THINK PYTHON: How to Think Like a Computer Scientist

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 is 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.

1.1 The Python programming language

The programming language you will learn is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.

There are also low-level languages, sometimes referred to as machine languages or assembly languages. Loosely speaking, computers can only run programs written in lowlevel languages. So programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

The advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Lowlevel languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. Figure 1.1 shows the structure of an interpreter.

A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation. Figure 1.2 shows the structure of a compiler.

Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: interactive mode and script mode. In interactive mode, you type Python programs and the interpreter displays the result:

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

The chevron, >>>, is the prompt the interpreter uses to indicate that it is ready. If you type 1 + 1, the interpreter replies 2.

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a script. By convention, Python scripts have names that end with .py.

To execute the script, you have to tell the interpreter the name of the file. If you have a script named dinsdale.py and you are working in a UNIX command window, you type python dinsdale.py. In other development environments, the details of executing scripts are different. You can find instructions for your environment at the Python website http://python.org.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.2 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 (strangely enough) compiling a program.

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, or some other device.
output: Display data on the screen or send data to a file or other device.
math: Perform basic mathematical operations like addition and multiplication.
conditional execution: Check for certain conditions and execute the appropriate code.
repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, thatâ€™s pretty much all there is to it. Every program you have 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.

That may be a little vague, but we will come back to this topic when we talk about algorithms.

1.3 What is debugging?

Programming is error-prone. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging. Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. Syntax refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so (1 + 2) is legal, but 8) is a syntax error.

In English, readers can tolerate most syntax errors, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

1.3.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.3.3 Semantic errors

The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program.

If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth.â€• (A. Conan Doyle, The Sign of Four) For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program. For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip.

According to Larry Greenfield, â€œOne of Linus earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux. (The Linux Users Guide Beta Version 1). Later chapters will make more suggestions about debugging and other programming practices.

Chapter 2 : Variables, expressions and statements

2.1 Values and types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and Hello, World!. These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so-called because it contains a string of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.

```
>>> type(3.2)
<type 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

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

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

Well, that is not what we expected at all! Python interprets 1,000,000 as a commaseparated sequence of integers. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it does not do the right thing.