1. General Introduction

1.1. The Way of the Program

1.2. Algorithms

1.3. The Python Programming Language

1.5. More About Programs

1.6. What is Debugging?

1.7. Syntax errors

1.8. Runtime Errors

1.9. Semantic Errors

1.10. Experimental Debugging

1.11. Formal and Natural Languages

1.12. A Typical First Program

1.13. Comments

1.14. Glossary

1.15. Exercises

1.1. 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.

1.2. Algorithms

If problem solving is a central part of computer science, then the solutions that you create through the problem solving process are also important. In computer science, we refer to these solutions as algorithms. An algorithm is a step by step list of instructions that if followed exactly will solve the problem under consideration.

Our goal in computer science is to take a problem and develop an algorithm that can serve as a general solution. Once we have such a solution, we can use our computer to automate the execution. As noted above, programming is a skill that allows a computer scientist to take an algorithm and represent it in a notation (a program) that can be followed by a computer. These programs are written in programming languages.

Check your understanding

intro-2-1: What is the most important skill for a computer scientist?

A. To think like a computer.

B. To be able to write code really well.

C. To be able to solve problems.

D. To be really good at math.

intro-2-2: An algorithm is:

A. A solution to a problem that can be solved by a computer.

B. A step by step list of instructions that if followed exactly will solve the problem under consideration.

C. A series of instructions implemented in a programming language.

D. A special kind of notation used by computer scientists.

1.3. The Python Programming Language

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

As you might infer from the name high-level language, there are also low-level languages, sometimes referred to as machine languages or assembly languages. Machine language is the encoding of instructions in binary so that they can be directly executed by the computer. Assembly language uses a slightly easier format to refer to the low level instructions. Loosely speaking, computers can only execute programs written in low-level languages. To be exact, computers can actually only execute programs written in machine language. Thus, programs written in a high-level language (and even those in assembly language) have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages. However, the advantages to high-level languages 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. Low-level 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.

Interpret illustration

A compiler reads the program and translates it completely before the program starts running. In this case, 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.

Compile illustration

Many modern languages use both processes. They are first compiled into a lower level language, called byte code, and then interpreted by a program called a virtual machine. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language.

There are two ways to use the Python interpreter: shell mode and program mode. In shell mode, you type Python expressions into the Python shell, and the interpreter immediately shows the result.

In the shell, the >>> is called the Python prompt. The interpreter uses the prompt to indicate that it is ready for instructions. We typed 2 + 3. The interpreter evaluated our expression and replied 5. On the next line it gave a new prompt indicating that it is ready for more input.

Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems.

Alternatively, you can write an entire program by placing lines of Python instructions in a file and then use the interpreter to execute the contents of the file as a whole. Such a file is often referred to as source code. For example, we used a text editor to create a source code file named firstprogram.py with the following contents:

print("My first program adds two numbers, 2 and 3:")

print(2 + 3)

By convention, files that contain Python programs have names that end with .py . Following this convention will help your operating system and other programs identify a file as containing python code.

$ python firstprogram.py

My first program adds two numbers, 2 and 3:

5

These examples show Python being run from a Unix command line. In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

Want to learn more about Python?

If you would like to learn more about installing and using Python, here are some video links. Installing Python for Windows shows you how to install the Python environment under Windows Vista, Installing Python for Mac shows you how to install under Mac OS/X, and Installing Python for Linux shows you how to install from the Linux command line. Using Python shows you some details about the Python shell and source code.

Check your understanding

intro-3-1: Source code is another name for:

A. the instructions in a program, stored in a file.

B. the language that you are programming in (e.g., Python).

C. the environment/tool in which you are programming.

D. the number (or "code") that you must input at the top of each program to tell the computer how to execute your program.

intro-3-2: What is the difference between a high-level programming language and a low-level programming language?

A. It is high-level if you are standing and low-level if you are sitting.

B. It is high-level if you are programming for a computer and low-level if you are programming for a phone or mobile device.

C. It is high-level if the program must be processed before it can run, and low-level if the computer can execute it without additional processing.

D. It is high-level if it is easy to program in and is very short; it is low-level if it is really hard to program in and the programs are really long.

intro-3-3: Pick the best replacements for 1 and 2 in the following sentence: When comparing compilers and interpreters, a compiler is like 1 while an interpreter is like 2.

A. 1 = a process, 2 = a function

B. 1 = translating an entire book, 2 = translating a line at a time

C. 1 = software, 2 = hardware

D. 1 = object code, 2 = byte code

1.5. More About Programs

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something as complex as rendering an html page in a web browser or encoding a video and streaming it across the network. It can also be a symbolic computation, such as searching for 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 and logic

Perform basic mathematical operations like addition and multiplication and logical operations like and, or, and not.

conditional execution

Check for certain conditions and execute the appropriate sequence of statements.

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 more or less like these. Thus, we can describe 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 sequences of these basic instructions.

Check your understanding

intro-5-1: A program is:

A. a sequence of instructions that specifies how to perform a computation.

B. something you follow along at a play or concert.

C. a computation, even a symbolic computation.

D. the same thing as an algorithm.

1.6. What is Debugging?

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called bugs and the process of tracking them down and correcting them is called debugging. Some claim that in 1945, a dead moth caused a problem on relay number 70, panel F, of one of the first computers at Harvard, and the term bug has remained in use since. For more about this historic event, see first bug.

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.

Check your understanding

intro-6-1: Debugging is:

A. tracking down programming errors and correcting them.

B. removing all the bugs from your house.

C. finding all the bugs in the program.

D. fixing the bugs in the program.

1.7. Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution of your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

Check your understanding

intro-7-1: Which of the following is a syntax error?

A. Attempting to divide by 0.

B. Forgetting a colon at the end of a statement where one is required.

C. Forgetting to divide by 100 when printing a percentage amount.

1.8. Runtime Errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. 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.

Check your understanding

intro-8-1: Which of the following is a run-time error?

A. Attempting to divide by 0.

B. Forgetting a colon at the end of a statement where one is required.

C. Forgetting to divide by 100 when printing a percentage amount.

1.9. 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. However, your program 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.

Check your understanding

intro-9-1: Which of the following is a semantic error?

A. Attempting to divide by 0.

B. Forgetting a colon at the end of a statement where one is required.

C. Forgetting to divide by 100 when printing a percentage amount.

intro-9-2: Who or what typically finds semantic errors?

A. The programmer.

B. The compiler / interpreter.

C. The computer.

D. The teacher / instructor.

1.10. 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 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. (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 kernel that contains millions 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, one of Linus’s earlier projects was a program that would switch between displaying 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.

Check your understanding

intro-10-1: The difference between programming and debugging is:

A. programming is the process of writing and gradually debugging a program until it does what you want.

B. programming is creative and debugging is routine.

C. programming is fun and debugging is work.

D. there is no difference between them.

1.11. Formal and Natural Languages

Natural languages are the languages that 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 rules about syntax. For example, 3+3=6 is a syntactically correct mathematical statement, but 3=+6$ is not. H2O is a syntactically correct chemical name, 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=+6$ is that $ is not a legal token in mathematics (at least as far as we know). Similarly, 2Zz is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement— that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal because you can’t place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

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

For example, when you hear the sentence, “The other shoe fell”, you understand that the other shoe is the subject and fell is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

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

Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, “The other shoe fell”, there is probably no shoe and nothing falling.

Tip

You’ll need to find the original joke to understand the idiomatic meaning of the other shoe falling. Yahoo! Answers thinks it knows!

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, 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.

program

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

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea 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. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Check your understanding

intro-11-1: The differences between natural and formal languages include:

A. natural languages can be parsed while formal languages cannot.

B. ambiguity, redundancy, and literalness.

C. there are no differences between natural and formal languages.

D. tokens, structure, syntax, and semantics.

intro-11-2: True or False: Reading a program is like reading other kinds of text.

A. True

B. False

1.12. A Typical First Program

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello, World! In Python, the source code looks like this.

print("Hello, World!")

This is an example of using the print function, which doesn’t actually print anything on paper. It displays a value on the screen. In this case, the result is the phrase:

Hello, World!

Here is the example in activecode. Give it a try!

1

print("Hello, World!")

2

The quotation marks in the program mark the beginning and end of the value. They don’t appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hello, World! program. By this standard, Python does about as well as possible.

Check your understanding

intro-12-2: The print function:

A. sends information to the printer to be printed on paper.

B. displays a value on the screen.

C. tells the computer to put the information in print, rather than cursive, format.

D. tells the computer to speak the information.

1.13. Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments.

A comment in a computer program is text that is intended only for the human reader - it is completely ignored by the interpreter. In Python, the # token starts a comment. The rest of the line is ignored. Here is a new version of Hello, World!.

1 #---------------------------------------------------

2 # This demo program shows off how elegant Python is!

3 # Written by Thomas, January 2021

4 # Anyone may freely copy or modify this program.

5 #---------------------------------------------------

6

​

7 print("Hello, World!") # Isn't this easy!

8

​

Notice that when you run this program, it still only prints the phrase Hello, World! None of the comments appear. You’ll also notice that we’ve left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse. Use them liberally!

Check your understanding

intro-13-2: What are comments for?

A. To tell the computer what you mean in your program.

B. For the people who are reading your code to know, in natural language, what the program is doing.

C. Nothing, they are extraneous information that is not needed.

D. Nothing in a short program. They are only needed for really large programs.

1.14. Glossary

algorithm

A general step by step process for solving a problem.

bug

An error in a program.

byte code

An intermediate language between source code and object code. Many modern languages first compile source code into byte code and then interpret the byte code with a program called a virtual machine.

comment

Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

compile

To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

debugging

The process of finding and removing any of the three kinds of programming errors.

exception

Another name for a runtime error.

executable

Another name for object code that is ready to be executed.

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.

high-level language

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

interpret

To execute a program in a high-level language by translating it one line at a time.

low-level language

A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

natural language

Any one of the languages that people speak that evolved naturally.

object code

The output of the compiler after it translates the program.

parse

To examine a program and analyze the syntactic structure.

portability

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

print function

A function used in a program or script that causes the Python interpreter to display a value on its output device.

problem solving

The process of formulating a problem, finding a solution, and expressing the solution.

program

A sequence of instructions that specifies to a computer actions and computations to be performed.

programming language

A formal notation for representing solutions.

Python shell

An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter for processing.

runtime error

An error that does not occur until the program has started to execute but that prevents the program from continuing.

semantic error

An error in a program that makes it do something other than what the programmer intended.

semantics

The meaning of a program.

shell mode

A style of using Python where we type expressions at the command prompt, and the results are shown immediately. Contrast with source code, and see the entry under Python shell.

source code

A program, stored in a file, in a high-level language before being compiled or interpreted.

syntax

The structure of a program.

syntax error

An error in a program that makes it impossible to parse — and therefore impossible to interpret.

token

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

1.15. Exercises

There are not currently any exercises for this chapter. It is included here as a subchapter for consistency with other chapters.

​