The complete program for printing the description of an earthquake given the Richter scale magnitude is provided below.

ch03/earthquake.py

```
##
#  This program prints a description of an earthquake, given the Richter scale magnitude.
#

# Obtain the user input.
richter = float(input("Enter a magnitude on the Richter scale: "))

# Print the description.
if richter >= 8.0 :
   print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else :
   print("No destruction of buildings")
```

16. In a game program, the scores of players A and B are stored in variables `scoreA` and `scoreB`. Assuming that the player with the larger score wins, write an `if/elif` sequence that prints out `"A won"`, `"B won"`, or `"Game tied"`.
17. Write a conditional statement with three branches that sets `s` to 1 if `x` is positive, to –1 if `x` is negative, and to 0 if `x` is zero.
18. How could you achieve the task of Self Check 17 with only two branches?
19. Beginners sometimes write statements such as the following:

```
if price > 100 :
   discountedPrice = price - 20
elif price <= 100 :
   discountedPrice = price - 10
```

Explain how this code can be improved.

20. Suppose the user enters `-1` into the earthquake program. What is printed?
21. Suppose we want to have the earthquake program check whether the user entered a negative number. What branch would you add to the `if` statement, and where?

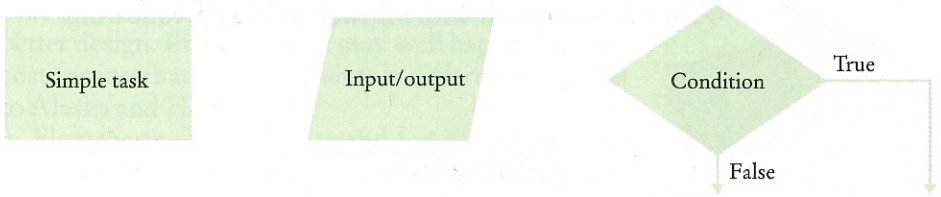
Practice It Now you can try these exercises at the end of the chapter: R3.22, P3.9, P3.34.

3.5 Problem Solving: Flowcharts

Flow charts are made up of elements for tasks, input/output, and decisions.

You have seen examples of flowcharts earlier in this chapter. A flowchart shows the structure of decisions and tasks that are required to solve a problem. When you have to solve a complex problem, it is a good idea to draw a flowchart to visualize the flow of control. The basic flowchart elements are shown in Figure 5.

Figure 5
Flowchart Elements



Each branch of a decision can contain tasks and further decisions.

The basic idea is simple enough. Link tasks and input/output boxes in the sequence in which they should be executed. Whenever you need to make a decision, draw a diamond with two outcomes (see Figure 6).

Each branch can contain a sequence of tasks and even additional decisions. If there are multiple choices for a value, lay them out as in Figure 7.

There is one issue that you need to be aware of when drawing flowcharts. Unconstrained branching and merging can lead to "spaghetti code", a messy network of possible pathways through a program.

Never point an arrow inside another branch.

There is a simple rule for avoiding spaghetti code: Never point an arrow *inside another branch*.

To understand the rule, consider this example: Shipping costs are $5 inside the United States, except that to Hawaii and Alaska they are $10. International shipping costs are also $10.

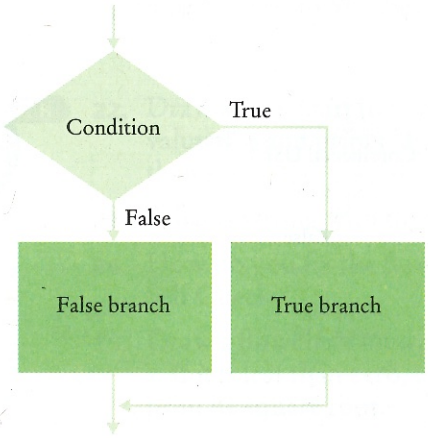


Figure 6 Flowchart with Two Outcomes

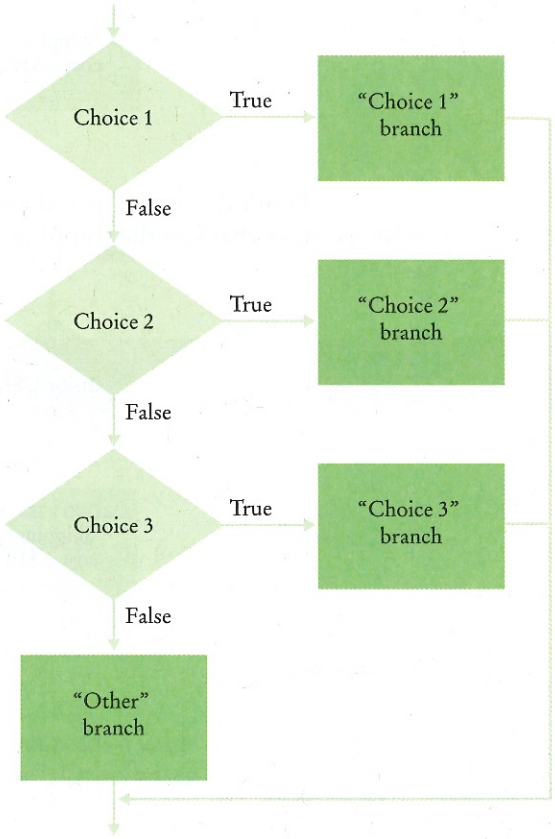
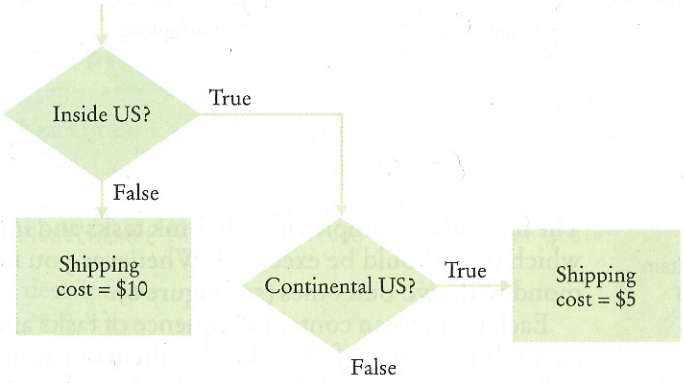
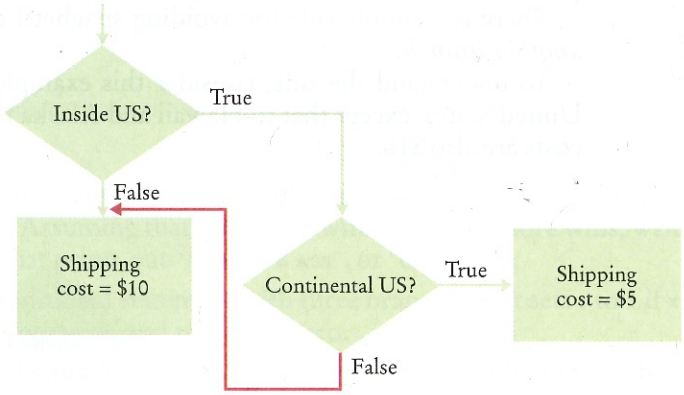


Figure 7 Flowchart with Multiple Choices

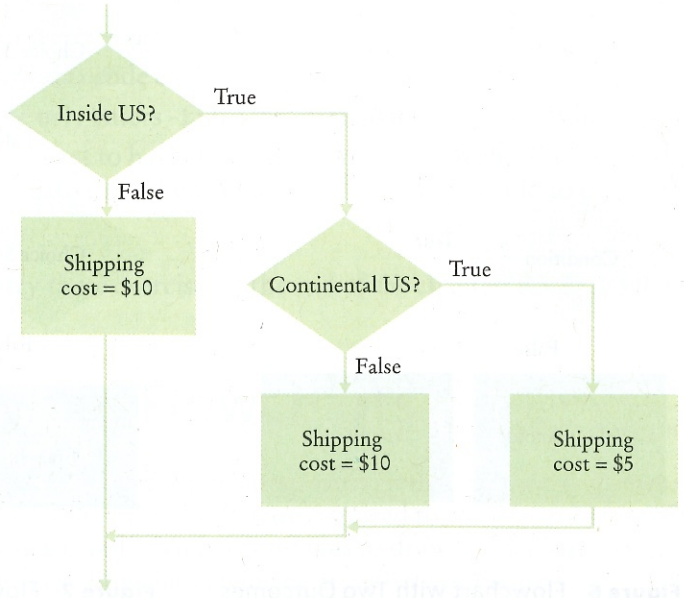
You might start out with a flowchart like the following:



Now you may be tempted to reuse the "shipping cost = $10" task:



Don't do that! The red arrow points inside a different branch. Instead, add another task that sets the shipping cost to $10, like this:



Not only do you avoid spaghetti code, but it is also a better design. In the future it may well happen that the cost for international shipments is different from that to Alaska and Hawaii.

Flowcharts can be very useful for getting an intuitive understanding of the flow of an algorithm. However, they get large rather quickly when you add more details. At that point, it makes sense to switch from flowcharts to pseudocode.

The complete program computing the shipping costs is provided below.

Spaghetti code has so many pathways that it becomes impossible to understand.

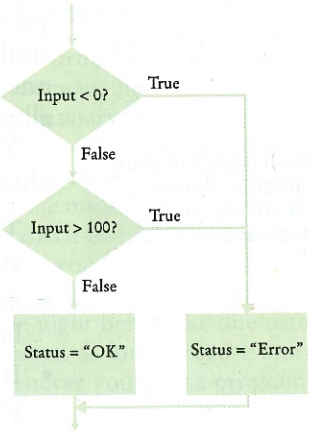
ch03/shipping.py

```
1  ##
2  #  A program to compute shipping costs.
3  #
4
5  # Obtain the user input.
6  country = input("Enter the country: ")
7  state = input("Enter the state or province: ")
8
9  # Compute the shipping cost.
10 shippingCost = 0.0
11
12 if country == "USA" :
13    if state == "AK" or state == "HI" :    # See Section 3.7 for the or operator
14       shippingCost = 10.0
15    else :
16       shippingCost = 5.0
17 else :
18    shippingCost = 10.0
19
20 # Print the results.
21 print("Shipping cost to %s, %s: $%.2f" % (state, country, shippingCost))
```

Program Run

```
Enter the country: USA
Enter the state or province: VA
Shipping cost to VA, USA: $5.00
```

22. Draw a flowchart for a program that reads a value `temp` and prints "Frozen" if it is less than zero.
23. What is wrong with the flowchart on the right?
24. How do you fix the flowchart of Self Check 23?
25. Draw a flowchart for a program that reads a value `x`. If it is less than zero, print "Error". Otherwise, print its square root.



26. Draw a flowchart for a program that reads a value `temp`. If it is less than zero, print "Ice". If it is greater than 100, print "Steam". Otherwise, print "Liquid".

Practice It Now you can try these exercises at the end of the chapter: R3.12, R3.13, R3.14.

Computing & Society 3.1 Denver's Luggage Handling System

Making decisions is an essential part of any computer program. Nowhere is this more obvious than in a computer system that helps sort luggage at an airport. After scanning the luggage identification codes, the system sorts the items and routes them to different conveyor belts. Human operators then place the items onto trucks. When the city of Denver built a huge airport to replace an outdated and congested facility, the luggage system contractor went a step further. The new system was designed to replace the human operators with robotic carts. Unfortunately, the system plainly did not work. It was plagued by mechanical problems, such as luggage falling onto the tracks and jamming carts. Equally frustrating were the software glitches. Carts would uselessly accumulate at some locations when they were needed elsewhere.

The airport had been scheduled to open in 1993, but without a functioning luggage system, the opening was delayed for over a year while the contractor tried to fix the problems. The contractor never succeeded, and ultimately a manual system was installed. The delay cost the city and airlines close to a billion dollars, and the contractor, once the leading luggage systems vendor in the United States, went bankrupt.

Clearly, it is very risky to build a large system based on a technology that has never been tried on a smaller scale. As robots and the software that controls them get better over time, they will take on a larger share of luggage handling in the future. But it is likely that this will happen in an incremental fashion.

The Denver airport originally had a fully automatic system for moving luggage, replacing human operators with robotic carts. Unfortunately, the system never worked and was dismantled before the airport was opened.

3.6 Problem Solving: Test Cases

Consider how to test the tax computation program from Section 3.3. Of course, you cannot try out all possible inputs of marital status and income level. Even if you could, there would be no point in trying them all. If the program correctly computes one or two tax amounts in a given bracket, then we have good reason to believe that all amounts will be correct.

Each branch of your program should be covered by a test case.

You want to aim for complete *coverage* of all decision points. Here is a plan for obtaining a comprehensive set of test cases:

- There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases.
- Test a handful of *boundary* conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking (which is discussed in Section 3.9), also test an invalid input, such as a negative income.