|  |  |
| --- | --- |
|  | Chapter 4 |
| Guess the Number |

Topics Covered In This Chapter:

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

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 can 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 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, we’ll 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!

## Source Code of Guess the Number

Open a new file editor window by clicking on the **File** ► **New Window**. In the blank window that appears, type in the source code and save it as guess.py. Then run the program by pressing **F5**. 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 3.4.2” 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/.

If you get errors after typing this code in, compare the code you typed 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)

## import statements

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 in a variable.

While Python includes many built-in functions, some functions exist in separate programs called modules. You can use these functions by importing their modules into your program with an import statement.

Line 2 imports the module named random so that the program can call random.randint(). This function will come up with a random number for the user to guess.

4. guessesTaken = 0

Line 4 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 at this point in the program, 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 the string argument you passed it on the screen.

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 instructions 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.

The randint() function is provided by the random module, so you must 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 separated by commas that follow the function name. 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 return a different integer. The randint() function returns 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. You’ll use randomness in many games. (Think of how many board games use dice.)

You can also try 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 one string is the 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 be grouped together in a block. Every line in 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’s 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.

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.

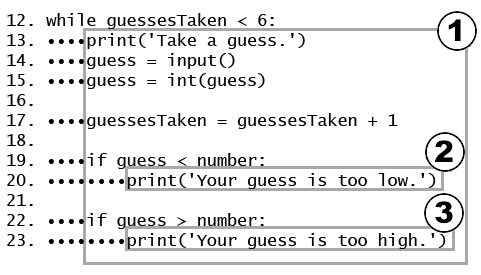


Figure 4-1: Blocks and their indentation. The black dots represent spaces.

Line 22 has only 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 (1). Line 20 is in a block in a block marked as (2). Line 23 is the only line in another block in a block marked as (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 (called bools for short) with comparison operators to form conditions. (Conditions are explained later.)

For example, try storing the Boolean values in variables:

>>> spam = True

>>> eggs = False

The data types that have been introduced so far are integers, floats, strings, and now bools. Every value in Python belongs to one data type.

## 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 < “less than” sign). The < sign is 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 instructions, 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 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

## The Difference Between = and ==

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 never 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 following block, called the while-block. (In the 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 after the condition. Statements that end with a colon expect a new block on the next line.

12. while guessesTaken < 6:

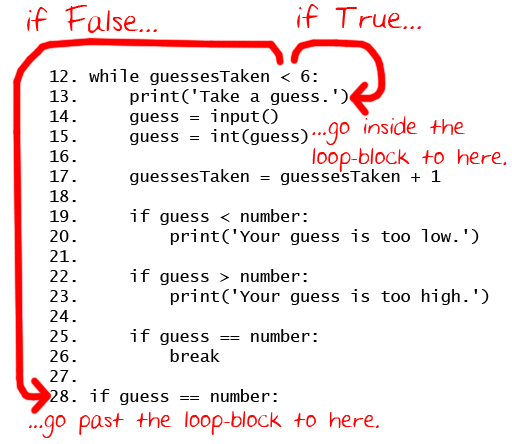


Figure 4-2: The while loop’s condition.

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 executing the code in the following block”.

The Player Guesses

13. print('Take a guess.') # There are four spaces in front of print.

14. guess = input()

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.

## Converting Values with the int(), float(), str(), and bool() Functions

15. guess = int(guess)

Line 15 calls a new function named int(). The int() function takes one argument and returns an integer value form of that argument. Try entering the following into the interactive shell:

>>> int('42')

42

>>> 3 + int('2')

5

The int('42') call will return the integer value 42. 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:

>>> 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'

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 always 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()

14. guess = input()

15. guess = int(guess)

On line 14 the guess variable originally held the string value of what the player typed. Line 15 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.

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)

The float(), str(), and bool() functions will similarly return float, string, and Boolean versions of the arguments passed to them. Try entering the following into the interactive shell:

>>> float('42')

42.0

>>> float(42)

42.0

>>> str(42)

'42'

>>> str(42.0)

'42.0'

>>> str(False)

'False'

>>> bool('')

False

>>> bool('any nonempty string')

True

Using the int(), float(), str(), and bool() functions, you can take a value of one data type and return it as a value of a different data type.

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 called incrementing the variable. Subtracting one from a variable’s integer or float value is called decrementing the variable.

## if statements

19. if guess < number:

20. print('Your guess is too low.') # There are eight spaces in front of print.

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.

Line 19 checks if the player’s guess is less than the computer’s secret number. If so, then the execution moves inside the if-block on line 20 and prints a message telling the player this.

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.

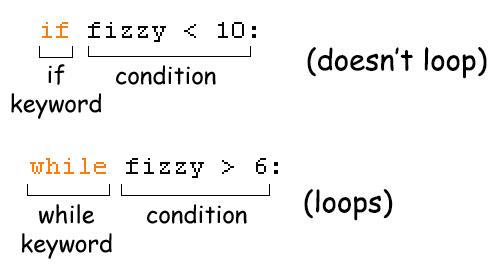


Figure 4-3: if and while statements.

22. if guess > number:

23. print('Your guess is too high.')

Line 22 checks if the player’s guess is greater than the secret number. If this condition is True, then the print() function call tells the player that their guess is too high.

## 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 secret number. 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.

The break statement is only found inside loops, such as in a while-block.

If the player’s guess isn’t equal to the secret number, 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 instruction 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 (guessesTaken < 6) 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, line 28.

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 on line 26 was executed (when the player guesses the number correctly).

Line 28 checks to see if the player guessed correctly. If so, the execution enters 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 str() function, which returns the string form of guessesTaken. Line 30 concatenates strings 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 “not equal to” comparison operator != to check if player’s last guess is not equal to the secret number. If this condition evaluates to True, the execution 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 overwrite 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!

You can change the game’s difficulty by changing the number of guesses the player gets. To give the player only four guesses, change the code on line 12:

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.

## 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** 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 can also call them conditions.
2. **Assignment statements** store values in variables so you can remember the values later in the program.
3. **The** if**,** while**, and** break **statements** are flow control statements that can cause the execution to skip instructions, loop over instructions, or break out of loops. Function calls also change the flow of execution by jumping to the instructions inside 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 that you, the programmer, have written.