Lecture Notes

Overview

Chapter 6 describes the role of functions in structuring code in a program. Students learn to employ top-down design to assign tasks to functions. Through several examples, students learn to use and define functions, including the use of optional parameters. Advanced topics like recursion, namespaces, and higher-order functions are covered, too.

Objectives

After completing this chapter, students will be able to:

- Explain why functions are useful in structuring code in a program
- Employ top-down design to assign tasks to functions
- Define a recursive function
- Explain the use of the namespace in a program and exploit it effectively
- Define a function with required and optional parameters
- Use higher-order functions for mapping, filtering, and reducing

Functions as Abstraction Mechanisms

Abstraction is a simplified view of a task or data structure that ignores complex detail. Note that effective designers must invent useful abstractions to control complexity. "In computer science, abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time."
Reference: http://en.wikipedia.org/wiki/Abstraction_(computer_science).

Functions Eliminate Redundancy

Use the example provided in the book to see how functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code. Note the importance of eliminating redundancy in order to easily test and debug code.

Functions Hide Complexity

Use the code of the sum function introduced in the previous sub-section as an example that functions serve as abstraction mechanisms by hiding complicated details, including length of code and number of interacting components.

Functions Support General Methods with Systematic Variations

Note that an algorithm is a *general method* for solving a class of problems.

Algorithms should be general enough to provide a solution to many *problem instances*, and a function should provide a general method with systematic variations.

A function's arguments provide the means for varying the problem instances that the function's algorithm solves.

Functions Support the Division of Labor

In a computer program, functions can enforce a division of labor. Each function should perform a single coherent task. Each of the tasks required by a system can be assigned to a function. Functions also allow for division of labor between multiple programmers working on a joint project. The function names must be set in advance, but after that one programmer can write a given function and all other programmers can use that function without knowing its exact implementation.

Problem Solving with Top-Down Design

Review the terms *top-down design*, *problem decomposition*, and *stepwise refinement*. "Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth." *Reference:* http://en.wikipedia.org/wiki/Top-down.

The Design of the Text-Analysis Program

Use Figure 6.1 to see how the text-analysis program of Chapter 4 could have been structured in terms of programmer-defined functions, specifying the arguments each function receives and the value it returns.

The Design of the Sentence-Generator Program

Use Figure 6.2 to see how the sentence-generator program of Chapter 5 used a top-down design that flows out of the top-down structure of the grammar.

The Design of the Doctor Program

Use Figure 6.3 to see how the doctor program of Chapter 5 used a *responsibility-driven design*.

Use the functions reply and changePerson of the doctor program to provide guidelines indicating when a problem should be decomposed into multiple functions and when it should be solved by a single function.

Design with Recursive Functions

Review the terms *recursive design* and *recursive function*.

Defining a Recursive Function

A recursive function is a function that calls itself. Explain that to avoid infinite recursion, recursive functions need to have a *base case* and a *recursive step*.

Use the Python code provided in the book to see how to convert displayRange into a recursive function, thereby clarifying how recursive functions are used and how loop functions can often be converted into recursive functions. Review the term *recursive call*. Most recursive functions expect at least one argument.

Tracing a Recursive Function

Use the example provided in the book to see how to trace a recursive function and to demonstrate why recursive functions fill up the stack faster than linear functions.

Using Recursive Definitions to Construct Recursive Functions

A *recursive definition* consists of equations that state what a value is for one or more base cases and one or more recursive cases.

Use the Fibonacci example provided in the book to see how to use a recursive definition to construct a recursive function.

Recursion in Sentence Structure

Use the example provided in the book to see that recursive solutions can often flow from the structure of a problem. In this case, a noun phrase can be modified by a prepositional phrase, which also contains another noun phrase.

Review the term *indirect recursion* and how to make sure that indirect recursion does not go on forever. While the depth of indirect recursion may vary, indirect recursion requires the same attention to base cases as direct recursion. For more information, visit: www.cs.sfu.ca/~tamaras/recursion/Direct_vs_Indirect.html.

Infinite Recursion

Infinite recursion arises when programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.

In the event of infinite recursion, the Python virtual machine will eventually run out of memory resources to manage the process.

The Costs and Benefits of Recursion

Use Figure 6.4 to describe the costs of using recursive functions. Introduce the terms *call stack* and *stack frame*.

Research why it is sometimes more convenient to develop recursive solutions despite the hidden costs. For more information on mastering recursive programming, read: http://www.ibm.com/developerworks/linux/library/l-recurs.html.

Managing a Program's Namespace

A program's *namespace* is the set of its variables and their values.

Module Variables, Parameters, and Temporary Variables

Use the sample code provided in the book to see the difference between *module variables*, *temporary variables*, *parameters*, and *method names*.

Scope

Review the term *scope* as it applies to programs, comparing it to the scope of a word in a sentence.

Review why that module variables, parameters, and temporary variables have different scope.

Note that a function can reference a module variable but cannot assign a new value to it under normal circumstances. Python has two built-in functions, locals and global, which provide dictionary-based access to local and global variables. For more information, visit: www.faqs.org/docs/diveintopython/dialect_locals.html.

Lifetime

With respect to the *lifetime* of variables in Python: when a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.

Review how the concept of lifetime explains the fact that a function can reference a module variable but cannot assign a new value to it.

Default (Keyword) Arguments

Use the examples provided in the book to see how to define and use *default* or *keyword* arguments.

The default arguments that follow a function call can be supplied in two ways: *by position* and *by keyword*. A note on default arguments: "The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls."
Reference: http://docs.python.org/tut/node6.html

Higher-Order Functions (Advanced Topic)

Review what a *higher-order function* is. Note that a higher-order function separates the task of transforming each data value from the logic of accumulating the results. Python supports multiple programming paradigms (primarily object oriented, imperative, and functional). For more information on how to use Python as a functional programming language, see the second reference listed below.
References:
http://en.wikipedia.org/wiki/Python_(programming_language)
http://gnosis.cx/publish/programming/charming_python_13.html

Functions as First-Class Data Objects

Functions are *first-class data objects* in Python. Use the examples provided in the book to see how this functionality can be very useful, particularly when trying to separate different sections of code and reduce redundancy.

Mapping

Review how Python supports *mapping* of functions.

The map function returns a map object, which must then be converted to a list or other desired object to be viewed and used.

Filtering

When *filtering*, a function called a *predicate* is applied to each value in a list. Note what happens when the predicate returns True and when it returns False.

Reducing

When *reducing*, we take a list of values and repeatedly apply a function to accumulate a single data value.

Using lambda to Create Anonymous Functions

Use the examples provided in the book to see how to use lambda to create *anonymous functions* in Python. Be sure to specify the syntax restrictions of lambda.

Creating Jump Tables

Use the example provided in the book to see how to create a *jump table* (i.e., a dictionary of functions keyed by command names). "In computer programming, a branch table (sometimes known as a jump table) is a term used to describe an efficient method of transferring program control (branching) to another part of a program (or a different program that may have been dynamically loaded) using a table of branch instructions. The branch table construction is commonly used when programming in assembly language but may also be generated by a compiler."
Reference: http://en.wikipedia.org/wiki/Branch_table

Additional Resources

1. Python Scopes and Name Spaces:
   www.network-theory.co.uk/docs/pytut/PythonScopesandNameSpaces.html

2. Default Argument Values:
   http://docs.python.org/tut/node6.html#SECTION006710000000000000000

3. Functional Programming in Python:
   http://gnosis.cx/publish/programming/charming_python_13.html

Key Terms

- **abstraction:** A simplified view of a task or data structure that ignores complex detail.
- **anonymous function:** A function without a name, constructed in Python using lambda.

- **base case:** The condition in a recursive algorithm that is tested to halt the recursive process.
- **call:** Any reference to a function or method by an executable statement. Also referred to as invoke.
- **call stack:** The trace of function or method calls that appears when Python raises an exception during program execution.
- **default argument:** Also called a keyword argument. A special type of parameter that is automatically given a value if the caller does not supply one.
- **filtering:** The successive application of a Boolean function to a list of arguments that returns a list of the arguments that make this function return True.
- **first-class data objects:** Data objects that can be passed as arguments to functions and returned as their values.
- **general method:** A method that solves a class of problems, not just one individual problem.
- **grammar:** The set of rules for constructing sentences in a language.
- **higher-order function:** A function that expects another function as an argument and/or returns another function as a value.
- **indirect recursion:** A recursive process that results when one function calls another, which results at some point in a second call to the first function.
- **infinite recursion:** In a running program, the state that occurs when a recursive function cannot reach a stopping state.
- **jump table:** A dictionary that associates command names with functions that are invoked when those functions are looked up in the table.
- **lambda:** The mechanism by which an anonymous function is created.
- **lifetime:** The time during which a data object, function call, or method call exists.
- **mapping:** The successive application of a function to a list of arguments that returns a list of results.
- **namespace(s):** The set of all of a program’s variables and their values.
- **path:** A sequence of edges that allows one vertex to be reached from another.
- **predicate:** A function that returns a Boolean value.
- **problem decomposition:** The process of breaking a problem into subproblems.
- **problem instance:** An individual problem that belongs to a class of problems.
- **recursion:** The process of a subprogram calling itself. A clearly defined stopping state must exist.
- **recursive call:** The call of a function that already has a call waiting in the current chain of function calls.
- **recursive definition:** A set of statements in which at least one statement is defined in terms of itself.
- **recursive design:** The process of decomposing a problem into subproblems of exactly the same form that can be solved by the same algorithm.
- **recursive function:** A function that calls itself.
- **recursive step:** A step in the recursive process that solves a similar problem of smaller size and eventually leads to a termination of the process.
- **reducing:** The application of a function to a list of its arguments to produce a single value.
- **responsibility-driven design:** The assignment of roles and responsibilities to different actors in a program.
- **root directory:** The top-level directory in a file system.
- **scope:** The area of program text in which the value of a variable is visible.
- **stack frame:** An area of computer memory reserved for local variables and parameters of method calls. Also known as activation record and run-time stack.

- **stepwise refinement:** The process of repeatedly subdividing tasks into subtasks until each subtask is easily accomplished. See also structured programming and top-down design.
- **structure chart:** A graphical method of indicating the relationship between modules when designing the solution to a problem.
- **temporary variable:** A variable that is introduced in the body of a function or method for the use of that subroutine only.
- **top-down design:** A method for coding by which the programmer starts with a top-level task and implements subtasks. Each subtask is then subdivided into smaller subtasks. This process is repeated until each remaining subtask is easily coded.