ch-4: Functions

September 2024

Introduction

- Best way to develop and maintain program is to construct it from smaller, more manageable pieces. This is called **divide and conquer**.
- Using existing functions as building blocks for creating new programs is a key aspect of **software reusability**—it's also a major benefit of object-oriented programming.
- Packaging code as a function allows you to execute it from various locations in your program just by calling the function, rather than duplicating the possibly lengthy code.
- This also makes programs easier to modify. When you change a function's code, all calls to the function execute the updated version.

Defining Functions

- You've called many built-in functions (`int`, `float`, `print`, `input`, `type`, `sum`, `len`, `min` and `max`) and a few functions from the statistics module (`mean`, `median` and `mode`).
- Each performed a single, well-defined task.
- You'll often define and call custom functions.

Defining a Custom Function

- A function definition begins with the **def** keyword, followed by the **function name**, a **set of parentheses** and a **colon (:)**.
- By convention function names should begin with a lowercase letter and in multiword names underscores should separate each word.
- The required parentheses contain the function's parameter list—a comma-separated list of parameters representing the data that the function needs to perform its task.

Defining Functions

- If the parentheses are empty, the function does not use parameters to perform its task.
- The indented lines after the colon (:) are the function's block.

Specifying a Custom Functions's Docstring

- The Style Guide for Python Code says that the first line in a function's block should be a docstring that briefly explains the function's purpose.

Returning a Result to a Function's Caller

- Functions are not run in a program until they are "called" or "invoked" in a program
- When a function finishes executing, it returns control to its caller—that is, the line of code that called the function.
- Function calls also can be embedded in expressions.

Defining Function

- There are two other ways to return control from a function to its caller:
  - Executing a return statement without an expression terminates the function and implicitly returns the value None to the caller.
    - The Python documentation states that None represents the absence of a value.
    - None evaluates to False in conditions.
  - When there's no return statement in a function, it implicitly returns the value None after executing the last statement in the function's block.
- What happens when you call a function

Defining Function

Function Characteristics

- Has a name
- Has (formal) parameters (0 or more)
- Has a docstring (optional but recommended)
- Has a body, a set of instructions to execute when function is called
- Returns something
  - Keyword `return`

HOW TO WRITE A FUNCTION

Function Name and Parameters:

```
def is_even(i):
```

Docstring:

```
""" Input:  i, a positive int.  Returns True if
i is even, otherwise False.  """
```

Function Body with Return Statements:

```
if i % 2 == 0:    return True
else:      return False
```

Modules, Packages, and Libraries

Module:

- A module is a single file (with a `.py` extension) containing Python code.
- It can define functions, classes, variables, and can also include runnable code.
- Modules allow logical organization of Python code into separate files.
- Example: If you have a file named `math_utils.py`, it is a module that can be imported and used in another script.

```
# math_utils.py (a module)
def add(a, b):
    return a + b
```

- You can import and use the module like this:

```
# main.py (importing and using the module)
import math_utils
print(math_utils.add(3, 5))
```

Modules, Packages, and Libraries

Package:

- A package is a collection of modules organized in a directory with a special `__init__.py` file.
- This file is required to make Python treat the directory as a package, allowing the grouping of related modules together.
- A package can contain:
  - Multiple modules (Python files like `module1.py`, `module2.py`).
  - Sub-packages (directories containing more modules and an `__init__.py` file).
- Example: Suppose you have the following directory structure:

```
math_tools/           # This is a package
    __init__.py       # Makes it a package
    arithmetic.py     # Module 1
    algebra.py        # Module 2
```

- You can import modules from this package as follows:

```
from math_tools import arithmetic
```

Modules, Packages, and Libraries

Library:

- A library is a collection of modules and packages.
- It is a broader term that generally refers to a bundle of reusable code.
- A library could be made up of multiple modules and packages that provide various functions for a specific purpose.

- Standard Library: Python comes with a built-in standard library that includes many modules and packages (like math, os, sys, etc.) for common tasks.
- Third-Party Libraries: These are libraries created by other developers that you can install (usually through pip). For example, NumPy, Pandas, and Requests are all third-party libraries, and they often contain multiple modules and packages.

Functions with Multiple Parameters

- Function maximum's Definition
- Python's Built-In max and min Functions
- For a list of Python's built-in functions and modules, see
  Click here to see the list

Random-Number Generation

- The random module, part of the Python Standard Library, provides a variety of tools for introducing randomness and chance in your programs.
- This module offers functions for generating random numbers, selecting random items from sequences, and working with randomness in different contexts, making it useful for simulations, games, data shuffling, and probabilistic algorithms.
- The randrange function generates an integer from the first argument value up to, but not including, the second argument value.
- Syntax: `random.randrange(start, stop, step)`
  - start(optional): The starting point of the range (inclusive). Defaults to 0 if not provided.
  - stop: The end of the range (exclusive). The generated number will be less than stop.
  - step(optional): The increment between numbers in the range. Defaults to 1.

Random-Number Generation

- Rolling a Six-Sided Die
- Tossing a coin
- Rolling a Six-Sided Die 6,000,000 Times
- Tossing a coin 2,000,000 Times
- Seeding the Random-Number Generator for Reproducibility: You can use the random module's seed function to seed the random-number generator yourself—this forces randrange to begin calculating its pseudorandom number sequence from the seed you specify.

Case Study: A Game of Chance

Simulate the popular dice game known as "craps." Here is the requirements statement:
You roll two six-sided dice, each with faces containing one, two, three, four, five and six spots, respectively. When the dice come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, you win. If the sum is 2, 3 or 12 on the first roll (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Math Module Functions

- The math module defines functions for performing various common mathematical calculations.
- Import statement: `import math`
- Some math module functions are summarized below—you can view the complete list at https://docs.python.org/3/library/math.html

Some math module functions

Function	Description	Example
`ceil(x)`	Rounds x to the smallest integer not less than x	`ceil(9.2)` is `10.0` `ceil(-9.8)` is `-9.0`
`floor(x)`	Rounds x to the largest integer not greater than x	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
`sin(x)`	Trigonometric sine of x (x in radians)	`sin(0.0)` is `0.0`
`cos(x)`	Trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
`tan(x)`	Trigonometric tangent of x (x in radians)	`tan(0.0)` is `0.0`
`exp(x)`	Exponential function e^x	`exp(1.0)` is `2.718282` `exp(2.0)` is `7.389056`
`log(x)`	Natural logarithm of x (base e)	`log(2.718282)` is `1.0` `log(7.389056)` is `2.0`
`log10(x)`	Logarithm of x (base 10)	`log10(10.0)` is `1.0` `log10(100.0)` is `2.0`
`pow(x, y)`	x raised to power y (x^y)	`pow(2.0, 7.0)` is `128.0` `pow(9.0, .5)` is `3.0`
`sqrt(x)`	square root of x	`sqrt(900.0)` is `30.0` `sqrt(9.0)` is `3.0`
`fabs(x)`	Absolute value of x—always returns a float. Python also has the built-in function `abs`, which returns an `int` or a `float`, based on its argument.	`fabs(5.1)` is `5.1` `fabs(-5.1)` is `5.1`
`fmod(x, y)`	Remainder of x/y as a floating-point number	`fmod(9.8, 4.0)` is `1.8`

Default Parameter Values

- When defining a function, you can specify that a parameter has a default parameter value.
- When calling a function, if you **omit** the argument for a parameter with a **default parameter value**, the default value for that parameter is **automatically passed**.
- Parameter with with default parameter value must appear to the right of parameters that do not have defaults.

Keyword Arguments

- When defining a function, you can use keyword arguments to pass arguments in *any order*.
- Each keyword *argument in a call* has the form *parametername*= *value*.
- In each function call, you must place keyword arguments after a function's positional arguments—that is, any arguments for which you do not specify the parameter name.
- Such arguments are assigned to the function's parameters left-to-right , based on the argument's positions in the argument list.
- Keyword arguments are also helpful for improving the readability of function calls, especially for functions with many arguments.

Arbitrary Argument Lists

- Function with arbitrary argument lists, such as `min` and `max` can receive *any* number of arguments.
- The function's documentation states that min has two required parameters (named `arg1` and `arg2`) and an optional third parameter of the form `*args`, indicating that the function can receive any number of additional arguments.
- The * before the parameter name tells Python to pack any remaining arguments into a tuple that's passed to the args parameter.

Arbitrary Argument Lists

Defining a Function with an Arbitrary Argument List

Consider the following function that can receive any number of arguments:

```
def average(*args):
    return sum(args)/len(args)
```

- The parameter name args is used by convention, but you may use any identifier.
- If the function has multiple parameters, the *args parameter must be the rightmost parameter.
- Note in our average definition that if the length of `args` is 0, a `ZeroDivisionError` occurs.
- The *operator, when applied to an iterable argument in a function call, unpacks its elements.

Arbitrary Argument Lists

Defining a Function with an Arbitrary Argument List

Consider the following function:

```
def product(*a):
    pr = 1
    for x in a:
        pr*=x
    return pr
```

- The `*a` allows the function `product` to accept any number of positional arguments.
- When the function is called, all the arguments passed to `product` are collected into a tuple `a`.
- Inside the function `product`, the variable `a` is itself a **tuple** that contains all the arguments passed.

Methods: Functions That Belong to Objects

- A method is simply a function that you call on an object using the form
  `object_name.method_name(arguments)`

```
In [1]: s = 'Hello'
In [2]: s.lower()  # call lower method on string object s
Out[2]: 'hello'
In [3]: s.upper() # call upper method on string object s
Out[3]: 'HELLO'
In [4]: s
Out[4]: 'Hello'
```

- In the "Object-Oriented Programming" chapter, you'll create custom types called classes and define custom methods that you can call on objects of those classes.

Scope Rules

- Each identifier has a scope that determines where you can use it in your program.
- For that portion of the program, the identifier is said to be "in scope."

Local Scope

- A local variable's identifier has local scope. It's "in scope" only from its definition to the end of the function's block.
- It "goes out of scope" when the function returns to its caller. So, a local variable can be used only inside the function that defines it.

Global Scope

- Identifiers defined outside any function (or class) have global scope—these may include functions, variables and classes.
- Variables with global scope are known as global variables.
- Identifiers with global scope can be used in a `.py` file or interactive session anywhere after they're defined.

Scope Rules

- In a function's block, the local variable shadows the global variable, making it inaccessible in the scope of the function's block.
- To modify a global variable in a function's block, you must use a global statement to declare that the variable is defined in the global scope

  ```
  global x
  ```

Blocks vs. Suites

- If the control statement is in the global scope, then any variables defined in the control statement have global scope.
- If the control statement is in a function's block, then any variables defined in the control statement have local scope.

Scope Rules

- If you define a variable named `sum`, it shadows the built-in function, making it inaccessible in your code.
- When you execute the following assignment, Python binds the identifier sum to the int object containing 15.
- At this point, the identifier `sum` no longer references the built-in function.

```
In [1]: sum = 10 + 5
In [2]: sum
Out[2]: 15
```

- So, when you try to use sum as a function, a `TypeError` occurs:

```
In [3]: sum([10, 5])
```

- Statements at Global Scope: Script statements at global scope execute as soon as they're encountered by the interpreter, whereas statements in a block execute only when the function is called.

import: A Deeper Look

- You've imported modules (such as `math` and `random`) with a statement like:
  `import` *`module_name`*
  then accessed their features via each module's name and a dot (.).
- Also, you've imported a specific identifier from a module (such as the decimal module's Decimal type) with a statement like:
  `from module_name import i`*`identifier`*
  then used that identifier without having to precede it with the module name and a dot (.).

import: A Deeper Look

Importing Multiple Identifiers from a Module

Caution: Avoid Wildcard Imports

- You can import all identifiers defined in a module with a wildcard import of the form

  ```
  from modulename import *
  ```

- This makes all of the module's identifiers available for use in your code.
- Importing a module's identifiers with a wildcard import can lead to subtle errors—it's considered a dangerous practice that you should avoid.

Binding Names for Modules and Module Identifiers

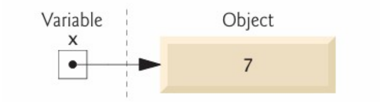
Passing Arguments to Functions: A Deeper Look

- In many programming languages, there are two ways to pass arguments-
  - pass by value (or call by value)
  - pass by reference (or call by reference)
- With pass-by-value, the called function receives a copy of the argument's value and works exclusively with that copy. Changes to the function's copy do not affect the original variable's value in the caller.
- With pass-by-reference, the called function can access the argument's value in the caller directly and modify the value if it's mutable.
- Python arguments are always passed by reference.
- Some people call this pass-by-object reference, because "everything in Python is an object.

Passing Arguments to Functions: A Deeper Look

Memory Addresses, References and "Pointers"

- You interact with an object via a reference, which behind the scenes is that object's address (or location) in the computer's memory—sometimes called a pointer in other languages.
- After an assignment like
  `x = 7`
  the variable x does not actually contain the value 7. Rather, it contains a reference to an object containing 7 (and some other data we'll discuss in later chapters) stored elsewhere in memory.
- You might say that x "points to" (that is, references) the object containing 7



Passing Arguments to Functions: A Deeper Look

Built-In Function id and Object Identities

- Consider
  ```
  In [1]:  x = 7
  ```
  Now `x` refers to (or points to) the integer object containing 7.
- No two separate objects can reside at the same address in memory, so every object in memory has a unique address.
- We can use the built-in `id` function to obtain a unique int value which identifies only that object while it remains in memory:
  ```
  In [2]: id(x)
  Out[2]: 4350477840
  ```
- The integer result of calling id is known as the object's identity.
- No two objects in memory can have the same identity.

Passing Arguments to Functions: A Deeper Look

- Passing an Object to a Function
- Testing Object Identities with the is Operator
  - Python's is operator, which returns True if its two operands have the same identity:
- Immutable Objects as Arguments
  - When a function receives as an argument a reference to an immutable (unmodifiable) object—such as an int, float, string or tuple—even though you have direct access to the original object in the caller, you cannot modify the original immutable object's value.
- Mutable Objects as Arguments
  - In the next chapter, we'll see that when a reference to a mutable object like a list is passed to a function, the function can modify the original object in the caller.

Function-Call Stack

- A function-call stack is a data structure used to manage the flow of function calls in Python.
- It operates as a Last-In, First-Out (LIFO) structure, similar to a stack of dishes.
- When a function is called, it is pushed onto the stack, and when it finishes executing, it is popped off the stack.
- The function at the **top** of the stack is always the one **currently executing**.
- This stack helps in managing function calls, local variables, and returning control to the calling function after execution is completed.
- It also plays a crucial role in recursive function calls by keeping track of each function's state.

Function-Call Stack

Stacks and Your Web Browser's Back Button
Stack Frames:

- The function-call stack supports the function call/return mechanism.
- Eventually, each function must return program control to the point at which it was called.
- For each function call, the interpreter *pushes* an entry called a **stack frame** (or an **activation record**) onto the stack.
- This entry contains the *return location* that the called function needs so it can return control to its caller.
- When the function finishes executing, the interpreter *pops* the function's stack frame, and control transfer to the *return location* that was popped.

Function-Call Stack

...

- The top stack frame always contains the information the currently executing function needs to return control to its caller.
- If before a function returns it makes a call to another function, the interpreter pushes a stack frame for that function call onto the stack.
- Thus, the return address required by the newly called function to return to its caller is now on top of the stack.

Function-Call Stack

Local Variables and Stack Frames

- Most functions have one or more parameters and possibly local variables that need to:
  - exist while the function is executing,
  - remain active if the function makes calls to other functions, and
  - "go away" when the function returns to its caller.
- A called function's stack frame is the perfect place to reserve memory for the function's local variables.
- That stack frame is pushed when the function is called and exists while the function is executing.
- When that function returns, it no longer needs its local variables, so its stack frame is *popped* from the stack, and its local variables no longer exist.

Stack Overflow: A stack overflow occurs when the function-call stack runs out of memory due to excessive function calls. This usually happens in cases of infinite recursion or a logic error where functions keep calling without returning.

Functional-Style Programming

- Python offers "functional-style" features that help you write code which is less likely to contain errors, more concise and easier to read, debug and modify.
- Functional style programs also can be easier to parallelize to get better performance on today's multicore processors.

Functional-style programming topics

avoiding side effects
closures
declarative programming (4)
decorators (10)
dictionary comprehensions (6)
`filter`/`map`/`reduce` (5)
`functools` module
generator expressions (5)
generator functions (12)
higher-order functions (5)
immutability (4)
internal iteration (4)
iterators (3)
`itertools` module (18)
`lambda` expressions (5)
lazy evaluation (5)
list comprehensions (5)
`operator` module (5, 13, 18)
pure functions (4)
`range` function (3, 4)
reductions (3, 5)
set comprehensions (6)

Functional-Style Programming

What vs. How

- Functional-style programming lets you simply say **what** you want to do. It hides many details of **how** to perform each task.
- Typically, **library code** handles the **how** to do.
- In `for` statement in many other programming languages, you must specify all the details of counter controlled iteration: a control variable, its initial value, how to increment it and a loop continuation condition that uses the control variable to to determine whether to continue iterating.
- This style of iteration is known as external iteration and is **error-prone**.
- External iteration **mutates** (i.e., modifies) the control variable, and the `for` statement's suit often mutates other variables as well.

Functional-Style Programming

- Functional-style programming emphasizes immutability. That is, it avoids operations that modify variables' values.
- You've already used list, string and built-in function **range** *iterators* with the `for` statement, and several *reductions* (functions `sum`, `len`, `min` and `max`).
- Specifying what, but not how, is an important aspect of internal iteration—a key functional-style programming concept.
- Stating what you want done rather than programming how to do it is known as declarative programming.
- In pure functional programming language you focus on writing pure functions. A pure function's result depends only on the argument(s) you pass to it.
- Also, given a particular argument (or arguments), a pure function always produces the same result.
- Also, a pure function does not have side effects.

Introduction to Data Science: Measures of Dispersion

- A measure of dispersion refers to the statistical tools used to describe the spread or variability of data points in a dataset.
- It quantifies how much the data deviates from the central value (like the mean or median).

Common measures of dispersion include:

- Range: The difference between the highest and lowest values.
- Variance: The average of the squared differences from the mean.
  `statistics.pvariance()`
- Standard Deviation: The square root of variance, representing the average distance of data points from the mean.
  `statistics.pstdev()`

Q.1 (Modified average Function) The average function we defined earlier can receive any number of arguments. If you call it with no arguments, however, the function causes a ZeroDivisionError. Reimplement average to receive one required argument and the arbitrary argument list argument *args, and update its calculation accordingly. Test your function. The function will always require at least one argument, so you'll no longer be able to get a ZeroDivisionError. When you call average with no arguments, Python should issue a TypeError indicating "average() missing 1 required positional argument."

Q.2 (Guess the Number) Write a script that plays “guess the number.” Choose the number to be guessed by selecting a random integer in the range 1 to 1000. Do not reveal this number to the user. Display the prompt "Guess my number between 1 and 1000 with the fewest guesses:". The player inputs a first guess. If the guess is incorrect, display "Too high. Try again." or "Too low. Try again." as appropriate to help the player “zero in” on the correct answer, then prompt the user for the next guess. When the user enters the correct answer, display "Congratulations. You guessed the number!", and allow the user to choose whether to play again.

Q.3 (Guess-the-Number Modification) Modify the previous exercise to count the number of guesses the player makes. If the number is 10 or fewer, display "Either you know the secret or you got lucky!" If the player makes more than 10 guesses, display "You should be able to do better!" Why should it take no more than 10 guesses? Well, with each "good guess," the player should be able to eliminate half of the numbers, then half of the remaining numbers, and so on. Doing this 10 times narrows down the possibilities to a single number. This kind of "halving" appears in many computer science applications. For example, in the "Computer Science Thinking: Recursion, Searching, Sorting and Big O" chapter, we'll present the high-speed binary search and merge sort algorithms, and you'll attempt the quicksort exercise—each of these cleverly uses halving to achieve high performance.

Q.4 (Date and Time) Python's datetime module contains a datetime type with a method today that returns the current date and time as a datetime object. Write a parameterless `date_and_time` function containing the following statement, then call that function to display the current date and time:

print(datetime.datetime.today())

On our system, the date and time display in the following format:

```
20180608 13:04:19.214180
```

Q.5 You have already simulated the popular dice game known as "craps." whose requirements statement is:

You roll two six-sided dice, each with faces containing one, two, three, four, five and six spots, respectively. When the dice come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, you win. If the sum is 2, 3 or 12 on the first roll (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Now, estimate the probability of wining this game.

Q.6 **(Simulation: The Tortoise and the Hare)** In this problem, you'll re-create the classic race of the tortoise and the hare. You'll use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

A clock ticks once per second. With each tick of the clock, your application should adjust the position of the animals according to the rules in the table below. Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (the "starting gate"). If an animal slips left before square 1, move it back to square 1.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Create two functions that generate the percentages in the table for the tortoise and the hare, respectively, by producing a random integer i in the range $1 \leq i \leq 10$. In the function for the tortoise, perform a “fast plod” when $1 \leq i \leq 5$, a “slip” when $6 \leq i \leq 7$ or a “slow plod” when $8 \leq i \leq 10$. Use a similar technique in the function for the hare.
Begin the race by displaying **BANG !!!!!**
AND THEY’RE OFF !!!!!
Then, for each tick of the clock (i.e., each iteration of a loop), display a 70-position line showing the letter **"T"** in the position of the tortoise and the letter **"H"** in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your application should display "OUCH!!!" at that position. All positions other than the "T", the "H" or the **"OUCH!!!"** (in case of a tie) should be blank.

After each line is displayed, test for whether either animal has reached or passed square 70. If so, display the winner and terminate the simulation.

If the tortoise wins, display **TORTOISE WINS!!! YAY!!!** If the hare wins, display Hare wins. **Yuch**. If both animals win on the same tick of the clock, you may want to favor the tortoise (the “underdog”), or you may want to display **”It’s a tie”**. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you’re ready to run your application, assemble a group of fans to watch the race. You’ll be amazed at how involved your audience gets!

Q.7 **(Arbitrary Argument List)** Calculate the product of a series of integers that are passed to the function product, which receives an arbitrary argument list. Test your function with several calls, each with a different number of arguments.

Q.8 **(Computer-Assisted Instruction)** Computer-assisted instruction (CAI) refers to the use of computers in education. Write a script to help an elementary school student learn multiplication. Create a function that randomly generates and returns a tuple of two positive one-digit integers. Use that function's result in your script to prompt the user with a question, such as

```
How much is 6 times 7?
```

For a correct answer, display the message "Very good!" and ask another multiplication question. For an incorrect answer, display the message "No. Please try again." and let the student try the same question repeatedly until the student finally gets it right.

Q.9 **(Computer-Assisted Instruction: Reducing Student Fatigue)** Varying the computer's responses can help hold the student's attention. Modify the previous exercise so that various comments are displayed for each answer. Possible responses to a correct answer should include 'Very good!', 'Nice work!' and 'Keep up the good work!' Possible responses to an incorrect answer should include 'No. Please try again.', 'Wrong. Try once more.' and 'No. Keep trying.' Choose a number from 1 to 3, then use that value to select one of the three appropriate responses to each correct or incorrect answer.

Q.10 **(Computer-Assisted Instruction: Difficulty Levels)** Modify the previous exercise to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems and at a difficulty level of 2, numbers as large as two digits.

Q.11 **(Computer-Assisted Instruction: Varying the Types of Problems)** Modify the previous exercise to allow the user to pick a type of arithmetic problem to study—1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only (avoid dividing by 0) and 5 means a random mixture of all these types.