CN101

Lecture 2-3

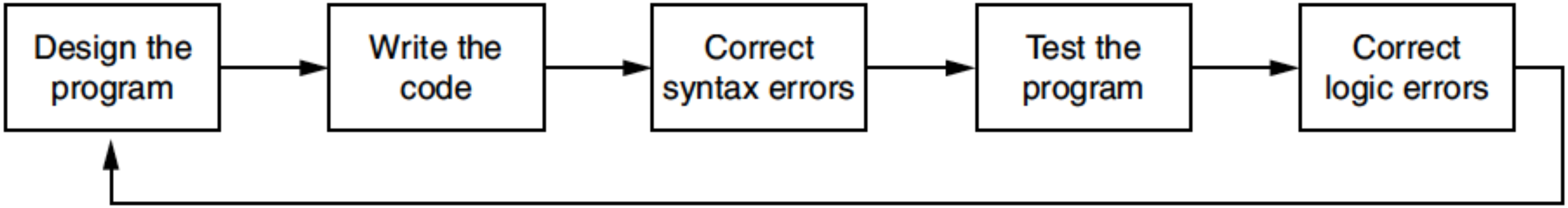
Input, Processing, and Output

Topics

- Designing a Program
- Input, Processing, and Output
- Displaying Output with `print` Function
- Comments
- Variables
- Reading Input from the Keyboard
- Performing Calculations
- More About Data Output
- Named Constants

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.)

- Design is the most important part of the program development cycle
- Understand the task that the program is to perform
    - Work with customer to get a sense what the program is supposed to do
    - Ask questions about program details
    - Create one or more software requirements

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

Pseudocode

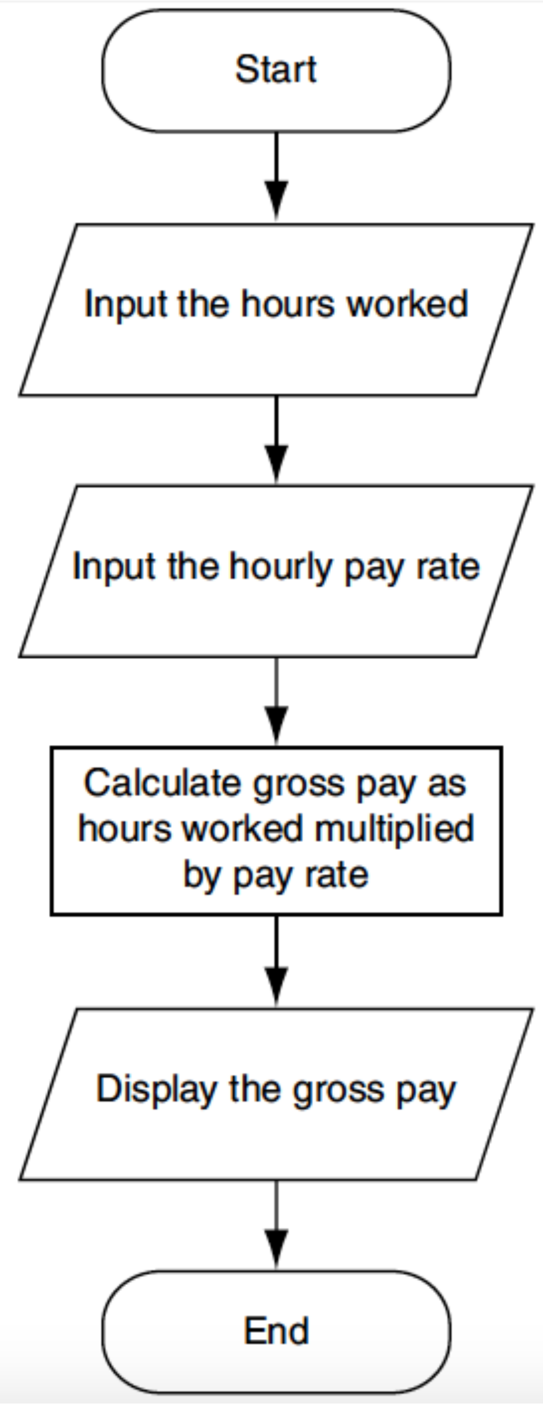
- Pseudocode: fake code
    - Informal language that has no syntax rule
    - Not meant to be compiled or executed
    - Used to create model program
        - No need to worry about syntax errors, can focus on program’s design
        - Can be translated directly into actual code in any programming language

Pseudocode (cont’d.)

- For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee.
- Here are the steps that you would take:
    1. Input the hours worked
    2. Input the hourly pay rate
    3. Calculate gross pay as hours worked multiplied by pay rate
    4. Display the gross pay

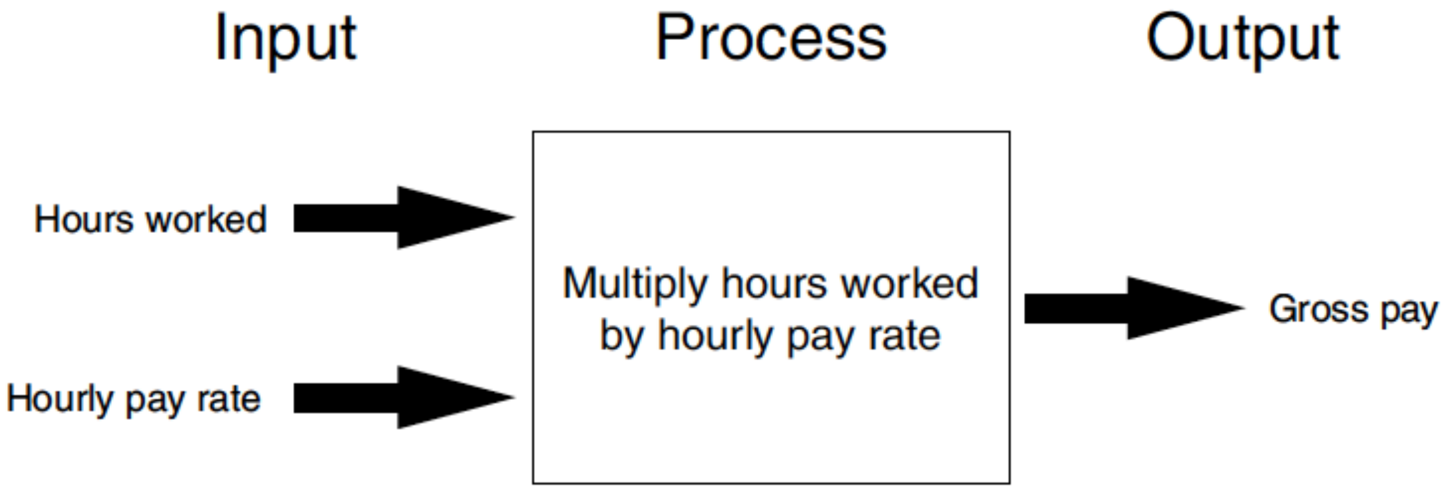
Flowcharts

- Flowchart: diagram that graphically depicts the steps in a program
  - Ovals are terminal symbols
  - Parallelograms are input and output symbols
  - Rectangles are processing symbols
  - Symbols are connected by arrows that represent the flow of the program


Start
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
End

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



Codes and Characters

- Each character is coded as a byte
- Most common coding system is ASCII (Pronounced as-key)
- ASCII = American National Standard Code for Information Interchange

ASCII Features

- 7-bit code
- 8th bit is unused (or used for a parity bit)
- 2^7 = 128 codes
- Two general types of codes:
  - 95 are “Graphic” codes (displayable on a console)
  - 33 are “Control” codes (control features of the console or communications channel)

Standard ASCII code (in decimal)

Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	**NUL**	16	**DLE**	32	**SP**	48	**0**	64	**@**	80	**P**	96	`	112	**p**
1	**SOH**	17	**DC1**	33	**!**	49	**1**	65	**A**	81	**Q**	97	**a**	113	**q**
2	**STX**	18	**DC2**	34	**"**	50	**2**	66	**B**	82	**R**	98	**b**	114	**r**
3	**ETX**	19	**DC3**	35	**#**	51	**3**	67	**C**	83	**S**	99	**c**	115	**s**
4	**EOT**	20	**DC4**	36	**$**	52	**4**	68	**D**	84	**T**	100	**d**	116	**t**
5	**ENQ**	21	**NAK**	37	**%**	53	**5**	69	**E**	85	**U**	101	**e**	117	**u**
6	**ACK**	22	**SYN**	38	**&**	54	**6**	70	**F**	86	**V**	102	**f**	118	**v**
7	**BEL**	23	**ETB**	39	**'**	55	**7**	71	**G**	87	**W**	103	**g**	119	**w**
8	**BS**	24	**CAN**	40	**(**	56	**8**	72	**H**	88	**X**	104	**h**	120	**x**
9	**HT**	25	**EM**	41	**)**	57	**9**	73	**I**	89	**Y**	105	**i**	121	**y**
10	**LF**	26	**SUB**	42	*	58	**:**	74	**J**	90	**Z**	106	**j**	122	**z**
11	**VT**	27	**ESC**	43	**+**	59	**;**	75	**K**	91	[	107	**k**	123	**{**
12	**FF**	28	**FS**	44	**,**	60	**<**	76	**L**	92	\	108	**l**	124	|
13	**CR**	29	**GS**	45	**-**	61	**=**	77	**M**	93	**]**	109	**m**	125	**}**
14	**SO**	30	**RS**	46	**.**	62	**>**	78	**N**	94	**^**	110	**n**	126	**~**
15	**SI**	31	**US**	47	**/**	63	**?**	79	**O**	95	_	111	**o**	127	**DEL**

SP means space.

Standard ASCII code (in decimal)

95 Graphic codes

Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	**NUL**	16	**DLE**	32	**SP**	48	**0**	64	**@**	80	**P**	96	`	112	**p**
1	**SOH**	17	**DC1**	33	**!**	49	**1**	65	**A**	81	**Q**	97	**a**	113	**q**
2	**STX**	18	**DC2**	34	**"**	50	**2**	66	**B**	82	**R**	98	**b**	114	**r**
3	**ETX**	19	**DC3**	35	**#**	51	**3**	67	**C**	83	**S**	99	**c**	115	**s**
4	**EOT**	20	**DC4**	36	**$**	52	**4**	68	**D**	84	**T**	100	**d**	116	**t**
5	**ENQ**	21	**NAK**	37	**%**	53	**5**	69	**E**	85	**U**	101	**e**	117	**u**
6	**ACK**	22	**SYN**	38	**&**	54	**6**	70	**F**	86	**V**	102	**f**	118	**v**
7	**BEL**	23	**ETB**	39	**'**	55	**7**	71	**G**	87	**W**	103	**g**	119	**w**
8	**BS**	24	**CAN**	40	**(**	56	**8**	72	**H**	88	**X**	104	**h**	120	**x**
9	**HT**	25	**EM**	41	**)**	57	**9**	73	**I**	89	**Y**	105	**i**	121	**y**
10	**LF**	26	**SUB**	42	*****	58	**:**	74	**J**	90	**Z**	106	**j**	122	**z**
11	**VT**	27	**ESC**	43	**+**	59	**;**	75	**K**	91	**[**	107	**k**	123	**{**
12	**FF**	28	**FS**	44	**,**	60	**<**	76	**L**	92	\	108	**l**	124	**|**
13	**CR**	29	**GS**	45	**-**	61	**=**	77	**M**	93	**]**	109	**m**	125	**}**
14	**SO**	30	**RS**	46	**.**	62	**>**	78	**N**	94	**^**	110	**n**	126	**~**
15	**SI**	31	**US**	47	**/**	63	**?**	79	**O**	95	**_**	111	**o**	127	**DEL**

SP means space.

Standard ASCII code (in decimal)

33 Control codes

Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	**NUL**	16	**DLE**	32	**SP**	48	**0**	64	**@**	80	**P**	96	`	112	**p**
1	**SOH**	17	**DC1**	33	**!**	49	**1**	65	**A**	81	**Q**	97	**a**	113	**q**
2	**STX**	18	**DC2**	34	**"**	50	**2**	66	**B**	82	**R**	98	**b**	114	**r**
3	**ETX**	19	**DC3**	35	**#**	51	**3**	67	**C**	83	**S**	99	**c**	115	**s**
4	**EOT**	20	**DC4**	36	**$**	52	**4**	68	**D**	84	**T**	100	**d**	116	**t**
5	**ENQ**	21	**NAK**	37	**%**	53	**5**	69	**E**	85	**U**	101	**e**	117	**u**
6	**ACK**	22	**SYN**	38	**&**	54	**6**	70	**F**	86	**V**	102	**f**	118	**v**
7	**BEL**	23	**ETB**	39	**'**	55	**7**	71	**G**	87	**W**	103	**g**	119	**w**
8	**BS**	24	**CAN**	40	**(**	56	**8**	72	**H**	88	**X**	104	**h**	120	**x**
9	**HT**	25	**EM**	41	**)**	57	**9**	73	**I**	89	**Y**	105	**i**	121	**y**
10	**LF**	26	**SUB**	42	*	58	**:**	74	**J**	90	**Z**	106	**j**	122	**z**
11	**VT**	27	**ESC**	43	**+**	59	**;**	75	**K**	91	[	107	**k**	123	**{**
12	**FF**	28	**FS**	44	**,**	60	**<**	76	**L**	92	\	108	**l**	124	**|**
13	**CR**	29	**GS**	45	**-**	61	**=**	77	**M**	93	**]**	109	**m**	125	**}**
14	**SO**	30	**RS**	46	**.**	62	**>**	78	**N**	94	**^**	110	**n**	126	**~**
15	**SI**	31	**US**	47	**/**	63	**?**	79	**O**	95	_	111	**o**	127	**DEL**

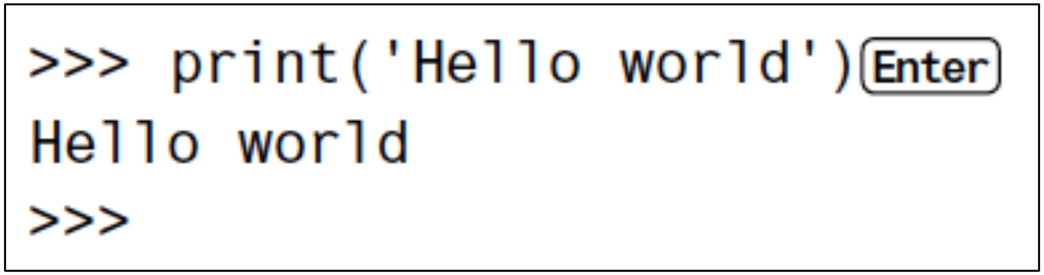
SP means space.

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
- Statements in a program execute in the order that they appear
  - From top to bottom

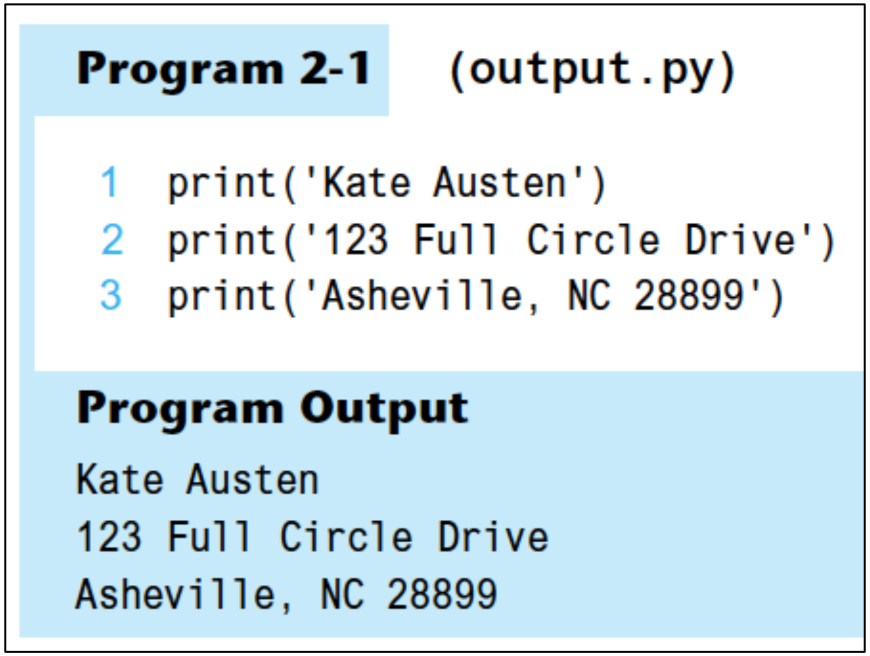
Displaying Output with the `print` Function (cont’d)

- In interactive mode



```
>>> print('Hello world')Enter
Hello world
>>>
```

- Script mode



Program 2-1 (output.py)

```
print('Kate Austen')
print('123 Full Circle Drive')
print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

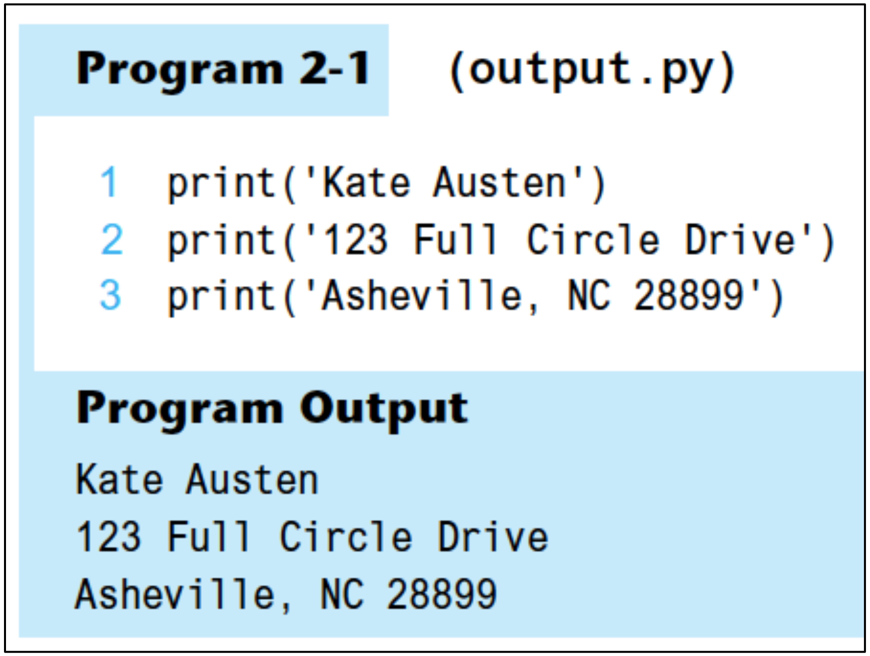
`chr(n)` and `ord(str)` functions

- Functions `chr(n)` and `ord(str)` access ASCII values
  - `print(chr(65))` displays the letter *A*
  - `print(ord('A'))` displays the number 65

```
>>> print(chr(65)) Enter
A
>>> print(ord('A')) Enter
65
>>>
```

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

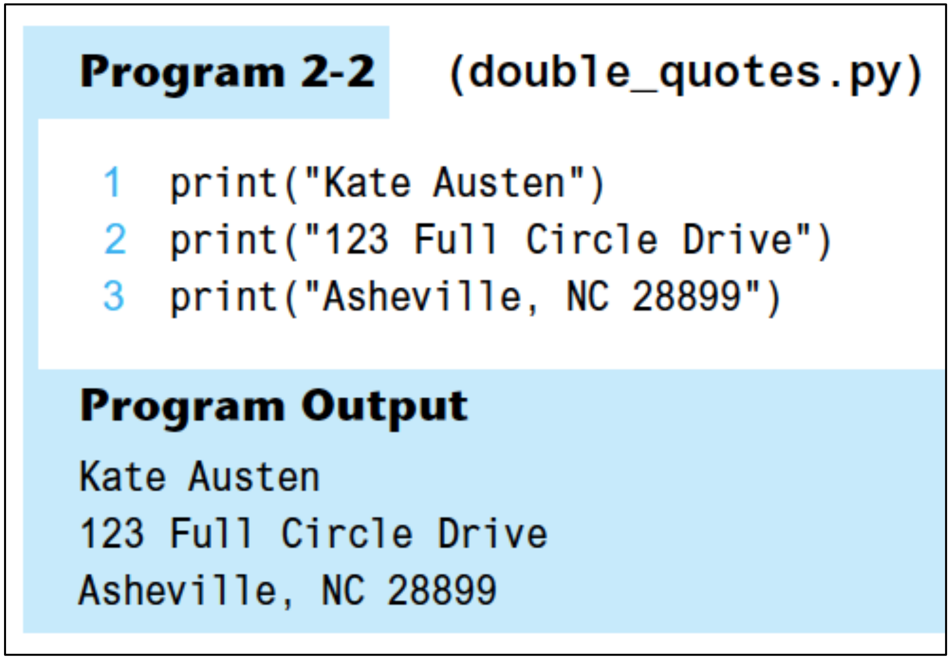


Program 2-1 (output.py)

```
print('Kate Austen')
print('123 Full Circle Drive')
print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```



Program 2-2 (double_quotes.py)

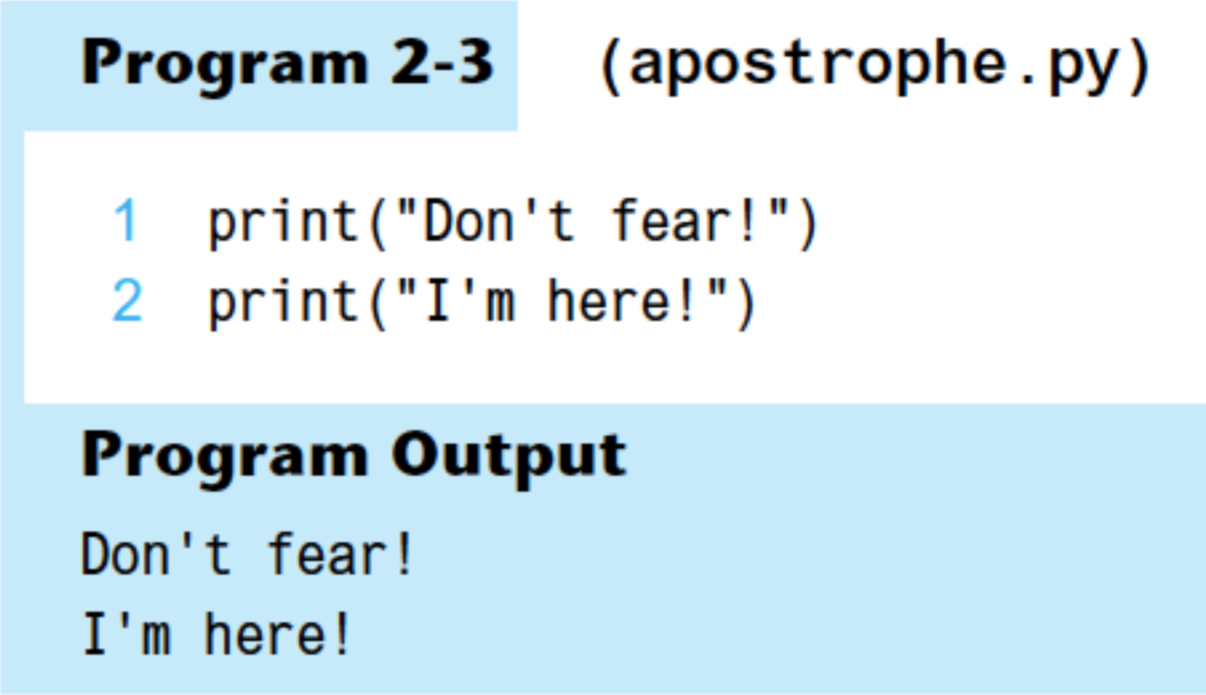
```
print("Kate Austen")
print("123 Full Circle Drive")
print("Asheville, NC 28899")
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Strings and String Literals (cont’d)

- If you want a string literal to contain either a single-quote or an apostrophe as part of the string, you can enclose the string literal in double-quote marks



Program 2-3 (apostrophe.py)

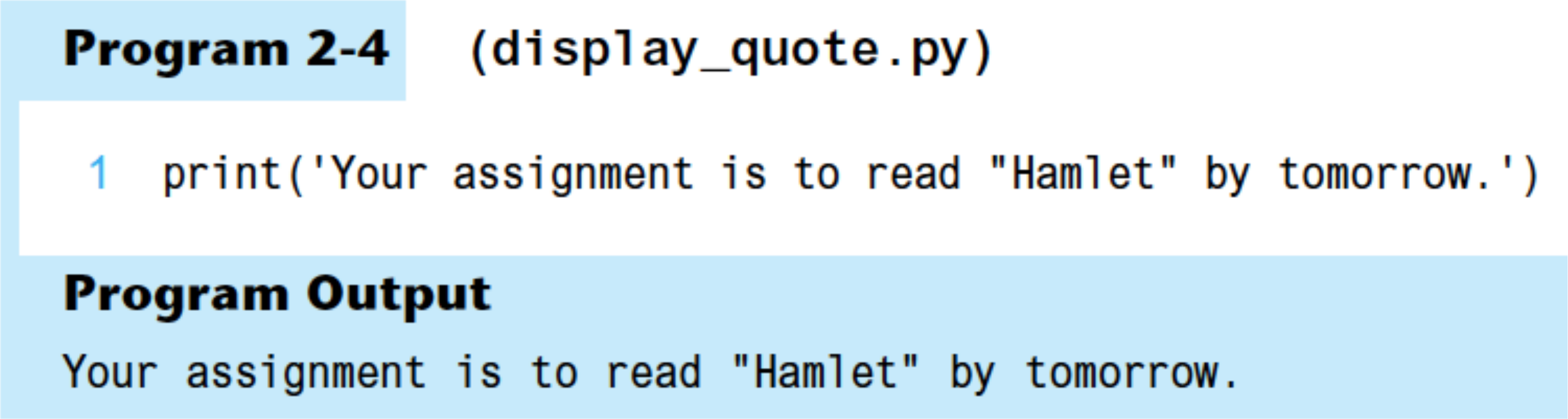
```python
print("Don't fear!")
print("I'm here!")
```

Program Output

```
Don't fear!
I'm here!
```

Strings and String Literals (cont’d)

- Similarly if you want a string literal to contain a double-quote, you can enclose the string literal in single-quote marks



Program 2-4 (display_quote.py)

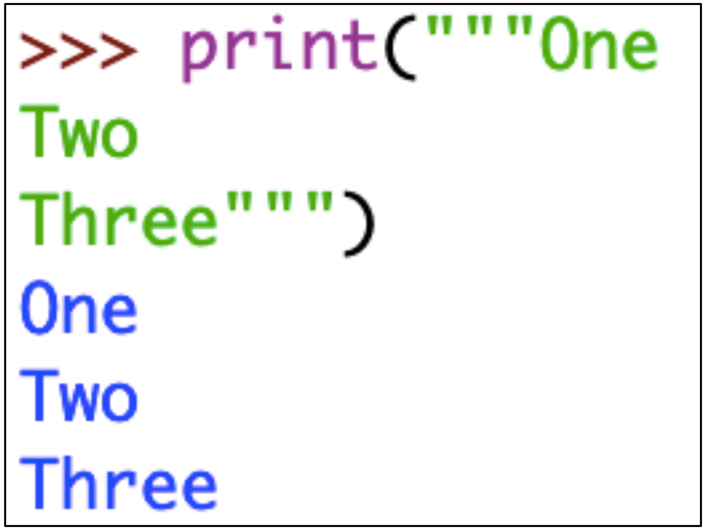
```python
print('Your assignment is to read "Hamlet" by tomorrow.')
```

Program Output

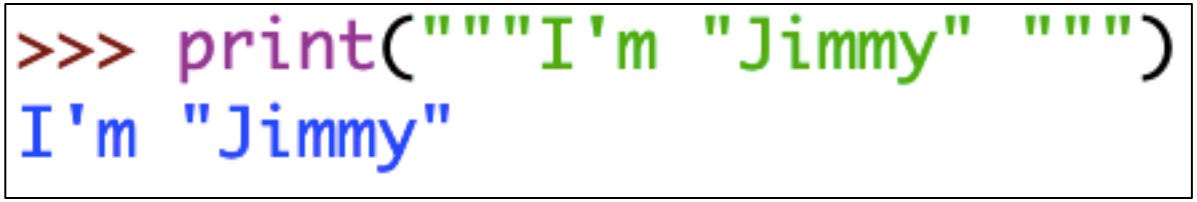
```
Your assignment is to read "Hamlet" by tomorrow.
```

Strings and String Literals (cont’d)

- String literal can be enclosed in triple quotes (''' or " " ")
  - Enclosed string can contain both single and double quotes and can have multiple lines
  - Here is an example:



```
>>> print("""One
Two
Three""")
One
Two
Three
```

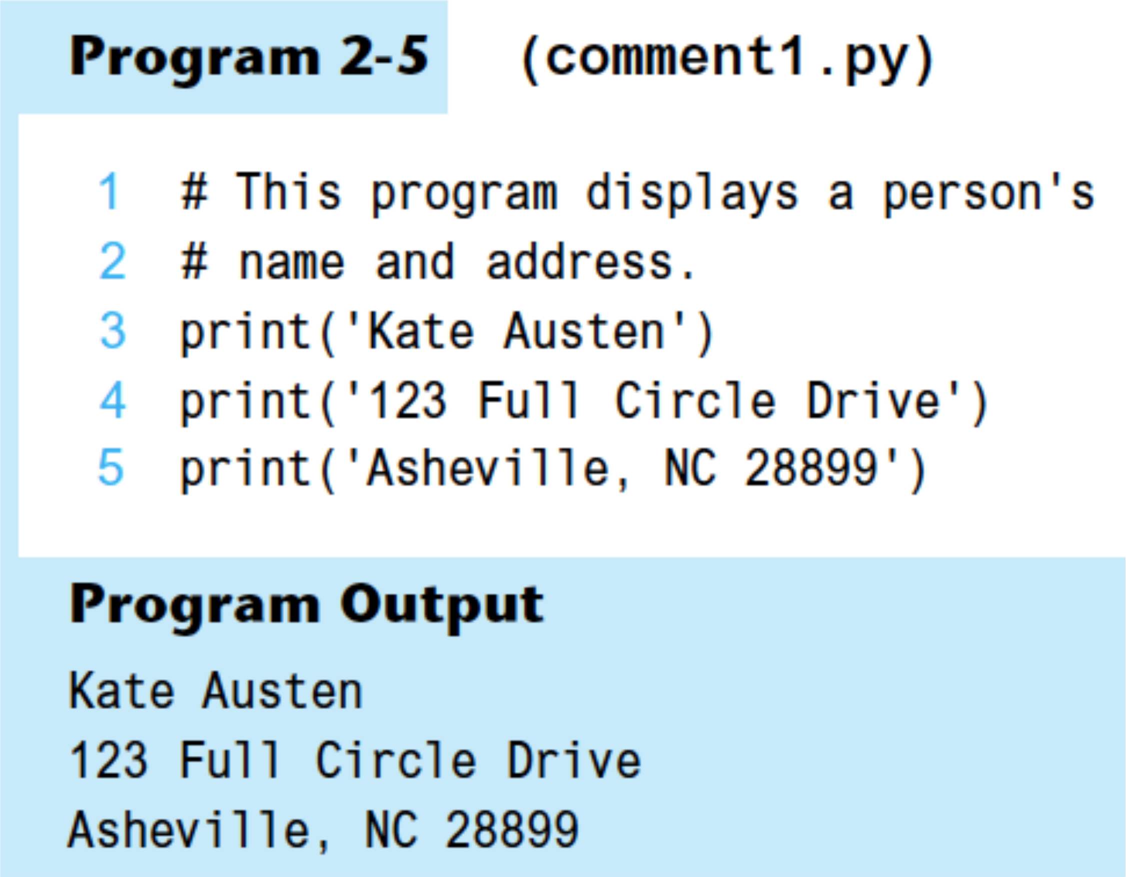


```
>>> print("""I'm "Jimmy" """)
I'm "Jimmy"
```

Comments

- Comments: notes of explanation within a program
    - Ignored by Python interpreter
        - Intended for a person reading the program’s code
    - Begin with a # character
- End-line comment: appears at the end of a line of code
    - Typically explains the purpose of that line

Comments (cont’d)



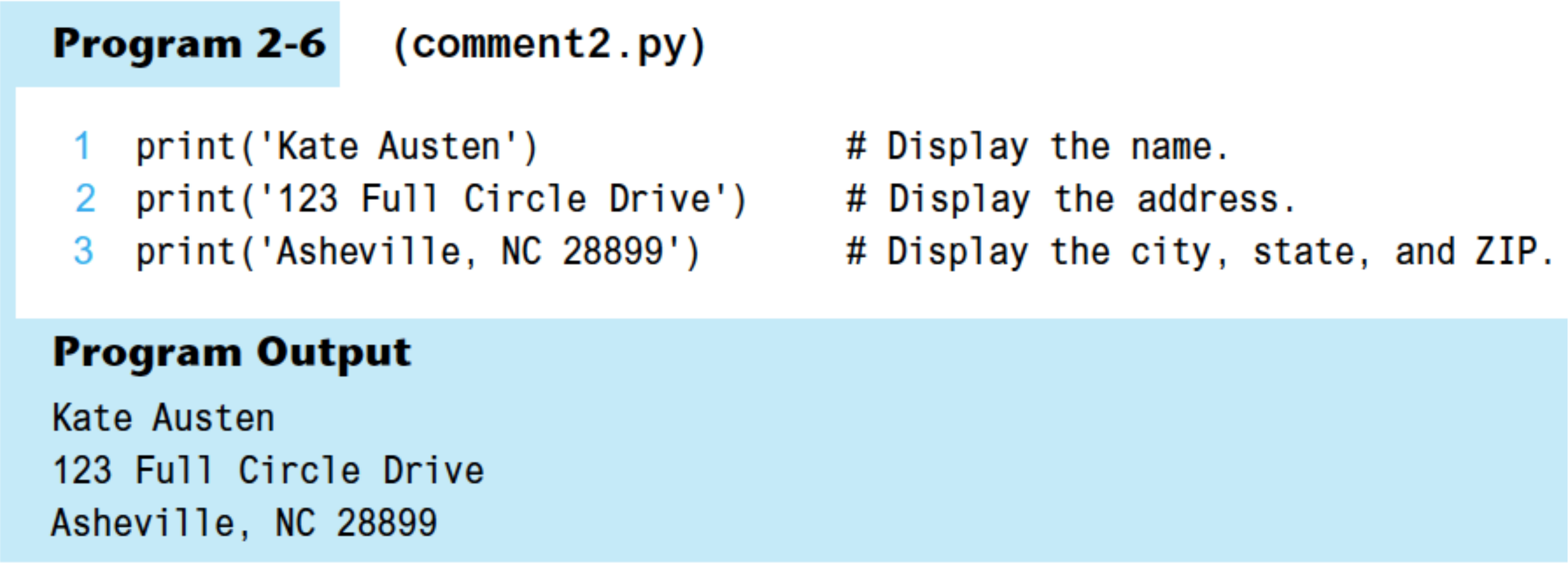
Program 2-5 (comment1.py)

```python
# This program displays a person's
# name and address.
print('Kate Austen')
print('123 Full Circle Drive')
print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Comments (cont’d)



Program 2-6 (comment2.py)

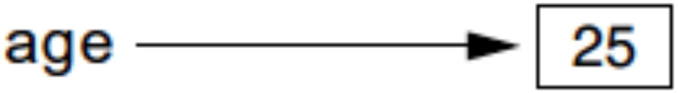
```
print('Kate Austen')              # Display the name.
print('123 Full Circle Drive')    # Display the address.
print('Asheville, NC 28899')      # Display the city, state, and ZIP.
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

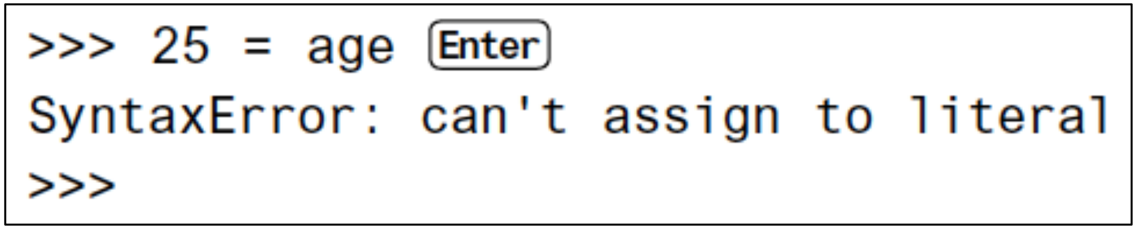
Variables

- 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 = 25`
    - Assignment operator: the equal sign (=)



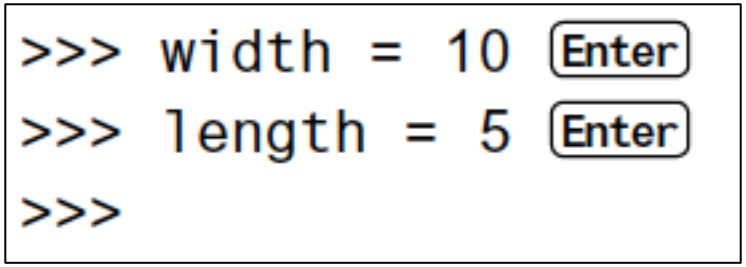
Variables (cont’d.)

- In assignment statement, variable receiving value must be on left side

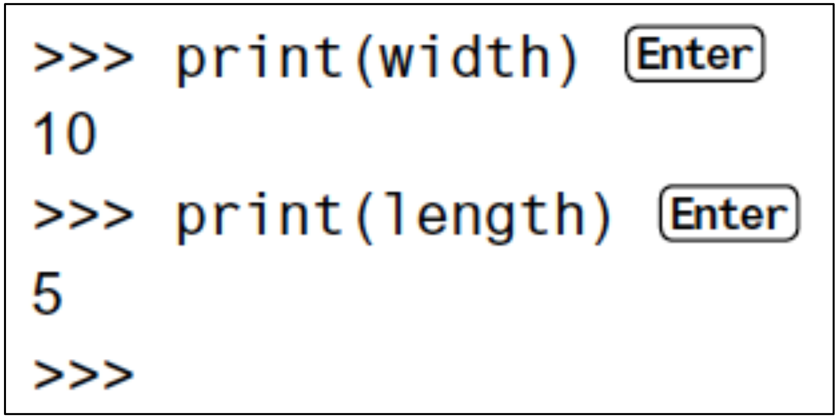


```
>>> 25 = age Enter
SyntaxError: can't assign to literal
>>>
```

- 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 if a value is assigned to it

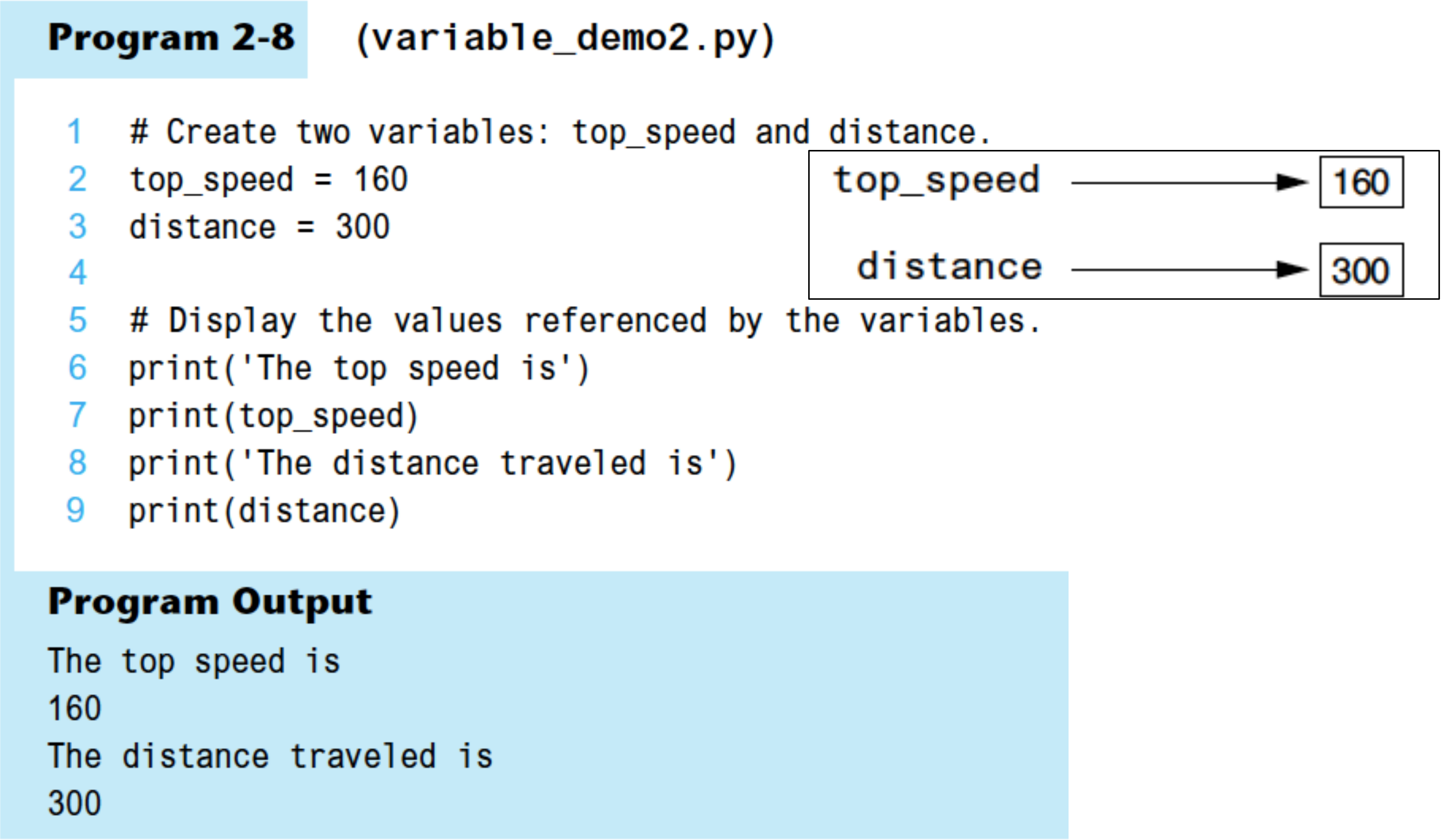


```
>>> width = 10 Enter
>>> length = 5 Enter
>>>
```



```
>>> print(width) Enter
10
>>> print(length) Enter
5
>>>
```

Example



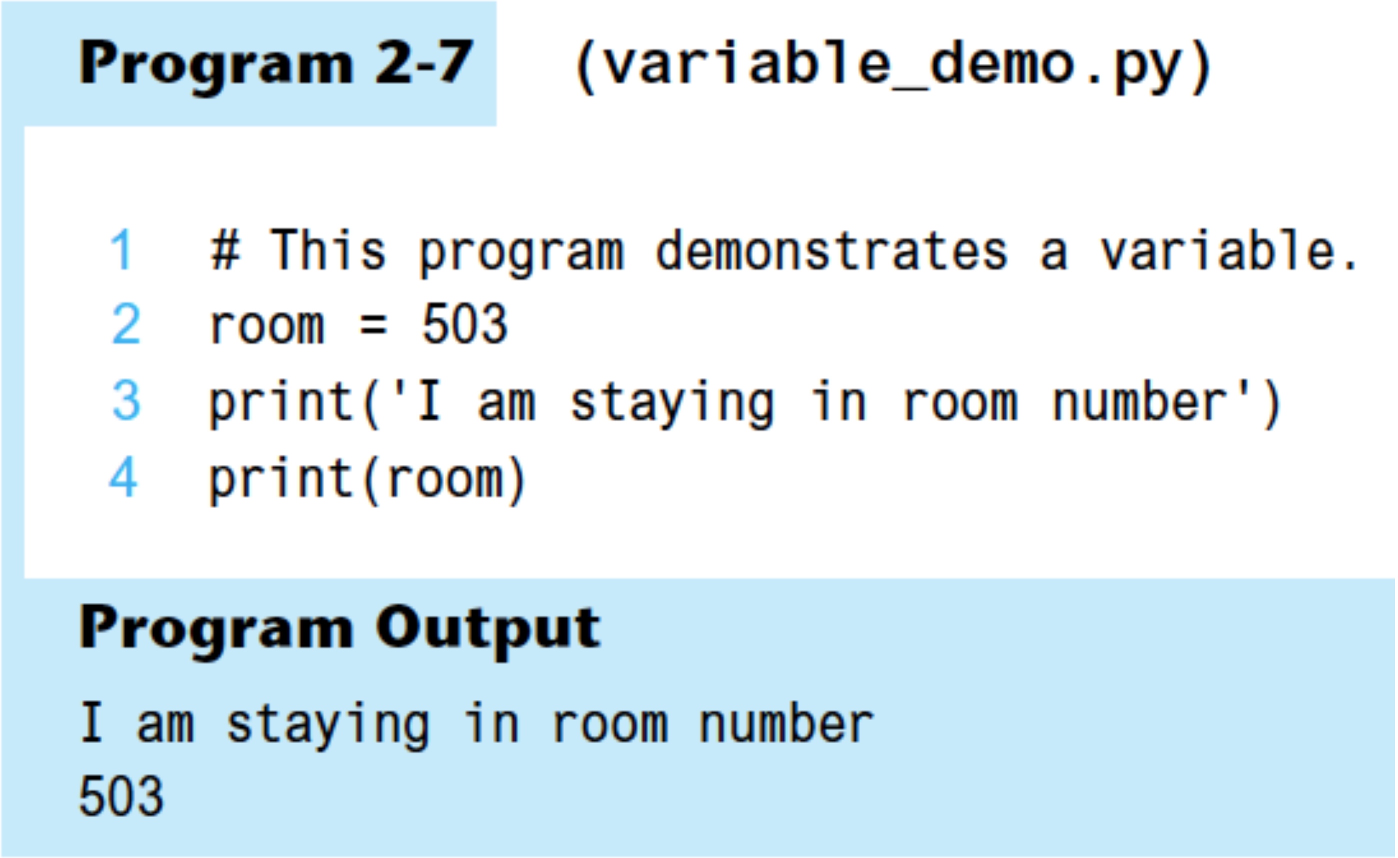
Program 2-8 (variable_demo2.py)

```
# Create two variables: top_speed and distance.
top_speed = 160
distance = 300

# Display the values referenced by the variables.
print('The top speed is')
print(top_speed)
print('The distance traveled is')
print(distance)
```

Program Output

```
The top speed is
160
The distance traveled is
300
```

Example



Program 2-7 (variable_demo.py)

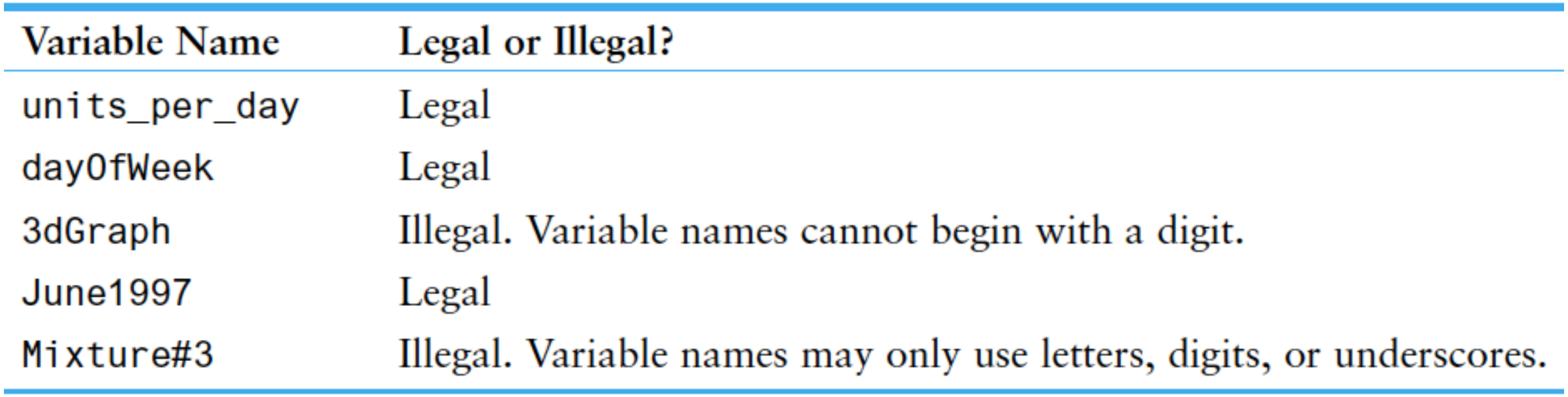
```
# This program demonstrates a variable.
room = 503
print('I am staying in room number')
print(room)
```

Program Output

```
I am staying in room number
503
```

Variable Naming Rules

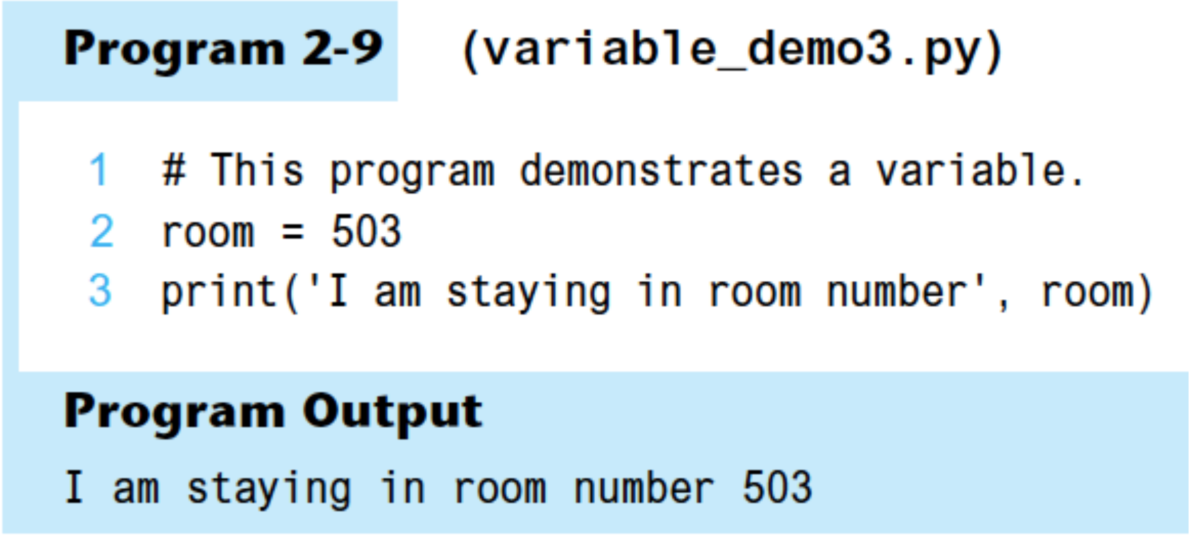
- Rules for naming variables in Python:
    - Variable name cannot be a Python key word
    - Variable name cannot contain spaces
    - First character must be a letter or an underscore
    - After first character may use letters, digits, or underscores
    - Variable names are case sensitive
- Variable name should reflect its use



Variable Name	Legal or Illegal?
`units_per_day`	Legal
`dayOfWeek`	Legal
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`June1997`	Legal
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

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



Program 2-9 (variable_demo3.py)

```
# This program demonstrates a variable.
room = 503
print('I am staying in room number', room)
```

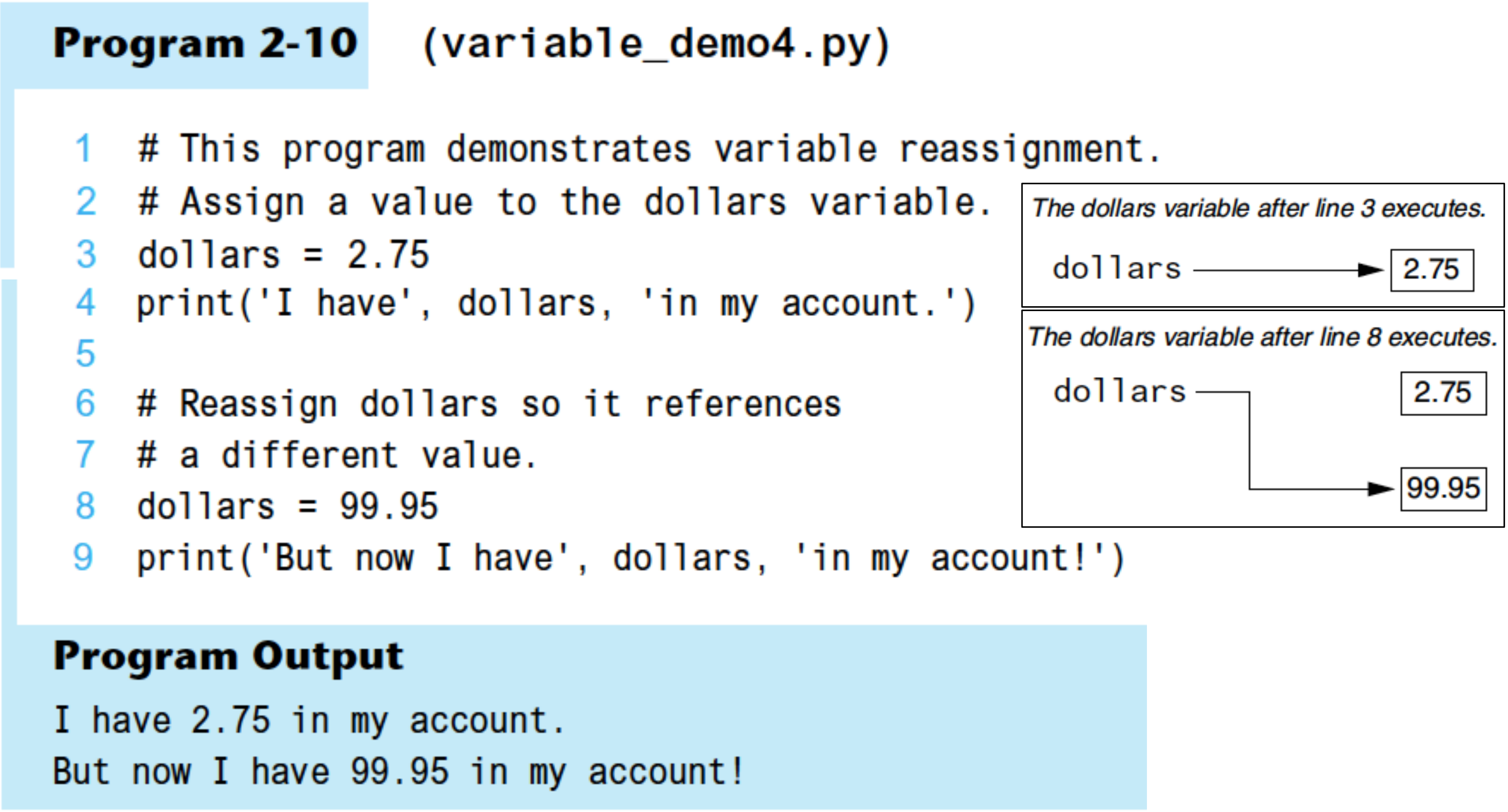
Program Output

```
I am staying in room number 503
```

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

Example



Program 2-10 (variable_demo4.py)

```
1 # This program demonstrates variable reassignment.
2 # Assign a value to the dollars variable.
3 dollars = 2.75
4 print('I have', dollars, 'in my account.')
5
6 # Reassign dollars so it references
7 # a different value.
8 dollars = 99.95
9 print('But now I have', dollars, 'in my account!')
```

The dollars variable after line 3 executes.

dollars
2.75

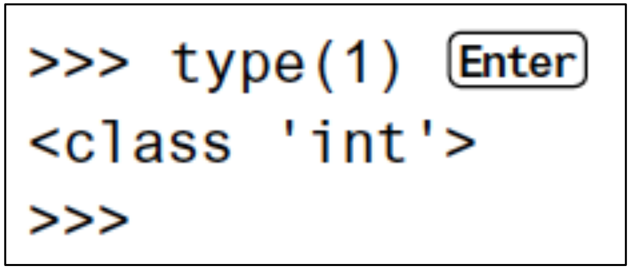
The dollars variable after line 8 executes.

Program Output

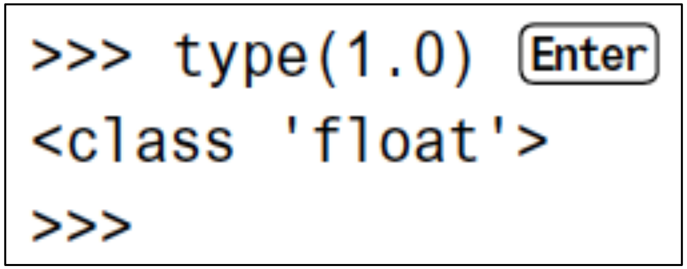
```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

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

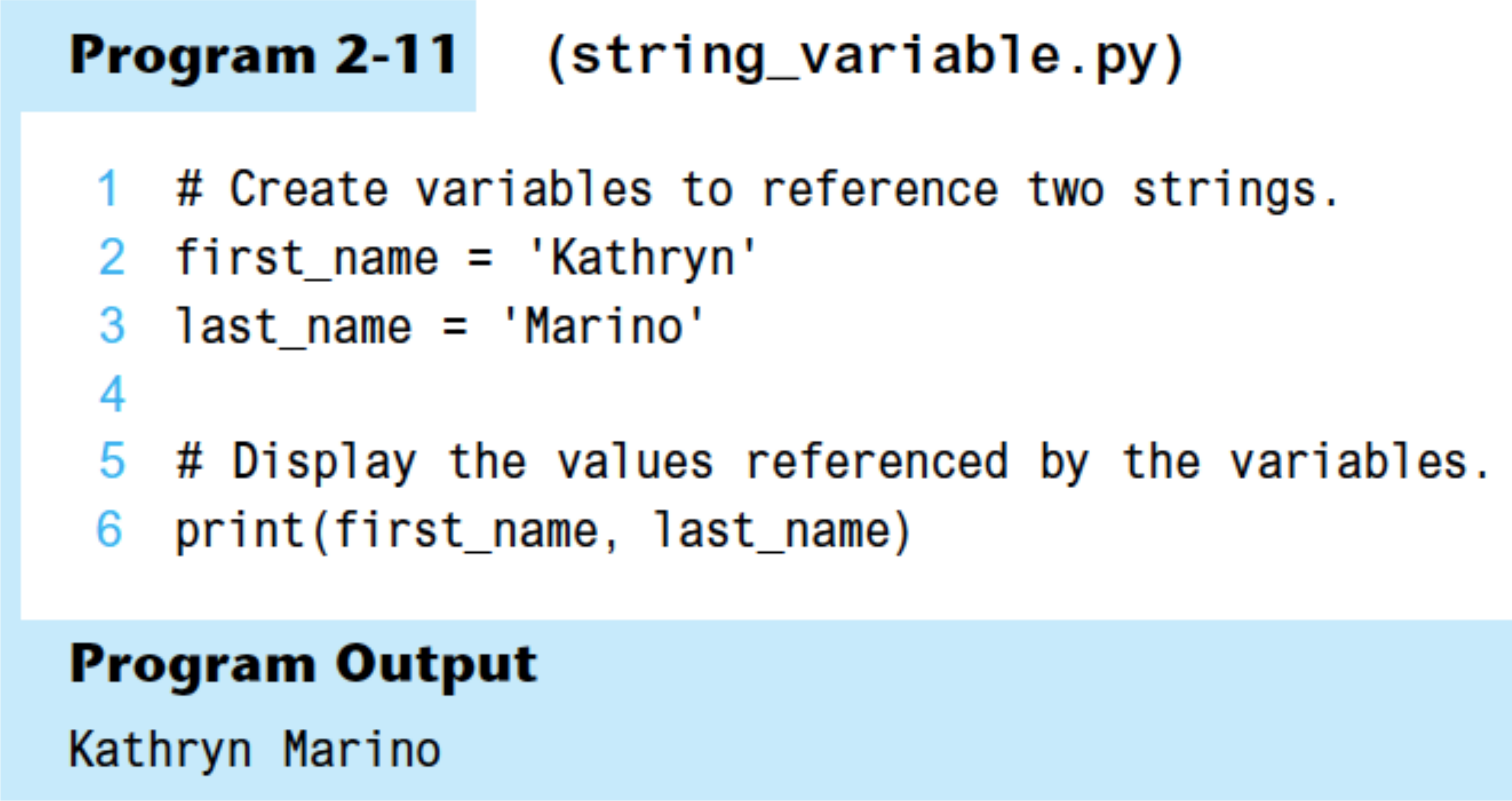


```
>>> type(1) Enter
<class 'int'>
>>>
```



```
>>> type(1.0) Enter
<class 'float'>
>>>
```

Storing Strings with the str Data Type



Program 2-11 (string_variable.py)

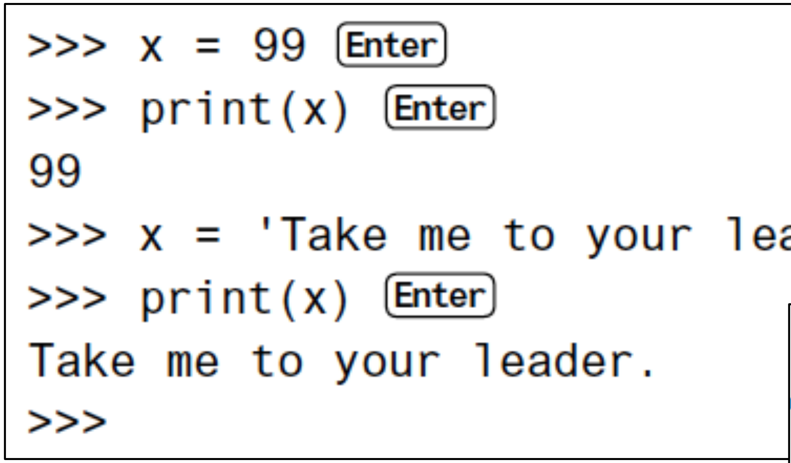
```
# Create variables to reference two strings.
first_name = 'Kathryn'
last_name = 'Marino'

# Display the values referenced by the variables.
print(first_name, last_name)
```

Program Output

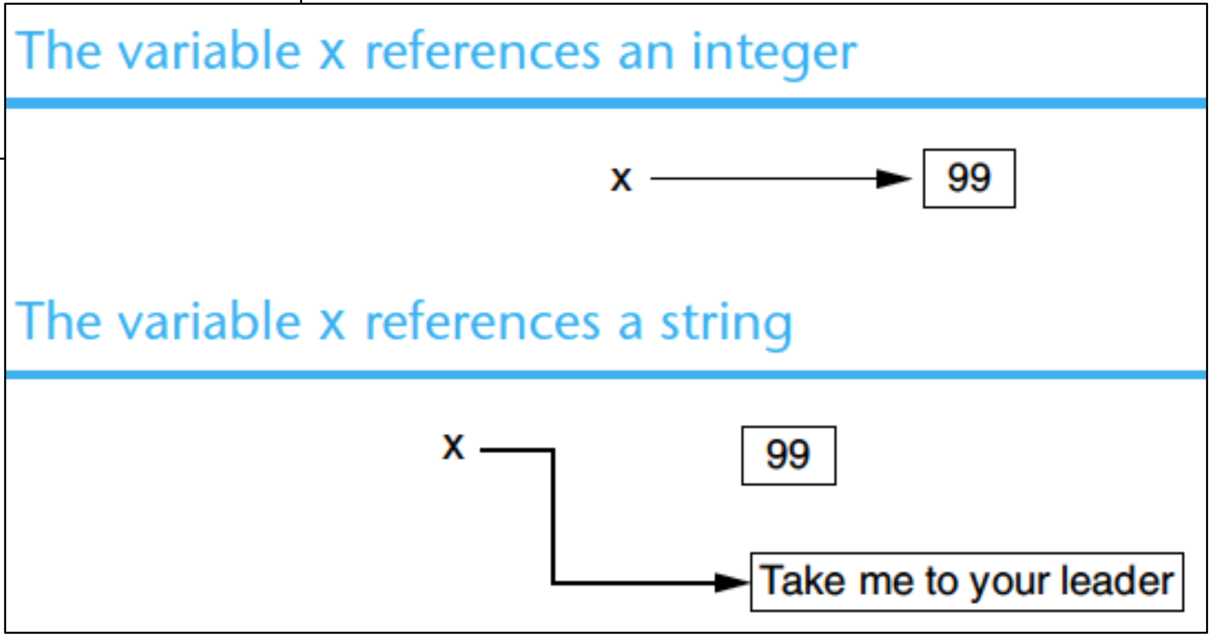
```
Kathryn Marino
```

Reassigning a Variable to a Different Type

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



```
>>> x = 99 Enter
>>> print(x) Enter
99
>>> x = 'Take me to your leader' Enter
>>> print(x) Enter
Take me to your leader.
>>>
```

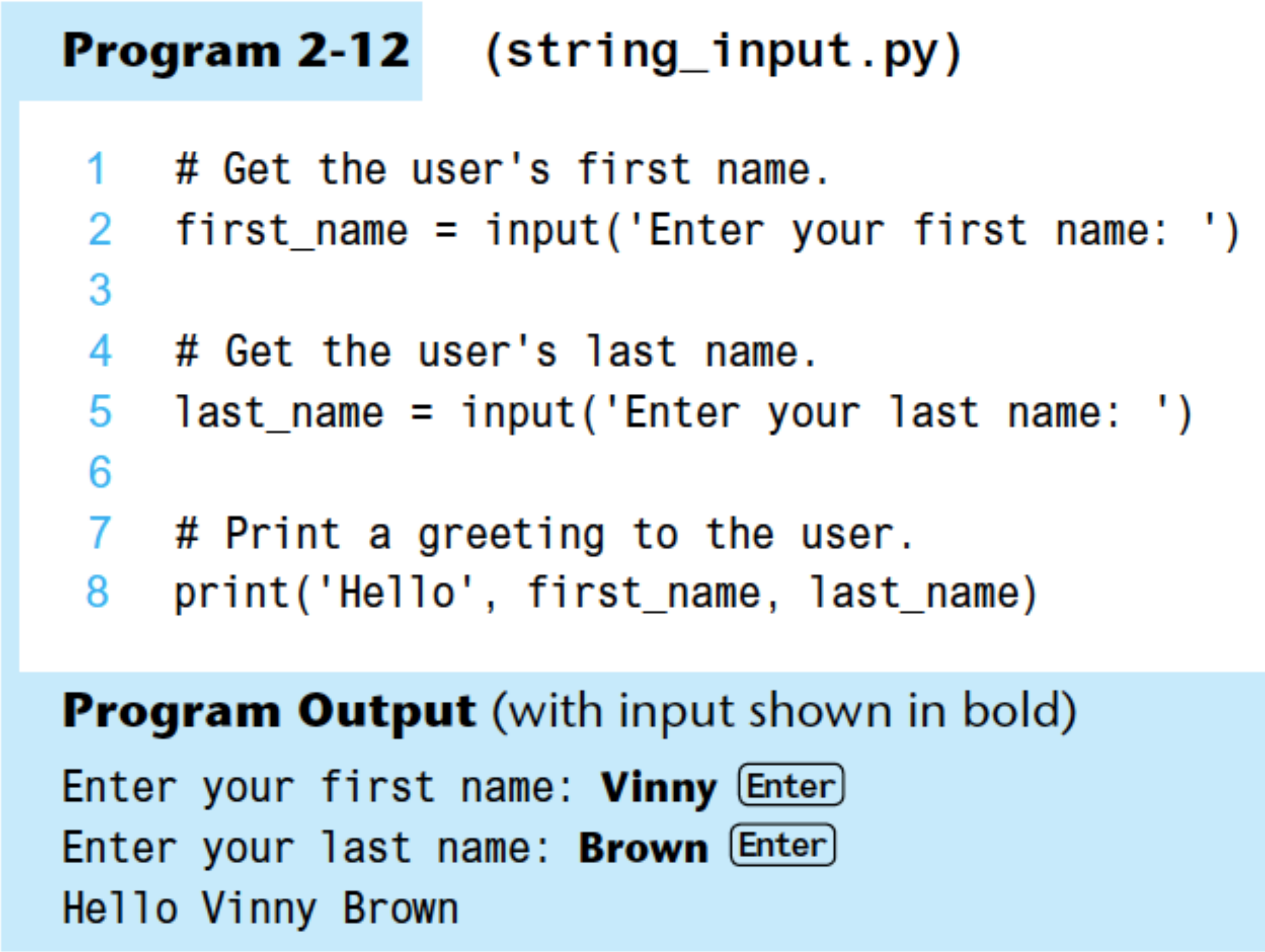


The variable x references an integer
x
99
The variable x references a string
x
99
Take me to your leader

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

Example



Program 2-12 (string_input.py)

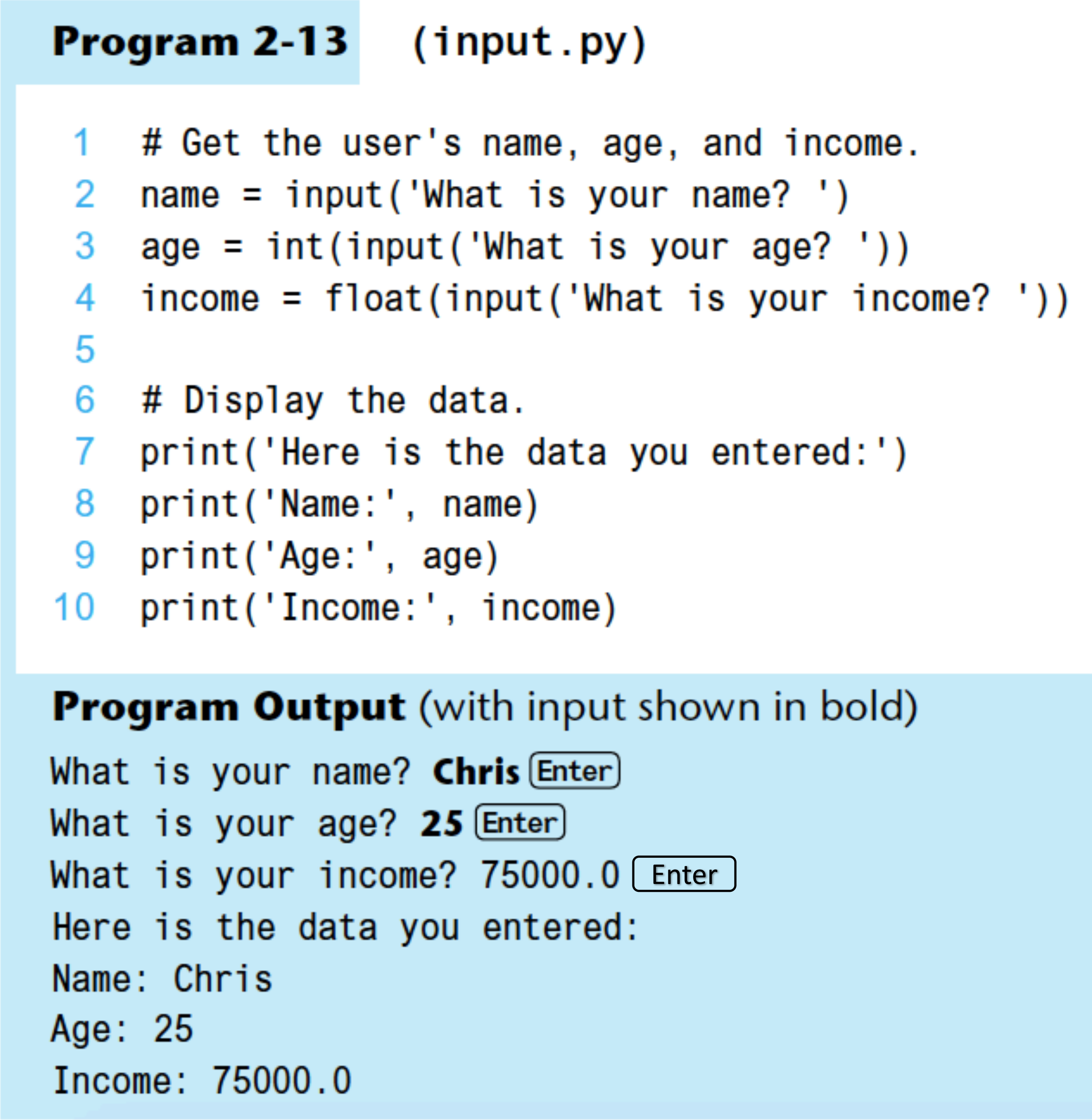
```
# Get the user's first name.
first_name = input('Enter your first name: ')

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

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

Program Output (with input shown in bold)

```
Enter your first name: Vinny Enter
Enter your last name: Brown Enter
Hello Vinny Brown
```

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



Program 2-13 (input.py)

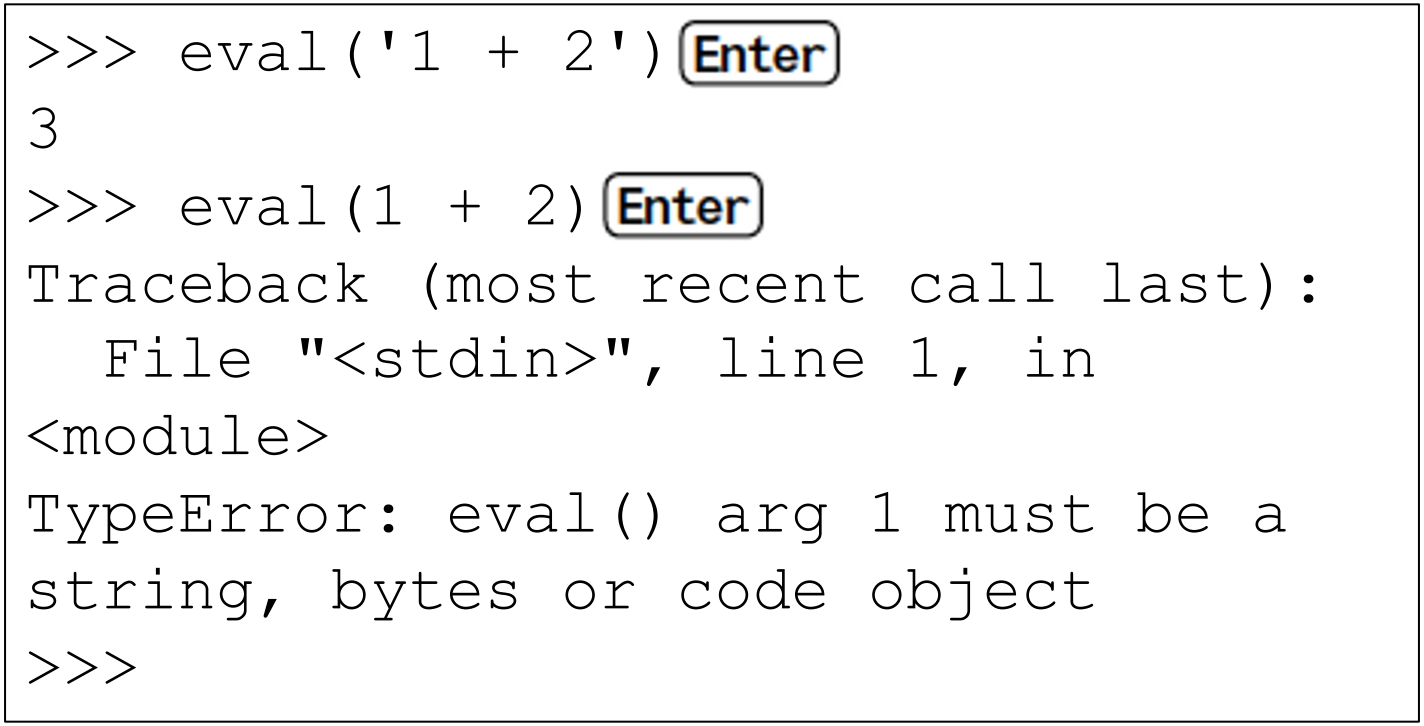
```
# Get the user's name, age, and income.
name = input('What is your name? ')
age = int(input('What is your age? '))
income = float(input('What is your income? '))

# Display the data.
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
print('Income:', income)
```

Program Output (with input shown in bold)

```
What is your name? Chris Enter
What is your age? 25 Enter
What is your income? 75000.0 Enter
Here is the data you entered:
Name: Chris
Age: 25
Income: 75000.0
```

eval() function

- The eval() function evaluates the specified expression, if the expression is a legal Python statement, it will be executed.



```
>>> eval('1 + 2') Enter
3
>>> eval(1 + 2) Enter
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: eval() arg 1 must be a
string, bytes or code object
>>>
```

input2.py

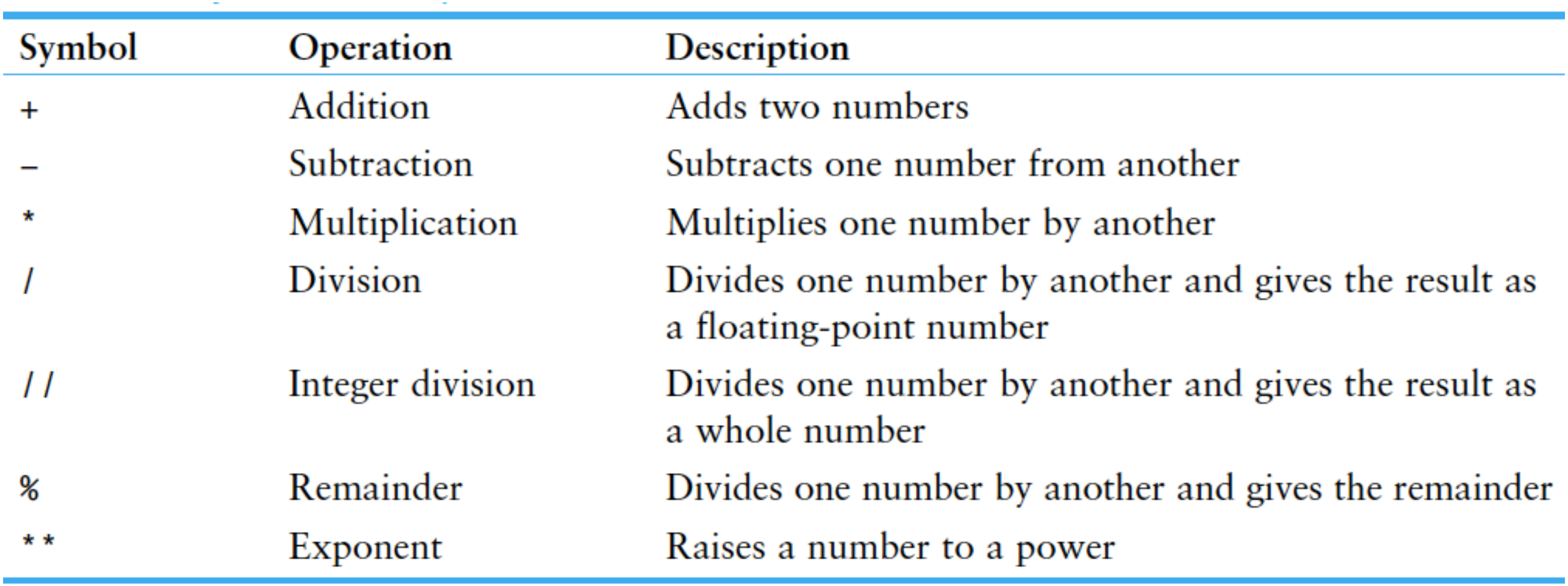
```python
# Get the user's name, age, and income.
name = input('What is your name? ‘)
age = eval(input('What is your age? ‘))
income = eval(input('What is your income? ‘))

# Display the data.
print('Here is the data you entered:’)
print('Name:', name)
print('Age:', age)
print('Income:', income)
```

```
What is your name? Peter Enter
What is your age? 35 Enter
What is your income? 10000.50 Enter
Here is the data you entered:
Name: Peter
Age: 35
Income: 10000.5
```

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

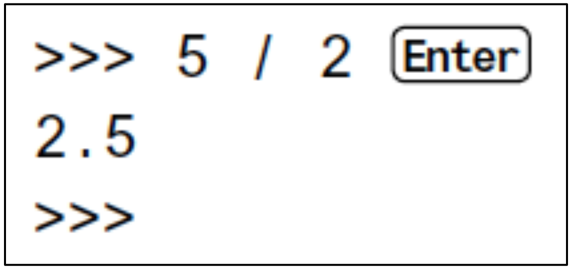
Performing Calculations (cont’d)



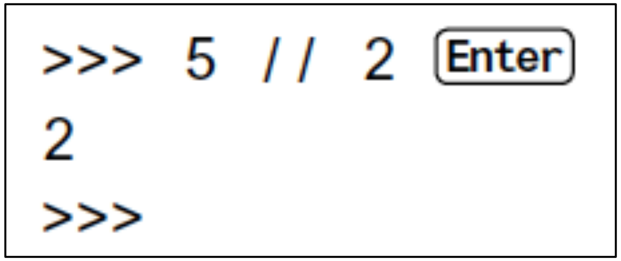
Symbol	Operation	Description
+	Addition	Adds two numbers
–	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result as a floating-point number
/ /	Integer division	Divides one number by another and gives the result as a whole number
%	Remainder	Divides one number by another and gives the remainder
* *	Exponent	Raises a number to a power

Performing Calculations (cont’d)

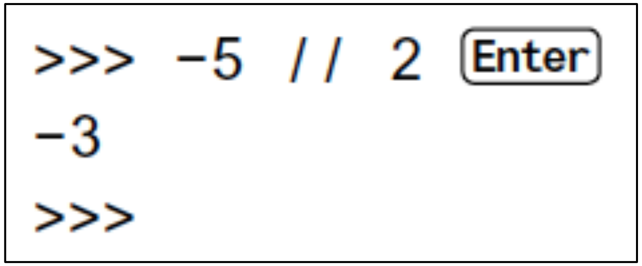
- Two types of division:
  - / operator performs floating point division
  - // operator performs integer division
    - Positive results truncated, negative rounded away from zero



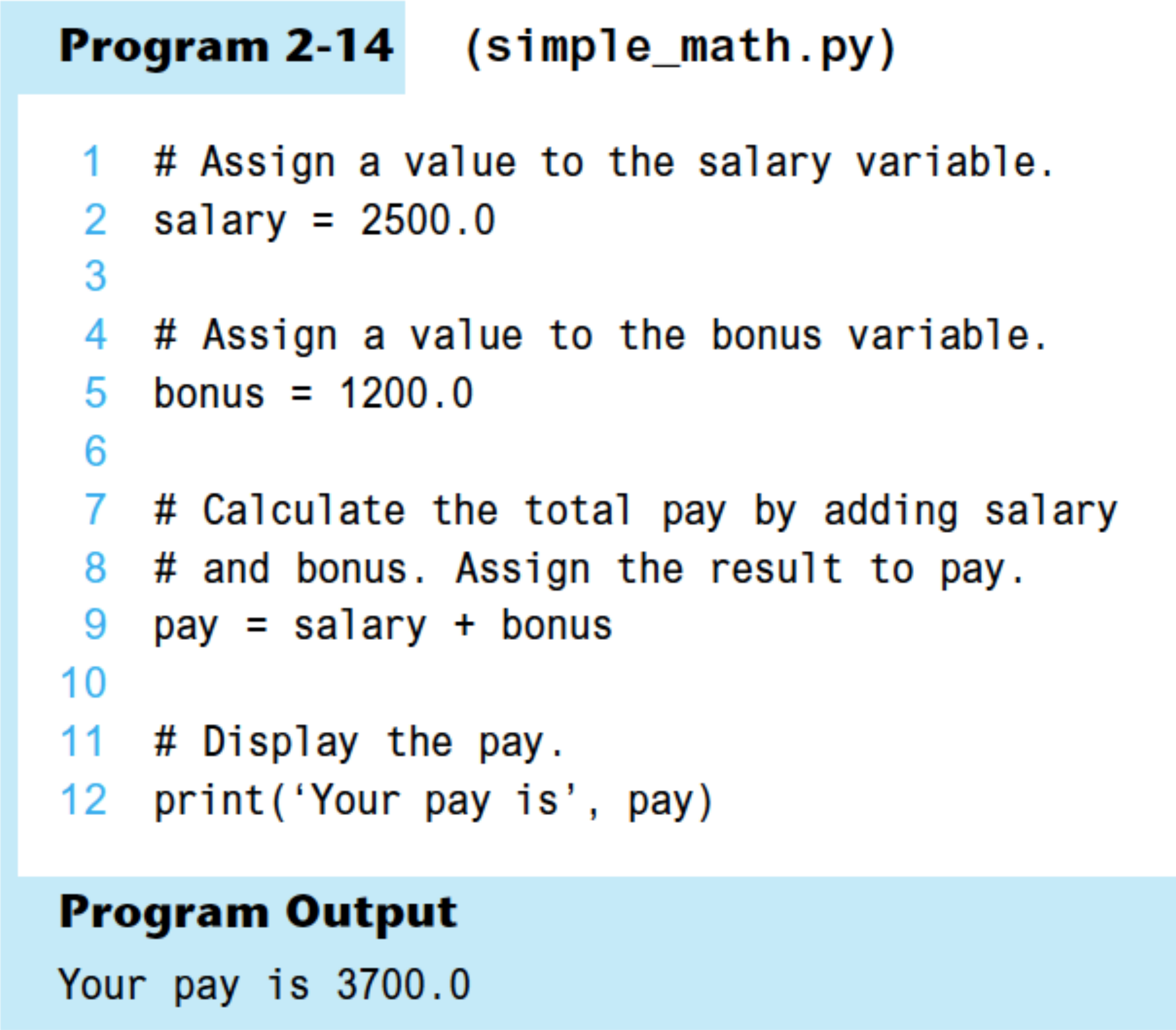
```
>>> 5 / 2 Enter
2.5
>>>
```



```
>>> 5 // 2 Enter
2
>>>
```



```
>>> -5 // 2 Enter
-3
>>>
```



Program 2-14 (simple_math.py)

```
# Assign a value to the salary variable.
salary = 2500.0

# Assign a value to the bonus variable.
bonus = 1200.0

# Calculate the total pay by adding salary
# and bonus. Assign the result to pay.
pay = salary + bonus

# Display the pay.
print(‘Your pay is’, pay)
```

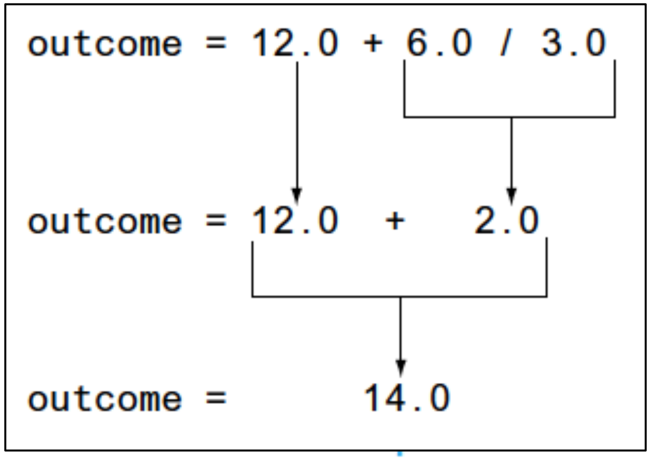
Program Output

```
Your pay is 3700.0
```

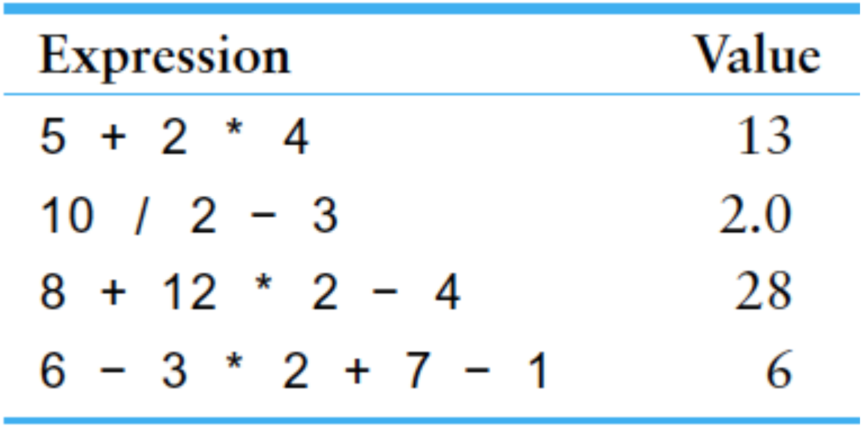
Operator Precedence and Grouping with Parentheses

- 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

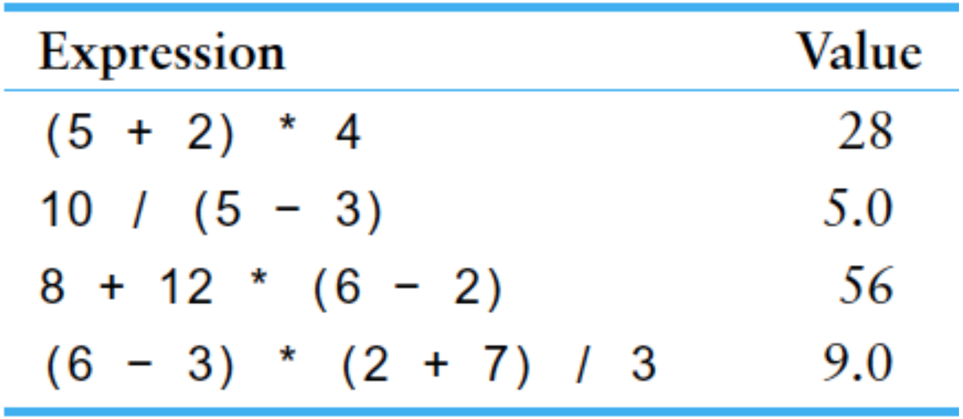
Example



```
outcome = 12.0 + 6.0 / 3.0
outcome = 12.0   +   2.0
outcome =        14.0
```



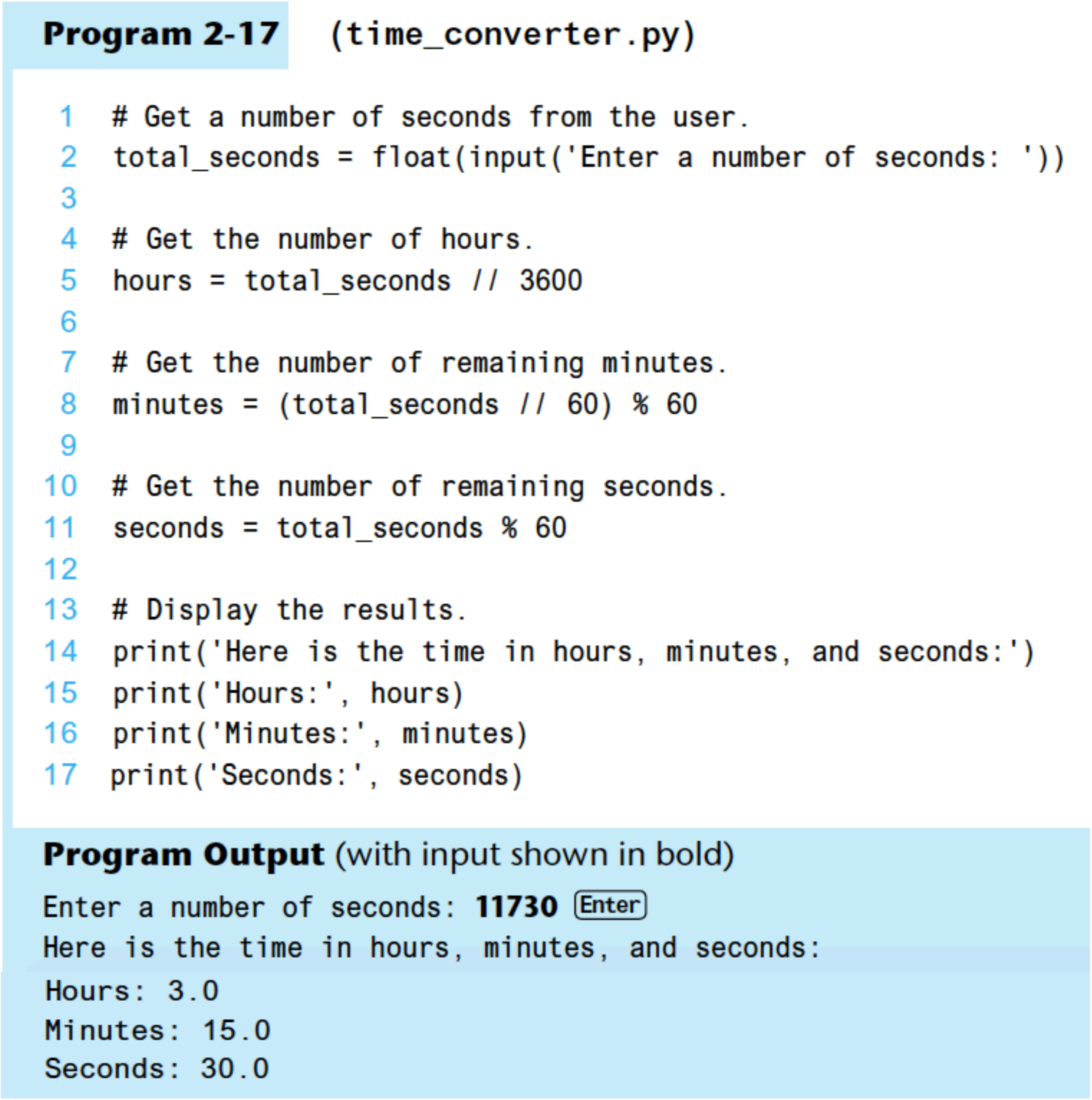
Expression	Value
5 + 2 * 4	13
10 / 2 - 3	2.0
8 + 12 * 2 - 4	28
6 - 3 * 2 + 7 - 1	6



Expression	Value
(5 + 2) * 4	28
10 / (5 - 3)	5.0
8 + 12 * (6 - 2)	56
(6 - 3) * (2 + 7) / 3	9.0

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



Program 2-17 (time_converter.py)

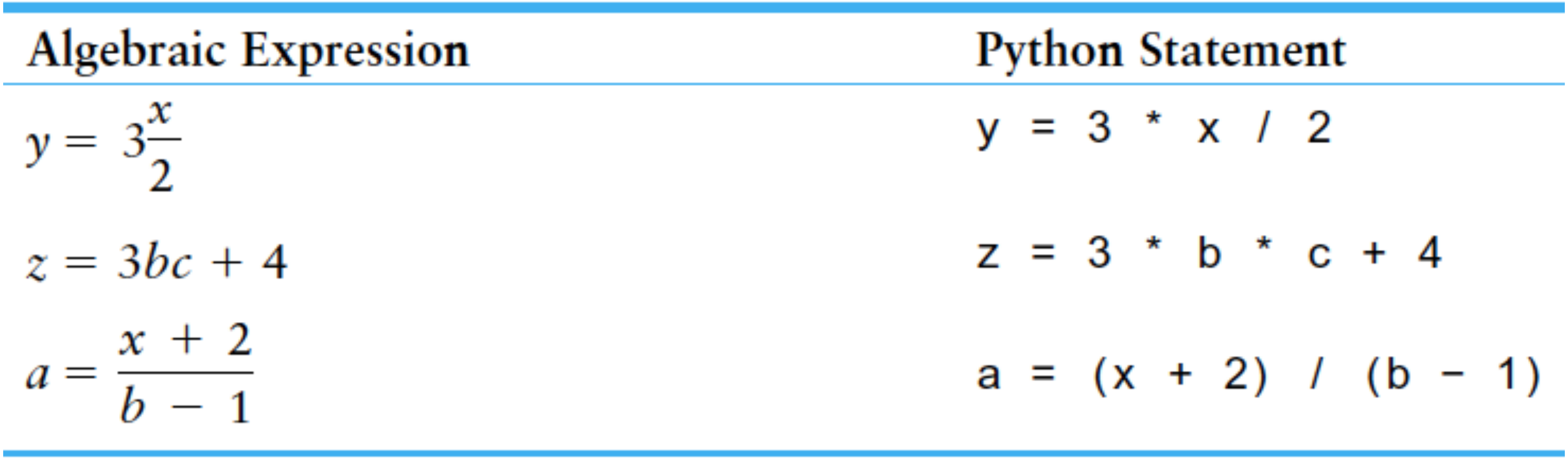
```python
# Get a number of seconds from the user.
total_seconds = float(input('Enter a number of seconds: '))

# 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:', hours)
print('Minutes:', minutes)
print('Seconds:', seconds)
```

Program Output (with input shown in bold)

```
Enter a number of seconds: 11730 Enter
Here is the time in hours, minutes, and seconds:
Hours: 3.0
Minutes: 15.0
Seconds: 30.0
```

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



Algebraic Expression	Python Statement
$y = 3\frac{x}{2}$	`y = 3 * x / 2`
$z = 3bc + 4$	`z = 3 * b * c + 4`
$a = \frac{x + 2}{b - 1}$	`a = (x + 2) / (b - 1)`

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

```
result = var1 * 2 + var2 * 3 + \
         var3 * 4 + var4 * 5
```

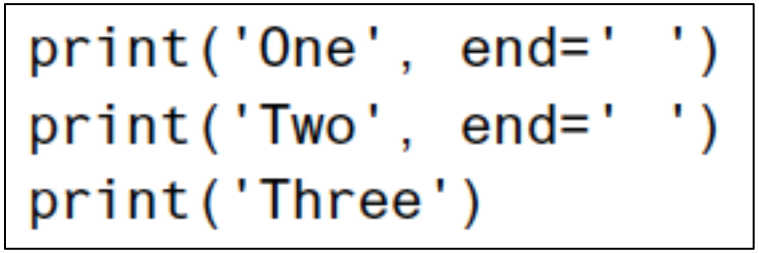
Breaking Long Statements into Multiple Lines

- Any part of a statement that is enclosed in parentheses can be broken without the line continuation character.

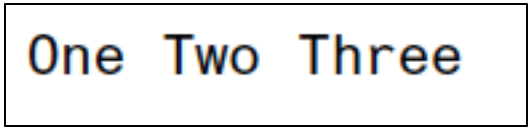
```
print("Monday's sales are", monday,
      "and Tuesday's sales are", tuesday,
      "and Wednesday's sales are", Wednesday)

total = (value1 + value2 +
         value3 + value4 +
         value5 + value6)
```

More About Data Output

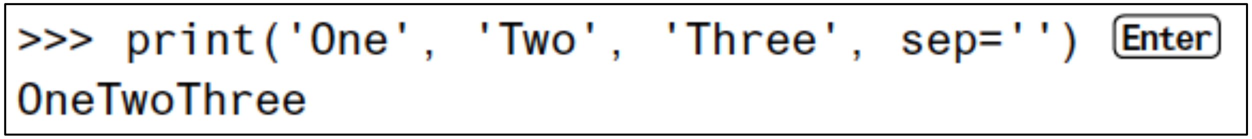
- `print` function displays line of output
  - Newline character at end of printed data
  - Special argument `end='`*`delimiter`*`'` causes `print` to place *`delimiter`* at end of data instead of newline character
- `print` function uses space as item separator
  - Special argument `sep='`*`delimiter`*`'` causes `print` to use *`delimiter`* as item separator



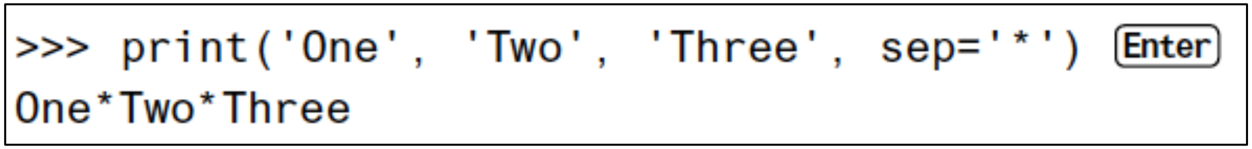
```
print('One', end=' ')
print('Two', end=' ')
print('Three')
```



```
One Two Three
```



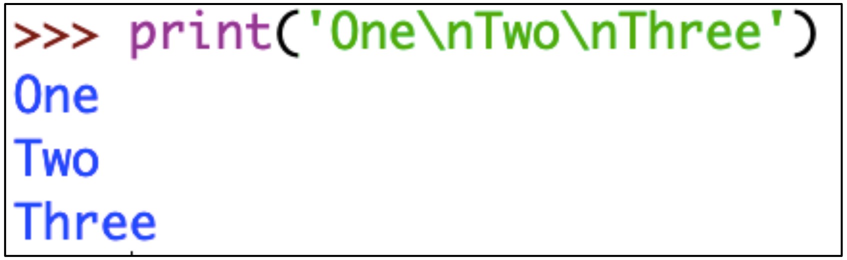
```
>>> print('One', 'Two', 'Three', sep='') Enter
OneTwoThree
```



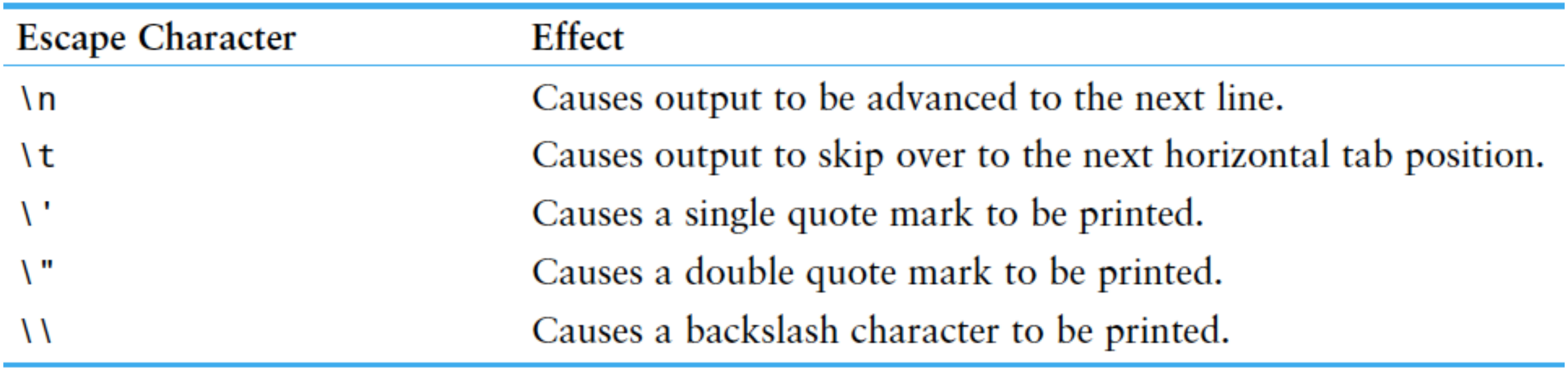
```
>>> print('One', 'Two', 'Three', sep='*') Enter
One*Two*Three
```

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



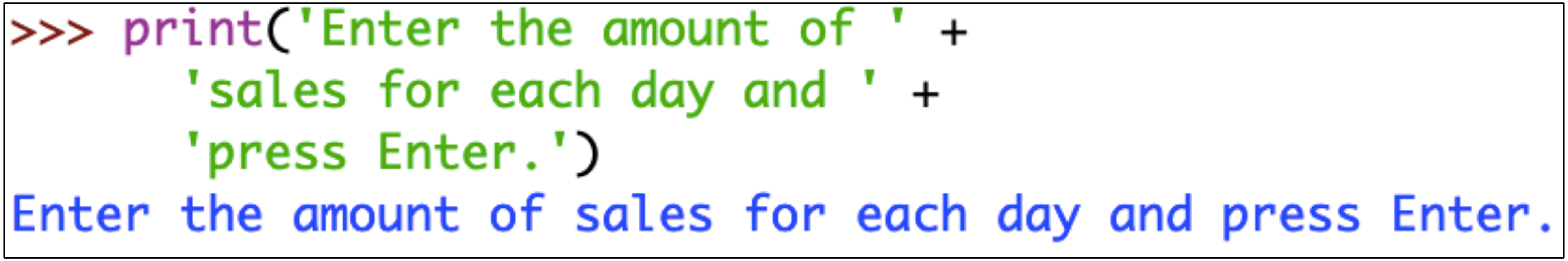
```
>>> print('One\nTwo\nThree')
One
Two
Three
```



Escape Character	Effect
`\n`	Causes output to be advanced to the next line.
`\t`	Causes output to skip over to the next horizontal tab position.
`\'`	Causes a single quote mark to be printed.
`\"`	Causes a double quote mark to be printed.
`\\`	Causes a backslash character to be printed.

More About Data Output (cont’d.)

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

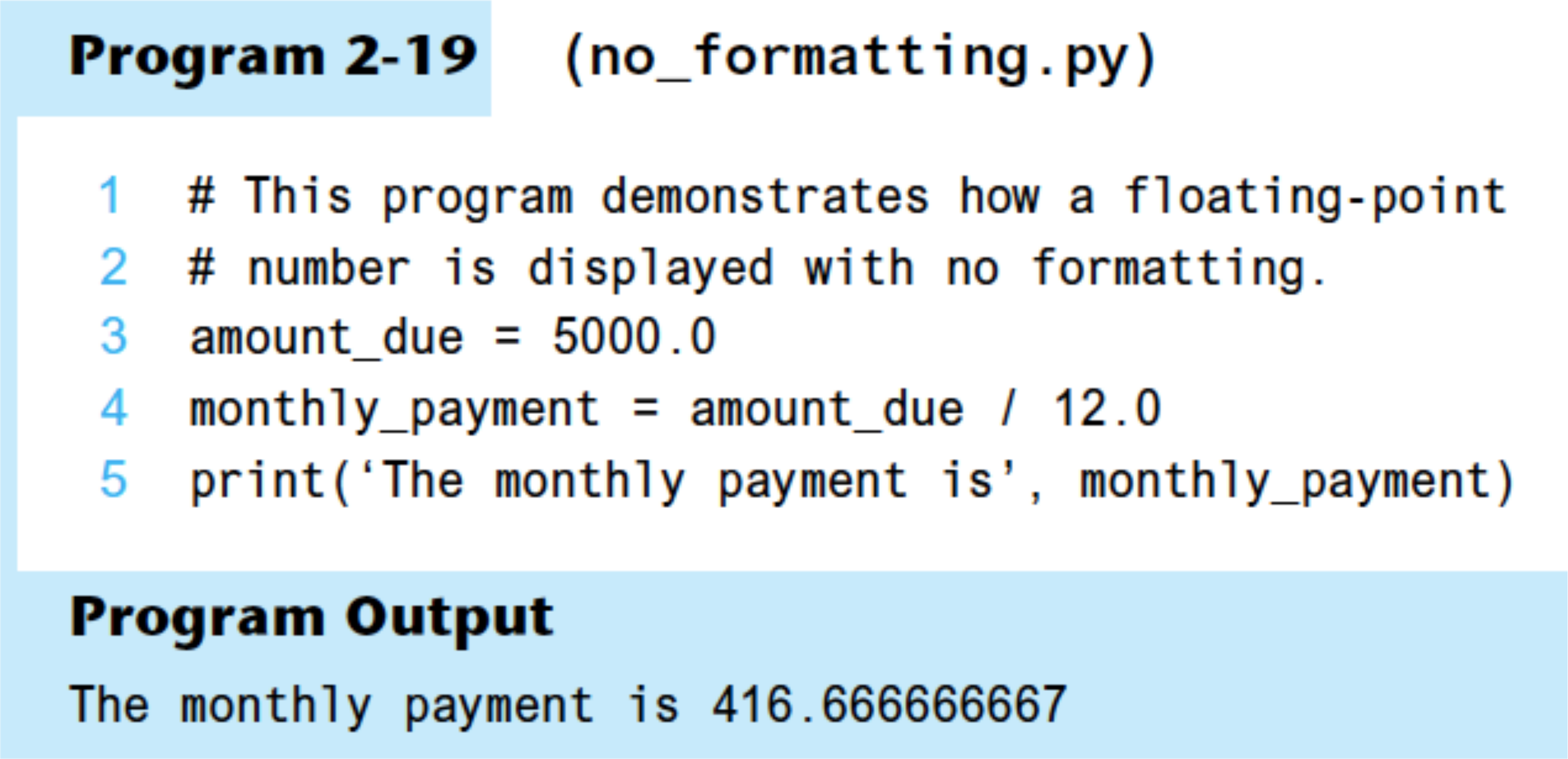


```
>>> print('Enter the amount of ' +
      'sales for each day and ' +
      'press Enter.')
Enter the amount of sales for each day and press Enter.
```

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 comma separators and the minimum field width used to display the value

Example



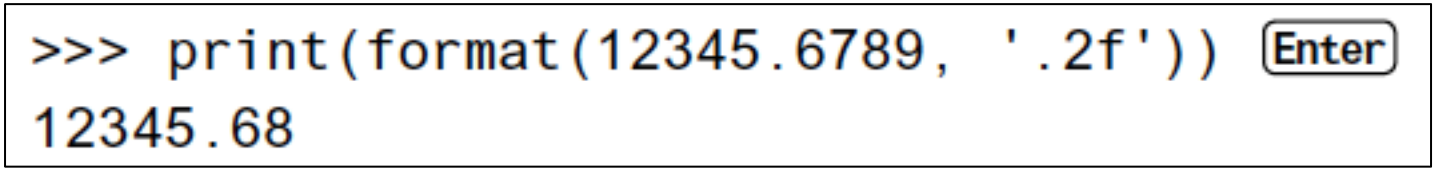
Program 2-19 (no_formatting.py)

```
# This program demonstrates how a floating-point
# number is displayed with no formatting.
amount_due = 5000.0
monthly_payment = amount_due / 12.0
print(‘The monthly payment is’, monthly_payment)
```

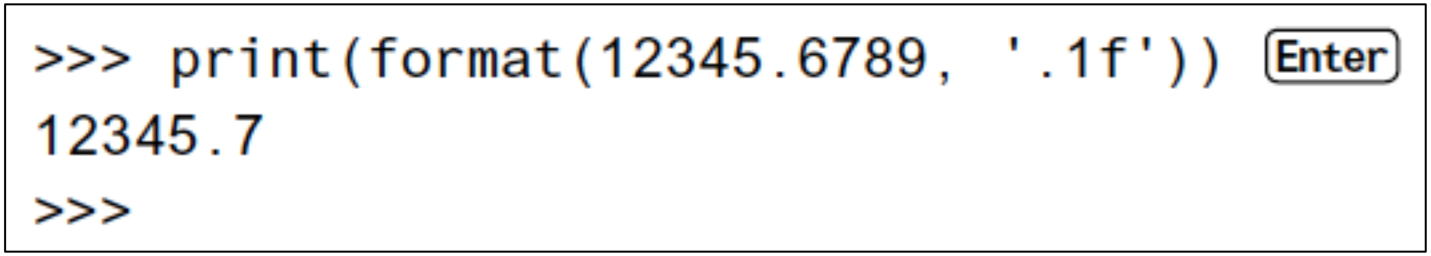
Program Output

```
The monthly payment is 416.666666667
```

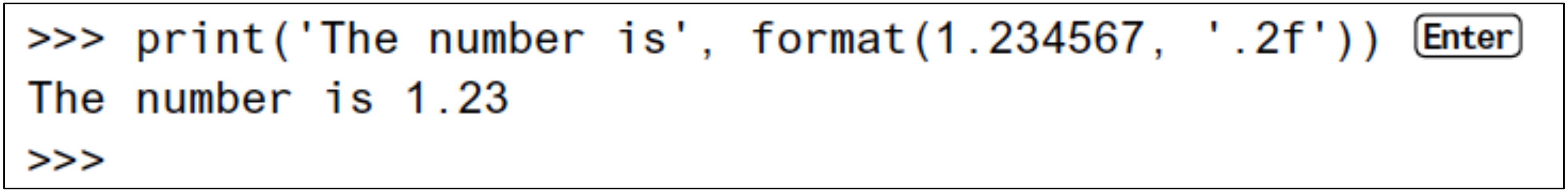
Example



```
>>> print(format(12345.6789, '.2f')) Enter
12345.68
```



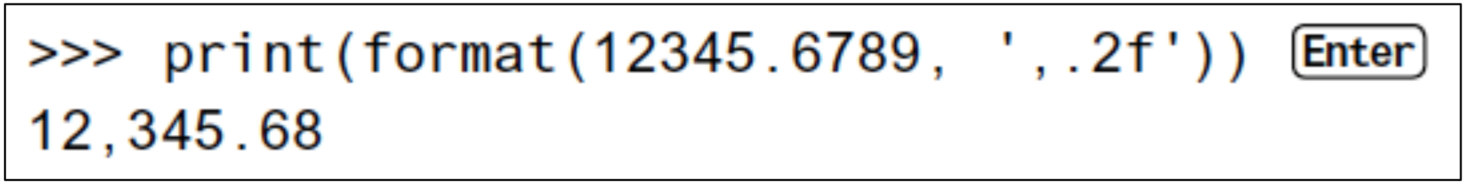
```
>>> print(format(12345.6789, '.1f')) Enter
12345.7
>>>
```



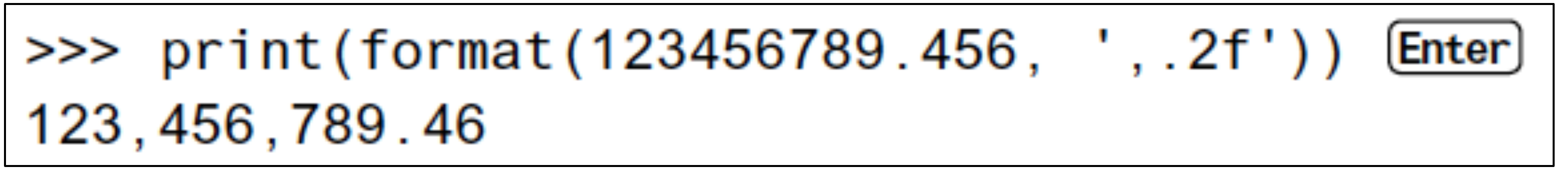
```
>>> print('The number is', format(1.234567, '.2f')) Enter
The number is 1.23
>>>
```

Inserting Comma Separators

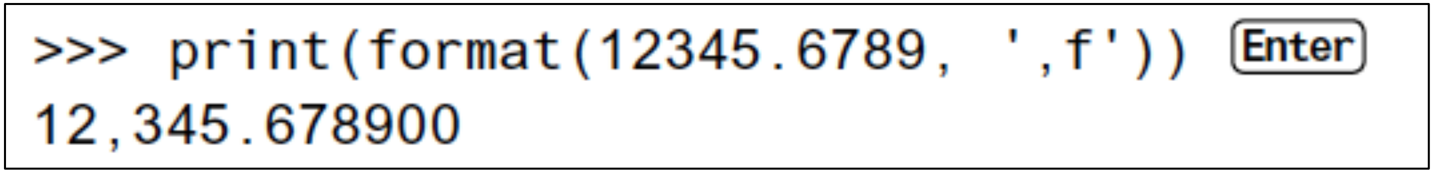
- If you want the number to be formatted with comma separators, you can insert a comma into the format specifier, as shown here:



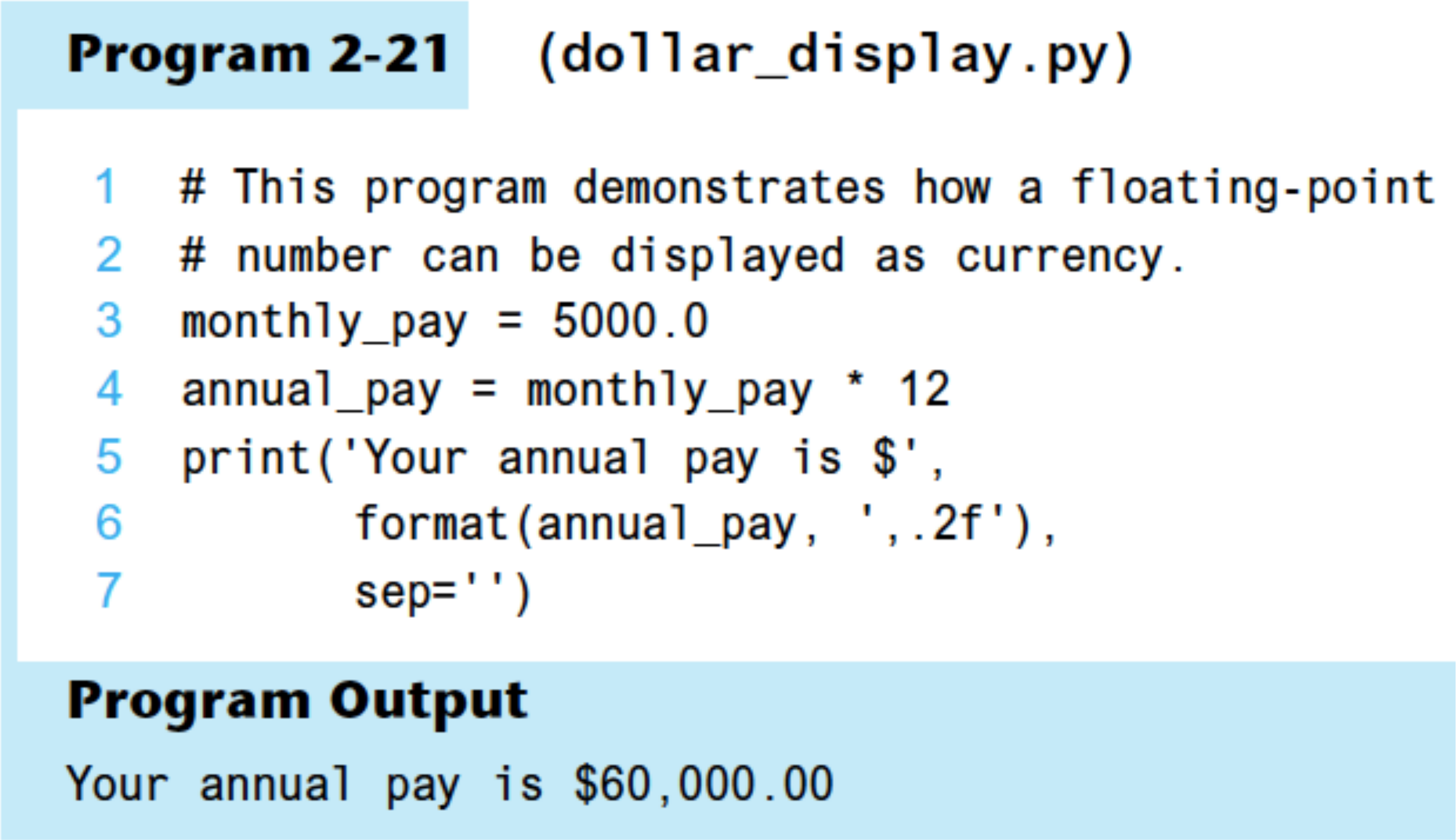
```
>>> print(format(12345.6789, ',.2f')) Enter
12,345.68
```



```
>>> print(format(123456789.456, ',.2f')) Enter
123,456,789.46
```



```
>>> print(format(12345.6789, ',f')) Enter
12,345.678900
```



Program 2-21 (dollar_display.py)

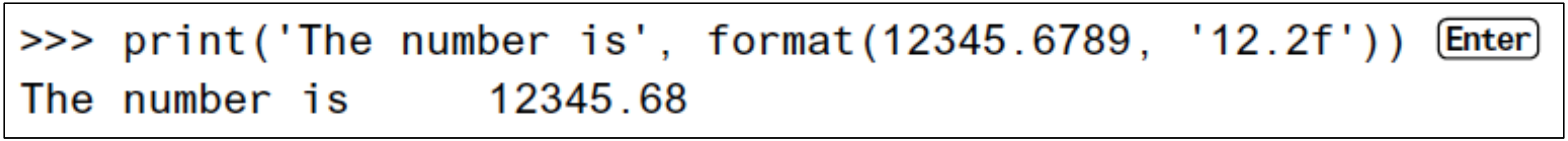
```
# This program demonstrates how a floating-point
# number can be displayed as currency.
monthly_pay = 5000.0
annual_pay = monthly_pay * 12
print('Your annual pay is $',
      format(annual_pay, ',.2f'),
      sep='')
```

Program Output

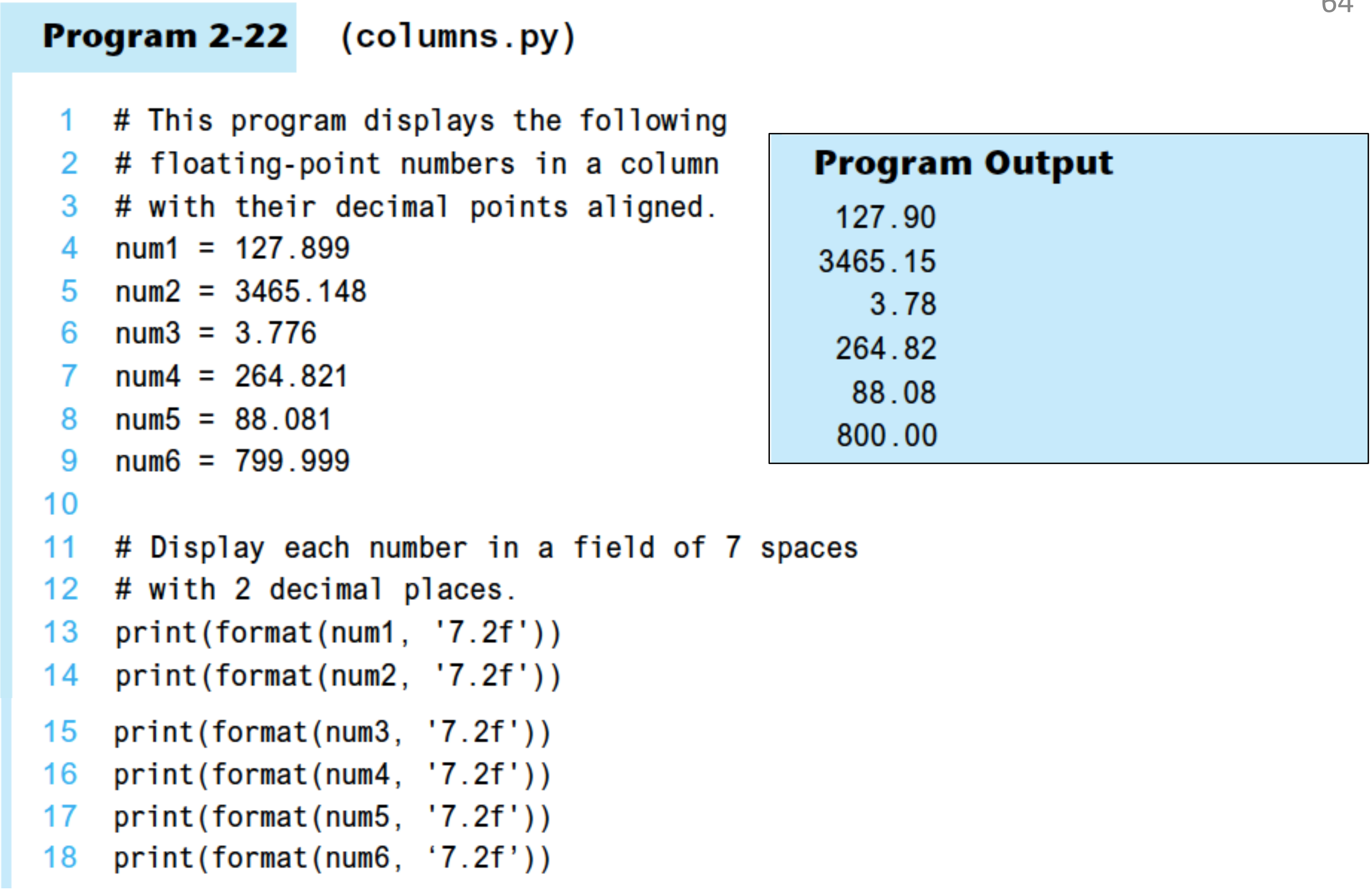
```
Your annual pay is $60,000.00
```

Specifying a Minimum Field Width

- The format specifier can also include a minimum field width, which is the minimum number of spaces that should be used to display the value. The following example prints a number in a field that is 12 spaces wide:



```
>>> print('The number is', format(12345.6789, '12.2f')) Enter
The number is     12345.68
```



Program 2-22 (columns.py)

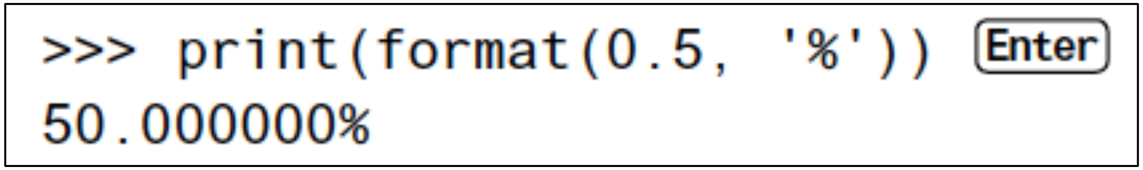
```
# This program displays the following
# floating-point numbers in a column
# with their decimal points aligned.
num1 = 127.899
num2 = 3465.148
num3 = 3.776
num4 = 264.821
num5 = 88.081
num6 = 799.999

# 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'))
```

Program Output

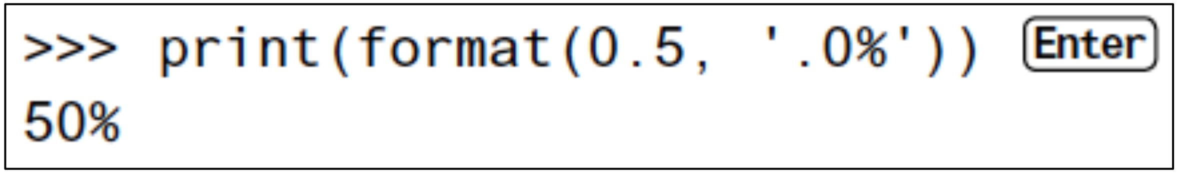
```
 127.90
3465.15
   3.78
 264.82
  88.08
 800.00
```

Formatting a Floating-Point Number as a Percentage

- The `%` symbol can be used in the format string of `format` function to format number as percentage



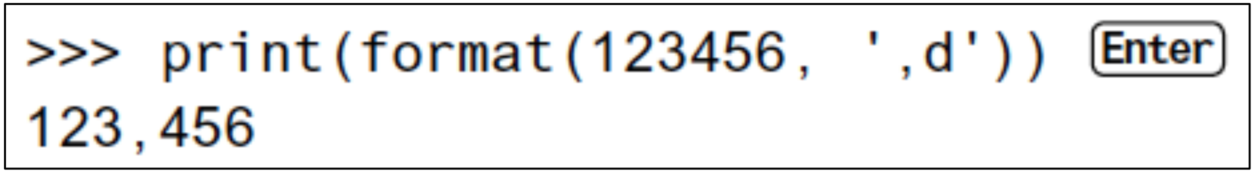
```
>>> print(format(0.5, '%'))
50.000000%
```



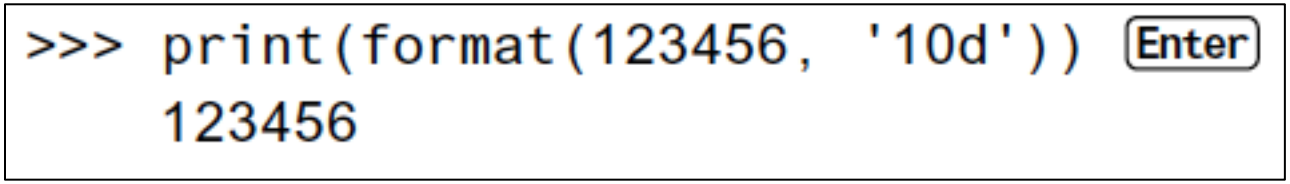
```
>>> print(format(0.5, '.0%'))
50%
```

Formatting Integers

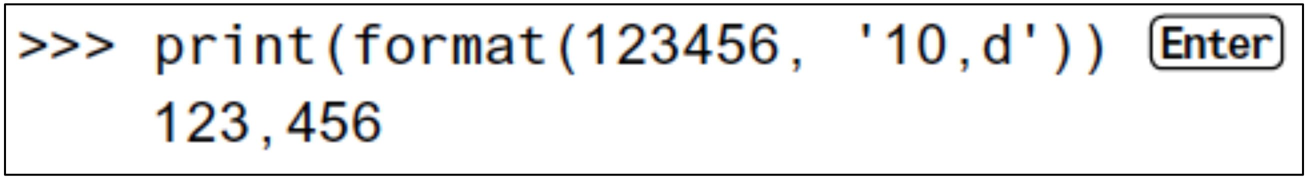
- To format an integer using `format` function:
  - Use `d` as the type designator
  - Do not specify precision
  - Can still use `format` function to set field width or comma separator



```
>>> print(format(123456, ',d')) Enter
123,456
```



```
>>> print(format(123456, '10d')) Enter
    123456
```



```
>>> print(format(123456, '10,d')) Enter
    123,456
```

Magic Numbers

- A magic number is an unexplained numeric value that appears in a program's code. Example:

```
amount = balance * 0.069
```

- What is the value 0.069? An interest rate? A fee percentage? Only the person who wrote the code knows for sure.

The Problem with Magic Numbers

- It can be difficult to determine the purpose of the number.

- If the magic number is used in multiple places in the program, it can take a lot of effort to change the number in each location, should the need arise.

- You take the risk of making a mistake each time you type the magic number in the program's code.
  - For example, suppose you intend to type 0.069, but you accidentally type .0069. This mistake will cause mathematical errors that can be difficult to find.

Named Constants

- You should use named constants instead of magic numbers.
- A named constant is a name that represents a value that does not change during the program's execution.
- Example:

```
INTEREST_RATE = 0.069
```

- This creates a named constant named `INTEREST_RATE`, assigned the value 0.069. It can be used instead of the magic number:

```
amount = balance * INTEREST_RATE
```

Advantages of Using Named Constants

- Named constants make code self-explanatory (self-documenting)
- Named constants make code easier to maintain (change the value assigned to the constant, and the new value takes effect everywhere the constant is used)
- Named constants help prevent typographical errors that are common when using magic numbers

Python String Formatting

- Python supports multiple ways to format text strings. For example, %-formatting, str.format(), and f-Strings.

%-formatting

- Strings in Python have a unique built-in operation that can be accessed with the % operator.

Program s–1

```
name = "Eric"
print("Hello, %s." % name)
```

Program Output

```
Hello, Eric.
```

%-formatting (cont'd)

Program s–2

```
name = "Eric"
age = 74
print("Hello, %s. You are %s." % (name, age))
```

Program Output

```
Hello, Eric. You are 74.
```

%-formatting (cont’d)

Program s-3

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print("Hello, %s %s. You are %s. You are a %s.
       You were a member of %s." %
      (first_name, last_name, age, profession, affiliation))
```

Program Output

```
Hello, Eric Idle. You are 74. You are a comedian. You
were a member of Monty Python.
```

str.format()

- str.format() was introduced in Python 2.6. With str.format(), the replacement fields are marked by curly braces

Program s-4

```
name = "Eric"
age = 74
print("Hello, {}. You are {}.".format(name, age))
```

Program Output

```
Hello, Eric. You are 74.
```

str.format() (cont’d)

- You can reference variables in any order by referencing their index.

Program s–5

```
name = "Eric"
age = 74
print("Hello, {1}. You are {0}.".format(age, name))
```

Program Output

```
Hello, Eric. You are 74.
```

str.format() (cont’d)

Program s-6

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print(("Hello, {first_name} {last_name}. You are {age}. " +
"You are a {profession}. You were a member of {affiliation}.") \
.format(first_name=first_name, last_name=last_name, age=age, \
profession=profession, affiliation=affiliation))
```

Program Output

```
Hello, Eric Idle. You are 74. You are a comedian. You were
a member of Monty Python.
```

Python 3's f-Strings

- Python 3.6 added a new string formatting approach called formatted string literals or “f-strings”.
- Also called “formatted string literals,” f-strings are string literals that have an f at the beginning and curly braces containing expressions that will be replaced with their values.

f-Strings

Program s–7

```
name = "Eric"
age = 74
print(f"Hello, {name}. You are {age}.")
```

Program Output

```
Hello, Eric. You are 74.
```

f-Strings (cont’d)

Program s–8

```
first_name = "Eric"
last_name = "Idle"
age = 74
profession = "comedian"
affiliation = "Monty Python"
print(f"Hello, {first_name} {last_name}. You are {age}. " +
      f"You are a {profession}. " +
      f"You were a member of {affiliation}.")
```

Program Output

```
Hello, Eric Idle. You are 74. You are a comedian. You were
a member of Monty Python.
```

f-Strings (cont’d)

Program s–9

```
name = "eric"
sentence = f'{name.title()} is funny.'
print(sentence)
```

Program Output

```
Eric is funny.
```

f-Strings (cont’d)

Program s−10

```
x = 3.14159265
print(f'PI = {x:.2f}')
```

Program Output

```
PI = 3.14
```

f-Strings (cont’d)

Program s−11

```
x = 12345.6789
print(f'x = {x:,.2f}')
```

Program Output

```
x = 12,345.68
```

f-Strings (cont’d)

Program s-12

```
s1 = 'ab'
s2 = 'abc'
s3 = 'abcd'
s4 = 'abcde'
print(f'{s1:10}')
print(f'{s2:<10}')
print(f'{s3:^10}')
print(f'{s4:>10}')
```

Program Output

```
ab
abc
   abcd
     abcde
```

f-Strings (cont’d)

Program s–13

```
a = 5
b = 10
print(f'Five plus ten is {a + b} and not {2 * (a + b)}.')
```

Program Output

```
Five plus ten is 15 and not 30.
```

Summary

- This chapter covered:
  - The program development cycle, tools for program design, and the design process
  - Ways in which programs can receive input, particularly from the keyboard
  - Ways in which programs can present and format output
  - Use of comments in programs
  - Uses of variables and named constants
  - Tools for performing calculations in programs