**Chapter 2 Building Blocks for Programs**

We have seen that Python can be used interactively. It is also possible to gather statements into a file, which is called a ***script***. The script can then be run, or executed, all at once rather than one statement or one cell at a time. We will also use the word ***program*** to refer to code that is stored in a script. This chapter will introduce coding constructs that are frequently used in programs.

**2.1 Executing Code**

Lines of code can be executed individually, in code cells, and in scripts.

We have seen that using one of the interpreters, code can be entered at the prompt and executed immediately; the results are shown after the input.

Using an environment such as Jupyter notebooks, multiple lines of code can be entered in one cell, and then the contents of the cell can be executed. This causes each line to be executed sequentially (for now, anyway!) and all of the results are shown.

For longer programs, it is frequently useful to use an editor to type in and save a script in a file. By convention, the file will have an extension of .py.

In the remainder of this book, short expressions and simple codes will be shown using the >>> prompt. Longer code, which may be entered as a script, will be shown in boxes.

For the most part, longer codes could potentially be entered either one line at a time, in code cells, or in scripts. However, there is one difference between scripts and interactive modes. Results of expressions are shown only in interactive modes, not in scripts. In scripts, expressions must be explicitly printed in order for the results to be seen.

**2.2 Comments**

***Comments*** are completely ignored by Python when code is executed. Comments are used by the coder to explain what the code is doing. Comments are everything from the # symbol to the end of a line. For example:

*>>> age = 33 # the age of the person*

Longer comments that cover several lines can be used to describe what a program, or a part of a program, is accomplishing. For example,

# This program will calculate the area of a circle. # The program will print the area in a nice sentence format.

Longer comments can also be created using ***docstrings***, which are strings contained in triple quotes. These are used to document user-defined functions, which will be described in Chapter 6.

**2.3 Input**

Rather than building values into a program by assigning them, it is frequently useful to ask the user for values. In Python, this is accomplished using the **input** function. In order for the user (the person running the program) to know what they are supposed to be entering, the code must ***prompt*** the user. The **input** function contains the prompt. The user’s entry should then be stored in a variable, which will always be a string. For example, assume that when prompted the user enters ‘It is now 4pm’ (without the quotes):

*>>> sometext = input('Please enter something: ')*

*>>> print(sometext)*

This would appear as:

Please enter something: It is now 4pm

It is now 4pm

The user entered ‘It is now 4pm’ (again, without the quotes) and hit the Return key, which sent the entered string to be the value of the variable *sometext*.

Since the **input** function always returns a string, if a number is required the string must be converted to the appropriate number type. For example, this code

*>>> age = input('How old are you? ')*

*>>> numage = int(age)*

*>>> print('This time next year you will be', numage+1)* would result in the following if the user enters 32:

How old are you? 32

This time next year you will be 33

The age ‘32’, returned as a string, was cast to the type **int**.

It will be easiest for now to have a separate **input** statement for every value that the user needs to enter. Although it is possible to prompt for more than one value in a single **input** statement, it will all be in one string variable. Text processing would be necessary to break the string into the various parts, and to convert strings containing numbers to number types. This will be demonstrated later.

**2.4 Formatting Output**

We have seen that the **print** function can be used to display combinations of text and numbers. Some special characters can be used within the strings. The ‘\n’ character, called the ***newline character***, moves the cursor down to the next line when printed.

| *print('See what \nthis does')*  *print('OK?')* |
| --- |

See what

this does

OK?

Explicitly printing the newline character immediately moves down to the next line.

The **print** function will automatically print a newline character at the end, so in this example the second **print** function started printing on the third line.

Without passing any arguments, **print()** will simply print a newline character. In the following example, doing so prints a blank line.

| print('\*\*\*\*')  print()  print('####') |
| --- |

\*\*\*\*

####

It is also possible to end the result from a **print** statement with something other than a newline character by specifying the **end** keyword.

| *print('See what \nthis does', end = ' ') print('OK?')* |
| --- |

See what

this does OK?

In this case, the first **print** statement ended with a space rather than a newline, so the second **print** began on that same line.

To print a single quote within a string that is specified using single quotes, use \’.

*>>> print('Isn\'t this grand')*

Isn't this grand

Of course, if the string is created using double quotes, that is not necessary.

*>>> print("Isn't necessary here!")*

*Isn't necessary here!*

To print a double quote within a string that is specified using double quotes, use \”.

*>>> print("The character \"a\" is a short string")* The character "a" is a short string

In order to format the output, ***f-strings*** (short for ***formatted strings***) can be used. These are created by putting the letter f (or F) in front of the string. By putting expressions in curly braces, the values of these expressions are filled in when the result is printed. For example,

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint}xx')*

The integer is xx1234xx

When printed, the curly braces are not shown but the value of the variable *myint* is displayed in that location. The xx’s are printed just to show the where the number begins and ends. Note that if we did not use the f-string, and instead just printed the text and number separately, the output would automatically have blank spaces:

*>>> print('The integer is xx', myint, 'xx')*

The integer is xx 1234 xx

So, the f-string allows us to control spacing. Inside the curly braces, after the name of the variable, we can add a colon and then ***format specifiers***. For example, ‘8d’ says to print an integer (‘d’ for integer!) in a field width of 8, meaning 8 characters altogether.

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint:8d}xx')*

The integer is xx 1234xx

Notice that by default the number is right-justified within the field width of 8, so there are 4 blank spaces and then the 4-digit number. In order to left-justify the number, the less than sign is used.

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint:<8d}xx')*

The integer is xx1234 xx

Similarly, strings can be printed within a specified field width using ‘s’ in the format specifier (‘s’ for string). Strings are left-justified. For example:

*>>> word = 'Hello'*

*>>> print(f'The word is xx{word:7s}xx')*

The word is xxHello xx

Strings can be right-justified using the greater than sign:

*>>> word = 'Hello'*

*>>> print(f'The word is xx{word:>7s}xx')*

The word is xx Helloxx

We will see in a later chapter that there is also a method that will right-justify a string.

For floats, there are more options for the format specifier. Using ‘f’ in the format specifier (‘f’ for float), the number of decimal places can be controlled. For example, ‘8.2f’ specifies a field width of 8 including 2 decimal places, and including the decimal point.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:8.2f}xx')*

The float is xx 12.35xx

Notice that the .345 was rounded to 2 decimal places, .35. A total of 8 characters was printed: 3 blank spaces, then 12.35 which is 5 characters. The less than sign can be used to left-justify floats. Also, it is not necessary to specify the field width. Just using ‘.2f’ as the format specifier, 2 decimal places are printed and the field width is adjusted according to the number being printed.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:.2f}xx')*

The float is xx12.35xx

This is usually preferable to specifying a field width, since it is more general.

For floats, two other specifiers besides ‘f’ are: ‘g’ which specifies a general format, and ‘e’ which specifies scientific notation.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:f}xx')*

The float is xx12.345000xx

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:g}xx')*

The float is xx12.345xx

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:e}xx')*

The float is xx1.234500e+01xx

For any type, putting an equal sign in the curly braces after the name of the variable shows both the variable name and its value.

*>>> myint = 32 + 1*

*>>> print(f'{myint = }')*

myint = 33

The f-strings used here have all been in calls to the **print** function. However, f-strings are just strings that are formatted, and can be stored in variables.

*>>> dolamt = f'${123.456:.2f}'*

*>>> dolamt*

'$123.46'

**2.5 Scripts with Input and Output**

An ***algorithm*** is a sequence of steps needed in order to solve a problem. When coding, it is frequently useful to write out an algorithm first. Once that has been accomplished, the algorithm is then translated into code.

A basic algorithm for many programs is:

• Get the necessary inputs

• Calculate results

• Print or display the results

A script that accomplishes this can therefore be broken down into three basic parts, to implement each of these steps.

For example, let’s write a program that will calculate the area of a rectangle. What would the input(s) be? For a rectangle, we would need to know the length and the width. In order for the user to know that they are to enter the length and the width, they need to be prompted to do

so. In this case, the user would need to know the units of measure. This might be accomplished by printing instructions before prompting. We also need to know that the area of a rectangle is the product of the length and width. Once the result is obtained, it is good practice to print the result in a nicely formatted sentence. In order to be very informative, it would be appropriate

to print not just the area, but the values used to calculate the area.

So, our algorithm would be:

• Print instructions and include the units

• Prompt the user for the inputs:

o Prompt the user and read in the length

o Prompt the user and read in the width

• Calculate the area as length \* width

• Print the area in a formatted sentence

Now let’s implement that algorithm in a Python script.

| # This script calculates the area of a rectangle  # Print instructions and prompt the user for the length and width  print('When prompted, please enter the length and width') print(' of a rectangle in units of meters.')  st\_length = input('Please enter the length: ')  rec\_length = float(st\_length)  st\_width = input('Please enter the width: ')  rec\_width = float(st\_width)  # Calculate the area  rec\_area = rec\_length \* rec\_width  # Print the results  print(f'For a rectangle with a length of {rec\_length:.1f} meters') print(f' and a width of {rec\_width:.1f} meters, the area') print(f' is {rec\_area:.3f} meters squared.') |
| --- |

When prompted, please enter the length and width

of a rectangle in units of meters.

Please enter the length: 3.32

Please enter the width: 4.1

For a rectangle with a length of 3.3 meters

and a width of 4.1 meters, the area

is 13.612 meters squared.

**2.6 Debugging**

We all make mistakes! ***Debugging*** is finding and fixing mistakes in code. There are several basic types of mistakes:

• ***Syntax errors***: These are mistakes such as leaving off the rightmost quote or brackets, and are flagged by Python.

• ***Logic errors***: Logic errors are mistakes in reasoning. The code is correct Python code; it executes properly but the results are not correct. An example might be multiplying instead of dividing when making a calculation. Sometimes it is difficult to even realize that a logic error has been made, since no error messages will result.

• ***Execution-time errors***: These are errors that are only found when the code is executed. For example, the code might divide a number by a variable, but the variable stores a zero.

Python error messages are usually easy to understand. For example,

| x = 4  y = float(input('Enter a number: ')) x/y |
| --- |

Enter a number: 0

--------------------------------------------------------------------------- ZeroDivisionError Traceback (most recent call last) /var/folders/j9/v92c26p10979jkyb81f\_nr\_c0000gn/T/ipykernel\_83392/1454302463.p y in <module>

1 x = 4

2 y = float(input('Enter a number: '))

----> 3 x/y

ZeroDivisionError: float division by zero

The name of the error message is “ZeroDivisionError”, which is self-explanatory. The line of code in which this occurred is also highlighted with an arrow.

For cases in which it is not obvious what went wrong, a good tactic is to print the values of variables.