Ch. 2: Input, Processing, & Output

Wednesday, January 2, 2019 12:54 AM

Overview:
This chapter introduces the program development cycle, variables, data types, and simple programs that are written as sequence structures. The student learns to write simple programs that read input from the keyboard, perform mathematical operations, and produce screen output. Pseudocode and flowcharts are also introduced as tools for designing programs. The chapter also includes an optional introduction to the turtle graphics library.

2.1 Designing a Program:

- Programs must be carefully designed before they are written
- Program development cycle:
    - Design the program
        - When programmers begin a new project, they should never jump right in and start writing code as the first step. They start by creating a design of the program. There are several ways to design a program, and later in this section, we will discuss some techniques that you can use to design your Python programs.
    - Write the code
        - After designing the program, the programmer begins writing code in a language such as Python. Recall from Chapter 1 that each language has its own rules, known as syntax, that much be followed when writing a program. A language's syntax rules dictates things such as how key words, operators , and punctuation characters can be used. A syntax error occurs when a programmer violates any of these rules.
    - Correct syntax errors
        - If the program contains a syntax error, or even a simple mistake such as a misspelled keyword, the compiler or interpreter will display an error message indicating what (and sometimes where) the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used)
    - Test the program
        - Once the code is an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)
    - Correct logic errors
        - If the program produces incorrect results, the programmer *debugs* the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the program discovers that the program's original design must be changed. In this event, the program development cycle starts over and continues until no errors can be found.

- Design is the most important part of the program development cycle.
    - You can think of the program's design as its foundation. If you built a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. 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:
  - Understand the task that the program is to perform
    - Work with the customer to get a sense of what the program is supposed to do.
    - Ask questions about the program details.
    - Create one or more software requirements.
  - Determine the steps that must be taken to perform the task.
    - Break down the required task into a series of steps.
    - Create an algorithm, listing logical steps that must be taken.

- **Algorithm: A set of well-defined logical steps that must be taken to perform a task.**

- To get a sense of what the program is supposed to do, the program usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.
- The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A *software requirement* is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

Pseudocode:

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

Ex:

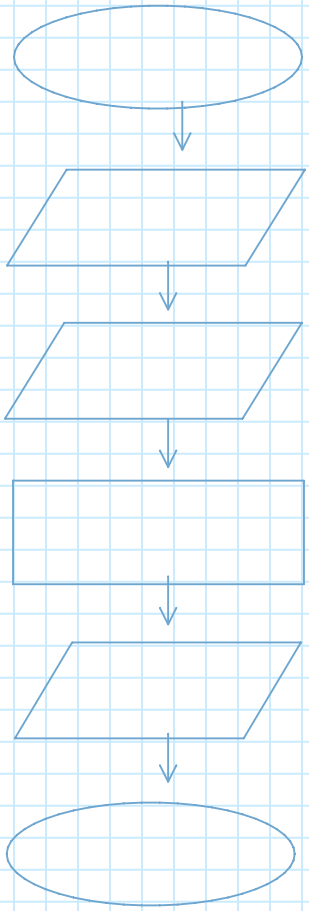
- Program to calculate employee pay.
  - *Input the hours worked*
  - *Input the hourly pay rate*
  - *Calculate gross pay as hours worked multiplied by pay rate*
  - *Display the gross pay*

Flowcharts:

Flowchart: A diagram that graphically depicts the steps in a program.

- Ovals are terminal symbols.
  - Ovals appear at the top & bottom of the flowchart and represent starting and ending points.
- Parallelograms are input and output symbols
  - Parallelograms represent steps in which the program needs to read or display output.
- Rectangles are processing symbols
  - Rectangles represent steps in which the program performs some process on data, such as a mathematical calculation.

- Symbols are connected by arrows that represent the flow of the program. They start with the beginning terminal oval and continue in order of the arrows until the ending terminal oval.



2.2: Input, Processing, & Output:

- Input is data that the program receives. When a program receives data, it usually processes it by performing some operation with it. The result of the operation is sent out of the program as output.
- Typically, the computer performs a three-step process:
  - Receive input.
    - **Input: Any data that the program receives while it is running.**
    - One common form of input is data that is typed on the keyboard.
  - Perform some process on the input
    - Ex: mathematical calculation
  - Produce output

2.3: Displaying Output with the *print* function:

- You can use the *print* function to display output in a Python program.
- **Function: Piece of prewritten code that performs an operation**
- ***Print* Function: Displays output on the screen.**
- **Argument: Data given to a function**
  - Example: data that is printed to a screen
- **Statements in a program execute in the order that they appear, from top to bottom.**

Strings & String Literals

- **String: A sequence of characters that is used as data.**
- **String literal: A string that appears in actual code of a program.**
    - Must be enclosed in a 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.

2.4: Comments

- **Comments: Notes of explanation within a program**
    - Ignored by the Python interpreter
        - Intended for a person reading the program's code.
    - Begins with the # character
    - Can also make block comment with (''')
- **End-line comment: Appears at the end of a line of code**
    - Typically explains the purpose of that line

2.5: Variables

- **Variable: A name that represents a value stored in the computer memory.**
    - Used to access and manipulate data stored in the memory
    - A variable references the value it represents
    - A variable can be passed as an argument to a function.
        - Variable name should not be enclosed in quotation marks.
- **Assignment statement: A statement used to create a variable and make it reference data.**
    - General format is *variable = expression.*
        - Example: *Age = 25*
    - Assignment operator: one equal sign (=)
- In any of these cases, the variable is created and then assigned to a value or expression which is then stored in the computer's memory.
- In an assignment statement, the variable that is receiving the assignment must appear on the left side of the assignment (=) operator.
- You cannot use a variable until you have assigned a value to it. An error will occur is you try to perform an operation on a variable, such as printing it, before it has been assigned to a variable.
- Python is case sensitive. So, VariableName =! variablename.
- Also, spelling *always* matters, so be sure you are spelling your variable names correctly when calling them.

  VARIABLE NAMING RULES:
    - No key words can be used as a variable name.
    - A variable name cannot contain spaces.
    - The first character must be one of the letters a-z, A-Z, or an underscore character (_).
    - After the first character you may use the letters a-z or A-Z, the digits 0-9, and underscores.
    - Uppercase and lowercase letters are distinct.
- In addition to these rules, variable names should always be meaningful and have an indication of what they are used for in the program.
- Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words.
    - Ex: *grosspay, payrate*
- Unfortunately, these names are not necessarily easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to come up with another way to make variable names with multiple words more readable.

- There are several solutions. One includes using underscores between words.
  - Ex: *gross_pay, pay_rate*
- There is also the camelCase naming convention, which is popular among python programmers.
  - Ex: *grossPay, payRate*
  - This style is called camelCase because the uppercase characters may resemble a camel's hump.

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 are displayed in the order in which they are passed to the function.
  - Items are automatically separated by a space when displayed on the screen.
    - Ex:

```
Room = 500
Print("I am staying in room number")
Print(room)

OR

Room = 500
Print("I am staying in room number", room)
```

Variable Reassignment

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

Numeric Data Types, Literals, and the *str* Data Type:

Because different types of numbers are stored and manipulated in different ways, python uses data types to categorize values in memory. When an integer is stored in memory, it is classified as an *int* and when a real number is stored in the memory, it is classified as a *float*.

- **Data types: categorize value in memory**
  - Ex; int for integer, float for real number, str for storing strings in the memory.
- **Numeric literal: number written in a program**
  - Whole numbers (that have no decimal points) are considered int, otherwise, the number is considered to be a float.
- Some operations behave differently depending on the data type.

Reassigning a Variable to a Different Type:

- **Keep in mind that in Python, a variable is just a name that refers to a piece of data in memory. It is a mechanism that makes it easier for you, the programmer, to store and retrieve data.**
- Internally, the Python interpreter keeps track of the variable names that you create and the pieces of data to which those variable names refer. Any time you need to retrieve that piece of data, you simply use the variable name that refers to it.
- A variable in Python can refer to items of any type.
- Variables are called variables 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.
- A value can still be in the computer's memory, but inaccessible because the variable is now assigned to a different value.

Reading Input from the Keyboard:
- Most programs need to read input from the user.
- The built-in *input* function reads input from the 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 Numbers with the *input* Function:
- *Input* function always returns a string.
- Built-in functions convert between data types.
  - *Int(item)* converts *item* to *int.*
  - *Float(item)* converts *item* to a *float.*
  - Nested function call general format: *function1(function2(argument))*
    - Value returned by function 2 is passed to function 1
  - Type conversion only works if the item is a valid numeric value, otherwise, it throws an exception.

Performing Calculations:
- **Math expression: performs calculations and gives a value**
- **Math operator: tool for performing calculation**
- **Operands: values surrounding operator**
  - Variables can be used as operands
- The resulting value is typically assigned to a variable
- **TWO TYPES OF DIVISION:**
  - **/ operator performs floating point division**
  - **// operator performs integer division**
    - **When the result is positive, the result is *truncated,* or thrown away.**
    - **When the result is negative, it is *rounded away from zero* to the nearest integer.**

Operator Precedence and Grouping with Parentheses:
- **Python operator precedence:**
  1. Operations enclosed in parentheses
     i. 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**
  - AKA the modulus operator
  - Ex; 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 the data types of the operands.**
  - Two *int* values: result is an *int*
  - Two *float* values: result is a *float*
  - *Int* and *float: int* temporarily converted to *float,* result of the operation is a *float*.
    - Mixed-type expression
  - Type conversion of *float* to *int* causes truncation of fractional part

Breaking Long Statements into Multiple Lines:

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

```
Result = var1 * 2 + var2 * 3\
        Var3 * 4 + var4 * 5
```

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

More About Data Output:

- *Print* function displays lines of output
  - Newline character at the end of printed data
    - Special argument *end = 'delimiter'* causes *print* to place *delimiter* at the end of data instead of a newline character.
  - *Print* function uses space as item separator.
    - Special argument *sep = 'delimiter'* causes *print* to use *delimiter* as an item separator
- Special characters appearing in string literal
  - Preceded by backslash (\)
    - Ex; newline (\n), horizontal tab (\t)
  - Treated as commands embedded in string
- Earlier, we learned that the + operator can be used to add two numbers.
- When + operator is used on two strings, it performs string concatenation, or appends one string to the end of another.
  - This is useful for breaking up a long string literal or substituting a variable value at the end of a string.

Formatting Numbers:

- You may not always be happy with the way numbers appear on the screen in the output. When a floating point number is displayed by the *print* function, it can appear with up to 12 significant digits.
- Can format display of numbers on screen using built-in *format* function
  - Two arguments:
    - Numeric value to be formatted
    - Format specifier
      - A string that contains special characters specifying how the numeric value should be formatted
  - The format function 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
- If you wish to display numbers in scientific notation, 'e' or 'E' can be used instead of 'f'.
    - The % symbol can be used in the format string of *format* function to format number as a percentage
    - To format an integer using the *format* function:
        - Use *d* as the type designator
        - Do not specify precision
        - Can still use *format* function to set field width or comma separator
- **Minimum field width: minimum number of spaces that should be used to display the value.**
    - Value comes before the decimal precision.
- To format a floating number as a percentage, you can use the % symbol to format.
- The % symbol causes the number to be multiplied by 100 and displayed with a % sign following it.

Magic Numbers:

- A magic number is an unexplained numeric value that appears in a program's code.
- 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 your program's code.
    - For example, suppose you intend to type 0.078 but you accidentally type 0.0078. This mistake will cause mathematical errors that can be difficult to find

Named Constants:

- You should used 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.

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 it used)
- Named constants help prevent typographical errors that are common when using magic numbers.

We Covered:

- The program development cycle, tools for program design, and the design progress
- 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