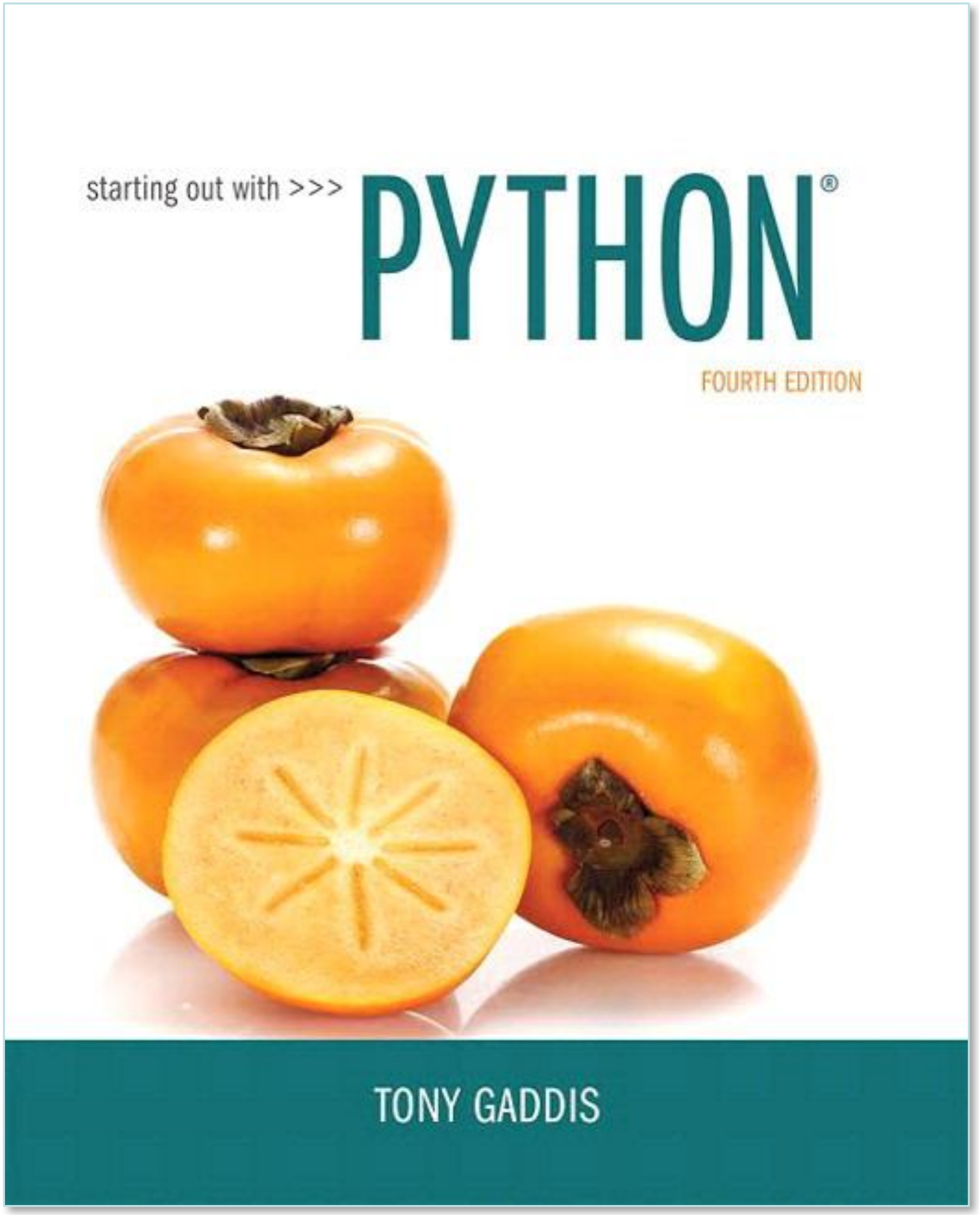
psb ACADEMY

SCHOOL OF ENGINEERING & TECHNOLOGY

CHAPTER 2

Input, Processing, and Output

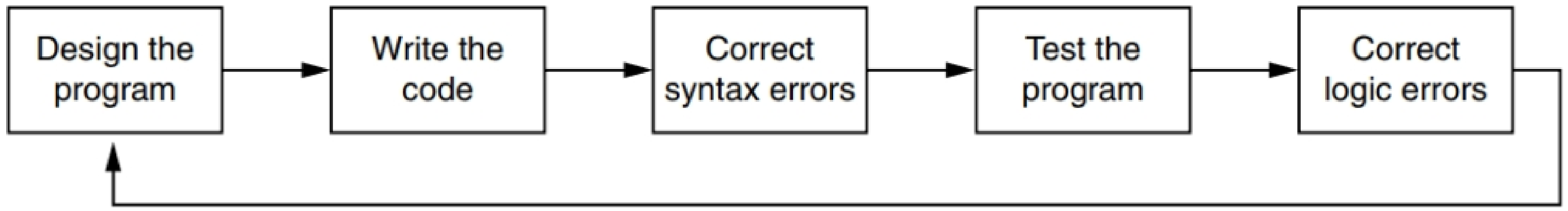


Topics

- 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
- Introduction to Turtle Graphics

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
- **<u>Algorithm</u>: set of well-defined logical steps that must be taken to perform a task**

Designing a Program (cont’d.)

Algorithm examples:

Task: Write a program to determine how to boil water?

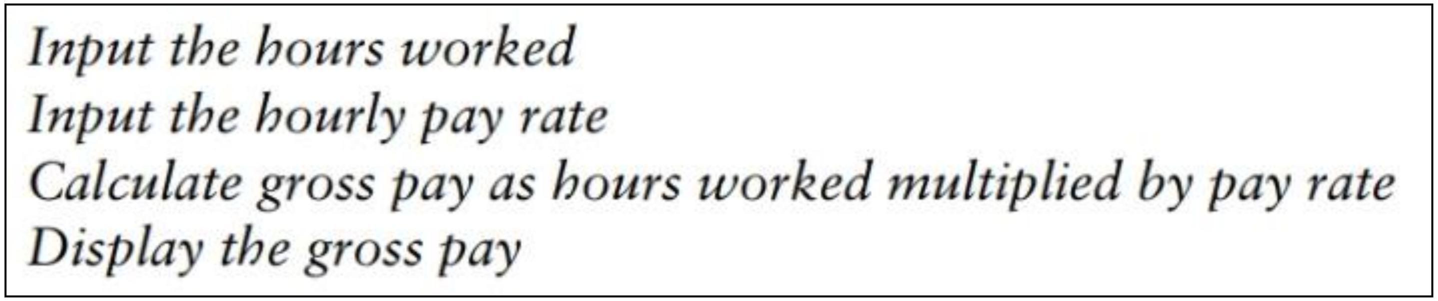
1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

Task: Write a program to calculate and display the gross pay for an hourly paid employee.

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in step 3.

Pseudocode

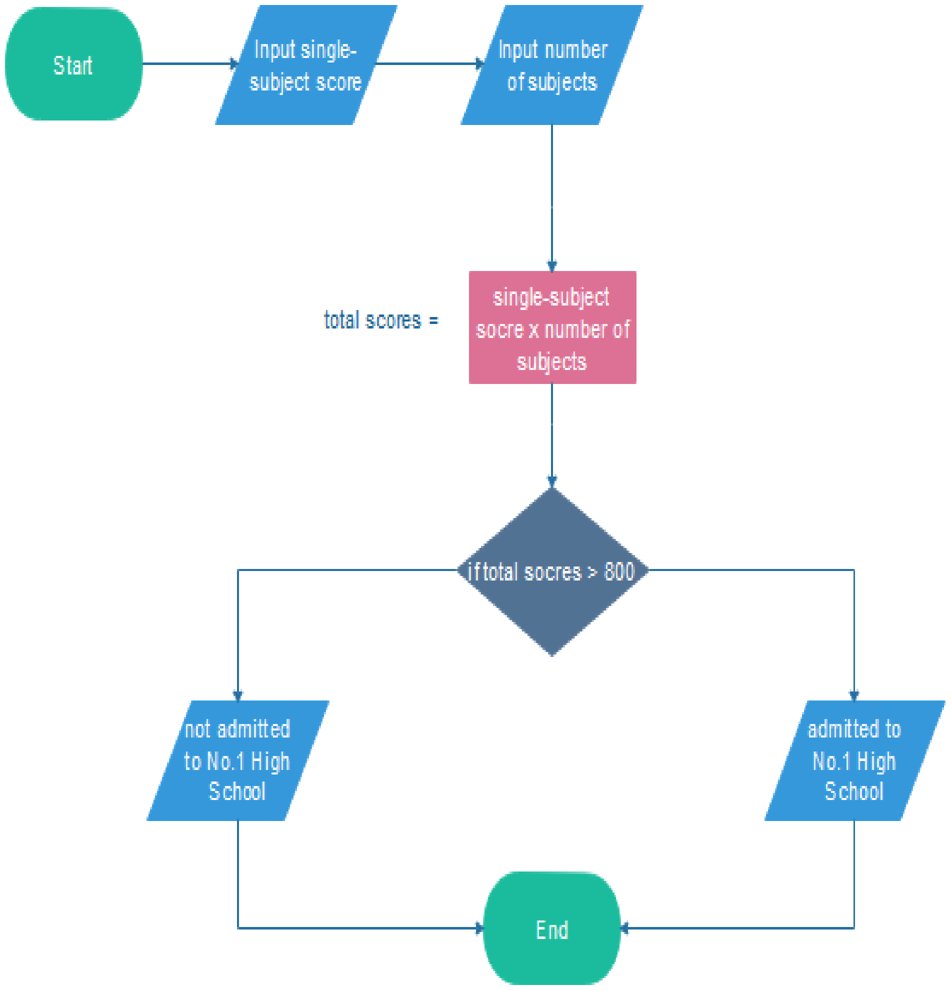
- **<u>Pseudocode</u>: fake code**
  - Informal language that has no syntax rule
  - Not meant to be compiled or executed
  - Used to create model program
    - No need to worry about syntax errors, can focus on program's design
    - Can be translated directly into actual code in any programming language



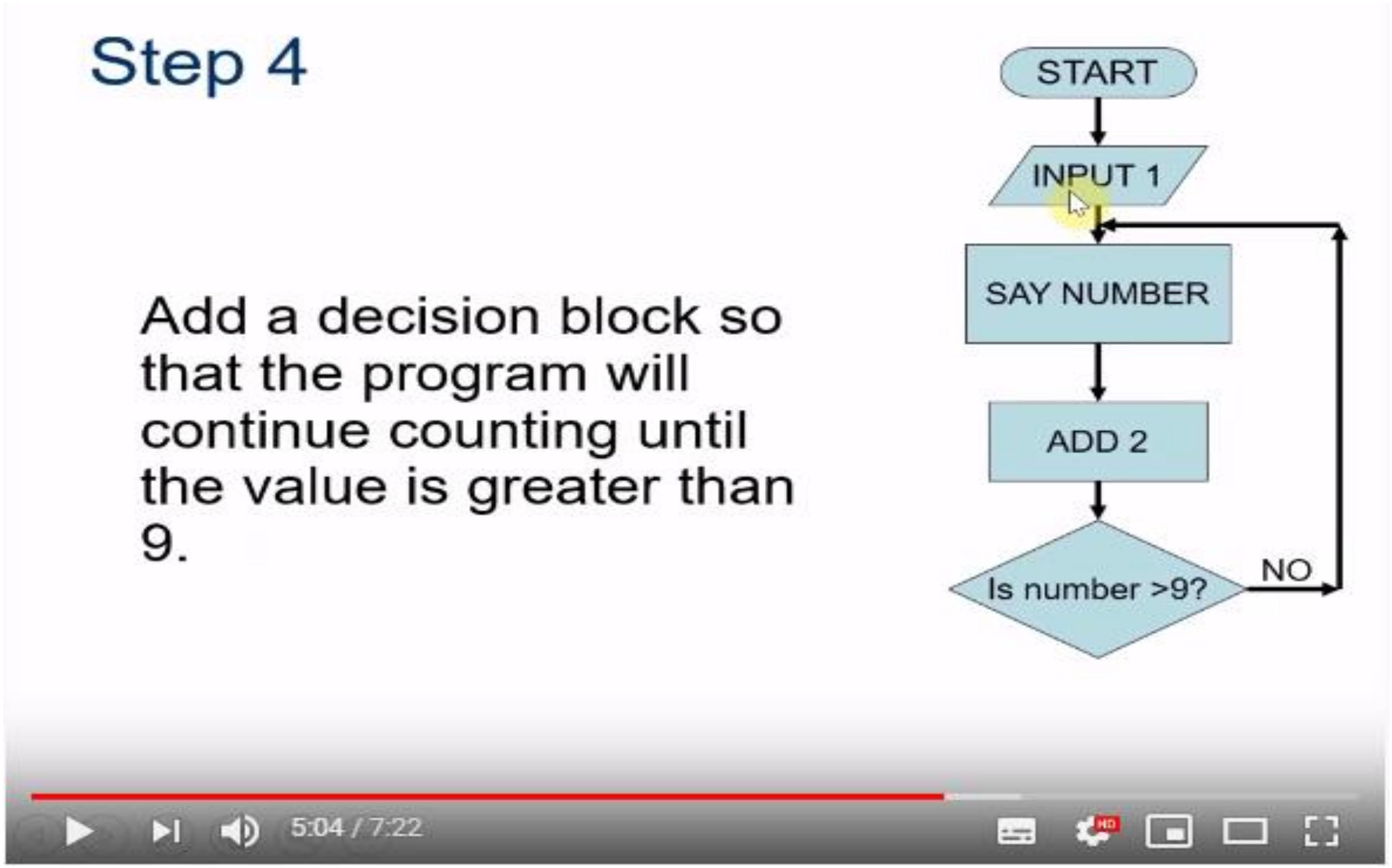
```
Input the hours worked
Input the hourly pay rate
Calculate gross pay as hours worked multiplied by pay rate
Display the gross pay
```

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
  - Diamonds are decisions points
  - Symbols are connected by arrows that represent the flow of the program


Start
Input single-subject score
Input number of subjects
total scores =
single-subject socre x number of subjects
if total socres > 800
not admitted to No.1 High School
admitted to No.1 High School
End

Demo

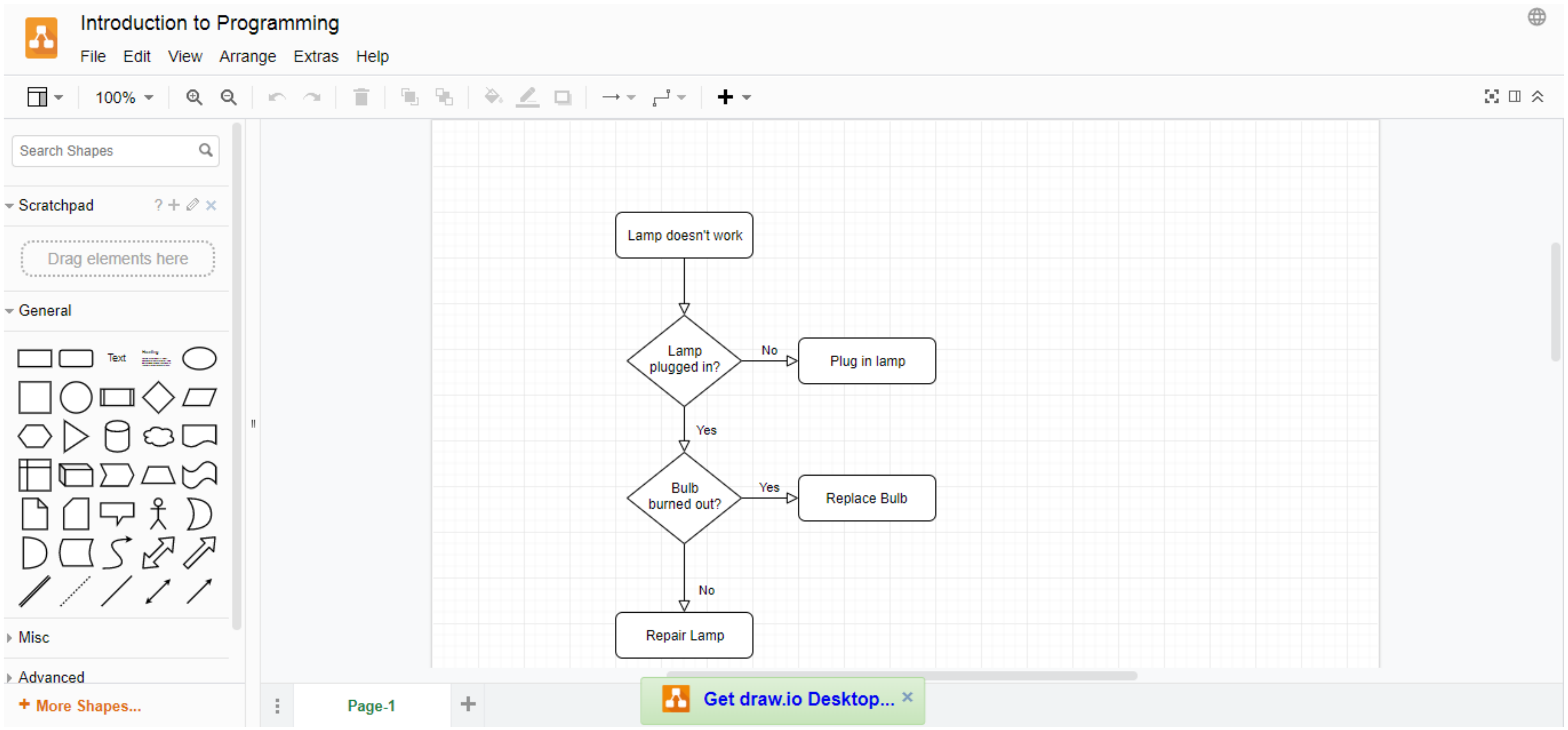

Step 4

Add a decision block so that the program will continue counting until the value is greater than 9.

https://www.youtube.com/watch?v=SWRDqTx8d4k

Flowcharts Development Tool

- https://www.draw.io/

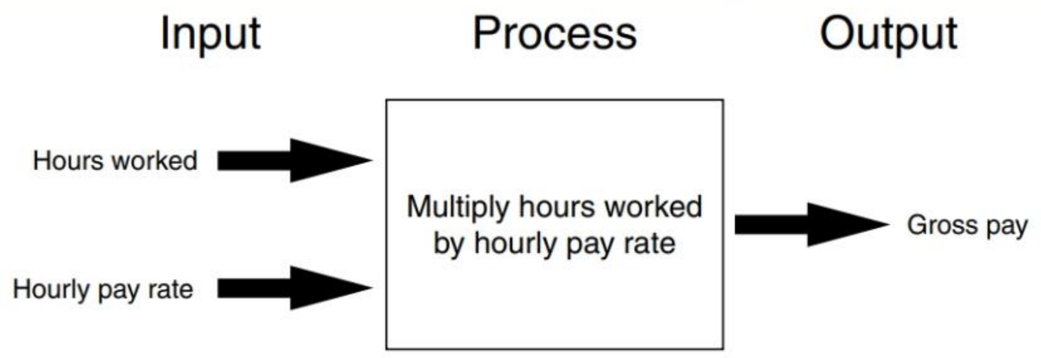


$$\begin{tikzcd}
\text{Lamp doesn't work} \arrow[d] & \\
\text{Lamp plugged in?} \arrow[r, "\text{No}"] \arrow[d, "\text{Yes}"] & \text{Plug in lamp} \\
\text{Bulb burned out?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{Replace Bulb} \\
\text{Repair Lamp} &
\end{tikzcd}$$

Input, Processing, and Output

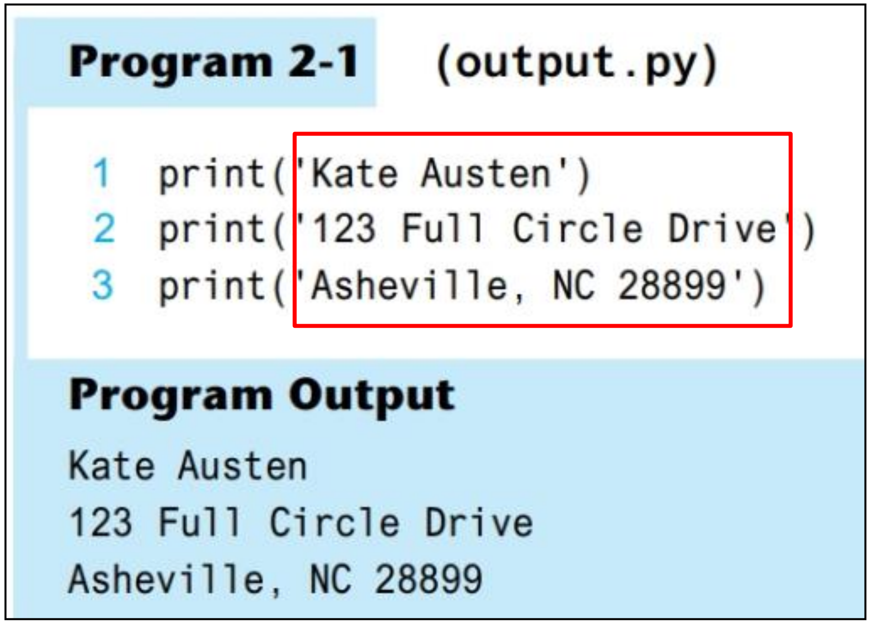
- **Typically, computer performs three-step process**
  - Receive input
    - Input: any data that the program receives while it is running
  - Perform some process on the input
    - Example: mathematical calculation
  - Produce output

Figure 2-3 The input, processing, and output of the pay calculating program



Displaying Output with the `print` Function

- **<u>Function</u>: piece of prewritten code that performs an operation**
- **<u>`print` function</u>: displays output on the screen**
- **<u>Argument</u>: 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



Program 2-1 (output.py)

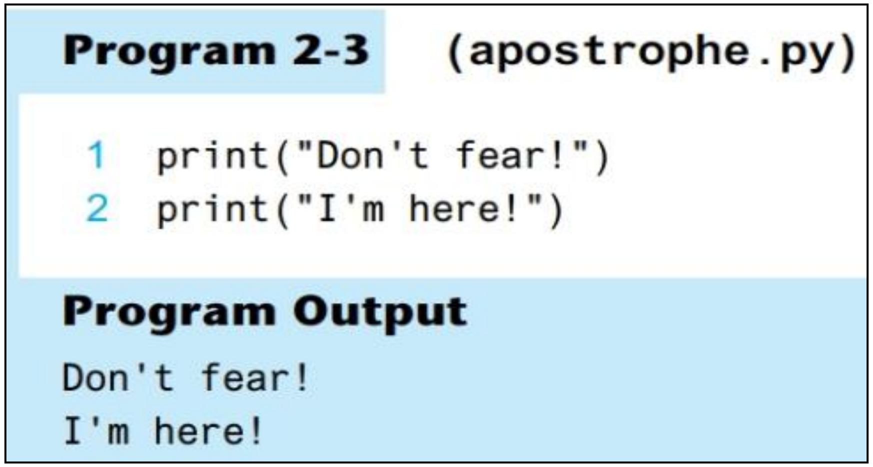
```
print('Kate Austen')
print('123 Full Circle Drive')
print('Asheville, NC 28899')
```

Program Output

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

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

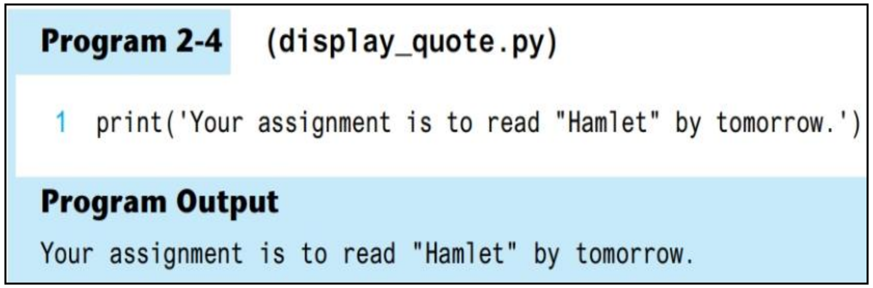


Program 2-3 (apostrophe.py)

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

Program Output

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

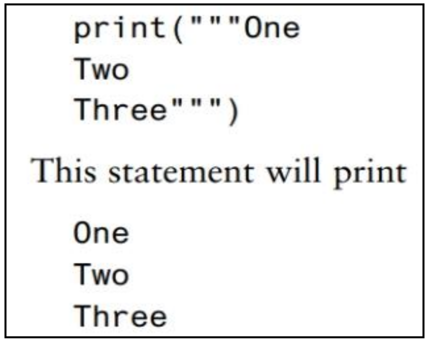


Program 2-4 (display_quote.py)

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

Program Output

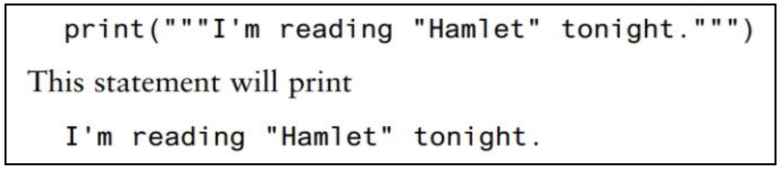
```
Your assignment is to read "Hamlet" by tomorrow.
```



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

This statement will print

```
One
Two
Three
```



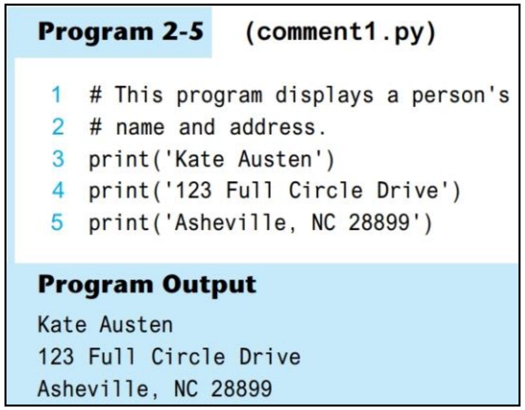
```
print("""I'm reading "Hamlet" tonight.""")
```

This statement will print

```
I'm reading "Hamlet" tonight.
```

Comments

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

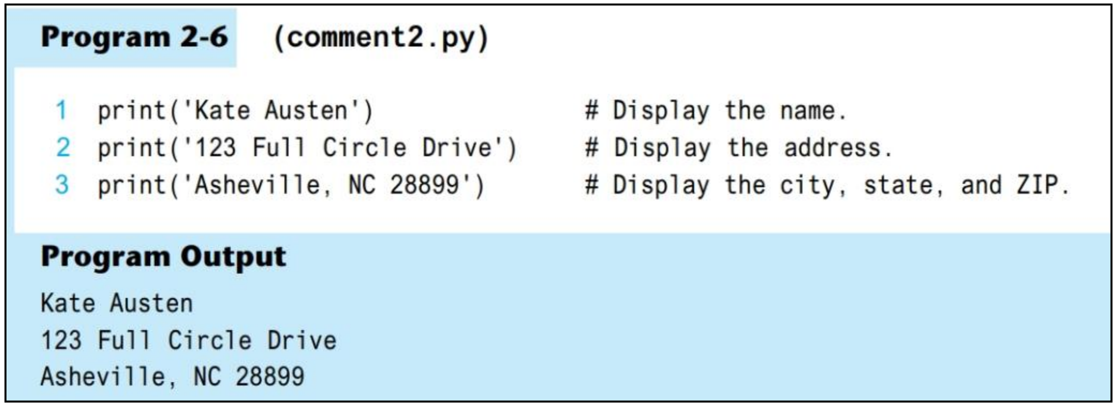


Program 2-5 (comment1.py)

```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
```



Program 2-6 (comment2.py)

```python
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

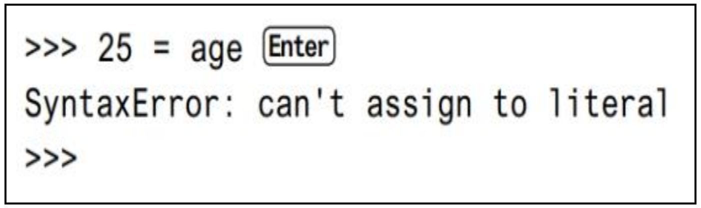
```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Variables

- **<u>Variable</u>: 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
- **<u>Assignment statement</u>: used to create a variable and make it reference data**
  - General format is `variable = expression`
    - Example: `age = 29`
    - <u>Assignment operator</u>: the equal sign (=)

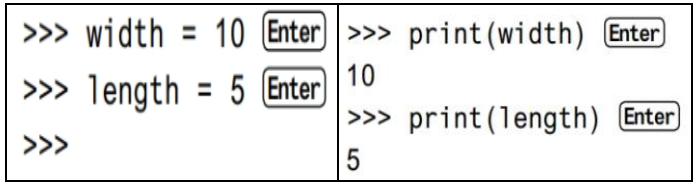
Variables (cont'd.)

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



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

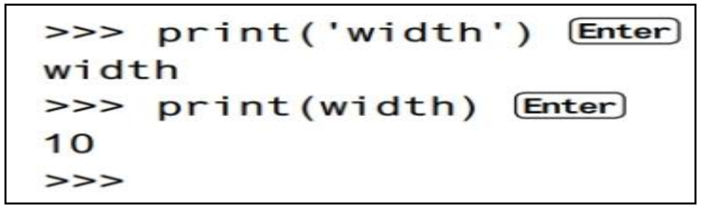
- A variable can be passed as an argument to a function



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

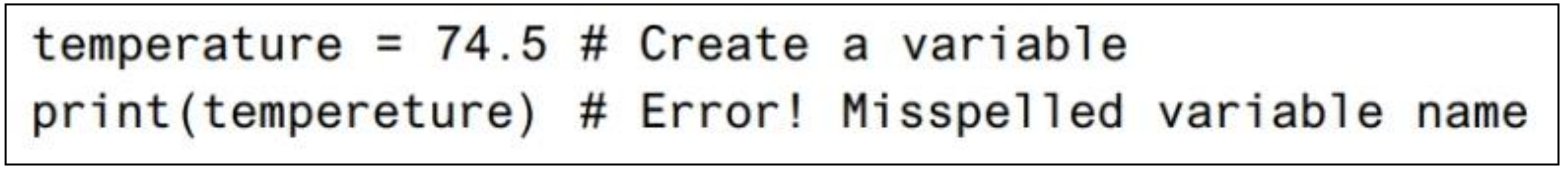
```
>>> print(width) Enter
10
>>> print(length) Enter
5
```

- Variable name should not be enclosed in quote marks



```
>>> print('width') Enter
width
>>> print(width) Enter
10
>>>
```

- You can only use a variable if a value is assigned to it



```
temperature = 74.5 # Create a variable
print(tempereture) # Error! Misspelled variable name
```

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**

```
grosspay
payrate
hotdogssoldtoday
```

Acceptable but not easily readable

```
gross_pay
pay_rate
hot_dogs_sold_today
```

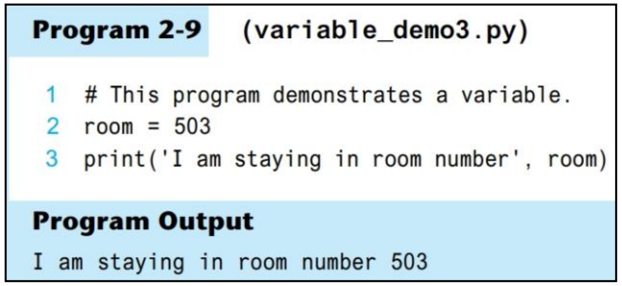
Much more readable convention used by many Python programmers

```
grossPay
payRate
hotDogsSoldToday
```

Another popular style (camelCase convention)

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

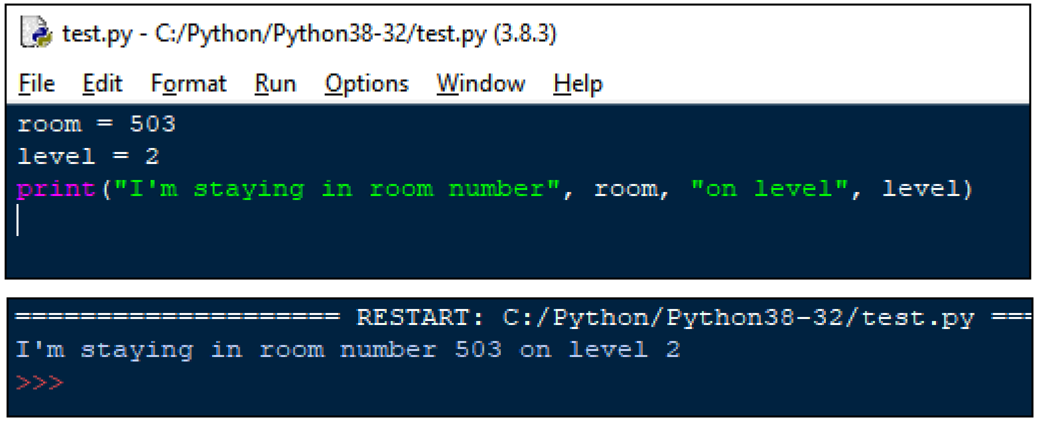


Program 2-9 (variable_demo3.py)

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

Program Output

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


test.py - C:/Python/Python38-32/test.py (3.8.3)
File Edit Format Run Options Window Help

```
room = 503
level = 2
print("I'm staying in room number", room, "on level", level)
```

```
==================== RESTART: C:/Python/Python38-32/test.py ===
I'm staying in room number 503 on level 2
>>>
```

Variable Reassignment

- **Variables can reference different values while program is running**
- **<u>Garbage collection</u>: removal of values that are no longer referenced by variables**
  - Carried out by Python interpreter

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!')
```

Program Output

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

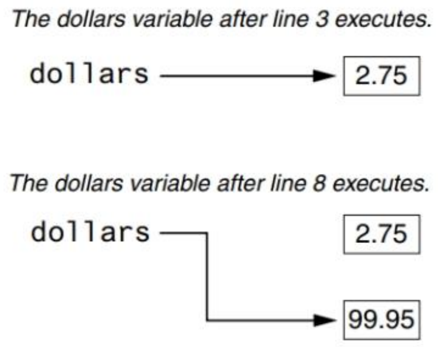
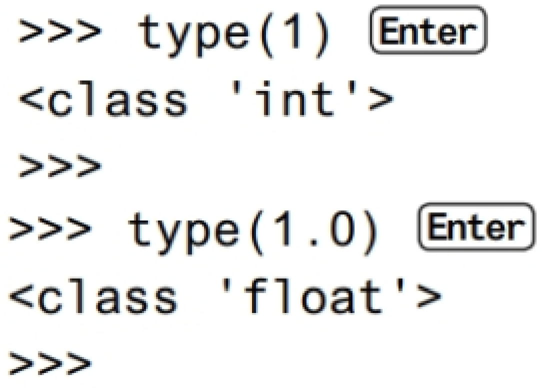

The dollars variable after line 3 executes.
dollars
2.75
The dollars variable after line 8 executes.
dollars
2.75
99.95

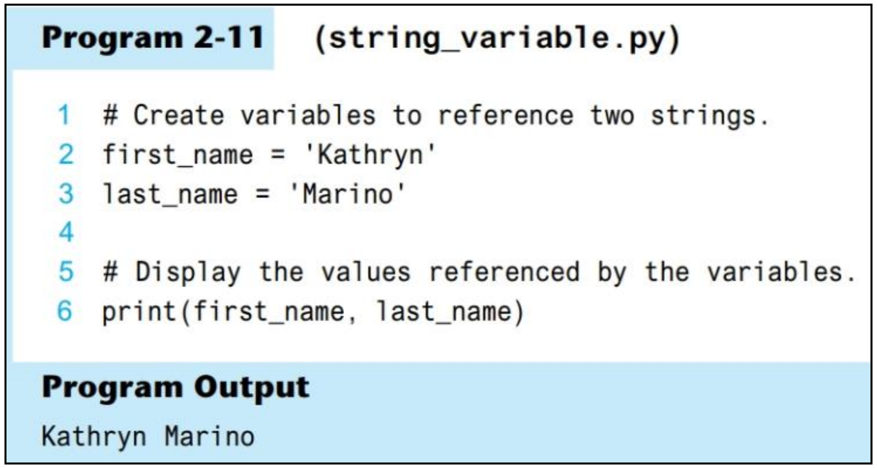
Figure 2-6 Variable reassignment in Program 2-10

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

- **<u>Data types</u>: categorize value in memory**
  - e.g., int for integer, float for real number, str used for storing strings in memory
- **<u>Numeric literal</u>: 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'>
>>>
```



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.
>>>
```

Figure 2-7 The variable x references an integer

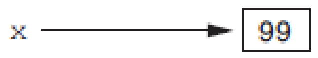
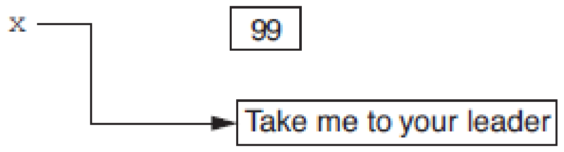
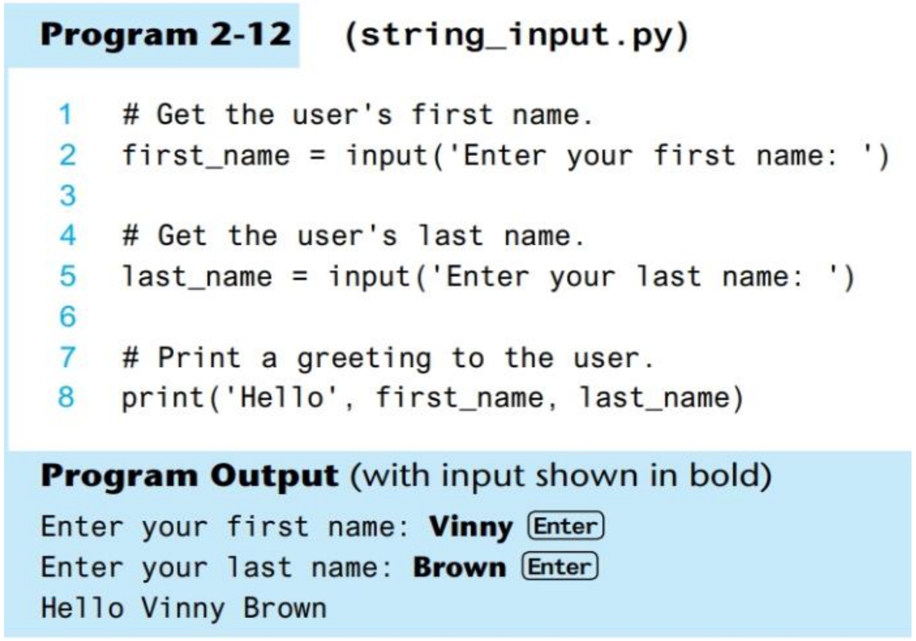


Figure 2-8 The variable x references a string



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



Program 2-12 (string_input.py)

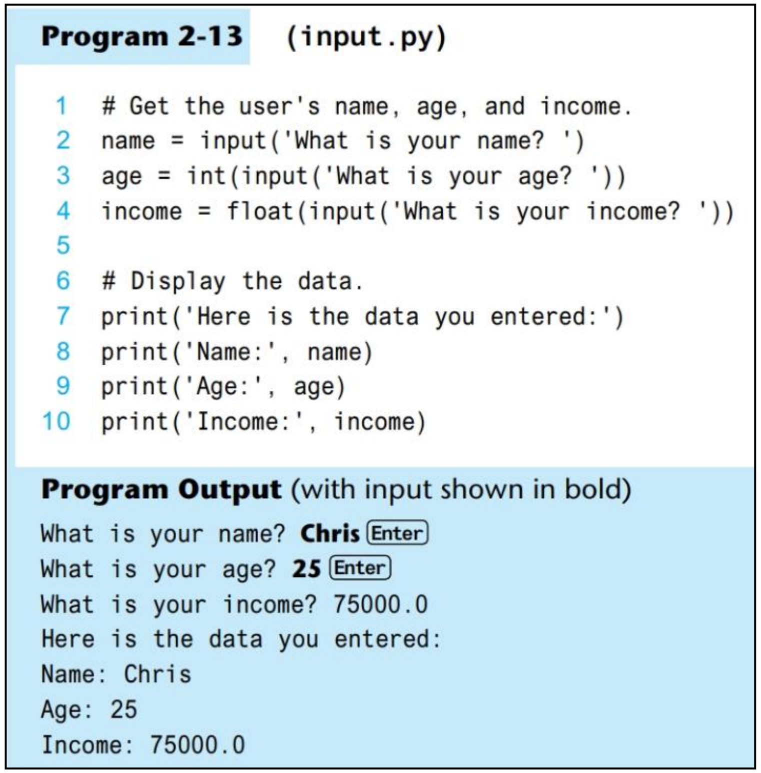
```
# Get the user's first name.
first_name = input('Enter your first name: ')

# 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

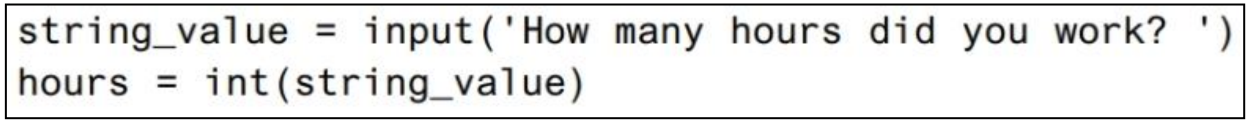
- **`input` function always returns a string**
- **Built-in functions convert between data types**
  - `int(`*`item`*`)` converts *`item`* to an `int`
  - `float(`*`item`*`)` converts *`item`* to a `float`
- Nested function call: general format: *`function1`*`(`*`function2`*`(`*`argument`*`))`
  - value returned by function2 is passed to function1
  - creates one less variable and does the same job


Program 2-13 (input.py)

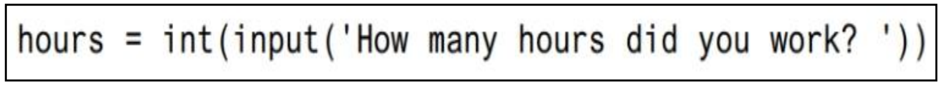
```
# Get the user's name, age, and income.
name = input('What is your name? ')
age = int(input('What is your age? '))
income = float(input('What is your income? '))

# 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
Here is the data you entered:
Name: Chris
Age: 25
Income: 75000.0
```



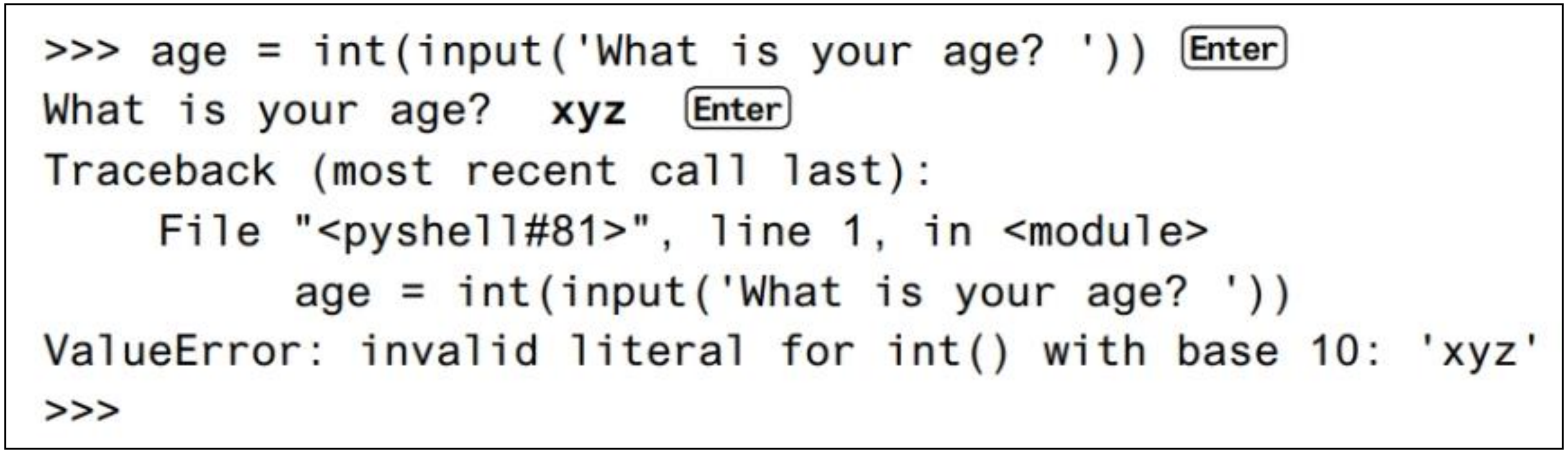
```
string_value = input('How many hours did you work? ')
hours = int(string_value)
```



```
hours = int(input('How many hours did you work? '))
```

Reading Numbers with the `input` Function

- **Type conversion**: Only works if item is valid numeric value, otherwise, throws exception



```
>>> age = int(input('What is your age? ')) Enter
What is your age?  xyz  Enter
Traceback (most recent call last):
    File "<pyshell#81>", line 1, in <module>
        age = int(input('What is your age? '))
ValueError: invalid literal for int() with base 10: 'xyz'
>>>
```

Be careful when using nested function call especially with input()!

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

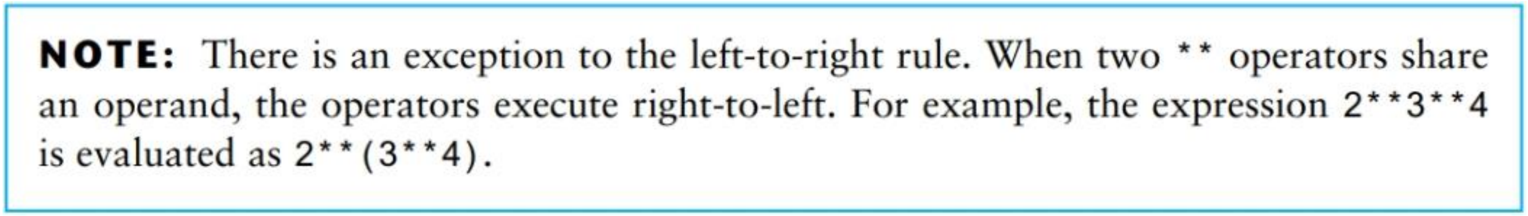
```
>>> 5 / 2 Enter
2.5
>>>
```

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

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

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



NOTE: There is an exception to the left-to-right rule. When two `**` operators share an operand, the operators execute right-to-left. For example, the expression `2**3**4` is evaluated as `2**(3**4)`.

The Exponent Operator and the Remainder (Modulus) Operator

- **<u>Exponent operator (`**`)</u>: 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

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

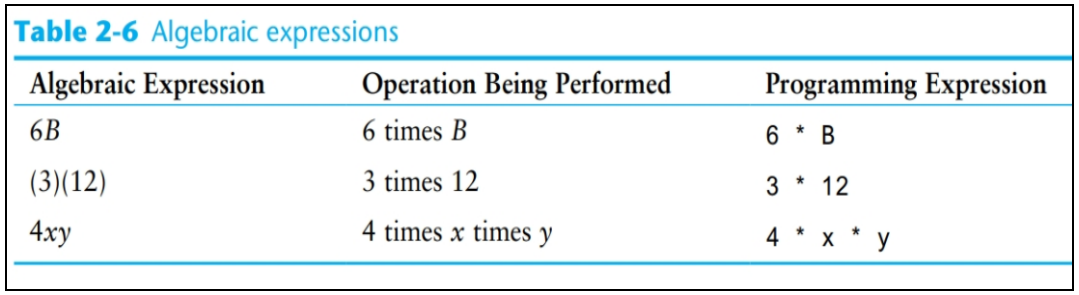


Table 2-6 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
$(3)(12)$	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

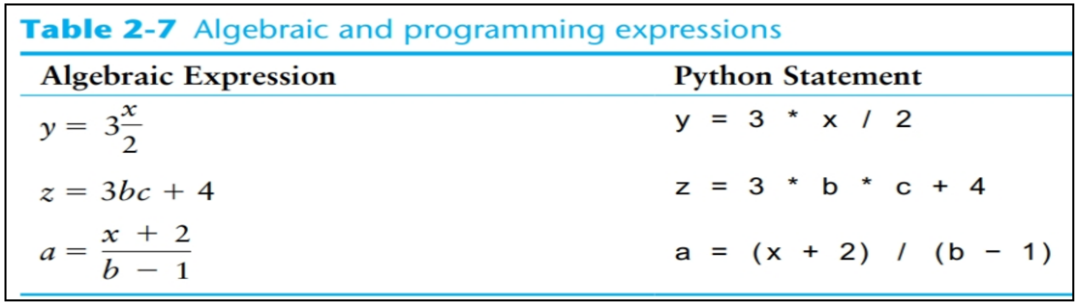


Table 2-7 Algebraic and programming expressions

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**
- **<u>Multiline continuation character (\)</u>: Allows to break a statement into multiple lines**

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

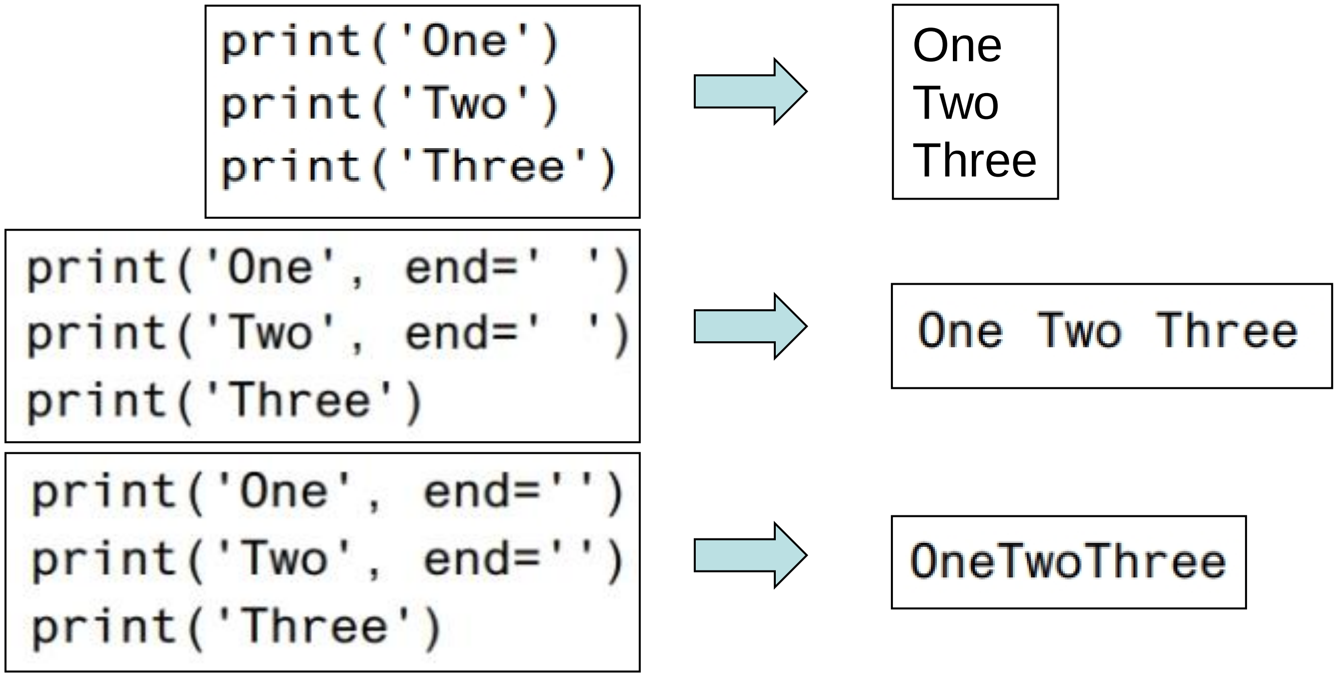
Breaking Long Statements into Multiple Lines

- **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)

- `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('One')
print('Two')
print('Three')
```

```
One
Two
Three
```

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

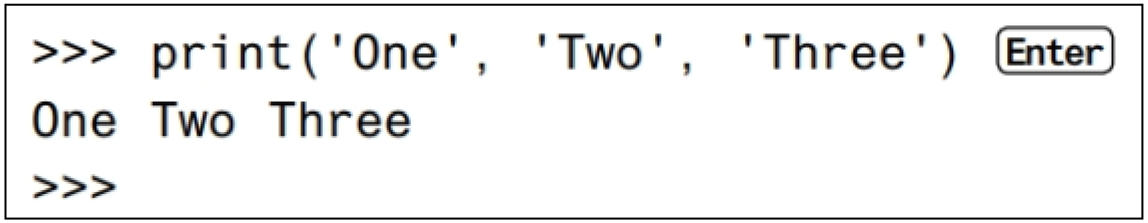
```
One Two Three
```

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

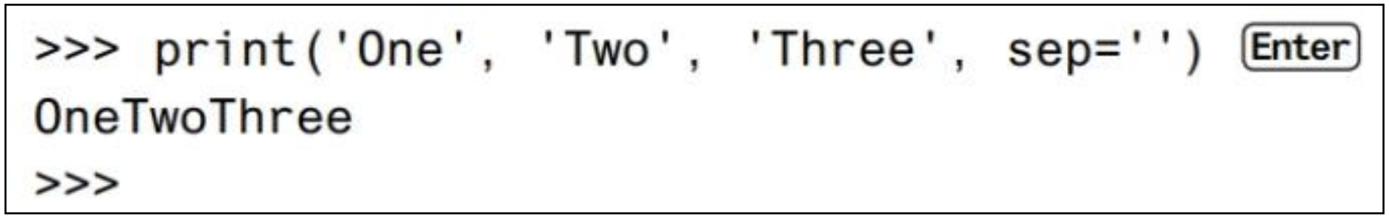
```
OneTwoThree
```

More About Data Output (print) - continued

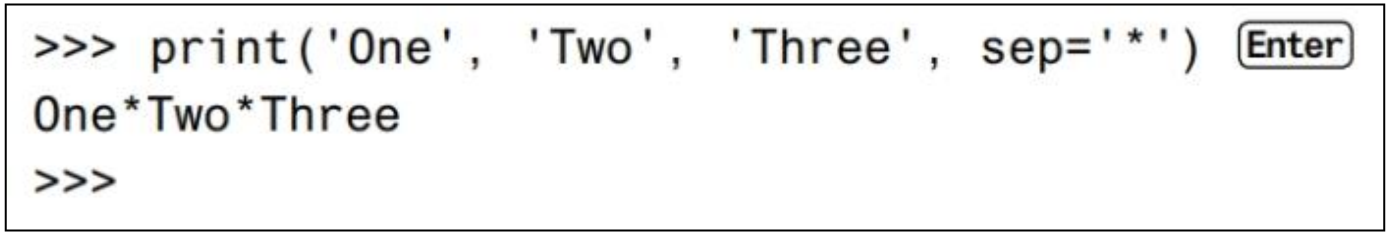
- **`print` function uses space as item separator**
  - Special argument `sep='`*`delimiter`*`'` causes `print` to use *`delimiter`* as item separator



```
>>> print('One', 'Two', 'Three') Enter
One Two Three
>>>
```



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



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

More About Data Output (escape characters)

- **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
```

```
print('Mon\tTues\tWed')
print('Thur\tFri\tSat')
```

```
Mon     Tues    Wed
Thur    Fri     Sat
```

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

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

```
print('The path is C:\\temp\\data.')
```

```
The path is C:\temp\data.
```

More About Data Output (string concatenation)

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

```
print('This is ' + 'one string.')
```

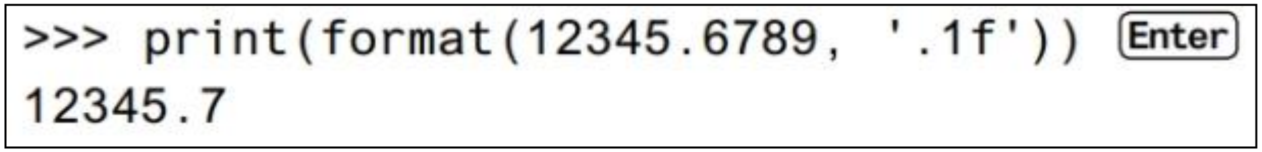
```
This is one string.
```

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

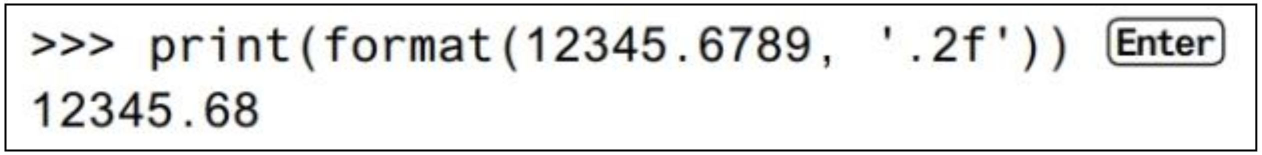
```
Enter the amount of sales for each day and press Enter.
```

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 scientific notation, comma separators, and the minimum field width used to display the value



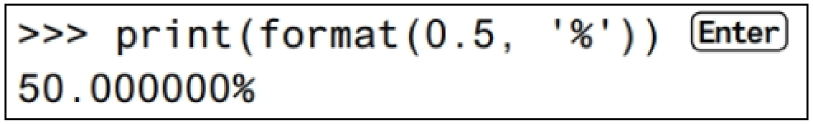
```
>>> print(format(12345.6789, '.1f')) Enter
12345.7
```



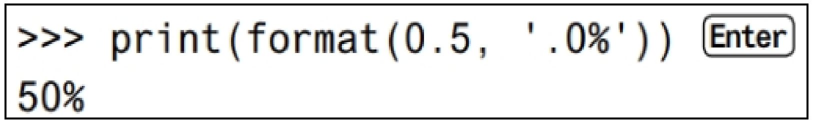
```
>>> print(format(12345.6789, '.2f')) Enter
12345.68
```

Formatting Numbers (cont'd.)

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

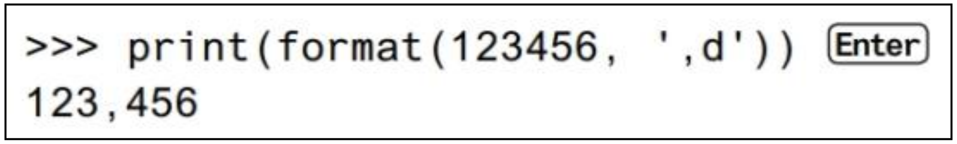


```
>>> print(format(0.5, '%')) Enter
50.000000%
```

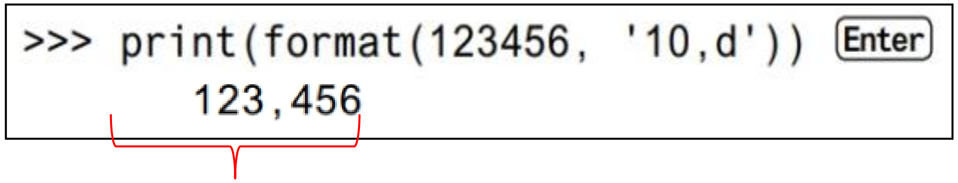


```
>>> print(format(0.5, '.0%')) Enter
50%
```

- **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, '10,d')) Enter
   123,456
```

Print with 10 spaces wide (padded 3 spaces in front)

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

- **Difficult to determine the purpose of the number**
- **Need to change in lots of places within program if magic number is used in multiple places**
- **Risk of making a typo mistake each time you use a 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 troubleshoot!

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
  - ⇒ Only need to change value assigned to the constant, new value takes effect everywhere the constant is used in program

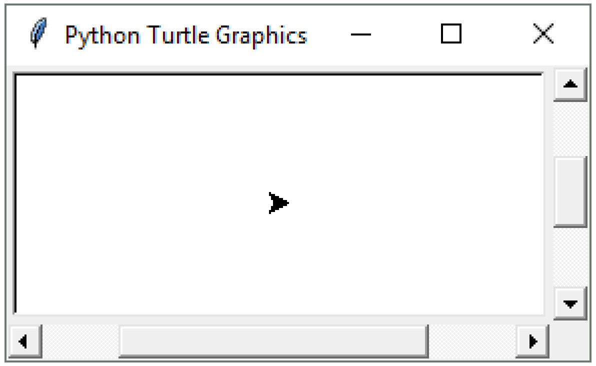
- Named constants help prevent typo mistakes or errors that are common when using magic numbers

Chapter 2 – Input, Processing & Output

QUESTION?

Introduction to Turtle Graphics

- **Python's turtle graphics system displays a small cursor known as a *turtle*.**



- **You can use Python statements to move the turtle around the screen, drawing lines and shapes.**

Introduction to Turtle Graphics

- **To use the turtle graphics system, you must import the turtle module with this statement:**

  ```
  import turtle
  ```

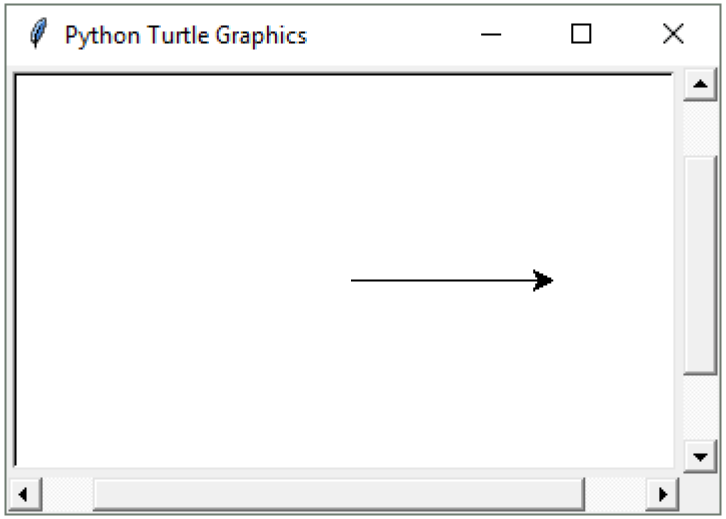
  This loads the turtle module into memory

- ***Note:*** *Import statement and Standard Library will be learned in Chapter 5 later*

Moving the Turtle Forward

- Use the **`turtle.forward(`*`n`*`)`** statement to move the turtle forward ***n*** pixels.

```
>>> import turtle
>>> turtle.forward(100)
>>>
```

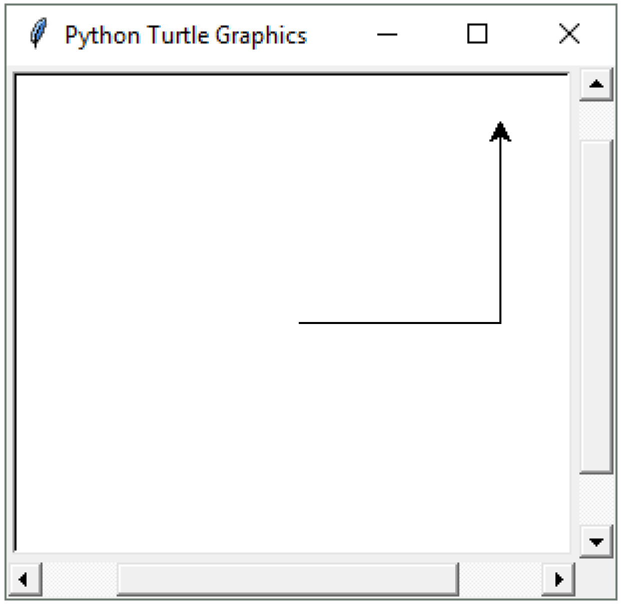


Turning the Turtle

- The turtle's initial heading is 0 degrees (east)
- Use the `turtle.right(`*`angle`*`)` statement to turn the turtle right by *`angle`* degrees.
- Use the `turtle.left(`*`angle`*`)` statement to turn the turtle left by *`angle`* degrees.

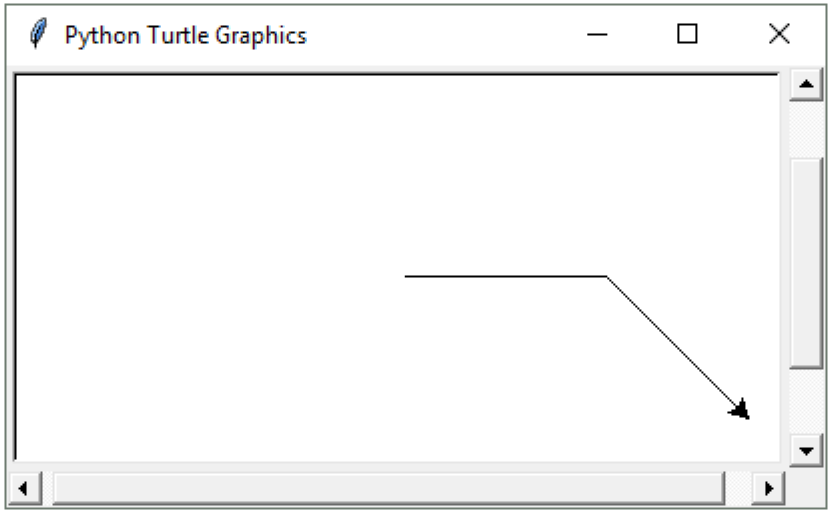
Turning the Turtle

```
>>> import turtle
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>>
```



Turning the Turtle

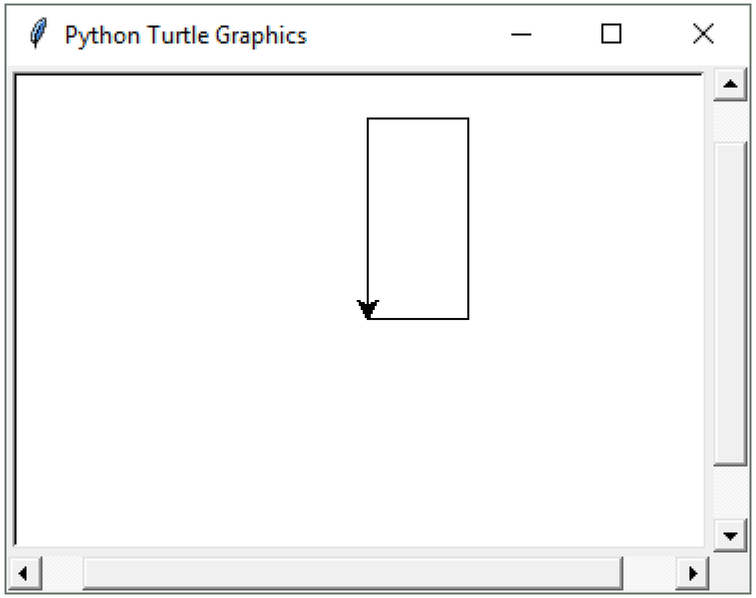
```
>>> import turtle
>>> turtle.forward(100)
>>> turtle.right(45)
>>> turtle.forward(100)
>>>
```



Setting the Turtle's Heading

- **Use the `turtle.setheading(`*`angle`*`)` statement to set the turtle's heading to a specific angle.**

```
>>> import turtle
>>> turtle.forward(50)
>>> turtle.setheading(90)
>>> turtle.forward(100)
>>> turtle.setheading(180)
>>> turtle.forward(50)
>>> turtle.setheading(270)
>>> turtle.forward(100)
>>>
```

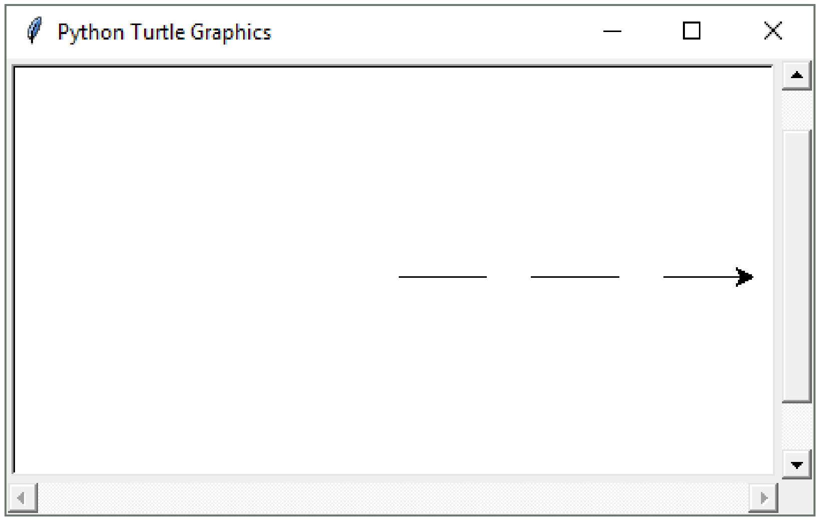


Setting the Pen Up or Down

- **When the turtle's pen is down, the turtle draws a line as it moves. By default, the pen is down.**
- **When the turtle's pen is up, the turtle does not draw as it moves.**
- **Use the `turtle.penup()` statement to raise the pen.**
- **Use the `turtle.pendown()` statement to lower the pen.**

Setting the Pen Up or Down

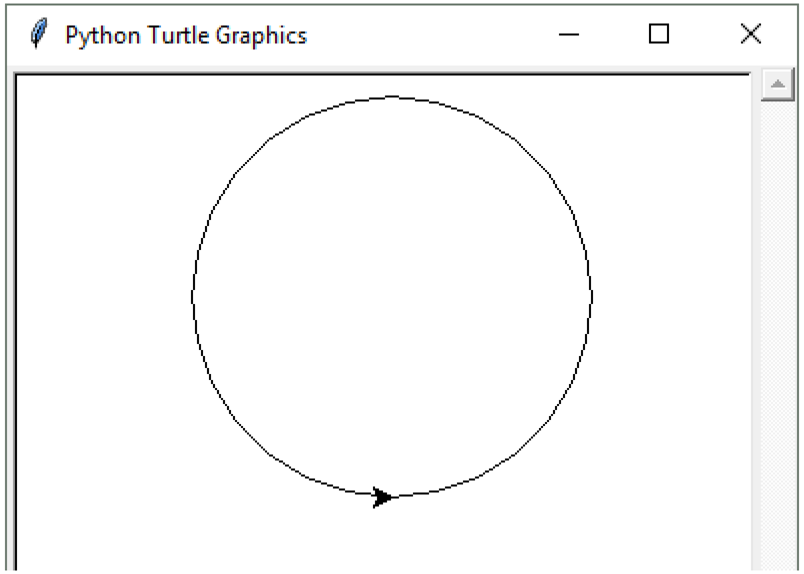
```
>>> import turtle
>>> turtle.forward(50)
>>> turtle.penup()
>>> turtle.forward(25)
>>> turtle.pendown()
>>> turtle.forward(50)
>>> turtle.penup()
>>> turtle.forward(25)
>>> turtle.pendown()
>>> turtle.forward(50)
>>>
```



Drawing Circles

- **Use the `turtle.circle(`*`radius`*`)` statement to draw a circle with a specified radius.**

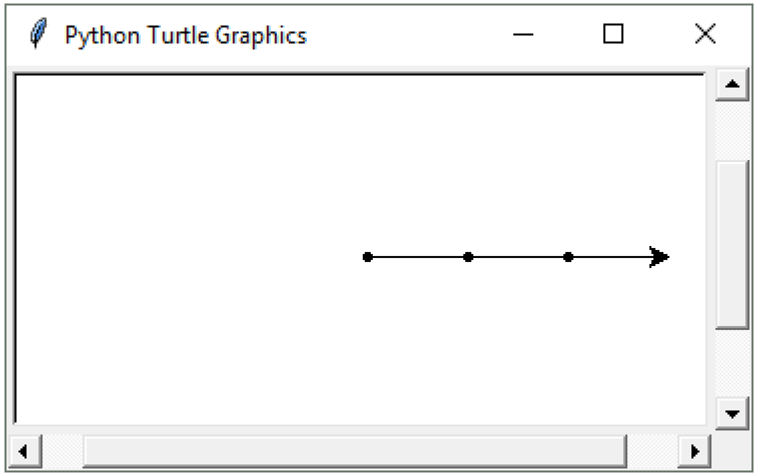
```
>>> import turtle
>>> turtle.circle(100)
>>>
```



Drawing Dots

- **Use the `turtle.dot()` statement to draw a simple dot at the turtle's current location.**

```
>>> import turtle
>>> turtle.dot()
>>> turtle.forward(50)
>>> turtle.dot()
>>> turtle.forward(50)
>>> turtle.dot()
>>> turtle.forward(50)
>>>
```

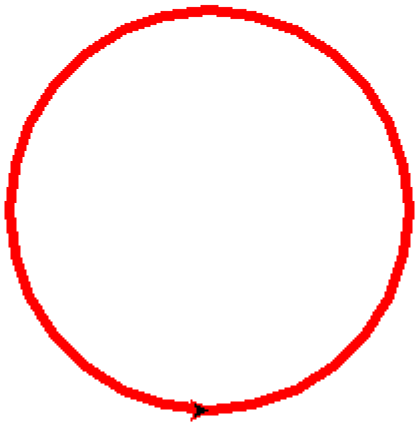


Changing the Pen Size and Drawing Color

- **Use the `turtle.pensize(`*`width`*`)` statement to change the width of the turtle's pen, in pixels.**

- **Use the `turtle.pencolor(`*`color`*`)` statement to change the turtle's drawing color.**
  - *See Appendix D in your textbook for a complete list of colors*.

```
>>> import turtle
>>> turtle.pensize(5)
>>> turtle.pencolor('red')
>>> turtle.circle(100)
>>>
```



Working with the Turtle's Window

- **Use the `turtle.bgcolor(`*`color`*`)` statement to set the window's background color.**
  - *See Appendix D in your textbook for a complete list of colors*.
- **Use the `turtle.setup(`*`width, height`*`)` statement to set the size of the turtle's window, in pixels.**
  - The *`width`* and *`height`* arguments are the width and height, in pixels.
  - For example, the following interactive session creates a graphics window that is 640 pixels wide and 480 pixels high:

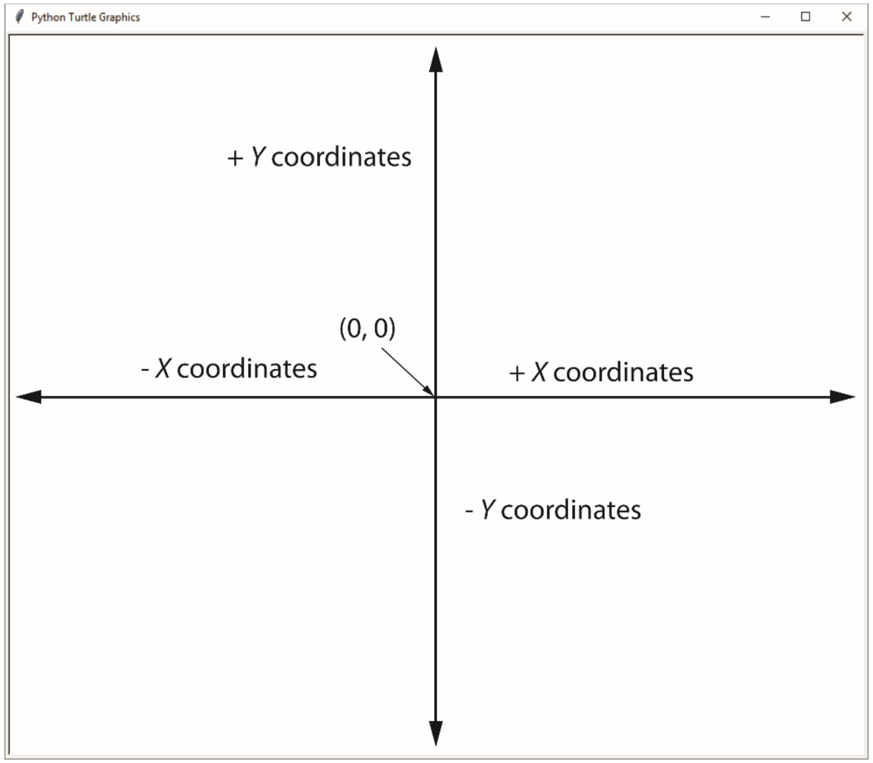
```
>>> import turtle
>>> turtle.setup(640, 480)
>>>
```

Resetting the Turtle's Window

- **The `turtle.reset()` statement:**
  - Erases all drawings that currently appear in the graphics window.
  - Resets the drawing color to black.
  - Resets the turtle to its original position in the center of the screen.
  - Does *not* reset the graphics window’s background color.
- **The `turtle.clear()` statement:**
  - Erases all drawings that currently appear in the graphics window.
  - Does *not* change the turtle's position.
  - Does *not* change the drawing color.
  - Does *not* change the graphics window’s background color.
- **The `turtle.clearscreen()` statement:**
  - Erases all drawings that currently appear in the graphics window.
  - Resets the drawing color to black.
  - Resets the turtle to its original position in the center of the screen.
  - Resets the graphics window’s background color to white.

Working with Coordinates

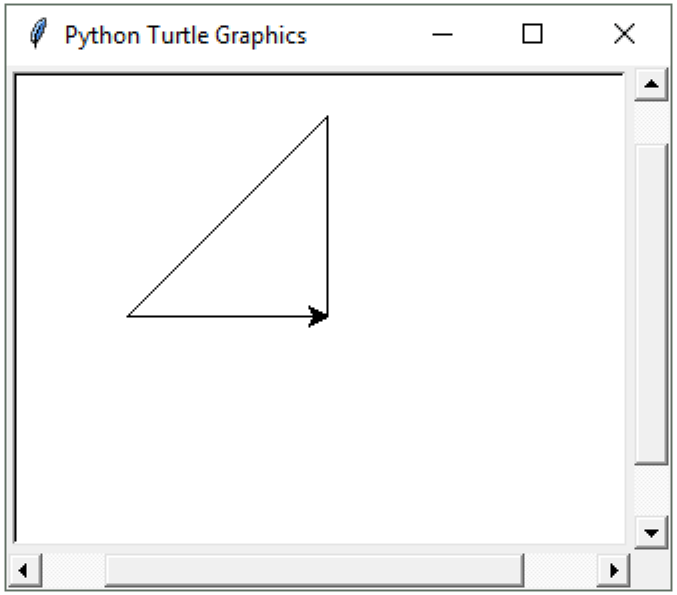
- **The turtle uses Cartesian Coordinates**



Moving the Turtle to a Specific Location

- **Use the `turtle.goto(`*`x, y`*`)` statement to move the turtle to a specific location.**

```
>>> import turtle
>>> turtle.goto(0, 100)
>>> turtle.goto(-100, 0)
>>> turtle.goto(0, 0)
>>>
```



- The `turtle.pos()` statement displays the turtle's current *X*,*Y* coordinates.
- The `turtle.xcor()` statement displays the turtle's current *X* coordinate and the `turtle.ycor()` statement displays the turtle's current *Y* coordinate.

Animation Speed

- **Use the `turtle.speed(speed)` command to change the speed at which the turtle moves.**
  - The *`speed`* argument is a number in the range of 0 through 10.
  - If you specify 0, then the turtle will make all of its moves instantly (animation is disabled).

Hiding and Displaying the Turtle

- **Use the `turtle.hideturtle()` command to hide the turtle.**
  - This command does not change the way graphics are drawn, it simply hides the turtle icon.

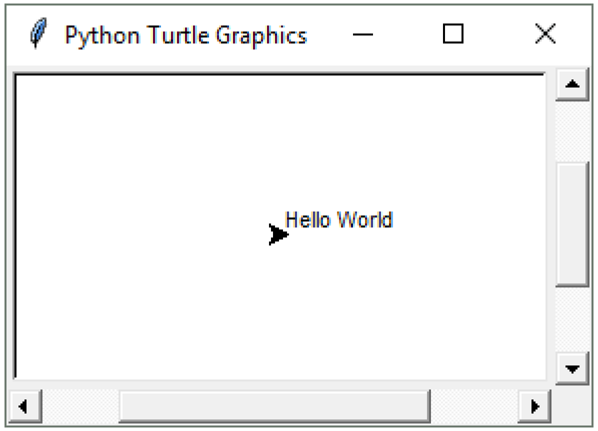
- **Use the `turtle.showturtle()` command to display the turtle.**

Displaying Text

- **Use the `turtle.write(`*`text`*`)` statement to display text in the turtle's graphics window.**
  - The *`text`* argument is a string that you want to display.
  - The lower-left corner of the first character will be positioned at the turtle's *X* and *Y* coordinates.

Displaying Text

```
>>> import turtle
>>> turtle.write('Hello World')
>>>
```

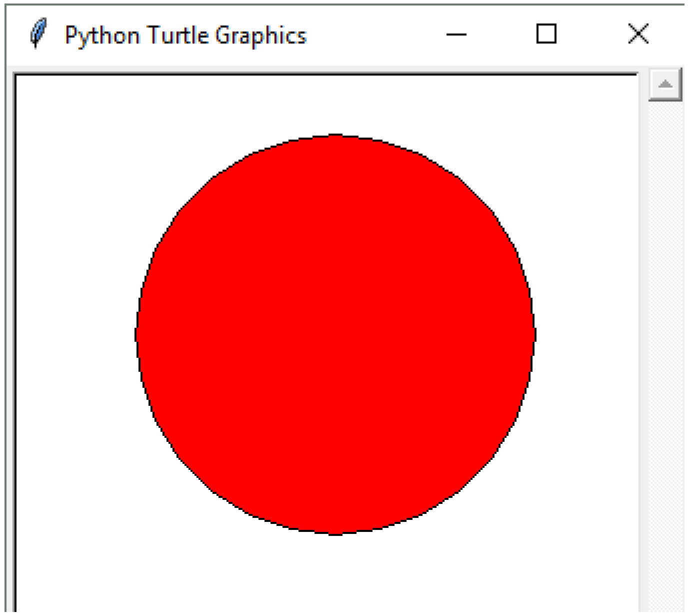


Filling Shapes

- **To fill a shape with a color:**
  - Use the `turtle.begin_fill()` command before drawing the shape
  - Then use the `turtle.end_fill()` command after the shape is drawn.
  - When the `turtle.end_fill()` command executes, the shape will be filled with the current fill color

Filling Shapes

```
>>> import turtle
>>> turtle.hideturtle()
>>> turtle.fillcolor('red')
>>> turtle.begin_fill()
>>> turtle.circle(100)
>>> turtle.end_fill()
>>>
```



Keeping the Graphics Window Open

- **When running a turtle graphics program outside IDLE, the graphics window closes immediately when the program is done.**
- **To prevent this, add the `turtle.done()` statement to the very end of your turtle graphics programs.**
  - This will cause the graphics window to remain open, so you can see its contents after the program finishes executing.
- **May use `time.sleep(`*`n`*`)` to hold the program for *n* seconds**
  - Remember to import the `time` module.

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
  - The turtle graphics system

References

- **Python Turtle Documentation:**
  - https://docs.python.org/3.3/library/turtle.html?highlight=turtle
- **Python Format Documentation:**
  - https://python-reference.readthedocs.io/en/latest/docs/functions/format.html
- **Basic Turtle Video**
  - https://www.youtube.com/watch?v=_O24rjQ6vTk
- **Other useful Python learning sites**
  - https://www.w3schools.com/python/default.asp
  - https://www.programiz.com/python-programming
  - https://realpython.com/learning-paths/python3-introduction/
  - https://www.geeksforgeeks.org/python-programming-language/?ref=lbp