**1.1我们的哲学**

很多人会认为这是两本书合二为一。 有一本书是编程入门，教你组织数据的基本概念和对数据进行操作的程序，最后是对普遍有用的算法的研究。 另一本书是编程语言入门:从一个层面上研究我们用来构建这些数据和程序的媒体。

显然，这些都不是不相干的话题。 我们通过一种或多种语言来学习编程，然后我们编写的程序就会自然而然地成为学习的对象，从而更全面地理解语言。 尽管如此，这些被认为是完全不同的主题，因此将分别讨论。 我们也是这样接近他们的。 唯一高贵的例外是有史以来最好的计算机科学书籍《计算机程序的结构与解释》。

我们已经认识到，这种分离既没有意义，也没有帮助。 这些主题是紧密交织在一起的，通过接受这种交织，结果很可能是一本好得多的书。 这是我对这种格式的实验。

**1.2可预测性作为主题**

组织学习编程和编程语言的方法有很多。 我的中心主题是可预测性的概念。

程序通常是静态的:它们生活在道德上，就像一张纸，一成不变。 但是，当我们运行一个程序时，它会产生一个复杂的、动态的行为，从而产生效用、乐趣和(有时)挫折。 每个编写程序的人最终都会关心——不管他们意识到与否——从前者预测后者。 有时我们甚至编写程序来帮助我们完成这项任务(我们将在示例、测试、程序检查、预测增长、关于程序的推理:第一次查看类型，以及其他地方看到)。 (本段包含链接：[Examples, Testing, and Program Checking](https://papl.cs.brown.edu/2018/testing.html" \t "_blank)  [Reasoning about Programs: A First Look at Types](https://papl.cs.brown.edu/2018/types.html)  [Predicting Growth](https://papl.cs.brown.edu/2018/predicting-growth.html)  )

可预测性名声不好。 在“程序推理”的伪装下，它开始被同时视为高尚和令人麻木的无聊。 它当然是高尚的，但我们将尝试以一种希望看起来完全自然、确实完全显而易见的方式来呈现它(因为我们相信它是如此)。 希望您在完成这项研究后，能够合理地确信可预测性在您自己的工作中的中心地位，并将其作为编程语言设计的一个度量标准。

**1.3本书的结构**

与其他一些教科书不同，这本书没有遵循自上而下的叙述。 相反，它有对话的流程，带有回溯。 我们经常会增量地构建程序，就像一对程序员那样。 我们会犯错误，不是因为我们不知道更好，而是因为这是你学习的最好方法。 包括错误使你不可能被动地阅读:你必须转而投入阅读材料，因为你永远无法确定你所读内容的准确性。

最后，你总会得到正确的答案。 然而，这种非线性的路径在短期内更令人沮丧(你经常会忍不住说，“现在就告诉我答案吧!”)，这使得这本书成为一个糟糕的参考指南(你不能打开任意一页，然后确定它说的是正确的)。 然而，这种挫折感就是学习的感觉。 我们不知道有什么办法。

在不同的地方你会遇到这样的情况:

**锻炼**

这是一个练习。 做试一试。

这是传统教科书上的练习。 这是你需要自己做的事情。 如果你正在使用这本书作为课程的一部分，这很可能已经作为家庭作业。 相反，你也会发现类似练习的问题是这样的:

***现在做!***

这里有一个活动! 你看到了吗?

当你遇到这种情况时，停下来。 在你继续之前，阅读、思考并形成一个答案。 你必须这样做，因为这实际上是一个练习，但答案已经在书中——最常见的是在紧接着的文本中(即 或者你可以通过运行一个程序来自己决定。 如果你只是继续读下去，你会在没有思考的情况下看到答案(或者根本没有看到，如果指令是运行一个程序的话)，所以你既不会测试你的知识，也不会提高你的直觉。 换句话说，这些都是鼓励积极学习的额外的、明确的尝试。 然而，最终我们只能鼓励它; 由你来练习。

**1.4本书的语言**

本书使用了一种新的编程语言Pyret。 Pyret是我们在函数式、面向对象和脚本语言以及它们的类型系统、程序分析和开发环境的编程和设计方面的丰富经验的产物。

该语言的语法受到Python的启发。 与Python不同，Pyret将强制缩进而不是解释缩进:也就是说，缩进将成为另一个语法格式良好的标准。 但这还没有实现。 它适合计算机科学教育的利基失踪了两个奇怪的问题简单的语言(有很多)Python的Python添加重要特性,缺乏对学习编程(如代数数据类型,可选注释变量,设计决策,更好地使开发环境建设,并大力支持测试)。 初学者可以在他们被照顾的知识中休息，而过去熟悉语言动物园的程序员，从蛇到单峰骆驼，应该发现Pyret熟悉和舒适。 (本段包含链接：[of which there are many](http://cs.brown.edu/~sk/Publications/Papers/Published/pmmwplck-python-full-monty/)  )

**1.1 Our Philosophy**

Many people would regard this as being two books in one. One book is an introduction to *programming*, teaching you basic concepts of organizing data and the programs that operate over them, ending in the investigation of universally useful algorithms. The other book is an introduction to *programming languages*: a study, from one level up, of the media by which we structure these data and programs.

Obviously, these are not unrelated topics. We learn programming through one or more languages, and the programs we write then become natural subjects of study to understand languages at large. Nevertheless, these are considered sufficiently different topics that they are approached separately. This is how we approached them, too.The one noble exception to this separation is the best computer science book ever written, *The Structure and Interpretation of Computer Programs*.

We have come to realize that this separation is neither meaningful nor helpful. The topics are deeply intertwined and, by accepting that interleaving, the result is likely to be a much better book. This is my experiment with that format.

**1.2 Predictability as a Theme**

There are many ways to organize the study of programming and programming languages. My central theme is the concept of *predictability*.

Programs are typically static: they live on the moral equivalent of a paper, unmoving and unchanging. But when we run a program, it produces a complex, dynamic behavior that yields utility, pleasure, and (sometimes) frustration. Everyone who writes programs ultimately cares—whether they realize it or not—in *predicting* the latter from the former. Sometimes we even write programs to help us with this task (as we’ll see in Examples, Testing, and Program Checking, Predicting Growth, Reasoning about Programs: A First Look at Types, and elsewhere).

Predictability has a bad rap. Under the guise of “program reasoning”, it came to be viewed simultaneously as both noble and mind-numbingly boring. It is certainly noble, but we will try to present it a way that will hopefully seem utterly natural, indeed entirely obvious (because we believe it is). Hopefully you’ll come away from this study reasonably convinced about the central place of predictability in your own work, and as a metric for programming language design.

**1.3 The Structure of This Book**

Unlike some other textbooks, this one does not follow a top-down narrative. Rather it has the flow of a conversation, with backtracking. We will often build up programs incrementally, just as a pair of programmers would. We will include mistakes, not because we don’t know better, but because *this is the best way for you to learn*. Including mistakes makes it impossible for you to read passively: you must instead engage with the material, because you can never be sure of the veracity of what you’re reading.

At the end, you’ll always get to the right answer. However, this non-linear path is more frustrating in the short term (you will often be tempted to say, “Just tell me the answer, already!”), and it makes the book a poor reference guide (you can’t open up to a random page and be sure what it says is correct). However, that feeling of frustration is the sensation of learning. We don’t know of a way around it.

At various points you will encounter this:

**Exercise**

This is an exercise. Do try it.

This is a traditional textbook exercise. It’s something you need to do on your own. If you’re using this book as part of a course, this may very well have been assigned as homework. In contrast, you will also find exercise-like questions that look like this:

***Do Now!***

There’s an activity here! Do you see it?

When you get to one of these, **stop**. Read, think, and formulate an answer before you proceed. You must do this because this is actually an *exercise*, but the answer is already in the book—most often in the text immediately following (i.e., in the part you’re reading right now)—or is something you can determine for yourself by running a program. If you just read on, you’ll see the answer without having thought about it (or not see it at all, if the instructions are to run a program), so you will get to neither (a) test your knowledge, nor (b) improve your intuitions. In other words, these are additional, explicit attempts to encourage active learning. Ultimately, however, we can only encourage it; it’s up to you to practice it.

**1.4 The Language of This Book**

This book uses a new programming language called Pyret. Pyret is the outgrowth of our deep experience programming in and designing functional, object-oriented, and scripting languages, as well as their type systems, program analyses, and development environments.

The language’s syntax is inspired by Python.Unlike Python, Pyret will enforce indentation rather than interpret it: that is, indentation will simply become another syntax well-formedness criterion. But that hasn’t been implemented yet. It fits the niche missing in computer science education of a simple language that sheds both the strange corner-cases (of which there are many) of Python while adding important features that Python lacks for learning programming (such as algebraic datatypes, optional annotations on variables, design decisions that better enable the construction of development environments, and strong support for testing). Beginning programmers can rest in the knowledge they are being cared for, while programmers with past acquaintance of the language menagerie, from serpents to dromedaries, should find Pyret familiar and comfortable.

文档中第 1 至 1000 字的翻译内容正在加载...

确定