介绍

张晓平

1 快速开始

The way we think about programming has undergone many changes in the years since the first electronic computers required patch cables and switches to convey instructions from human to machine. As is the case with many aspects of society, changes in computing technology provide computer scientists with a growing number of tools and platforms on which to practice their craft. Advances such as faster processors, high-speed networks, and large memory capacities have created a spiral of complexity through which computer scientists must navigate. Throughout all of this rapid evolution, a number of basic principles have remained constant. The science of computing is concerned with using computers to solve problems.

自计算机通过电缆和交换机来传递人的指令起，人们对编程的看法就发生了许多变化。与社会的许多方面一样，计算技术的变化为计算机科学家提供了越来越多的工具和平台来实践他们的工艺。计算机的快速发展，诸如快速处理器、高速网络和大存储器容量已经让计算机科学家陷入高度复杂螺旋中。在所有这些快速演变中，一些基本原则保持不变。计算机科学关注用计算机来解决问题。

You have no doubt spent considerable time learning the basics of problem-solving and hopefully feel confident in your ability to take a problem statement and develop a solution. You have also learned that writing computer programs is often hard. The complexity of large problems and the corresponding complexity of the solutions can tend to overshadow the fundamental ideas related to the problem-solving process.

或许你花了很多时间学习了解决问题的基础知识，希望自己能把问题弄清楚并提出解决方案。接着你可能还会发现编程有些难。问题以及解决方案的复杂性可能会掩盖求解过程中的一些基本思想。

This chapter emphasizes two important areas for the rest of the text. First, it reviews the framework within which computer science and the study of algorithms and data structures must fit, in particular, the reasons why we need to study these topics and how understanding these topics helps us to become better problem solvers. Second, we review the Python programming language. Although we cannot provide a detailed, exhaustive reference, we will give examples and explanations for the basic constructs and ideas that will occur throughout the remaining chapters.

本章的其余部分将着重介绍两个重要的领域。首先回顾一下计算机科学与研究算法和数据结构所必须适应的框架，特别是我们需要研究这些主题的原因，以及如何理解这些主题有助于我们更好的解决问题。其次我们将回顾 Python 编程语言。这里不提供详尽的参考，但我们将在其余章节中给出基本数据结构的示例和解释。

2 什么是计算机科学

Computer science is often difficult to define. This is probably due to the unfortunate use of the word "computer" in the name. As you are perhaps aware, computer science is not simply the study of

computers. Although computers play an important supporting role as a tool in the discipline, they are just that–tools.

计算机科学不好定义，由于在名字中有“计算机”一词。然而，计算机科学并非简单地研究计算机，尽管计算机作为一种工具在学科中发挥重要的支持作用，但它们只是工具。

Computer science is the study of problems, problem-solving, and the solutions that come out of the problem-solving process. Given a problem, a computer scientist’ s goal is to develop an algorithm, a step-by-step list of instructions for solving any instance of the problem that might arise. Algorithms are finite processes that if followed will solve the problem. Algorithms are solutions.

计算机科学研究问题、解决问题并生成解决问题的方案。给定一个问题，计算机科学家的目标是开发一个算法，一系列的指令列表，用于解决可能出现的问题。算法遵循它有限的过程就可以解决问题。

Computer science can be thought of as the study of algorithms. However, we must be careful to include the fact that some problems may not have a solution. Although proving this statement is beyond the scope of this text, the fact that some problems cannot be solved is important for those who study computer science. We can fully define computer science, then, by including both types of problems and stating that computer science is the study of solutions to problems as well as the study of problems with no solutions.

计算机科学可以被认为是对算法的研究。但是，我们必须清楚地认识到，一些问题可能没有解决方案。虽然证明这种说法正确性超出了本文的范围，但一些问题不能解决的事实对于那些研究计算机科学的人是很重要的。可以这么说，计算机科学研究有解决方案和没有解决方案的问题。

It is also very common to include the word computable when describing problems and solutions. We say that a problem is computable if an algorithm exists for solving it. An alternative definition for computer science, then, is to say that computer science is the study of problems that are and that are not computable, the study of the existence and the nonexistence of algorithms. In any case, you will note that the word “computer” did not come up at all. Solutions are considered independent from the machine.

当描述问题及其解决方案时，会提到计算一词。若存在一个算法解决某个问题，就称该问题是可计算的。计算机科学的另一个定义是：计算机科学是研究那些可计算和不可计算的问题，研究是不是存在一种算法来解决它。请注意这里没有涉及到“计算机”一词，解决方案与机器无关。

Computer science, as it pertains to the problem-solving process itself, is also the study of abstraction. Abstraction allows us to view the problem and solution in such a way as to separate the so-called logical and physical perspectives. The basic idea is familiar to us in a common example.

计算机科学，因涉及问题解决过程本身，是关于抽象的研究。抽象使我们能从逻辑视角和物理视角来分别看待问题及解决方案。基本思想跟我们常见的例子一样。

Consider the automobile that you may have driven to school or work today. As a driver, a user of the car, you have certain interactions that take place in order to utilize the car for its intended purpose. You get in, insert the key, start the car, shift, brake, accelerate, and steer in order to drive. From an abstraction point of view, we can say that you are seeing the logical perspective of the automobile. You are using the functions provided by the car designers for the purpose of transporting you from one location to another. These functions are sometimes also referred to as the interface.

假设你开车上学或上班。作为司机，也就是汽车的用户，你为了让汽车载你到目的地，你会和汽车有些互动，如上汽车、插钥匙、点火、换挡、制动、加速和转向。从抽象的角度，你所看到的是汽车的逻辑视角。你使用的是汽车设计者提供的功能，将你从一个地方载到另一个地方。这些功能有时也被称为接口。

On the other hand, the mechanic who must repair your automobile takes a very different point of view. She not only knows how to drive but must know all of the details necessary to carry out all the functions that we take for granted. She needs to understand how the engine works, how the transmission

shifts gears, how temperature is controlled, and so on. This is known as the physical perspective, the details that take place "under the hood."

另一方面，汽车修理师傅则有一个截然不同的视角。他不仅知道如何开车，还必须知道所有必要的细节，使我们认为理所当然的功能运行起来。他需要了解发动机如何工作、变速箱如何变速、温度如何控制等等。这就是物理视角，细节发生在"引擎盖下"。

The same thing happens when we use computers. Most people use computers to write documents, send and receive email, surf the web, play music, store images, and play games without any knowledge of the details that take place to allow those types of applications to work. They view computers from a logical or user perspective. Computer scientists, programmers, technology support staff, and system administrators take a very different view of the computer. They must know the details of how operating systems work, how network protocols are configured, and how to code various scripts that control function. They must be able to control the low-level details that a user simply assumes.

我们用电脑时也会发生同样的情况。大多数人使用计算机写文档、收发电子邮件、上网冲浪、播放音乐、存储图像和玩游戏，但他们并不知道这些应用程序工作的细节。他们从逻辑或用户角度看待计算机。计算机科学家、程序员、技术支持人员和系统管理员看待计算机的角度截然不同。他们必须知道操作系统如何工作、如何配置网络协议以及如何编写控制功能的各种脚本。总言之，他们必须能够控制底层的细节。

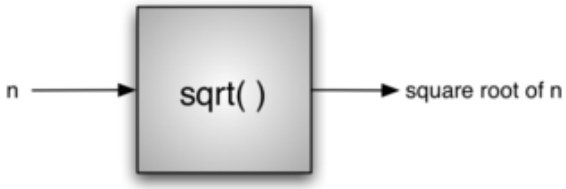
The common point for both of these examples is that the user of the abstraction, sometimes also called the client, does not need to know the details as long as the user is aware of the way the interface works. This interface is the way we as users communicate with the underlying complexities of the implementation. As another example of abstraction, consider the Python math module. Once we import the module, we can perform computations such as

这两个例子的共同点是用户态的抽象，也称为客户端，不需要知道细节，只要用户知道接口的工作方式。这个接口是用户与底层沟通的方式。作为抽象的另一个例子，Python 数学模块。一旦导入模块，我们可以执行计算

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

This is an example of procedural abstraction. We do not necessarily know how the square root is being calculated, but we know what the function is called and how to use it. If we perform the import correctly, we can assume that the function will provide us with the correct results. We know that someone implemented a solution to the square root problem but we only need to know how to use it. This is sometimes referred to as a "black box" view of a process. We simply describe the interface: the name of the function, what is needed (the parameters), and what will be returned. The details are hidden inside (see Figure 1).

这是一个抽象的例子。我们没必要知道如何计算平方根，只需知道函数是什么以及如何使用它。如果导入正确，我们就认为函数会提供正确的结果。我们知道，有人实现了平方根问题的解决方案，但我们只需知道如何去使用它。这是一个"黑盒子"，其接口可描述为：函数名、参数、返回值，其细节隐藏在内部：



3 什么是编程

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer. Although many programming languages and many different types of computers exist, the important first step is the need to have the solution. Without an algorithm there can be no program.

编程是将算法转换为编程语言的过程，以便被计算机执行。然而，编程语言有很多，计算的种类也不同，首先我们要有解决方案，没有算法就没有程序。

Computer science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

计算机科学不研究编程，但编程却是计算机科学家的重要能力。编程通常是表达解决方案的方式。因此，这种语言表现形式和创造它的过程成为该学科的基本部分。

Algorithms describe the solution to a problem in terms of the data needed to represent the problem instance and the set of steps necessary to produce the intended result. Programming languages must provide a notational way to represent both the process and the data. To this end, languages provide control constructs and data types.

算法描述了依据问题实例数据所产生的解决方案和产生预期结果所需的一套步骤。编程语言必须提供一种表示方法来表示过程和数据。为此，它提供了控制结构和数据类型。

Control constructs allow algorithmic steps to be represented in a convenient yet unambiguous way. At a minimum, algorithms require constructs that perform sequential processing, selection for decision-making, and iteration for repetitive control. As long as the language provides these basic statements, it can be used for algorithm representation.

控制结构允许以方便而明确的方式表示算法步骤。至少，算法需要执行顺序处理、决策选择和重复控制迭代。只要语言提供这些基本语句，它就可以表达算法。

All data items in the computer are represented as strings of binary digits. In order to give these strings meaning, we need to have data types. Data types provide an interpretation for this binary data so that we can think about the data in terms that make sense with respect to the problem being solved. These low-level, built-in data types (sometimes called the primitive data types) provide the building blocks for algorithm development.

计算机中的所有数据项都由一串一串的二进制数表示。为了让这些二进制串有意义，就需要有数据类型。数据类型为二进制数据提供解释，以便我们能够根据实际问题来思考数据。这些底层的内置数据类型（有时称为原始数据类型）为算法开发提供了基础。

For example, most programming languages provide a data type for integers. Strings of binary digits in the computer' s memory can be interpreted as integers and given the typical meanings that we commonly associate with integers (e.g. 23, 654, and -19). In addition, a data type also provides a description of the operations that the data items can participate in. With integers, operations such as addition, subtraction, and multiplication are common. We have come to expect that numeric types of data can participate in

these arithmetic operations.

例如，大多数编程语言为整数提供数据类型。内存中的二进制数据可以解释为整数，并且能给予一个我们通常与整数（例如 23,654 和 -19）相关联的含义。此外，数据类型还提供数据项参与的操作的描述。对于整数，诸如加法、减法和乘法的操作是常见的。我们期望数值类型的数据可以参与这些算术运算。

The difficulty that often arises for us is the fact that problems and their solutions are very complex. These simple, language-provided constructs and data types, although certainly sufficient to represent complex solutions, are typically at a disadvantage as we work through the problem-solving process. We need ways to control this complexity and assist with the creation of solutions.

通常我们遇到的困难是问题及其解决方案非常复杂。由语言提供的简单的结构和数据类型，虽然可以表示复杂的解决方案，但在实际中却不好用。我们需要一些方法控制这种复杂性，以助于形成更好的解决方案。

4 为什么要学习数据结构和抽象数据类型

To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. These models allow us to describe the data that our algorithms will manipulate in a much more consistent way with respect to the problem itself.

为了管理问题的复杂性及解决过程，计算机科学家使用抽象使他们能够专注于"大局"而不会迷失在细节中。通过对问题进行建模，我们能够利用更好和更有效的问题解决过程。这些模型允许我们以更加一致的方式描述我们的算法将要处理的数据。

Earlier, we referred to procedural abstraction as a process that hides the details of a particular function to allow the user or client to view it at a very high level. We now turn our attention to a similar idea, that of data abstraction. An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user' s view. This is called information hiding.

之前，我们将过程抽象称为隐藏特定函数的细节的过程，以允许用户或客户端在高层查看它。我们现在将注意力转向类似的思想，即数据抽象的思想。抽象数据类型（有时缩写为 ADT ）是对我们如何查看数据和允许的操作的逻辑描述，而不用考虑如何实现它们。这意味着我们只关心数据表示什么，而不关心它最终将如何构造。通过提供这种级别的抽象，我们围绕数据创建一个封装。通过封装实现细节，我们将它们从用户的视图中隐藏。这称为信息隐藏。

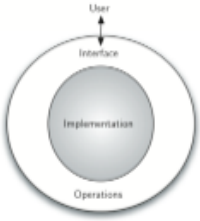


Figure 2 shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract

data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.

Figure 2 展示了抽象数据类型是什么以及如何操作。用户与接口交互，使用抽象数据类型指定的操作。抽象数据类型是用户与之交互的 shell。实现隐藏在更深的底层。用户不关心实现的细节。

The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will allow us to define the complex data models for our problems without giving any indication as to the details of how the model will actually be built. This provides an implementation-independent view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

抽象数据类型（通常称为数据结构）的实现将要求我们使用一些程序构建和原始数据类型的集合来提供数据的物理视图。正如我们前面讨论的，这两个视角的分离将允许我们将问题定义复杂的数据模型，而不给出关于模型如何实际构建的细节。这提供了独立于实现的数据视图。由于通常有许多不同的方法来实现抽象数据类型，所以这种实现独立性允许程序员在不改变数据的用户与其交互的方式的情况下切换实现的细节。用户可以继续专注于解决问题的过程。

5 为什么要学习算法

Computer scientists learn by experience. We learn by seeing others solve problems and by solving problems by ourselves. Being exposed to different problem-solving techniques and seeing how different algorithms are designed helps us to take on the next challenging problem that we are given. By considering a number of different algorithms, we can begin to develop pattern recognition so that the next time a similar problem arises, we are better able to solve it.

计算机科学家经常通过经验学习。我们通过看别人解决问题和自己解决问题来学习。接触不同的问题解决技术，看不同的算法设计有助于我们承担下一个具有挑战性的问题。通过思考许多不同的算法，我们可以开始开发模式识别，以便下一次出现类似的问题时，我们能够更好地解决它。

Algorithms are often quite different from one another. Consider the example of sqrt seen earlier. It is entirely possible that there are many different ways to implement the details to compute the square root function. One algorithm may use many fewer resources than another. One algorithm might take 10 times as long to return the result as the other. We would like to have some way to compare these two solutions. Even though they both work, one is perhaps "better" than the other. We might suggest that one is more efficient or that one simply works faster or uses less memory. As we study algorithms, we can learn analysis techniques that allow us to compare and contrast solutions based solely on their own characteristics, not the characteristics of the program or computer used to implement them.

算法通常彼此完全不同。考虑前面看到的 sqrt 的例子。完全可能的是，存在许多不同的方式来实现细节以计算平方根函数。一种算法可以使用比另一种更少的资源。一个算法可能需要 10 倍的时间来返回结果。我们想要一些方法来比较这两个解决方案。即使他们都工作，一个可能比另一个 "更好"。我们建议使用一个更高效，或者一个只是工作更快或使用更少的内存的算法。当我们研究算法时，我们可以学习分析技术，允许我们仅仅根据自己的特征而不是用于实现它们的程序或计算机的特征来比较和对比解决方案。

In the worst case scenario, we may have a problem that is intractable, meaning that there is no algorithm that can solve the problem in a realistic amount of time. It is important to be able to distinguish between those problems that have solutions, those that do not, and those where solutions

exist but require too much time or other resources to work reasonably.

在最坏的情况下，我们可能有一个难以处理的问题，这意味着没有算法可以在实际的时间量内解决问题。重要的是能够区分具有解决方案的那些问题，不具有解决方案的那些问题，以及存在解决方案但需要太多时间或其他资源来合理工作的那些问题。

There will often be trade-offs that we will need to identify and decide upon. As computer scientists, in addition to our ability to solve problems, we will also need to know and understand solution evaluation techniques. In the end, there are often many ways to solve a problem. Finding a solution and then deciding whether it is a good one are tasks that we will do over and over again.

经常需要权衡，我们需要做决定。作为计算机科学家，除了我们解决问题的能力，我们还需要了解解决方案评估技术。最后，通常有很多方法来解决问题。找到一个解决方案，我们将一遍又一遍比较，然后决定它是否是一个好的方案。