Input, Processing, and Output

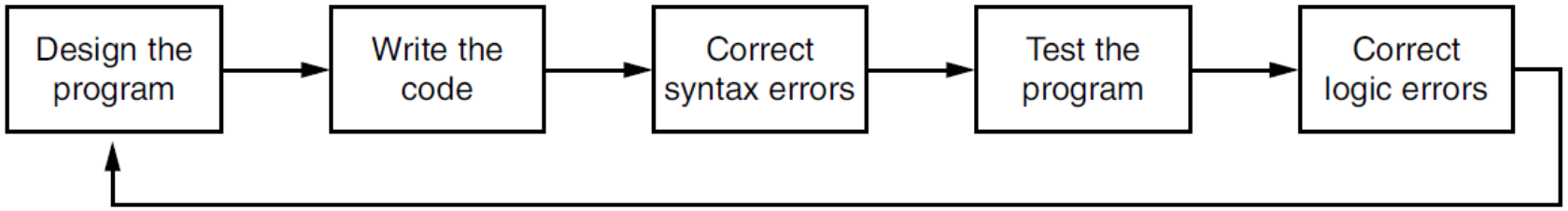
陳建良

Designing a Program

- Programs must be carefully designed before they are written.
- During the design process, programmers use tools such as **pseudocode** and **flowcharts** to create models of programs.

The Program Development Cycle

- The process of creating a program that works correctly typically requires the five phases.
- The entire process is known as the ***program development cycle***.



More About the Design Process

- The process of designing a program is arguably the most important part of the cycle.
- You can think of a program's design as its foundation.
- If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.
- The process of designing a program can be summarized in the following two steps:
  1. Understand the task that the program is to perform.
  2. Determine the steps that must be taken to perform the task.

Understand the Task That the Program Is to Perform

- Well what is the Task ?
- We begin answering this by asking/interviewing our client/end user.
- We, the programmers, must ask lots of questions and get as many details as possible about the task.
- We might forget questions… and that is okay. We tend to have follow up sessions.
- Once we have met with the client we generally construct a "software requirement" document.
- This agreement between us and the client establishes what the program should actually do

Determine the Steps That Must Be Taken to Perform the Task

- Next we break down the task into a series of concrete steps that can be followed (like a recipe).
- Remember that computers need each step to be broken down into minute detail.
- They don't have the ability to infer intermediate steps like we can!
- For example, suppose someone asks you how to boil water.

Boiling Water… How do I tell a computer to do that?

You might break down that task into a series of steps as follows:

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.

- This is an example of an *algorithm*, which is a set of well-defined logical steps that must be taken to perform a task.
- Notice the steps in this algorithm are sequentially ordered.

- A programmer breaks down the task that a program must perform in a similar way.
- An algorithm is created, which lists all of the logical steps that must be taken.
- For example, suppose you have been asked to 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.

- Of course, this algorithm isn't ready to be executed on the computer.
- The steps in this list have to be translated into code.
- Programmers commonly use two tools to help them accomplish this.
  - Pseudocode
  - Flowcharts

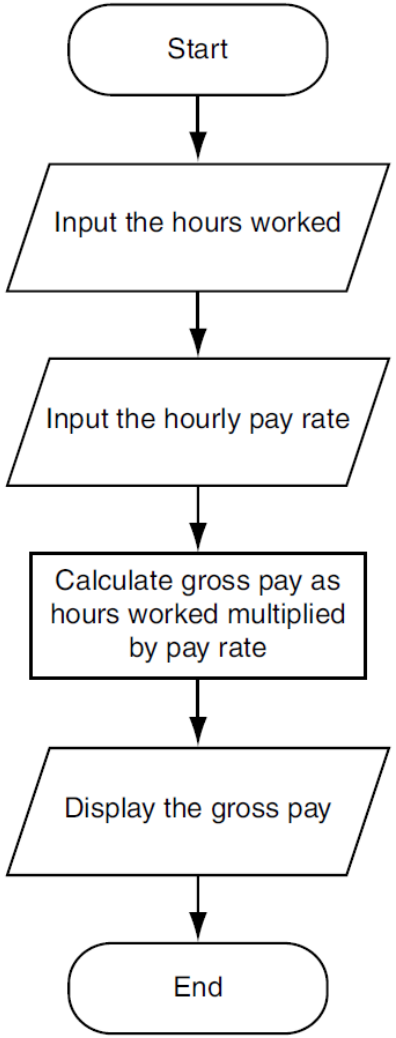
Pseudocode

- The word "pseudo" means fake, so *pseudocode* is fake code.
- It is an informal language that has no syntax rules and is not meant to be compiled or executed.
- Programmers use pseudocode to create models, or "mock-ups," of programs.
- Programmers don't have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design.

- Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:
  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*
- Each statement in the pseudocode represents an operation that can be performed in Python.
- For example, Python can read input that is typed on the keyboard, perform mathematical calculations, and display messages on the screen.

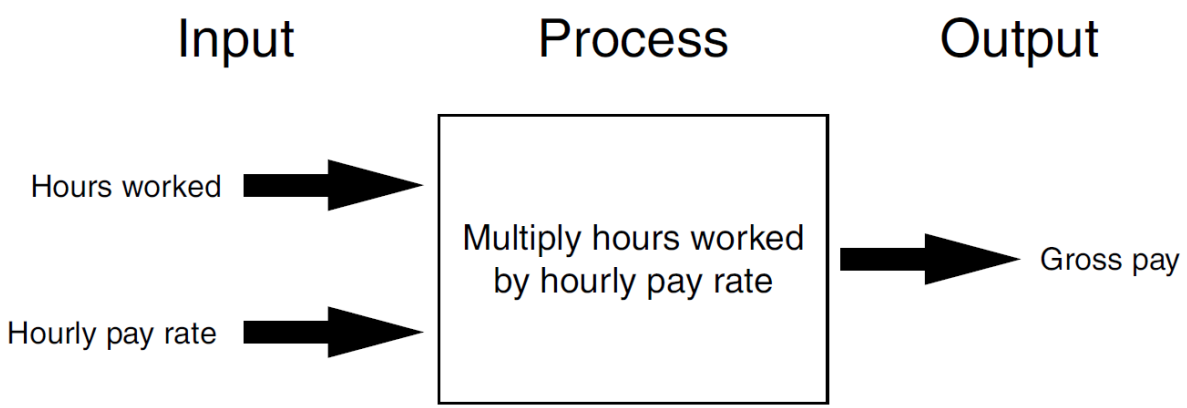
Flowcharts

- A *flowchart* is a diagram that graphically depicts the steps that take place in a program.
- Notice there are three types of symbols ir the flowchart: ovals, parallelograms, and a rectangle.
- Each of these symbols represents a step in the program, as described here:
  - The ovals are called *terminal symbols*.
  - Parallelograms are used as *input symbols* and *output symbols*.
  - Rectangles are used as *processing symbols*.



Input, Processing, and Output

- Computer programs typically perform the following three-step process:
  1. Input is received.
  2. Some process is performed on the input.
  3. Output is produced.



Displaying Output with the print Function

- A *function* is a piece of prewritten code that performs an operation.
- Python has numerous built-in functions that perform various operations.
- The most fundamental built-in function is the print function, which displays output on the screen.
- Functions always begin with a keyword followed by a series of parenthesis.

  Ex: print ()
- You can "pass" one or more "arguments" into a function by placing data inside the parenthesis

  Ex: print('Hello world')

- Different functions "expect" different arguments. The print function, for example, expects printed text as an argument.
- When you ask Python to run a function we say that you have "called" the function.
- Suppose your instructor tells you to write a program that displays your name and address on the computer screen.

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

Program Output

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

Strings and String Literals

- In programming terms, a sequence of characters that is used as data is called a ***string***.
- When a string appears in the actual code of a program, it is called a ***string literal***.
- In Python code, string literals must be enclosed in quote marks.
- The quote marks simply mark where the string data begins and ends.
- In Python, you can enclose string literals in a set of single-quote marks (') or a set of double-quote marks (").

- 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.

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

Program Output

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

- you can use single-quote marks to enclose a string literal that contains doublequotes as part of the string.

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

Program Output

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

- Python also allows you to enclose string literals in triple quotes (either """ or '“).
- Triple quoted strings can contain both single quotes and double quotes as part of the string.
- Triple quotes can also be used to surround multiline strings, something for which single and double quotes cannot be used. Here is an example:

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

Printing Multiple Arguments

- The print() function can accept zero or more more arguments.
- If you decide to pass multiple arguments to the print() function you should separate them by a comma.
- Example: print ("Hello! My name is", "Ada")
- Note that when Python executes the function call above it will insert a space between the two arguments for you automatically.
- Also note that the print() function will automatically add a line break after it prints the last argument it was passed.

Line Endings

- When using the print() function you probably have noticed that Python automatically places a newline character at the end of each line.
- You can override this behavior and have Python use a character of your choice by using the optional '**end**' argument when using the print() function.
- Example:
  - print ('one', end='')
  - print ('two', end='')

Separating print() function arguments

- By default, Python will place a space between arguments that you use in print() function calls.
- You can override this behavior by using the optional 'sep' argument.
- Example:
  - print ('one', 'two', sep='*')

Line Endings and Separators

- You can use both the 'sep' and the 'end' arguments at the same time in the same print() function call.
- Example:
  - print ('a', 'b', 'c', sep='*', end='')

Escape Characters

- An *escape character* is a special character that is preceded with a backslash (\), appearing inside a string literal.

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.

Comments

- Comments are short notes placed in different parts of a program, explaining how those parts of the program work.
- Although comments are a critical part of a program, they are ignored by the Python interpreter.
- Comments are intended for any person reading a program's code, not the computer.

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

Program Output

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

- Programmers commonly write end-line comments in their code.
- An *end-line comment* is a comment that appears at the end of a line of code.
- It usually explains the statement that appears in that line.

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

Program Output

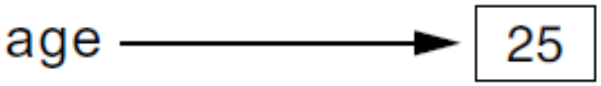
```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Variables

- Programs use variables to access and manipulate data that is stored in memory.
- A *variable* is a name that represents a value in the computer's memory.
- For example
  - a program that calculates the sales tax on a purchase might use the variable name tax to represent that value in memory.
  - a program that calculates the distance between two cities might use the variable name distance to represent that value in memory.
- When a variable represents a value in the computer's memory, we say that the variable *references* the value.

Creating Variables with Assignment Statements

- You use an ***assignment statement*** to create a variable and make it reference a piece of data.
- age = 25
- After this statement executes, a variable named age will be created, and it will reference the value 25.



- The arrow that points from **age** to the value 25 indicates that the name **age** references the value.

- An assignment statement is written in the following general format:

  variable = expression

- The equal sign (=) is known as the ***assignment operator***.
- **variable** is the name of a variable and **expression** is a value.
- After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

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

Program Output

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

Variable Naming Rules

- Although you are allowed to make up your own names for variables, you must follow these rules:
  - You cannot use one of Python's key words as a variable name.
  - A variable name cannot contain spaces.
  - The first character must be one of the letters a through z, A through Z, or an underscore character (_).
  - After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
  - Uppercase and lowercase characters are distinct. This means the variable name ItemsOrdered is not the same as itemsordered.

- you CAN'T use the following reserved words/keywords when declaring a variable in your program.

'False','None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue','def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global','if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass','raise', 'return', 'try', 'while', 'with', 'yield'

- A variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words.

- For example, consider the following variable names:
  - grosspay
  - payrate
  - hotdogssoldtoday
- Unfortunately, these names are not easily read by the human eye because the words aren't separated.
- Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name.
- One way to do this is to use the underscore character to represent a space.

- For example
  - gross_pay
  - pay_rate
  - hot_dogs_sold_today
- This style of naming variables is popular among Python programmers.
- There are other popular styles, however, such as the *camelCase* naming convention.
- camelCase names are written in the following manner:
  - The variable name begins with lowercase letters.
  - The first character of the second and subsequent words is written in uppercase.

- For example, the following variable names are written in camelCase:
  - grossPay
  - payRate
  - hotDogsSoldToday

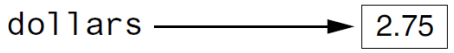
Variable Name	Legal or Illegal?
`units_per_day`	
`dayOfWeek`	
`3dGraph`	
`June1997`	
`Mixture#3`	

Variable Reassignment

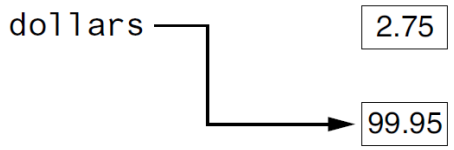
- Variables are called "variable" because they can reference different values while a program is running.
- When you assign a value to a variable, the variable will reference that value until you assign it a different value.

```
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.



The dollars variable after line 8 executes.



Multiple Variables

- It's possible to set the value of multiple variables on the same line. For example:
  - x, y, z = 'a', 'b', 'c'
- You can also assign the same value to multiple variables at the same time by doing the following:
  - a = b = c = 100

Numeric Data Types and Literals

- Python uses *data types* to categorize values in memory.
  - e.g., int for integer, float for real number, str used for storing strings in memory
- A number that is written into a program's code is called a *numeric literal*.
- Some operations behave differently depending on data type
- How Python determines the data type of a number?

- When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:
  - A numeric literal that is written as a whole number with no decimal point is considered an int. Examples are 7, 124, and −9.
  - A numeric literal that is written with a decimal point is considered a float. Examples are 1.5, 3.14159, and 5.0.
- As an experiment, you can use the built-in **type** function in interactive mode to determine the data type of a value.

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

Storing Strings with the str Data Type

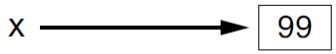
- In addition to the int and float data types, Python also has a data type named **str**, which is used for storing strings in memory.

Reassigning a Variable to a Different Type

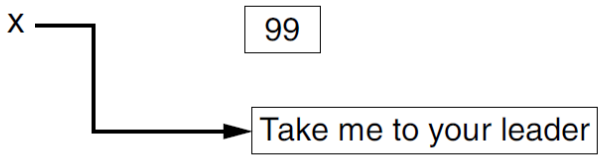
- A variable is just a name that refers to a piece of data in memory.
- The Python interpreter keeps track of the variable names that you create and the pieces of data to which those variable names refer.
- A variable in Python can refer to items of any type.
- A variable has been assigned an item of one type, it can be reassigned an item of a different 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



The variable x references a string



Challenge

- You're working on a simple inventory management system for a small store.
- You'd like to store the name and price of two different products and print them out in the following format.

Item: Bread, Price: $ 2.99

Item: Eggs, Price: $ 1.99

Reading Input from the Keyboard

- Programs commonly need to read input typed by the user on the keyboard. We will use the Python functions to do this.
- We use Python's built-in **input** function to read input from the keyboard.
- The input function reads a piece of data that has been entered at the keyboard and returns that piece of data, **as a string**, back to the program.
- use the input function in an assignment statement that follows this general format:

variable = input(*prompt*)

- **prompt** is a string that is displayed on the screen.
- **variable** is the name of a variable that references the data that was entered on the keyboard.
- Here is an example of a statement
  - name = input('What is your name? ')

```
1  # Get the user's first name.
2  first_name = input('Enter your first name: ')
3
4  # Get the user's last name.
5  last_name = input('Enter your last name: ')
6
7  # Print a greeting to the user.
8  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

- The input function always returns the user's input as a string, even if the user enters numeric data.
- Python has built-in functions that you can use to convert a string to a numeric type.

Function	Description
`int(item)`	You pass an argument to the `int()` function and it returns the argument's value converted to an `int`.
`float(item)`	You pass an argument to the `float()` function and it returns the argument's value converted to a `float`.

- For example, suppose you are writing a payroll program and you want to get the number of hours that the user has worked. Look at the following code:

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

- This one statement does all the work that the previously shown two statements do, and it creates only one variable:

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

- This one statement uses *nested function* calls.

Challenge

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
```

- The int() and float() functions work only if the item that is being converted contains a valid numeric value.
- If the argument cannot be converted to the specified data type, an error known as an exception occurs.
- An *exception* is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with.

```
>>> 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'
>>>
```

Performing Calculations

- A programmer's tools for performing calculations are *math operators*.
- A *math expression* performs a calculation and gives a value.
- The values on the right and left of the + operator are called *operands*.

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

- Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

  1. *Get the original price of the item.*
  2. *Calculate 20 percent of the original price. This is the amount of the discount.*
  3. *Subtract the discount from the original price. This is the sale price.*
  4. *Display the sale price.*

Program Output (with input shown in bold)

```
Enter the item's original price: 100.00 Enter
The sale price is 80.0
```

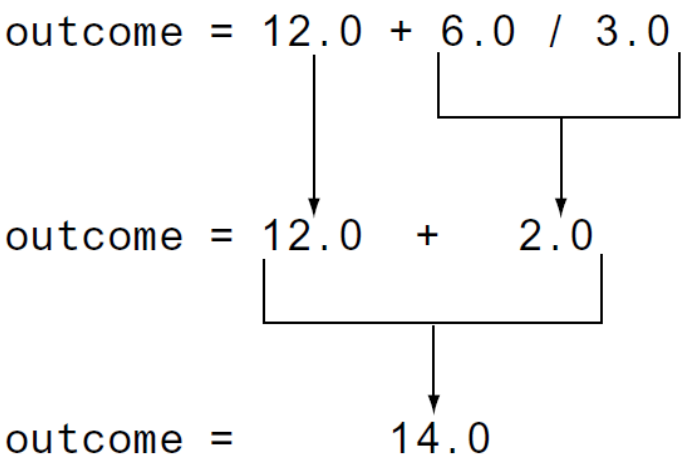
Floating-Point and Integer Division

- Python has two different division operators.
  - The / operator performs floating-point division.
  - The // operator performs integer division.
- The difference between them is that the / operator gives the result as a floating-point value, and the // operator gives the result as a whole number.

Operator Precedence

- You can write statements that use complex mathematical expressions involving several operators.
- Python follows the same order of operations that you learned in math class.
- First, operations that are enclosed in parentheses are performed first.
- Then, when two operators share an operand, the operator with the higher *precedence* is applied first.
- The precedence of the math operators, from highest to lowest, are:
  1. Exponentiation: **
  2. Multiplication, division, and remainder: * / // %
  3. Addition and subtraction: + −

- Notice the multiplication (*), floating-point division (/), integer division (//), and remainder (%) operators have the same precedence.
- The addition (+) and subtraction (−) operators also have the same precedence.
- When two operators with the same precedence share an operand, the operators execute from left to right.



Grouping with Parentheses

- Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others.

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

- Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. *Get the first test score.*
2. *Get the second test score.*
3. *Get the third test score.*
4. *Calculate the average by adding the three test scores and dividing the sum by 3.*
5. *Display the average.*

Program Output (with input shown in bold)

```
Enter the first test score: 90 Enter
Enter the second test score: 80 Enter
Enter the third test score: 100 Enter
The average score is 90.0
```

Challenge

- Ask the user to input a number of seconds as a whole number. Then express the time value inputted as a combination of minutes and seconds.

  Enter seconds: 110

  That's 1 minute and 50 seconds

The Exponent Operator

- Two asterisks written together (**) is the exponent operator, and its purpose is to raise a number to a power.

```
>>> 4**2 Enter
16
>>> 5**3 Enter
125
>>> 2**10 Enter
1024
>>>
```

The Remainder Operator

- In Python, the % symbol is the remainder operator. (This is also known as the *modulus operator*.)

```
# 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)
```

Converting Math Formulas to Programming Statements

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`

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)`

- Suppose you want to deposit a certain amount of money into a savings account and leave it alone to draw interest for the next 10 years. At the end of 10 years, you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

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

- The terms in the formula are as follows:
  - *P* is the present value, or the amount that you need to deposit today.
  - *F* is the future value that you want in the account. (In this case, *F* is $10,000.)
  - *r* is the annual interest rate.
  - *n* is the number of years that you plan to let the money sit in the account.

Program Output

```
Enter the desired future value: 10000.0 Enter
Enter the annual interest rate: 0.05 Enter
Enter the number of years the money will grow: 10 Enter
You will need to deposit this amount: 6139.13253541
```

Displaying Multiple Items with the + Operator

- The + operator is used to add two numbers.
- When the + operator is used with two strings, however, it performs ***string concatenation***.
- This means that it appends one string to another.
- For example
  - print('This is ' + 'one string.')

Mixed-type Expressions and Data Type Conversion

- Python follows these rules when evaluating mathematical expressions:
  1. When an operation is performed on two int values, the result will be an int.
  2. When an operation is performed on two float values, the result will be a float.
  3. When an operation is performed on an int and a float, the int value will be temporarily converted to a float and the result of the operation will be a float.
- An expression that uses operands of different data types is called a ***mixed-type expression***.

Breaking Long Statements into Multiple Lines

- Long statements cannot be viewed on screen without scrolling and cannot be printed without cutting off.
- Python allows you to break a statement into multiple lines by using the *line continuation character*, which is a backslash (\).
- For example, here is a statement that performs a mathematical calculation

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

- Python also allows you to break any part of a statement that is enclosed in parentheses into multiple lines without using the line continuation character.
- For example

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

- Another example

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

Formatting Numbers

- You might not always be happy with the way that numbers, especially floating-point numbers, are displayed on the screen.
- When a floating-point number is displayed by the print function, it can appear with up to 12 significant digits.

```
1  # This program demonstrates how a floating-point
2  # number is displayed with no formatting.
3  amount_due = 5000.0
4  monthly_payment = amount_due / 12.0
5  print('The monthly payment is', monthly_payment)
```

Program Output

```
The monthly payment is 416.666666667
```

- Python gives us a way to do just that, and more, with the built-in format function.
- You pass two arguments to the function: a numeric value and a format specifier.
- The *format specifier* is a string that contains special characters specifying how the numeric value should be formatted.
- An example
  - format(12345.6789, '.2f')
  - The .2 specifies the precision. It indicates that we want to round the number to two decimal places.
  - The f specifies that the data type of the number we are formatting is a floating-point number.

Formatting in Scientific Notation

- If you prefer to display floating-point numbers in scientific notation, you can use the letter e or the letter E instead of f.

```
>>> print(format(12345.6789, 'e')) Enter
1.234568e+04
>>> print(format(12345.6789, '.2e')) Enter
1.23e+04
>>>
```

Inserting Comma Separators

- If you want the number to be formatted with comma separators, you can insert a comma into the format specifier.

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

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

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.

```
In [31]: print('The number is', format(12345.6789, '12,.2f'))
The number is    12,345.68

In [32]: print('The number is', format(12345.6789, '12.2f'))
The number is     12345.68
```

Formatting a Floating-point Number as a Percentage

- You can use the % symbol to format a floatingpoint number as a percentage.

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

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

Formatting Integers

- There are two differences to keep in mind when writing a format specifier that will be used to format an integer:
  - You use **d** as the type designator.
  - You cannot specify precision.

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

Introduction to Turtle Graphics

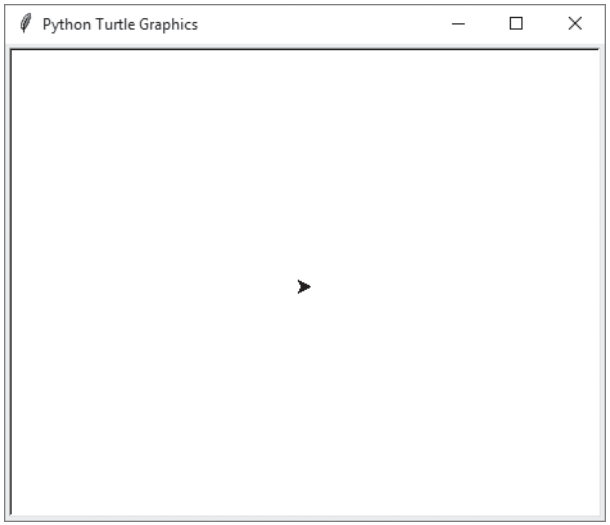
- Python has a *turtle graphics* system that simulates a robotic turtle.
- The system displays a small cursor (the turtle) on the screen. You can use Python statements to move the turtle around the screen, drawing lines and shapes.
- The first step in using Python's turtle graphics system is to write the following statement:

```
import turtle
```

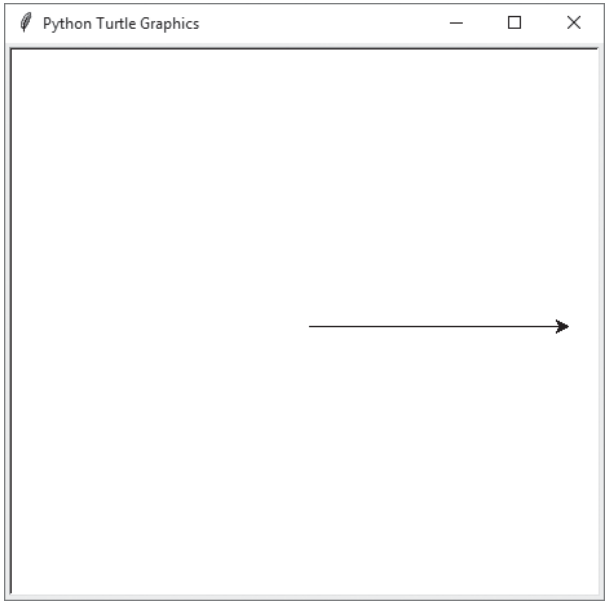
- The import turtle statement loads the turtle module into memory so the Python interpreter can use it.

Drawing Lines With the Turtle

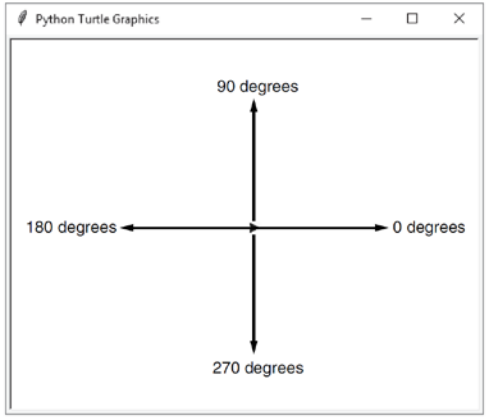
- you can enter the **turtle.showturtle()** command to display the turtle in its window.
- Here is an example
  - >>> import turtle
  - >>> turtle.showturtle()
  - >>> turtle.done()
  - >>> turtle.Terminator

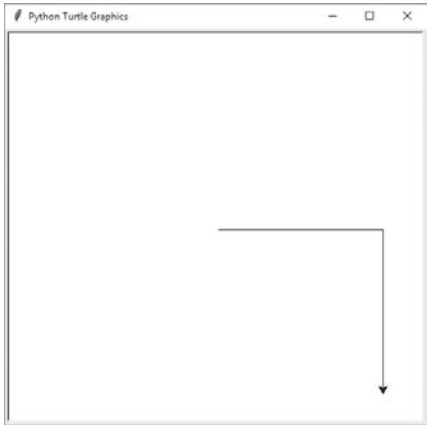


- If we instruct the turtle to move forward, it will move in the direction that the arrowhead is pointing.
- You can use the **turtle.forward(*n*)** command to move the turtle forward *n* pixels.
- Here is an example
  - >>> import turtle
  - >>> turtle.forward(200)
  - >>> turtle.done()
  - >>> turtle.Terminator



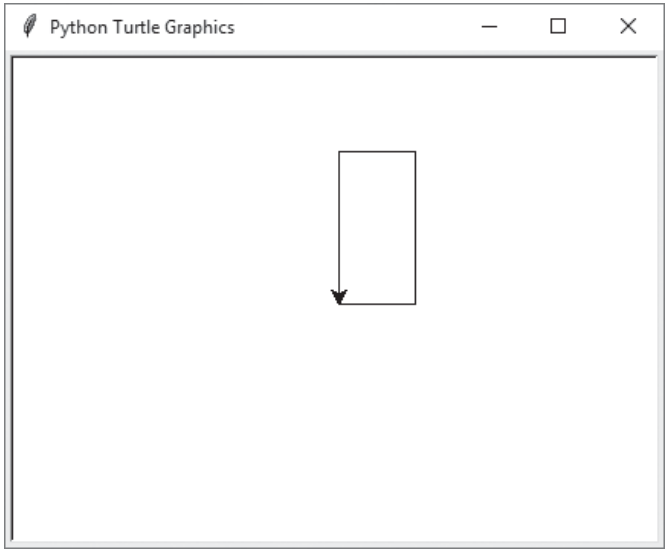
- When the turtle first appears, its default heading is 0 degrees (East).
- You can turn the turtle so it faces a different direction by using either the **turtle.right(*angle*)** command, or the **turtle.left(*angle*)** command.
- Here is an example
  - >>> import turtle
  - >>> turtle.forward(200)
  - >>> turtle.right(90)
  - >>> turtle.forward(200)
  - >>> turtle.done()
  - >>> turtle.Terminator





Setting the Turtle's Heading to a Specific Angle

- You can use the **turtle.setheading(*angle*)** command 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)
  - >>> turtle.done()
  - >>> turtle.Terminator



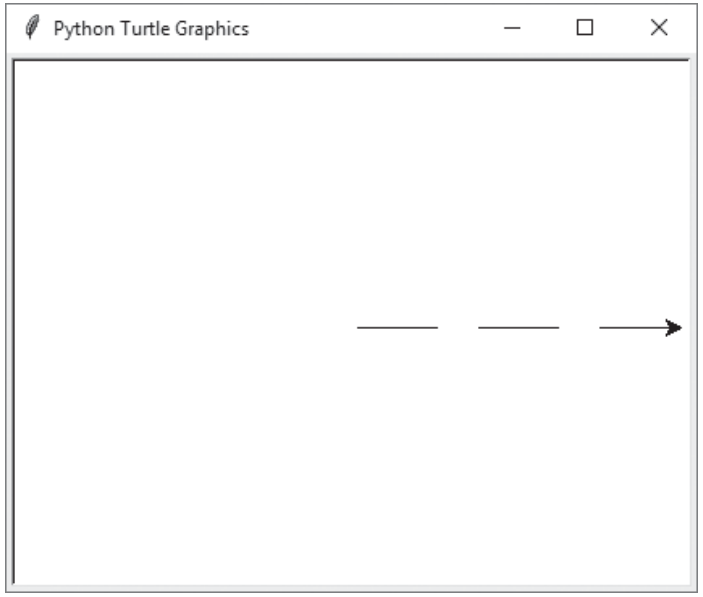
Getting the Turtle's Current Heading

- In an interactive session, you can use the **turtle.heading()** command to display the turtle's current heading.
- Here is an example:
  - >>> import turtle
  - >>> turtle.heading()
  - 0.0
  - >>> turtle.setheading(180)
  - >>> turtle.heading()
  - 180.0
  - >>> turtle.done()
  - >>> turtle.Terminator

Moving the Pen Up And Down

- When the pen was down, it was in contact with the paper and would draw a line as the turtle moved.
- When the pen was up, it was not touching the paper, so the turtle could move without drawing a line.
- In Python, you can use the turtle.penup() command to raise the pen, and the turtle. pendown() command to lower the pen.

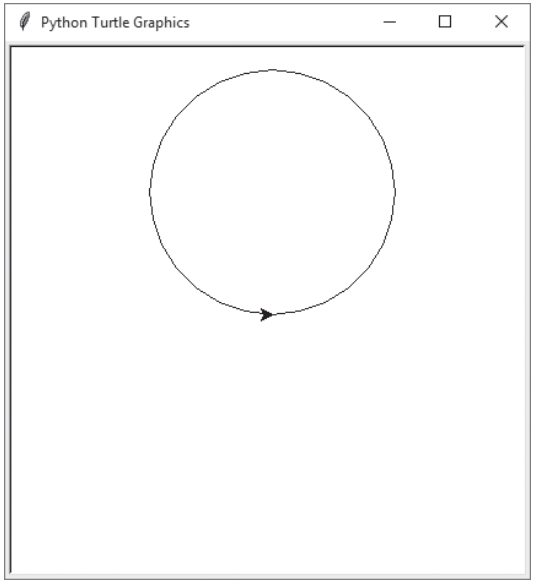
- >>> import turtle
- >>> turtle.forward(50)
- >>> turtle.penup()
- >>> turtle.forward(25)
- >>> turtle.pendown()
- >>> turtle.forward(50)
- >>> turtle.penup()
- >>> turtle.forward(25)
- >>> turtle.pendown()
- >>> turtle.forward(50)
- >>> turtle.done()
- >>> turtle.Terminator



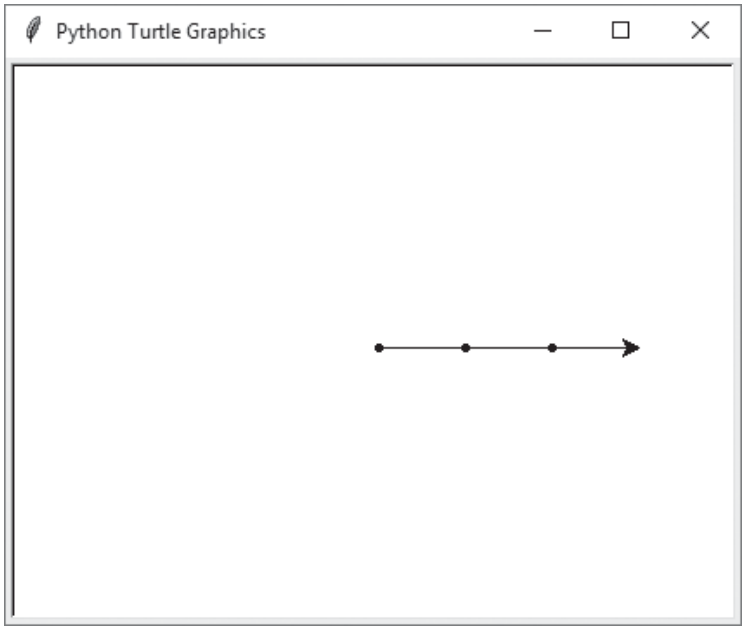
Drawing Circles and Dots

- You can use the **turtle.circle(*radius*)** command to make the turtle draw a circle with a radius of *radius* pixels.

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



- You can use the **turtle.dot()** command to make the turtle draw a simple dot.
  - >>> import turtle
  - >>> turtle.dot()
  - >>> turtle.forward(50)
  - >>> turtle.dot()
  - >>> turtle.forward(50)
  - >>> turtle.dot()
  - >>> turtle.forward(50)
  - >>> turtle.done()
  - >>> turtle.Terminator



Changing the Drawing Color

- You can use the **turtle.pencolor(*color*)** command to change the turtle's drawing color.
  - >>> import turtle
  - >>> turtle.pencolor('red')
  - >>> turtle.circle(100)
  - >>> turtle.done()
  - >>> turtle.Terminator
- Some of the more common colors are 'red', 'green', 'blue', 'yellow', and 'cyan'.

Changing the Background Color

- You can use the **turtle.bgcolor(*color*)** command to change the background color of the turtle's graphics window.
  - >>> import turtle
  - >>> turtle.bgcolor('gray')
  - >>> turtle.pencolor('red')
  - >>> turtle.circle(100)
  - >>> turtle.done()
  - >>> turtle.Terminator

Resetting the Screen

- There are three commands that you can use to reset the turtle's graphics window:
  - The **turtle.reset()** command erases all drawings that currently appear in the graphics window, resets the drawing color to black, and resets the turtle to its original position in the center of the screen. This command does not reset the graphics window's background color.
  - The **turtle.clear()** command simply erases all drawings that currently appear in the graphics window. It does not change the turtle's position, the drawing color, or the graphics window's background color.
  - The **turtle.clearscreen()** command erases all drawings that currently appear in the graphics window, resets the drawing color to black, reset the graphics window's background color to white, and resets the turtle to its original position in the center of the graphics window.

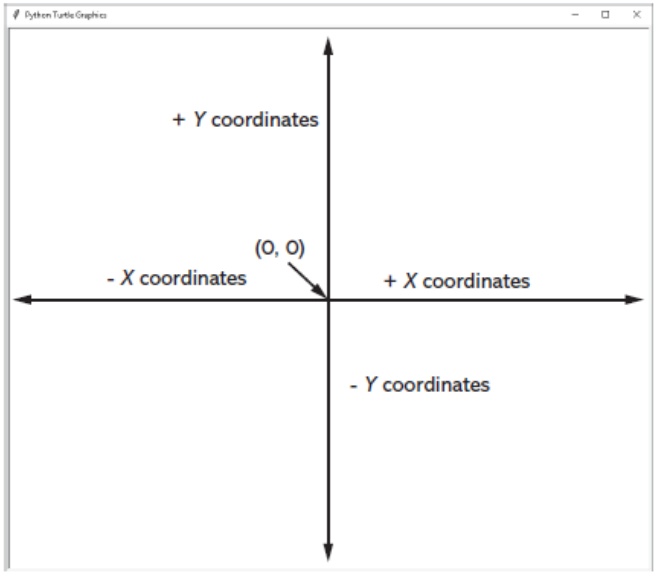
Specifying the Size of the Graphics Window

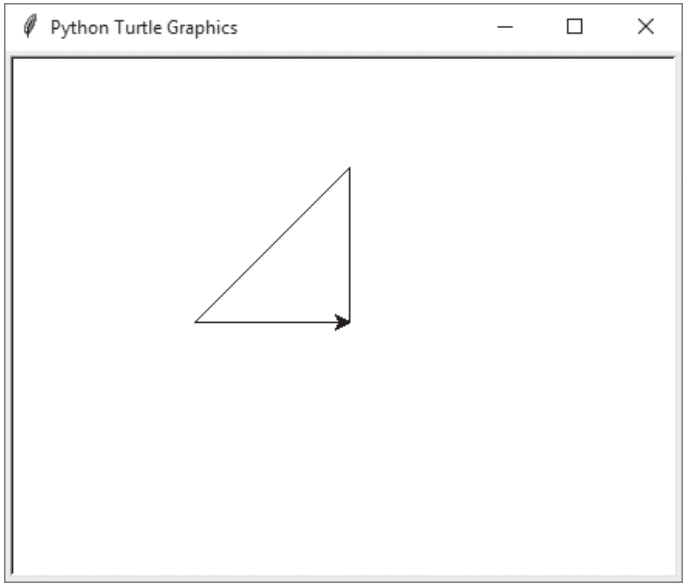
- You can use the **turtle.setup(*width*, *height*)** command to specify a size for the graphics window.
  - >>> import turtle
  - >>> turtle.setup(640, 480)
  - >>> turtle.done()
  - >>> turtle.Terminator

Moving the Turtle to a Specific Location

- A Cartesian coordinate system is used to identify the position of each pixel in the turtle's graphics window.
- Each pixel has an *X* coordinate and a *Y* coordinate.
- The *X* coordinate identifies the pixel's horizontal position, and the *Y* coordinate identifies its vertical position.
- The pixel in the center of the graphics window is at the position (0, 0), which means that its *X* coordinate is 0 and its *Y* coordinate is 0.
- You can use the **turtle.goto(*x*, *y*)** command to move the turtle from its current location to a specific position in the graphics window.

- >>> import turtle
- >>> turtle.goto(0, 100)
- >>> turtle.goto(−100, 0)
- >>> turtle.goto(0, 0)
- >>> turtle.done()
- >>> turtle.Terminator





Getting the Turtle's Current Position

- You can use the **turtle.pos()** command to display the turtle's current position.
  - >>> import turtle
  - >>> turtle.goto(100, 150)
  - >>> turtle.pos()
  - (100.00, 150.00)
  - >>> turtle.done()
  - >>> turtle.Terminator
- You can also use the **turtle.xcor()** command to display the turtle's *X* coordinate, and the **turtle.ycor()** command to display the turtle's *Y* coordinate.

Controlling the Turtle's Animation Speed

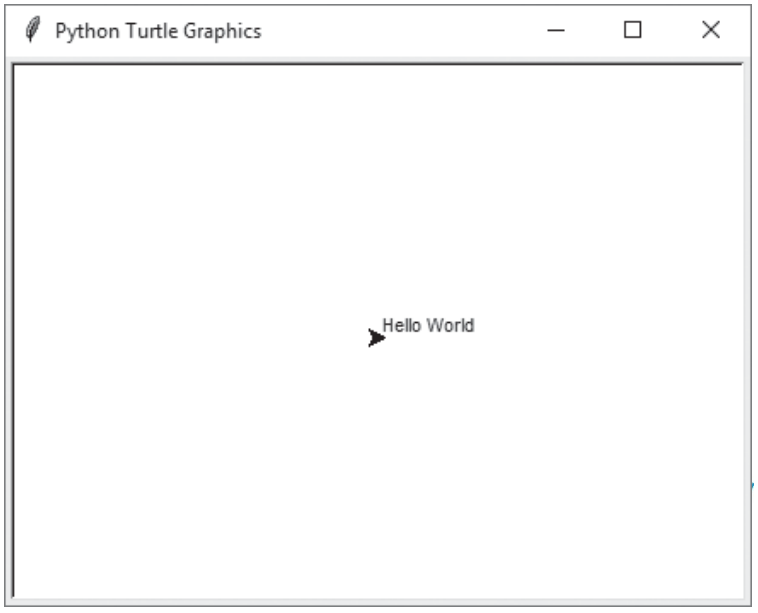
- You can use the turtle.speed(*speed*) command to change the speed at which the turtle moves.
- The *speed* argument is a number in the range of 0 through 10.
- If you specify 0, then the turtle will make all of its moves instantly (animation is disabled).
- If you specify a *speed* value in the range of 1 through 10, then 1 is the slowest speed, and 10 is the fastest speed.
  - >>> import turtle
  - >>> turtle.speed(1)
  - >>> turtle.circle(100)
  - >>> turtle.done()
  - >>> turtle.Terminator

Hiding the Turtle

- If you don't want the turtle to be displayed, you can use the turtle.hideturtle() command to hide it.
- This command does not change the way graphics are drawn, it simply hides the turtle icon.
- When you want to display the turtle again, use the turtle.showturtle() command.

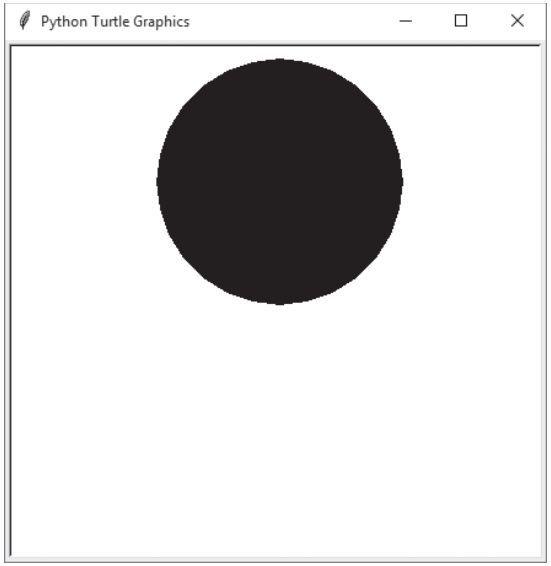
Displaying Text in the Graphics Window

- You can use the **turtle.write(text)** command to display text in the graphics window.
- When the string is displayed, the lower-left corner of the first character will be positioned at the turtle's X and Y coordinates.
  - >>> import turtle
  - >>> turtle.write('Hello World')
  - >>> turtle.done()
  - >>> turtle.Terminator

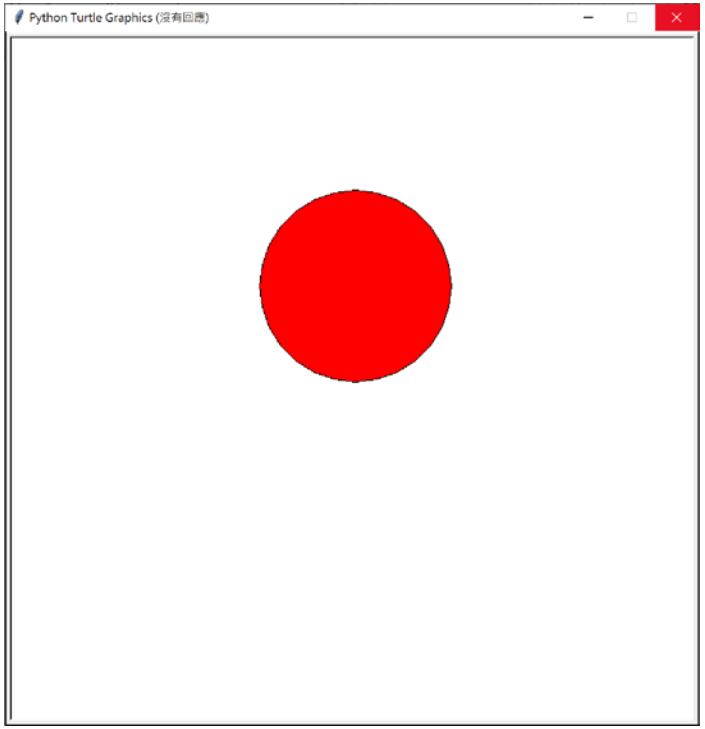


Filling Shapes

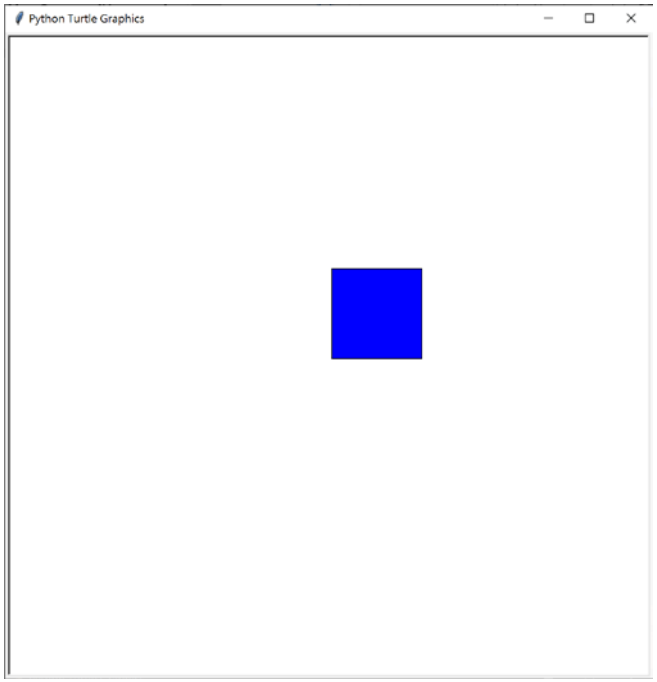
- To fill a shape with a color, you use the **turtle.begin_fill()** command before drawing the shape, then you use the **turtle.end_fill()** command after the shape is drawn.
  - >>> import turtle
  - >>> turtle.hideturtle()
  - >>> turtle.begin_fill()
  - >>> turtle.circle(100)
  - >>> turtle.end_fill()
  - >>> turtle.done()
  - >>> turtle.Terminator



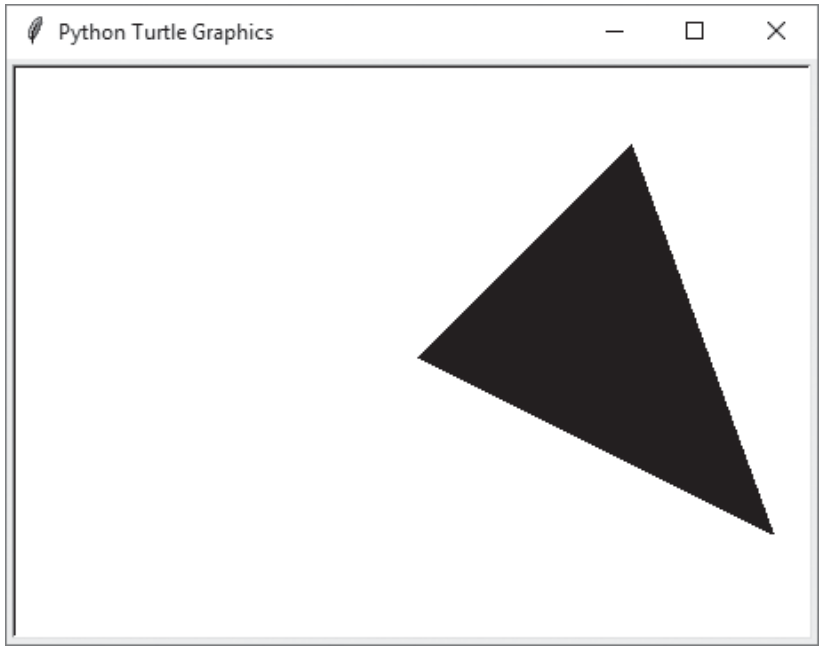
- You can change the fill color with the **turtle.fillcolor(*color*)** command.
  - >>> import turtle
  - >>> turtle.hideturtle()
  - >>> turtle.fillcolor('red')
  - >>> turtle.begin_fill()
  - >>> turtle.circle(100)
  - >>> turtle.end_fill()
  - >>> turtle.done()
  - >>> turtle.Terminator



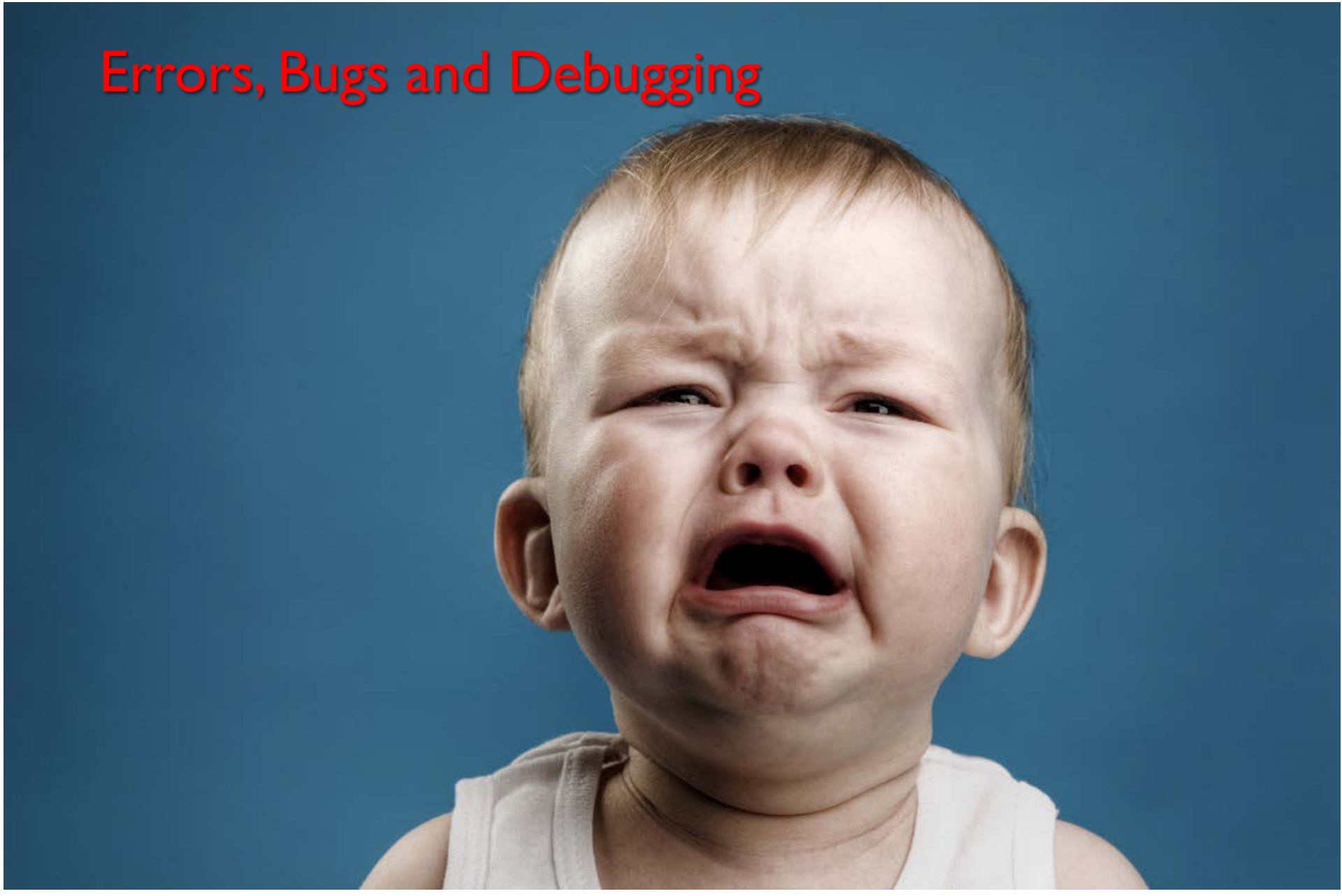
- >>> import turtle
- >>> turtle.hideturtle()
- >>> turtle.fillcolor('blue')
- >>> turtle.begin_fill()
- >>> turtle.forward(100)
- >>> turtle.left(90)
- >>> turtle.forward(100)
- >>> turtle.left(90)
- >>> turtle.forward(100)
- >>> turtle.left(90)
- >>> turtle.forward(100)
- >>> turtle.end_fill()
- >>> turtle.done()
- >>> turtle.Terminator



- >>> import turtle
- >>> turtle.hideturtle()
- >>> turtle.begin_fill()
- >>> turtle.goto(120, 120)
- >>> turtle.goto(200, –100)
- >>> turtle.end_fill()
- >>> turtle.done()
- >>> turtle.Terminator



Errors, Bugs and Debugging



Debugging

- De-bugging a program is the process of finding and resolving errors

Types of Errors

- We have **syntax errors** which is code that does not follow the rules of the language.
- i.e. We use a single quote where a double quote is needed… A colon is missing… or we use a keyword as a variable name.
- We have **runtime errors** which typically involves a program "crashing" or not running as expected.
- i.e. You are dividing two numbers but do not test for a zero divisor. This causes a run time error when the program tries to divide by zero.

Types of Errors

- We have **logic errors**. These tend to be harder to find and involve code that is syntactically correct, will run but the anticipated result is outright wrong.
- i.e. Your program prints "2+2=5"

Basic Debugging Techniques

- Set small, incremental goals for your program. Don't try and write large programs all at once.
- Stop and test your work often as you go. Celebrate small successes.
- Use comments to have Python ignore certain lines that are giving you trouble.