CHAPTER 1

The Way of the Program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The Way of the Program".

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

What Is a Program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

input:
: Get data from the keyboard, a file, the network, or some other device.

output:
: Display data on the screen, save it in a file, send it over the network, etc.

math:
: Perform basic mathematical operations like addition and multiplication.

conditional execution:
: Check for certain conditions and run the appropriate code.

repetition:
: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Running Python

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

To avoid that problem, I recommend that you start out running Python in a browser. Later, when you are comfortable with Python, I'll make suggestions for installing Python on your computer.

There are a number of web pages you can use to run Python. If you already have a favorite, go ahead and use it. Otherwise I recommend PythonAnywhere. I provide detailed instructions for getting started at *http://tinyurl.com/thinkpython2e*.

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but I include some notes about Python 2.

The Python **interpreter** is a program that reads and executes Python code. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `python` on a command line. When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is `3.4.0` in this example, begins with 3, which indicates that you are running Python 3. If it begins with 2, you are running (you guessed it) Python 2.

The last line is a **prompt** that indicates that the interpreter is ready for you to enter code. If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 1 + 1
2
```

Now you're ready to get started. From here on, I assume that you know how to start the Python interpreter and run code.

The First Program

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words "Hello, World!" In Python, it looks like this:

```
>>> print('Hello, World!')
```

This is an example of a **print statement**, although it doesn't actually print anything on paper. It displays a result on the screen. In this case, the result is the words

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

The parentheses indicate that `print` is a function. We'll get to functions in Chapter 3.

In Python 2, the print statement is slightly different; it is not a function, so it doesn't use parentheses.

```
>>> print 'Hello, World!'
```

This distinction will make more sense soon, but that's enough to get started.

Arithmetic Operators

After "Hello, World", the next step is arithmetic. Python provides **operators**, which are special symbols that represent computations like addition and multiplication.

The operators `+`, `-`, and `*` perform addition, subtraction, and multiplication, as in the following examples:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

The operator `/` performs division:

```
>>> 84 / 2
42.0
```

You might wonder why the result is `42.0` instead of `42`. I'll explain in the next section.

Finally, the operator `**` performs exponentiation; that is, it raises a number to a power:

```
>>> 6**2 + 6
42
```

In some other languages, `^` is used for exponentiation, but in Python it is a bitwise operator called XOR. If you are not familiar with bitwise operators, the result will surprise you:

```
>>> 6 ^ 2
4
```

I won't cover bitwise operators in this book, but you can read about them at *http://wiki.python.org/moin/BitwiseOperators*.

Values and Types

A **value** is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `2`, `42.0`, and `'Hello, World!'`

These values belong to different **types**: `2` is an **integer**, `42.0` is a **floating-point number**, and `'Hello, World!'` is a **string**, so-called because the letters it contains are strung together.

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

In these results, the word "class" is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type `int`, strings belong to `str`, and floating-point numbers belong to `float`.

What about values like `'2'` and `'42.0'`? They look like numbers, but they are in quotation marks like strings:

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is not a legal *integer* in Python, but it is legal:

```
>>> 1,000,000
(1, 0, 0)
```

That's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

Formal and Natural Languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

Formal languages tend to have strict **syntax** rules that govern the structure of statements. For example, in mathematics the statement 3 + 3 = 6 has correct syntax, but 3 + = 3$6 does not. In chemistry H_2O is a syntactically correct formula, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3 + = 3$6 is that $ is not a legal token in mathematics (at least as far as I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation *Zz*.

The second type of syntax rule pertains to the way tokens are combined. The equation 3 + = 3 is illegal because even though + and = are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called **parsing**.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

ambiguity:
: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy:
: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness:
: Natural languages are full of idiom and metaphor. If I say, "The penny dropped", there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry:
: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose:
: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs:

> The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Debugging

Programmers make mistakes. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a section, like this one, with my suggestions for debugging. I hope they help!

Glossary

problem solving:
: The process of formulating a problem, finding a solution, and expressing it.

high-level language:
: A programming language like Python that is designed to be easy for humans to read and write.

low-level language:
: A programming language that is designed to be easy for a computer to run; also called "machine language" or "assembly language".

portability:
: A property of a program that can run on more than one kind of computer.

interpreter:
: A program that reads another program and executes it.

prompt:
: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

program:
: A set of instructions that specifies a computation.

print statement:
: An instruction that causes the Python interpreter to display a value on the screen.

operator:
: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

value:
: One of the basic units of data, like a number or string, that a program manipulates.

type:
: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer:
: A type that represents whole numbers.

floating-point:
: A type that represents numbers with fractional parts.

string:
: A type that represents sequences of characters.

natural language:
: Any one of the languages that people speak that evolved naturally.

formal language:
: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token:
: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

syntax:
: The rules that govern the structure of a program.

parse:
: To examine a program and analyze the syntactic structure.

bug:
: An error in a program.

debugging:
: The process of finding and correcting bugs.

Exercises

Exercise 1-1.

It is a good idea to read this book in front of a computer so you can try out the examples as you go.

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the "Hello, world!" program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

1. In a print statement, what happens if you leave out one of the parentheses, or both?
2. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?

3. You can use a minus sign to make a negative number like `-2`. What happens if you put a plus sign before a number? What about `2++2`?
4. In math notation, leading zeros are okay, as in `02`. What happens if you try this in Python?
5. What happens if you have two values with no operator between them?

Exercise 1-2.

Start the Python interpreter and use it as a calculator.

1. How many seconds are there in 42 minutes 42 seconds?
2. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.
3. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?