Chapter 6

Design with Functions

At a Glance

Instructor's Manual Table of Contents

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

Teaching Tips

Functions as Abstraction Mechanisms

1. Introduce the concept of *abstraction*, and note that effective designers must invent useful abstractions to control complexity.

Teaching Tip	"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

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

Functions Hide Complexity

1. Use the code of the `sum` function introduced in the previous sub-section to explain 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

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

2. Stress that algorithms should be general enough to provide a solution to many *problem instances*, and explain that a function should provide a general method with systematic variations.

3. Explain that a function's arguments provide the means for varying the problem instances that the function's algorithm solves.

Functions Support the Division of Labor

1. Explain that 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.

Teaching Tip	Point out that 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

1. Introduce the terms *top-down design*, *problem decomposition*, and *stepwise refinement*.

Teaching Tip	"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

1. Use Figure 6.1 to describe 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

1. Use Figure 6.2 to show 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

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

2. 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.

Quick Quiz 1

1. What is an abstraction?
   Answer: An abstraction hides detail and thus allows a person to view many things as just one thing. We use abstractions to refer to the most common tasks in everyday life.

2. True or False: Functions serve as abstraction mechanisms by hiding complicated details.
   Answer: True

3. True or False: The individual problems that make up a class of problems are known as problem variations.
   Answer: False

4. What is top-down design?
   Answer: One popular design strategy for programs of any significant size and complexity is called top-down design. This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as problem decomposition. As each subproblem is isolated, its solution is assigned to a function.

Design with Recursive Functions

1. Introduce the terms *recursive design* and *recursive function*.

Defining a Recursive Function

1. Explain that 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*.

2. Use the Python code provided in the book to show 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. Introduce the term *recursive call*.

3. Use the recursive version of the `sum` function to explain that most recursive functions expect at least one argument.

Tracing a Recursive Function

1. Use the example provided in the book to show 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

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

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

Recursion in Sentence Structure

1. Use the example provided in the book to explain 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.

2. Introduce the term *indirect recursion*. Explain how to make sure that indirect recursion does not go on forever.

Teaching Tip	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

1. Explain that *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.

2. Use an example to show that 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

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

2. Provide some insight on why it is sometimes more convenient to develop recursive solutions despite the hidden costs.

Teaching Tip	For more information on mastering recursive programming, read: http://www.ibm.com/developerworks/linux/library/l-recurs.html.

Case Study: Gathering Information from a File System

1. Guide students as they step through this section.

Request

1. Ask students to write a program that allows the user to obtain information about the file system.

Analysis

1. Have your students taken an operating systems course before? If not, you will have to introduce several terms required to understand this case study, like *root directory*, *path*, and *parent* (of a node in a file system).

Design

1. Explain that the program can be structured according to two sets of tasks: those concerned with implementing a menu-driven command processor and those concerned with executing the commands. Outline the advantages of this structure.

Implementation (Coding)

1. Stress that the functions used in the program lead to a division of labor.

Quick Quiz 2

1. What is a recursive function?
   Answer: A recursive function is a function that calls itself.

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

3. True or False: Inefficient recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.
   Answer: False

4. What is a stack frame?
   Answer: At program startup, the PVM reserves an area of memory named a call stack. For each call of a function, the PVM must allocate a small chunk of memory called a stack frame. In this type of storage, the system places the values of the arguments and the return address for the particular function call. Space for the function call's return value is also reserved in its stack frame. When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code and the memory for the stack frame is deallocated.

Managing a Program's Namespace

1. Explain that a program's *namespace* is the set of its variables and their values.

Module Variables, Parameters, and Temporary Variables

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

Scope

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

2. Use one or more examples to explain that module variables, parameters, and temporary variables have different scope.

3. Stress that a function can reference a module variable but cannot assign a new value to it under normal circumstances.

Teaching Tip	Python has two built-in functions, `locals` and `globals`, which provide dictionary-based access to local and global variables. For more information, visit: www.faqs.org/docs/diveintopython/dialect_locals.html.

Lifetime

1. Explain the following 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.

2. Demonstrate 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

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

2. Explain that the default arguments that follow a function call can be supplied in two ways: *by position* and *by keyword*.

Teaching Tip	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)

1. Explain what a *higher-order function* is. Stress that a higher-order function separates the task of transforming each data value from the logic of accumulating the results.

Teaching Tip	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

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

Mapping

1. Use one or more examples to show how Python supports *mapping* of functions.

2. Point out that 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

1. Use one or more examples to explain that when *filtering*, a function called a *predicate* is applied to each value in a list. Explain what happens when the predicate returns True and when it returns False.

Reducing

1. Use one or more examples to explain that 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

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

Creating Jump Tables

1. Use the example provided in the book to show how to create a *jump table* (i.e., a dictionary of functions keyed by command names).

Teaching Tip	"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

Quick Quiz 3

1. What is a program's namespace?
   Answer: A program's namespace is the set of its variables and their values.

2. True or False: When module variables are introduced in a program, they are immediately given a value.
   Answer: True

3. True or False: In a program, the context that gives a name a meaning is called its lifetime.
   Answer: False

4. Explain what the following statement means: "In Python, functions can be treated as first-class data objects."

   Answer: This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries.

Class Discussion Topics

1. Have students used top-down design principles before? If so, ask them to discuss their experiences in class.

2. Are your students familiar with any functional programming language? If so, ask them to compare the use of functions as first class objects in that language with those in Python.

Additional Projects

1. Convert Euclid's algorithm to find the greatest common divisor (GCD) of two positive integers (Project 3.8) to a recursive function named `gcd`.

2. Ask your students to do some research on functional programming and lambda calculus. Can Python be used as a functional programming language? They should document their findings in a 1-2 page report.

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#SECTION00671000000000000000

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.