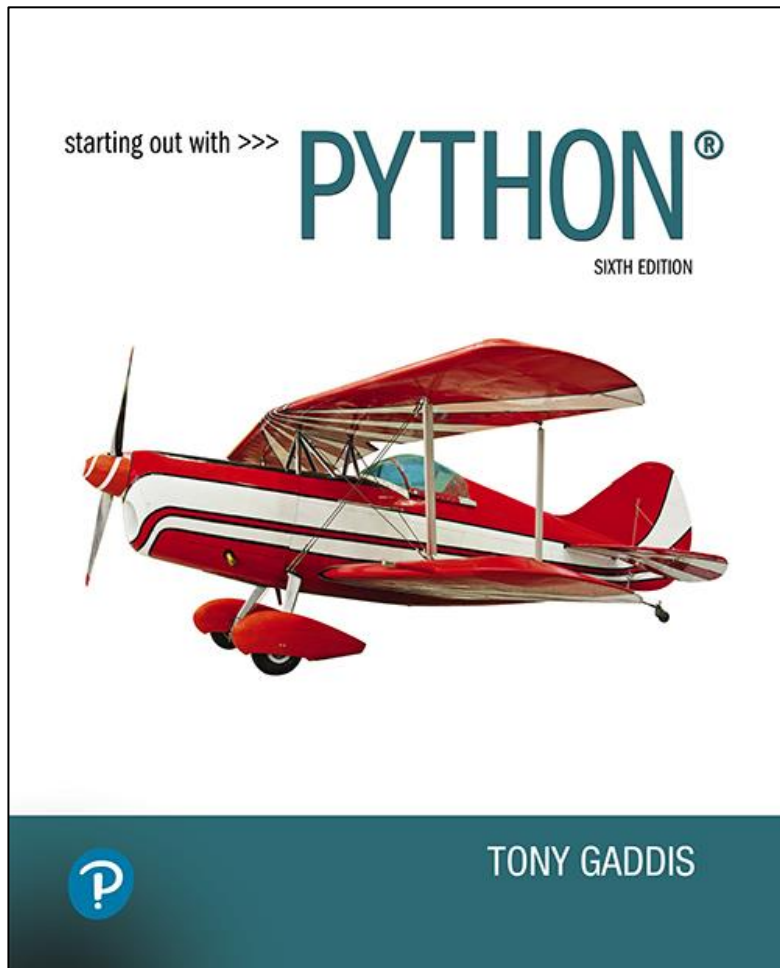


Starting Out with Python

Sixth Edition



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
- String Concatenation
- More About The `print` Function
- Displaying Formatted Output
- Named Constants
- Introduction to Turtle Graphics

Designing a Program (1 of 3)

- **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 (2 of 3)

- **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 (3 of 3)

- **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**

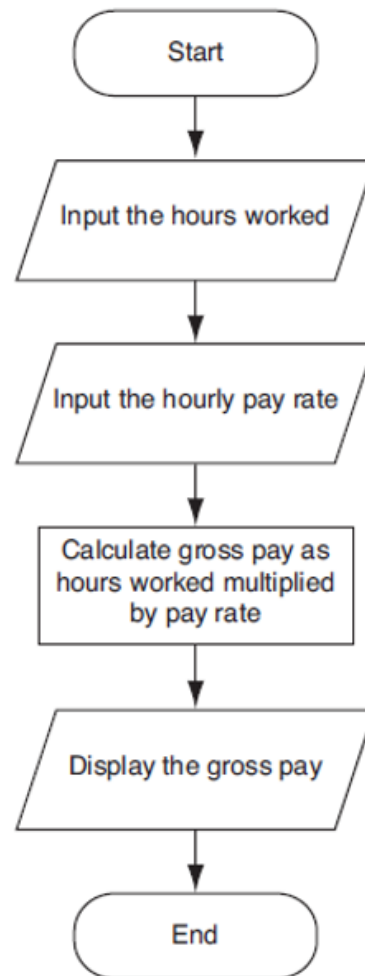
Pseudocode

- **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

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

Figure 2-2 Flowchart for the Pay Calculating Program



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 (1 of 2)

- **print function: displays output on the screen**

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

Displaying Output With the Print Function (2 of 2)

- **Function:** piece of prewritten code that performs an operation
- **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

Program Execution Order

- **In a simple program, statements execute in the order they appear**
 - From top to bottom

```
print('Programming')  
print('is')  
print('fun!')
```

Program Output

```
Programming  
is  
fun!
```

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

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

Variables (1 of 4)

- **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 (2 of 4)

`temperature = 75` ← Assigns 75 to the `temperature` variable.

`cost = 87.99` ← Assigns 87.99 to the `cost` variable.

`name = 'Monty'` ← Assigns the string 'Monty' to the `name` variable.

Variables (3 of 4)

- **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 if a value is assigned to it**

Variables (4 of 4)

- You can assign values to multiple variables in a single statement.

`x, y, z = 0, 1, 2`

- This statement makes the following assignments
 - `x = 0`
 - `y = 1`
 - `z = 2`
- This is known as multiple assignment
- Notice the variable names on the left side of the `=` operator are separated by commas, and the values on the right side of the `=` operator are separated by commas.

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**

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

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**

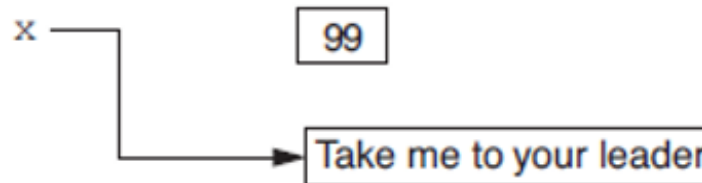
Reassigning a Variable to a Different Type

- 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 (1 of 2)

- **Most programs need to read input from the user**
- **Built-in `input` function displays a prompt and 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

Reading Input From the Keyboard (2 of 2)

- **Example:**

```
name = input('What is your name? ')
```

- **This statement does the following:**

- Displays the string 'What is your name? '
- Reads input, as a string, from the keyboard
- Assigns the string that was read from the keyboard to the name variable

Reading Numbers With the `input` Function (1 of 3)

- `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, causes an error

Reading Numbers With the `input` Function (2 of 3)

- **Example:**

```
number = int(input('Enter a number: '))
```

- **This statement does the following:**

- Displays the string `'Enter a number: '`
- Reads input, as a string, from the keyboard
- Converts the input to an `int`
- Assigns the resulting `int` to the `number` variable

Reading Numbers With the `input` Function (3 of 3)

- **Example:**

```
score = float(input('Enter your score: '))
```

- **This statement does the following:**
 - Displays the string `'Enter your score: '`
 - Reads input, as a string, from the keyboard
 - Converts the input to a `float`
 - Assigns the resulting `float` to the `score` variable

Performing Calculations (1 of 3)

- **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

Performing Calculations (2 of 3)

- **Operators**

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 (3 of 3)

- **Two types of division: floating-point and integer**
 - The `/` operator performs floating point division
 - Gives the result as a floating-point number
 - The `//` operator performs integer division
 - Gives the result as an integer
 - Positive results are truncated
 - Negative results are rounded away from zero

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

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

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 (1 of 2)

- 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 (2 of 2)

- **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)
```

String Concatenation (1 of 2)

- **To append one string to the end of another string**
- **Use the + operator to concatenate strings**

```
>>> message = 'Hello ' + 'world'  
>>> print(message)  
Hello world  
>>>
```

String Concatenation (2 of 2)

- You can use string concatenation to break up a long string literal

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

This statement will display the following:

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

Implicit String Literal Concatenation (1 of 2)

- **Two or more string literals written adjacent to each other are implicitly concatenated into a single string**

```
>>> my_str = 'one' 'two' 'three'  
>>> print(my_str)  
onetwothree
```


Implicit String Literal Concatenation (2 of 2)

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

This statement will display the following:

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

More About the `print` Function (1 of 2)

- **`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

More About the `print` Function (2 of 2)

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

Displaying Formatted Output With F-Strings (1 of 8)

- **An f-string is a special type of string literal that is prefixed with the letter `f`**

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

- **F-strings support placeholders for variables**

```
>>> name = 'Johnny'  
>>> print(f'Hello {name}.')  
Hello Johnny.
```

Displaying Formatted Output With F-Strings (2 of 8)

- **Placeholders can also be expressions that are evaluated**

```
>>> print(f'The value is {10 + 2}.')  
The value is 12.
```

```
>>> val = 10  
>>> print(f'The value is {val + 2}.')  
The value is 12.
```

Displaying Formatted Output With F-Strings (3 of 8)

- **Format specifiers can be used with placeholders**

```
>> num = 123.456789
>> print(f'{num:.2f} ')
123.46
>>>
```

- **.2f means:**
 - round the value to 2 decimal places
 - display the value as a floating-point number

Displaying Formatted Output With F-Strings (4 of 8)

- **Other examples:**

```
>> num = 1000000.00  
>> print(f'{num:,.2f} ')  
1,000,000.00
```

```
>>> discount = 0.5  
>>> print(f'{discount:.0%} ')  
50%
```

Displaying Formatted Output With F-Strings (5 of 8)

- **Other examples:**

```
>> num = 123456789
>> print(f'{num:,d} ')
123,456,789
```

```
>>> num = 12345.6789
>>> print(f'{num:.2e} ')
1.23e+04
```


Displaying Formatted Output With F-Strings (6 of 8)

- **Specifying a minimum field width:**

```
>>> num = 12345.6789
>>> print(f'The number is {num:12,.2f}')
```

The number is 12,345.68

Field width = 12

The number is

			1	2	,	3	4	5	.	6	8
--	--	--	---	---	---	---	---	---	---	---	---

Field width = 12

Displaying Formatted Output With F-Strings (7 of 8)

- **Aligning values within a field**
 - Use < for left alignment
 - Use > for right alignment
 - Use ^ for center alignment
- **Examples:**
 - `print(f' {num:<20.2f} ')`
 - `print(f' {num:>20.2f} ')`
 - `print(f' {num:^20.2f} ')`

Displaying Formatted Output With F-Strings (8 of 8)

- **The order of designators in a format specifier**
 - When using multiple designators in a format specifier, write them in this order:

`[alignment] [width] [,] [.precision] [type]`

- **Example:**
 - `print(f' {number:^10, .2f}')`

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:**

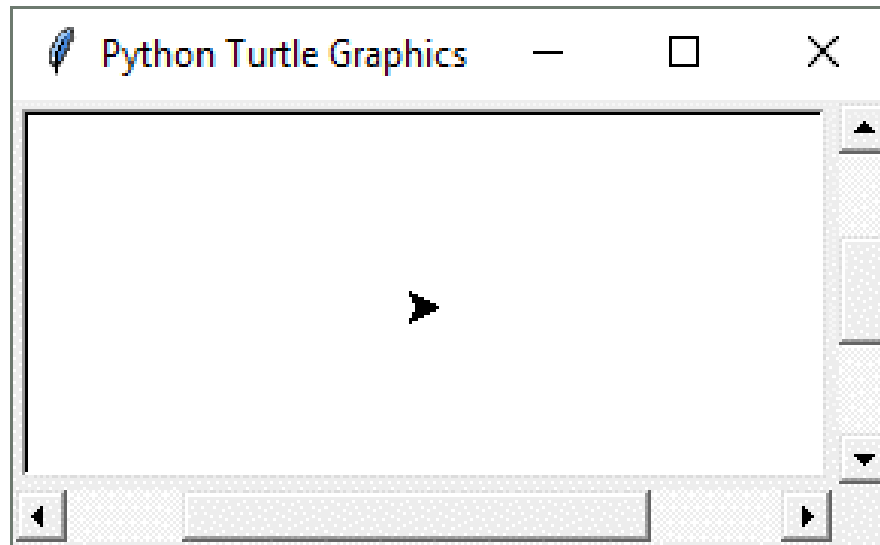
```
amount = balance * INTEREST_RATE
```

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

Introduction to Turtle Graphics (1 of 2)

- 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 (2 of 2)

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

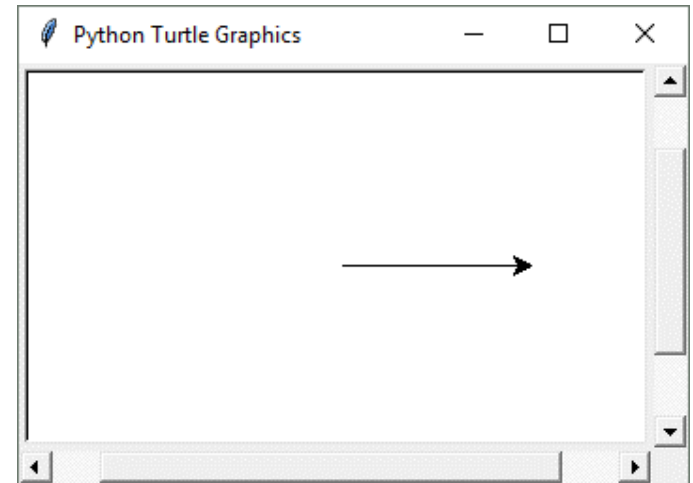
```
import turtle
```

This loads the turtle module into memory

Moving the Turtle Forward

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

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

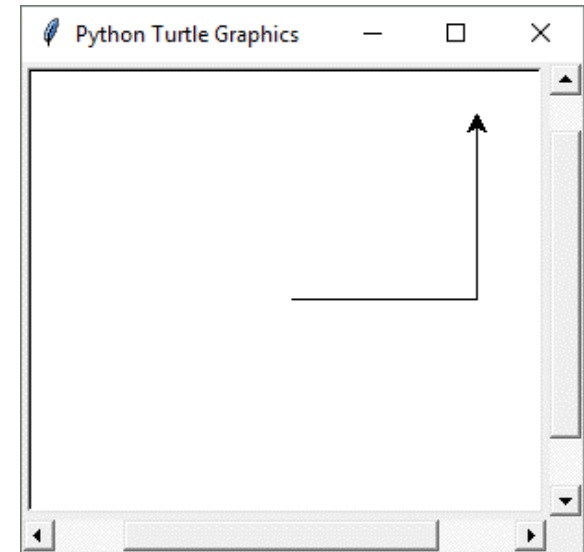


Turning the Turtle (1 of 3)

- 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.

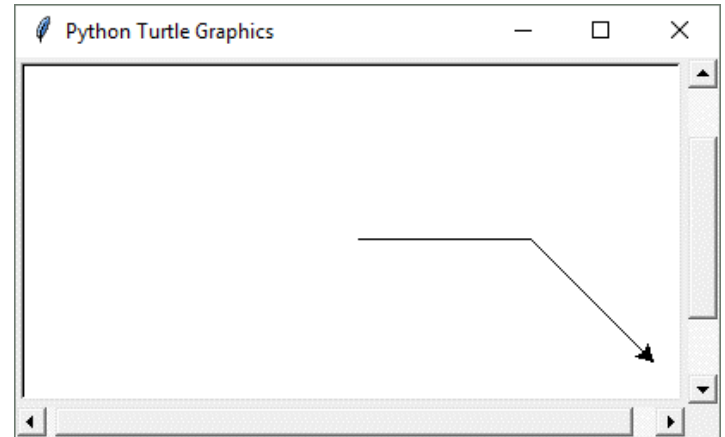
Turning the Turtle (2 of 3)

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



Turning the Turtle (3 of 3)

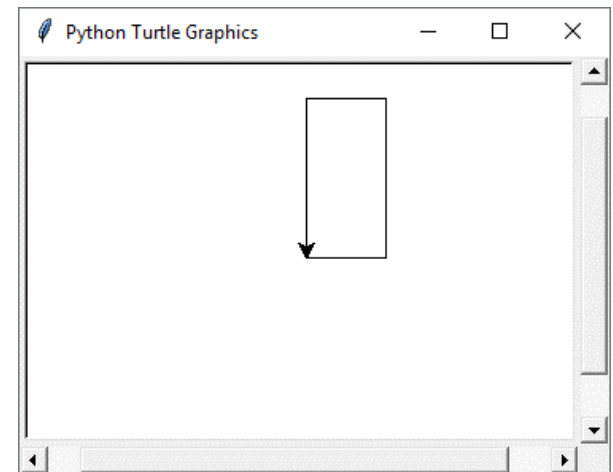
```
>>> import turtle
>>> turtle.forward(100)
>>> turtle.right(45)
>>> turtle.forward(100)
>>>
```



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)
>>>
```

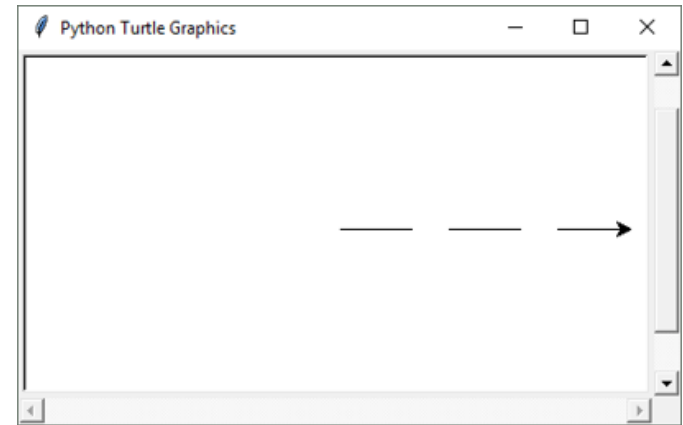


Setting the Pen Up or Down (1 of 2)

- 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 (2 of 2)

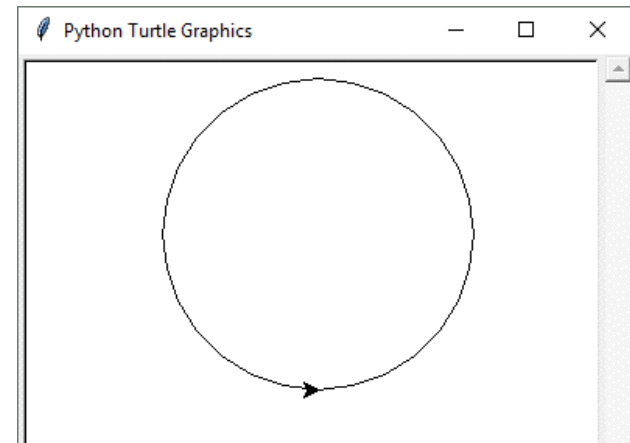
```
>>> import turtle
>>> turtle.forward(50)
>>> turtle.penup()
>>> turtle.forward(25)
>>> turtle.pendown()
>>> turtle.forward(50)
>>> turtle.penup()
>>> turtle.forward(25)
>>> turtle.pendown()
>>> turtle.forward(50)
>>>
```



Drawing Circles

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

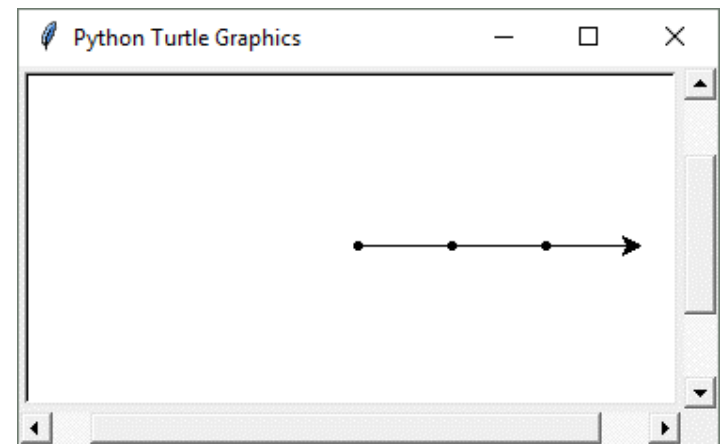
```
>>> import turtle
>>> turtle.circle(100)
>>>
```



Drawing Dots

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

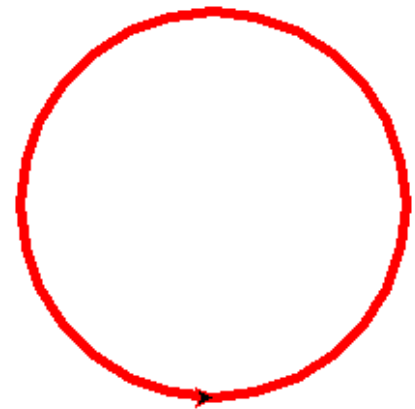
```
>>> import turtle
>>> turtle.dot()
>>> turtle.forward(50)
>>> turtle.dot()
>>> turtle.forward(50)
>>> turtle.dot()
>>> turtle.forward(50)
>>>
```



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:

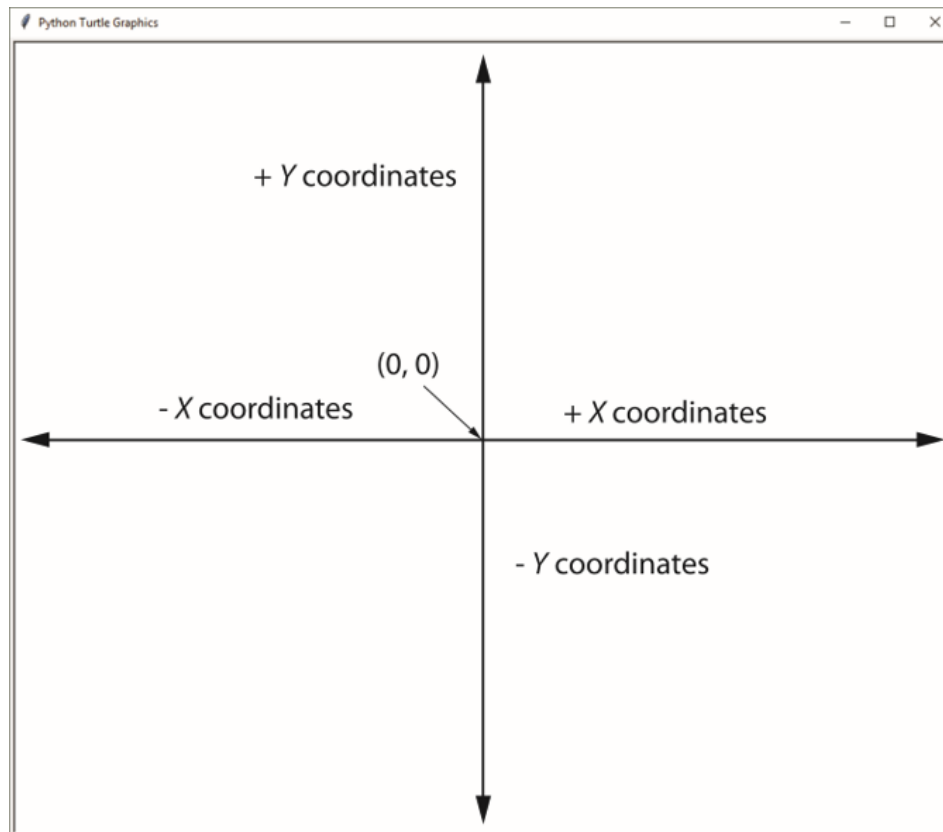
```
>>> import turtle
>>> turtle.setup(640, 480)
>>>
```

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

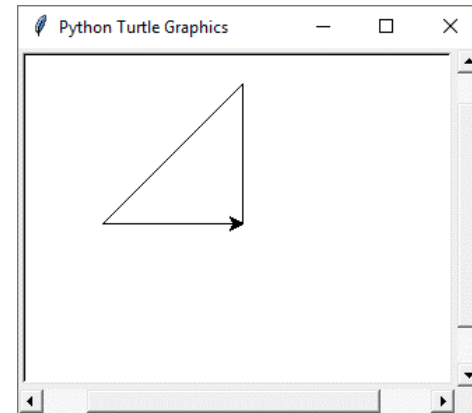
- The turtle uses **Cartesian Coordinates**



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

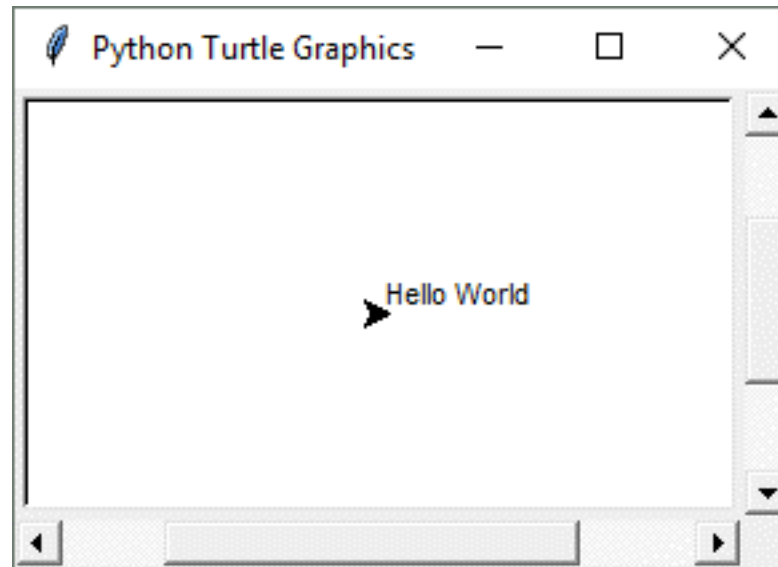
- Use the `turtle.hideturtle()` command to hide the turtle.
 - This command does not change the way graphics are drawn, it simply hides the turtle icon.
- Use the `turtle.showturtle()` command to display the turtle.

Displaying Text (1 of 2)

- 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 (2 of 2)

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

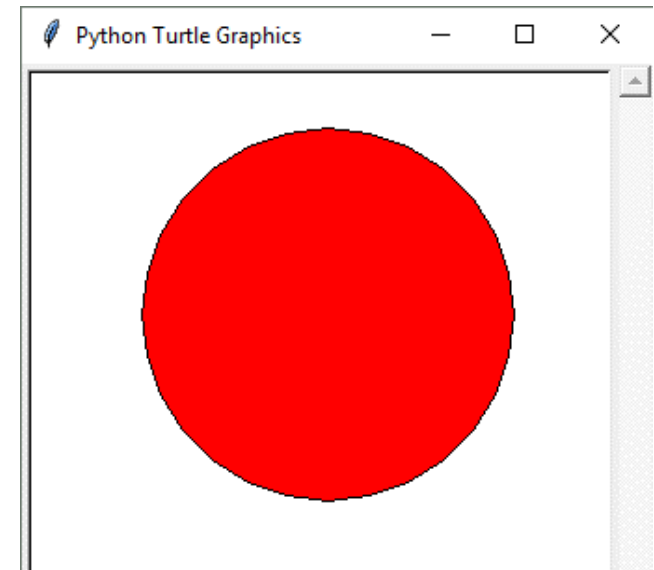


Filling Shapes (1 of 2)

- **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

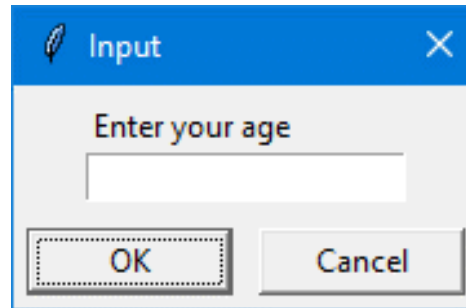
Filling Shapes (2 of 2)

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

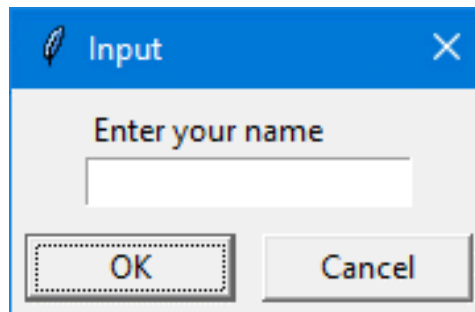


Getting Input With a Dialog Box (1 of 2)

```
>>> import turtle  
>>> age = turtle.numinput('Input', 'Enter your age')
```



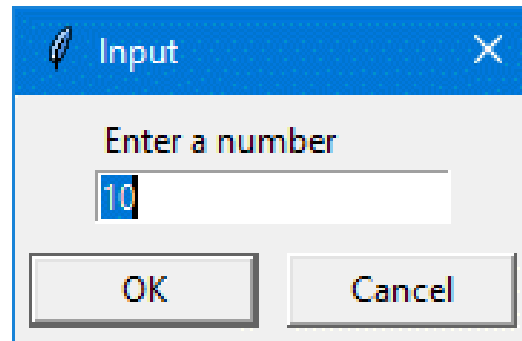
```
>>> import turtle  
>>> name = turtle.textinput('Input', 'Enter your name')
```



Getting Input With a Dialog Box (2 of 2)

- **Specifying a default value, minimum value, and maximum value with `turtle.numinput`:**

```
>>> import turtle
>>> num = turtle.numinput('Input', 'Enter a number',
                          default=10, minval=0, maxval=100)
```



- **An error message will be displayed if the input is less than `minval` or greater than `maxval`**

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.

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

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.