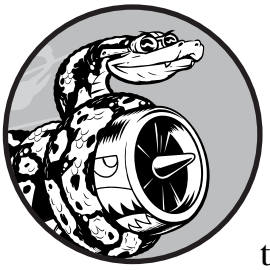
7

USER INPUT AND WHILE LOOPS



Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user. For a simple example, let's say someone wants to find out whether they're old enough to vote. If you write a program to answer this question, you need to know the user's age before you can provide an answer. The program will need to ask the user to enter, or *input*, their age; once the program has this input, it can compare it to the voting age to determine if the user is old enough and then report the result.

In this chapter you'll learn how to accept user input so your program can then work with it. When your program needs a name, you'll be able to prompt the user for a name. When your program needs a list of names, you'll be able to prompt the user for a series of names. To do this, you'll use the `input()` function.

You'll also learn how to keep programs running as long as users want them to, so they can enter as much information as they need to; then, your program can work with that information. You'll use Python's `while` loop to keep programs running as long as certain conditions remain true.

With the ability to work with user input and the ability to control how long your programs run, you'll be able to write fully interactive programs.

How the input() Function Works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.

For example, the following program asks the user to enter some text, then displays that message back to the user:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

The `input()` function takes one argument: the *prompt*, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt `Tell me something, and I will repeat it back to you:` . The program waits while the user enters their response and continues after the user presses ENTER. The response is assigned to the variable `message`, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTE *Sublime Text and many other editors don't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run these programs from a terminal. See "Running Python Programs from a Terminal" on page 12.*

Writing Clear Prompts

Each time you use the `input()` function, you should include a clear, easy-to-follow prompt that tells the user exactly what kind of information you're looking for. Any statement that tells the user what to enter should work. For example:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric
Hello, Eric!
```

Sometimes you'll want to write a prompt that's longer than one line. For example, you might want to tell the user why you're asking for certain input. You can assign your prompt to a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

greeter.py

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

This example shows one way to build a multi-line string. The first line assigns the first part of the message to the variable `prompt`. In the second line, the operator `+=` takes the string that was assigned to `prompt` and adds the new string onto the end.

The prompt now spans two lines, again with space after the question mark for clarity:

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Using int() to Accept Numerical Input

When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

The user enters the number 21, but when we ask Python for the value of `age`, it returns `'21'`, the string representation of the numerical value entered. We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But if you try to use the input as a number, you'll get an error:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
❷ TypeError: unorderable types: str() >= int()
```

When you try to use the input to do a numerical comparison ❶, Python produces an error because it can't compare a string to an integer: the string `'21'` that's assigned to `age` can't be compared to the numerical value `18` ❷.

We can resolve this issue by using the `int()` function, which tells Python to treat the input as a numerical value. The `int()` function converts a string representation of a number to a numerical representation, as shown here:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age = int(age)
>>> age >= 18
True
```

In this example, when we enter `21` at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` ❶. Now Python can run the conditional test: it compares `age` (which now represents the numerical value 21) and `18` to see if `age` is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

The program can compare `height` to `48` because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that they're tall enough:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

When you use numerical input to do calculations and comparisons, be sure to convert the input value to a numerical representation first.

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (`%`), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
```

```
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, `if number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

TRY IT YOURSELF

7-1. Rental Car: Write a program that asks the user what kind of rental car they would like. Print a message about that car, such as "Let me see if I can find you a Subaru."

7-2. Restaurant Seating: Write a program that asks the user how many people are in their dinner group. If the answer is more than eight, print a message saying they'll have to wait for a table. Otherwise, report that their table is ready.

7-3. Multiples of Ten: Ask the user for a number, and then report whether the number is a multiple of 10 or not.

Introducing while Loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or *while*, a certain condition is true.

The while Loop in Action

You can use a `while` loop to count up through a series of numbers. For example, the following `while` loop counts from 1 to 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by assigning `current_number` the value 1. The `while` loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints `1` and then adds 1, making the current number `2`. Because 2 is less than 5, Python prints `2` and adds 1 again, making the current number `3`, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

The programs you use every day most likely contain `while` loops. For example, a game needs a `while` loop to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit. Programs wouldn't be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so `while` loops are quite useful.

Letting the User Choose When to Quit

We can make the *parrot.py* program run as long as the user wants by putting most of the program inside a `while` loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

parrot.py

```
❶ prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
```

```
❷ message = ""
❸ while message != 'quit':
      message = input(prompt)
      print(message)
```

At ❶, we define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, `'quit'`). Then we set up a variable `message` ❷ to keep track of whatever value the user enters. We define `message` as an empty string, `""`, so Python has something to check the first time it reaches the `while` line. The first time the program runs and Python reaches the `while` statement, it needs to compare the value of `message` to `'quit'`, but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give `message` an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the `while` loop work. This `while` loop ❸ runs as long as the value of `message` is not `'quit'`.

The first time through the loop, `message` is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is assigned to `message` and printed; then, Python reevaluates the condition in the `while` statement. As long as the user has not entered the word `'quit'`, the prompt is displayed again and Python waits for more input. When the user finally enters `'quit'`, Python stops executing the `while` loop and the program ends:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

This program works well, except that it prints the word `'quit'` as if it were an actual message. A simple `if` test fixes this:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end. It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one `while` statement becomes complicated and difficult.

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to `True` and stop running when any of several events sets the value of the flag to `False`. As a result, our overall `while` statement needs to check only one condition: whether or not the flag is currently `True`. Then, all our other tests (to see if an event has occurred that should set the flag to `False`) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call `active` (though you can call it anything), will monitor whether or not the program should continue running:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

❶ active = True
❷ while active:
    message = input(prompt)

❸     if message == 'quit':
        active = False
❹     else:
        print(message)
```