Chapter 2

Software Development, Data Types, and Expressions

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 2 covers topics related to software development, data types, and expressions. Students first learn about the basic phases of the software development process. Next, they learn how to use strings, integer, and floating point numbers in Python. Simple as well as mixed-mode arithmetic expressions are covered. Students will also learn how to initialize and use variables with appropriate names. Students learn about modules, how to import functions from library modules, and how to call functions with arguments and use the returned values appropriately. Finally, they learn how to construct a simple Python program that performs inputs, calculations, and outputs, as well as how to use docstrings to document Python programs.

Objectives

After completing this chapter, students will be able to:

- Describe the basic phases of software development: analysis, design, coding, and testing
- Use strings for the terminal input and output of text
- Use integers and floating point numbers in arithmetic operations
- Construct arithmetic expressions
- Initialize and use variables with appropriate names
- Import functions from library modules
- Call functions with arguments and use returned values appropriately
- Construct a simple Python program that performs inputs, calculations, and outputs
- Use docstrings to document Python programs

Teaching Tips

The Software Development Process

1. Explain that there are several approaches to *software development*, the *waterfall model* being one of them.

2. Use Figure 2.1 to describe the role of each of the phases of the waterfall model.

3. Stress that modern software development is usually *incremental* and *iterative*. Introduce the term *prototype*.

4. Note that programs rarely work as hoped the first time they are run. Use Figure 2.2 to stress that it is important to perform extensive and careful testing in all of the development phases.

5. Use Figure 2.3 to explain that the cost of developing software is not spread equally over the phases.

Teaching Tip	For more information on the software development process, visit http://www.selectbs.com/analysis-and-design/what-is-a-software-development-process.

Case Study: Income Tax Calculator

1. Guide students as they step through this case study.

Request

1. Explain that the customer requests a program that computes a person's income tax.

Analysis

1. Stress that the inclusion of an interface at this point (see Figure 2.4) is a good idea because it allows the customer and the programmer to discuss the program's intended behavior in a context understandable to both.

Design

1. Ask your students to try to code the algorithm before they take a look at the code provided in the next section.

2. Explain the concept of *pseudocode* and how it can be used to create a design regardless of the specific programming language that will be used for implementation.

Implementation (Coding)

1. Explain that pseudocode and other design elements make it easier for a programmer to create a program that carries out the required computation. Pseudocode also allows for greater coding accuracy.

2. Stress that it is always important to provide descriptive comments to document a program.

Testing

1. Did the program work as expected the first time? Ask students to list the errors they had (if any) and describe how they fixed them.

2. Explain the purpose of testing, making sure to stress the importance of thorough testing and of planning the testing process.

3. Explain how to test a program using the IDLE window.

4. Explain the difference between *syntax errors* and *logic* or *design errors*. Explain the term *correct program*, distinguishing it from a program with logic or design errors.

5. Explain the purpose of a *test suite*, and how it represents all inputs. Use Table 2.1 to demonstrate the creation of a test suite and to explain how to determine what values should be included in a test suite for a specific program.

Strings, Assignment, and Comments

1. Use this section to introduce the use of strings for the output of text and the documentation of Python programs.

Data Types

1. Introduce the terms *data type*, *literal*, and *numeric data type*, emphasizing the differences and relationship between them. Table 2.2 lists some Python data types.

String Literals

1. Explain the term string literal and use real examples to illustrate how to create string literals in Python.

2. Introduce the term *empty string*, and be sure to distinguish the empty string from a string containing only white spaces.

Teaching Tip	For more information about string literals, visit http://docs.python.org/ref/strings.html.

3. Show students how to create strings that span multiple-lines, and introduce them to the *newline character*, \n.

Escape Sequences

1. Define the term *escape sequence*, and use Table 2.3 to introduce some of the most useful escape sequences in Python.

String Concatenation

1. Using one or more examples, demonstrate how to perform string concatenation in Python using the + operator.

2. Explain that the * operator allows you to build a string by repeating another string a given number of times.

Variables and the Assignment Statement

1. Remind students the meaning of the term *variable*.

2. Explain the naming rules that apply to variables in Python. Provide several examples to illustrate each of these rules.

Teaching Tip	CamelCase is a standard identifier naming convention for several programming languages. For more information, visit http://www.c2.com/cgi/wiki?CamelCase.

3. Introduce the term *symbolic constant*, and note that programmers usually use all uppercase letters to name symbolic constants.

4. Use one or more examples to show how to write an *assignment statement* in Python. Introduce the term *variable reference* and explain the difference between *defining* or *initializing* a variable and variable references.

5. Explain the purposes variables have in a program. Be sure to explain the term *abstraction* in this context.

Program Comments and Docstrings

1. Explain the purpose and importance of *program comments*.

2. Provide examples of how to include *docstrings* and *end-of-line* comments in Python. Stress that using these types of comments appropriately is important.

3. Review the guidelines for creating good documentation of code. You can use the list on Page 53 of the text as a guide.

Teaching Tip	The pydoc module can be used to display information about a Python object, including its docstring. For more information, visit http://epydoc.sourceforge.net/docstrings.html.

Quick Quiz 1

1. Modern software development is usually incremental and _____.
   Answer: iterative

2. What is a correct program?
   Answer: A correct program produces the expected output for any legitimate input.

3. True or False: In programming, a data type consists of a set of values and a set of operations that can be performed on those values.
   Answer: True

4. The newline character _____ is called an escape sequence.
   Answer: \n

Numeric Data Types and Character Sets

1. Explain that the following section provides a brief overview of numeric data types and their cousins, character sets.

Integers

1. Explain how integer literals are written in Python.

Floating-Point Numbers

1. Explain that real numbers have infinite precision, and point out that this is not possible on a computer due to memory limits.

2. Use Table 2.4 to introduce the use of *floating-point* numbers in Python. Explain that these numbers can be written using *decimal notation* or *scientific notation*.

Teaching Tip	Complex numbers are also supported in Python. For more information, visit http://docs.python.org/library/stdtypes.html.

Character Sets

1. Explain that in Python, character literals look just like string literals and are of the string type. Point out that character literals belong to several different *character sets*, among them the *ASCII set* and the *Unicode set*.

2. Provide a brief overview of the ASCII character set, using Table 2.5 as a guide.

Teaching Tip	You can find more information about Python's Unicode support at http://www.amk.ca/python/howto/unicode.

3. Use a few examples to show how to convert characters to and from ASCII using the `ord` and `chr` functions.

Expressions

1. Explain that *expressions* provide an easy way to perform operations on data values to produce other values.

2. Note that when entered at the Python shell prompt, an expression's operands are evaluated first. The operator is then applied to these values to compute the final value of the expression.

Arithmetic Expressions

1. Introduce the term *arithmetic expression*. Use Table 2.6 to describe the arithmetic operators available in Python.

2. Explain the difference between binary operators and unary operators. Give examples of each type of operator.

3. Briefly list the *precedence rules* that apply to arithmetic operators, using the list on Pages 58-59 as a guide. Be sure to explain the meaning of the terms *left associative* and *right associative*, and to point out that you can use parentheses to change the order of evaluation in an arithmetic expression.

4. Use Table 2.7 to show a few examples of how arithmetic expressions are evaluated. Introduce the term *semantic error* and clearly explain the difference between a semantic error and a syntax error.

5. Stress that when both operands of an expression are of the same numeric type, the resulting value is of that type; when each operand is of a different type, the resulting value is of the more general type.

6. Explain how the backslash character \ can be used to break an expression onto multiple lines.

Mixed-Mode Arithmetic and Type Conversions

1. Use a few examples to show how *mixed-mode arithmetic* can be problematic. Show how to use a *type conversion function* (see Table 2.8) to solve this problem.

2. Stress that the `int` function converts a `float` to an `int` by truncation, not by rounding. Show how to use the `round` function in cases when rounding is desirable.

3. Use one or more examples to show that type conversion also occurs in the construction of strings from numbers and other strings. Explain how to use the `str` function to solve this problem.

4. Explain that Python is a *strongly typed programming language*.

Quick Quiz 2

1. Real numbers have _____ precision, which means that the digits in the fractional part can continue forever.
   Answer: infinite

2. True or False: In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 255.
   Answer: False

3. True or False: The precedence rules you learned in algebra apply during the evaluation of arithmetic expressions in Python.
   Answer: True

4. Exponentiation and assignment operations are right _____, so consecutive instances of these are evaluated from right to left.
   Answer: associative

Using Functions and Modules

1. Explain that Python includes many useful functions, which are organized in libraries of code called *modules*.

Teaching Tip	Python's global module index is available at http://docs.python.org/modindex.html.

Teaching Tip	"A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere." *Reference:* http://docs.python.org/tut/node8.html

Calling Functions: Arguments and Return Values

1. Introduce the terms *function*, *argument/parameter* (*optional* and *required*), and *returning a value*.

2. Define the concept of a *default behavior* of a function, and explain how this may be changed by calling the function using optional arguments.

3. Use an example to show how to obtain more information about a function by using `help`.

The `math` Module

1. Show how to use functions in the `math` module, both by importing the whole module and by importing individual resources.

The Main Module

1. Explain that, like any module, the *main module* can be imported. Show how this is equivalent to importing a Python script as a module.

2. Use the example provided in the book to show how to import the main module created in the case study of this chapter.

Program Format and Structure

1. Provide some guidance on how a typical Python program should look. Use the bullet points on Pages 67-68 as a guide.

Running a Script from a Terminal Command Prompt

1. Use Figures 2.6 and 2.7 to show how to run a script from a terminal command prompt.

2. Note that Python installations enable you to launch Python scripts by double-clicking the files from the OS's file browser. Explain what the fly-by-window problem is, and how to solve it: add an input statement at the end of the script that pauses until the user presses the ENTER or RETURN key.

Quick Quiz 3

1. What is a module?
   Answer: Python includes many useful functions, which are organized in libraries of code called modules.

2. A(n) _____ is a chunk of code that can be called by name to perform a task.
   Answer: function

3. True or False: Arguments are also known as literals.
   Answer: False

4. The statement _____ would import all of the `math` module's resources.
   Answer: `from math import *`

Class Discussion Topics

1. Are your students familiar with other software development models? If so, ask them to discuss their similarities and differences with the waterfall model. Which one do they like best?

2. Have your students used functions in other languages?

Additional Projects

1. Ask students to do some research and find out what other escape sequences are available in Python. They should compile a list of each of these with a short description of their functions.

2. Python provides other useful modules besides the `math` module. Ask students to do some research and find one or two other modules that they think will be useful in future projects. They should provide a brief description of each module, and include a list of two to five functions that belong to each module.

Additional Resources

1. String Literals:
   http://docs.python.org/ref/strings.html

2. Python Docstrings:
   http://epydoc.sourceforge.net/docstrings.html

3. Unicode HOWTO:
   http://www.amk.ca/python/howto/unicode

4. Global Module Index:
   http://docs.python.org/modindex.html

5. Module tutorial:
   http://docs.python.org/tutorial/modules.html

Key Terms

- **abstraction:** A simplified view of a task or data structure that ignores complex detail.
- **analysis:** The phase of the software life cycle in which the programmer describes what the program will do.
- **Argument(s):** A value or expression passed in a method call.
- **arithmetic expression:** A sequence of operands and operators that computes a value.

- **ASCII character set:** The American Standard Code for Information Interchange ordering for a character set.
- **assignment operator:** The symbol =, which is used to give a value to a variable.
- **assignment statement:** A method of giving values to variables.
- **character set:** The list of characters available for data and program statements.
- **coding:** The process of writing executable statements that are part of a program to solve a problem. See also implementation.
- **comments:** Nonexecutable statements used to make a program more readable.
- **concatenation:** An operation in which the contents of one data structure are placed after the contents of another data structure.
- **correct program:** A program that produces an expected output for any legitimate input.
- **data type(s):** A set of values and operations on those values.
- **default behavior:** Behavior that is expected and provided under normal circumstances.
- **design:** The phase of the software life cycle in which the programmer describes how the program will accomplish its tasks.
- **design error:** An error such that a program runs, but unexpected results are produced. Also referred to as a logic error.
- **docstring:** A sequence of characters enclosed in triple quotation marks (“““) that Python uses to document program components such as modules, classes, methods, and functions.
- **empty string:** A string that contains no characters.
- **end-of-line comment:** Part of a single line of text in a program that is not executed, but serves as documentation for readers.
- **escape sequence:** A sequence of two characters in a string, the first of which is /. The sequence stands for another character, such as the tab or newline.
- **expression:** A description of a computation that produces a value.
- **float:** A Python data type used to represent numbers with a decimal point, for example, a real number or a floating-point number.
- **floating-point number:** A data type that represents real numbers in a computer program.
- **function(s):** A chunk of code that can be treated as a unit and called to perform a task.
- **Implementation:** The phase of the software life cycle in which the program is coded in a programming language.
- **integer:** A positive or negative whole number, or the number 0. The magnitude of an integer is limited by a computer’s memory.
- **integer arithmetic operations:** Operations allowed on data of type `int`. These include the operations of addition, subtraction, multiplication, division, and modulus to produce integer answers.
- **left associative:** The property of an operator such that repeated applications of it are evaluated from left to right (first to last).
- **literal:** An element of a language that evaluates to itself, such as 34 or “hi there.”.
- **logic error:** See design error
- **main module:** The software component that contains the point of entry or start-up code of a program.
- **mixed-mode arithmetic:** Expressions containing data of different types; the values of these expressions will be of either type, depending on the rules for evaluating them.
- **module(s):** An independent program component that can contain variables, functions, and classes.

- **newline character:** A special character (‘\n’) used to indicate the end of a line of characters in a string or a file stream.
- **optional arguments:** Arguments to a function or method that may be omitted.
- **Parameter(s):** See argument(s)
- **precedence rules:** Rules that govern the order in which operators are applied in expressions.
- **prototype:** A trimmed-down version of a class or software system that still functions and allows the programmer to study its essential features.
- **pseudocode:** A stylized half-English, half-code language written in English but suggesting program code.
- **required arguments:** Arguments that must be supplied by the programmer when a function or method is called.
- **returning a value:** The process whereby a function or method makes the value that it computes available to the rest of the program.
- **scientific notation:** The representation of a floating-point number that uses a decimal point and an exponent to express its value.
- **semantic error:** A type of error that occurs when the computer cannot carry out the instruction specified.
- **semantics:** The rules for interpreting the meaning of a program in a language.
- **software development life cycle (SDLC):** The process of development, maintenance, and demise of a software system. Phases include analysis, design, coding, testing/verification, maintenance, and obsolescence.
- **string(s) (string literals):** One or more characters, enclosed in double quotation marks, used as a constant in a program.
- **strongly-typed programming language:** A language in which the types of operands are checked prior to applying an operator to them, and which disallows such applications, either at run time or at compile time, when operands are not of the appropriate type.
- **symbolic constant:** A name that receives a value at program start-up and whose value cannot be changed.
- **test suite:** A set of test cases that exercise the capabilities of a software component.
- **type conversion function:** A function that takes one type of data as an argument and returns the same data represented in another type.
- **Unicode:** A character set that uses 16 bits to represent over 65,000 possible characters. These include the ASCII character set as well as symbols and ideograms in many international languages.
- **variable:** A memory location, referenced by an identifier, whose value can be changed during execution of a program.
- **variable reference:** The process whereby the computer looks up and returns the value of a variable.
- **waterfall model:** A series of steps in which a software system trickles down from analysis to design to implementation. See also software development life cycle.