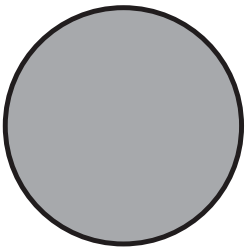
2

MAKING DECISIONS



Most programs that we use on a daily basis behave differently depending on what happens during their execution. For example, when a word-processor program asks us whether we want to save our work, it makes a decision based on our response: saving our work if we answer "yes" and not saving our work if we answer "no." In this chapter, we'll learn about `if` statements, which let our programs make decisions.

We'll solve two problems: determining the result of a basketball game and determining whether a phone number belongs to a telemarketer.

Problem #3: Winning Team

In this problem, we'll need to output a message that depends on the outcome of a basketball game. To do that, we'll learn all about `if` statements. We'll also learn how we can store and manipulate true and false values in our programs.

This is DMOJ problem `ccc19j1`.

The Challenge

In basketball, three plays score points: a three-point shot, a two-point shot, and a one-point free throw.

You just watched a basketball game between the Apples and Bananas, and recorded the number of successful three-point, two-point, and one-point plays for each team. Indicate whether the game was won by the Apples, the game was won by the Bananas, or the game was a tie.

Input

There are six lines of input. The first three give the scoring for the Apples, and the latter three give the scoring for the Bananas.

- The first line gives the number of successful three-point shots for the Apples.
- The second line gives the number of successful two-point shots for the Apples.
- The third line gives the number of successful one-point free throws for the Apples.
- The fourth line gives the number of successful three-point shots for the Bananas.
- The fifth line gives the number of successful two-point shots for the Bananas.
- The sixth line gives the number of successful one-point free throws for the Bananas.

Each number is an integer from 0 to 100.

Output

The output is a single character.

- If the Apples scored more points than the Bananas, output `A` (*A* for Apples).
- If the Bananas scored more points than the Apples, output `B` (*B* for Bananas).
- If the Apples and Bananas scored the same number of points, output `T` (*T* for Tie).

Conditional Execution

We can make a lot of headway here by using what we learned in Chapter 1. We can use `input` and `int` to read each of the six integers from the input. We can use variables to hang on to those values. We can multiply the number of successful three-point shots by 3 and the number of successful two-point shots by 2. We can use `print` to output an `A`, `B`, or `T`.

What we haven't learned yet is how our programs can make a decision about the outcome of the game. I can demonstrate why we need this through two test cases.

First, consider this test case:

```
5
1
3
1
1
1
```

The Apples scored $5 * 3 + 1 * 2 + 3 = 20$ points, and the Bananas scored $1 * 3 + 1 * 2 + 1 = 6$ points. The Apples won the game, so the correct output is

```
A
```

Second, consider this test case, where the Apples' and Bananas' scores have been swapped:

```
1
1
1
5
1
3
```

This time, the Bananas won the game, so the correct output is

```
B
```

Our program must be able to compare the total points scored by the Apples and the total points scored by the Bananas and use the result of that comparison to choose whether to output `A`, `B`, or `T`.

We can use Python's `if` statement to make these kinds of decisions. A *condition* is an expression that's true or false, and an `if` statement uses conditions to determine what to do. `if` statements lead to *conditional execution*, so named because the execution of our program is influenced by conditions.

We'll first learn about a new type that lets us represent true or false values, and how we can build expressions of this type. Then, we'll use such expressions to write `if` statements.

The Boolean Type

Pass an expression to Python's `type` function, and it'll tell you the type of the expression's value:

```
>>> type(14)
<class 'int'>
>>> type(9.5)
```

```
<class 'float'>
>>> type('hello')
<class 'str'>
>>> type(12 + 15)
<class 'int'>
```

One Python type we haven't met yet is the Boolean (`bool`) type. Unlike integers, strings, and floats, which have billions of possible values, there are only two Boolean values: `True` and `False`.

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

What can we do with these values? With numbers, we had mathematical operators like `+` and `-` that let us combine values into more complex expressions. We'll need a new set of operators that work with Boolean values.

Relational Operators

Is `5` greater than `2`? Is `4` less than `1`? We can make such comparisons using Python's *relational operators*. They produce `True` or `False` and are therefore used to write *Boolean expressions*.

The `>` operator takes two operands and returns `True` if the first is greater than the second, and `False` otherwise:

```
>>> 5 > 2
True
>>> 9 > 10
False
```

Similarly, we have the `<` operator for less-than:

```
>>> 4 < 1
False
>>> -2 < 0
True
```

There's also `>=` for greater-than-or-equal-to, and `<=` for less-than-or-equal-to:

```
>>> 4 >= 2
True
>>> 4 >= 4
True
```

```
>>> 4 >= 5
False
>>> 8 <= 6
False
```

To determine equality, we use the `==` operator. That's two equal signs, not one. Remember that one equal sign, `=`, is used in an assignment statement; it has nothing to do with checking equality.

```
>>> 5 == 5
True
>>> 15 == 10
False
```

For inequality, we use the `!=` operator. It returns `True` if the operands are not equal and `False` if they are equal:

```
>>> 5 != 5
False
>>> 15 != 10
True
```

Real programs wouldn't evaluate expressions whose values we already know. We don't need Python to tell us that `15` doesn't equal `10`, for example. More typically, we'd use variables in these kinds of expressions. For example, `number != 10` is an expression whose value depends on what `number` refers to.

The relational operators also work on strings. When checking equality, case matters:

```
>>> 'hello' == 'hello'
True
>>> 'Hello' == 'hello'
False
```

One string is less than another if it comes first in alphabetical order:

```
>>> 'brave' < 'cave'
True
>>> 'cave' < 'cavern'
True
>>> 'orange' < 'apple'
False
```

But things can be surprising when lowercase and uppercase characters are both involved:

```
>>> 'apple' < 'Banana'
False
```

Weird, right? It has to do with the way that characters are stored internally in a computer. Generally, uppercase characters come alphabetically before lowercase characters. And check this out:

```
>>> '10' < '4'
True
```

If these were numbers, then the result would be `False`. But strings are compared character by character from left to right. Python compares the `'1'` and `'4'`, and because `'1'` is smaller, it returns `True`. Be sure that your values have the types you think they have!

One relational operator that works on strings but not numbers is `in`. It returns `True` if the first string occurs at least once in the second, and `False` otherwise:

```
>>> 'ppl' in 'apple'
True
>>> 'ale' in 'apple'
False
```

CONCEPT CHECK

What is the output of the following code?

```
a = 3
b = (a != 3)
print(b)
```

A. `True`
B. `False`
C. `3`
D. This code produces a syntax error.

Answer: B. The expression `a != 3` evaluates to `False`; `b` is then made to refer to this `False` value.

The if Statement

We'll now explore several variations of Python's `if` statement.

if by Itself

Suppose we have our final scores in two variables, `apple_total` and `banana_total`, and we want to output `A` if `apple_total` is greater than `banana_total`. Here's how we can do that:

```
>>> apple_total = 20
>>> banana_total = 6
>>> if apple_total > banana_total:
...     print('A')
...
A
```

Python outputs `A`, as we'd expect.

An `if` statement starts with the keyword `if`. A *keyword* is a word that has special meaning to Python and cannot be used as a variable name. The keyword `if` is followed by a Boolean expression, followed by a colon, followed by one or more indented statements. The indented statements are often referred to as the *block* of the `if` statement. The block executes if the Boolean expression is `True` and is skipped if the Boolean expression is `False`.

Notice that the prompt changes from `>>>` to `...`. That's a reminder that we're inside the block of the `if` statement and must indent the code. I've chosen to indent by four spaces, so to indent the code, press the spacebar four times. Some Python programmers press the TAB key to indent, but we'll exclusively use spaces in this book.

Once you type `print('A')` and hit ENTER, you should see another `...` prompt. Since we don't have anything else to put in this `if` statement, press ENTER again to dismiss this prompt and return to the `>>>` prompt. This extra press of ENTER is a quirk of the Python shell; such blank lines are not required when we write a Python program in a file.

Let's see an example of putting two statements in the block of an `if` statement:

```
>>> apple_total = 20
>>> banana_total = 6
>>> if apple_total > banana_total:
...     print('A')
...     print('Apples win!')
...
A
Apples win!
```

Both `print` calls execute, producing two lines of output.

Let's try another `if` statement, this one with a Boolean expression that's `False`:

```
>>> apple_total = 6
>>> banana_total = 20
>>> if apple_total > banana_total:
...     print('A')
```

```
...
```

The `print` function is *not* called this time: `apple_total > banana_total` is `False`, so the block of the `if` statement is skipped.

if with elif

Let's use three successive `if` statements to print `A` if the Apples win, `B` if the Bananas win, and `T` if it's a tie:

```
>>> apple_total = 6
>>> banana_total = 6
>>> if apple_total > banana_total:
...     print('A')
...
>>> if banana_total > apple_total:
...     print('B')
...
>>> if apple_total == banana_total:
...     print('T')
...
T
```

The blocks of the first two `if` statements are skipped, because their Boolean expressions are `False`. But the block of the third `if` statement executes, producing the `T`.

When you put one `if` statement after another, they're independent. Each Boolean expression is evaluated, regardless of whether the previous Boolean expressions were `True` or `False`.

For any given values of `apple_total` and `banana_total`, only one of our `if` statements can run. For example, if `apple_total < banana_total` is `True`, then the first `if` statement will run, but the other two will not. It's possible to write the code to highlight that only one block of code is allowed to run. Here's how we can do that:

```
❶ >>> if apple_total > banana_total:
...     print('A')
❷ ... elif banana_total > apple_total:
...     print('B')
... elif apple_total == banana_total:
...     print('T')
...
T
```

This is now a single `if` statement, not three separate `if` statements. For this reason, don't press ENTER at the `...` prompt; instead, type the `elif` line.

To execute this `if` statement, Python begins by evaluating the first Boolean expression ❶. If it's `True`, then `A` is output, and the rest of the `elif`s are skipped. If it's `False`, then Python continues, evaluating the second Boolean expres-

sion ❷. If it's `True`, then `B` is output, and the remaining `elif` is skipped. If it's `False`, then Python continues, evaluating the third Boolean expression ❸. If it's `True`, then `T` is output.

The keyword `elif` stands for "else-if." Use this as a reminder that an `elif` block is checked only if nothing "else" before it in the `if` statement was executed.

This version of the code is equivalent to the previous code where we used three separate `if` statements. Had we wanted to allow the possibility of executing more than one block, we'd have to use three separate `if` statements, not a single `if` statement with `elif` blocks.

if with else

We can use the `else` keyword to run code if all the Boolean expressions in the `if` statement are `False`. Here's an example:

```
>>> if apple_total > banana_total:
...     print('A')
... elif banana_total > apple_total:
...     print('B')
... else:
...     print('T')
...
T
```

Python evaluates the Boolean expressions from top to bottom. If any of them is `True`, Python runs the associated block and skips the rest of the `if` statement. If all the Boolean expressions are `False`, Python executes the `else` block.

Notice that there is no longer a test for `apple_total == banana_total`. The only way to get to the `else` part of the `if` statement is if `apple_total > banana_total` is `False` and `banana_total > apple_total` is `False`, that is, if the values are equal.

Should you use separate `if` statements? An `if` statement with `elif`s? An `if` statement with an `else`? It often comes down to preference. Use a chain of `elif`s if you want at most one block of code to execute. An `else` can help make the code clearer and removes the need to write a catchall Boolean expression. What's far more important than the precise styling of an `if` statement is writing correct logic!

CONCEPT CHECK

What is the value of `x` after the following code runs?

```
x = 5
if x > 2:
    x = -3
if x > 1:
    x = 1
else:
    x = 3
```

A. `-3`
B. `1`
C. `2`
D. `3`
E. `5`

Answer: D. Because `x > 2` is `True`, the block of the first `if` statement executes. The assignment `x = -3` makes `x` refer to `-3`. Now for the second `if` statement. Here, `x > 1` is `False`, so the `else` block runs, and `x = 3` makes `x` refer to `3`. I'd suggest changing `if x > 1` to `elif x > 1` and observing how the behavior of the program changes!

CONCEPT CHECK

Do the following two snippets of code do exactly the same thing? Assume that `temperature` already refers to a number.

Snippet 1:

```
if temperature > 0:
    print('warm')
elif temperature == 0:
    print('zero')
else:
    print('cold')
```

Snippet 2:

```
if temperature > 0:
    print('warm')
elif temperature == 0:
    print('zero')
print('cold')
```

A. Yes
B. No

Answer: B. Snippet 2 *always* prints `cold` as its final line of output, because `print('cold')` is not indented! It is not associated with any `if` statement.

Solving the Problem

It's time to solve Winning Team. In this book, I'll generally present the full code and then discuss it. But as our solution here is longer than those in Chapter 1, I've decided in this case to present the code in three pieces before presenting it as a whole.

First, we need to read the input. This requires six calls of `input`, because we have two teams and three pieces of information for each team. We also need to convert each piece of input to an integer. Here's the code:

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())
```

Second, we need to determine the number of points scored by the Apples and the Bananas. For each team, we add the points from three-point, two-point, and one-point plays. We can do that as follows:

```
apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one
```

Third, we produce the output. If the Apples win, we output `A`; if the Bananas win, we output `B`; otherwise, we know that the game is a tie, so we output `T`. We use an `if` statement to do this, as follows:

```
if apple_total > banana_total:
    print('A')
```

```
elif banana_total > apple_total:
    print('B')
else:
    print('T')
```

That's all the code we need. See Listing 2-1 for the complete solution.

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())

apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one

if apple_total > banana_total:
    print('A')
elif banana_total > apple_total:
    print('B')
else:
    print('T')
```

Listing 2-1: Solving Winning Team

If you submit our code to the judge, you should see that all test cases pass.

CONCEPT CHECK

Does the following version of the code correctly solve the problem?

```
apple_three = int(input())
apple_two = int(input())
apple_one = int(input())

banana_three = int(input())
banana_two = int(input())
banana_one = int(input())

apple_total = apple_three * 3 + apple_two * 2 + apple_one
banana_total = banana_three * 3 + banana_two * 2 + banana_one

if apple_total < banana_total:
    print('B')
```

```
elif apple_total > banana_total:
    print('A')
else:
    print('T')
```

A. Yes
B. No

Answer: A. The operators and order of the code are different, but the code is still correct. If the Apples lose, we output `B` (because the Bananas win); if the Apples win, we output `A`; otherwise, we know that the game is a tie, so we output `T`.

Before continuing, you might like to try solving exercise 1 from "Chapter Exercises" on page 45.

Problem #4: Telemarketers

Sometimes we need to encode more complex Boolean expressions than those that we have seen so far. In this problem, we'll learn about Boolean operators that help us do this.

This is DMOJ problem `ccc18j1`.

The Challenge

In this problem, we'll assume that phone numbers are four digits. A phone number belongs to a telemarketer if its four digits satisfy all three of the following properties:

- The first digit is `8` or `9`.
- The fourth digit is `8` or `9`.
- The second and third digits are the same.

For example, a phone number whose four digits are `8119` belongs to a telemarketer.

Determine whether a phone number belongs to a telemarketer, and indicate whether we should answer the phone or ignore it.

Input

There are four lines of input. These lines give the first, second, third, and fourth digits of the phone number, respectively. Each digit is an integer between 0 and 9.

Output

If the phone number belongs to a telemarketer, output `ignore`; otherwise, output `answer`.

Boolean Operators

What has to be true about a phone number that belongs to a telemarketer? Its first digit has to be `8` *or* `9`. *And*, its fourth digit has to be `8` *or* `9`. *And*, the second and third digits have to be the same. We can encode this "or" and "and" logic using Python's *Boolean operators*.

or Operator

The `or` operator takes two Boolean expressions as its operands. It returns `True` if at least one operand is `True`, and `False` otherwise:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

The only way to get `False` out of the `or` operator is if both of its operands are `False`.

We can use `or` to tell us whether a digit is an `8` or a `9`:

```
>>> digit = 8
>>> digit == 8 or digit == 9
True
>>> digit = 3
>>> digit == 8 or digit == 9
False
```

Remember from "Integer and Floating-Point Numbers" on page 9 that Python uses operator precedence to determine the order that operators are applied. The precedence of `or` is lower than the precedence of relational operators, which means that we don't often need parentheses around operands. For example, in `digit == 8 or digit == 9`, the two operands to `or` are `digit == 8` and `digit == 9`. It's the same as if we'd written it as `(digit == 8) or (digit == 9)`.

In English, it makes sense if someone says "if the digit is 8 or 9." But writing that won't work in Python:

```
>>> digit = 3
>>> if digit == 8 or 9:
...     print('yes!')
...
```

```
yes!
```

Notice that I've (incorrectly!) written the second operand as `9` instead of `digit == 9`. Python responds by outputting `yes!`, which is certainly not what we'd want given that `digit` refers to `3`. The reason is that Python considers nonzero numbers to be `True`. Since `9` is considered `True`, this makes the whole `or` expression `True`. Carefully double-check your Boolean expressions to avoid these kinds of mistakes when translating from natural language to Python.

and Operator

The `and` operator returns `True` if both of its operands are `True`, and `False` otherwise:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

The only way to get `True` out of the `And` operator is if both of its operands are `True`.

The precedence of `and` is higher than `or`. Here's an example of why this matters:

```
>>> True or True and False
True
```

Python interprets that expression like this, with the `and` happening first:

```
>>> True or (True and False)
True
```

The result is `True` because the first operand of `or` is `True`.

We can force the `or` to happen first by including parentheses:

```
>>> (True or True) and False
False
```

The result is `False` because the second operand of `and` is `False`.

not Operator

Another important Boolean operator is `not`. Unlike `or` and `and`, `not` takes only one operand (not two). If its operand is `True`, not returns `False`, and vice versa:

```
>>> not True
False
>>> not False
True
```

The precedence of `not` is higher than `or` and `and`.

CONCEPT CHECK

Here's an expression and versions of that expression with parentheses. Which of them evaluates to `True`?

A. `not True and False`
B. `(not True) and False`
C. `not (True and False)`
D. None of the above

Answer: C. The expression `(True and False)` evaluates to `False`; the `not` therefore makes the full expression `True`.

CONCEPT CHECK

Consider the expression `not a or b`.

Which of the following makes the expression `False`?

A. `a False, b False`
B. `a False, b True`
C. `a True, b False`
D. `a True, b True`
E. More than one of the above

Answer: C. If `a` is `True`, then `not a` is `False`. Since `b` is `False`, too, both operands to `or` are `False`, so the whole expression evaluates to `False`.

Solving the Problem

With Boolean operators at the ready, we can tackle the Telemarketers problem. Our solution is in Listing 2-2.

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

❶ if ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('ignore')
else:
    print('answer')
```

Listing 2-2: Solving Telemarketers

As in Winning Team, we start by reading the input and converting it to integers.

The high-level structure of our `if` statement ❶ is three expressions connected by `and` operators; each of them must be `True` for the entire expression to be `True`. We require that the first number be `8` or `9`, that the fourth number be `8` or `9`, and that the second and third numbers be equal. If all three of these conditions hold, then we know that the phone number belongs to a telemarketer, and we output `ignore`. Otherwise, the phone number does not belong to a telemarketer, and we output `answer`.

I've split the Boolean expression over three lines. This requires wrapping the entire expression in an additional pair of parentheses, as I have done. (Without those parentheses, you'll get a syntax error, because there's no indication to Python that the expression is continuing on the next line.)

Python style guides suggest that a line be no longer than 79 characters. A line with the full Boolean expression would squeak in there at 76 characters. But I think the three-line version is clearer, highlighting each condition that must be `True` on its own line.

We have a good solution here. To explore a little further, let's discuss some alternate approaches.

Our code uses a Boolean expression to detect when a phone number belongs to a telemarketer. We could have also chosen to write code that detects when a phone number does *not* belong to a telemarketer. If the phone number doesn't belong to a telemarketer, we should output `answer`; otherwise, we should output `ignore`.

If the first digit isn't `8` and isn't `9`, then the phone number doesn't belong to a telemarketer. Or, if the fourth digit isn't `8` and isn't `9`, then the phone number doesn't belong to a telemarketer. Or, if the second and third digits aren't equal, then the phone number doesn't belong to a telemarketer. If even one of these expressions is `True`, then the phone number doesn't belong to a telemarketer.

See Listing 2-3 for a version of the code that captures this logic.

```
num1 = int(input())
num2 = int(input())
```

```
num3 = int(input())
num4 = int(input())

if ((num1 != 8 and num1 != 9) or
        (num4 != 8 and num4 != 9) or
        (num2 != num3)):
    print('answer')
else:
    print('ignore')
```

Listing 2-3: Solving Telemarketers, alternate approach

It's not easy getting all of those `!=`, `or`, and `and` operators correct! Notice, for example, that we've had to change all `==` operators to `!=`, all `or` operators to `and`, and all `and` operators to `or`.

An alternate approach is to use the `not` operator to negate the "is-a-telemarketer" expression in one shot. See Listing 2-4 for that code.

```
num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

if not ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('answer')
else:
    print('ignore')
```

Listing 2-4: Solving Telemarketers, not operator

Which of these solutions do you find most intuitive? There's often more than one way to structure the logic of an `if` statement, and we should use the one that's easiest to get right. To me, Listing 2-2 is the most natural, but you may feel otherwise!

Choose your favorite version and submit it to the judge. You should see that all test cases pass.

Comments

We should always strive to make our programs as clear as possible. This helps to avoid introducing errors when programming and makes it easier to fix our code when errors do slip in. Meaningful variable names, spaces around operators, blank lines to segment the program into its logical pieces, simple `if` statement logic: all of these practices can improve the quality of the code we write. Another good habit is adding *comments* to our code.

A comment is introduced by the `#` character and continues until the end of the line. Python ignores comments, so they have no impact on what our program does. We add comments to remind ourselves, or others, about de-

sign decisions that we've made. Assume that the person reading the code knows Python, so avoid comments that simply restate what the code is doing. Here's code with an unnecessary comment:

```
>>> x = 5
>>> x = x + 1  # Increase x by 1
```

That comment adds nothing beyond what we already know about assignment statements.

See Listing 2-5 for a version of Listing 2-2 with comments.

```
❶ # ccc18j1, Telemarketers

num1 = int(input())
num2 = int(input())
num3 = int(input())
num4 = int(input())

❷ # Telemarketer number: first digit 8 or 9, fourth digit 8 or 9,
# second digit and third digit are same
if ((num1 == 8 or num1 == 9) and
        (num4 == 8 or num4 == 9) and
        (num2 == num3)):
    print('ignore')
else:
    print('answer')
```

Listing 2-5: Solving Telemarketers, comments added

I've added three comment lines: the one at the top ❶ reminds us of the problem code and name, and the two before the `if` statement ❷ remind us of the rules for detecting a telemarketer phone number.

Don't go overboard with comments. Whenever possible, write code that doesn't require comments in the first place. But for tricky code or to document why you chose to do something in a particular way, a well-placed comment now can save time and frustration later.

Input and Output Redirection

When you submit Python code to the judge, it runs many test cases to determine whether your code is correct. Is someone there, dutifully waiting for new code and then frantically hammering test cases at it from the keyboard?

No way! It's all automated. There's no one typing test cases at the keyboard. How does the judge test our code, then, if we satisfy a call to `input` by typing something from the keyboard?

The truth is that `input` isn't necessarily reading input from the keyboard. It's reading from a source of input called *standard input*, which, by default, is the keyboard.

It's possible to change standard input so that it refers to a file rather than the keyboard. The technique is called *input redirection*, and it's what the judge uses to provide input.

We can also try input redirection ourselves. For programs whose input is small—just a line of text or a couple of integers—input redirection may not save us much. But for programs whose test cases can be tens or hundreds of lines long, input redirection makes it much easier to test our work. Rather than typing the same test case over and over, we can store it in a file and then run our program on it as many times as we want.

Let's try input redirection on Telemarketers. Navigate to your *programming* folder and create a new file called *telemarketers_input.txt*. In that file, type the following:

```
8
1
1
9
```

The problem specifies that we should provide one integer per line, so we've written them one per line here.

Save the file. Now enter **`python telemarketers.py < telemarketers_input.txt`** to run your program using input redirection. Your program should output `ignore`, just as it would if you'd typed the test case from the keyboard.

The `<` symbol instructs your operating system to use a file rather than the keyboard to provide input. After the `<` symbol comes the name of the file that contains the input.

To try your program on different test cases, just modify the *telemarketers_input.txt* file and run your program again.

We can also change where our output goes, though we won't need to for this book. The `print` function outputs to *standard output*, which, by default, is the screen. We can change standard output so that it instead refers to a file. We do so using *output redirection*, which is written as a `>` symbol followed by a filename.

Enter **`python telemarketers.py > telemarketers_output.txt`** to run your program using output redirection. Provide four integers of input, and you should be back to your operating system prompt. But you shouldn't see any output from your Telemarketers program! That's because we've redirected the output to file `telemarketers_output.txt`. If you open `telemarketers_output.txt` in your text editor, you should see the output there.

Be careful with output redirection. If you use a filename that already exists, your old file will be overwritten! Always double-check that you're using the filename you intended.

Summary

In this chapter, you learned how to use `if` statements to direct what your programs do. The key ingredient of an `if` statement is a Boolean expression, which is an expression with a `True` or `False` value. To build up Boolean ex-

pressions, we use relational operators such as `==` and `>=`, and we use Boolean operators such as `and` and `or`.

Deciding what to do based on what is `True` and `False` makes our programs more flexible, able to adapt to the situation at hand. But our programs are still limited to handling small amounts of input and output—whatever we can read with individual calls to `input` and `print`. In the next chapter, we'll start learning about loops, which let us repeat code so that we can process as much input and output as we like.

Want to work with 100 values? How about 1,000? And with just a small amount of Python code? It is a little early for me to be provoking you, I know, because you still have the following exercises to do. But when you're ready, read on!

Chapter Exercises

Here are some exercises for you to try:

1. DMOJ problem `ccc06j1`, Canadian Calorie Counting
2. DMOJ problem `ccc15j1`, Special Day
3. DMOJ problem `ccc15j2`, Happy or Sad
4. DMOJ problem `dmopc16c1p0`, C.C. and Cheese-Kun
5. DMOJ problem `ccc07j1`, Who is in the Middle

Notes

Winning Team is originally from the 2019 Canadian Computing Competition, Junior Level.

Telemarketers is originally from the 2018 Canadian Computing Competition, Junior Level.