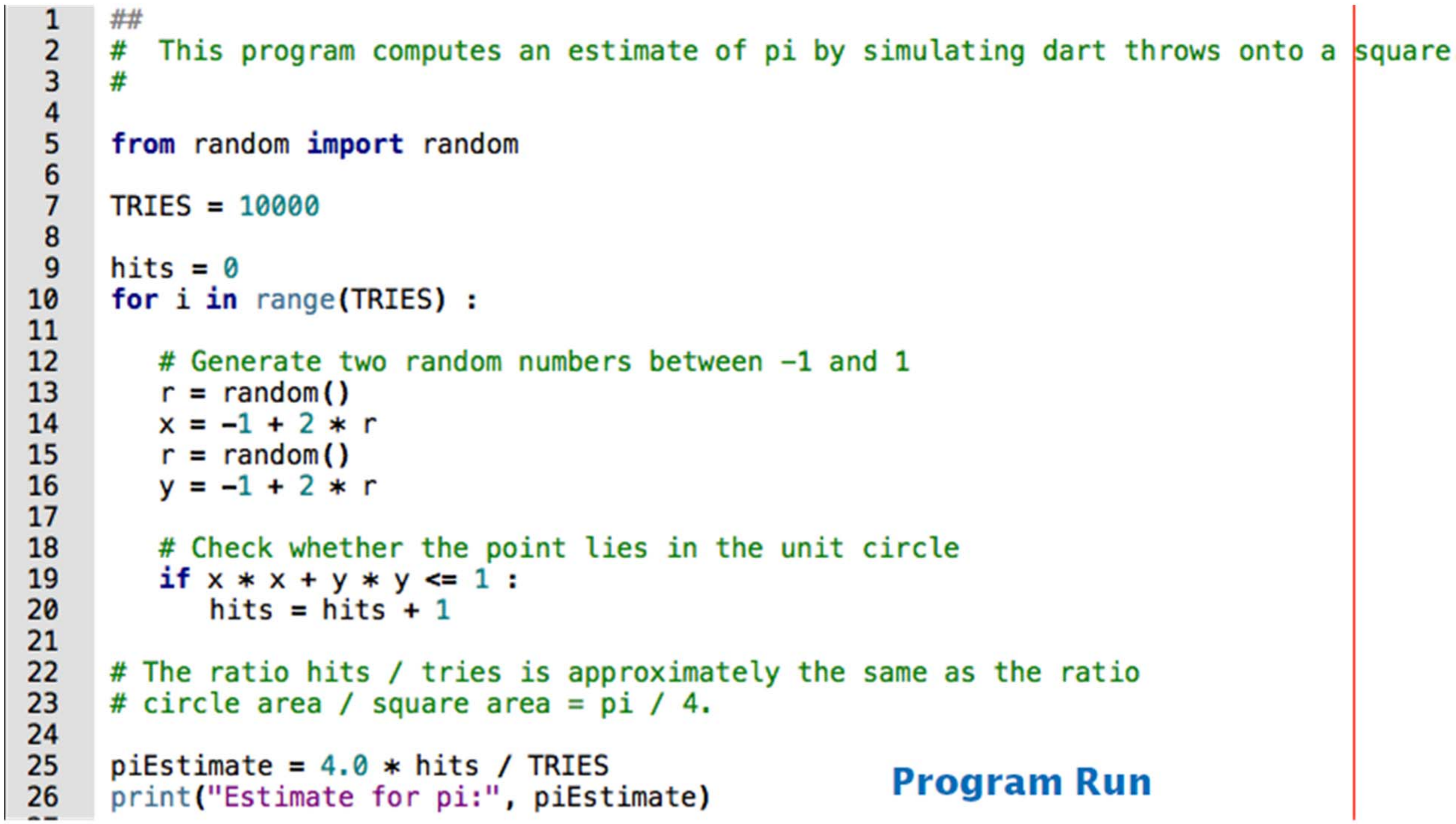
```
1 5
6 4
1 1
4 5
6 4
3 2
4 2
3 5
5 2
4 5
```

The Monte Carlo Method

- Used to find approximate solutions to problems that cannot be precisely solved
- Example: Approximate PI using the relative areas of a circle inside a square
  - Uses simple arithmetic
  - Hits are inside circle
  - Tries are total number of tries
  - Ratio is 4 x Hits / Tries



Monte Carlo Example



```python
##
#  This program computes an estimate of pi by simulating dart throws onto a square
#

from random import random

TRIES = 10000

hits = 0
for i in range(TRIES) :

   # Generate two random numbers between -1 and 1
   r = random()
   x = -1 + 2 * r
   r = random()
   y = -1 + 2 * r

   # Check whether the point lies in the unit circle
   if x * x + y * y <= 1 :
      hits = hits + 1

# The ratio hits / tries is approximately the same as the ratio
# circle area / square area = pi / 4.

piEstimate = 4.0 * hits / TRIES
print("Estimate for pi:", piEstimate)
```

Program Run

```
Estimate for pi: 3.1464
```

Summary

Summary: Two Types of Loops

- **while** Loops
- **for** Loops
- **while** loops are very commonly used (general purpose)
- Uses of the **for** loop:
  - The **for** loop can be used to iterate over the contents of any container.
  - A **for** loop can also be used as a count-controlled loop that iterates over a range of integer values.

Summary

- Each loop requires the following steps:
  - Initialization (setup variables to start looping)
  - Condition (test if we should execute loop body)
  - Update (change something each time through)
- A loop executes instructions repeatedly while a condition is True.
- An off-by-one error is a common error when programming loops.
  - Think through simple test cases to avoid this type of error.

Summary

- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a boolean variable to control a loop.
  - Set the variable to `True` before entering the loop
  - Set it to `False` to leave the loop.
- Loops can be used in conjunction with many string processing tasks

Summary

- In a simulation, you use the computer to simulate an activity.
  - You can introduce randomness by calling the random number generator.