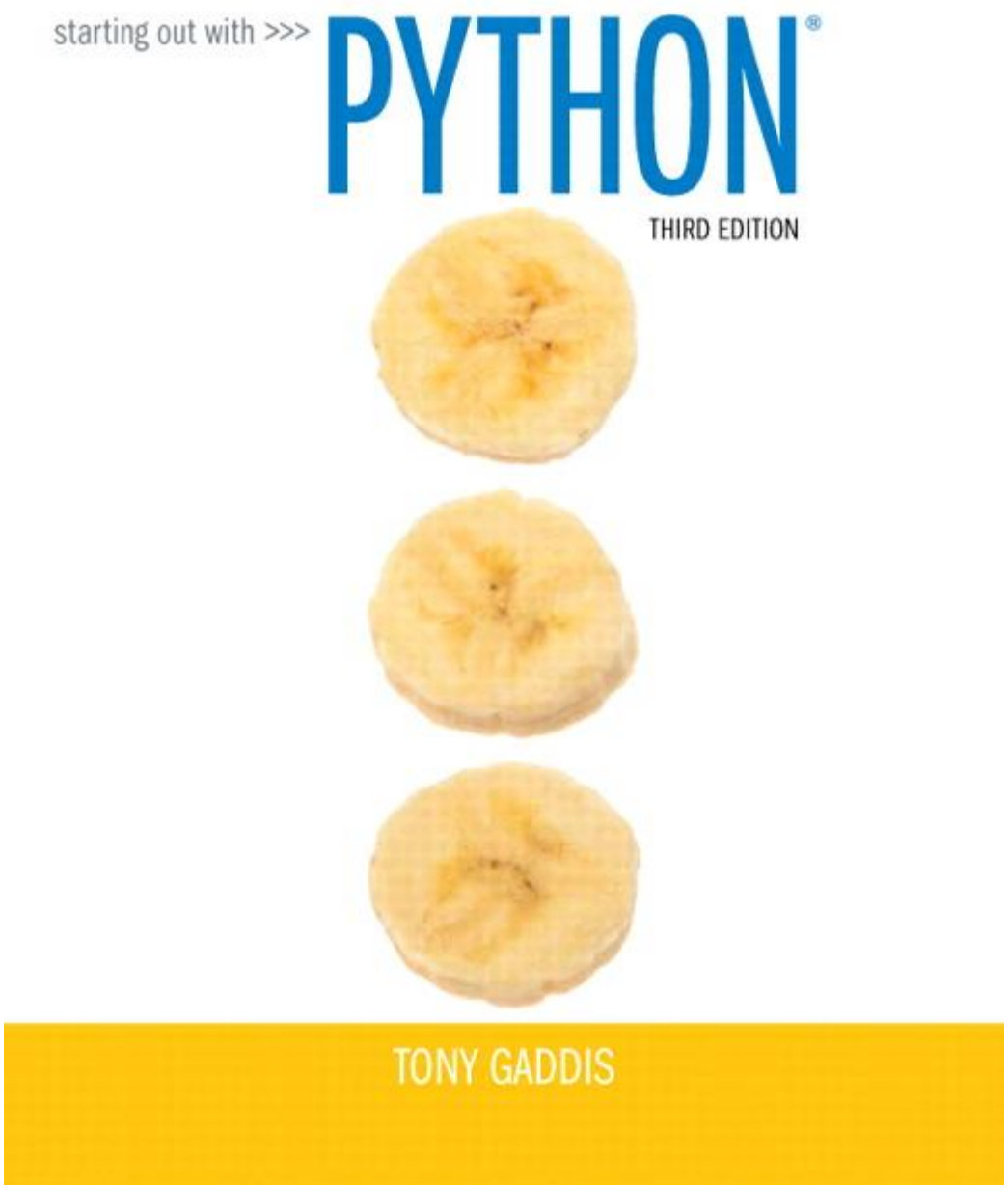
CHAPTER 2

Input, Processing, and Output



Addison-Wesley
is an imprint of
PEARSON

Designing a Program

- **Programs must be designed before they are written**
- **Program development cycle:**
  - Design the program
  - Write the code
  - Correct syntax errors
  - Test the program
  - Correct logic errors

Designing a Program (cont'd.)

- **Determine the steps that must be taken to perform the task**
  - Break down required task into a series of steps
  - Create an algorithm, listing logical steps that must be taken
- **Algorithm: set of well-defined logical steps that must be taken to perform a task**

Input, Processing, and Output

- **Typically, computer performs three-step process**
  - Receive input
    - Input: any data that the program receives while it is running
  - Perform some process on the input
    - Example: mathematical calculation
  - Produce output

Displaying Output with the `print` Function

- **Function: piece of prewritten code that performs an operation**
- **`print` function: displays output on the screen**
- **Argument: data given to a function**
  - Example: data that is printed to screen
  - print("Hello world!") → Hello world is an argument
- **Statements in a program execute in the order that they appear**
  - From top to bottom

Strings and String Literals

- **String: sequence of characters that is used as data**
- **String literal: string that appears in actual code of a program**
  - Must be enclosed in single (‘) or double (“) quote marks
  - String literal **can be enclosed** in triple quotes (''' or """)
    - Enclosed string can contain both single and double quotes and can have multiple lines
    - print (“””one

      two

      three”””)

Comments

- **Comments: notes of explanation within a program**
  - Ignored by Python interpreter
    - Intended for a person reading the program's code
  - Begin with a Pound sign #
- **End-line comment: appears at the end of a line of code**
  - Typically explains the purpose of that line
  - print("David") #Display the name

Variables

- **Variable: name that represents a value stored in the computer memory**
  - Used to access and manipulate data stored in memory
  - A variable references the value it represents
- **Assignment statement: used to create a variable and make it reference data**
  - General format is `variable = expression`
    - Example: `age = 29`
    - Assignment operator: the equal sign (=)

Variables (cont'd.)

- **In assignment statement, variable receiving value must be on left side**
- **A variable can be passed as an argument to a function**
  - Variable name should not be enclosed in quote marks
- **You can only use a variable after a value is assigned to it**

Variable Naming Rules

- **Rules for naming variables in Python:**
  - Variable name cannot be a Python key word
  - Variable name cannot contain spaces
  - First character must be a letter or an underscore
  - After first character may use letters, digits, or underscores
  - Variable names are case sensitive
- **Variable name should reflect its use**
- **My habit: all small characters with under lines → has_found, test_for_loop**

Displaying Multiple Items with the `print` Function

- **Python allows one to display multiple items with a single call to `print`**
  - Items are separated by commas when passed as arguments
  - Arguments displayed in the order they are passed to the function
  - Items are automatically separated by a space when displayed on screen

```
# This program demonstrates a variable.
room = 503
building = “science”

print(room, building)

print('I am staying in room number {}’.format(room))

print('I am staying in room number {} at building named {}’.
                                        format(room, building))
```

Use the parentheses to include variable

Variable Reassignment

- **Variables can reference different values while program is running**
- **Garbage collection: removal of values that are no longer referenced by variables**
  - Carried out by Python interpreter
- **A variable can refer to item of any type**
  - Variable that has been assigned to one type can be reassigned to another type

Numeric Data Types, Literals, and the `str` Data Type

- **Data types: categorize value in memory**
  - e.g., int for integer, float for real number, str used for storing strings in memory
- **Numeric literal: number written in a program**
  - No decimal point considered int, otherwise, considered float
- **Some operations behave differently depending on data type**

```
# This program demonstrates variable reassignment.
# Assign a value to the dollars variable.
dollars = 2.75
print('I have', dollars, 'in my account.')

# Reassign dollars so it references
# a different value.
dollars = 99.95
print('But now I have', dollars, 'in my account!')
```

By python command line

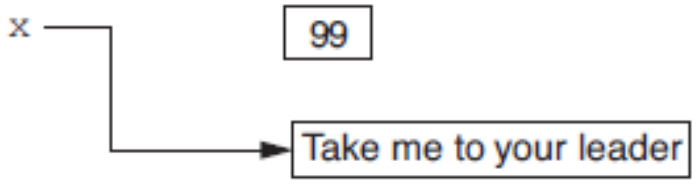
```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
```

Reassigning a Variable to a Different Type

- A variable in Python can refer to items of any type

Figure 2-7 The variable x references an integer

Figure 2-8 The variable x references a string



Reading Input from the Keyboard

- **Most programs need to read input from the user**
- **Built-in `input` function reads input from keyboard**
  - Returns the data as a string
  - Format: *`variable`* `= input(`*`prompt`*`)`
    - `prompt` is typically a string instructing user to enter a value
  - Does not automatically display a space after the prompt

```
# Get the user's first name.
first_name = input('Enter your first name: ')

# Get the user's last name.
last_name = input('Enter your last name: ')

# Print a greeting to the user.
print('Hello {} {}’.format(first_name, last_name))
```

Reading Numbers with the `input` Function

- **`input` function always returns a string**
- **Built-in functions convert between data types**
  - `int(`*`item`*`)` converts *`item`* to an `int`
  - `float(`*`item`*`)` converts *`item`* to a `float`
  - Nested function call: general format:
    `function1``(`*`function2`*`(`*`argument`*`))`
    - value returned by function2 is passed to function1
  - Type conversion only works if item is valid numeric value, otherwise, throws exception

```
# Get the user's name, age, and income.
name = input('What is your name? ')
age = int(input('What is your age? '))
income = float(input('What is your income? '))
```

How about you input a non-integer value?

Performing Calculations

- **Math expression: performs calculation and gives a value**
  - Math operator: tool for performing calculation
  - Operands: values surrounding operator
    - Variables can be used as operands
  - Resulting value typically assigned to variable
- **Two types of division:**
  - / operator performs floating point division
  - / / operator performs integer division
    - Positive results truncated, negative rounded away from zero → -2.25 → -3
    - **14//3 = 4, -14//3 = -5, int(-14/3) = -4**

```
# This program gets an item's original price and
# calculates its sale price, with a 20% discount.

# Get the item's original price.
original_price = float(input("Enter the item's original price: "))

# Calculate the amount of the discount.
discount = original_price * 0.2

# Calculate the sale price.
sale_price = original_price - discount

# Display the sale price.
print('The sale price is {}’.format(sale_price))
```

Operator Precedence and Grouping with Parentheses

- **Python operator precedence:**
  1. Operations enclosed in parentheses
     - Forces operations to be performed before others
  2. Exponentiation (**)
  3. Multiplication (*), division (/ and //), and remainder (%)
  4. Addition (+) and subtraction (-)
- **Higher precedence performed first**
  - Same precedence operators execute from left to right

The Exponent Operator and the Remainder Operator

- **Exponent operator (`**`): Raises a number to a power**
  - `x ** y` = x^y
- **Remainder operator (`%`): Performs division and returns the remainder**
  - a.k.a. modulus operator
  - e.g., `4%2=0, 5%2=1`
  - Typically used to convert times and distances, and to detect odd or even numbers

```
# Get a number of seconds from the user.
total_seconds = float(input('Enter a number of seconds: '))

# Get the number of hours.
hours = total_seconds // 3600

# Get the number of remaining minutes.
minutes = (total_seconds // 60) % 60

# Get the number of remaining seconds.
seconds = total_seconds % 60

# Display the results.
print('Here is the time in hours, minutes, and seconds:')
print('Hours: {}'.format(hours))
print('Minutes: {}'.format(minutes))
print('Seconds: {}'.format(seconds))
```

Converting Math Formulas to Programming Statements

- **Operator required for any mathematical operation**
- **When converting mathematical expression to programming statement:**
  - May need to add multiplication operators
  - May need to insert parentheses

$$P = \frac{F}{(1+r)^n}$$

```
# Get the desired future value.
future_value = float(input('Enter the desired future value: '))

# Get the annual interest rate.
rate = float(input('Enter the annual interest rate: '))

# Get the number of years that the money will appreciate.
years = int(input('Enter the number of years the money will grow: '))

# Calculate the amount needed to deposit.
present_value = future_value / (1.0 + rate)**years

# Display the amount needed to deposit.
print('You will need to deposit this amount: {}'.format(present_value))
```

Mixed-Type Expressions and Data Type Conversion

- **Data type resulting from math operation depends on data types of operands**
  - Two `int` values: result is an `int`
  - Two `float` values: result is a `float`
  - `int` and `float`: `int` temporarily converted to `float`, result of the operation is a `float`
    - Mixed-type expression
  - Type conversion of `float` to `int` causes truncation of fractional part

Breaking Long Statements into Multiple Lines

- **Long statements cannot be viewed on screen without scrolling and cannot be printed without cutting off**
- **Multiline continuation character (\): Allows to break a statement into multiple lines**
  - Example:

```
print('my first name is',\
first_name)
```

More About Data Output

- **`print` function displays line of output**
  - Newline character at end of printed data
  - Special argument `end='`*`delimiter`*`'` causes `print` to place *`delimiter`* at end of data instead of newline character
  - example:

    print("one", end=' ')

    print("two", end=' ')

    → result will be one two

More About Data Output

- **`print` function uses space as item separator**
  - Special argument `sep='`*`delimiter`*`'` causes `print` to use *`delimiter`* as item separator
  - example

    print("one", "two", "three", sep='*')

    → one*two*three

More About Data Output (cont'd.)

- **Special characters appearing in string literal**
  - Preceded by backslash (\)
    - Examples: newline (\n), horizontal tab (\t)
  - Treated as commands embedded in string
  - Example

    Print('One\nTwo')

    → One

    Two

More About Data Output (cont'd.)

- **When + operator used on two strings in performs string concatenation**
  - Useful for breaking up a long string literal

Formatting Numbers

- **Can format display of numbers on screen using built-in `format` function**
  - Two arguments:
    - Numeric value to be formatted
    - Format specifier
  - Returns string containing formatted number
  - Format specifier typically includes precision and data type
    - Can be used to indicate scientific notation, comma separators, and the minimum field width used to display the value

```
# This program demonstrates how a floating-point
# number can be formatted.
amount_due = 5000.0
monthly_payment = amount_due / 12
print('The monthly payment is', \
      format(monthly_payment, '.2f'))
```

```
# This program displays the following
# floating-point numbers in a column
# with their decimal points aligned.
num1 = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.821
num5 = 88.081
num6 = 799.999

# Display each number in a field of 7 spaces
# with 2 decimal places.
print(format(num1, '7.2f'))
print(format(num2, '7.2f'))
print(format(num3, '7.2f'))
print(format(num4, '7.2f'))
print(format(num5, '7.2f'))
print(format(num6, '7.2f'))
```

Formatting Numbers (cont'd.)

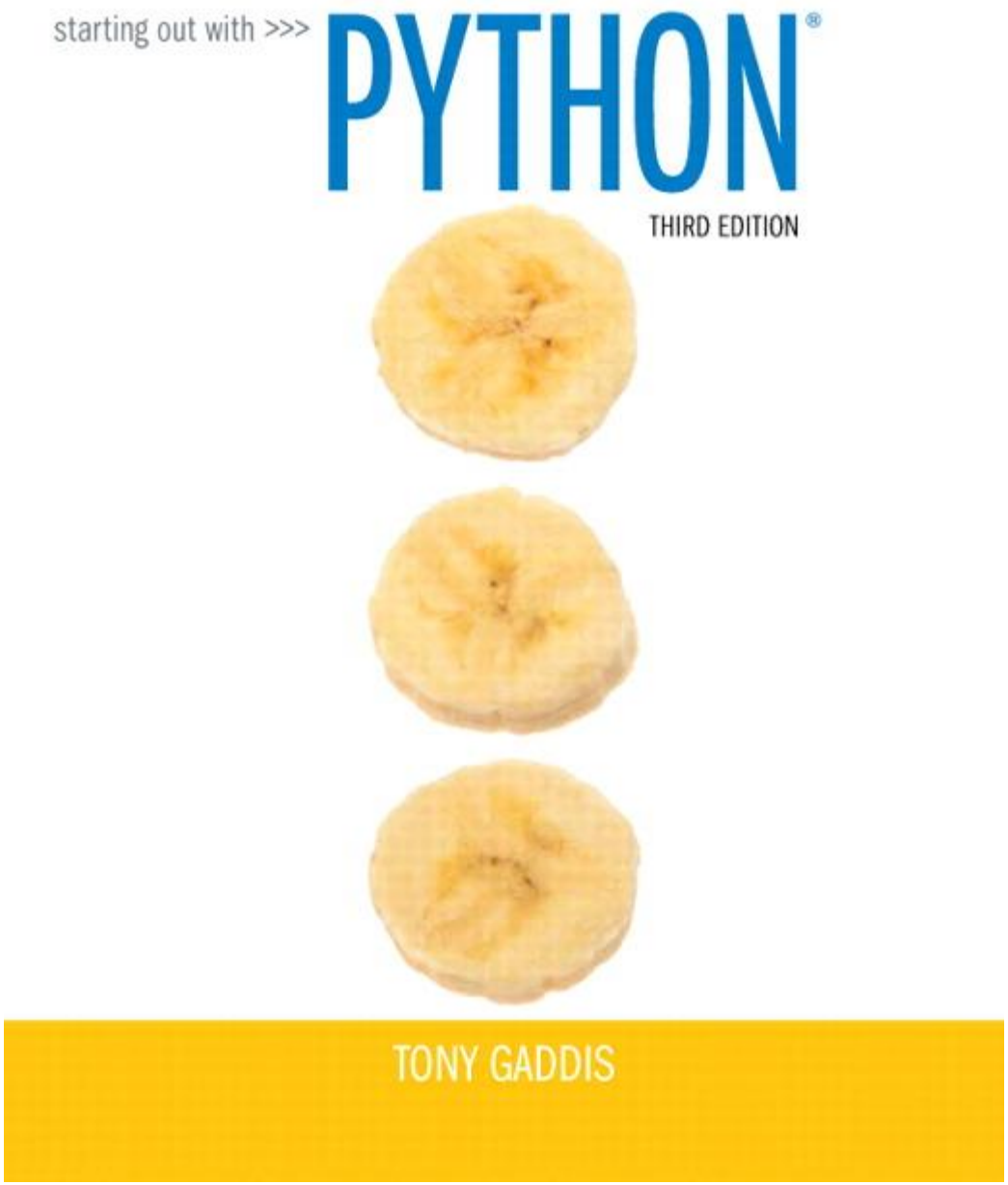
- **The `%` symbol can be used in the format string of `format` function to format number as percentage**
  - print(format(0.5, '%')) → 50.000000%
  - print(format(0.5, '.0%')) → 50%

Formatting Numbers (cont'd.)

- **To format an integer using `format` function:**
  - Use `d` as the type designator
  - Do not specify precision
  - Can still use `format` function to set field width or comma separator
  - print(format(123456, ',d')) → 123,456

CHAPTER 3

Decision structures and Boolean logic



Addison-Wesley
is an imprint of
PEARSON

The `if` Statement

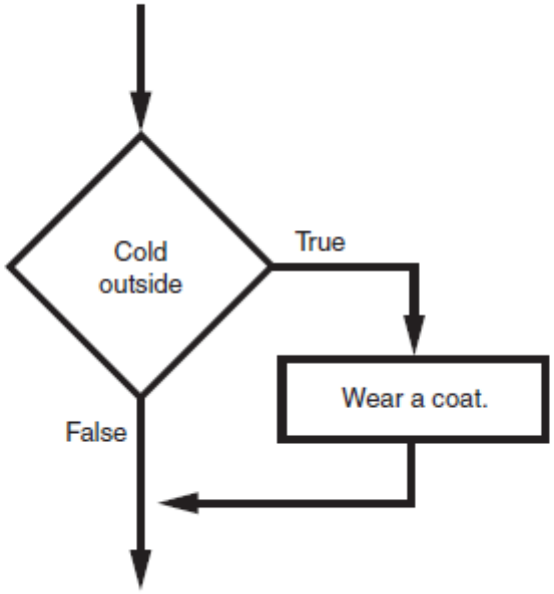
- **Control structure: logical design that controls order in which set of statements execute**
- **Sequence structure: set of statements that execute in the order they appear**
- **Decision structure: specific action(s) performed only if a condition exists**
  - Also known as selection structure

The `if` Statement (cont'd.)

- **In flowchart, diamond represents true/false condition that must be tested**
- **Actions can be *conditionally executed***
  - Performed only when a condition is true
- **Single alternative decision structure: provides only one alternative path of execution**
  - If condition is not true, exit the structure

The `if` Statement (cont'd.)

Figure 3-1 A simple decision structure



The `if` Statement (cont'd.)

- **Python syntax:**

```
if condition:
    Statement  ← You have to indent
    Statement  ← indent should be identical
```

- **First line known as the `if` clause**
  - Includes the keyword `if` followed by condition
    - The condition can be true or false
    - When the `if` statement executes, the condition is tested, and if it is true the block statements are executed. otherwise, block statements are skipped

Boolean Expressions and Relational Operators

- **Boolean expression: expression tested by if statement to determine if it is true or false**
  - Example: a > b
    - `true` if a is greater than b; `false` otherwise
- **Relational operator: determines whether a specific relationship exists between two values**
  - Example: greater than (>)

Boolean Expressions and Relational Operators (cont'd.)

- **>= and <= operators test more than one relationship**
  - It is enough for one of the relationships to exist for the expression to be true
- **== operator determines whether the two operands are equal to one another**
  - Do not confuse with assignment operator (=)
- **!= operator determines whether the two operands are not equal**

Boolean Expressions and Relational Operators (cont'd.)

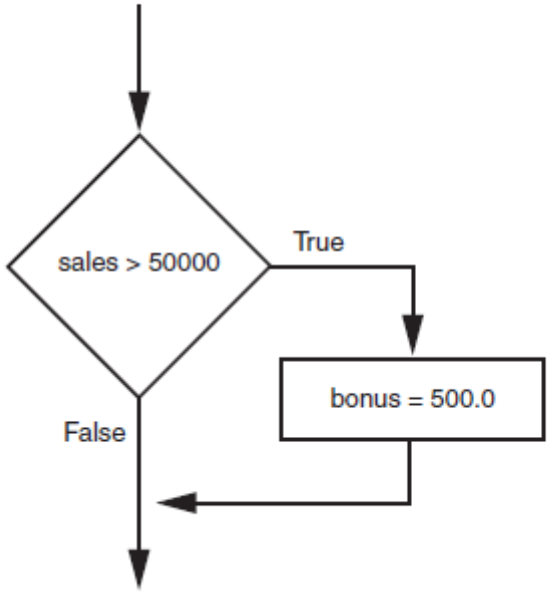
Table 3-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

Boolean Expressions and Relational Operators (cont'd.)

- **Using a Boolean expression with the > relational operator**

Figure 3-3 Example decision structure



Boolean Expressions and Relational Operators (cont'd.)

- **Any relational operator can be used in a decision block**
  - Example: `if balance == 0`
  - Example: `if payment != balance`
- **It is possible to have a block inside another block**
  - Example: `if` statement inside a function
  - Statements in inner block must be indented with respect to the outer block

```
# This program gets three test scores and displays
# their average.  It congratulates the user if the
# average is a high score.

# The high score variable holds the value that is
# considered a high score.
high_score = 95

# Get the three test scores.
test1 = int(input('Enter the score for test 1: '))
test2 = int(input('Enter the score for test 2: '))
test3 = int(input('Enter the score for test 3: '))

# Calculate the average test score.
average = (test1 + test2 + test3) / 3

# Print the average.
print('The average score is', average)

# If the average is a high score,
# congratulate the user.
if average >= high_score:
    print('Congratulations!')
    print('That is a great average!')
```

The `if-else` Statement

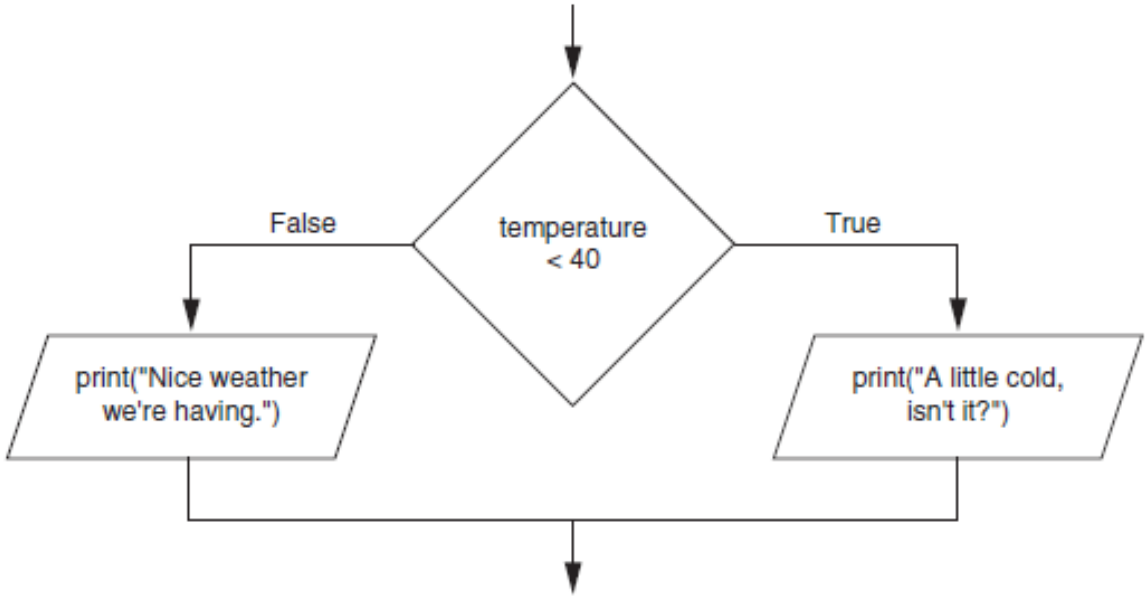
- **Dual alternative decision structure: two possible paths of execution**
  - One is taken if the condition is true, and the other if the condition is false
  - Syntax:

```
if condition:
        statements
else:
        other statements
```

  - `if` clause and `else` clause must be aligned
  - Statements must be consistently indented

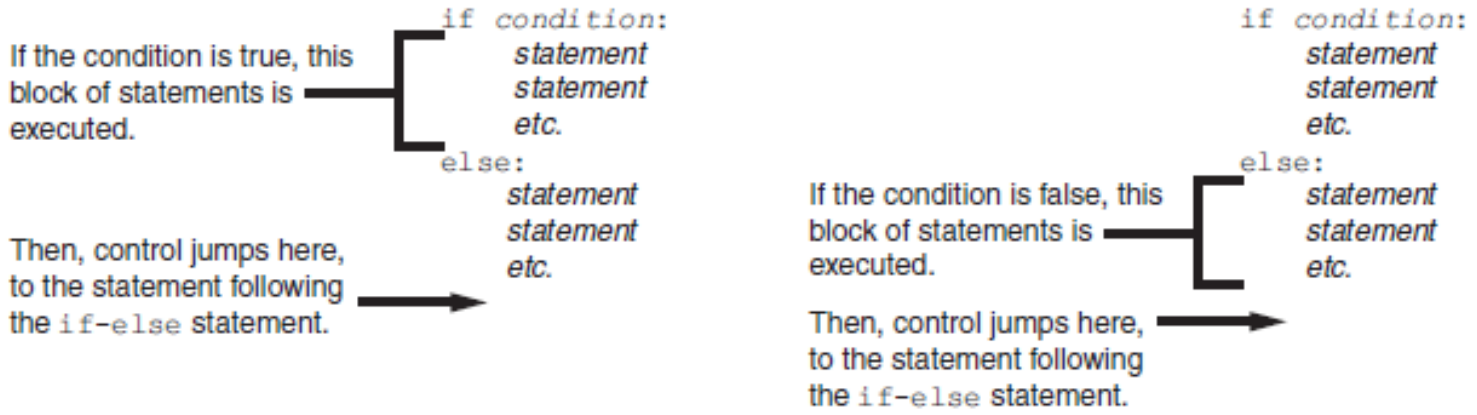
The `if-else` Statement (cont'd.)

Figure 3-5 A dual alternative decision structure



The `if-else` Statement (cont'd.)

Figure 3-6 Conditional execution in an `if-else` statement



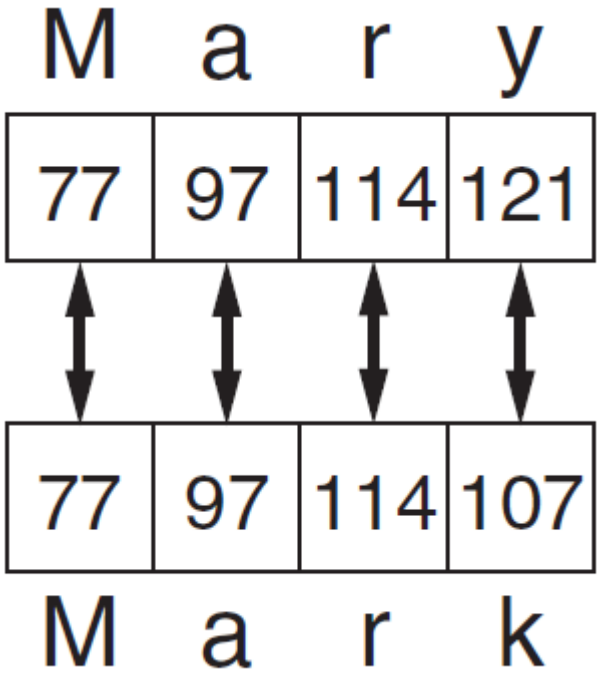
```
# Variables to represent the base hours and
# the overtime multiplier.
base_hours = 40       # Base hours per week
ot_multiplier = 1.5   # Overtime multiplier

# Get the hours worked and the hourly pay rate.
hours = float(input('Enter the number of hours worked: '))
pay_rate = float(input('Enter the hourly pay rate: '))

# Calculate and display the gross pay.
if hours > base_hours:
    # Calculate the gross pay with overtime.
    # First, get the number of overtime hours worked.
    overtime_hours = hours - base_hours

    # Calculate the amount of overtime pay.
    overtime_pay = overtime_hours * pay_rate * ot_multiplier

    # Calculate the gross pay.
    gross_pay = base_hours * pay_rate + overtime_pay
else:
    # Calculate the gross pay without overtime.
    gross_pay = hours * pay_rate

# Display the gross pay.
print('The gross pay is {}'.format(gross_pay))
```

Comparing Strings

- **Strings can be compared using the == and != operators**
- **String comparisons are case sensitive**
- **Strings can be compared using >, <, >=, and <=**
  - Compared character by character based on the ASCII values for each character
  - If shorter word is substring of longer word, longer word is greater than shorter word

Comparing Strings (cont'd.)

Figure 3-9 Comparing each character in a string



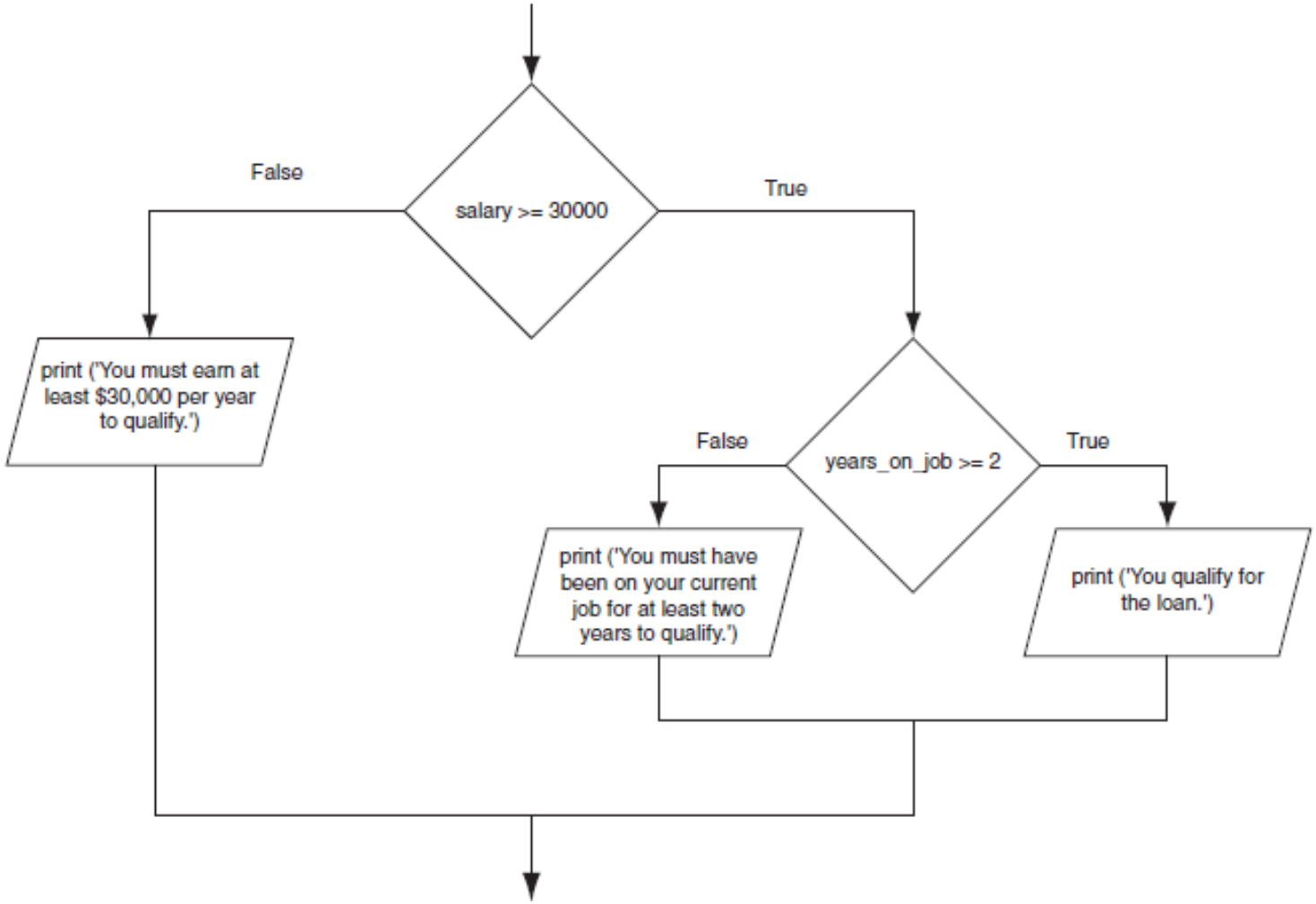
→ Mary > Mark

→ ord('M') = 77
→ chr(77) = 'M'

Nested Decision Structures and the `if-elif-else` Statement

- **A decision structure can be nested inside another decision structure**
  - Commonly needed in programs
  - Example:
    - Determine if someone qualifies for a loan, they must meet two conditions:
      - Must earn at least $30,000/year
      - Must have been employed for at least two years
    - Check first condition, and if it is true, check second condition

Figure 3-12 A nested decision structure



Nested Decision Structures and the `if-elif-else` Statement (cont'd.)

- **Important to use proper indentation in a nested decision structure**
  - Important for Python interpreter
  - Makes code more readable for programmer
  - Rules for writing nested if statements:
    - `else` clause should align with matching `if` clause
    - Statements in each block must be consistently indented

The `if-elif-else` Statement

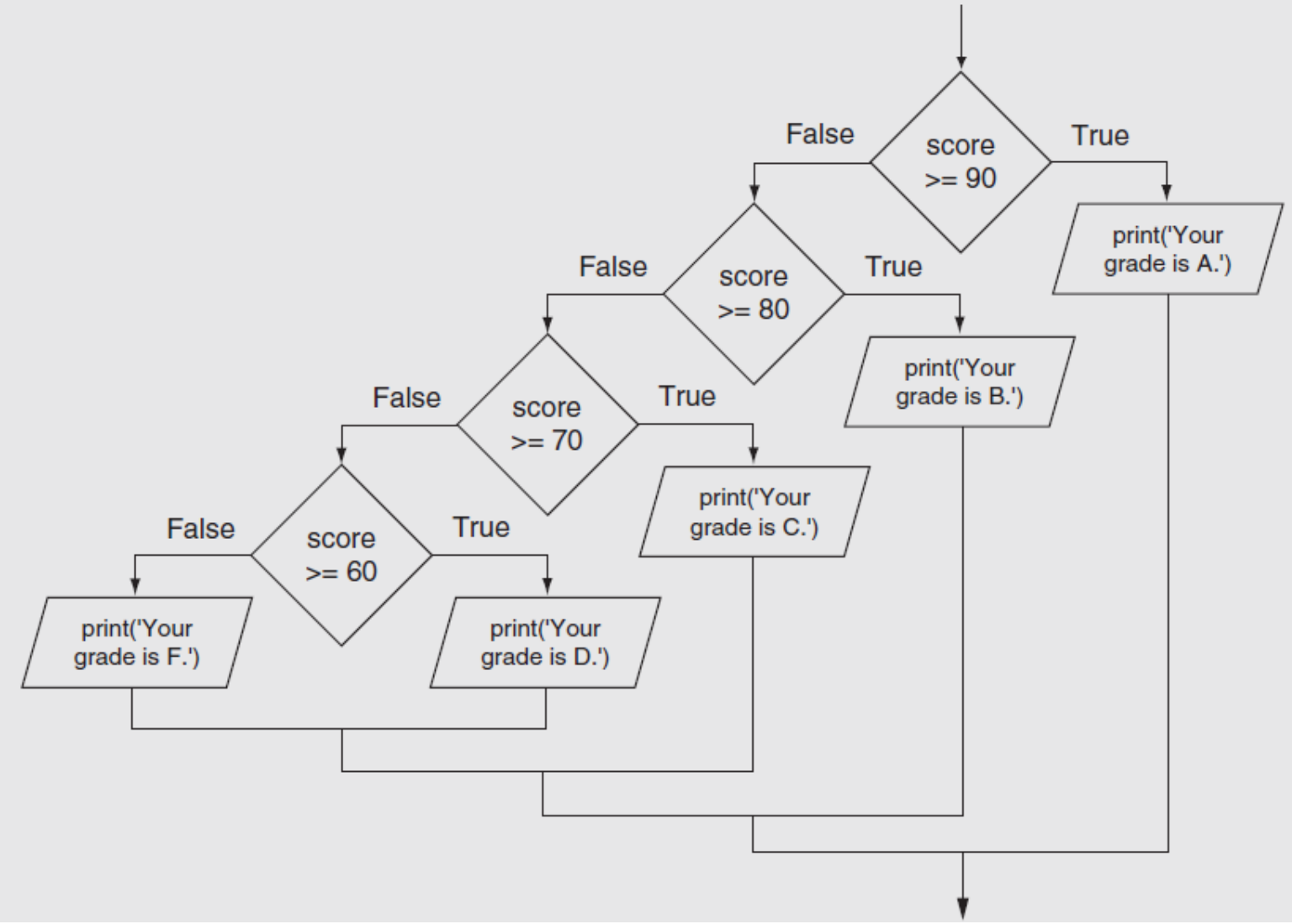
- **`if-elif-else` statement: special version of a decision structure**
  - Makes logic of nested decision structures simpler to write
    - Can include multiple `elif` statements
  - Syntax:

```
if condition1
      statements
elif condition2
      statements
else
      statements
```

The `if-elif-else` Statement (cont'd.)

- **Alignment used with `if-elif-else` statement:**
  - `if`, `elif`, and `else` clauses are all aligned
  - Conditionally executed blocks are consistently indented
- **`if-elif-else` statement is never required, but logic easier to follow**
  - Can be accomplished by nested `if-else`
    - Code can become complex, and indentation can cause problematic long lines

Figure 3-15 Nested decision structure to determine a grade



Logical Operators

- **Logical operators: operators that can be used to create complex Boolean expressions**
  - **`and` operator** and **`or` operator**: binary operators, connect two Boolean expressions into a compound Boolean expression
  - **`not` operator**: unary operator, reverses the truth of its Boolean operand

The `and` Operator

- **Takes two Boolean expressions as operands**
  - Creates compound Boolean expression that is true only when both sub expressions are true
  - Can be used to simplify nested decision structures
- **Truth table for the `and` operator**

Expression	Value of the Expression
false and false	false
false and true	false
true and false	false
true and true	true

The `or` Operator

- **Takes two Boolean expressions as operands**
  - Creates compound Boolean expression that is true when either of the sub expressions is true
  - Can be used to simplify nested decision structures
- **Truth table for the `or` operator**

Expression	Value of the Expression
false and false	false
false and true	true
true and false	true
true and true	true

Short-Circuit Evaluation

- **Short circuit evaluation: deciding the value of a compound Boolean expression after evaluating only one sub expression**
  - Performed by the `or` and `and` operators
    - For `or` operator: If left operand is true, compound expression is true. Otherwise, evaluate right operand
    - For `and` operator: If left operand is false, compound expression is false. Otherwise, evaluate right operand

The `not` Operator

- **Takes one Boolean expressions as operand and reverses its logical value**
  - Sometimes it may be necessary to place parentheses around an expression to clarify to what you are applying the not operator
- **Truth table for the `not` operator**

Expression	Value of the Expression
true	false
false	true

Checking Numeric Ranges with Logical Operators

- **To determine whether a numeric value is within a specific range of values, use `and`**
  - Example: `x >= 10 and x <= 20`
  - → **you can do this 10<=x<=20, but I do not like it**
- **To determine whether a numeric value is outside of a specific range of values, use `or`**
  - Example: `x < 10 or x > 20`

Boolean Variables

- **Boolean variable: references one of two values, `True` or `False`**
  - Represented by `bool` data type
  - For example, **continue_flag = True**
- **Commonly used as flags**
  - Flag: variable that signals when some condition exists in a program
    - Flag set to `False` → condition does not exist
    - Flag set to `True` → condition exists

```
# This program determines whether a bank customer
# qualifies for a loan.

min_salary = 30000.0  # The minimum annual salary
min_years = 2         # The minimum years on the job

# Get the customer's annual salary.
salary = float(input('Enter your annual salary: '))

# Get the number of years on the current job.
years_on_job = int(input('Enter the number of ' +
                         'years employed: '))

# Determine whether the customer qualifies.
if salary >= min_salary or years_on_job >= min_years:
    print('You qualify for the loan.')
else:
    print('You do not qualify for this loan.')
```

None Type

- **None type: as nothing or NULL**
  - For example, **container = None**
  - Decide the usage later

Miscellaneous

- The use of “is”
  - if continuous_flag is True:
  - if continuous_flag is not True:
  - if container is None:

a=“aa”
b=“aa”
a is b = ?

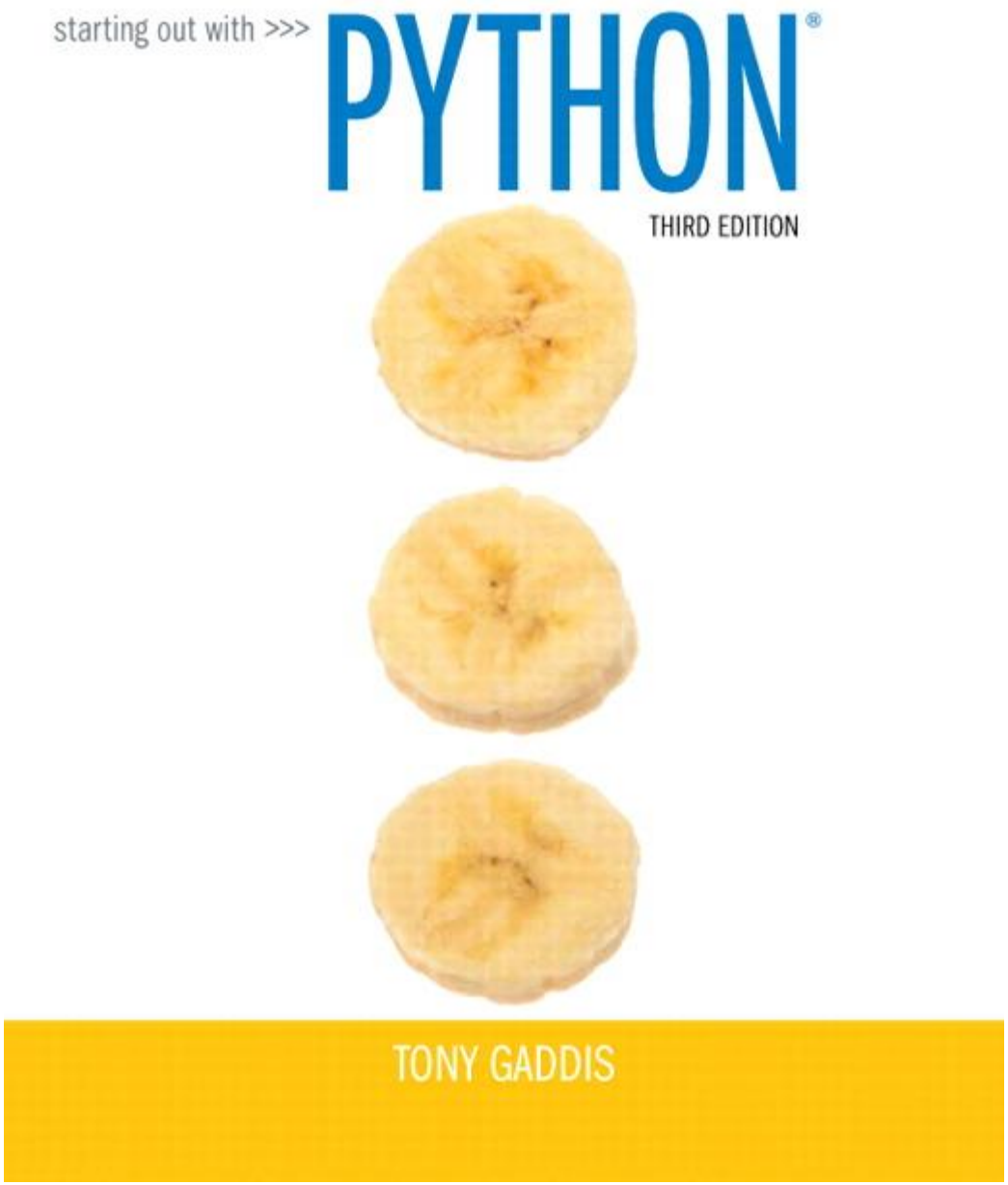
a=1
b=1
a is b = ?

a = 1
a is 1 =?

- Apply the “is” in judgement only → use to judge Boolean or None
- The “is” is used to judge if two variable are located in the same memory space

CHAPTER 4

Repetition structures



Addison-Wesley
is an imprint of
PEARSON

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

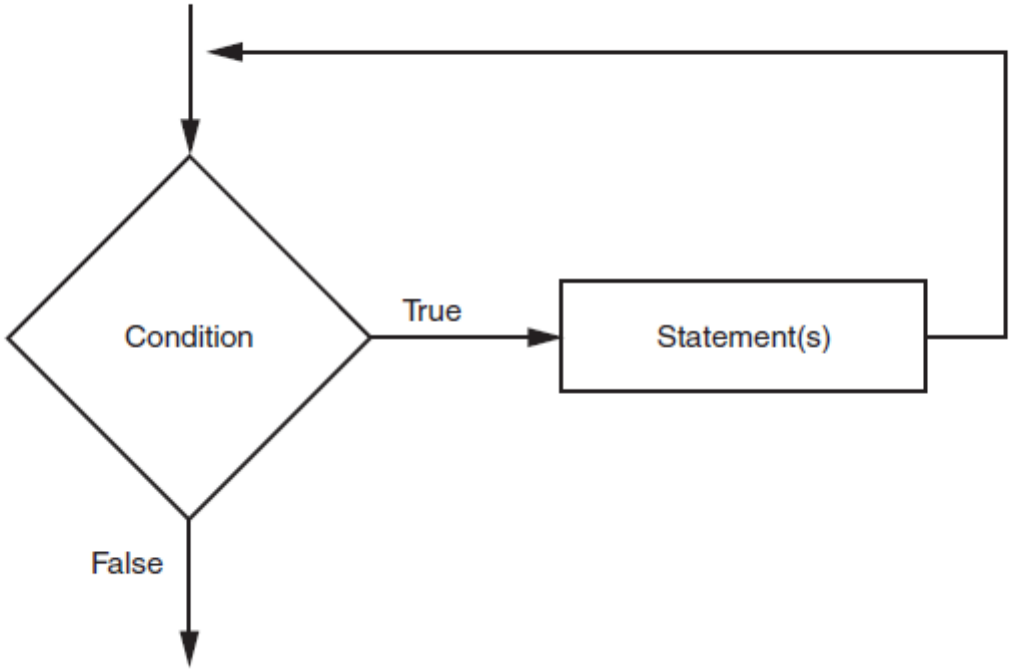
The `while` Loop: a Condition-Controlled Loop

- **`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:
    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

```
# 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): ')
```

```
# Create a variable to represent the maximum
# temperature.
max_temp = 102.5

# Get the substance's temperature.
temperature = float(input("Enter the substance's Celsius
temperature: "))

# As long as necessary, instruct the user to
# adjust the thermostat.
while temperature > max_temp:
    print('The temperature is too high.')
    print('Turn the thermostat down and wait')
    print('5 minutes. Then take the temperature')
    print('again and enter it.')
    temperature = float(input('Enter the new Celsius
temperature: '))

# Remind the user to check the temperature again
# in 15 minutes.
print('The temperature is acceptable.')
print('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

```
while True:
    # 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='')
```

The for Loop: a Count-Controlled Loop

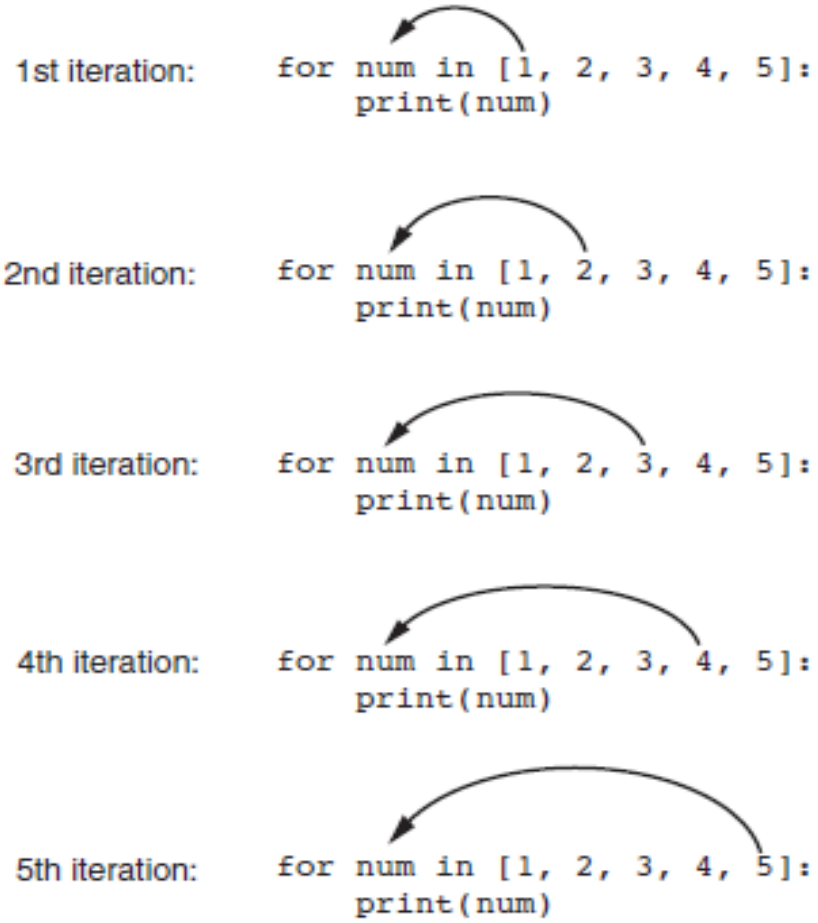
for each

- **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

Figure 4-4 The `for` loop



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 x in range(5):
    print(x)
```

[0, 1, 2, 3, 4]

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

[1, 2, 3, 4]

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

[1, 3, 5, 7, 9]

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 root of each number in a range

```
# 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)
```

```
# This program converts the speeds 60 kph
# through 130 kph (in 10 kph increments)
# to mph.

start_speed = 60              # Starting speed
end_speed = 131               # Ending speed
increment = 10                # Speed increment
conversion_factor = 0.6214    # Conversion factor

# Print the table headings.
print('KPH\tMPH')
print('--------------')

# Print the speeds.
for kph in range(start_speed, end_speed, increment):
    mph = kph * conversion_factor
    print(kph, '\t', format(mph, '.1f'))
```

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

```
# This program uses a loop to display a
# table of numbers and their squares.

# Get the ending limit.
print('This program displays a list of numbers')
print('(starting at 1) and their squares.')
end = int(input('How high should I go? '))

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

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

```
# This program uses a loop to display a
# table of numbers and their squares.

# Get the starting value.
print('This program displays a list of numbers')
print('and their squares.')
start = int(input('Enter the starting number: '))

# Get the ending limit.
end = int(input('How high should I go? '))

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

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

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 (10, 0, -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

```
# This program calculates the sum of a series
# of numbers entered by the user.

max = 5   # The maximum number

# Initialize an accumulator variable.
total = 0.0

# Explain what we are doing.
print('This program calculates the sum of')
print(max, 'numbers you will enter.')

# Get the numbers and accumulate them.
for counter in range(max):
    number = int(input('Enter a number: '))
    total = total + number

# Display the total of the numbers.
print('The total is', total)
```

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**
- **<u>Augmented assignment operators</u>: 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`

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

```
# This program displays property taxes.
TAX_FACTOR = 0.0065   # Represents the tax factor.
# Get the first lot number.
print('Enter the property lot number')
print('or enter 0 to end.')
lot = int(input('Lot number: '))

# Continue processing as long as the user
# does not enter lot number 0.
while lot != 0:
    # Get the property value.
    value = float(input('Enter the property value: '))

    # Calculate the property's tax.
    tax = value * TAX_FACTOR

    # Display the tax.
    print('Property tax: $', format(tax, ',.2f'), sep='')

    # Get the next lot number.
    print('Enter the next lot number or')
    print('enter 0 to end.')
    lot = int(input('Lot number: '))
```

Input Validation Loops

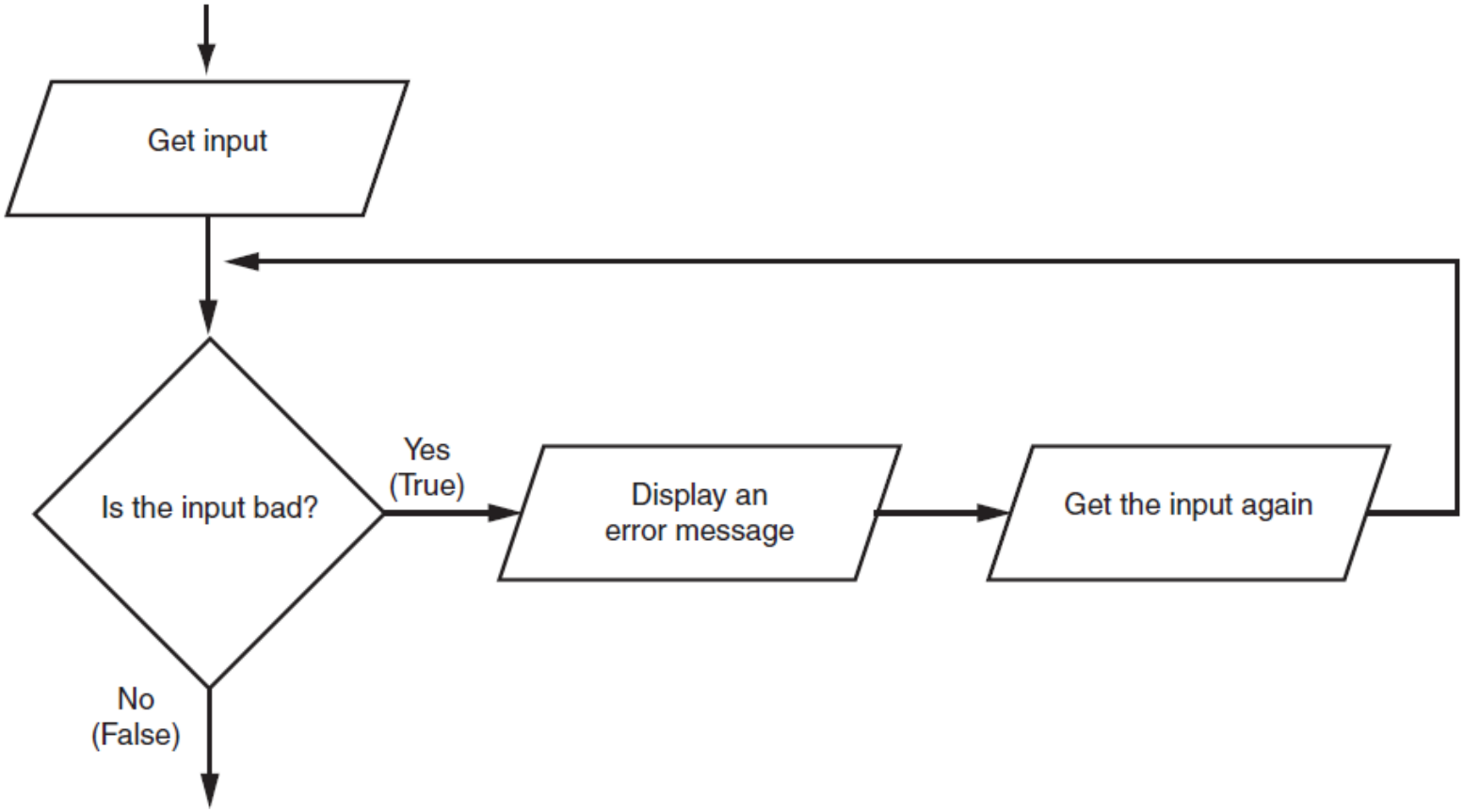
- **Computer cannot tell the difference between good data and bad data**
  - If user provides bad input, program will produce bad output
  - GIGO: garbage in, garbage out
  - It is important to design program such that bad input is never accepted

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.)

Figure 4-7 Logic containing an input validation loop

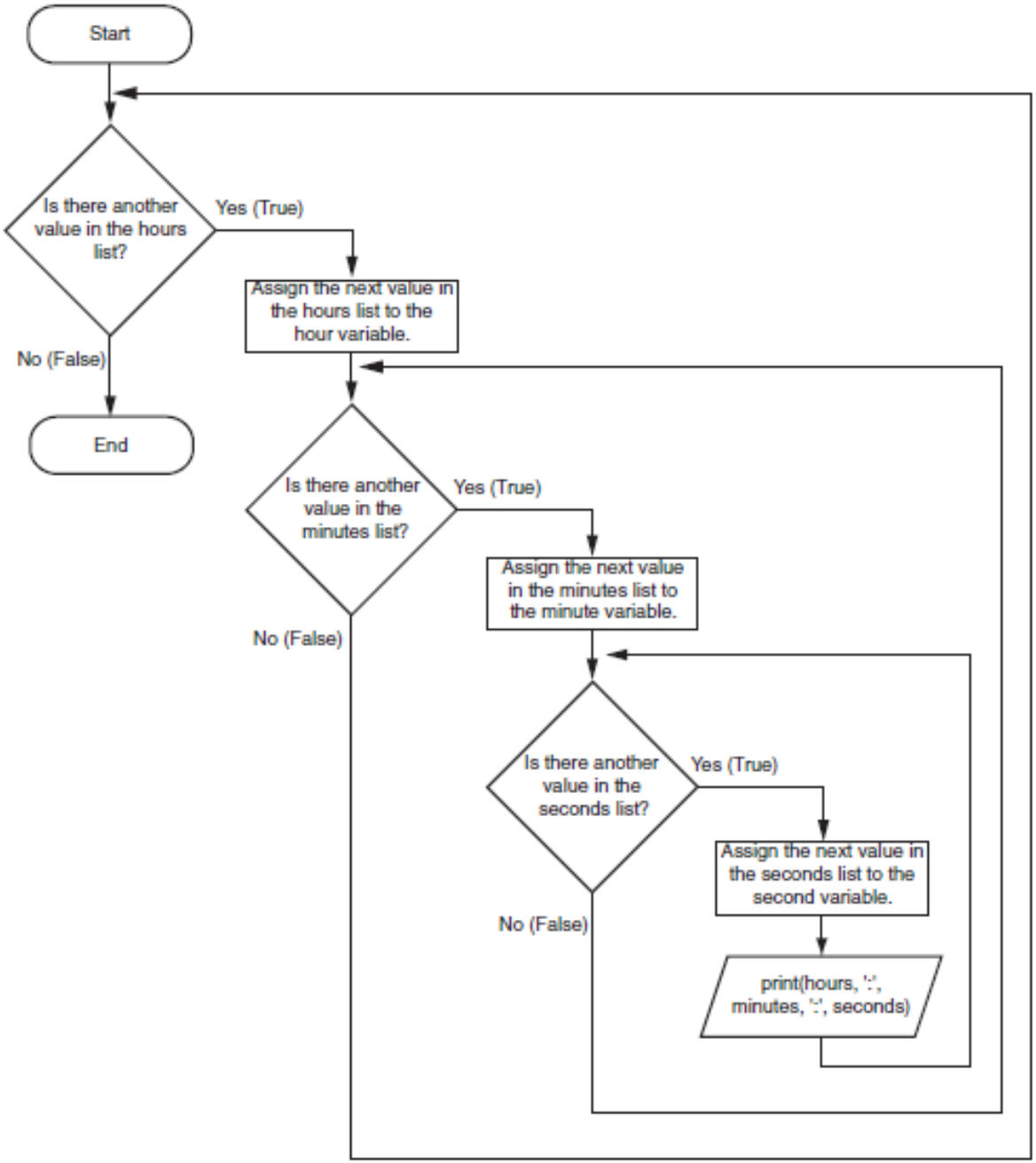


```
# This program calculates retail prices.

mark_up = 2.5  # The markup percentage
another = 'y'  # Variable to control the loop.

# Process one or more items.
while another == 'y' or another == 'Y':
    # Get the item's wholesale cost.
    wholesale = float(input("Enter the item's " + "wholesale cost: "))

    # Validate the wholesale cost.
    while wholesale < 0:
        print('ERROR: the cost cannot be negative.')
        wholesale = float(input('Enter the correct ' + 'wholesale cost:'))

    # Calculate the retail price.
    retail = wholesale * mark_up

    # Display the retail price.
    print('Retail price: $', format(retail, ',.2f'))

    # Do this again?
    another = input('Do you have another item? ' + '(Enter y for yes): ')
```

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”

Figure 4-8 Flowchart for a clock simulator



Nested Loops (cont'd.)

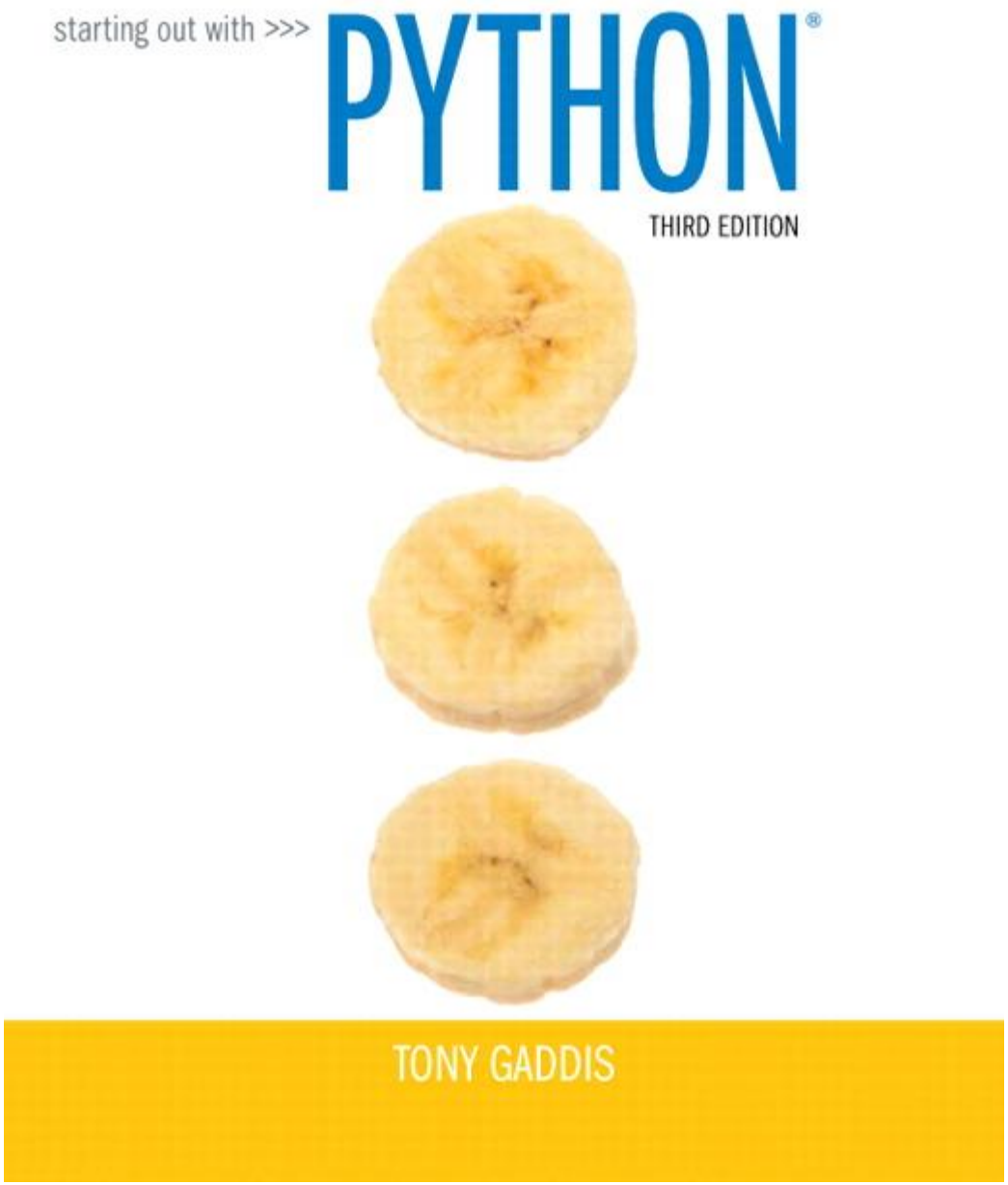
- **Key points about nested loops:**
  - Inner loop goes through all of its iterations for each iteration of outer loop
  - Inner loops complete their iterations faster than outer loops
  - Total number of iterations in nested loop:

```
number_iterations_inner  x
number_iterations_outer
```

```
# Get the number of students.
num_students = int(input('How many students do you have? '))

# Get the number of test scores per student.
num_test_scores = int(input('How many scores per student? '))

# Determine each students average test score.
for student in range(num_students):
    # Initialize an accumulator for test scores.
    total = 0.0
    # Get a student's test scores.
    print('Student number', student + 1)
    print('-----------------')
    for test_num in range(num_test_scores):
        print('Test number', test_num + 1, end='')
        score = float(input(': '))
        # Add the score to the accumulator.
        total += score
    # Calculate the average test score for this student.
    average = total / num_test_scores
    # Display the average.
    print('The average for student number', student + 1, \
          'is:', format(average, '.1f'))
```

CHAPTER 5

Functions

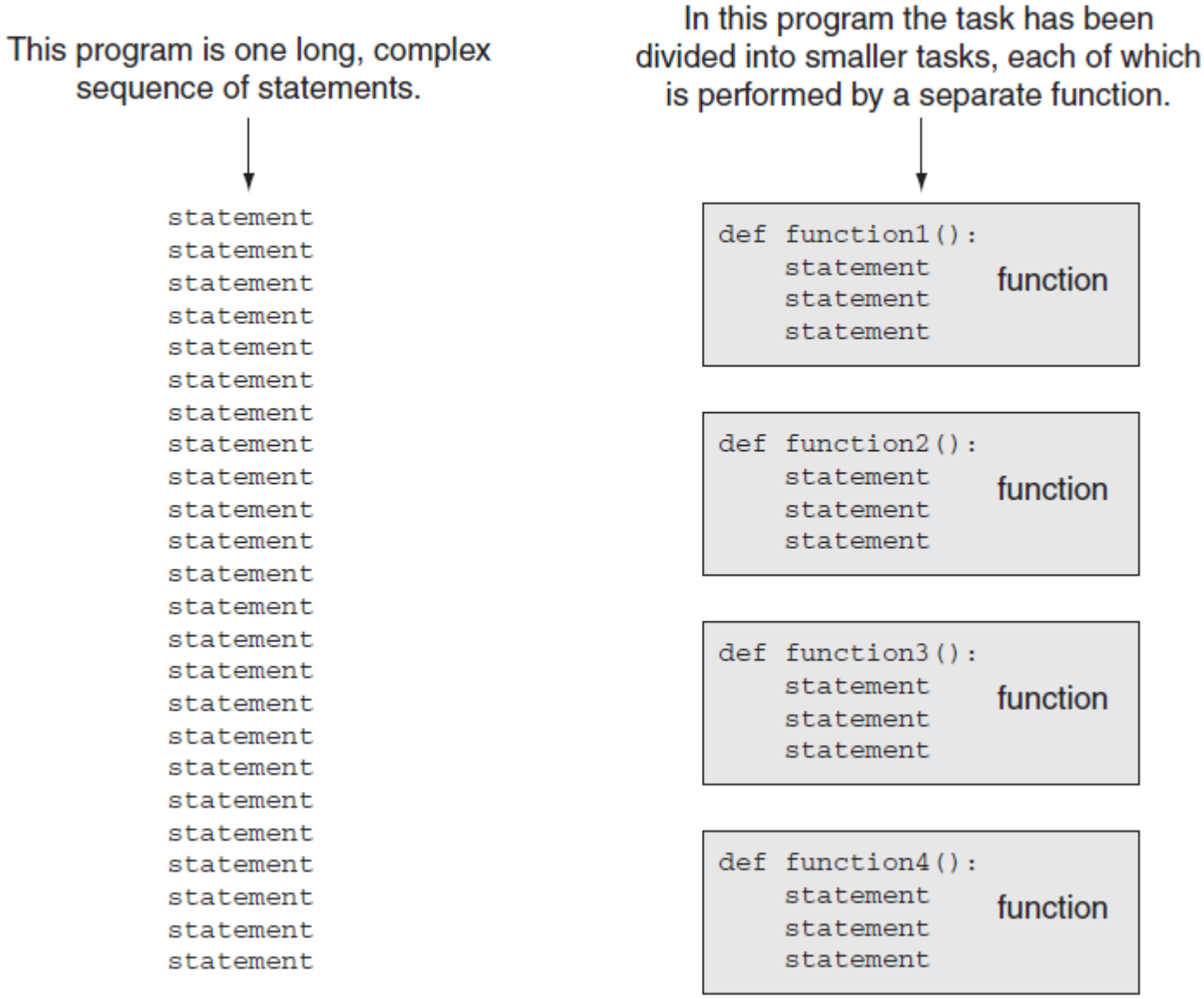


Addison-Wesley
is an imprint of
PEARSON

Introduction to Functions

- **Function: group of statements within a program that perform as specific task**
  - Usually one task of a large program
    - Functions can be executed in order to perform overall program task
  - Known as *divide and conquer* approach
- **Modularized program: program wherein each task within the program is in its own function**

Figure 5-1 Using functions to divide and conquer a large task



Benefits of Modularizing a Program with Functions

- **The benefits of using functions include:**
  - Simpler code
  - Code reuse
    - write the code once and call it multiple times
  - Better testing and debugging
    - Can test and debug each function individually
  - Faster development
  - Easier facilitation of teamwork
    - Different team members can write different functions

Void Functions and Value-Returning Functions

- **A void function:**
  - Simply executes the statements it contains and then terminates.
- **A value-returning function:**
  - Executes the statements it contains, and then it returns a value back to the statement that called it.
    - The `input`, `int`, and `float` functions are examples of value-returning functions.

Defining and Calling a Function

- **Functions are given names**
  - Function naming rules:
    - Cannot use key words as a function name
    - Cannot contain spaces
    - First character must be a letter or underscore
    - All other characters must be a letter, number or underscore
    - Uppercase and lowercase characters are distinct

Defining and Calling a Function (cont'd.)

- **Function name should be descriptive of the task carried out by the function**
  - Often includes a verb
- **Function definition: specifies what function does**

```
def function_name():
        statement
        statement
```

 # function name → small characters with underlines

Defining and Calling a Function (cont'd.)

- **Function header: first line of function**
  - Includes keyword `def` and function name, followed by parentheses and colon
- **Block: set of statements that belong together as a group**
  - Example: the statements included in a function

Defining and Calling a Function (cont'd.)

- **Call a function to execute it**
  - When a function is called:
    - Interpreter jumps to the function and executes statements in the block
    - Interpreter jumps back to part of program that called the function
      - Known as function return

```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I am Arthur')
    print('King of the Britons')

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

Defining and Calling a Function (cont'd.)

- **`main` function: called when the program starts**
  - Calls other functions when they are needed
  - Defines the *mainline logic* of the program

  - Actually, no main function is needed in Python
  - But, you can still have a main function for clarify

```
# This program has two functions. First we
# define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we define the message function.
def message():
    print('I am Arthur')
    print('King of the Britons.')

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

Indentation in Python

- **Each block must be indented**
  - Lines in block must begin with the same number of spaces
    - Use tabs or spaces to indent lines in a block, but not both as this can confuse the Python interpreter
    - IDLE automatically indents the lines in a block
  - Blank lines that appear in a block are ignored

Designing a Program to Use Functions

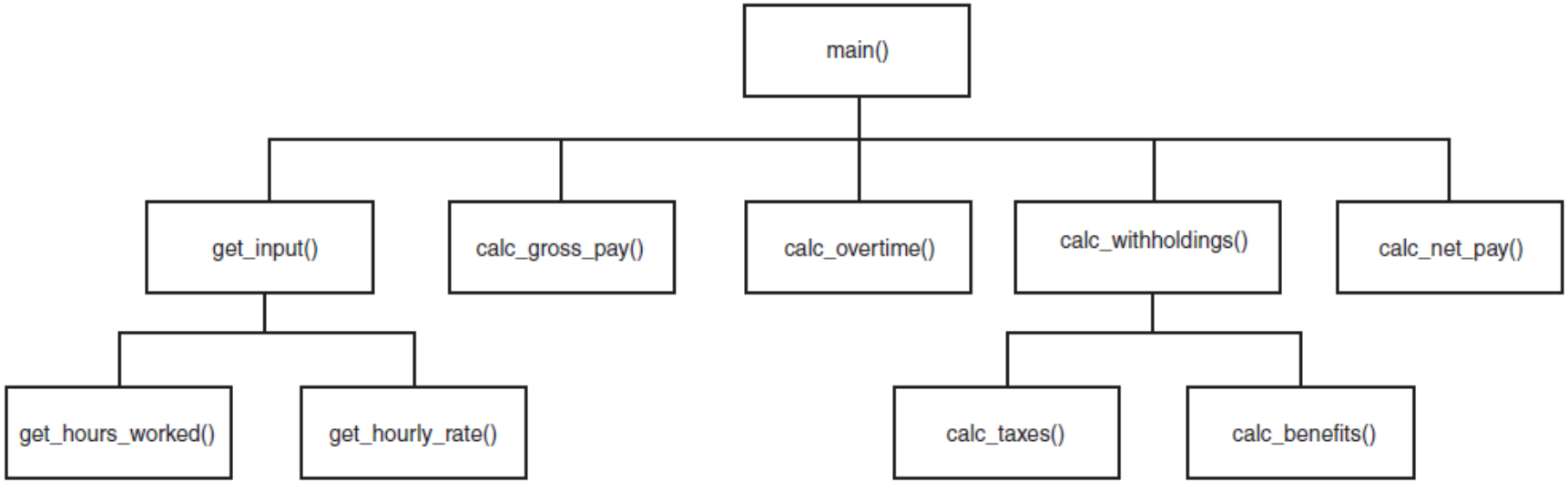
- **In a flowchart, function call shown as rectangle with vertical bars at each side**
  - Function name written in the symbol
  - Typically draw separate flow chart for each function in the program
    - End terminal symbol usually reads `Return`
- **Top-down design: technique for breaking algorithm into functions**

Designing a Program to Use Functions (cont'd.)

- **Hierarchy chart: depicts relationship between functions**
  - AKA structure chart
  - Box for each function in the program, Lines connecting boxes illustrate the functions called by each function
  - Does not show steps taken inside a function

Designing a Program to Use Functions (cont'd.)

Figure 5-10 A hierarchy chart



Local Variables

- **Local variable: variable that is assigned a value inside a function**
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable
- **Scope: the part of a program in which a variable may be accessed**
  - For local variable: function in which created

```
# Definition of the main function.
def main():
    get_name()
    print('Hello', name)     # This causes an error!

# Definition of the get_name function.
def get_name():
    name = input('Enter your name: ')

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

Local Variables (cont'd.)

- **Local variable cannot be accessed by statements inside its function which precede its creation**
- **You should create local variable before using**
- **Different functions may have local variables with the same name**
  - Each function does not see the other function's local variables, so no confusion

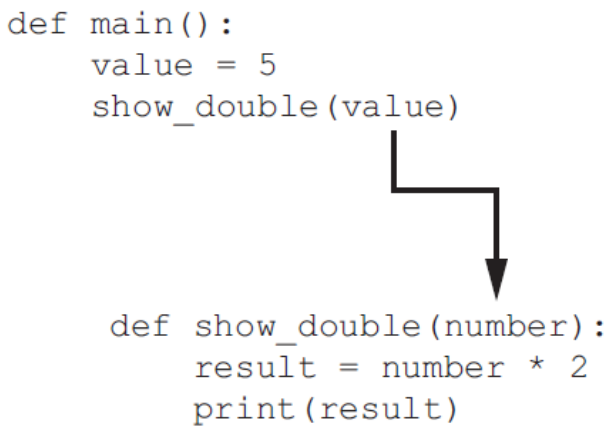
```
# This program demonstrates two functions that
# have local variables with the same name.

def main():
    # Call the texas function.
    texas()
    # Call the california function.
    california()

# Definition of the texas function. It creates
# a local variable named birds.
def texas():
    birds = 5000
    print('texas has', birds, 'birds.')

# Definition of the california function. It also
# creates a local variable named birds.
def california():
    birds = 8000
    print('california has', birds, 'birds.')

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

Passing Arguments to Functions

- **Argument: piece of data that is sent into a function**
  - Function can use argument in calculations
  - When calling the function, the argument is placed in parentheses following the function name

Passing Arguments to Functions (cont'd.)

Figure 5-13 The `value` variable is passed as an argument



Passing Arguments to Functions (cont'd.)

- **Parameter variable: variable that is assigned the value of an argument when the function is called**
  - The parameter and the argument reference the same value
  - General format:

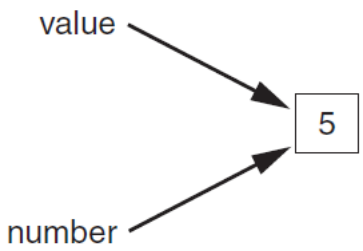
```
def function_name(parameter):
```

  - Scope of a parameter: the function in which the parameter is used

Passing Arguments to Functions (cont'd.)

Figure 5-14 The `value` variable and the `number` parameter reference the same value

```
def main():
    value = 5
    show_double(value)


def show_double(number):
    result = number * 2
    print(result)
```



```
def main():
    # display the intro screen.
    intro()
    # Get the number of cups.
    cups_needed = int(input('Enter the number of cups: '))
    # Convert the cups to ounces.
    cups_to_ounces(cups_needed)

# The intro function displays an introductory screen.
def intro():
    print('This program converts measurements')
    print('in cups to fluid ounces. For your')
    print('reference the formula is:')
    print('    1 cup = 8 fluid ounces')

# The cups_to_ounces function accepts a number of
# cups and displays the equivalent number of ounces.
def cups_to_ounces(cups):
    ounces = cups * 8
    print('That converts to', ounces, 'ounces.')

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

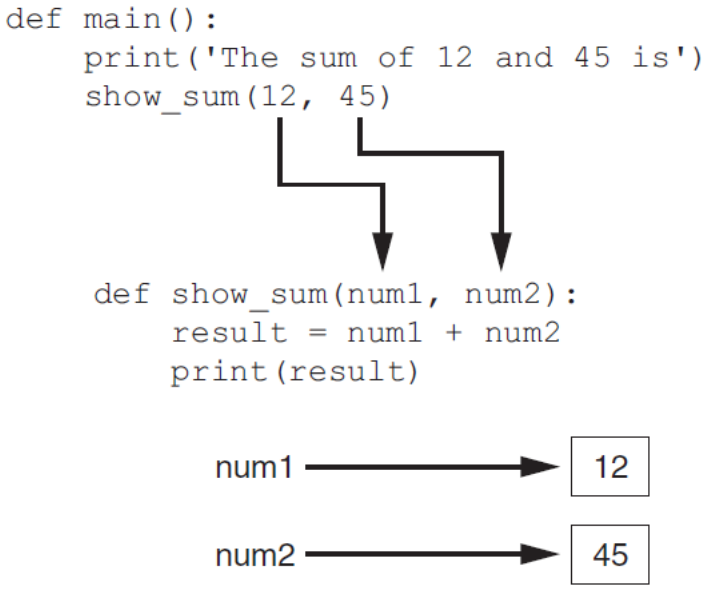
Passing Multiple Arguments

- **Python allows writing a function that accepts multiple arguments**
  - Parameter list replaces single parameter
    - Parameter list items separated by comma
- **Arguments are passed *by position* to corresponding parameters**
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

```
# This program demonstrates a function that accepts
# two arguments.

def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)

# The show_sum function accepts two arguments
# and displays their sum.
def show_sum(num1, num2):
    result = num1 + num2
    print(result)

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

Passing Multiple Arguments (cont'd.)

Figure 5-16 Two arguments passed to two parameters



Making Changes to Parameters

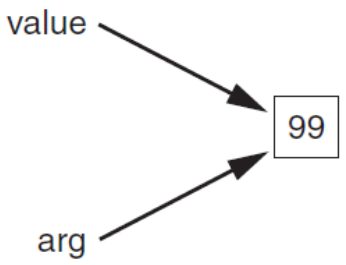
- **Changes made to a parameter value within the function do not affect the argument**
  - Known as *pass by value*
  - Provides a way for unidirectional communication between one function and another function
    - Calling function can communicate with called function

```
# This program demonstrates what happens when you
# change the value of a parameter.

def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)

def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)

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

Making Changes to Parameters (cont'd.)

Figure 5-17 The value variable is passed to the `change_me` function

```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)
```

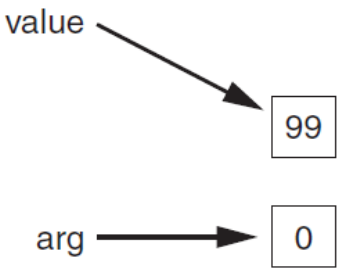


Making Changes to Parameters (cont'd.)

- **Figure 5-18**
  - The `value` variable passed to the `change_me` function cannot be changed by it

Figure 5-18 The `value` variable is passed to the `change_me` function

```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)
```



Keyword Arguments

- **Keyword argument: argument that specifies which parameter the value should be passed to**
  - Position when calling function is irrelevant
  - General Format:

```
function_name(parameter=value)
```

- **Possible to mix keyword and positional arguments when calling a function**
  - Positional arguments must appear first

```
# This program demonstrates keyword arguments.

def main():
    # Show the amount of simple interest using 0.01 as
    # interest rate per period, 10 as the number of periods,
    # and $10,000 as the principal.
    show_interest(rate=0.01, periods=10, principal=10000.0)

# The show_interest function displays the amount of
# simple interest for a given principal, interest rate
# per period, and number of periods.

def show_interest(principal, rate, periods): # unseq .. ok!
    interest = principal * rate * periods
    print('The simple interest will be $', \
          format(interest, ',.2f'), \
          sep='')

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

```
# This program demonstrates passes two strings as
# keyword arguments to a function.

def main():
    first_name = input('Enter your first name: ')
    last_name = input('Enter your last name: ')
    print('Your name reversed is')
    reverse_name(last=last_name, first=first_name)

def reverse_name(first, last):
    print(last, first)

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

Default Arguments

- **Default argument: give default value to arguments**

```
def show_interest(principal, rate = 0.1, periods=0.9):
    interest = principal * rate * periods
    print('The simple interest will be $', \
          format(interest, ',.2f'), \
          sep='')

# Call the function.
show_interest(0.2, 0.4, 0.5)
show_interest(0.3)
```

Global Variables and Global Constants

- **Global variable: created by assignment statement written outside all the functions**
  - Can be accessed by any statement in the program file, including from within a function
  - If a function needs to assign a value to the global variable, the global variable must be redeclared within the function
    - General format: `global` *`variable_name`*

```
# Create a global variable.
my_value = 10

# The show_value function prints
# the value of the global variable.
def show_value():
    print(my_value)

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

```
# Create a global variable.
number = 0

def main():
    global number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print('The number you entered is', number)

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

Global Variables and Global Constants (cont'd.)

- **Reasons to avoid using global variables:**
  - Global variables making debugging difficult
    - Many locations in the code could be causing a wrong variable value
  - Functions that use global variables are usually dependent on those variables
    - Makes function hard to transfer to another program
  - Global variables make a program hard to understand

Global Constants

- **Global constant: global name that references a value that cannot be changed**
  - Permissible to use global constants in a program
  - To simulate global constant in Python, create global variable and do not re-declare it within functions

```
# The following is used as a global constant to represent
# the contribution rate.
CONTRIBUTION_RATE = 0.05

def main():
    gross_pay = float(input('Enter the gross pay: '))
    bonus = float(input('Enter the amount of bonuses: '))
    show_pay_contrib(gross_pay)
    show_bonus_contrib(bonus)

def show_pay_contrib(gross):
    contrib = gross * CONTRIBUTION_RATE
    print('Contribution for gross pay: $', \
          format(contrib, ',.2f'), \
          sep='')

def show_bonus_contrib(bonus):
    contrib = bonus * CONTRIBUTION_RATE
    print('Contribution for bonuses: $', \
          format(contrib, ',.2f'), \
          sep='')

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

Introduction to Value-Returning Functions: Generating Random Numbers

- **void function: group of statements within a program for performing a specific task**
  - Call function when you need to perform the task
- **Value-returning function: similar to void function, returns a value**
  - Value returned to part of program that called the function when function finishes executing

Standard Library Functions and the `import` Statement

- **Standard library: library of pre-written functions that comes with Python**
  - *Library functions* perform tasks that programmers commonly need
    - Example: `print`, `input`, `range`
    - Viewed by programmers as a “black box”
- **Some library functions built into Python interpreter**
  - To use, just call the function

Standard Library Functions and the `import` Statement (cont'd.)

- **Modules: files that stores functions of the standard library**
  - Help organize library functions not built into the interpreter
  - Copied to computer when you install Python
- **To call a function stored in a module, need to write an `import` statement**
  - Written at the top of the program
  - Format: `import module_name`

Standard Library Functions and the `import` Statement (cont'd.)

Figure 5-19 A library function viewed as a black box

Generating Random Numbers

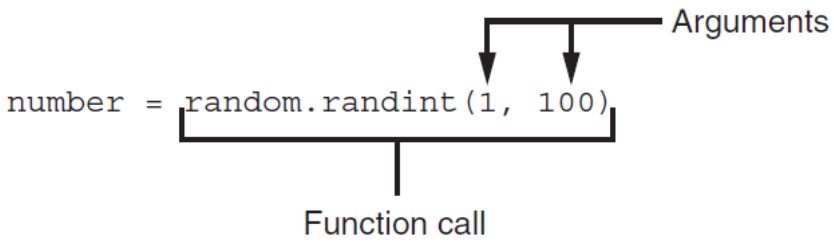
- **Random number are useful in a lot of programming tasks**
- **`random` module: includes library functions for working with random numbers**
- **Dot notation: notation for calling a function belonging to a module**
  - Format: `module_name.function_name()`

Generating Random Numbers (cont'd.)

- **`randint` function: generates a random number in the range provided by the arguments**
  - Returns the random number to part of program that called the function
  - Returned integer can be used anywhere that an integer would be used
  - You can experiment with the function in interactive mode

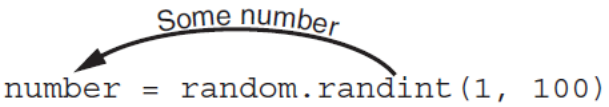
Generating Random Numbers (cont'd.)

Figure 5-20 A statement that calls the `random` function



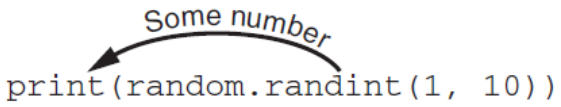
Generating Random Numbers (cont'd.)

Figure 5-21 The `random` function returns a value



A random number in the range of 1 through 100 will be assigned to the `number` variable.

Figure 5-22 Displaying a random number



A random number in the range of 1 through 10 will be displayed.

```
# This program displays a random number
# in the range of 1 through 10.
import random
def main():
    # Get a random number.
    number = random.randint(1, 10)
    # Display the number.
    print('The number is', number)
# Call the main function.
main()
```

```
# This program displays five random
# numbers in the range of 1 through 100.
import random

def main():
    for count in range(5):
        print(random.randint(1, 100))

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

```
# This program simulates the rolling of dice.
import random

# Constants for the minimum and maximum random numbers
MIN = 1
MAX = 6

def main():
    # Create a variable to control the loop.
    again = 'y'

    # Simulate rolling the dice.
    while again == 'y' or again == 'Y':
        print('Rolling the dice...')
        print('Their values are:')
        print(random.randint(MIN, MAX))
        print(random.randint(MIN, MAX))

        # Do another roll of the dice?
        again = input('Roll them again? (y = yes): ')

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

```
# This program simulates 10 tosses of a coin.
import random

# Constants
HEADS = 1
TAILS = 2
TOSSES = 10

def main():
    for toss in range(TOSSES):
        # Simulate the coin toss.
        if random.randint(HEADS, TAILS) == HEADS:
            print('Heads')
        else:
            print('Tails')

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

Generating Random Numbers (cont'd.)

- **`randrange` function: similar to `range` function, but returns randomly selected integer from the resulting sequence**
  - Same arguments as for the `range` function
- **`random` function: returns a random float in the range of 0.0 and 1.0**
  - Does not receive arguments
- **`uniform` function: returns a random float but allows user to specify range**

Random Number Seeds

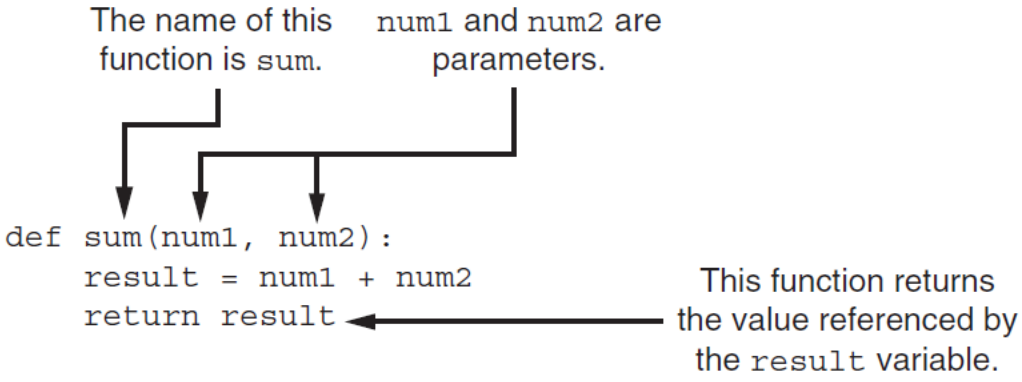
- **Random number created by functions in random module are actually pseudo-random numbers**
- **<u>Seed value</u>: initializes the formula that generates random numbers**
  - Need to use different seeds in order to get different series of random numbers
    - By default uses system time for seed
    - Can use `random.seed()` function to specify desired seed value, **e.g.**, `random.seed(10)`

Writing Your Own Value-Returning Functions

- **To write a value-returning function, you write a simple function and add one or more `return` statements**
  - Format: `return` *`expression`*
    - The value for *`expression`* will be returned to the part of the program that called the function
  - The expression in the `return` statement can be a complex expression, such as a sum of two variables or the result of another value-returning function

Writing Your Own Value-Returning Functions (cont'd.)

Figure 5-23 Parts of the function



```
# This program uses the return value of a function.

def main():
    # Get the user's age.
    first_age = int(input('Enter your age: '))

    # Get the user's best friend's age.
    second_age = int(input("Enter your best friend's age: "))

    # Get the sum of both ages.
    total = sum(first_age, second_age)

    # Display the total age.
    print('Together you are', total, 'years old.')

# The sum function accepts two numeric arguments and
# returns the sum of those arguments.
def sum(num1, num2):
    result = num1 + num2
    return result

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

How to Use Value-Returning Functions

- **Value-returning function can be useful in specific situations**
  - Example: have function prompt user for input and return the user's input
  - Simplify mathematical expressions
  - Complex calculations that need to be repeated throughout the program
- **Use the returned value**
  - Assign it to a variable or use as an argument in another function

```
DISCOUNT_PERCENTAGE = 0.20

# The main function.
def main():
    reg_price = get_regular_price()
    sale_price = reg_price - discount(reg_price)
    print('The sale price is $', format(sale_price, ',.2f'), sep='')

# The get_regular_price function prompts the
# user to enter an item's regular price and it
# returns that value.
def get_regular_price():
    price = float(input("Enter the item's regular price: "))
    return price

# The discount function accepts an item's price
# as an argument and returns the amount of the
# discount, specified by DISCOUNT_PERCENTAGE.
def discount(price):
    return price * DISCOUNT_PERCENTAGE

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

```
def main():
    sales = get_sales()
    advanced_pay = get_advanced_pay()
    comm_rate = determine_comm_rate(sales)
    pay = sales * comm_rate - advanced_pay

def get_sales():
    monthly_sales = float(input('Enter the monthly sales: '))
    return monthly_sales

def get_advanced_pay():
    print('Enter the amount of advanced pay, or’)
    print('enter 0 if no advanced pay was given.’)
    advanced = float(input('Advanced pay: ‘))
    return advanced

def determine_comm_rate(sales):
    if sales < 10000.00:
        rate = 0.10
    else:
        rate = 0.18
    return rate

main()
```

Returning Strings

- **You can write functions that return strings**
- **For example:**

```
def get_name():
    # Get the user's name.
    name = input('Enter your name: ')
    # Return the name.
    return name
```

Returning Boolean Values

- **Boolean function: returns either `True` or `False`**
  - Use to test a condition such as for decision and repetition structures
    - Common calculations, such as whether a number is even, can be easily repeated by calling a function
  - Use to simplify complex input validation code

    Ex: `while is_invalid(model):`

Returning Multiple Values

- **In Python, a function can return multiple values**
  - Specified after the `return` statement separated by commas
    - For example: **`return first, last.`**
  - When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value
    - **For example:**

```
first_name, last_name = get_name()
```

The `math` Module

- **`math` module: part of standard library that contains functions that are useful for performing mathematical calculations**
  - Typically accept one or more values as arguments, perform mathematical operation, and return the result
  - Use of module requires an `import math` statement

```
# This program demonstrates the sqrt function.
import math

def main():
    # Get a number.
    number = float(input('Enter a number: '))

    # Get the square root of the number.
    square_root = math.sqrt(number)

    # Display the square root.
    print('The square root of', number, 'is', square_root)

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

```
# This program calculates the length of a right
# triangle's hypotenuse.
import math

def main():
    # Get the length of the triangle's two sides.
    a = float(input('Enter the length of side A: '))
    b = float(input('Enter the length of side B: '))

    # Calculate the length of the hypotenuse.
    c = math.hypot(a, b)
    # distance between (0, 0) to (a,b)

    # Display the length of the hypotenuse.
    print('The length of the hypotenuse is', c)

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

The `math` Module (cont'd.)

Table 5-2 Many of the functions in the `math` module

`math` Module Function	Description
`acos(x)`	Returns the arc cosine of `x`, in radians.
`asin(x)`	Returns the arc sine of `x`, in radians.
`atan(x)`	Returns the arc tangent of `x`, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to `x`.
`cos(x)`	Returns the cosine of `x` in radians.
`degrees(x)`	Assuming `x` is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to `x`.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (`x`, `y`).
`log(x)`	Returns the natural logarithm of `x`.
`log10(x)`	Returns the base-10 logarithm of `x`.
`radians(x)`	Assuming `x` is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of `x` in radians.
`sqrt(x)`	Returns the square root of `x`.
`tan(x)`	Returns the tangent of `x` in radians.

The `math` Module (cont'd.)

- **The `math` module defines variables `pi` and `e`, which are assigned the mathematical values for *pi* and *e***
  - Can be used in equations that require these values, to get more accurate results
- **Variables must also be called using the dot notation**
  - Example:

```
circle_area = math.pi * radius**2
```

Storing Functions in Modules

- **In large, complex programs, it is important to keep code organized**
- **Modularization: grouping related functions in modules**
  - Makes program easier to understand, test, and maintain
  - Make it easier to reuse code for multiple different programs
    - Import the module containing the required function to each program that needs it

Storing Functions in Modules (cont'd.)

- **Module is a file that contains Python code**
  - Contains function definition but does not contain calls to the functions
    - Importing programs will call the functions
- **Rules for module names:**
  - File name should end in `.py`
  - Cannot be the same as a Python keyword
- **Import module using `import` statement**

Separate programs into different file

→ In circle and rectangle, they only contain the needed modules

→ geometry.py will import circle/rectangle modules

```
# This program allows the user to choose various
# geometry calculations from a menu. This program
# imports the circle and rectangle modules.
import circle
import rectangle

# Constants for the menu choices
AREA_CIRCLE_CHOICE = 1
…… …… ……
…… …… ……
```

Menu Driven Programs

- **Menu-driven program: displays a list of operations on the screen, allowing user to select the desired operation**
  - List of operations displayed on the screen is called a *menu* ***(as shown in the example before)***
- **Program uses a decision structure to determine the selected menu option and required operation**
  - Typically repeats until the user quits

typing

```
from typing import Dict

def get_first_name(full_name: str) -> str:
    return full_name.split(" ")[0]

fallback_name: Dict[str, str] = {
    "first_name": "UserFirstName",
    "last_name": "UserLastName"
}

raw_name: str = input("Please enter your name: ")
first_name: str = get_first_name(raw_name)

# If the user didn't type anything in, use the fallback name
if not first_name:
    first_name = get_first_name(fallback_name)

print(f"Hi, {first_name}!")
```

指示輸入與回傳型別 → type hint

跟指定的不一樣會發生什麼事?

Week 2 Quiz 1

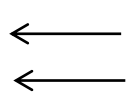
- **Print**

```
Please enter a value: 7
*******
******
*****
****
***
**
*
**
***
****
*****
******
*******
```

Week 2 Quiz 2

- **Print**

```
please enter a value: 7
##
# #
#  #
#   #
#    #
#     #
#      #
#       #
```



Week 2 Quiz 3

- **Print**

```
Please enter odd number 10
Please enter odd number 12
Please enter odd number 9
     *
    ***
   *****
  *******
*********
  *******
   *****
    ***
     *
```


Avoid wrong input

Week 2 Quiz 4

```
mspan@stu-000000005:~/python$ python3 ch5-1.py
Number of rows: 3
Number of column: 2
Grid size: 1
+-+-+
| | |
+-+-+
| | |
+-+-+
| | |
+-+-+
mspan@stu-000000005:~/python$ python3 ch5-1.py
Number of rows: 2
Number of column: 7
Grid size: 2
+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+
```

Week 2 Quiz 5

- **Number guessing game**
  - At least two functions: guessing() and main()
    - guessing() return if success to main()
    - You should have a limit on guess times

```
Please guess a number from 0 to 100: 90
Please guess a number from 0 to 90: 20
Please guess a number from 20 to 90: 10
Please guess a number from 20 to 90: 40
Please guess a number from 20 to 40: 25
Please guess a number from 25 to 40: 39
You passed
```

```
Please guess a number from 0 to 100: 1
Please guess a number from 1 to 100: 2
Please guess a number from 2 to 100: 3
Please guess a number from 3 to 100: 4
Please guess a number from 4 to 100: 5
Please guess a number from 5 to 100: 6
Achieve limitted
```

Week 2 Main

- **Implement a menu to execute the above five programs**
  - Store the above five quizzes in five different python files

```
ryanpan@RyanPanPC /Volumes/MyWork
1. Print double triangle
2. Print spacing triangle
3. Print diamond
4. Print grid
5. Guessing game
Please select: 1
Please enter a value: 5
*****
****
***
**
*
**
***
****
*****

Test again (y)?
```

main.py

```
import quiz1, quiz2, quiz3, quiz4, quiz5

def main():
    keep_going = "y"
    while keep_going == "y":
```