2.3 Variables and types

A paragraph in English is a sequence of words written according to the grammar rules of English so that it is meaningful. The words of the paragraph are either standard words of English, which you can find in a dictionary, or names of objects, which can be almost anything, such as "Caerdydd" or "2Pac". The situation in programming is very similar. A program is a sequence of words and symbols written according to the grammar rules of a programming language (in our case the rules of Python) so that it can be executed on a computer. The words of the program are either the keywords of the language, such as "def", "if", and "else", or they are the names of objects that are constructed in the program, or they are textual data.

Program 2.1.1 showed two ways that a program can make named objects. The line

```
def main():
```

starts the definition of a function called main(). We could name this function anything else. For example, in the following program the main function computes the area of a rectangle, so it is called area() rather than main().

```
# This asks for the dimensions of
# a rectangle and computes its area.

def area():
    print( "This finds the area of a rectangle." )
    L = eval( input( "Enter the length: " ) )
    W = eval( input( "Enter the width: ") )
    print( "The area is ", L*W )

area()
```

Program 2.3.1: The top-level function is called area()

Although the Python language allows this, as we said in Section 2.1 our convention is to always name the top-level function of our programs main().

Definitions are one way to assign a name to an object. Program 2.3.1 illustrates the other common way to make a named object. The line

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

creates a variable called L, and gets its initial value from input from the program's user. In any programming language a *variable* is a name attached to a location in memory where we can store data. In many languages, including both C and Java, variables must be *declared*, or registered with the program before they can be used. When a variable is declared the programmer must say what kind of data it will hold so that the right amount of memory can be assigned to it. Python does things differently. In Python a variable is created when we attach a value to a name; no declaration is necessary. Furthermore, we can change the kind of data attached to a variable at any point in the program; the system determines from usage what kind of data a variable holds and adjusts its internal memory accordingly. This is very convenient, but it is not without its problems, which we will discuss below. For now, just focus on the fact that we create a variable by assigning a value to a name.

Finally, note that in Program 2.1.1 we used the **input** function by itself:

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

while in Program 2.3.1 we also invoke the **eval** function:

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

The **input** function always returns a string. If this is what we want, for example if we expect the user to enter a name, we don't need to do anything to it; we just pass this string on to whatever variable it is being assigned to. On the other hand, if we expect the user to enter a number we need to translate the string returned by **input** into a numerical representation. That is the role of the **eval** function – it takes a string and evaluates it into its natural Python representation. For more on this see section 2.6.

Assignments

The = symbol is used for assigning a value to a variable. The variable receiving a value is always on the left side of this symbol, and on the right side is an expression of the value being assigned. In Program 2.3.1 there are two assignment statements:

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

and

```
W = eval( input( "Enter the width: ") )
```

The first of these creates variable L and reads a value from the user to put into L. The right side of an assignment statement can be anything that computes to a value. For example,

```
Z1 = 23*5
```

and

```
Z2 = 1 + eval(input( "Enter a number: " ))
```

are both valid statements. The first of these puts the number 115 into variable Z1; the second reads a number from the user, adds 1 to this value, and stores the result in variable Z2. In Program 2.3.1 we could introduce a variable that holds the area of the rectangle as follows:

```
L = eval( input( "Enter the length: " ) )
W = eval( input( "Enter the width: ") )
A = L*W
print( A )
```

Sometimes we need to increase the value of a variable. For example, in a predator-prey simulation we might have a variable foxCount that represents the current number of foxes. If a new fox is born, we need to add one to this count. Here is a line of code that does this:

```
foxCount = foxCount + 1
```

When assignment statements are evaluated, the right side is always computed first, and that value is given to the variable on the left side. This statement takes the current value of variable foxCount, adds 1 to it, and that result becomes the new value of foxCount. If the old value of foxCount was 97, after this statement is excuted its new value will be 98.

There is a shorthand notation for updating the values of variables. The statement x += a is short for x = x + a: the value of a is added to variable x. Similar definitions apply for −=, *= and /=, though += is by far the most commonly used of these notations.

Names

Every program you write will have variables and functions, and you will need to come up with names for these objects. Python doesn't care what names you use: as far as the Python system is concerned pq23x is as good a name as FoxCount. For human readers names make a difference. You are more likely to write programs that work if you use good names. There are many different naming conventions, and unless you are working in a group programming environment where everyone needs to use the same conventions, it doesn't really matter which conventions you use. Here are some guidelines that will help you write successful programs:

- Names are composed of letters, digits, and the underscore character. All names in Python must start with a letter. 1x is not a legal name, though x1 is.
- Use names that have some meaning for your program. You could use x to represent the width of a rectangle, but W, width, and rectangleWidth are all better choices.
- Use names composed of English words. In a program that deals with rectangles and circles, you could use A1 to represent the area of a rectangle and A2 to represent the area of a circle, but is is easy to forget which is which. A better choice is to use rectangleArea and circleArea . When you want to use more than one English word in a name, push the words together and capitalize the first letter of each word (or the first letter

of each word after the first). Another option is to use underscores to separate words, as in rectangle_area . I tend to use capitalization rather than underscores because I find capital letters easier to type, but you might make a different choice.

- Avoid cute names. It might seem fun to give variables the names of your friends, but a few days later you are likely to forget which name goes with which object. If you use descriptive names you are more likely to finish your programs early, and then you can spend quality time with your friends.
- Be consistent with capitalization. Some of the most common bugs are introduced by capitalizing a variable one way in one part of a program and a different way in another part of the program, and so creating two different variables. Some people use a convention that function names all start with a capital letter and variables names start with a lowercase letter. With this convention prime might be a variable that represents a prime number while Prime() is a function concerned with prime numbers. You arent likely to confuse a variable with a function (though it occasionally happens); the real point of this convention is that if you have a variable that represents a part number in an inventory program you know it will be spelled partNumber and not PartNumber.

Example 1. As an illustration of variables and assignment statements, here is a common situation. We have variables x and y that have values. We want to interchange these values give x the value currently in y, and y the value currently in x. The following code doesnt work:

```
x = y
y = x
```

The statement x = y results in the old value of x being lost; after this x and y have the same value. For example, if we start with x having the value 23 and y having 45, then after x = y both will have the value 45. The statement y = x then has no effect; after it both variables will still have the value 45. There is no way to do this with just two variables; to make it work we need to introduce a new variable to hold one of the values for us. Here is code that does work:

```
oldX = x
x = y
y = oldX
```

The variable oldX holds onto the original value of x, so this value is still around after we make the assignment x = y. We can then use this variable to give the right value to y. Note that order matters: the following code has the same three assignment statements, but it doesnt work:

```
x = y
oldX = x
y = oldX
```

Try an example, such as starting values 23 for x and 45 for y, and convince yourself that this code doesnt interchange the values of x and y.

Types

Computers can manipulate many different kinds of data. In some problems the essential data is numeric; other problems need textual information and still others need complex data, such as the position and orientation of planets in their orbits around the sun. Different kinds of data allow for different operations: we can multiply two integers, but it doesn't make sense to multiply two English words. A *type* is a collection of values together with the basic operations that can be performed on these values. For example, the Integer type has the values 0, 1, -1, 2, -2, 3, -3, and so forth, together with operations like addition, subtraction, multiplication and division. You need to be aware of the types of the data you have stored in your variables, and to use the variables consistently according to these types.

Python has several basic types of data, together with some ways to build more complex types from these primitive types. The four most common elementary types are:

- *Integers.* These have whole numbers as values and a wide variety of arithmetic operators for manipulating them.
- *Floating Point numbers*, sometimes called "floats". These have decimal values: 3.14159, -3.5, or 6.0. Again, there is a wide variety of arithmetic operators available for working with floats. Note that the floating point number 6.0 is different from the integer number 6 because they are elements of different types. They are also represented differently in the computer's memory; using one when you mean the other can get you into trouble.
- *Strings.* These are sequences of individual characters, such as "Scooby Doo" or 'Plan 9 From Outer Space'. We write strings inside quotation marks (either single quotes or double quotes). Strings also have their own operations, which we will discuss later.
- *Booleans.* There are only two Boolean values: True and False. Note that they are written with capital initial letters: true is not a Boolean value. They are also not strings: True and "True" are completely different objects. You might not see at the moment why Booleans are useful, but starting with Chapter 3 we will make frequent use of them. By the way, the unusual name "Boolean" is a tribute to George Boole, a 19th Century English mathematician who was one of the original developers of symbolic logic.

The next two sections discuss numeric types, then strings. In Chapter 3 we will discuss the Boolean type. There is also a Complex Number type built into Python, but we will not make use of it in these notes.