Functions as Abstraction Mechanisms

- An **abstraction** hides detail
  - Allows a person to view many things as just one thing
- We use abstractions to refer to the most common tasks in everyday life
  - For example, the expression “doing my laundry”
- Effective designers must invent useful abstractions to control complexity

Functions Eliminate Redundancy

- Functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code

```
def sum(lower, upper):
    """
    Arguments: A lower bound and an upper bound
    Returns: the sum of the numbers between the arguments
             and including them
    """
   result = 0
   while lower <= upper:
      result += lower
      lower += 1
   return result

>>> sum(1, 4)      # The summation of the numbers 1..4
10
>>> sum(50, 100)   # The summation of the numbers 50..100
3825
```


Functions Hide Complexity

- Functions serve as abstraction mechanisms is by hiding complicated details
- For example, consider the previous `sum` function
  - The idea of summing a range of numbers is simple; the code for computing a summation is not
- A function call expresses the idea of a process to the programmer
  - Without forcing him/her to wade through the complex code that realizes that idea

Functions Support General Methods with Systematic Variations

- An algorithm is a **general method** for solving a class of problems
- The individual problems that make up a class of problems are known as **problem instances**
  - What are the problem instances of our summation algorithm?
- Algorithms should be general enough to provide a solution to many problem instances
  - A function should provide a general method with systematic variations

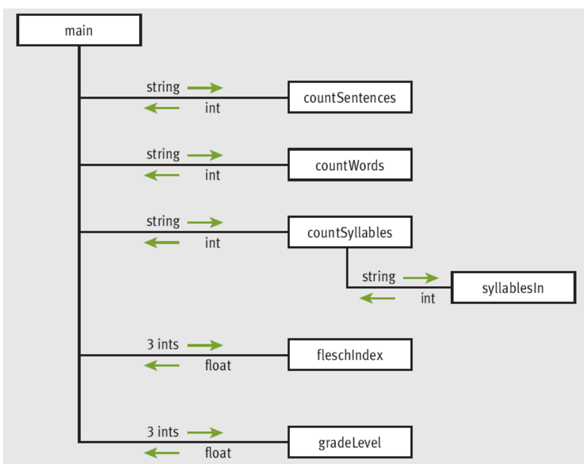
Functions Support the Division of Labor

- In a well-organized system, each part does its own job in collaborating to achieve a common goal
- In a computer program, functions can enforce a division of labor
  - Each function should perform a single coherent task
    - Example: Computing a summation
- Each of the tasks required by a system can be assigned to a function
  - Including the tasks of managing or coordinating the use of other functions

Problem Solving with Top-Down Design

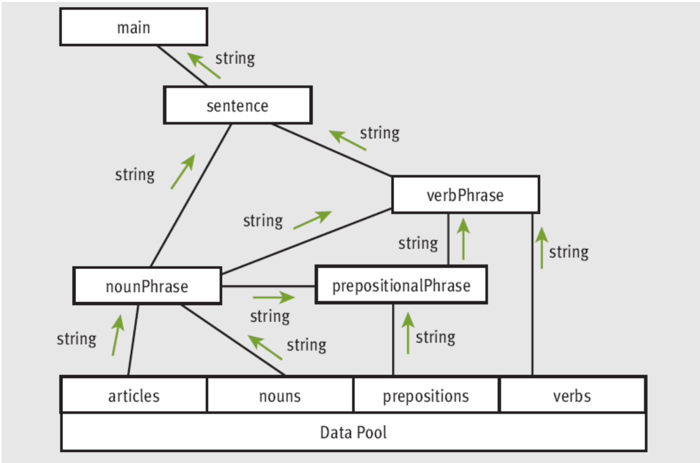
- **Top-down design** starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems
  - Process known as **problem decomposition**
- As each subproblem is isolated, its solution is assigned to a function
- As functions are developed to solve subproblems, solution to overall problem is gradually filled out
  - Process is also called **stepwise refinement**

The Design of the Text-Analysis Program



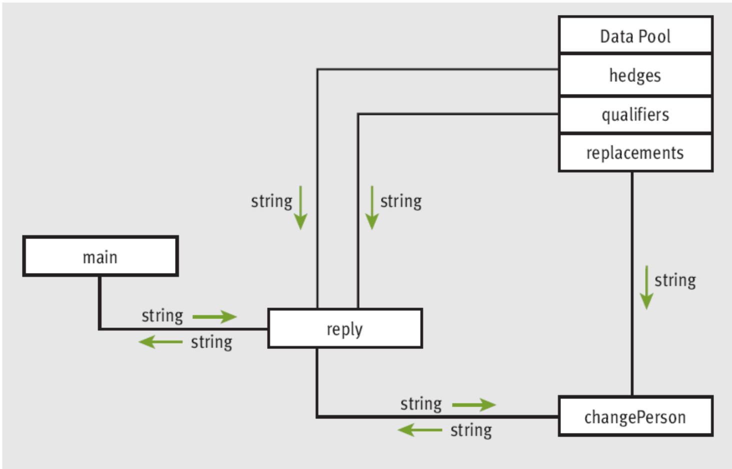
[FIGURE 6.1] A structure chart for the text-analysis program

The Design of the Sentence-Generator Program



[FIGURE 6.2] A structure chart for the sentence generator program

The Design of the Doctor Program



[FIGURE 6.3] A structure chart for the doctor program

Design with Recursive Functions

- In top-down design, you decompose a complex problem into a set of simpler problems and solve these with different functions
- In some cases, you can decompose a complex problem into smaller problems of the same form
  - Subproblems can be solved using the same function
    - This design strategy is called **recursive design**
    - Resulting functions are called **recursive functions**

Defining a Recursive Function

- A recursive function is a function that calls itself
  - To prevent function from repeating itself indefinitely, it must contain at least one selection statement
    - Statement examines **base case** to determine whether to stop or to continue with another **recursive step**
- To convert `displayRange` to a recursive function:

```
def displayRange(lower, upper):
    """Outputs the numbers from lower to upper."""
    while lower <= upper:
        print(lower)
        lower = lower + 1
```

  - You can replace loop with a selection statement and assignment statement with a **recursive call**

Defining a Recursive Function (continued)

- Making `displayRange` recursive (continued):

```
def displayRange(lower, upper):
    """Outputs the numbers from lower to upper."""
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

- Most recursive functions expect at least one argument
- Another example: Recursive version of `sum`

```
def sum(lower, upper):
    """Returns the sum of the numbers from lower to upper."""
    if lower > upper:
        return 0
    else:
        return lower + sum(lower + 1, upper)
```


Tracing a Recursive Function

```
def sum(lower, upper, margin):
    """Returns the sum of the numbers from lower to upper,
    and outputs a trace of the arguments and return values
    on each call."""
    blanks = " " * margin
    print(blanks, lower, upper)
    if lower > upper:
        print(blanks, 0)
        return 0
    else:
        result = lower + sum(lower + 1, upper, margin + 4)
        print(blanks, result)
        return result

>>> sum(1, 4, 0)
1 4
    2 4
        3 4
            4 4
                5 4
                0
            4
        7
    9
10
10
>>>
```

Using Recursive Definitions to Construct Recursive Functions

- Recursive functions are frequently used to design algorithms that have a **recursive definition**
  – A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases
- Example: Fibonacci sequence

1 1 2 3 5 8 13 . . .

```
Fib(n) = 1, when n = 1 or n = 2
Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2
```

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```


Recursion in Sentence Structure

- Recursive solutions can often flow from the structure of a problem
- Example: Structure of sentences in a language
  – A noun phrase can be modified by a prepositional phrase, which also contains another noun phrase

```
Noun phrase = Article Noun [Prepositional phrase]
```

```
def nounPhrase():
    """Returns a noun phrase, which is an article followed
    by a noun and an optional prepositional phrase."""
    phrase = random.choice(articles) + " " + random.choice(nouns)
    prob = random.randint(1, 4)
    if prob == 1:
        return phrase + " " + prepositionalPhrase()← Indirect recursion
    else:
        return phrase
```

Infinite Recursion

- **Infinite recursion** arises when programmer fails to specify base case or to reduce size of problem in a way that terminates the recursive process
  - In fact, the Python virtual machine eventually runs out of memory resources to manage the process

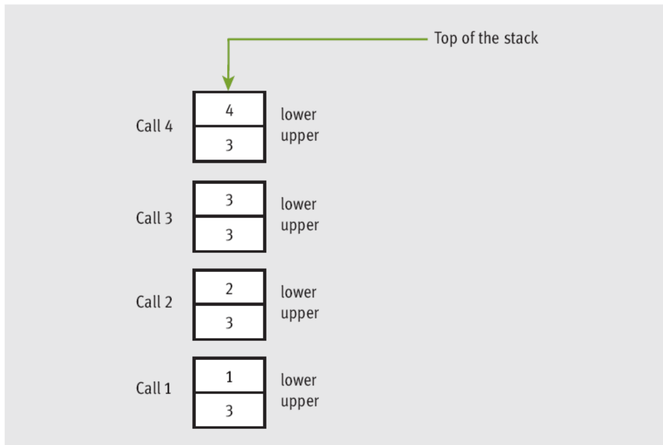
```
>>> def runForever(n):
       if n > 0:
          runForever(n)
       else:
          runForever(n - 1)

>>> runForever(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in runForever
RuntimeError: maximum recursion depth exceeded
```


The Costs and Benefits of Recursion

- PVM reserves an area of memory for the **call stack**
- For each call of a function, the PVM must allocate on the call stack a **stack frame**, which contains:
  - Values of the arguments
  - Return address for the particular function call
  - Space for the function call’s return value
- When a call returns, return address is used to locate the next instruction, and stack frame is deallocated
- Amount of memory needed for a loop does not grow with the size of the problem’s data set

The Costs and Benefits of Recursion (continued)

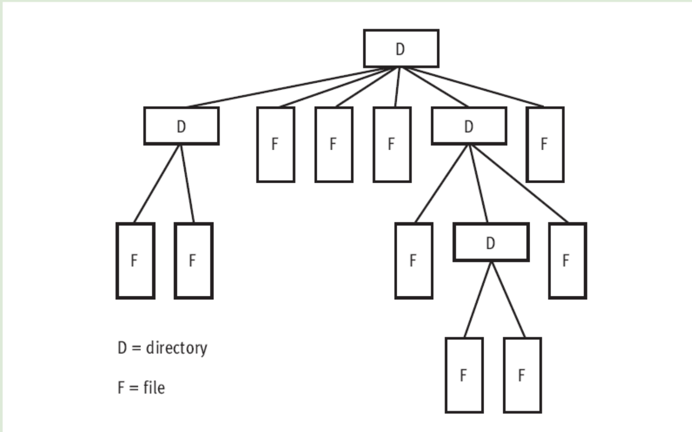


[FIGURE 6.4] The stack frames for `displayRange(1, 3)`

Case Study: Gathering Information from a File System

- Request: Write a program that allows the user to obtain information about the file system
- Analysis:
  - File systems are tree-like structures
  - At the top of the tree is the **root directory**
  - Under the root are files and subdirectories
  - Each directory in the system except the root lies within another directory called its **parent**
  - Example of a path (UNIX-based file system):
    - `/Users/KenLaptop/Book/Chapter6/Chapter6.doc`

Case Study: Gathering Information from a File System (continued)



[FIGURE 6.5] The structure of a file system

Case Study: Gathering Information from a File System (continued)

```
/Users/KenLaptop/Book/Chapter6
1   List the current directory
2   Move up
3   Move down
4   Number of files in the directory
5   Size of the directory in bytes
6   Search for a filename
7   Quit the program
Enter a number:
```

[FIGURE 6.6] The command menu of the **filesys** program

- When user enters a number, program runs command; then, displays CWD and menu again
- An unrecognized command produces an error message

Case Study: Gathering Information from a File System (continued)

COMMAND	WHAT IT DOES
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the CWD, print an error message; otherwise, move to this directory and make it the CWD.
Number of files in the directory	Prints the number of files in the CWD and all of its subdirectories.
Size of the directory in bytes	Prints the total number of bytes used by the files in the CWD and all of its subdirectories.
Search for a filename	Prompts the user for a search string. Prints a list of all the filenames (with their paths) that contain the search string, or "String not found."
Quit the program	Prints a signoff message and exits the program.

[TABLE 6.1] The commands in the **filesys** program

Case Study: Gathering Information from a File System (continued)

- Design:

```
function main()
    while True
        print(os.getcwd())
        print(MENU)
        Set command to acceptCommand()
        runCommand(command)
        if command == QUIT
            print("Have a nice day!")
            break
function countFiles(path)
    Set count to 0
    Set lyst to os.listdir(path)
    for element in lyst
        if os.path.isfile(element)
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("..")
    return count
```

Case Study: Gathering Information from a File System (continued)

```
"""
Program: filesys.py
Author: Ken

Provides a menu-driven tool for navigating a file system
and gathering information on files.
"""

import os, os.path

QUIT = '7'

COMMANDS = ('1', '2', '3', '4', '5', '6', '7')

MENU = """1   List the current directory
2   Move up
3   Move down
4   Number of files in the directory
5   Size of the directory in bytes
6   Search for a filename
7   Quit the program"""
…
```


Managing a Program's Namespace

- A program's **namespace** is the set of its variables and their values
  - You can control it with good design principles

Module Variables, Parameters, and Temporary Variables

`doctor.py` file (module name is `doctor`):

```
replacements = {"I":"you", "me":"you", "my":"your",
                "we":"you", "us":"you", "mine":"yours"}
 A module variable                   A parameter name
def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""A temporary variable
    words = sentence.split()
    replyWords = []                         A method name
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

- **Module variables** and **temporary variables** receive their values as soon as they are introduced
- **Parameters** behave like a variable and are introduced in a function or method header
  - Do not receive a value until the function is called

Scope

- **Scope:** Area in which a name refers to a given value
  - Temporary variables are restricted to the body of the functions in which they are introduced
  - Parameters are invisible outside function definition
  - The scope of module variables includes entire module below point where they are introduced
    - A function can reference a module variable, but can't under normal circumstances assign a new value to it

Lifetime

- Variable's **lifetime:** Period of time when variable has memory storage associated with it
  - When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM
- Module variables come into existence when introduced and generally exist for lifetime of program that introduces or imports them
- Parameters and temporary variables come into existence when bound to values during call, but go out of existence when call terminates

Default (Keyword) Arguments

- Arguments provide the function's caller with the means of transmitting information to the function
- Programmer can specify optional arguments with default values in any function definition:

```
def <function name>(<required args>,
                    <key-1> = <val-1>, … <key-n> = <val-n>)
```

  - Following the required arguments are one or more **default or keyword arguments**
  - When function is called with these arguments, default values are overridden by caller's values

Default (Keyword) Arguments (continued)

```
def repToInt(repString, base):
    """Converts the repString to an int in the base
    and returns this int."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base ** exponent
        exponent -= 1
    return decimal
```

```
def repToInt(repString, base = 2):

>>> repToInt("10", 10)
10
>>> repToInt("10", 8)   # Override the default to here
8
>>> repToInt("10", 2)   # Same as the default, not necessary
2
>>> repToInt("10")      # Base 2 by default
2
>>>
```


Default (Keyword) Arguments (continued)

- The default arguments that follow can be supplied in two ways:
  - By **position**
  - By **keyword**

```
def example(required, option1 = 2, option2 = 3):
    print(required, option1, option2)

>>> example(1)                  # Use all the defaults
1 2 3
>>> example(1, 10)              # Override the first default
1 10 3
>>> example(1, 10, 20)          # Override all the defaults
1 10 20
>>> example(1, option2 = 20)    # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10)  # Note the order
1 10 20
>>>
```

Higher-Order Functions (Advanced Topic)

- A **higher-order function** expects a function and a set of data values as arguments
  - Argument function is applied to each data value and a set of results or a single data value is returned
- A higher-order function separates task of transforming each data value from logic of accumulating the results

Functions as First-Class Data Objects

- Functions can be assigned to variables, passed as arguments, returned as the values of other functions, and stored in data structures

```
>>> abs                          # See what a function looks like
<built-in function abs>
>>> import math
>>> math.sqrt
<built-in function sqrt>
>>> f = abs                      # f is an alias for abs
>>> f                            # Evaluate f
<built-in function abs>
>>> f(-4)                        # Apply f to an argument
4
>>> funcs = [abs, math.sqrt]     # Put the functions in a list
>>> funcs
[<built-in function abs>, <built-in function sqrt>]
>>> funcs[1](2)                  # Apply math.sqrt to 2
1.4142135623730951
```

Functions as First-Class Data Objects (continued)

- Passing a function as an argument is no different from passing any other datum:

```
>>> def example(functionArg, dataArg):
        return functionArg(dataArg)

>>> example(abs, -4)
4
>>> example(math.sqrt, 2)
1.4142135623730951
>>>
```


Mapping

- **Mapping** applies a function to each value in a sequence and returns a new sequence of the results

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)           # Convert all strings to ints
<map object at 0x14cbd90>
>>> words                     # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words))   # Reset variable to change it
>>> words
[231, 20, -45, 99]
>>>
```

Mapping (continued)

```
def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

```
def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""

    def getWord(word):
        replacements.get(word, word)

    return " ".join(map(getWord, sentence.split()))
```


Filtering

- When **filtering**, a function called a **predicate** is applied to each value in a list
  - If predicate returns **`True`**, value is added to a new list; otherwise, value is dropped from consideration

```
>>> def odd(n): return n % 2 == 1

>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
>>>
```

Reducing

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

```
>>> from functools import reduce
>>> def add(x, y): return x + y

>>> def multiply(x, y): return x * y

>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
>>>
```


Using `lambda` to Create Anonymous Functions

- A **`lambda`** is an **anonymous function**
  - When the **`lambda`** is applied to its arguments, its expression is evaluated and its value is returned

```
lambda <argname-1, ..., argname-n>: <expression>
```

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data)    # Produce the sum
10
>>> reduce(lambda x, y: x * y, data)    # Produce the product
24
```

```
def sum(lower, upper):
    """Returns the sum of the numbers from lower to upper."""
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y,
                      range(lower, upper + 1))
```

Creating Jump Tables

- A **jump table** is a dictionary of functions keyed by command names

```
def runCommand(command):        # How simple can it get?
    jumpTable[command]()
```

```
# The functions named insert, replace, and remove
# are defined earlier

jumpTable = {}
jumpTable['1'] = insert
jumpTable['2'] = replace
jumpTable['3'] = remove
```


Summary

- A function serves as abstraction mechanism and eliminates redundant patterns of code
- Top-down design is strategy that decomposes complex problem into simpler subproblems and assigns their solutions to functions
- A structure chart is diagram of relationships among cooperating functions
- Recursive design is special case of top-down design, in which complex problem is decomposed into smaller problems of the same form

Summary (continued)

- A recursive function is a function that calls itself
  - Parts: Base case and recursive step
  - Can be computationally expensive
- Programmers must avoid infinite recursion
- Program namespace structured in terms of module variables, parameters, and temporary variables
- Scope can be used to control the visibility of names in a namespace
- The lifetime of a variable is duration of program execution during which it uses memory storage

Summary (continued)

- Functions are first-class data objects
- Higher-order functions can expect other functions as arguments and/or return functions as values
- A mapping function expects a function and a list of values as arguments
- A predicate is a Boolean function
- A filtering function expects a predicate and a list of values as arguments
- A reducing function expects a function and a list of values as arguments


EAST CENTRAL UNIVERSITY
1909