2.6 Input and Output

We have already seen Python's **input**(), **eval**() and **print**() functions. In this section we will look at them with a little more care.

Input from the user

Python's **input**() function is very basic. When it is called the system waits for the user to type something and hit the Return key. When this is completed the function takes whatever the user has typed, packages it into a string, and returns it. It does not evaluate what the user has typed in any way. You can use this function with no argument, in which case it just halts and waits for input, or you can give it a string which it interprets as a prompt and prints before it waits for the input:

```
name = input( "Who are you? " )
```

I usually leave one blank space at the end of the prompt string (just before the closing double-quotes) so the user's input doesn't run right up against the prompt; it makes the input dialog easier to read.

The **input**() function will only get you a string. If you want something else, you need to evaluate in some way the string that **input**() returns. The easiest way to do this is with the **eval**() function. **eval**(**input**(<prompt>)) prints the prompt, waits for input, and then tries to interpret the input as a Python value. This is easy and straightforward if the user has entered a number, as in

```
length = eval( input( "Enter the length: " ) )
```

We will usually restrict our use of **eval**() to this situation: interpreting strings as numbers. It is possible to evaluate more elaborate strings. The string "[2, 4, 6, 8]" evaluates to a *list* with 4 elements. For more on lists, see Chapter 6. The string "1, 2, 3" is evaluated as a *tuple* with 3 elements. Tuples are also discussed in Chapter 6. The problem with trying to use **eval**(**input**()) for these is getting the user to type in exactly the right format. The wrong format, such as omitting the commas between the elements, will cause **eval**() to crash on the input string.

One option for inputting compound data is to give the user a format and then use the string method split () to separate it into fields that can be easily evaluated. For example, suppose we want the user to enter a date. We might signal the input with

```
dateString=input("Enter a date in the form m/d/y: ")
```

We can then use the string split () method to divide the input into fields. split () takes one string argument that it regards as a *separator*, and it splits the string into a list of fields determined by this separator. For example, if dateString is "10/29/1929" then dateString.split("/") is the list ["1","29", "1929"]. The individual fields in this are still strings, but we can use the **eval**() function to turn them into numbers. Here is a complete block of code that does this, giving integer values to the variables month, day, and year.

```
dateString=input("Enter a date in the form m/d/y: ")
dateList=dateString.split("/")
month=eval(dateList[0])
day=eval(dateList[1])
year=eval(dateList[2])
```

Printing

The **print**() function is used to make output appear on the *console*, the computer screen. The simple statement

```
print(x)
```

causes the value of x to be printed. You can print multiple values with

```
print(x, y, z)
```

By default a single space is printed between each of the values. We will see below how to change this default. The **print**() function also terminates its line of output, so the code

```
print("Marvin Krislov")
print("Oberlin College")
```

results in

```
Marvin Krislov
Oberlin College
```

Again, we will see how to change this default. The full **print**() function is

```
print( <values>, sep=<separator>, end=<terminator> )
```

with all three fields optional. Here <values> is either a single value or a comma-separated list of values, such as 23, 45. If the values are omitted, only the end-string is printed. <separator> is any string we want to appear between the values. For example,

```
print( 23, 45, 58, sep="***")
```

will print

```
23***45***58
```

If there is only one value the separator is not printed at all. By default the separator is a string consisting of one single space character: " ".

It is often convenient to build up a line of output through several calls to **print**(). The end-string, or terminator, defaults to the newline character, which in Python is written "\n". This is what causes **print**() to end the current line and prepare to start a new line. If we change the end-string to something else we stay on the same line after printing, so the next call to **print**() appears on the same line. If we want to do this the most common value to use for the end-string is the empty string, or "". For example, the following code

```
print("Marvin Krislov", end="")
print("Oberlin College")
```

will print

```
Marvin KrislovOberlin College
```

Here is a simple example of how we might use these techniques to print a table in nice orderly columns. The following code is a little redundant; we will be able to do this much more simply after we see *loops* in Chapter 3.

```
print( "%5d" % 123, end="" )
print( "%5d" % 5, end="" )
print( "%5d" % 17, end="" )
print( "%5d" % 444, end="" )
print()
print( "%5d" % 11, end="" )
print( "%5d" % 9, end="" )
print( "%5d" % 123, end="" )
print( "%5d" % 3, end="" )
print()
print( "%5d" % 422, end="" )
print( "%5d" % 73, end="" )
print( "%5d" % 12, end="" )
print( "%5d" % 9, end="" )
print()
```

This prints

```
123    5   17  444
 11    9  123    3
422   73   12    9
```