Chapter 1

Computer Systems and Software

Programmable software is what makes a computer a powerful tool. Each different program essentially "rewires" the computer to allow it to perform a different task. In this course, you will learn basic principles of writing software in the Python programming language.

Python is a popular scripting language available as a free download from `http://www.python.org/`. Follow the instructions given there to install the latest production version of Python 3 on your system. The examples in this text were written with Python 3.2.

The CPU and RAM

In order to write software, it will be helpful to be able to imagine what happens inside a computer when your program runs. We begin with a rough picture and gradually fill in details along the way.

When a program is ready to run, it is loaded into RAM, usually from long-term storage such as a hard drive. **RAM** is an acronym for random access memory, which is the working **memory** of a computer. RAM is **volatile**, meaning that it requires electricity to keep its contents.

Once a program is loaded into RAM, the **CPU**, or central processing unit, executes the instructions of the program, one at a time. Each type of CPU has its own **instruction set**, and you might be surprised at how limited these instruction sets are. Most instructions boil down to a few simple types: load data, perform arithmetic, and store data. What is amazing is that these small steps can be combined in different ways to build programs that are enormously complex, such as games, spreadsheets, and physics simulations.

Computer Languages

CPU instruction sets are also known as **machine languages**. The key point to remember about machine languages is that in order to be run by a CPU, a program *must* be written in the machine language of that CPU. Unfortunately, machine languages are not meant to be read or written by humans. They are really just specific sequences of bits in memory. (We will explain bits later if you do not know what they are.)

Because of this, people usually write software in a **higher-level language**:

Level	Language	Purposes
Higher	Python	Scripts
	Java	Applications
	C, C++	Applications, Systems
	Assembly Languages	Specialized Tasks
Lower	Machine Languages	

This table is not meant to be precise, but, for example, most programmers would agree that C and C++ are closer to the machine than Python.

Compilation and Interpretation

Now if CPUs can only run programs written in their own machine language, how do we run programs written in Python, Java, or C++? The answer is that the program must be translated into machine language first.

There are two main types of translation: compilation and interpretation. When a program is **compiled**, it is completely translated into machine language, producing an executable file. C and C++ programs are usually compiled, and when they are compiled in Microsoft Windows®, for example, the executable files generally end in `.exe`.

On the other hand, when a program is **interpreted**, it is translated "on-the-fly." No separate executable file is created. Instead, the translator program (the **interpreter**) is running and it translates your program so that the CPU can execute it. Python programs are usually interpreted.

The Python Interpreter

When you start Python, you are in immediate contact with a Python interpreter. If you provide it with legitimate Python, the interpreter will translate your code so that it can be executed by the CPU. The interpreter displays the version of Python that it will interpret and then shows that it is ready for your input with this prompt:

```
>>>
```

The interpreter will translate and execute any legal Python code that is typed at the prompt. For example, if we enter the following statement:

```
>>> print("Hello!")
```

the interpreter will respond accordingly. Try it and see.

Remember that as you learn new Python constructs, you can always try them out in the interpreter without having to write a complete program. Experiment—the interpreter will not mind.

A Python Program

Still, our focus will be on writing complete Python programs. Here is a short example:

Listing 1.1: Area of a Circle

```
from math import pi
r = 12
area = pi * r ** 2
print("The area of a circle with radius", r, "is", area)
```

Even if you have never seen Python before, you can probably figure out what this program does.

Start the Python **IDLE** application and choose "New Window" from the File menu. Type Listing 1.1 into the new window that appears. Save it as `circle.py`, and then either choose "Run Module" from the Run menu or press F5 to run the program. This program illustrates many important Python concepts, including variables, numeric expressions, assignment, output, and using the standard library. We will examine these components in the next chapter.

Why Computer Science?

Here are a few things to consider as we begin:

1. Software is everywhere. If you are skeptical, search for "weather forecast toaster."

2. Similarly, computation is having a significant impact on the way other disciplines do their work. And for almost every field X, there is a new interdisciplinary field "Computational X."

3. Programming develops your ability to solve problems. Because machine languages are so simplistic, you have to tell the computer *everything* it needs to do in order to solve a problem. Furthermore, running a program provides concrete feedback on whether or not your solution is correct.

4. Computer science develops your ability to understand systems. Software systems are among the most complicated artifacts ever created by humans, and learning to manage complexity in a program will help you learn to manage it in other areas.

5. Programming languages are tools for creation: they let you build cool things. There is nothing quite like getting an idea for a program and seeing it come to life. And then showing it to all your friends.

Exercises

1.1 Experiment with Listing 1.1 by making changes to various parts of the code. Be sure to try some things that break the program (cause errors), just to see how the interpreter reacts.

1.2 Modify Listing 1.1 to also compute and display the circumference of the circle.

1.3 The December 1978 issue of the IEEE Computer Society journal *Computer* contained the following description of a new computer that fit "on the top of any business desk":

> The PCC 2000 consists of a 3MHz 8085A microprocessor, two 32K memory boards, two FD514 double-density, 8.5-inch floppy disk drives, a 12-inch upper/lower case video display...

(a) Compare the PCC 2000's CPU speed and amount of RAM to current desktops.

(b) Does the PCC 2000 appear to have a hard disk? If not, what does it use for long-term storage?

(c) Research the capacity of one of these floppy disks.

(d) List possible reasons that the PCC 2000 has two floppy disk drives instead of one.

Chapter 2

Python Program Components

Let's look at another Python program that creates short nonsense sentences:

Listing 2.1: Mad Lib

```
# madlib.py
# Your Name

adjective = input("Enter an adjective: ")
noun = input("Enter a noun: ")
verb = input("Enter a verb: ")
adverb = input("Enter an adverb: ")
print("A", adjective, noun, "should never", verb, adverb)
```

Type this program into your Python environment, save it as `madlib.py`, and run it a few times to play with it. Enter any responses you like when the interpreter asks for them. Then consider the following aspects of both this program and Listing 1.1.

Variables

Every program accomplishes its work by manipulating data. **Variables** are names that refer to data in memory. Variable names, also known as **identifiers** in Python, may consist of upper and lower alphabetic characters, the underscore (_), and, except for the first character, the digits 0–9. There is a small set of reserved Python **keywords** that may not be used as variable names; otherwise, you are free to choose any names you wish. You should already be able to see how meaningful identifiers can make a program easier to follow. The variables in Listing 2.1 are `adjective`, `noun`, `verb`, and `adverb`.

Program Statements and Syntax

A program is a sequence of **statements**, which are individual commands executed one after another by the Python interpreter. Every statement must have the correct syntax; the **syntax** of any language is the precise form that it is written in. Thus, if a statement does not have the correct form, the Python

interpreter will respond with a **syntax error**. Three types of statements are used in Listings 1.1 and 2.1:

Assignment statements are used to give variables a value. The syntax of an assignment statement is:

```
<variable> = <expression>
```

Read assignment statements from right to left:

1. Evaluate the expression on the right.
2. Assign the variable on the left to refer to that value.

For example, the assignment

```
x = 10 - 17 + 5
```

computes the right side to be -2 and then assigns `x` to refer to -2.

$\Longrightarrow$ Caution: An assignment statement "=" is not like a mathematical equals sign. Technically, it is a **delimiter** because it helps the interpreter delimit between the variable on the left and the expression on the right.

Print statements are used to produce program output. The syntax of a `print()` statement is:

```
print(<expression1>, <expression2>, ...)
```

Expressions inside quotation marks (either single or double) are printed literally, whereas expressions outside quotation marks have their value printed. Expressions are separated by single spaces in the output, and each print statement produces output on a separate line.

Technically, `print()` is a **built-in function** that we call in order to print, but it is generally called as its own separate statement.

Import statements are used to access library functions or data, as in the first line of Listing 1.1. The **Python Standard Library** consists of many **modules**, each of which adds a specific set of additional functionality. The statement:

```
from <module> import <name1>, <name2>, ...
```

allows the listed names to be used in your program.

The `math` module includes the constant `pi` and functions such as `sin`, `cos`, `log` and `exp`. Both **`import`** and **`from`** are reserved Python keywords and so may not be used as the names of variables. Keywords appear in bold in program listings.

Data Types

Listings 1.1 and 2.1 use three different types of data:

Strings are sequences of characters inside single or double quotation marks.

Integers are whole number values such as 847, −19, or 7.

Floats (short for "floating point") are values that use a decimal point and therefore may have a fractional part. For example, 3.14159, −23.8, and even 7.0 (because it has a decimal point) are all considered floats.

A variable in Python may refer to any type of data.

Expressions

Recall that the right-hand side of every assignment statement must be an **expression**, which in Python is just something that can be evaluated to give a value.

Input expressions are used to ask the user of a program for information. They almost always appear on the right side of an assignment statement; using an `input` expression allows the variable on the left side to refer to different values each time the program is run.

`input(prompt)`	Display `prompt` and return user input as a string.

The **prompt** is printed to alert the user that the program is waiting for input.

Numeric expressions use the **arithmetic operations** `+`, `-`, `*`, `/`, `//`, and `**` for addition, subtraction, multiplication, division, integer division, and raising to a power. These follow the normal rules of arithmetic and **operator precedence**: exponentiation is done first, then multiplication and division (left to right), and finally addition and subtraction (also left to right). Thus, `1 + 2 * 3` evaluates to 7 because multiplication is done before addition. Parentheses may be used to change the order of operations.

Integer division rounds down to the nearest integer, so `7//2` equals 3 and `-1//3` equals −1. Integer division applied to floats rounds down to the nearest integer, but the result is still of type float.

Finally, variables must be assigned a value before being used in an expression.

Comments

Comments begin with a pound sign `#` and signify that whatever follows is to be completely ignored by the interpreter.

```
# Text that helps explain your program to others
```

Comments may appear anywhere in a Python program and are meant for human readers rather than the interpreter, in order to explain some aspect of the code.

Recap

There was a lot of terminology here, so let's summarize what is most important:

1. Variables refer to data...
2. ...via assignment statements. Remember to read them from right to left.
3. Data can be of type string, integer, or float (so far). The data type of a variable determines what you can do with it.

Exercises

2.1 Give three names that are not legal Python identifiers.

2.2 Use Listing 1.1 to:

(a) List each variable and give the type of data stored in it. Hint: `pi` is a variable.

(b) Identify the assignment statements, and explain the effect of each.

2.3 Determine the output of Listing 1.1 if the `print()` statement is changed to:

```
print("The area of a circle with radius r is area")
```

Explain the result.

2.4 Use Listing 2.1 to:

(a) List each assignment statement and identify the variable whose content changes in each.

(b) Describe what happens if the final space in any of the `input` statements is deleted.

2.5 Determine the value of each of these Python expressions:

(a) `1 + 2 * 3 - 4 * 5 + 6`

(b) `1 + 2 ** 3 * 4 - 5`

(c) `1 / 2 - 3 / 4`

(d) `1 // 2 - 3 // 4 + 5 // 6`

(e) `1 + 4 - 2 / 2`

(f) `(1 + 4 - 2) / 2`

2.6 Determine the output of each of these code fragments:

(a)
```
x = 10
y = 15
print(x, y)
y = x
print(x, y)
x = 5
print(x, y)
```

(b)
```
x = 10
y = 15
print(x, y)
y = x
x = y
print(x, y)
x = 5
print(x, y)
```

(c)
```
x = 10
y = 15
z = 20
x = z
z = y
y = x
print(x, y, z)
```

2.7 Modify Listing 2.1 to create your own Mad Lib program. You may want to look ahead to Chapter 8 to use the `sep=` or `end=` options of the `print()` statement to better control your output.

2.8 Explain the difficulty with using `input()` to get numeric values at this stage. (We will address it soon.)

2.9 Modify Listing 1.1 to write a program `average.py` that calculates and prints the average of two numbers.

2.10 Modify Listing 1.1 to write a program `rect.py` that calculates and prints the area and perimeter of a rectangle given its length and width. Run your program with several different values of the length and width to test it.

2.11 Modify Listing 1.1 to write a program `cube.py` that calculates and prints the volume and surface area of a cube given its width. Run your program with different values of the width to find the point at which volume equals surface area.

2.12 Modify Listing 1.1 to write a program `sphere.py` that calculates and prints the volume and surface area of a sphere given its radius. Look up the formulae if you do not remember them. Run your program with different values of the radius to find the point at which volume equals surface area.