第1章. 你是否需要Python，以及如何掌握Python?

从2021年10月开始，Python在TIOBE编程语言排行榜上位列榜首，成为了全世界最受欢迎的编程语言[1]。Python的火爆使Python培训班如雨后春笋般涌现，各种关于Python的书籍也汗牛充栋。那么你是否有必要“跟风”学Python呢？如果是又该如何花费最少的时间和精力掌握它呢？这就是本章所要解决的两个现实问题。

1-1. 第一个幻觉——Python是一门新编程语言

Python近年来的流行使很多人产生了这样一种错误印象——Python是一门新编程语言。事实上，Python由荷兰程序员吉多·范罗苏姆（Guido van Rossum）从1989年12月开始编写，于1991年2月正式发布[2]。

作为比较，下面列出目前在工业界得到广泛应用（或具有良好前景）的通用编程语言的诞生时间：

- C：1972年[3]。（你爷爷还是你爷爷。）
- C++：1985年[4]。（你爸爸还是你爸爸。）
- Java：1995年5月[5]。（刺激1995三兄弟的大哥。）
- PHP：1995年6月[6]。（刺激1995三兄弟的二哥。）
- JavaScript：1995年12月[7]。（刺激1995三兄弟的三弟。）
- C#：2000年[8]。（新秀小兄弟。）
- Go：2009年[9]。（乖儿子。）
- Swift：2014年[10]。（少年，取代了20世纪80年代中期诞生的叔叔辈的Objective-C[11]。）

根据诞生年份，Python是老大哥辈的，与Visual Basic同辈。Microsoft公司已经于2020年宣布停止Visual Basic的演化，宣告这门编程语言的生命走向了倒计时[12]，而Python却同时开始挑战C和Java的统治地位。因此，对Python的准确描述应该是——“大器晚成”。

1-2. 第二个幻觉——Python很容易学

“Python很容易学”——这是宣传Python时不会遗漏的一大卖点。这句话不能说是错误的，因为相对于其他编程语言，Python确实很容易理解。然而这并不意味着Python的语法很简单。事实上，Python的语法至少比C的语法复杂一个数量级。如果以了解所有的语法作为学习完一门编程语言的标准，那么学习Python所需的时间可能大于学习C、Fortran和Pascal所需的时间之和。Python的语法很复杂，但很容易理解；C的语法很简单，但很难理解。明确这两者的区别是你在决定学习Python之前必须做好的心理准备。

图1-1是本书对主流编程语言的分类。注意“中级语言”这一提法并没有得到学术界的广泛认可，他们倾向于将机器语言和汇编语言之外的所有编程语言都统称为“高级语言”。然而鉴于面向对象语言和过程化语言的差别是如此之大，本书将前者称为“高级语言”，将后者称为“中级语言”。

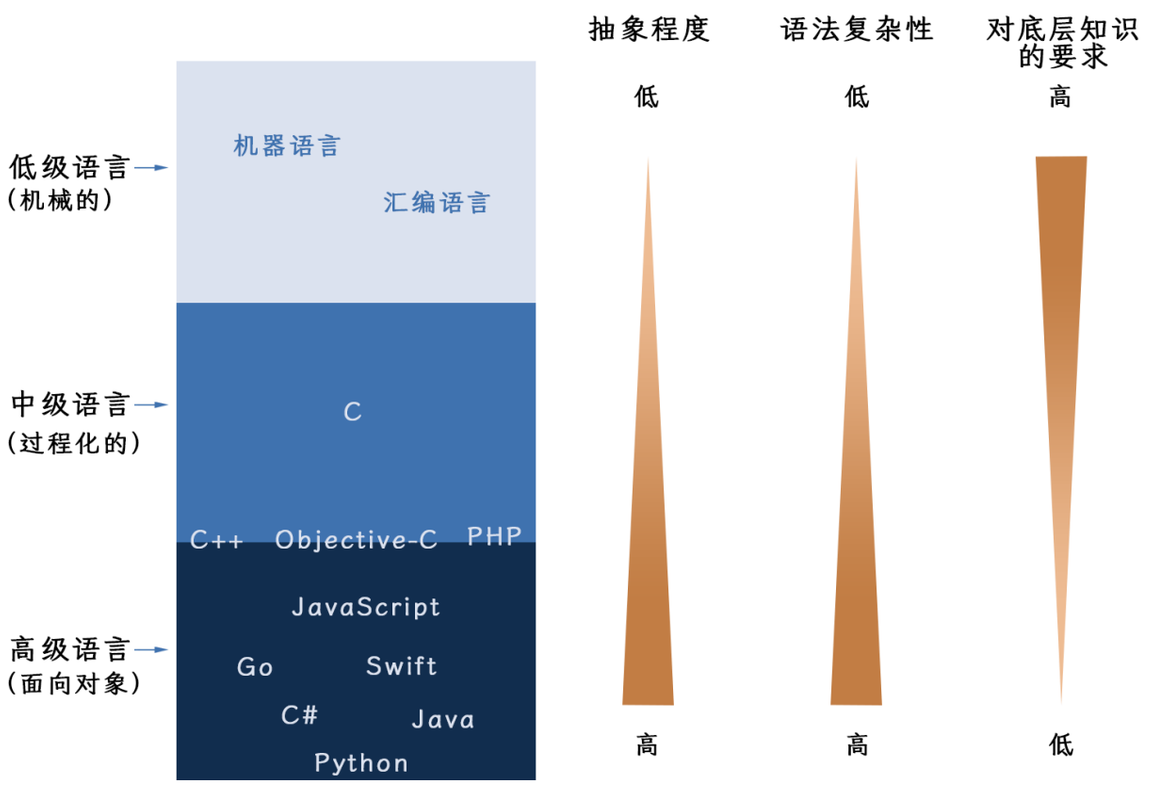

图1-1. 编程语言的分类

大体上，编程语言的抽象程度越高，其语法也就越复杂，其用法对计算机系统底层知识的要求也就越低。

语法最简单的是低级语言。例如ARM指令集架构只有约60条指令，x86指令集架构也只有约170条指令，且每条指令的语法和功能都极其简单。低级语言令人望而生畏的地方在于，用它们编程必须对计算机硬件的每个细节都了如指掌，例如清楚这条指令将数据从内存的哪个位置拷贝到了哪个寄存器，那条指令又将内存中的哪部分数据发送给了哪个外围设备以执行I/O操作。这是从机器的角度去编写程序，完全不符合人类的思维方式，因此非常困难。

中级语言的语法比低级语言要复杂一些。例如C的语法包括数据类型、分支、循环和宏等等。这已经在很大程度上使编程符合人类的思维方式，但依然没有完全屏蔽底层细节。我们必须小心地考虑每个变量在内存空间中的储存方式、访问限制和生存期，乃至字节的排列顺序和有效位之间的空隙。这使得除了少数计算机专家之外，大部分人依然觉得编程很困难。

因为引入了面向对象技术，高级语言的语法必然比中级语言的语法复杂至少一个数量级。而付出这一代价的收获是，绝大部分计算机系统底层细节被屏蔽，编程更符合人类的思维方式，极大降低了编程的门槛。

需要强调的是，高级语言并非抽象层度越高语法就越复杂（因此图1-1并非完全准确）。虽然高级语言都试图使编程更符合人类的思维方式，但因为各自的定位不同，所以在语法复杂性上依然存在很大差异。图1-1列出的高级语言按照其定位可以这样分类：

➤ C++、Objective-C、Go和Swift：定位是代替C开发在单机上运行的大型高性能应用程序。它们必须提供足够的特性以使编程者能够充分利用硬件和操作系统的性能，且必须注重执行速度，因此在所有编程语言中属于语法最复杂的一批（C++是公认的语法最复杂的编程语言）。C++和Objective-C都兼具过程化和面向对象的特性，抽象层度很低。Go和Swift分别是它们的替代品，属于较纯粹的面向对象语言（但Go实现面向对象的方式较特殊），抽象层度较高。

➤ Java和C#：定位是开发既能够在单机上运行也能在计算机集群上运行的超大型应用程序。这些程序被用C或前一类高级语言开发的中间件（更准确地说是解释器）解释执行。它们以牺牲一定的执行速度为代价，换取足够高的容错性和稳健性，以及开发效率的极大提升。它们属于较纯粹的面向对象语言，语法复杂性仅次于前一类高级语言，抽象层度则仅次于Python。

➤ PHP：定位是快速开发小型应用程序。强调保持代码轻量级，追求极致的开发效率。原本属于中级语言，后来添加了对面向对象的支持（与Visual Basic类似），所以语法复杂性最低，抽象层度也最低。

➤ JavaScript：定位是以网页的形式开发用户界面。至今依然不能严格支持面向对象，抽象层度不高。但由于用户界面本身具有很高的复杂性，所以其语法也相当复杂（低于Java和C#，但远高于PHP），且每年都在增加新的特性，但这些特性的发展方向与其他高级语言都不同。

➤ Python：定位是??（将在本章后面详细说明。）最存粹的面向对象语言，抽象程度最高，语法复杂性与Java和C#同级别。

综上所述，也许我们可以认为“Python是一门容易学习的编程语言”，但绝不能认为“Python是一门非常简单的编程语言”。当你阅读完本书后就会发现，Python的语法特性非常多，甚至可以用“包罗万象”来形容。当你学习完了Python之后再学其他的编程语言（C++除外），很可能会发现后者的语法特性还没有Python多。

1-3. 第三个幻觉——学会Python就能找到高薪职位

Python并不是一种能独立撑起一门职业的技术，且在目前需要用到它的职业中都算不上核心技能。这意味着仅掌握Python而不会其他技术，甚至很难找到工作，更不用说找到高薪职位。如果以快速入职IT行业为目标，那么你应该去学习Java、C#、JavaScript或PHP，因为它们才是工业界需求量最大的编程岗位在使用的编程语言（至少在编写本书时）。

Python最大的软肋是执行速度慢。

一位网名为“niklas-heer”的编程者在GitHub上发起了一个编程语言执行速度测试项目[13]，测试方法是用莱布尼兹公式计算圆周率π。根据其测试结果，Python 3.6（以CPython作为解释器）的执行速度排名倒数第二，仅快于Julia，其用时是速度最快的C和Go用时的40倍。当然还有很多其他关于编程语言执行速度的测试，采用不同基准会得到不同结果，但在所有这些测试中官方版Python的排名都在倒数几位。虽然Python 3.11相较于之前的版本速度有了很大提升（相对于Python 3.10提高了约10%～60%），但依然不足以改变“Python执行速度慢于大多数其他编程语言”这一论断。

为什么会这样呢？

首先，对于计算机来说，世界上只有一种编程语言——即由它的硬件所支持的指令形成的机器语言。用任何其他编程语言（包括汇编语言）编写的程序，都要在被翻译成机器语言之后才能执行，而这种翻译有两种方式。第一种方式是“编译（compile）”，即由“编译器（compilers）”将源代码全部翻译成机器指令，保存为二进制的可执行文件，然后被一次性执行。第二种方式是“解释（interpret）”，即由“解释器（interpreters）”将源代码逐句翻译成机器指令并逐句执行，翻译与执行同时进行，不存在可执行文件。

支持编译的编程语言被称为“编译语言（compiled languages）”，用其编写的程序对应的文件习惯上被称为“源文件（source files）”，典型例子包括C、C++、Objective-C、Go和Swift。支持解释的编程语言被称为“解释语言（interpreted languages）”，用其编写的程序对应的文件习惯上被称为“脚本（scripts）”，典型例子包括Java、C#、PHP、JavaScript和Python。汇编语言既不属于编译语言也不属于解释语言，将其翻译成机器语言的过程就是“汇编（assembly）”。除了汇编语言和机器语言本身之外，其他编程语言或者是编译语言，或者是解释语言。

编译语言和解释语言的优缺点正好互补。编译语言支持一次编译多次执行，执行速度快。但每次编译源代码所花时间较长，而任何一点微小的改动都需要重新编译至少一个文件，所以开发效率低。解释语言则支持编写完后直接执行，因此开发效率高，部署灵活，代价则是执行速度慢。

为了弥补执行速度慢的缺点，目前几乎所有主流解释语言都采用了这样的技术：首先将脚本完整地翻译为用“中间语言（intermediate language，IL）”编写的二进制文件，然后在

执行该二进制文件过程中使用“即时（just-in-time，JIT）”编译提速。IL虽然是二进制指令集合，但与平台无关，需要进一步被翻译成机器指令才能执行，因此本质上依然是一种解释语言。而JIT则会分析IL脚本的结构，将其中最常用的组件（例如一个被多次调用的函数）翻译成机器指令并保存起来，以实现优化。IL和JIT都能提高解释语言的执行速度，但JIT的作用远远超过IL的作用，这是因为将解释语言翻译为IL的速度必须足够快，以避免丢失解释语言最重要的优势，因此不可能做太多优化。

Java、C#、PHP和JavaScript都同时使用了IL和JIT。但Python的官方解释器CPython为了确保一项重要的技术优势（将在本章后面详细说明），只通过所谓的“字节码（bytecodes）”来支持IL，而不支持JIT。

其次，不论源文件还是脚本，最终被执行时在逻辑上都可以看作一条机器指令流，因此它们本身可以被视为机器指令的一种抽象存在形式（机器语言被逆翻译为了其他编程语言）。然而我们不能简单地认为该机器指令流只是将源文件或脚本对应的机器指令从头到尾一一拼接起来形成的。事实上，这些机器指令是以“函数（functions）”为单位组织的，这些函数之间通过相互调用形成了该机器指令流。基于这种相互调用，该机器指令流可以在某个位置分成多条支流，而这些支流又在某个位置汇合到主流中。显然，如果这些支流可以被同时执行，将成倍提高程序的执行速度。

事实上，一个程序被执行后对应一个“进程（processes）”，即相应机器指令流与硬件和操作系统分配的资源（虚拟内存空间和I/O接口等）合在一起形成的抽象概念。而机器指令流的主流和支流又都分别被抽象为一个“线程（threads）”，其中主流对应“主线程（main thread）”，总是存在；支流对应的线程则不一定存在。若进程中只有主线程，则该程序是“单线程的（single-threaded）”；反之，该程序是“多线程（multi-threaded）”的。多线程程序相对于单线程程序的优势在于，如果计算机的CPU具有n个核心，则理论上可以同时运行n个线程将执行速度提高n倍，这就是“线程级并行（thread-level parallelism）”的原理。

然而CPython对于线程的支持是不完全的，因为它使用了“全局解释器锁（global interpreter lock，GIL）”，使得在任意时刻CPU中至多有1个核心在执行Python线程。这意味着，官方版Python在逻辑上支持多线程，但却不支持线程级并行。这样做的目的是在保证支持Python语法的前提下简化CPython的实现，使Python线程本身的执行速度更快。CPython的这一设计在单核处理器上算是一种优化，但却无法利用多核处理器的并行计算能力。需要注意的是，线程在执行I/O操作时通常会进入较长时间的阻塞状态，在该状态会自动释放GIL，不再影响其他线程的执行。所以当多线程程序中的线程大部分是I/O密集的时，GIL对执行速度造成的不良影响可以忽略。换句话说，GIL只在由多核处理器执行计算密集型的多线程程序时才会对执行速度有较显著的不利影响。

最后，Python本身的一些语法特性也对执行速度有不利影响。下面举几个例子。如果一种编程语言的变量必须先声明其类型再被访问，则该编程语言是“静态类型的（static-typing）”的；否则该编程语言是“动态类型的（dynamic-typing）”。静态类型语言的优点是可以事先规划如何在内存中为变量分配空间，有利于提高执行速度；缺点则是必须先构

思好需要用到多少个变量，缺乏灵活性。动态类型语言的优缺点则正好相反：优点是编程时想到哪里就写到哪里，需要变量时直接使用即可；缺点则是会损失一定的执行速度。Python是动态类型语言。

在将某类型的变量或常量用作某运算的操作数，或某函数调用的实际参数，而该运算或函数调用其实需要另一种类型时，如果一种编程语言会自动将变量或常量的类型转换为运算或函数调用所需要的类型，则是“弱类型的（weakly typed）”；否则该编程语言是“强类型的（strongly typed）”。强类型语言必须对每次运算和函数调用都进行类型检查。对于编译语言来说这些检查是在编译阶段完成的，只需要做一次，执行时不再需要考虑，所以能够提高执行速度。对于解释语言来说这些检查是在解释阶段完成的，需要做多次（除非JIT将编译结果保存了起来），反而会降低执行速度。Python同时是解释语言和强类型语言。

Python具有反射机制，且一切属性访问都通过反射实现。换句话说，当我们在Python脚本中通过语法特性“obj.attribute”访问一个对象的属性时，实际将执行函数调用object.__getattribute__(obj, "attribute")。因为需要维护函数栈帧，函数调用比直接访问属性速度要明显低得多，而优点则是可以对属性访问进行更多控制。

从上面的讨论可以看出，Python在设计之初的定位就不是为了开发大规模高性能的应用。截止本书被编写时，尚没有纯粹用Python开发大规模高性能应用的成功案例，因此工业界纯粹使用Python开发产品的可能性很小，相应的岗位需求也很少。目前在工业界使用Python的场合主要是五个领域：服务器运维、软件测试、信息安全研究、数据分析和人工智能。在这五个领域中，Python都只是一个方便的工具，核心知识和技能都在别的方面。

值得一提的是，目前Python社区中有一个团体的努力方向是让Python的执行速度能够媲美Java和C#，其工作成果是PyPy这款非官方的Python解释器。PyPy实现了JIT，没有GIL的限制，且提供了更高效的垃圾回收机制，极大地提高了Python脚本的执行速度。niklas-heer的编程语言执行速度测试结果表明，PyPy版Python相对于其他解释语言没有任何劣势。然而由于下节会详细说明的原因，PyPy版Python并不是Python的标准用法。

1-4. Python真正的优势是什么?

经过上面的讨论，自然会产生这样的疑惑：一门语法相当复杂，执行速度又很慢的编程语言，凭什么在诞生30多年后成为全世界最受欢迎的编程语言？这一问题的答案是Python成功坚守住了自身的定位：

1. **真正使编程符合人类的思维方式，且逻辑严谨的高级语言。**
2. **既能支持快速原型开发，又能将原型无缝升级为产品的胶水语言。**

面向对象技术正是为了使编程符合人类的思维方式而被提出的，其核心思想在于实现这样一种抽象层次：编程时需要打交道的一切事物都是所谓的“对象（objects）”，通过“类（classes）”来为对象分类，通过“引用（references）”来操纵对象，而信息的传递与处

理则通过访问对象的“属性（attributes）”来实现。该思想可以被概括为“一切皆对象”。“一切皆对象”是作为Java的广告词被提出的，然而到目前为止，真正实现了“一切皆对象”且同时足够强大的编程语言只有Python。

事实上，面向对象思想早已有之。诞生于1967年的Simula67被认为是面向对象思想的发源地。1971年诞生的Smalltalk、1985年诞生的Eiffel和1986年诞生的Modula-3都对面向对象技术的发展起到了重要的推动作用。然而与C比起来，所有上述编程语言的功能都太弱了，因而都停留于学术界，没能对工业界造成显著的影响。

使面向对象技术在工业界站稳脚跟的编程语言是C++和Objective-C，然而它们都只是给C增添了面向对象支持。前面已经说明，PHP和JavaScript也都不是纯粹的面向对象语言。Go则更加有趣，是对C++的精简，澄清了C++中很多含混不清的概念，例如不再区分指针和引用，然后通过结构体和指针巧妙地实现了面向对象的所有功能。与其说Go是面向对象的，不如说Go是对如何用过程化的语法实现面相对象的功能的诠释。因此在主流编程语言中，相对纯粹的面向对象语言只有Java、C#、Swift和Python。

然而Java、C#和Swift都没有真正实现“一切皆对象”：

- Java中存在8种“基本类型”：byte、short、int、long、float、double、char和boolean。对象的属性引用的数据可以属于这些基本类型，但它们不能被视为对象。
- C#将类型分为“值类型”和“引用类型”，只有属于引用类型的数据能被视为对象，但值类型和引用类型之间可以通过“装箱”和“拆箱”相互转换。
- Swift同样将类型分为“值类型”和“引用类型”，但两者之间不能转换。

这种数据类型二分法的内在逻辑，体现在将数据作为函数调用的实际参数时传递方式上的不同：

- 按值传递：先复制数据以得到一个副本，然后传入该副本。适用于Java中的基本类型，以及C#和Swift中的值类型。
- 按引用传递：直接传入数据在内存中的地址，也就是它的引用。适用于Java中的对象，以及C#和Swift中的引用类型。

这两种不同的传递方式会带来微妙的差异：按值传递的数据在函数体内不论被怎么处理，受影响的都是其副本，函数返回后原数据保持不变；按引用传递的数据在函数体内被处理造成的影响，函数返回后依然会保留。这种差异按照人类的思维习惯是很难理解的，必须从机器的角度去思考，而这就背离了“使编程符合人类的思维方式”这一设计目标。

而在Python中，不存在“基本类型”和“值类型”之类的概念，连最基本的字符串、数字和布尔值也都是对象。更近一步，甚至代码和文件在Python中也都是对象。所以在执行Python脚本时，函数调用中的所有实际参数都按引用传递，不存在上述微妙的差异，因此也就不需要从机器的角度去思考。

没有编程经验的人通常会低估“使编程符合人类的思维方式”的意义，但只要他们亲自去编写一些程序，就会发现这有多么重要。坊间流传着非常多的关于程序员的笑话，而这些笑话在博人一笑的背后，却反映了这样一个事实：长期从机器的角度思考会导致忘记人类原有的思维习惯，而这正是程序员被认为呆板木讷不善交际的根源。（请放过格子衬衫，它们并没犯什么错。）

下面引用一个广为流传的笑话来说明长期从机器的角度思考的程序员将受到什么样的精神戕害（这个例子虽然与面向对象技术无关，但能很好地说明机器和人类思维方式的区别）：程序员的老婆在他上班前嘱咐他说“晚上回来买一个西瓜，如果看到西红柿，就买两个。”于是程序员晚上回家的时候带了两个西瓜。

大概率的，这是一位C程序员。在他听到老婆说的话时，自动将这段人类语言翻译成了如下C代码：

```
int watermelon, tomato

watermelon = 1

if (tomato > 0){
    watermelon = 2
}
```

为什么会这样翻译呢？这是因为C是静态类型语言，变量需要先声明再访问，因此C程序员习惯于先进行全局扫描以确定整个程序需要多少个变量。而全局扫描他老婆所说的人类语言，就会认为“买两个”这一操作应作用于“西瓜”这个变量，而“西红柿”这个变量只是作为判断条件的。

而Python程序员会怎么做呢？Python是动态类型语言，编程是一个顺序过程，想到哪就写到哪。所以当他听到老婆说“晚上回来买一个西瓜”，就会立刻将其翻译成如下代码：

```
watermelon = 1
```

然后听到“如果看到西红柿”，会立刻将其翻译成如下代码：

```
if tomato > 0:
```

最后听到“就买两个”，会立刻将其翻译成如下代码：

```
tomato = 2
```

于是，Python程序员的老婆在晚上如愿以偿地得到了一个西瓜和两个西红柿。

除了使编程符合人类的思维方式外，Python的语法在细节上处处体现了逻辑严谨性。

举例来说，C提供了while、do … while和for三种语句来实现循环，它们的功能其实是重复的（任何一个循环都可以分别用这三种语句来等价实现）；而实现分支的语句则包括if和switch两种，前者涵盖了后者（任何一条switch语句都可以转换成等价的if语句，但反之不然）。由于C的巨大影响力，其他编程语言都或多或少会提供一些功能重叠的语句。

但在Python中分支语句包含if和match，两者功能的重叠部分很少：if语句用于实现典型的分支；match语句用于实现模式匹配。类似的，在Python中循环语句包含while和for，而它们功能的重叠部分也很少：while语句用于实现典型的循环；for语句用于迭代一个可迭代对象。由于坚持让每种功能都只能通过一种语法特性来实现，仅当已有语法特性无法很好实现一种新功能时才添加新语法特性，所以Python的语法明显比其他高级语言的语法的逻辑更严谨。

更有甚者，Python一个著名的特点是通过缩进而非大括号来标明代码块的层次结构。这一语法特性强制编程者使用较好的代码风格。事实上，Python通过一个专门的PEP文档——PEP 8[14]——来规范Python脚本的代码风格。（PEP是“Python增强提案（Python Enhancement Proposals）”的缩写，此类文档用于记录不具有强制性但相对重要的建议性信息。）所有遵循PEP 8的Python脚本将具有几乎完全相同的代码风格，这能极大提高代码的交流效率。

现在你可以真正理解为什么说“Python很容易学”了——这不是因为Python的语法简单，而是因为Python精心设计的语法完全符合人类的思维方式且逻辑严谨。这个优点并不像“执行速度快”、“功能强大”、“语法简单”等等那样显而易见，所以Python的成功之路走得艰辛且漫长。然而随着一代代编程者在教训中积累了经验，他们中越来越多的人发现用Python编程就好像用英文的某种方言来吩咐计算机做工作一样简便，直至最终用脚投票给了Python这一没有任何大公司背书的纯社区推动的开源编程语言。“人生苦短，我用Python。”

当然，如果Python只实现了其定位中的第一点，没有实现第二点，那么它将只适合教学和完成简单任务，相当于高级语言版的Pascal（在中级语言中以适合教学著称）和BASIC（在中级语言中以适合完成简单任务著称）。Python是一门强大的编程语言，绝非仅能开发玩具级别的程序。而这完全要归功于Python的第二点定位，即作为一门胶水语言。Python能够无缝衔接C，这使得Python能够方便地从C的强大中借力。

Python程序的组成单元是“模块（modules）”，大体上可以认为每个模块对应一个文件。而Python的官方解释器CPython提供了一套C API，使我们可以用C或兼容C的编程语言（C++和Objective-C）来开发“扩展模块（extension modules）”。扩展模块在逻辑上与模块是等同的，因此可以替代后者；然而它们必须被编译为动态链接库（Unix中的.so文件，以及Windows中的.dll文件和.pyd文件），因此是编译执行而非解释执行的，执行速度在大

多数情况下能超过任何使用了JIT的解释语言（然而在较特殊的情况下JIT可能速度更快[15]），当以C编写时甚至有可能超过C++、Objective-C、Go和Swift。

这一设计理念堪称是颠覆性的。当追求执行速度时，C是最佳方案，然而代价则是开发效率极低。C++、Objective-C、Go和Swift的理念是牺牲一定的执行速度以换取开发效率的提升。Java和C#的理念则是进一步牺牲执行速度以获取开发效率的更大提升。然而Python的理念则是：先用Python完整地开发出一个逻辑通顺但执行速度慢的原型，再以模块为单位逐步将其机械地翻译为C代码，这样在保证了接近极致的执行速度前提下，开发效率依然能够媲美PHP。

另外，即便将模块替换为了扩展模块，该模块也应该被保留以方便该程序的维护。当发现某个扩展模块存在漏洞，或者需要升级时，不要直接修改扩展模块，而应先将其替换为原来的模块，然后对模块进行修改和升级，最后再改写为新的扩展模块。这样能大幅度提高维护的效率。该模块被称为该扩展模块的“伴随模块（accompanying modules）”。

上面的叙述其实只是极端条件假设下的开发范式——追求最高的执行速度。然而对于实际项目来说，对执行速度的要求通常没那么严格，而开发效率更加重要。因此更常见的开发方式是仅将对执行速度影响很大的模块替换为扩展模块，而其余模块保留不变。这样的程序本质上是Python脚本和C源文件的混合体。Python在人工智能领域的成功应用证明了这种开发方式的可行性：数据处理中最耗时且具有泛用性的部分已经用C和C++编写为了框架（例如PyTorch），而CPython能将这些框架当作扩展模块来使用。另外，Python自身的“标准库（standard library）”其实就是模块和扩展模块的混合，其中扩展模块负责完成最耗时的任务。

需要强调的是这一套C API是CPython独有的，Python的所有非官方实现——包括PyPy——都不支持。扩展模块与JIT不兼容，导致CPython不得不放弃JIT技术。但扩展模块可以通过释放GIL来支持线程级并行，其对程序执行速度的提升不是JIT能够比拟的。简而言之，PyPy能使Python程序的执行速度达到Java和C#的级别，而当采用上述开发方式时，CPython就能使Python和C的混合程序的执行速度超过Java和C#的级别。这就是为什么Python官方对引入JIT技术以及取消GIL兴趣不大。

然而采用上述开发方式的前提是编程者同时掌握C（或C++、Objective-C）和Python，然后还要掌握CPython提供的C API（这同时也意味着需要对CPython的底层运行机制有深入的理解），这是一个不低的技术门槛。这是现实中PyPy依然有大量的用户，而Python 3.11也将提高Python脚本的执行速度作为重点的原因。但从C借力来提供执行速度的理念依然是充满希望的，所以人们又发明了很多技术来推动这一理念的落地。这些技术中必须要提到的是出现于2012年的Nuitka[16]和出现于2008年的Cython[17]。

Nuitka是一种Python打包工具，即能够生成可执行文件，又能够生成扩展模块。在生成扩展模块时，按照如下步骤执行：

- 步骤一：将用户编写的Python脚本都转换为C源代码。
- 步骤二：用C编译器将C源代码编译成动态链接库。

注意这些扩展模块的执行依然需要CPython。而在生成可执行文件时，按照如下步骤执行：

- 步骤一：将用户编写的Python脚本和它们导入的其他模块都转换为C源代码。
- 步骤二：用C编译器将C源代码编译成静态链接库文件。
- 步骤三：将静态链接库、Python脚本导入的扩展模块和CPython的核心libpython一起用链接器链接成可执行文件。

Cython既可以指一种新的编程语言，也可以指处理用这种编程语言写成的源代码的编译器。简单来说，Cython的语法是Python的语法的超集，使我们可以直接调用C函数。而Cython源文件在被编译时会先被转换为C源文件，再生成可执行文件或动态链接库。由于既可以通过Python语法使用提供了Python接口的库，又可以通过C语法使用提供了C接口的库，所以Cython是功能最强大的。Cython不依赖于CPython，仅当被用于编写扩展模块时才需要用到CPython提供的C API。但Cython支持的Python语法与CPython保持一致。

综上所述，我们至少有4种途径提高一个Python脚本的执行速度：

- 用PyPy直接执行Python脚本。
- 用Nuitka直接将Python脚本打包为扩展模块。
- 用Cython结合CPython提供的C API基于该Python脚本编写扩展模块。
- 用C结合CPython提供的C API基于该Python脚本编写扩展模块。

它们的效果依次递增。

由于Nuitka自动生成的C代码很难像手工编写的C代码那样优化，所以Nuitka相对于PyPy的优势并不明显，在较特殊的情况甚至差一些[18]。本书在第17章会详细讨论Nuitka的用法。

用Cython编写的扩展模块的执行速度约为Nuitka生成扩展模块的执行速度的5倍[19]。用Cython直接编写可执行程序，或不作为扩展模块使用的动态链接库时，由于不依赖于libpython，所以其速度可以得到进一步提高。然而Cython源文件不能直接被CPython执行，也不能使用Python标准库中的工具来调试和分析，因此Cython不得不提供一套自己的生态系统。本书不讨论Cython，感兴趣的读者请自行查阅相关资料。

用C结合CPython提供的C API编写的扩展模块依然是执行速度最快的。然而本书也不详细讨论这些C API，以及CPython的底层运行机制。感兴趣的读者请自行查阅Python官方手册的相关部分。

1-5. Python被学术界和工业界采纳的过程

最早被学术界广泛用于教学的编程语言是Pascal，因为它逻辑严谨。然而Pascal不具有开发实用性程序的能力。随着C在工业界的流行，学术界也逐渐用C替代了Pascal以满足工业界的需求，然而事实证明这并不是一个聪明的主意：掌握C的前提是对计算机底层细节有足够的了解，而学习计算机底层细节的前提是对编程足够熟悉，这就形成了悖论。后来面向对象技术被工业界所推崇，学术界的教学语言也从C变成了C++，导致上述悖论带来的恶果进一步加重，教学质量断崖式下跌，师生都苦不堪言。

2008年，MIT的埃里克·格里姆森（Eric Grimson）教授和约翰·谷泰格（John Guttag）教授开创性地在MIT公开课“计算机科学和编程导论”中使用了Python[20]。这一选择对Python的推广产生了巨大的影响，越来越多的大学也开始采用Python作为教学编程语言，越来越多的师生发现了Python兼具逻辑严谨性和实用性，这也导致Python被越来越多的实验室用来开发科研用软件。

在学术界，计算机最主要的应用是分析科研数据，而这需要极高的处理速度和精度。计算机的这一应用领域被称为“科学和工程计算”，最早使用的编程语言是Fortran，后来改成了C。然而不论Fortran还是C都对编程者有很高的要求，并不是每个实验室都能招募到足以胜任该工作的研究者。这一需求催生出了像MATLAB和SAS这样的软件，它们都是由商业软件公司开发的。然而师生们逐渐发现这类商业软件存在价格昂贵、无法定制、不够灵活等问题。

当Python在实验室中逐渐流行后，招募合格的编程者已经不再是一个问题，一些研究者开始考虑摆脱对商业软件公司的依赖。而这导致了下列扩展模块被研发出来并开源化：

- numpy：用于进行高精度矩阵计算。
- scipy：建立在numpy之上，用于进行数值方法、最优化方法、微分方程数值解、信号处理等等领域的计算，相当于MATLAB的核心功能。
- matplotlib：建立在numpy之上，用于进行数据可视化，覆盖了MATLAB的绘图功能。
- pandas：建立在numpy之上，用于进行统计分析，相当于SAS。

值得强调的是numpy是个非常重要的扩展模块，涉及科学和工程计算的扩展模块几乎都建立在其上。

随着人工智能的新一轮热潮，Python也被人工智能研究者采用，而为了弥补其执行速度慢的缺陷则开发了前面提到的人工智能框架。注意这些框架也是建立在numpy之上的。为了方便使用，数据分析和人工智能领域的研究者们将Python、包管理器conda和其他相关软件放在一起形成了一个Python发行版，称为“Anaconda”。使用Anaconda的人通常自称为“数据科学家”。

工业界对Python的采纳其实早于学术界，只不过并没有重度地使用Python。服务器运维、软件测试和信息安全研究都离不开编写一些轻量级的小工具，或者更准确的说是一些shell脚本。早期的shell脚本是利用shell本身的语法，以及像awk和seed这样的工具将shell命令拼合在一起形成的。然而以这种方式编写的shell脚本严重缺乏可移植性，功能也不够强大。后来，Perl成为了编写此类shell脚本的瑞士军刀，直到今天依然有相当多的人在使用。然而Perl类似于PHP，对面向对象的支持是后来添加的，没有Python逻辑清晰。所以上述三个领域的编程者早在2008年之前就开始广泛使用Python代替Perl，而在2008年之后这一趋势更加明显。

随着Python的流行，工业界的Python编程者也不再满足于用Python编写CLI界面的shell脚本。他们希望能直接用Python开发GUI界面的桌面应用，以及Web应用。由于Python在设计之初没有考虑到这些需求，所以他们需要结合使用表1-1中列出的某种框架。

表1-1. 具有代表性的Python框架

范畴	领域	框架
Web开发	**后端**	Django、Flask
	前端	Brython、Flexx
应用开发	**桌面应用**	Kivy、BeeWare、wxPython、PyQt
	移动应用	Kivy、BeeWare

➤ Django和Flask使我们能够将Python当成Web后端脚本语言来使用。它们在底层都基于WSGI协议实现了Web服务器与Python脚本之间的数据交换，但提供的编程组件差异较大。Django与PHP中的Zend Framework类似，是一个重量级的框架，自带大量组件，实现了包括数据库访问、用户认证、权限管理和缓存控制在内的Web后端常用功能。Flask则是一个轻量级框架，仅提供了实现Web后端最核心功能的少量组件，但易于扩展。

➤ Brython和Flexx使我们能够将Python当成Web前端脚本语言来使用。它们本质上都是从Python到JavaScript的转换器，浏览器最终执行的依然是JavaScript脚本。Brython侧重于网页开发，通过brython.js和brython_stdlib.js用JavaScript实现了一个Python解释器，使网页可以直接嵌入Python脚本。注意Brython的执行速度和CPython相当，慢于JavaScript解释器，因此嵌入的Python脚本必须足够简单。Flexx侧重于WebApp开发，通过PScript转换器静态地将Python脚本转换为JavaScript脚本，再将JavaScript脚本嵌入网页。由于从浏览器的角度不知道这一转换过程，因此Python脚本可以很复杂。

➤ 事实上，Python标准库本身提供了tk*系列模块作为使用tcl/tk框架的接口，以支持开发GUI界面的桌面应用。然而tcl/tk本身功能较弱，开发出的GUI较简陋，所以实践中通常不会使用这些模块，而是使用wxPython框架或PyQt框架。wxPython其实是wxWidgets提供的针对Python的接口，PyQt则是Qt提供的针对Python的接口。wxWidgets和Qt都是用C++编写的桌面应用开发库，非常成熟且功能强大。

➤ Kivy和BeeWare侧重于移动应用开发，兼顾桌面应用开发。两者都宣称能保证同一套代码在所有桌面操作系统和移动操作系统上运行。

在本书被编写时，Python社区已经形成了一个蓬勃发展的生态圈。由官方维护的“Python Package Index（PyPI）”收集了Python社区开发出的第三方库。截止2022年底，PyPI中已经有近40万个分发包（在第17章详细讨论），极大扩展了Python的功能，使其几乎能被应用于任何领域。

大量成名已久的软件，例如OpenGL和OpenCV，都提供了Python接口，使自己可以被作为扩展模块来使用，以融入Python的生态圈。主流云计算平台也都提供了针对Python的接口。这些迹象都表明Python已经被工业界承认为一种重要的编程语言。

1-6. 学习Python的推荐路径

阅读到这里，你已经对Python有了足够多的了解，可以判断自己是否有必要学习这门编程语言了。如果你的答案为“是”，那么请继续阅读本书后面的内容。在这一节，我将给出个人推荐的Python学习路径。

学习任何一门编程语言都是一个艰巨且充满痛苦的任务，而掌握正确的学习方法至关重要，因为它可能给你带来数十倍的效率提升。

在给出具体的学习路径之前，请先思考这样一个问题：“我们是如何掌握一种技能的？”

让我们从最简单的生活技能开始。假设你从没学过厨艺，现在想学会用微波炉烹制菜肴。你首先需要请教身边的亲戚朋友中会用微波炉做菜的人，获得一些微波炉和烹饪的基本常识。然后你需要阅读微波炉的说明书，学习微波炉的详细使用方法。当你按照说明书成功烹制出一款菜肴后，就可以宣称已经学会了这项生活技能。随着你之后不断使用微波炉，会积累更多的经验，总结出更多技巧，并对微波炉的原理了解更多。你在整个过程中花费的时间和精力不会太多，否则微波炉就卖不出去了。

下面考虑相对复杂一些的技能——驾驶车辆。这比学会用微波炉烹制菜肴要困难得多，而汽车的使用手册也比微波炉的说明书要厚得多。你依然可以找亲戚朋友教你驾驶的入门技巧，但为了拿到驾驶证必须去专门的驾校进行足够长时间的学习，以通过关于交规和驾驶技

能的测试。在这之后，你依然需要耐心阅读完汽车的说明书，以保证能正确的操控它和养护它，至此你才能自称是一名合格的司机。此后随着驾龄的增长，你同样会积累更多的经验，总结出更多技巧，并对汽车的原理了解更多。你在这个过程中花费的时间和精力，将是学习用微波炉烹制菜肴的几十倍，但你依然觉得它是值得的，因为驾驶车辆比用微波炉烹饪要有用得多。

编程这项技能比驾驶车辆更加复杂，为了学习编程花费的时间和精力将几十倍于学习驾驶车辆，几百上千倍于学习使用微波炉。当然，编程也远比驾驶车辆和用微波炉烹饪要有用得多。然而，学习编程的过程与学习驾驶车辆和学习使用微波炉烹饪的过程没有本质上的不同。你首先需要入门，成为“初学者（beginners）”。然后你需要按照初级 → 中级 → 高级的顺序逐步掌握这门编程语言的语法，直到你能看懂该编程语言的工业标准或官方手册才能自称使用该语言的“专业人士（professionals）”。此后，你需要持续使用该编程语言做项目，在这一过程中一方面你会逐渐理解该编程语言的每种语法特性背后的逻辑和实现细节，成为该语言的“专家（experts）”；另一方面你会逐渐积累编程技巧，并对计算机系统底层细节越来越了解，成为一名编程“大师（masters）”。

➤ 本书并不是针对初学者的，它的目标是让你达到能看懂Python官方手册[21]的程度，成为一名专业的Python程序员。假如你对编程一无所知，那么你需要先上一门入门级的编程课程（例如MIT公开课6.00），或阅读一本入门级的编程书籍（例如埃里克·马西斯（Eric Mathes）编写的《Python编程：从入门到实践》）。

➤ 本书是对Python官方手册的解释和说明，内容涵盖了官方手册中的“教程”和“语言参考”两部分，以及“标准库参考”中的如下组件：“内置函数”、“内置常量”、“内置类型”、“内置异常”、collections.abc、weakref、types、numbers、functools、io、asyncio、typing、pdb、“Python Profilers分析器”、ensurepip、venv、sys、builtins、__main__、warnings、abc和site。从下一章开始，每节的开头都会标明本节内容对应官方手册的哪部分。本书中的全部例子都将严格遵循PEP 8推荐的代码风格，这些列子可以从Gitee下载[22]。

➤ 本书覆盖所有的Python语法，但没有完整讨论Python标准库，因为它过于庞大。在阅读完本书后，你应该具备直接查阅官方手册中关于标准库部分的能力。如果你想获得其他人关于Python标准库的诠释，可以阅读道格·赫尔曼（Doug Hellmann）编写的《Python3标准库》，它涵盖了Python标准库的绝大部分（但信息并不是最新的）。

➤ 本书将篇幅平均地分配给每个语法点：即便某个语法点易于掌握，本书依然会给出足够的说明和例子；而对于那些较难掌握的语法点，本书会略微增加说明的详细程度和例子的数量，但不会太多。卢西亚诺·拉马略（Luciano Ramalho）编写的《流畅的Python》则聚焦于较难掌握的语法点，解释了它们的背后逻辑和实现细节，如果你想成为Python专家，建议阅读这本书。

➤ 成为编程大师的路程非常遥远，需要学习和掌握的知识将成几何倍数增长。这里仅推荐两本我认为特别有意义的书。一本是肯尼思·赖茨（Kenneth Reitz）和塔尼娅·施洛瑟（Tanya Schlusser）编写的开源书《Python编程之美：最佳实践指南》[23]。该书详细讨论了实际项目开发以及编写高质量代码所需的各种技巧。另一本是大卫·比斯利（David Beazley）和布莱恩·琼斯（Brian K. Jones）编写的《Python Cookbook》。它收集了大量经典的Python编程实例，并将它们分门别类以供查询。模仿这本书中的例子是提高Python编程技巧的最好方法。

➤ 如果你想将Python应用于某个特定领域，例如数据分析或人工智能，那么你还需要学习该领域的相关框架本身的运行逻辑，以及这些框架提供给Python的接口。

➤ 真正的编程大师必须具有开发大规模高性能程序的能力，这意味着你必须额外学习C，大学里计算机相关专业的核心课程，以及CPython提供的C API。到了这一步，你甚至能够读懂实现CPython和Python标准库的源代码，达到“RTFSC”的境界。总而言之，想成为编程大师，需要你用一生的时间去追寻。

最后，在编写本书时Python的最新稳定版本是3.11。本书中介绍到Python 3.11特有的语法特性时会特别说明，当你使用Python 3.10或更低版本时不要使用它们。

---

[1] Brandon Vigliarolo (2021-10-11). Python ends C and Java’s 20-year reign atop the TIOBE index. TechRepublic. https://www.techrepublic.com/article/python-ends-c-and-javas-20-year-reign-atop-the-tiobe-index/.

[2] "A Brief Timeline of Python". Python-History. https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.

[3] S. C. Johnson; D. M. Ritchie (1978). "Portability of C Programs and the UNIX System". Bell System Tech. J. 57 (6): 2021-2048. https://www.bell-labs.com/usr/dmr/www/portpap.pdf.

[4] Bjarne Stroustrup (7 March 2010). "Bjarne Stroustrup's FAQ: When was C++ invented?". https://www.stroustrup.com/bs_faq.html#invention.

[5] "JAVASOFT SHIPS JAVA 1.0". https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml.

[6] Rasmus Lerdorf (June 8, 1995). "Announce: Personal Home Page Tools (PHP Tools) version 1.0".

[7] "Netscape and Sun Announce JavaScript, The Open, Cross-Platform Object Scripting Language for Enterprise Networks and The Internet". https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html.

[8] "InfoQ eMag: A Preview of C# 7". https://www.infoq.com/minibooks/emag-c-sharp-preview/.

[9] Robert Griesemer; Rob Pike; Ken Thompson; Ian Taylor; Russ Cox; Jini Kim; Adam Langley. "Hey! Ho! Let's Go!". Google Open Source. https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html.

[10] "Swift Has Reached 1.0". https://developer.apple.com/swift/blog/?id=14.

[11] Caleb Garling. "iPhone Coding Language Now World's Third Most Popular". Wired. July-09-2012. https://www.wired.com/2012/07/apple-objective-c/.

[12] Liam Tung. "Microsoft: We won't evolve Visual Basic programming language but we'll open it to .NET 5". ZDNet. March 13, 2020. https://www.zdnet.com/article/microsoft-we-wont-evolve-visual-basic-programming-language-but-well-open-it-to-net-5/.

[13] "Speed comparison of programming languages". https://github.com/niklas-heer/speed-comparison.

[14] "PEP 8 -- Style Guide for Python Code". https://www.python.org/dev/peps/pep-0008/.

[15] Noah Gibbs. "When is JIT Faster Than A Compiler?" Shopify Engineering. https://shopify.engineering/when-jit-faster-than-compiler.

[16] "Nuitka the Python Compiler". https://nuitka.net/.

[17] "Cython C-Extensions for Python". https://cython.org/.

[18] "Are the Nuitka programs faster?". https://pybenchmarks.org/u64q/benchmark.php?test=all&lang=nuitka&lang2=pypy&data=u64q.

[19] Cleaopatra. "Nuitka vs Cython vs PyPy: Know the Differences Between the Three". https://eduwyre.com/article/nuitka-vs-cython-vs-pypy.

[20] MIT OpenCourseware. "Introduction to Computer Science and Programming". https://ocw.mit.edu/courses/6-00-introduction-to-computer-science-and-programming-fall-2008/.

[21] Python 3.11 documentation. https://docs.python.org/zh-cn/3.11/.

[22] "wwy_gladiolus/iopm3.11". https://gitee.com/wwy_gladiolus/iopm3.11.

[23] "The Hitchhiker's Guide to Python". https://docs.python-guide.org/.