Decision Making

2

In this chapter we explore how to make choices in our programs. Decision making is valuable when something we want to do depends on some user input or some other value that is not known when we write our program. This is quite often the case and Python, along with all interesting programming languages, has the ability to compare values and then take one action or another depending on that outcome.

For instance, you might write a program that reads data from a file and takes one action or another based on the data it read. Or, a program might get some input from a user and then take one of several actions based on that input.

To make a choice in Python you write an *if* statement. An if statement takes one of two forms. It may be just an *if* statement. In this case, if the condition evaluates to true then it will evaluate the *then statements*. If the condition is not true the computer will skip to the statements after the if statement.

```
<statements before if statement>
if <condition>:
   <then statements>
<statements after if statement>
```

Figure 2.1 depicts this graphically. An if statement evaluates the conditional expression and then goes to one of two places depending on the outcome. Notice the indentation in the *if* statement above. The indentation indicates the *then statements* are part of the *if* statement. Indentation is very important in Python. Indentation determines the control flow of the program. Figure 2.1 graphically depicts this as well. If the condition evaluates to true, a detour is taken to execute the *then statements* before continuing on after the *if* statement.

Generally, we want to know if some value in our program is equal to, greater, or less than another value. The comparison operators, or relational operators, in Python allow us to compare two values. Any value in your program, usually a variable, can be compared with another value to see how the two values relate to each other.

Figure 2.2 lists the operators you can use to compare two values. Each of these operators is written between the two values or variables you want to compare. They evaluate to either true or false depending on the two values. When the condition

K.D. Lee, *Python Programming Fundamentals*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6642-9_2

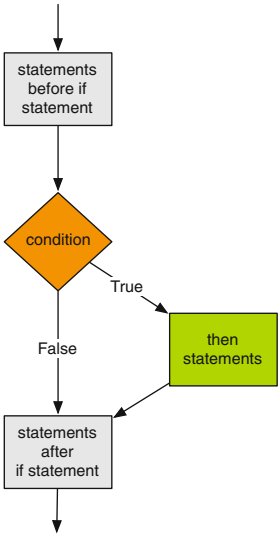

Fig. 2.1 If statement

Operator	Condition
<	Less Than
>	Greater Than
<=	Less Than or Equal to
>=	Greater Than or Equal to
==	Equal to
!=	Not Equal to

Fig. 2.2 Relational operators

evaluates to true, the *then statements* are executed. Otherwise, the *then statements* are skipped.

Example 2.1 An if statement is best described by giving an example. Assume we want to see if a number entered by a user is divisible by 7. We can write the program pictured in Fig. 2.3 to decide this. The program gets some input from the user. Remember that *input* reads a string from the user. The *int* converts the string to an integer. Then, the *num* variable is checked to see if it is divisible by 7. The `%` is called the modulo or just the mod operator. It gives us the remainder after dividing by the divisor (i.e. 7 in this case). If the remainder after dividing by 7 is 0 then the number entered by the user is divisible by 7.

An important feature of a debugger is the ability to step over our code and watch the computer execute each statement. This is called *stepping over* or *stepping into* our

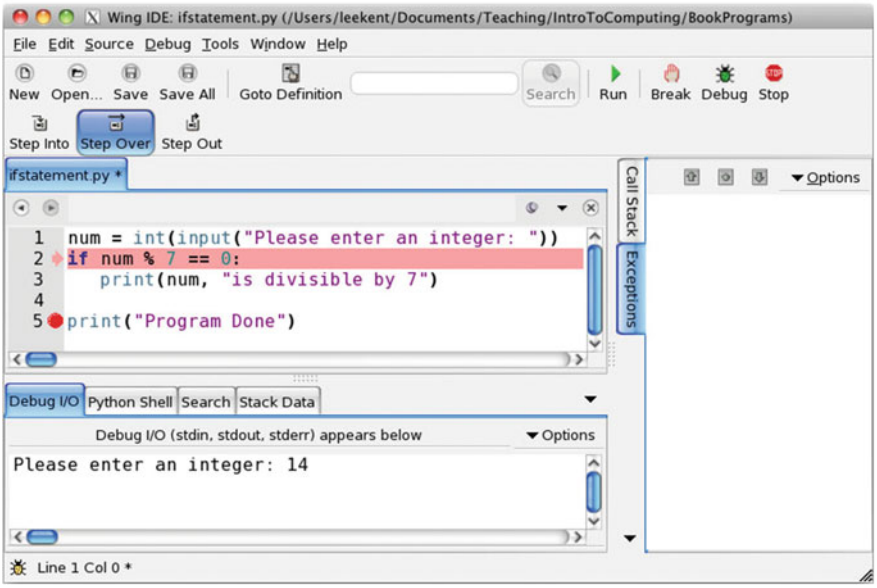

Fig. 2.3 Stepping into and over

code. Figure 2.3 depicts how this is done. For now stepping into and stepping over code do relatively the same thing. To begin stepping through a program you press the *Step Into* button. Once the program is started, you press the *Step Over* button to avoid jumping to other code that your program might call. Stepping into and over code can be very useful in understanding exactly what your program is doing.

Practice 2.1 Write a short program that asks the user to enter the name of a month. If the user enters "December" your program should print "Merry Christmas!". No matter what you enter, your program should print "Have a Happy New Year!" just before the program terminates. Then, use *Step Into* and *Step Over* to execute each statement that you wrote. Run your program at least twice to see how it behaves when you enter "December" and how it behaves when you enter something else.

Sometimes, you may want your program to do one thing if a condition is true and something else if a condition is false. Notice that the *if* statement does something only when the condition evaluates to true and does not do anything otherwise. If you want one thing to happen when a condition is true and another to happen if the condition is false then you need to use an *if-else* statement. An *if-else* statement adds a keyword of *else* to do something when the condition evaluates to false. An *if-else* statement looks like this.

```
<statements before if statement>
if <condition>:
   <then statements>
else:
   <else statements>
<statements after if statement>
```

If the condition evaluates to true, the *then statements* are executed. Otherwise, the *else statements* are executed. Figure 2.4 depicts this graphically. The control of your program branches to one of two locations, the *then statements* or the *else statements* depending on the outcome of the *condition*.

Again, indentation is very important. The *else* keyword must line up with the *if* statement to be properly paired with the *if* statement by Python. If you don't line up the *if* and the *else* in exactly the same columns, Python will not know that the *if* and the *else* go together. In addition, the *else* is only paired with the closest *if* that is in the same column. Both the *then statements* and the *else statements* must be indented and must be indented the same amount. Python is very picky about indentation because indentation in Python determines the flow of control in the program.

In the case of the *if-else* statement, either the *then statements* or the *else statements* will be executed. This is in contrast to the *if* statement that is described in Fig. 2.1. When learning about *if* statements this seems to be where some folks get stuck. The statements that are conditionally executed are those statements that are indented under the *if* or the *else*.

In either case, after executing the *if* or the *if-else* statement control proceeds to the next statement after the *if* or *if-else*. The statement after the *if-else* statement is the next line of the program that is indented the same amount as the *if* and the *else*.

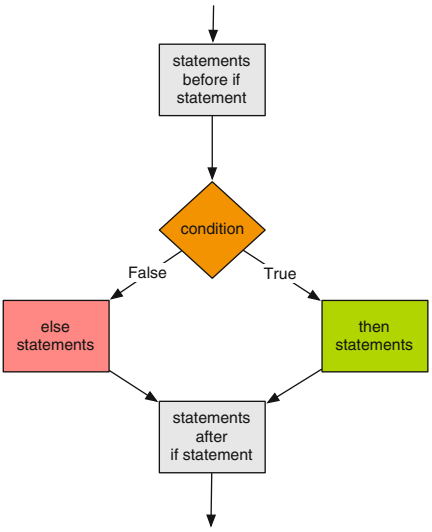

Fig. 2.4 If-else statement

Example 2.2 Consider a program that finds the maximum of two integers. The last line before the *if-else* statement is the *y* = assignment statement. The first line after the *if-else* statement is the *print*("*Done.*") statement.

```
x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))
if x > y:
    print(x,"is greater than",y)
else:
    print(y,"is greater than or equal to",x)
print("Done.")
```

Practice 2.2 Modify the program from practice Problem 2.1 to print "Merry Christmas!" if the month is December and "You'll have to wait" otherwise. It should still print "Have a Happy New Year!" in either case as the last line of output. Then run the program at least twice using step into and over to see how it behaves when "December" is entered and how the program behaves when anything else is entered.

2.1 Finding the Max of Three Integers

Any statement may be placed within an *if* statement, including other *if* statements. When you want to check multiple conditions there may be a need to put one *if* statement inside another. It can happen, but not very often. For instance, you may need to know if a value entered by a user is between two numbers. This could be written using two if statements, the outer *if* statement checking to see if the value entered is greater than some minimum, and the inner *if* statement checking to see of the value entered is less than some maximum. There are other ways to check to see if a value is between a maximum and minimum, but nested *if* statements can be used in this kind of circumstance.

Let's consider another possibility. Suppose you are asked to write a program that finds the maximum of three integers. This can be accomplished by writing nested *if* statements. Figure 2.5 depicts the flow of control for such a program.

We could determine which of the three integers, x, y and z, was the greatest by first comparing two of them, say x and y. Then, depending on the outcome of that condition, we would compare two more integers. By nesting if statements we can arrive at a decision about which is greatest. This code gets a bit complicated because we have three *if* statements to deal with, two of which are nested inside the third statement.

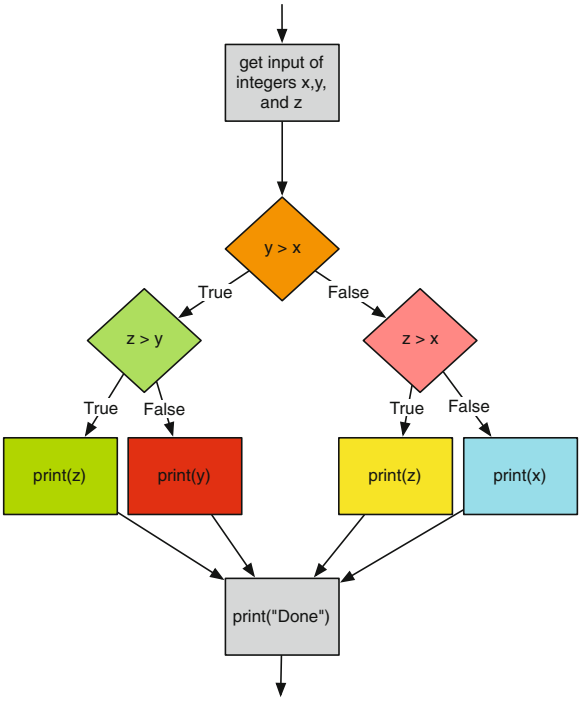

Fig. 2.5 Max of three integers

Example 2.3 While you wouldn't normally write code like this, it is provided here to show how *if* statements may be nested. The code prints the maximum of three integers entered by the user.

```
x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))
z = int(input("Please enter a third integer:"))
if y > x:
    if z > y:
        print(z, "is greatest.")
    else:
        print(y, "is greatest.")
else:
    if z > x:
        print(z, "is greatest.")
    else:
        print(x, "is greatest.")
print("Done.")
```

2.2 The Guess and Check Pattern

There is no way a good programmer would write a program that included the code that appeared in Example 2.3. It is too complicated. Instead, it would be much better to use a pattern or idiom called *Guess and Check*. Using this pattern involves first making a guess as to a correct solution and storing that guess in a variable. Then, you use one or more *if* statements to check that guess to see if it was correct or not. If it was not a correct guess, then the variable can be updated with a new guess. Finally, when the guess has been thoroughly checked, it should equal the value we were looking for.

Example 2.4 Consider the max of three program in Example 2.3. This could be rewritten using the guess and check pattern if we first make a guess as to the maximum value and then fix it if needed.

```
x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))
z = int(input("Please enter a third integer:"))
# Here is our initial guess
maxNum = x
if y > maxNum:       # Fix our guess if needed
    maxNum = y
if z > maxNum:       # Fix our guess again if needed
    maxNum = z
print(maxNum,"is greatest.")
print("Done.")
```

The code in Examples 2.3 and 2.4 get the same input and print exactly the same thing. However, the code in Example 2.4 is much easier to understand, mainly because the control flow is simplified by not having nested *if* statements. Notice that no *else* clauses were needed in Example 2.4. So, the code is simplified by having two *if* statements instead of three. It is simplified by having no nested *if* statements. Finally it is simplified because there are no use of *else* clauses in either of the *if* statements.

Practice 2.3 Use the guess and check pattern to determine if a triangle is a perfect triangle. A perfect triangle has side lengths that are multiples of 3, 4 and 5. Ask the user to enter the shortest, middle, and longest sides of a triangle and then print "It is a perfect triangle "if it is and "It is not a perfect triangle" if it isn't. You may assume that the side lengths are integers. Let your guess be that the message you will print is "It is a perfect triangle".

2.3 Choosing from a List of Alternatives

Sometimes you may write some code where you need to choose from a list of alternatives. For instance, consider a menu driven program. You may want to print a list of choices and have a user pick from that list. In such a situation you may want to use an *if* statement and then nest an if statement inside of the *else* clause. An example will help clarify the situation.

Example 2.5 Consider writing a program where we want the user to enter two floats and then choose one of several options.

```
x = float(input("Please enter a number:"))
y = float(input("Please enter a second number:"))

print("1) Add the two numbers")
print("2) Subtract the two numbers")
print("3) Multiply the two numbers")
print("4) Divide the two numbers")

choice = int(input("Please enter your choice:"))

print("The answer is:",end="")

if choice == 1:
    print(x + y)
else:
    if choice == 2:
        print(x - y)
    else:
        if choice == 3:
            print(x * y)
        else:
            if choice == 4:
                print(x / y)
            else:
                print("You did not enter a valid choice.")
```

Do you notice the stair step pattern that appears in the code in Example 2.5? This stair stepping is generally considered ugly and a nuisance by programmers. Depending on how much you indent each line, the code can quickly go off the right side of the screen or page. The need to select between several choices presents itself often enough that Python has a special form of the *if* statement to handle this. It is the *if-elif* statement. In this statement, one, and only one, alternative is chosen. The first alternative whose condition evaluates to *True* is the code that will be executed. All other alternatives are ignored. The general form of the *if-elif* statement is given here.

```
<statements before if statement>
if <first condition>:
   <first alternative>
elif <second condition>:
   <second alternative>
```

```
elif <third condition>:
   <third alternative>
else:
   <catch-all alternative>
<statements after the if statement>
```

There can be as many alternatives as are needed. In addition, the *else* clause is optional so may or may not appear in the statement. If we revise our example using this form of the *if* statement it looks a lot better. Not only does it look better, it is easier to read and it is still clear which choices are being considered. In either case, if the conditions are not mutually exclusive then priority is given to the first condition that evaluates to true. This means that while a condition may be true, its statements may not be executed if it is not the first true condition in the if statement.

Example 2.6 Here is a revision of Example 2.5 that looks a lot nicer.

```
1  x = float(input("Please enter a number:"))
2  y = float(input("Please enter a second number:"))
3
4  print("1) Add the two numbers")
5  print("2) Subtract the two numbers")
6  print("3) Multiply the two numbers")
7  print("4) Divide the two numbers")
8
9  choice = int(input("Please enter your choice:"))
10
11 print("The answer is:",end="")
12
13 if choice == 1:
14    print(x + y)
15 elif choice == 2:
16    print(x - y)
17 elif choice == 3:
18    print(x * y)
19 elif choice == 4:
20    print(x / y)
21 else:
22    print("You did not enter a valid choice.")
```

Practice 2.4 Write a short program that asks the user to enter a month and prints a message depending on the month entered according to the messages in Fig. 2.6. Then use the *step into and over* ability of the debugger to examine the code to see what happens.

Month	Message
January	Hello Snow!
February	More Snow!
March	No More Snow!
April	Almost Golf Time
May	Time to Golf
June	School's Out
July	Happy Fourth
August	Still Golfing
September	Welcome Back!
October	Fall Colors
November	Turkey Day
December	Merry Christmas!

Fig. 2.6 Messages

2.4 The Boolean Type

Conditions in *if* statements evaluate to *True* or *False*. One of the types of values in Python is called *bool* which is short for Boolean. George Boole was an English Mathematician who lived during the 1800s. He invented the Boolean Algebra and it is in honor of him that *true* and *false* are called Boolean values today [13].

In an *if* statement the condition evaluates to true or false. The Boolean value of the condition decides which branch is to be executed. The only requirement for a condition in an *if* statement is that it evaluates to true or false. So writing *if True* . . . would mean that the *then statements* would always be executed. Writing such an *if* statement doesn't really make sense, but using Boolean values in *if* statements sometimes does.

Example 2.7 Consider a program that must decide if a value is between 0 and 1. The program below uses a Boolean expression to discover if that is the case or not.

```
x = int(input("Please enter a number:"))
if x >= 0 and x <= 1:
    print(x, "is between 0 and 1")
```

Because an *if* statement only requires that the condition evaluates to true or false, any expression may be used as long as the result of evaluating it is true or false. Compound Boolean expressions can be built from simple expressions by using the logical operators *and*, *or*, and *not*. The *and* of two Boolean values is true when both Boolean values are true as shown in Fig. 2.7. The *or* of two Boolean values is true when one or the other is true, or when both are true as depicted in Fig. 2.8. The *not* of a Boolean value is true when the original value was false. This is shown in Fig. 2.9.

The three figures describe the *truth-tables* for each of the Boolean operators. A truth-table can be constructed for any compound Boolean expression. In each of the

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Fig. 2.7 The and operator

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Fig. 2.8 The or operator

A	not A
False	True
True	False

Fig. 2.9 The not operator

truth tables, *A* and *B* represent any Boolean expression. The tables show what the Boolean value of the expression *A and B*, *A or B*, and *not A* would be, given the values of *A* and *B* in the table. The *and*, *or*, and *not* logical operators can be strung together in all sorts of ways to produce complex Boolean expressions, but writing a program with complex Boolean expressions is generally a bad idea since it is difficult to understand the logic of complex expressions.Keeping track of whether to use *and* or *or* when *not* is involved in the expression is difficult and should be avoided if possible.

There are at least a couple of ways that negation (i.e. the use of the *not* operator) can be avoided in *if* statements. The statement can be rewritten to test the opposite of what you first considered. Another technique is to use the guess and check pattern. The following two examples illustrate how this can be done.

Example 2.8 Consider a club where you must be under 18 and over 15 to join. Here is a first try at a program that tells you whether you can join or not.

```
1  age = int(input("Please enter your age:"))
2  if (not age > 15) and (not age < 18):
3      print("You can't join")
4  else:
5      print("You can join")
```

Does this program do the job? In fact, as it is written here everyone can join the club. The problem is with the choice of *and* in the Boolean expression. It should have been *or*. The correct program would be written as follows.

```
age = int(input("Please enter your age:"))
if (not age > 15) or (not age < 18):
    print("You can't join")
else:
    print("You can join")
```

While the program above is correct, it is still difficult to understand why it is correct. The problem is the use of negation with the *or* operator. A much better way to write it would be to remove the negation in the expression.

```
age = int(input("Please enter your age:"))
if age > 15 and age < 18:
    print("You can join")
else:
    print("You can't join")
```

Example 2.9 The guess and check pattern can be applied to Boolean values as well. If you need to decide a *yes* or *no* question, you can make a guess and then fix it if needed.

```
age = int(input("Please enter your age:"))
member = True
if age <= 15:
    member = False
if age >= 18:
    member = False
if member:
    print("You can join")
else:
    print("You can't join")
```

The technique used in Example 2.9 is especially useful when there are a number of conditions that must be checked to make sure that the *yes* or *no* answer is correct. In fact, when the exact number of conditions is unknown, this technique may be necessary. How the exact number of conditions to check can be unknown will become clearer in the next chapter.

Practice 2.5 Write a program that determines whether you can run for president. To run for president the constitution states: *No Person except a natural born Citizen, or a Citizen of the United States, at the time of the Adoption of this Constitution, shall be eligible to the Office of President; neither shall any Person be eligible to that Office who shall not have attained to the Age of thirty five Years, and been fourteen Years a Resident within the United States [7].* Ask three questions of the user and use the guess and check pattern to determine if they are eligible to run for President.

2.5 Short Circuit Logic

Once in a while using the guess and check pattern may not produce the desired results. There are situations where you may want to evaluate one condition only if another condition is true or false. An example should make this clear.

Example 2.10 Consider a program that checks to see if one integer evenly divides another.

```
top = int(input("Please enter the numerator:"))
bottom = int(input("Please enter the denominator:"))

if bottom != 0 and top % bottom == 0:
    print("The numerator is evenly divided by the denominator.")
else:
    print("The fraction is not a whole number.")
```

Dividing *top* by *bottom* would result in a run-time error if *bottom* were 0. However, division by 0 will never happen in this code because Python, and most programming languages, uses short-circuit logic. This means that since both *A* and *B* must be true in the expression *A and B* for the expression to evaluate to *true*, if it turns out that *A* evaluates to *false* then there is no point in evaluating *B* and therefore it is skipped. In other words, Boolean expressions are evaluated from left to right until the truth or falsity of the expression can be determined and the condition evaluation terminates. This is exactly what we want in the code in Example 2.10.

Practice 2.6 In Minnesota you can fish if you are 15 years old or less and your parent has a license. If you are 16 years old or more you need to have your own license. Write a program that uses short circuit logic to tell someone if they are legal to fish in Minnesota. First ask them how old they are, whether they have a license or not, and whether their parent has a license or not.

2.6 Comparing Floats for Equality

In Python, real numbers or *floats* are represented using eight bytes. That means that 2^{64} different real numbers can be represented. This is a lot of real numbers, but not enough. Since there are infinitely many real numbers between any two real numbers, computers will never be able to represent all of them.

Because *floats* are only approximations of real numbers, there is some round-off error expected when dealing with real numbers in a program. Generally this round-off error is small and is not much of a problem unless you are comparing two real numbers

for equality. If you need to do this then you need to subtract the two numbers and see if the difference is insignificant since the two numbers may be slightly different.

So, to compare two *floats* for equality you can subtract the two and see if the difference is small relative to the two numbers.

Example 2.11 This program compares a guess with the result of dividing two *floats* and tells you if you are correct or not.

```
top = float(input("Please enter the numerator:"))
bottom = float(input("Please enter the denominator:"))

guess = float(input("Please enter your guess:"))

result = top/bottom
biggest = abs(result)

if abs(guess) > biggest:
   biggest = abs(guess)

# require the answer is within 1/10th Percent
# of the correct value.
if abs((guess-result)/biggest) < .001:
   print("You guessed right!")
else:
   print("Sorry, that's wrong. The correct value was",result)
```

Notice in the program in Example 2.11 that the *abs* function returns the absolute value of the *float* given to it so it doesn't matter if the numbers you are comparing are positive or negative. The code will work either way. In this example, 0.001 or 1/10th of 1 % difference was deemed close enough. Depending on your application, that value may be different.

Practice 2.7 Use the guess and check pattern to determine if a triangle is a perfect triangle. You must allow the user to enter any side length for the three sides of the triangle, not just integers. A perfect triangle has side lengths that are multiples of 3, 4 and 5. Ask the user to enter the three side lengths and then print "It is a perfect triangle" if it is and "It is not a perfect triangle" if it isn't.

2.7 Exception Handling

Sometimes things go wrong in a program and it is out of your control. For instance, if the user does not enter the proper input an error may occur in your program. Python includes exception handling so programmers can handle errors like this. Generally, if there is a possibility something could go wrong you should probably use some exception handling. To use exception handling you write a *try-except* statement.

```
<statements before try-except>
try:
   <try-block statements>
except [Exception]:
   <except-block statements>
<statements after the try-except code>
```

A try-except block may monitor for any exception or just a certain exception. There are many possible exceptions that might be caught. For instance, a *ValueError* exception occurs when you try to convert an invalid value to an integer. A *ZeroDivisionError* exception occurs when you try to divide by zero. In the general form shown above, the *Exception* is optional. That's what the square brackets (i.e. [ ]) mean. You don't actually write the square brackets. They mean the exception is optional in this case. If the exception is omitted then any exception is caught.

Exception handling can be used to check user input for validity. It can also be used internally in the program to catch calculations that might result in an error depending on the values involved in the calculation. When a *try* block is executed if a run-time error occurs that the try-except block is monitoring then program control immediately skips to the beginning of the except block. If no error occurs while executing the *try* block then control skips the *except* block and continues with the statement following the *try-except* statement. If an error occurs and the *except* block is executed, then when the except block finishes executing control goes to the next statement after the *try-except* statement (Fig. 2.10).

Example 2.12 Here is a bulletproof version of the program first presented in Example 2.10. This example does not use short-circuit logic. It uses exception handling instead. Notice the use of *exit(0)* below. This is a Python function that exits the program immediately, skipping anything that comes after it.

```
1  try:
2      top = int(input("Please enter the numerator:"))
3  except ValueError: # This try-except catches only ValueErrors
4      print("You didn't enter an integer.")
5      exit(0)
6
7  try:
8      bottom = int(input("Please enter the denominator:"))
9  except: # This try-except catches any exception
10     print("You didn't enter an integer.")
11     exit(0)
12
13 try:
14   if top % bottom == 0:
15       print("The numerator is evenly divided by the" + \
16         "denominator.")
17     else:
18       print("The fraction is not a whole number.")
19 except ZeroDivisionError:
20     print("The denominator cannot be 0.")
```

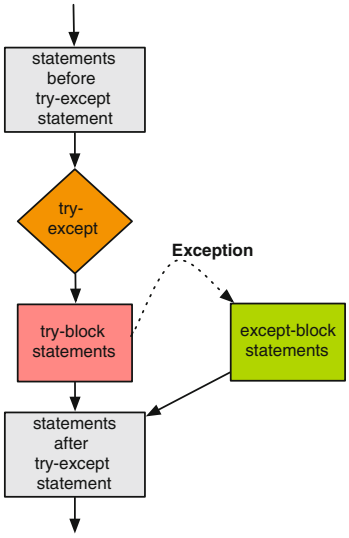

Fig. 2.10 Try-except statement

Try-except statements are useful when either reading input from the user or when using data that was read earlier in the program. Example 2.12 uses three *try-except* statements. The first two catch any non-integer input that might be provided. The last catches a division by zero error.

Practice 2.8 Add exception handling to the program in practice Problem 2.6 so that if the user answers something other than their age that the program prints "You did not enter your age correctly".

2.8 Review Questions

1. What is the difference between an *if* statement and an *if-else* statement? Be sure to state what the difference in meaning is between the two, not just the addition of the *else* keyword.
2. What type of value is returned by the relational operators?
3. What does it mean to *Step Over* code? What is that referring to?
4. What is a *nested if* statement?
5. How can *nested if* statements be avoided?
6. What is the general pattern for *Guess and Check*?

7. What is the Mathematician George Boole famous for?
8. When is it difficult to determine whether *and* or *or* should be used in an *if* statement?
9. What is short circuit logic? When does it apply? Give an example of when it would apply. Do not use the example in the book.
10. What is the problem with comparing floats for equality?
11. If an exception occurs on line 2 of while executing this code give the line numbers of this program in the order that they are executed. What is the output from the program?

```
1 try:
2     x = int(input("Please enter an integer:"))
3     y = int(input("Please enter another integer:"))
4 except:
5     print("You entered an invalid integer.")
6 print("The product of the two integers is",x*y)
```

2.9 Exercises

1. Type in the code of Example 2.6. Execute the code using a debugger like the one included with the Wing IDE 101. Step into and over the code using the debugger. Enter a menu choice of 1. Using the line numbers in Example 2.6, which lines of the program are executed when you enter a 1 for the menu choice. List these lines. Do the same for each of the other menu choice values. If you run the program and enter a menu choice of 5, which lines of the program are executed. If you use the debugger to answer this question you will be guaranteed to get it right and you'll learn a little about using a debugger.
2. Write a program that prints a user's grade given a percent of points achieved in the class. The program should prompt the user to enter his/her percent of points. It should then print a letter grade A, A−, B+, B, B−, C+, C, C−, D+, D, D−, F. The grading scale is given in Fig. 2.11. Use exception handling to check the input from the user to be sure it is valid. Running the program should look like this:

```
Please enter your percentage achieved in the class: 92.32
You earned an A- in the class.
```

3. Write a program that converts centimeters to yards, feet, and inches. There are 2.54 cm in an inch. You can solve this problem by doing division, multiplication, addition, and subtraction. Converting a float to an int at the appropriate time will help in solving this problem. When you run the program it should look exactly like this (except possibly for decimal places in the inches):

```
How many centimeters do you want to convert? 127.25
This is 1 yard, 1 foot, 2.098425 inches.
```

Grade	If Greater Than Or Equal To
A	93.33
A-	90
B+	86.67
B	83.33
B-	80
C+	76.67
C	73.33
C-	70
D+	66.67
D	63.33
D-	60
F	0

Fig. 2.11 Grading scale

This is a modification of the program in Exercise 5 of Chap. 1. In this version of it you should print "yard" when there is one yard, and "yards" when there is more than one yard. If there are zero yards then it should not print "yard" or "yards". The same thing applies to "feet". Use an *if* statement to determine the label to print and if the label should be printed at all.

4. Write a program that computes the minimum number of bills and coins needed to make change for a person. For instance, if you need to give $34.36 in change you would need one twenty, one ten, four ones, a quarter, a dime, and a penny. You don't have to compute change for bills greater than $20 dollar bills or for fifty cent pieces. You can solve this problem by doing division, multiplication, subtraction, and converting floats to ints when appropriate. So, when you run the program it should look exactly like this:

```
How much did the item cost: 65.64
How much did the person give you: 100.00
The person's change is $34.36
The bills or the change should be:
1 twenty
1 ten
4 ones
1 quarter
1 dime
1 penny
```

This is a modification of the program in Exercise 6 of Chap. 1. In this version, only non-zero amounts of bills and change should be printed. In addition, when only one bill or coin is needed for a particular denomination, you should use the singular version of the word. When more than one bill or coin for a denomination is needed, the plural of the label should be used.

5. Write a program that asks the user to enter an integer less than 50 and then prints whether or not that integer is prime. To determine if a number less than 50 is prime you only need to divide by all prime numbers that are less than or equal to

the square root of 50. If any of them evenly divide the number then it is not prime. Use the guess and check pattern to solve this problem. Use exception handling to check the input from the user to be sure it is valid. A run of the program should look like this:

```
Please enter an integer less than 50: 47
47 is prime.
```

6. Write a program that converts a decimal number to its binary equivalent. The decimal number should be read from the user and converted to an *int*. Then you should follow the algorithm presented in Example 1.1 to convert the decimal number to its binary equivalent. The binary equivalent must be a string to get the correct output. In this version of the program you must handle all 16-bit signed integers. That means that you must handle numbers from −32768 to 32767. In this version of the program you should not print any leading 0's. Leading 0's should be omitted from the output.
   If you want to check your work, you can use the *bin* function. The *bin* function will take a decimal number and return a string representation of that binary number. However, you should not use the *bin* function in your solution.
   The output from the program must be identical to this:

```
Please enter a number: 83
The binary equivalent of 83 is 1010011.
```

7. Write a program that prompts the user to enter a 16-bit binary number (a string of 1's and 0's). Then, the program should print the decimal equivalent. Be sure to handle both negative and positive binary numbers correctly. If the user enters less than 16 digits you should assume that the digits to the left of the last digit are zeroes. When run the output should look like this:

```
Please enter a 16-bit binary number: 1010011
The base 10 equivalent of the binary number 1010011 is 83.
```

   To handle negative numbers correctly you first need to detect if it is a negative number. A 16-digit binary number is negative if it is 16 digits long and the left-most digit is a 1. To convert a negative number to its integer equivalent, first take the 1's complement of the number. Then convert the 1's complement to an integer, then add 1 to the integer and negate the result to get the 2's complement.
   The conversion from bits to an integer can be carried out by multiplying each bit by the power of 2 that it represents as described in Sect. 1.5 of Chap. 1.
8. Converting numbers to any base can be accomplished using the algorithm from Example 1.1. For instance, an integer can be converted to hexadecimal using this algorithm. Hexadecimal numbers are base 16. That means there are 16 possible values for one digit. Counting in hexadecimal starts 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12 and so on. The algorithm changes so that instead of dividing by

2 you divide by 16. The one gotcha is that if the remainder after dividing is greater or equal to 10 (base 10) then you should not append the base 10 value to the string. Instead you should append a, b, c, d, e, or f. You can use if statements to determine the correct value to append. Write a program that prompts the user to enter an integer and then prints its hexadecimal equivalent. Traditionally, hexadecimal numbers start with a "0x" to identify them as hex, so your output should look like this:

```
Please enter an integer: 255
The hexadecimal equivalent is 0x00ff
```

Your program should handle any base 10 integer from 0 to 65535. There is a function called *hex* in Python that converts integers to their hexadecimal representation. You may not use this in implementing this program, but you may use it to see if your program is producing the correct output. For instance, calling *hex(255)* will return the string 0xff.
You should check the input that the user enters to make sure that it is in the valid range accepted by your program.

2.10 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

2.10.1 Solutions to Practice Problem 2.1

```
1 month = input("Please enter a month:")
2 if month =="December":
3     print("Merry Christmas!")
4 print("Have a Happy New Year!")
```

2.10.2 Solutions to Practice Problem 2.2

```
1 month = input("Please enter a month:")
2 if month =="December":
3     print("Merry Christmas!")
4 else:
5     print("You'll have to wait")
6 print("Have a Happy New Year!")
```

2.10.3 Solutions to Practice Problem 2.3

```
sideone = int(input( \
"Please enter length of shortest side of triangle:"))
sidetwo = int(input( \
"Please enter length of middle side of triangle:"))
sidethree = int(input( \
"Please enter length of longest side of triangle:"))

msg ="It is a perfect triangle."

if sideone % 3 != 0:
   msg = "It is not a perfect triangle."

if sidetwo % 4 != 0:
   msg = "It is not a perfect triangle."

if sidethree % 5 != 0:
   msg = "It is not a perfect triangle."

if sideone**2 + sidetwo**2 != sidethree**2:
   msg = "It is not a perfect triangle."
print(msg)
```

2.10.4 Solutions to Practice Problem 2.4

```
month = input("Please enter a month:")
if month =="January":
    msg ="Hello Snow!"
elif month =="February":
    msg ="More Snow!"
elif month =="March":
    msg ="No More Snow!"
elif month =="April":
    msg ="Almost Golf Time"
elif month =="May":
    msg ="Time to Golf"
elif month =="June":
    msg ="School's Out"
elif month =="July":
    msg ="Happy Fourth"
elif month =="August":
    msg ="Still Golfing"
elif month =="September":
    msg ="Welcome Back!"
elif month =="October":
    msg ="Fall Colors"
elif month =="November":
    msg ="Turkey Day"
elif month =="December":
    msg ="Merry Christmas!"
else:
    msg ="You entered an incorrect month."

print(msg)
```

2.10.5 Solutions to Practice Problem 2.5

```
age = int(input("Please enter your age:"))
resident = input( \
 "Are you a natural born citizen of the U.S. (yes/no)?")
years = int(input( \
 "How many years have you resided in the U.S.?"))

eligible = True
if age < 35:
    eligible = False

if resident !="yes":
    eligible = False

if years < 14:
    eligible = False

if eligible:
    print("You can run for president!")
else:
    print("You are not eligible to run for president!")
```

2.10.6 Solutions to Practice Problem 2.6

```
age = int(input("What is your age?"))
license = input( \
 "Do you have a fishing license in MN (yes/no)?")
parentlic = input( \
 "Does your parent have a fishing license (yes/no)?")

if (age < 16 and parentlic =="yes") or license =="yes":
    print("You are legal to fish in MN.")
else:
    print("You are not legal to fish in MN.")
```

2.10.7 Solutions to Practice Problem 2.7

```
sideone = float(input( \
 "Please enter length of shortest side of triangle:"))
sidetwo = float(input( \
 "Please enter length of middle side of triangle:"))
sidethree = float(input( \
 "Please enter length of longest side of triangle:"))

ratio = sideone / 3

msg ="It is a perfect triangle."

if abs((ratio - sidetwo / 4) / sidetwo) > 0.001:
    msg ="It is not a perfect triangle."

```

```
if abs((ratio - sidethree / 5) / sidethree) > 0.001:
    msg ="It is not a perfect triangle."

print(msg)
```

2.10.8 Solutions to Practice Problem 2.8

```
try:
    age = int(input("What is your age?"))
except:
    print("You did not enter your age correctly.")
    exit(0)

license = input( \
 "Do you have a fishing license in MN (yes/no)?")
parentlic = input( \
 "Does your parent have a fishing license (yes/no)?")

if (age < 16 and parentlic =="yes") or license =="yes":
    print("You are legal to fish in MN.")
else:
    print("You are not legal to fish in MN.")
```

http://www.springer.com/978-1-4471-6641-2

Python Programming Fundamentals

Lee, K.D.

2014, XII, 239 p. 64 illus., 53 illus. in color., Softcover
ISBN: 978-1-4471-6641-2