Chapter 4

GUESS THE NUMBER

Topics Covered In This Chapter:

- `import` statements
- Modules
- Arguments
- `while` statements
- Conditions
- Blocks
- Booleans
- Comparison operators
- The difference between `=` and `==`
- `if` statements
- The `break` keyword
- The `str()` and `int()` functions
- The `random.randint()` function

The "Guess the Number" Game

In this chapter, you're going to make a "Guess the Number" game. The computer will think of a random number from 1 to 20, and ask you to guess it. The computer will tell you if each guess is too high or too low. You win if you guess the number within six tries.

This is a good game to code because it uses random numbers, loops, and input from the user in a fairly short program. You'll learn how to convert values to different data types (and why you would need to do this). Since this program is a game, call the user the **player**, but "user" would be correct too.

Sample Run of "Guess the Number"

Here's what the program looks like to the player when run. The text that the player types in is in **bold**.

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1 and 20.
Take a guess.
```

```
10
Your guess is too high.
Take a guess.
2
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3 guesses!
```

Enter this code exactly as it appears here, and then save it by clicking on **File ► Save As**. Give it the filename *guess.py*. Then press **F5** to run the program. Don't worry if you don't understand the code now, I'll explain it step by step.

Guess the Number's Source Code

Here's the source code for the "Guess the Number" game. When you enter this code into the file editor, be sure to pay attention to the spacing at the front of some of the lines. Some lines have four or eight spaces of indentation.

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not Python 2. When the IDLE window starts, it will say something like "Python 3.4.0" at the top. If you have Python 2 installed, you can have Python 3 installed at the same time. To download Python 3, go to https://python.org/download/.

You can download this code from http://invpy.com/guess.py. If you get errors after typing this code in, compare it to the book's code with the online diff tool at http://invpy.com/diff/guess.

guess.py

```
 1. # This is a guess the number game.
 2. import random
 3.
 4. guessesTaken = 0
 5.
 6. print('Hello! What is your name?')
 7. myName = input()
 8.
 9. number = random.randint(1, 20)
10. print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
11.
12. while guessesTaken < 6:
13.     print('Take a guess.') # There are four spaces in front of print.
14.     guess = input()
15.     guess = int(guess)
16.
```

```
17.     guessesTaken = guessesTaken + 1
18.
19.     if guess < number:
20.         print('Your guess is too low.') # There are eight spaces in front
of print.
21.
22.     if guess > number:
23.         print('Your guess is too high.')
24.
25.     if guess == number:
26.         break
27.
28. if guess == number:
29.     guessesTaken = str(guessesTaken)
30.     print('Good job, ' + myName + '! You guessed my number in ' +
guessesTaken + ' guesses!')
31.
32. if guess != number:
33.     number = str(number)
34.     print('Nope. The number I was thinking of was ' + number)
```

Take look at each line of code in turn to see how this program works.

The `import` statement

```
1. # This is a guess the number game.
2. import random
```

The first line is a comment. Remember that Python will ignore everything after the # sign. This just reminds us what this program does.

The second line is an **import statement**. Remember, statements are instructions that perform some action but don't evaluate to a value like expressions do. You've already seen statements: assignment statements store a value into a variable (but the statement itself doesn't evaluate to anything).

While Python includes many built-in functions, some functions exist in separate programs called modules. **Modules** are Python programs that contain additional functions. Use the functions by importing their modules into your program with an `import` statement.

The `import` statement has the `import` keyword followed by the module name. Line 2 imports the module named `random`. The `random` module contains several functions related to random numbers. One of these functions will come up with the random number for the user to guess.

```
4. guessesTaken = 0
```

This line creates a new variable named `guessesTaken`. You'll store the number of guesses the player has made in this variable. Since the player hasn't made any guesses so far, store the integer `0` here.

```
6. print('Hello! What is your name?')
7. myName = input()
```

Lines 6 and 7 are the same as the lines in the Hello World program that you saw in Chapter 3. Programmers often reuse code from their other programs to save themselves work.

Line 6 is a function call to the `print()` function. Remember that a function is like a mini-program inside your program. When your program calls a function, it runs this mini-program. The code inside the `print()` function displays on the screen the string argument you passed it.

Line 7 lets the user type in their name and stores it in the `myName` variable. (Remember, the string might not really be the player's name. It's just whatever string the player typed. Computers are dumb and just follow their programs no matter what.)

The `random.randint()` function

```
9. number = random.randint(1, 20)
```

Line 9 calls a new function named `randint()` and stores the return value in `number`. Remember, function calls can be part of expressions because they evaluate to a value.

Because the `randint()` function is provided by the `random` module, precede it with `random.` (don't forget the period!) to tell Python that the function `randint()` is in the `random` module.

The `randint()` function will return a random integer between (and including) the two integer arguments you pass to it. Line 9 passes `1` and `20` between the parentheses that follow the function name (separated by a comma). The random integer that `randint()` returns is stored in a variable named `number`; this is the secret number the player is trying to guess.

Just for a moment, go back to the interactive shell and enter `import random` to import the `random` module. Then enter `random.randint(1, 20)` to see what the function call evaluates to. It will return an integer between `1` and `20`. Repeat the code again and the function call will probably return a different integer. The `randint()` function returns some random integer each time, just as rolling dice you'll get a random number each time.

```
>>> import random
```

```
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
```

Use the `randint()` function when you want to add randomness to your games. And you'll use randomness in most games. Think of how many board games use dice.

You can also try out different ranges of numbers by changing the arguments. For example, enter `random.randint(1, 4)` to only get integers between `1` and `4` (including both `1` and `4`). Or try `random.randint(1000, 2000)` to get integers between `1000` and `2000`. For example, enter the following into the interactive shell. The results you get when you call the `random.randint()` function will probably be different (it is random, after all).

```
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
```

You can change the game's code slightly to make the game behave differently. Try changing line 9 and 10 from this:

```
 9. number = random.randint(1, 20)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 20.')
```

…into these lines:

```
 9. number = random.randint(1, 100)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 100.')
```

And now the computer will think of an integer between `1` and `100` instead of `1` and `20`. Changing line 9 will change the range of the random number, but remember to change line 10 so that the game also tells the player the new range instead of the old one.

Welcoming the Player

```
10. print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
```

On line 10 the `print()` function welcomes the player by name, and tells them that the computer is thinking of a random number.

It may look like there's more than one string argument in line 10, but look at the line carefully. The plus signs concatenate the three strings to evaluate down to one string. And that is the one string argument passed to the `print()` function. If you look closely, you'll see that the commas are inside the quotes and part of the strings themselves.

Loops

```
12. while guessesTaken < 6:
```

Line 12 is a `while` statement, which indicates the beginning of a `while` loop. **Loops** let you execute code over and over again. However, you need to learn a few other concepts first before learning about loops. Those concepts are blocks, Booleans, comparison operators, conditions, and the `while` statement.

Blocks

Several lines of code can group together in a block. A **block** of code has the same minimum amount of indentation. You can tell where a block begins and ends by looking at the number of spaces at the front of the lines. This is the line's **indentation**.

A block begins when a line's indentation increases (usually by four spaces). Any following line also indented by four spaces is part of the block. The block ends when there is a line of code with the same indentation before the block started. This means blocks can exist within other blocks.

Figure 4-1 is a diagram of code with the blocks outlined and numbered. The spaces are black squares to make them easier to count.

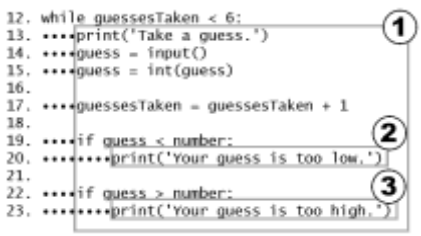

Figure 4-1: Blocks and their indentation. The black dots represent spaces.

In Figure 4-1, line 12 has no indentation and isn't inside any block. Line 13 has an indentation of four spaces. Since this indentation is larger than the previous line's indentation, a new block has

started. This block is labeled (1) in Figure 4-1. This block will continue until a line with zero spaces (the original indentation before the block began). Blank lines are ignored.

Line 20 has an indentation of eight spaces. Eight spaces is more than four spaces, which starts a new block. This block is labeled (2) in Figure 4-1. This block is inside of another block.

Line 22 only has four spaces. Because the indentation has decreased, you know that block has ended. Line 20 is the only line in that block. Line 22 is in the same block as the other lines with four spaces.

Line 23 increases the indentation to eight spaces, so again a new block has started. It is labeled (3) in Figure 4-1.

To recap, line 12 isn't in any block. Lines 13 to 23 all in one block (marked as block 1). Line 20 is in a block in a block (marked as block 2). And line 23 is the only line in another block in a block (marked as block 3).

The Boolean Data Type

The Boolean data type has only two values: `True` or `False`. These values must be typed with a capital "T" and "F". The rest of the value's name must be in lowercase. You will use Boolean values (also called **bools**) with comparison operators to form conditions. (Conditions are explained later.)

The data types that have been introduced so far are integers, floats, strings, and now bools.

Comparison Operators

Line 12 has a `while` statement:

```
12. while guessesTaken < 6:
```

The expression that follows the `while` keyword (the `guessesTaken < 6` part) contains two values (the value in the variable `guessesTaken`, and the integer value `6`) connected by an operator (the `<` sign, the "less than" sign). The `<` sign is called a **comparison operator**.

Comparison operators compare two values and evaluate to a `True` or `False` Boolean value. A list of all the comparison operators is in Table 4-1.

Table 4-1: Comparison operators.

Operator Sign	Operator Name
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

You've already read about the +, -, *, and / math operators. Like any operator, the comparison operators combine with values to form expressions such as `guessesTaken < 6`.

Conditions

A **condition** is an expression that combines two values with a comparison operator (such as `<` or `>`) and evaluates to a Boolean value. A condition is just another name for an expression that evaluates to `True` or `False`. Conditions are used in `while` statements (and a few other places, explained later.)

For example, the condition `guessesTaken < 6` asks, "is the value stored in `guessesTaken` less than the number `6`?" If so, then the condition evaluates to `True`. If not, the condition evaluates to `False`.

In the case of the "Guess the Number" program, on line 4 you stored the value `0` in `guessesTaken`. Because `0` is less than `6`, this condition evaluates to the Boolean value of `True`. The evaluation would look like this:

```
guessesTaken < 6
             ▼
           0 < 6
             ▼
            True
```

Experiment with Booleans, Comparison Operators, and Conditions

Enter the following expressions in the interactive shell to see their Boolean results:

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10
False
>>> 10 < 11
True
```

```
>>> 10 < 10
False
```

The condition `0 < 6` returns the Boolean value `True` because the number `0` is less than the number `6`. But because `6` isn't less than `0`, the condition `6 < 0` evaluates to `False`. `50` isn't less than `10`, so `50 < 10` is `False`. `10` is less than `11`, so `10 < 11` is `True`.

Notice that `10 < 10` evaluates to `False` because the number `10` isn't smaller than the number `10`. They are exactly the same size. If Alice were the same height as Bob, you wouldn't say that Alice is taller than Bob or that Alice is shorter than Bob. Both of those statements would be false.

Now try entering these expressions into the interactive shell:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Goodbye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Goodbye' != 'Hello'
True
```

Try not to confuse the assignment operator (=) and the "equal to" comparison operator (==). The equal sign (=) is used in assignment statements to store a value to a variable, while the equal-equal sign (==) is used in expressions to see whether two values are equal. It's easy to accidentally use one when you meant to use the other.

Just remember that the "equal to" comparison operator (==) has two characters in it, just as the "not equal to" comparison operator (!=) has two characters in it.

String and integer values will not be equal to each other. For example, try entering the following into the interactive shell:

```
>>> 42 == 'Hello'
False
>>> 42 != '42'
```

```
True
```

Looping with `while` statements

The `while` statement marks the beginning of a loop. Loops can execute the same code repeatedly. When the execution reaches a `while` statement, it evaluates the condition next to the `while` keyword. If the condition evaluates to `True`, the execution moves inside the while-block. (In your program, the while-block begins on line 13.) If the condition evaluates to `False`, the execution moves all the way past the while-block. (In "Guess the Number", the first line after the while-block is line 28.)

A `while` statement always has a colon (the : sign) after the condition.

```
12. while guessesTaken < 6:
```

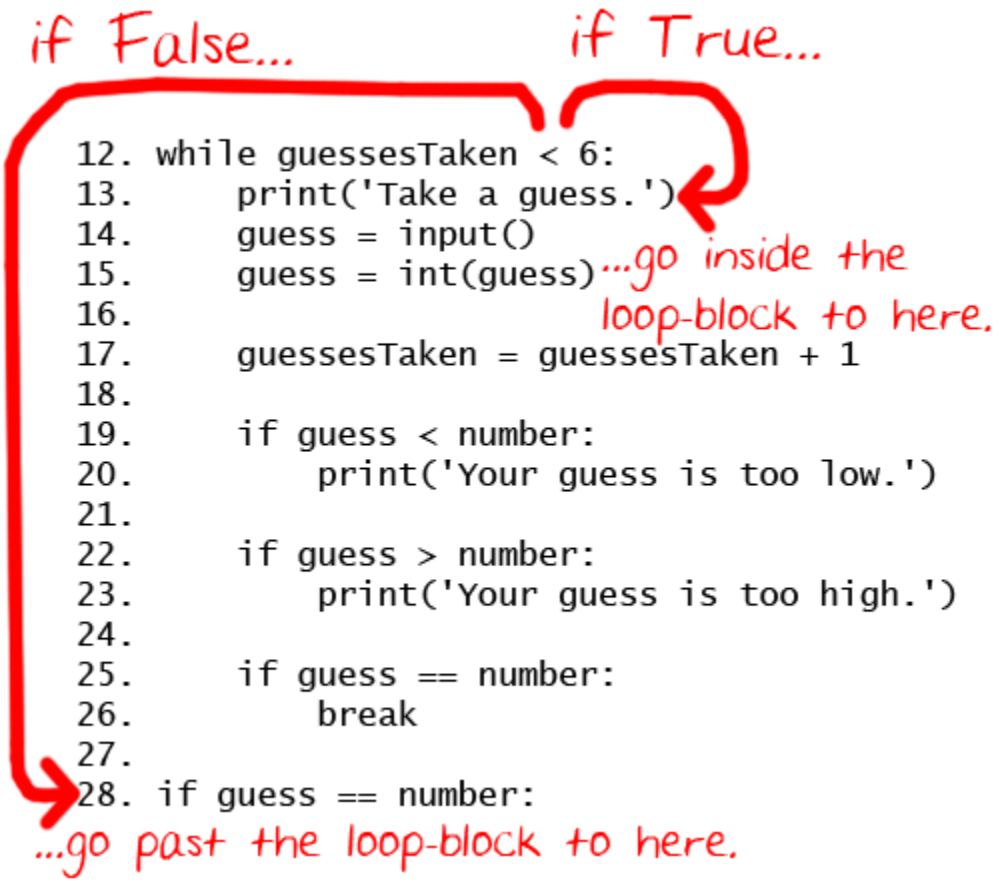

Figure 4-2: The while loop's condition.

Figure 4-2 shows how the execution flows depending on the condition. If the condition evaluates to `True` (which it does the first time, because the value of `guessesTaken` is `0`), execution will enter the while-block at line 13 and keep going down. Once the program reaches the end of the while-block, instead of going down to the next line, the execution loops back up to the `while` statement's line (line 12) and re-evaluates the condition. As before, if the condition is `True` the execution enters the while-block again. Each time the execution goes through the loop is called an **iteration**.

This is how the loop works. As long as the condition is `True`, the program keeps executing the code inside the while-block repeatedly until the first time the condition is `False`. Think of the `while` statement as saying, "while this condition is true, keep iterating through the code in this block".

The code inside the while-block handles accepting a guess from the player and checking if it is greater than, less than, or equal to the secret number. You can change the game's difficulty by changing the number of guesses the player gets. To give the player only four guesses, change this line:

```
12. while guessesTaken < 6:
```

into this line:

```
12. while guessesTaken < 4:
```

Code later in the while-block increases the `guessesTaken` variable by `1` on each iteration. By setting the condition to `guessesTaken < 4`, you ensure that the code inside the loop only runs four times instead of six. This makes the game much more difficult. To make the game easier, set the condition to `guessesTaken < 8` or `guessesTaken < 10`. This will cause the loop to run a few more times and accept more guesses from the player.

Of course, if you removed line 17 (`guessesTaken = guessesTaken + 1`) altogether then the `guessesTaken` would never increase. The `while` loop's condition would always be `True`! This would give the player an unlimited number of guesses. The player might like this, but a programmer would consider this a bug.

The Player Guesses

Lines 13 to 17 ask the player to guess what the secret number is and lets them enter their guess. That number is stored in a variable named `guess`.

```
14.     print('Take a guess.') # There are four spaces in front of print.
15.     guess = input()
```

Converting Strings to Integers with the `int()` function

```
16.     guess = int(guess)
```

In line 15, you call a new function called `int()`. The `int()` function takes one argument and returns an integer value form that argument. Try entering the following into the interactive shell:

```
>>> int('42')
42
>>> int(42)
42
>>> int('forty-two')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('forty-two')
ValueError: invalid literal for int() with base 10: 'forty-two'
>>> 3 + int('2')
5
```

The `int('42')` call will return the integer value `42`.The `int(42)` call will do the same (though it is kind of pointless to get an integer value form of an integer value). However, even though you can pass a string to the `int()` function, you cannot pass just any string. Passing `'forty-two'` to `int()` will result in an error. The string you pass to `int()` must be made up of numbers.

The `3 + int('2')` line shows an expression that uses the return value of int() as part of an expression. It evaluates to the integer value `5`:

```
3 + int('2')
  ▼
3 + 2
  ▼
  5
```

Remember, the `input()` function returns **a string** of text the player typed. If the player types `5`, the `input()` function will return the string value `'5'`, not the integer value `5`. Python cannot use the `<` and `>` comparison operators to compare a string and an integer value:

```
>>> 4 < '5'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 < '5'
```

```
TypeError: unorderable types: int() < str()
```

On line 15 the `guess` variable originally held the string value of what the player typed. Line 16 overwrites the string value in `guess` with the integer value returned by `int()`. This lets the code later in the program compare if `guess` is greater than, less than, or equal to the secret number in the `number` variable. Python cannot compare a string value with an integer value to see if one is greater or less than the other, even if that string value is something like `'5'`.

One last thing: Calling `int(guess)` doesn't change the value in the `guess` variable. The code `int(guess)` is an expression that evaluates to the integer value form of the string stored in the `guess` variable. What changes `guess` is the assignment statement: `guess = int(guess)`

Incrementing Variables

```
17.     guessesTaken = guessesTaken + 1
```

Once the player has taken a guess, the number of guesses should be increased by one.

On the first iteration of the loop, `guessesTaken` has the value of `0`. Python will take this value and add `1` to it. `0 + 1` evaluates to `1`, which is stored as the new value of `guessesTaken`. Think of line 17 as meaning, "the `guessesTaken` variable should be one more than what it already is".

Adding one to a variable's integer or float value is **incrementing** the variable. Subtracting one from a variable's integer or float value is **decrementing** the variable.

`if` statements

Is the Player's Guess Too Low?

```
19.     if guess < number:
20.         print('Your guess is too low.') # There are eight spaces in front
of print.
```

Line 19 checks if the player's guess is less than the computer's secret number. If so, then the execution moves to line 20 and prints a message telling the player this. Line 19 is an `if` statement. The execution will run the code in the following block if the `if` statement's condition evaluates to True. If the condition is `False`, then the code in the if-block is skipped. Using `if` statements, you can make the program only run certain code when you want it to.

The `if` statement works almost the same as a `while` statement, too. But unlike the while-block, the execution doesn't jump back to the `if` statement at the end of the if-block. It just continues

down to the next line. In other words, `if` statements don't loop. See Figure 4-3 for a comparison of the two statements.

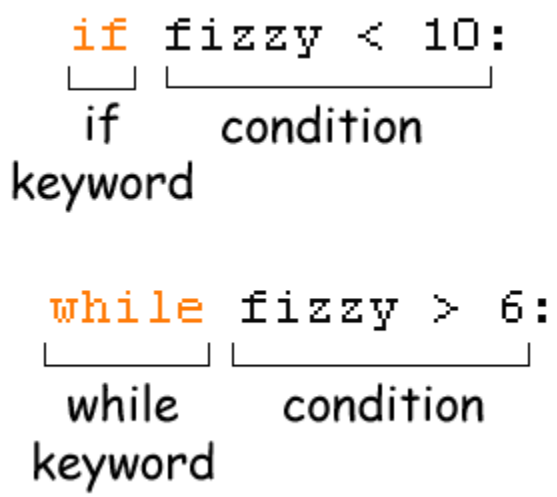

Figure 4-3: if and `while` statements.

Is the Player's Guess Too High?

```
22.     if guess > number:
23.         print('Your guess is too high.')
```

Line 22 checks if the player's guess is greater than the random integer. If this condition is `True`, then the `print()` function call tells the player that their guess is too big.

Leaving Loops Early with the `break` statement

```
25.     if guess == number:
26.         break
```

The `if` statement on line 25 checks if the guess is equal to the random integer. If it is, the program runs the break statement on line 26.

A **break statement** tells the execution to jump immediately out of the while-block to the first line after the end of the while-block. (The `break` statement doesn't bother rechecking the `while` loop's condition, it just breaks out of the loop immediately.)

The `break` statement is just the `break` keyword by itself, with no condition or colon.

If the player's guess isn't equal to the random integer, the execution reaches the bottom of the while-block. This means the execution will loop back to the top and recheck the condition on line 12 (`guessesTaken < 6`). Remember after the `guessesTaken = guessesTaken + 1` line of code executed, the new value of `guessesTaken` is `1`. Because `1 < 6` is `True`, the execution enters the loop again.

If the player keeps guessing too low or too high, the value of `guessesTaken` will change to `2`, then `3`, then `4`, then `5`, then `6`. When `guessesTaken` has the number `6` stored in it, the `while` statement's condition is `False`, since `6` isn't less than `6`. Because the `while` statement's condition is `False`, the execution moves to the first line after the while-block.

If the player guessed the number correctly, the `break` statement would move the execution to the first line after the while-block.

The remaining lines of code run when the player has finished guessing, either because the player guessed the correct number or run out of guesses.

Check if the Player Won

```
28. if guess == number:
```

Line 28 has no indentation, which means the while-block has ended and this is the first line after the while-block. The execution left the while-block either because the `while` statement's condition was `False` (when the player runs out of guesses) or the `break` statement executed (when the player guesses the number correctly). Line 28 checks to see if the player guessed correctly. If so, the execution enter the if-block at line 29.

```
29.     guessesTaken = str(guessesTaken)
30.     print('Good job, ' + myName + '! You guessed my number in ' +
guessesTaken + ' guesses!')
```

Lines 29 and 30 only execute if the condition in the `if` statement on line 28 was True (that is, if the player correctly guessed the computer's number).

Line 29 calls the new function `str()`, which returns the string form of an argument. This code gets the string form of the integer in `guessesTaken` since only strings concatenate strings together.

Line 30 concatenates strings together to tell the player they have won and how many guesses it took them. Only string values can concatenate to other strings. This is why line 29 had to change `guessesTaken` to the string form. Otherwise, trying to concatenate a string to an integer would cause Python to display an error.

Check if the Player Lost

```
32. if guess != number:
```

Line 32 uses the comparison operator `!=` to check if player's last guess is not equal to the secret number. If this condition evaluates to `True`, the exeution moves into the if-block on line 33.

Lines 33 and 34 are inside the if-block, and only execute if the condition on line 32 was `True`.

```
33.     number = str(number)
34.     print('Nope. The number I was thinking of was ' + number)
```

In this block, the program tells the player what the secret number they failed to guess correctly was. This requires concatenating strings, but number stores an integer value. Line 33 will replace number with a string form so that it can be concatenated to the `'Nope. The number I was thinking of was '` string on line 34.

At this point, the execution has reached the end of the code, and the program terminates. Congratulations! You've just programmed your first real game!

Flow Control Statements

In previous chapters, the program execution started at the top instruction in program and went straight down, executing each instruction in order. But with the `while`, `if`, `else`, and `break` statements, you can cause the execution to loop and skip instructions based on conditions. The name for these kinds of statements is **flow control statement**, since they change the "flow" of the program execution as it moves around your program.

Summary

If someone asked you, **"What exactly is programming anyway?"** what could you say to them? Programming is just the action of writing code for programs, that is, creating programs that can be executed by a computer.

"But what exactly is a program?" When you see someone using a computer program (for example, playing your "Guess the Number" game), all you see is some text appearing on the screen. The program decides what exact text to show on the screen (the program's **output**), based on its instructions and on the text that the player typed on the keyboard (the program's **input**). A **program** is just a collection of instructions that act on the user's input.

"What kind of instructions?" There are only a few different kinds of instructions, really.

1. **Expressions.** Expressions are values connected by operators. Expressions are all evaluated down to a single value, as `2 + 2` evaluates to `4` or `'Hello' + ' ' + 'World'` evaluates to `'Hello World'`. When expressions are next to the if and `while` keywords, you also call them conditions.
2. **Assignment statements.** Assignment statements store values in variables so you can remember the values later in the program.
3. **The `if`, `while`, and `break` flow control statements.** Flow control statements can cause the flow to skip instructions, loop over instructions, or break out of loops. Function calls also change the flow of execution by jumping to the start of a function.
4. **The `print()` and `input()` functions.** These functions display text on the screen and get text from the keyboard. This is called **I/O** (pronounced like the letters, "eye-oh"), because it deals with the Input and Output of the program.

And that's it, just those four things. Of course, there are many details about those four types of instructions. In this book you'll learn about new data types and operators, new flow control statements, and many other functions that come with Python. There are also different types of I/O such as input from the mouse or outputting sound and graphics instead of just text.

For the person using your programs, they only care about that last type, I/O. The user types on the keyboard and then sees things on the screen or hears things from the speakers. But for the computer to figure out what sights to show and what sounds to play, it needs a program, and programs are just a bunch of instructions you, the programmer, have written.