First, estimate realistically how much time it will take you to:

- Design the program logic.
- Develop test cases.
- Type the program in and fix syntax errors.
- Test and debug the program.

For example, for the income tax program I might estimate an hour for the design; 30 minutes for developing test cases; an hour for data entry and fixing syntax errors; and an hour for testing and debugging. That is a total of 3.5 hours. If I work two hours a day on this project, it will take me almost two days.

Make a schedule for your programming work and build in time for problems.

Then think of things that can go wrong. Your computer might break down. You might be stumped by a problem with the computer system. (That is a particularly important concern for beginners. It is *very* common to lose a day over a trivial problem just because it takes time to track down a person who knows the magic command to overcome it.) As a rule of thumb, *double* the time of your estimate. That is, you should start four days, not two days, before the due date. If nothing went wrong, great; you have the program done two days early. When the inevitable problem occurs, you have a cushion of time that protects you from embarrassment and failure.

3.7 Boolean Variables and Operators

The Boolean type `bool` has two values, `False` and `True`.

Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a *Boolean variable*. Boolean variables are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

In Python, the `bool` data type has exactly two values, denoted `False` and `True`. These values are not strings or integers; they are special values, just for Boolean variables. Here is the initialization of a variable set to `True`:

```
failed = True
```

You can use the value later in your program to make a decision:

```
if failed :   # Only executed if failed has been set to true
   . . .
```

A Boolean variable is also called a flag because it can be either up (true) or down (false).

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**. In Python, the `and` operator yields `True` only when both conditions are true. The `or` operator yields `True` if at least one of the conditions is true.

Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water. (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.) Water is liquid if the temperature is greater than zero and less than 100:

```
if temp > 0 and temp < 100 :
   print("Liquid")
```

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

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

A	not A
True	False
False	True

Figure 8 Boolean Truth Tables

The condition of the test has two parts, joined by the `and` operator. Each part is a Boolean value that can be true or false. The combined expression is true if both individual expressions are true. If either one of the expressions is false, then the result is also false (see Figure 8).

Python has two Boolean operators that combine conditions: `and` and `or`.

The Boolean operators `and` and `or` have a lower precedence than the relational operators. For that reason, you can write relational expressions on either side of the Boolean operators without using parentheses. For example, in the expression

```
temp > 0 and temp < 100
```

the expressions `temp > 0` and `temp < 100` are evaluated first. Then the `and` operator combines the results. (Appendix B shows a table of the Python operators and their precedences.)

Conversely, let's test whether water is not liquid at a given temperature. That is the case when the temperature is at most 0 or at least 100. Use the `or` operator to combine the expressions:

```
if temp <= 0 or temp >= 100 :
   print("Not liquid")
```

Figure 9 shows flowcharts for these examples.

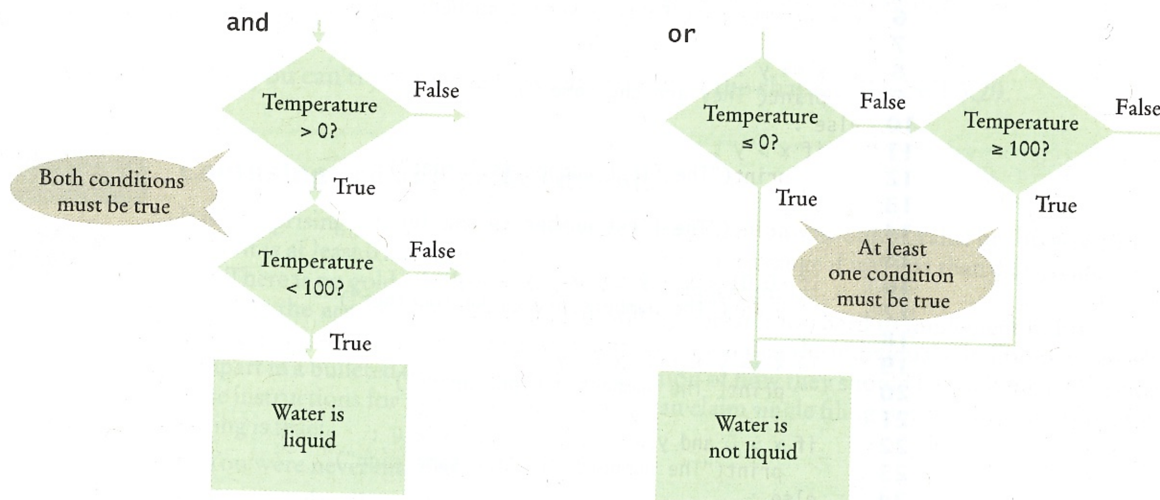

Figure 9 Flowcharts for `and` and `or` Combinations

At this geyser in Iceland, you can see ice, liquid water, and steam.

To invert a condition, use the `not` operator.

Sometimes you need to *invert* a condition with the `not` Boolean operator. The `not` operator takes a single condition and evaluates to `True` if that condition is false and to `False` if the condition is true. In this example, output occurs if the value of the Boolean variable `frozen` is `False`:

```
if not frozen :
   print("Not frozen")
```

Table 5 illustrates additional examples of evaluating Boolean operators. The following program demonstrates the use of Boolean expressions.

ch03/compare2.py

```
1  ##
2  #  This program demonstrates comparisons of numbers, using Boolean expressions.
3  #
4
5  x = float(input("Enter a number (such as 3.5 or 4.5): "))
6  y = float(input("Enter a second number: "))
7
8  if x == y :
9     print("They are the same.")
10 else :
11    if x > y :
12       print("The first number is larger")
13    else :
14       print("The first number is smaller")
15
16    if -0.01 < x - y and x - y < 0.01 :
17       print("The numbers are close together")
18
19    if x == y + 1 or x == y - 1 :
20       print("The numbers are one apart")
21
22    if x > 0 and y > 0 or x < 0 and y < 0 :
23       print("The numbers have the same sign")
24    else :
25       print("The numbers have different signs")
```

Program Run

```
Enter a number (such as 3.5 or 4.5): 3.25
Enter a second number: -1.02
The first number is larger
The numbers have different signs
```

Table 5 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 and 200 < 100`	`False`	Only the first condition is true.
`0 < 200 or 200 < 100`	`True`	The first condition is true.
`0 < 200 or 100 < 200`	`True`	The `or` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x and x < 100 or x == -1`	`(0 < x and x < 100) or x == -1`	The `and` operator has a higher precedence than the `or` operator (see Appendix B).
`not (0 < 200)`	`False`	`0 < 200` is true, therefore its negation is `false`.
`frozen == True`	`frozen`	There is no need to compare a Boolean variable with `True`.
`frozen == False`	`not frozen`	It is clearer to use `not` than to compare with `False`.

SELF CHECK

31. Suppose `x` and `y` are two integers. How do you test whether both of them are zero?

32. How do you test whether at least one of them is zero?

33. How do you test whether *exactly one of them* is zero?

34. What is the value of `not not frozen`?

35. What is the advantage of using the type `bool` rather than strings `"false"`/`"true"` or integers 0/1?

Practice It Now you can try these exercises at the end of the chapter: R3.29, P3.29.

Common Error 3.3

Confusing `and` and `or` Conditions

It is a surprisingly common error to confuse `and` and `or` conditions. A value lies between 0 and 100 if it is at least 0 *and* at most 100. It lies outside that range if it is less than 0 *or* greater than 100. There is no golden rule; you just have to think carefully.

Often the `and` or `or` is clearly stated, and then it isn't too hard to implement it. But sometimes the wording isn't as explicit. It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined. Consider these instructions for filing a tax return. You can claim single filing status if any one of the following is true:

- You were never married.
- You were legally separated or divorced on the last day of the tax year.
- You were widowed, and did not remarry.

Since the test passes if *any one* of the conditions is true, you must combine the conditions with or. Elsewhere, the same instructions state that you may use the more advantageous status of "married filing jointly" if all five of the following conditions are true:

- Your spouse died less than two years ago and you did not remarry.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of the tax year.
- You paid over half the cost of keeping up your home for this child.
- You filed a joint return with your spouse the year he or she died.

Because *all* of the conditions must be true for the test to pass, you must combine them with an and operator.

Programming Tip 3.4

Readability

Programs are more than just instructions to be executed by a computer. A program implements an algorithm and is commonly read by other people. Thus, it is important for your programs not only to be correct but also to be easily read by others. While many programmers focus only on a readable layout for their code, the choice of syntax can also have an impact on the readability.

To help provide readable code, you should never compare against a literal Boolean value (True or False) in a logical expression. For example, consider the expression in this if statement:

```
if frozen == False :
   print("Not frozen")
```

A reader of this code may be confused as to the condition that will cause the if statement to be executed. Instead, you should use the more acceptable form

```
if not frozen :
   print("Not frozen")
```

which is easier to read and explicitly states the condition.

It is also important to have appropriate names for variables that contain Boolean values. Choose names such as done or valid, so that it is clear what action should be taken when the variable is set to True or False.

Special Topic 3.3

Chaining Relational Operators

In mathematics, it is very common to combine multiple relational operators to compare a variable against multiple values. For example, consider the expression

```
0 <= value <= 100
```

Python also allows you to chain relational operators in this fashion. When the expression is evaluated, the Python interpreter automatically inserts the Boolean operator and to form two separate relational expressions

```
value >= 0 and value <= 100
```

Relational operators can be chained arbitrarily. For example, the expression a < x > b is perfectly legal. It means the same as a < x and x > b. In other words, x must exceed both a and b.

Most programming languages do not allow multiple relational operators to be combined in this fashion; they require explicit Boolean operators. Thus, when first learning to program, it is good practice to explicitly insert the Boolean operators. That way, if you must later change

to a different programming language, you will avoid syntax errors generated by chaining relational operators in a logical expression.

Special Topic 3.4

Short-Circuit Evaluation of Boolean Operators

The and and or operators are computed using **short-circuit evaluation**. In other words, logical expressions are evaluated from left to right, and evaluation stops as soon as the truth value is determined. When an and is evaluated and the first condition is false, the second condition is not evaluated, because it does not matter what the outcome of the second test is.

The and and or operators are computed using *short-circuit evaluation:* As soon as the truth value is determined, no further conditions are evaluated.

For example, consider the expression

```
quantity > 0 and price / quantity < 10
```

Suppose the value of quantity is zero. Then the test quantity > 0 fails, and the second test is not attempted. That is just as well, because it is illegal to divide by zero.

Similarly, when the first condition of an or expression is true, then the remainder is not evaluated because the result must be true.

In a short circuit, electricity travels along the path of least resistance. Similarly, short-circuit evaluation takes the fastest path for computing the result of a Boolean expression.

Special Topic 3.5

De Morgan's Law

Humans generally have a hard time comprehending logical conditions with *not* operators applied to *and/or* expressions. De Morgan's Law, named after the logician Augustus De Morgan (1806–1871), can be used to simplify these Boolean expressions.

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States.

```
if not (country == "USA" and state != "AK" and state != "HI") :
    shippingCharge = 20.00
```

This test is a little bit complicated, and you have to think carefully through the logic. When it is *not* true that the country is USA *and* the state is not Alaska *and* the state is not Hawaii, then charge $20.00. Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but it takes human programmers to write and maintain the code. Therefore, it is useful to know how to simplify such a condition.

De Morgan's Law has two forms: one for the negation of an and expression and one for the negation of an or expression:

`not (A and B)`	is the same as	`not A or not B`
`not (A or B)`	is the same as	`not A and not B`

De Morgan's law tells you how to negate and and or conditions.

Pay particular attention to the fact that the and and or operators are *reversed* by moving the not inward. For example, the negation of "the state is Alaska *or* it is Hawaii",

```
not (state == "AK" or state == "HI")
```

is "the state is not Alaska *and* it is not Hawaii":

```
state != "AK" and state != "HI"
```

Now apply the law to our shipping charge computation:

```
not (country == "USA" and state != "AK" and state != "HI")
```

is equivalent to

```
not (country == "USA") or not (state != "AK") or not (state != "HI")
```

Because two negatives cancel each other out, the result is the simpler test

```
country != "USA" or state == "AK" or state == "HI"
```

In other words, higher shipping charges apply when the destination is outside the United States or to Alaska or Hawaii.

To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.

3.8 Analyzing Strings

Use the `in` operator to test whether a string occurs in another.

Sometimes it is necessary to determine if a string contains a given substring. That is, one string contains an exact match of another string. Given this code segment,

```
name = "John Wayne"
```

the expression

```
"Way" in name
```

yields `True` because the substring `"Way"` occurs within the string stored in variable `name`. Python also provides the inverse of the `in` operator, `not in`:

```
if "-" not in name :
   print("The name does not contain a hyphen.")
```

Sometimes we need to determine not only if a string contains a given substring, but also if the string begins or ends with that substring. For example, suppose you are given the name of a file and need to ensure that it has the correct extension.

```
if filename.endswith(".html") :
   print("This is an HTML file.")
```

The `endswith` string method is applied to the string stored in `filename` and returns `True` if the string ends with the substring `".html"` and `False` otherwise. Table 6 describes additional string methods available for testing substrings.

Table 6 Operations for Testing Substrings

Operation	Description
substring `in` *s*	Returns `True` if the string *s* contains *substring* and `False` otherwise.
s`.count(`*substring*`)`	Returns the number of non-overlapping occurrences of *substring* in the string *s*.
s`.endswith(`*substring*`)`	Returns `True` if the string *s* ends with the `substring` and `False` otherwise.
s`.find(`*substring*`)`	Returns the lowest index in the string *s* where *substring* begins, or –1 if *substring* is not found.
s`.startswith(`*substring*`)`	Returns `True` if the string *s* begins with *substring* and `False` otherwise.