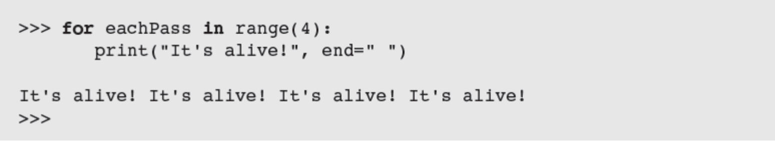
Definite Iteration: The `for` Loop

- Repetition statements (or **loops**) repeat an action
- Each repetition of action is known as **pass** or **iteration**
- Two types of loops
  - Those that repeat action a predefined number of times (**definite iteration**)
  - Those that perform action until program determines it needs to stop (**indefinite iteration**)

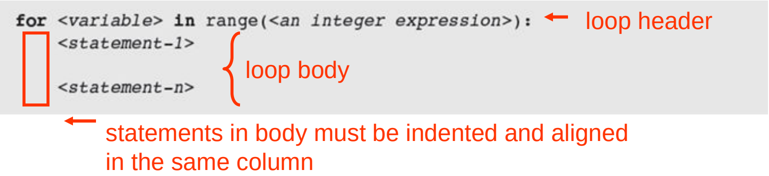
Executing a Statement a Given Number of Times

- Python’s **`for`** loop is the control statement that most easily supports definite iteration



```
>>> for eachPass in range(4):
        print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
>>>
```

- The form of this type of loop is:



```
for <variable> in range(<an integer expression>):
    <statement-1>

    <statement-n>
```

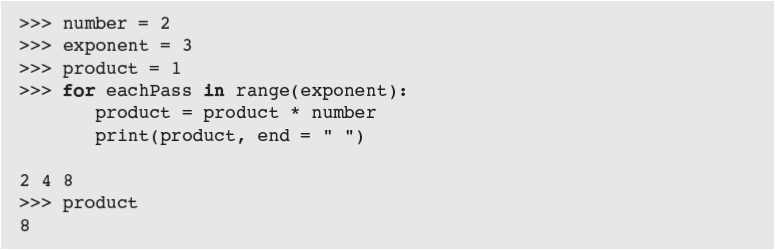
loop header

loop body

statements in body must be indented and aligned in the same column

Executing a Statement a Given Number of Times (continued)

- Example: Loop to compute an exponentiation for a non-negative exponent

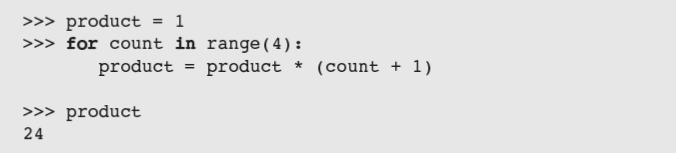


```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
        product = product * number
        print(product, end = " ")

2 4 8
>>> product
8
```

- If the exponent were 0, the loop body would not execute and value of **`product`** would remain as 1

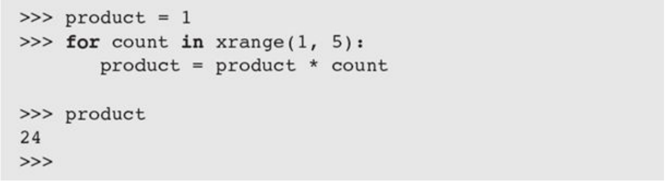
Count-Controlled Loops

- Loops that count through a range of numbers



```
>>> product = 1
>>> for count in range(4):
        product = product * (count + 1)

>>> product
24
```

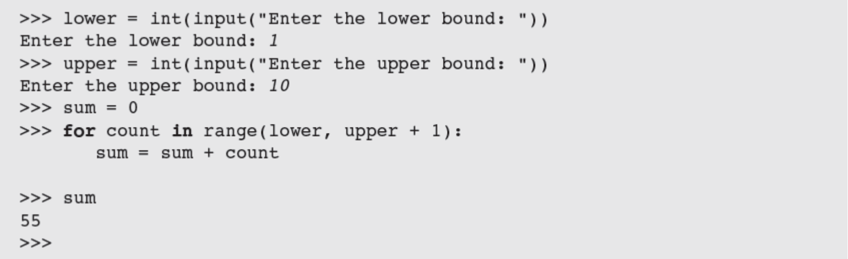
- To specify a explicit lower bound:



```
>>> product = 1
>>> for count in xrange(1, 5):
        product = product * count

>>> product
24
>>>
```

Count-Controlled Loops (continued)

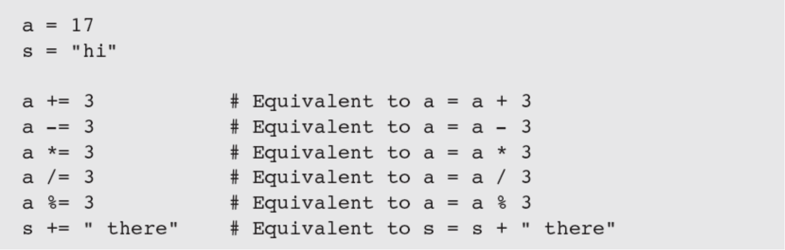
- Example: bound-delimited **summation**



```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> sum = 0
>>> for count in range(lower, upper + 1):
       sum = sum + count

>>> sum
55
>>>
```

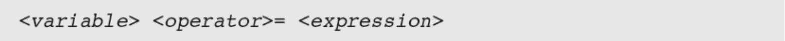
Augmented Assignment

- **Augmented assignment operations**:



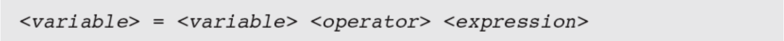
```
a = 17
s = "hi"

a += 3          # Equivalent to a = a + 3
a -= 3          # Equivalent to a = a - 3
a *= 3          # Equivalent to a = a * 3
a /= 3          # Equivalent to a = a / 3
a %= 3          # Equivalent to a = a % 3
s += " there"   # Equivalent to s = s + " there"
```

- Format:



```
<variable> <operator>= <expression>
```

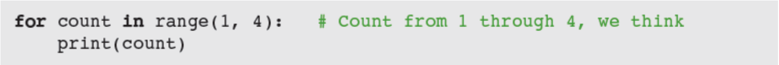
Equivalent to:



```
<variable> = <variable> <operator> <expression>
```

Loop Errors: Off-by-One Error

- Example:



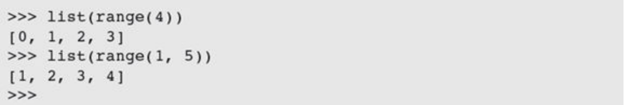
```
for count in range(1, 4):   # Count from 1 through 4, we think
    print(count)
```

Loop actually counts from 1 through 3

- This is not a syntax error, but rather a logic error

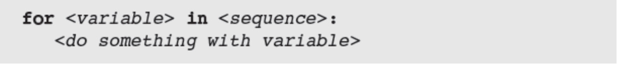
Traversing the Contents of a Data Sequence

- **`range`** returns a **list**



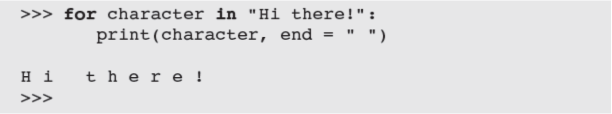
```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>>
```

- Strings are also sequences of characters
- Values in a sequence can be visited with a **`for`** loop:



```
for <variable> in <sequence>:
   <do something with variable>
```

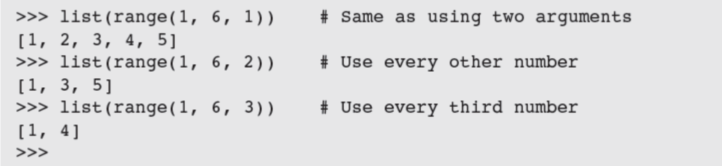
- Example:



```
>>> for character in "Hi there!":
       print(character, end = " ")

H i   t h e r e !
>>>
```

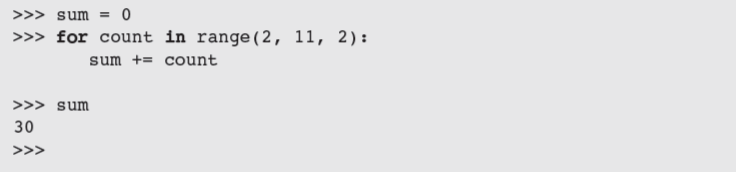
Specifying the Steps in the Range

- **`range`** expects a third argument that allows you specify a **step value**



```
>>> list(range(1, 6, 1))    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))    # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))    # Use every third number
[1, 4]
>>>
```

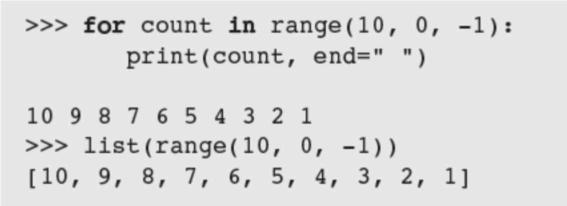
- Example in a loop:



```
>>> sum = 0
>>> for count in range(2, 11, 2):
       sum += count

>>> sum
30
>>>
```

Loops That Count Down

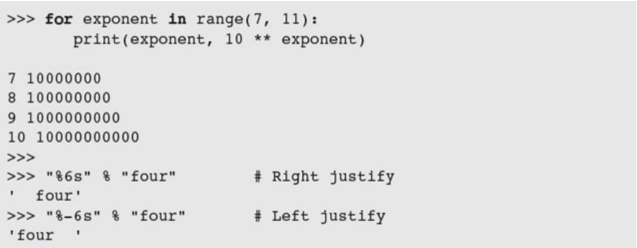
- Example:



```
>>> for count in range(10, 0, -1):
       print(count, end=" ")

10 9 8 7 6 5 4 3 2 1
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

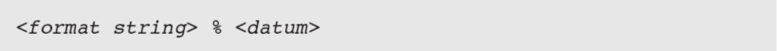
Formatting Text for Output

- Many data-processing applications require output that has **tabular format**
- **Field width**: Total number of data characters and additional spaces for a datum in a formatted string



```
>>> for exponent in range(7, 11):
       print(exponent, 10 ** exponent)

7 10000000
8 100000000
9 1000000000
10 10000000000
>>>
>>> "%6s" % "four"        # Right justify
'  four'
>>> "%-6s" % "four"       # Left justify
'four  '
```

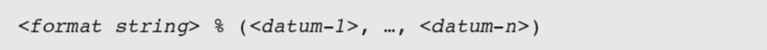
Formatting Text for Output (continued)



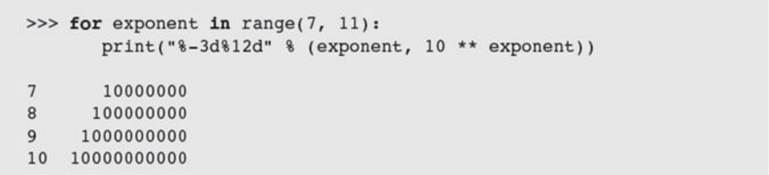
```
<format string> % <datum>
```

- This version contains **format string**, **format operator `%`**, and single data value to be formatted
- To format integers, letter **`d`** is used instead of **`s`**

- To format sequence of data values:



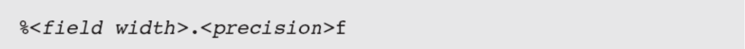
```
<format string> % (<datum-1>, …, <datum-n>)
```



```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))

7      10000000
8     100000000
9    1000000000
10  10000000000
```

Formatting Text for Output (continued)

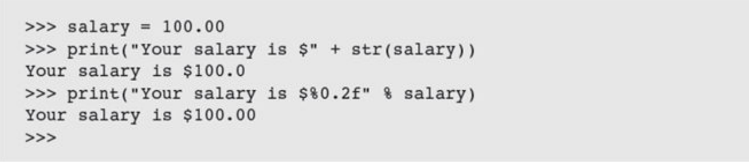
- To format data value of type **`float`**:



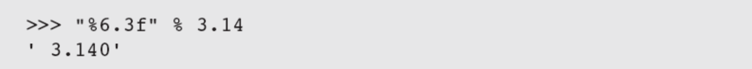
```
%<field width>.<precision>f
```

where **`.<precision>`** is optional

- Examples:



```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
>>>
```



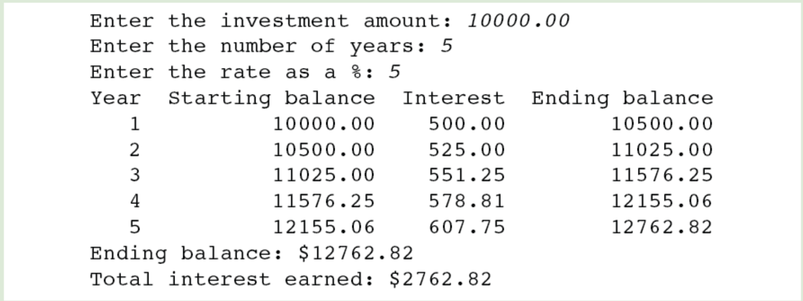
```
>>> "%6.3f" % 3.14
' 3.140'
```

Case Study: An Investment Report

- Request:
  - – Write a program that computes an investment report

Case Study: An Investment Report (continued)

- Analysis:



```
Enter the investment amount: 10000.00
Enter the number of years: 5
Enter the rate as a %: 5
Year  Starting balance  Interest  Ending balance
   1          10000.00    500.00        10500.00
   2          10500.00    525.00        11025.00
   3          11025.00    551.25        11576.25
   4          11576.25    578.81        12155.06
   5          12155.06    607.75        12762.82
Ending balance: $12762.82
Total interest earned: $2762.82
```

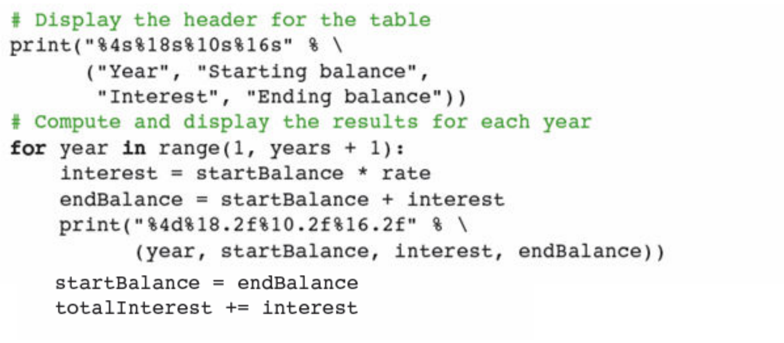
[FIGURE 3.1] The user interface for the investment report program

Case Study: An Investment Report (continued)

- Design:
  - Receive the user’s inputs and initialize data
  - Display the table’s header
  - Compute results for each year and display them
  - Display the totals

Case Study: An Investment Report (continued)

- Coding:



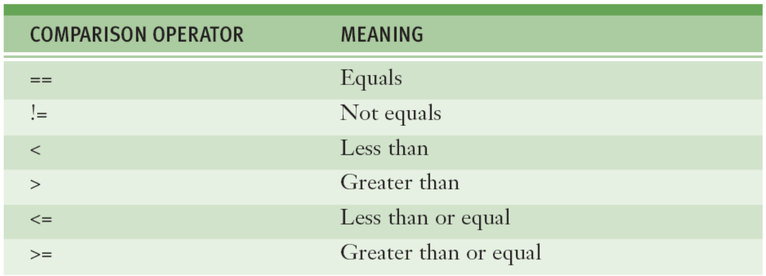
```
# Display the header for the table
print("%4s%18s%10s%16s" % \
      ("Year", "Starting balance",
       "Interest", "Ending balance"))
# Compute and display the results for each year
for year in range(1, years + 1):
    interest = startBalance * rate
    endBalance = startBalance + interest
    print("%4d%18.2f%10.2f%16.2f" % \
          (year, startBalance, interest, endBalance))
    startBalance = endBalance
    totalInterest += interest
```

Selection: `if` and `if-else` Statements

- **Selection statements** allow a computer to make choices
  - Based on a **condition**

The Boolean Type, Comparisons, and Boolean Expressions

- **Boolean data type** consists of two values: true and false (typically through standard **`True`**/**`False`**)



COMPARISON OPERATOR	MEANING
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

[TABLE 3.2] The comparison operators

- Example: `4 != 4` evaluates to `False`

`if-else` Statements

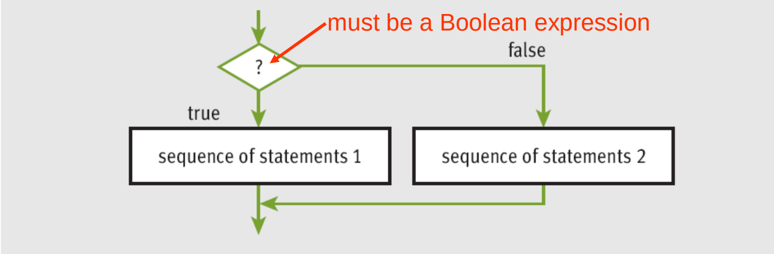
- Also called a **two-way selection statement**
- Often used to check inputs for errors:

```
import math

area = float(input("Enter the area: "))
if area > 0:
    radius = math.sqrt(area / math.pi)
    print("The radius is", radius)
else:
    print("Error: the area must be a positive number")
```

- Syntax:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

`if-else` Statements (continued)



must be a Boolean expression

[FIGURE 3.2] The semantics of the **`if-else`** statement

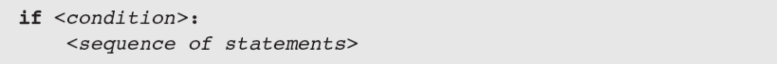
```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Better alternative:

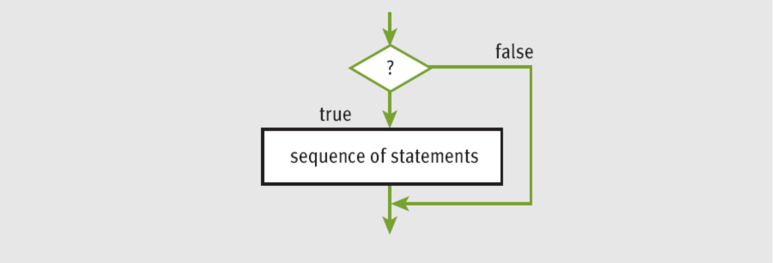
```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
print("Maximum:", max(first, second))
print("Minimum:", min(first, second))
```

One-Way Selection Statements

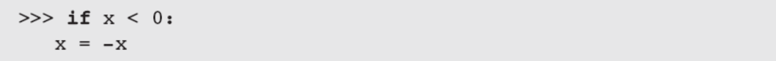
- Simplest form of selection is the ***if* statement**



```
if <condition>:
    <sequence of statements>
```



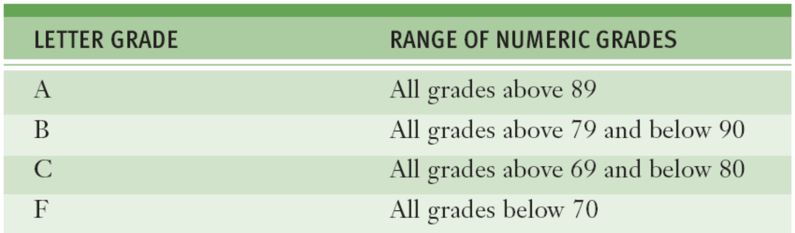
[FIGURE 3.3] The semantics of the **if** statement



```
>>> if x < 0:
   x = -x
```

Multi-way `if` Statements

- A program may be faced with testing conditions that entail more than two alternative courses of action

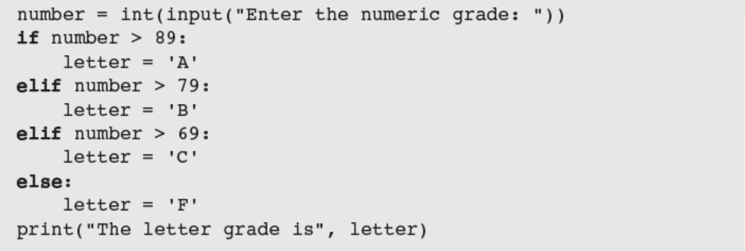


LETTER GRADE	RANGE OF NUMERIC GRADES
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
F	All grades below 70

[TABLE 3.3] A simple grading scheme

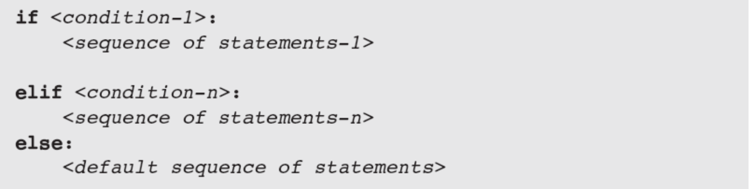
- Can be described in code by a **multi-way selection statement**

Multi-way `if` Statements (continued)



```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

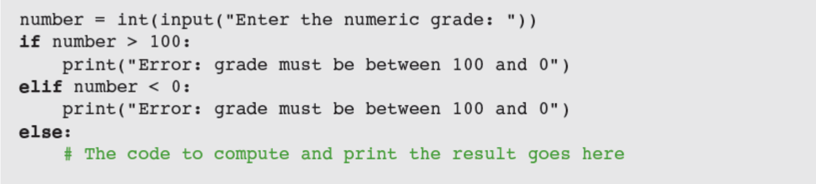
- Syntax:



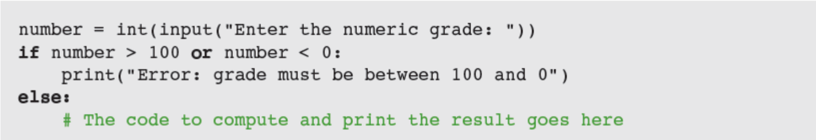
```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Logical Operators and Compound Boolean Expressions

- Often a course of action must be taken if either of two conditions is true: Below are two approaches



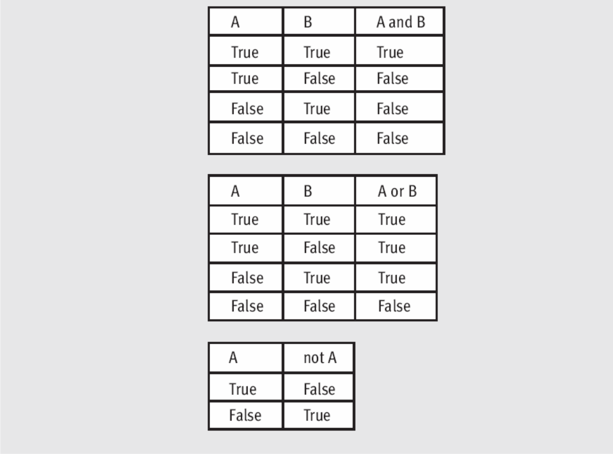
```
number = int(input("Enter the numeric grade: "))
if number > 100:
    print("Error: grade must be between 100 and 0")
elif number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```



```
number = int(input("Enter the numeric grade: "))
if number > 100 or number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

  - Could we use the `and` logical operator instead?

Logical Operators and Compound Boolean Expressions (continued)



A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

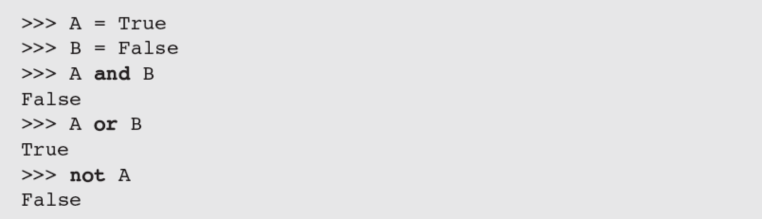
A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

[FIGURE 3.4] The truth tables for **and**, **or**, and **not**

Logical Operators and Compound Boolean Expressions (continued)

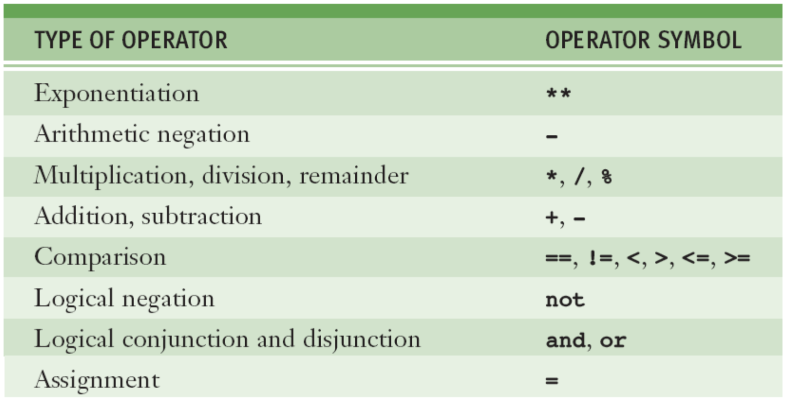
- Next example verifies some of the claims made in the previous truth tables:



```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

- The logical operators are evaluated after comparisons but before the assignment operator
  - **not** has higher precedence than **and** and **or**

Logical Operators and Compound Boolean Expressions (continued)

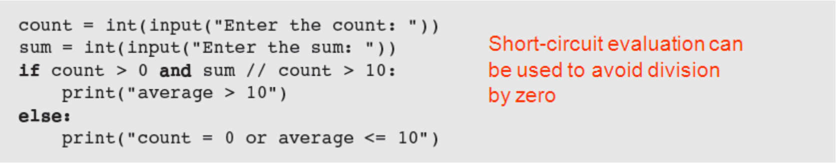


TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	`**`
Arithmetic negation	`-`
Multiplication, division, remainder	`*`, `/`, `%`
Addition, subtraction	`+`, `-`
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical negation	`not`
Logical conjunction and disjunction	`and`, `or`
Assignment	`=`

[TABLE 3-4] Operator precedence, from highest to lowest

Short-Circuit Evaluation

- In **`(A and B)`**, if **`A`** is false, then so is the expression, and there is no need to evaluate **`B`**
- In **`(A or B)`**, if **`A`** is true, then so is the expression, and there is no need to evaluate **`B`**
- **Short-circuit evaluation:** Evaluation stops as soon as possible



```
count = int(input("Enter the count: "))
sum = int(input("Enter the sum: "))
if count > 0 and sum // count > 10:
    print("average > 10")
else:
    print("count = 0 or average <= 10")
```

Short-circuit evaluation can be used to avoid division by zero

Testing Selection Statements

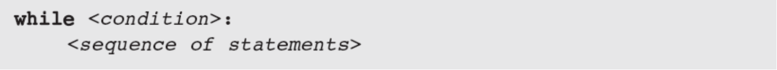
- Tips:
  - Make sure that all of the possible branches or alternatives in a selection statement are exercised
  - After testing all of the actions, examine all of the conditions
  - Test conditions that contain compound Boolean expressions using data that produce all of the possible combinations of values of the operands

Conditional Iteration: The `while` Loop

- The **`while`** loop can be used to describe conditional iteration
  - Example: A program's input loop that accepts values until user enters a **sentinel** that terminates the input

The Structure and Behavior of a `while` Loop

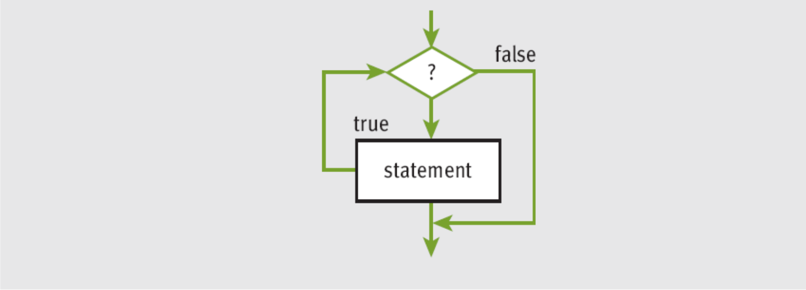
- Conditional iteration requires that condition be tested within loop to determine if it should continue
  - Called **continuation condition**



```
while <condition>:
    <sequence of statements>
```

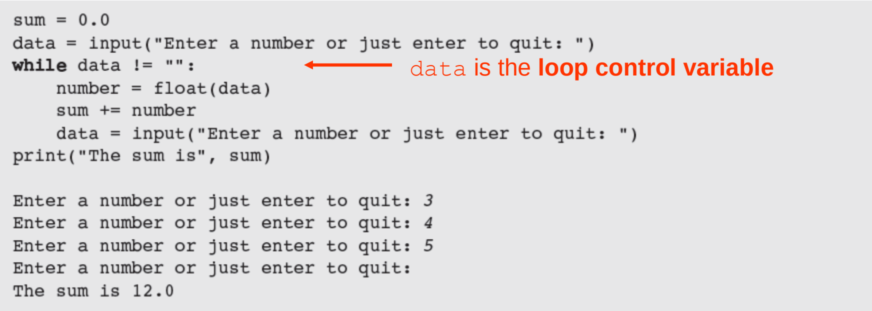
  - Improper use may lead to **infinite loop**
- **`while`** loop is also called **entry-control loop**
  - Condition is tested at top of loop
  - Statements within loop can execute zero or more times

The Structure and Behavior of a `while` Loop (continued)



[FIGURE 3.5] The semantics of a **`while`** loop

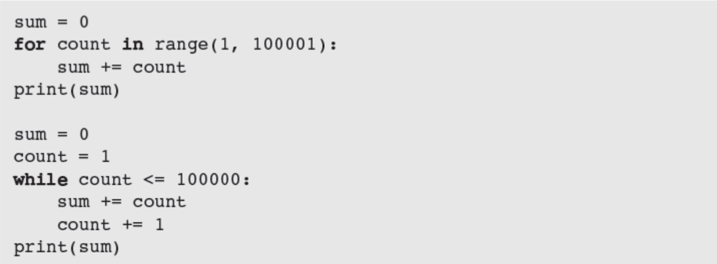
The Structure and Behavior of a `while` Loop (continued)



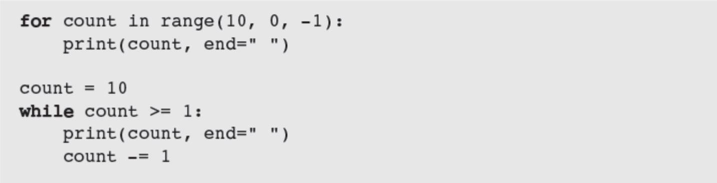
```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

← `data` is the **loop control variable**

Count Control with a `while` Loop



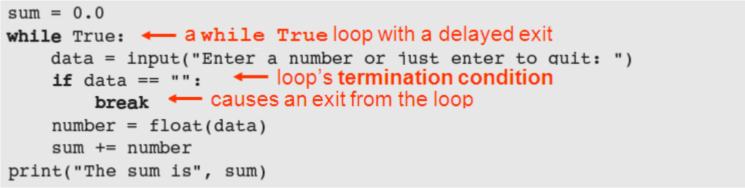
```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)

sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```



```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

The `while True` Loop and the `break` Statement

- **`while`** loop can be complicated to write correctly
  - Possible to simplify its structure and improve its readability



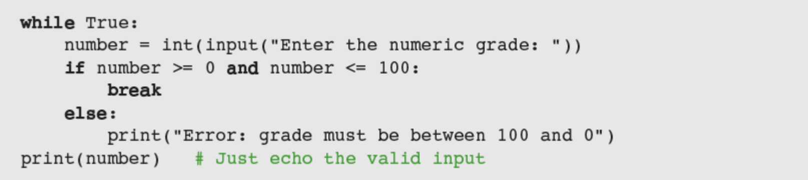
```
sum = 0.0
while True:
    data = input("Enter a number or just enter to quit: ")
    if data == "":
        break
    number = float(data)
    sum += number
print("The sum is", sum)
```

a **`while True`** loop with a delayed exit

loop's **termination condition**

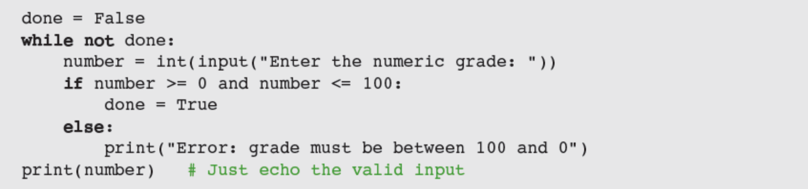
causes an exit from the loop

The `while True` Loop and the `break` Statement (continued)



```
while True:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        break
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

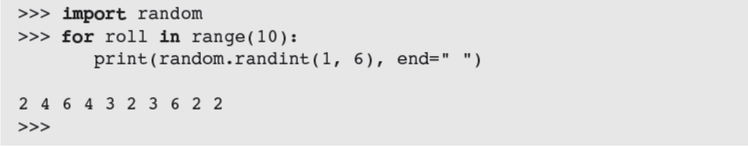
- Alternative: Use a Boolean variable to control loop



```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

Random Numbers

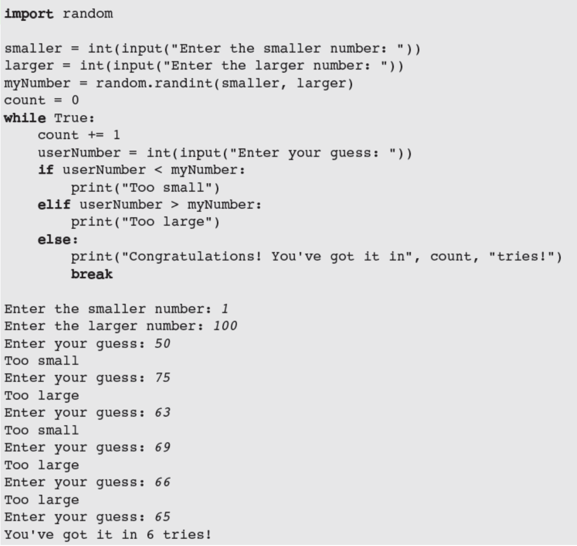
- Programming languages include resources for generating **random numbers**
- **`random`** module supports several ways to do this
  - **`randint`** returns random number from among numbers between two arguments, included



```
>>> import random
>>> for roll in range(10):
        print(random.randint(1, 6), end=" ")

2 4 6 4 3 2 3 6 2 2
>>>
```

- Example: A simple guessing game

Random Numbers (continued)



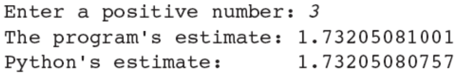
```
import random

smaller = int(input("Enter the smaller number: "))
larger = int(input("Enter the larger number: "))
myNumber = random.randint(smaller, larger)
count = 0
while True:
    count += 1
    userNumber = int(input("Enter your guess: "))
    if userNumber < myNumber:
        print("Too small")
    elif userNumber > myNumber:
        print("Too large")
    else:
        print("Congratulations! You've got it in", count, "tries!")
        break

Enter the smaller number: 1
Enter the larger number: 100
Enter your guess: 50
Too small
Enter your guess: 75
Too large
Enter your guess: 63
Too small
Enter your guess: 69
Too large
Enter your guess: 66
Too large
Enter your guess: 65
You've got it in 6 tries!
```

Loop Logic, Errors, and Testing

- Errors to rule out during testing **`while`** loop:
  - Incorrectly initialized loop control variable
  - Failure to update this variable correctly within loop
  - Failure to test it correctly in continuation condition
- To halt loop that appears to hang during testing, type **`Control+c`** in terminal window or IDLE shell
- If loop must run at least once, use a **`while True`** loop with delayed examination of termination condition
  - Ensure a **`break`** statement to be reached eventually

Case Study: Approximating Square Roots

- Request:
  - Write a program that computes square roots
- Analysis:

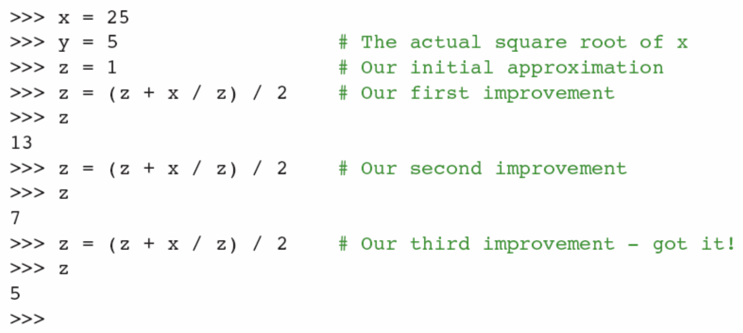


```
Enter a positive number: 3
The program's estimate: 1.73205081001
Python's estimate:      1.73205080757
```

- Design:
  - Use Newton’s square root approximation algorithm:
    - Square root *y* of a positive number *x* is the number *y* such that $y^2 = x$
    - If initial estimate of *y* is *z*, a better estimate of *y* can be obtained by taking the average of ***z*** together with *x*/*z*

Case Study: Approximating Square Roots (continued)

- A quick session with the Python interpreter shows this method of successive approximations in action:



```
>>> x = 25
>>> y = 5                   # The actual square root of x
>>> z = 1                   # Our initial approximation
>>> z = (z + x / z) / 2     # Our first improvement
>>> z
13
>>> z = (z + x / z) / 2     # Our second improvement
>>> z
7
>>> z = (z + x / z) / 2     # Our third improvement – got it!
>>> z
5
>>>
```

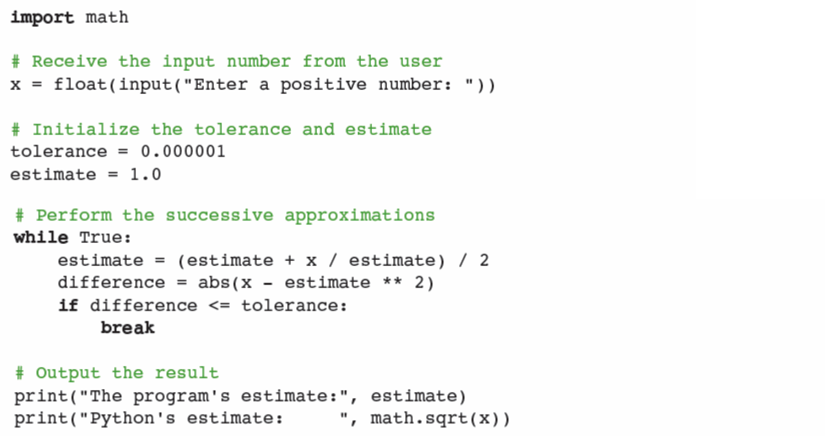
Case Study: Approximating Square Roots (continued)

- Design (continued): Algorithm

```
set x to the user’s input value
set tolerance to 0.000001
set estimate to 1.0
while True
    set estimate to (estimate + x / estimate) / 2
    set difference to abs(x - estimate ** 2)
    if difference <= tolerance:
          break
output the estimate
```

Case Study: Approximating Square Roots (continued)

- Implementation (Coding):



```
import math

# Receive the input number from the user
x = float(input("Enter a positive number: "))

# Initialize the tolerance and estimate
tolerance = 0.000001
estimate = 1.0

# Perform the successive approximations
while True:
    estimate = (estimate + x / estimate) / 2
    difference = abs(x - estimate ** 2)
    if difference <= tolerance:
        break

# Output the result
print("The program's estimate:", estimate)
print("Python's estimate:     ", math.sqrt(x))
```

Summary

- Control statements determine order in which other statements are executed in program
- Definite iteration is process of executing set of statements fixed, predictable number of times
  - Example: use **`for`** loop
- **`for`** loop consists of header and set of statements called body
  - Can be used to implement a count-controlled loop
    - Use `range` to generate sequence of numbers
  - Can traverse and visit the values in any sequence

Summary (continued)

- A format string and its operator `%` allow programmer to format data using field width and precision
- An off-by-one error occurs when loop does not perform intended number of iterations, there being one too many or one too few
- Boolean expressions evaluate to **`True`** or **`False`**
  - Constructed using logical operators: **`and`**, **`or`**, **`not`**
  - Python uses short-circuit evaluation in compound Boolean expressions
- Selection statements enable program to make choices

Summary (continued)

- **`if-else`** is a two-way selection statement
- Conditional iteration is the process of executing a set of statements while a condition is true
  - Use **`while`** loop (which is an entry-control loop)
- A **`break`** can be used to exit a loop from its body
- Any **`for`** loop can be converted to an equivalent **`while`** loop
- Infinite loop: Continuation condition never becomes false and no other exit points are provided
- **`random.randint`** returns a random number

dws4.me
EAST CENTRAL UNIVERSITY
GRATIA SCIENTIAE
FIUNT HOMINES
1909