第 1 章　Python 编程基础

1.1 编程语言是什么

其实，程序指的就是一系列指令，用来告诉计算机做什么，而编写程序的关键在于，我们需要用计算机可以理解的语言来提供这些指令。

虽然借助 Siri（Apple）、Google Now（Android）、Cortana（Microsoft）等技术，我们可以使用汉语直接告诉计算机做什么，比如“Siri，打开酷狗音乐”，但使用过这些系统的读者都知道，它尚未完全成熟，再加上我们语言充满了模糊和不精确因素，使得设计一个完全理解人类语言的计算机程序，仍然是一个有待解决的问题。

为了有效避开所有影响给计算机传递指令的因素，计算机科学家设计了一些符号，这些符号各有其含义，且之间无二义性，通常称它们为编程语言。编程语言中的每个结构，都有固定的使用格式（称为语法）以及精确的含义（称为语义）。换句话说，编程语言指定了成套的规则，用来编写计算机可以理解的指令。习惯上，我们将这一条条指令称为计算机代码，而用编程语言来编写算法的过程称为编码。

本教程要讲解的 Python 就是一种编程语言，除此之外，你肯定也听说过其他一些编程语言，如 C、C++、Java、Ruby 等。至今，计算机科学家已经开发了成百上千种编程语言，且随着时间演变，这些编程语言又产生了多个不同的版本。但无论是哪个编程语言，也无论有多少个版本，虽然它们在细节上可能有所不同，无疑它们都有着固定的、无二义性的语法和语义。

以上提到的编程语言，都是高级计算机语言，设计它们的目的是为了方便程序员理解和使用。但严格来说，计算机硬件只能理解一种非常低级的编程语言，称为机器语言。

比如说，让计算机对 2 个数做求和操作，那么 CPU 可能要执行以下指令：

1. 将位于内存空间位置在 2001 的数加载到 CPU 中；
2. 再将位于内存空间位置在 2002 的数也加载到 CPU 中；
3. 在 CPU 中，对这 2 个数做求和操作；
4. 将结果存储在位置为 2003 的内存空间。

可以看到，对 2 个数执行求和操作需要做这么多工作，且这还只是笼统地描述，实际会更加复杂。

而使用 Python 这样的高级语言，对 2 个数求和可以很自然地用 c = a + b 表示，但由此带来的问题是，我们需要设计一种方法，将高级语言翻译成计算机可以执行的机器语言，有两种方法可以实现，分别是使用编译器和解释器。

使用编译器将自身等效转换成机器语言的高级语言，通常称为编译型语言；而使用解释器将自身转换成机器语言的高级语言，称为解释型语言，Python 就是解释型编程语言的一种。

> 关于编译型语言和解释型语言的含义和区别，后续章节会进行详细介绍。

1.2 编译型语言和解释型语言的区别

我们编写的源代码是人类语言，我们自己能够轻松理解；但是对于计算机硬件（CPU），源代码就是天书，根本无法执行，计算机只能识别某些特定的二进制指令，在程序真正运行之前必须将源代码转换成二进制指令。

所谓的二进制指令，也就是机器码，是 CPU 能够识别的硬件层面的“代码”，简陋的硬件（比如古老的单片机）只能使用几十个指令，强大的硬件（PC 和智能手机）能使用成百上千个指令。

然而，究竟在什么时候将源代码转换成二进制指令呢？不同的编程语言有不同的规定：

- 有的编程语言要求必须提前将所有源代码一次性转换成二进制指令，也就是生成一个可执行程序（Windows 下的 .exe），比如 C 语言、C++、Golang、Pascal（Delphi）、汇编等，这种编程语言称为编译型语言，使用的转换工具称为编译器。
- 有的编程语言可以一边执行一边转换，需要哪些源代码就转换哪些源代码，不会生成可执行程序，比如 Python、JavaScript、PHP、Shell、MATLAB 等，这种编程语言称为解释型语言，使用的转换工具称为解释器。

简单理解，编译器就是一个“翻译工具”，类似于将中文翻译成英文、将英文翻译成俄文。但是，翻译源代码是一个复杂的过程，大致包括词法分析、语法分析、语义分析、性能优化、生成可执行文件等五个步骤，期间涉及到复杂的算法和硬件架构。解释器与此类似，有兴趣的读者请参考《编译原理》一书，本文不再赘述。

Java 和 C# 是一种比较奇葩的存在，它们是半编译半解释型的语言，源代码需要先转换成一种中间文件（字节码文件），然后再将中间文件拿到虚拟机中执行。Java 引领了这种风潮，它的初衷是在跨平台的同时兼顾执行效率；C# 是后来的跟随者，但是 C# 一直止步于 Windows 平台，在其它平台鲜有作为。

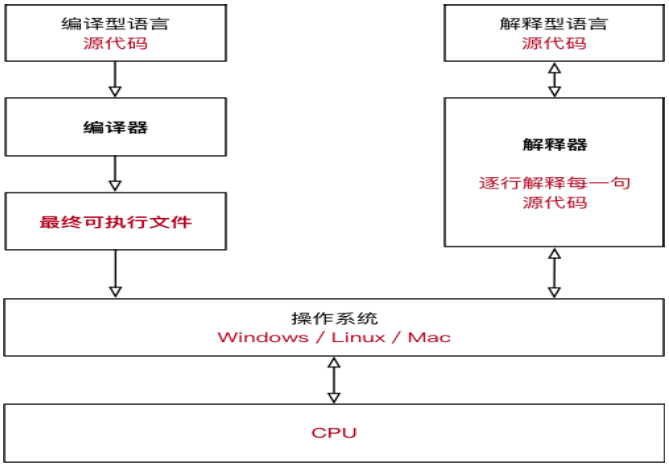


图 1 编译型语言和解释型语言的执行流程

那么，编译型语言和解释型语言各有什么特点呢？它们之间有什么区别？

编译型语言

对于编译型语言，开发完成以后需要将所有的源代码都转换成可执行程序，比如 Windows 下的.exe 文件，可执行程序里面包含的就是机器码。只要我们拥有可执行程序，就可以随时运行，不用再重新编译了，也就是“一次编译，无限次运行”。

在运行的时候，我们只需要编译生成的可执行程序，不再需要源代码和编译器了，所以说编译型语言可以脱离

开发环境运行。

编译型语言一般是不能跨平台的，也就是不能在不同的操作系统之间随意切换。

编译型语言不能跨平台表现在两个方面：

1) 可执行程序不能跨平台

可执行程序不能跨平台很容易理解，因为不同操作系统对可执行文件的内部结构有着截然不同的要求，彼此之间也不能兼容。不能跨平台是天经地义，能跨平台反而才是奇葩。

比如，不能将 Windows 下的可执行程序拿到 Linux 下使用，也不能将 Linux 下的可执行程序拿到 Mac OS 下使用（虽然它们都是类 Unix 系统）。

另外，相同操作系统的不同版本之间也不一定兼容，比如不能将 x64 程序（Windows 64 位程序）拿到 x86 平台（Windows 32 位平台）下运行。但是反之一般可行，因为 64 位 Windows 对 32 位程序作了很好的兼容性处理。

2) 源代码不能跨平台

不同平台支持的函数、类型、变量等都可能不同，基于某个平台编写的源代码一般不能拿到另一个平台下编译。我们以 C 语言为例来说明。

【实例 1】在 C 语言中要想让程序暂停可以使用“睡眠”函数，在 Windows 平台下该函数是 Sleep()，在 Linux 平台下该函数是 sleep()，首字母大小写不同。其次，Sleep() 的参数是毫秒，sleep() 的参数是秒，单位也不一样。

以上两个原因导致使用暂停功能的 C 语言程序不能跨平台，除非在代码层面做出兼容性处理，非常麻烦。

【实例 2】虽然不同平台的 C 语言都支持 long 类型，但是不同平台的 long 的长度却不同，例如，Windows 64 位平台下的 long 占用 4 个字节，Linux 64 位平台下的 long 占用 8 个字节。

我们在 Linux 64 位平台下编写代码时，将 0x2f1e4ad23 赋值给 long 类型的变量是完全没有问题的，但是这样的赋值在 Windows 平台下就会导致数值溢出，让程序产生错误的运行结果。

让人苦恼的，这样的错误一般不容易察觉，因为编译器不会报错，我们也记不住不同类型的取值范围。

解释型语言

对于解释型语言，每次执行程序都需要一边转换一边执行，用到哪些源代码就将哪些源代码转换成机器码，用不到的不进行任何处理。每次执行程序时可能使用不同的功能，这个时候需要转换的源代码也不一样。

因为每次执行程序都需要重新转换源代码，所以解释型语言的执行效率天生就低于编译型语言，甚至存在数量级的差距。计算机的一些底层功能，或者关键算法，一般都使用 C/C++ 实现，只有在应用层面（比如网站开发、批处理、小工具等）才会使用解释型语言。

在运行解释型语言的时候，我们始终都需要源代码和解释器，所以说它无法脱离开发环境。

当我们说“下载一个程序（软件）”时，不同类型的语言有不同的含义：

- 对于编译型语言，我们下载到的是可执行文件，源代码被作者保留，所以编译型语言的程序一般是闭源的。
- 对于解释型语言，我们下载到的是所有的源代码，因为作者不给源代码就没法运行，所以解释型语言的程序一般是开源的。

相比于编译型语言，解释型语言几乎都能跨平台，“一次编写，到处运行”是真是存在的，而且比比皆是。那么，为什么解释型语言就能快平台呢？

这一切都要归功于解释器！

我们所说的跨平台，是指源代码跨平台，而不是解释器跨平台。解释器用来将源代码转换成机器码，它就是一个可执行程序，是绝对不能跨平台的。

官方需要针对不同的平台开发不同的解释器，这些解释器必须要能够遵守同样的语法，识别同样的函数，完成同样的功能，只有这样，同样的代码在不同平台的执行结果才是相同的。

你看，解释型语言之所以能够跨平台，是因为有了解释器这个中间层。在不同的平台下，解释器会将相同的源代码转换成不同的机器码，解释器帮助我们屏蔽了不同平台之间的差异。

关于 Python

Python 属于典型的解释型语言，所以运行 Python 程序需要解释器的支持，只要你在不同的平台安装了不同的解释器，你的代码就可以随时运行，不用担心任何兼容性问题，真正的“一次编写，到处运行”。

Python 几乎支持所有常见的平台，比如 Linux、Windows、Mac OS、Android、FreeBSD、Solaris、PocketPC 等，你所写的 Python 代码无需修改就能在这些平台上正确运行。也就是说，Python 的可移植性是很强的。

总结

我们将编译型语言和解释型语言的差异总结为下表：

类型	原理	优点	缺点
编译型语言	通过专门的编译器，将所有源代码一次性转换成特定平台（Windows、Linux 等）执行的机器码（以可执行文件的形式存在）。	编译一次后，脱离了编译器也可以运行，并且运行效率高。	可移植性差，不够灵活。
解释型语言	由专门的解释器，根据需要将部分源代码临时转换成特定平台的机器码。	跨平台性好，通过不同的解释器，将相同的源代码解释成不同平台下的机器码。	一边执行一边转换，效率很低。

1.3 Python 是什么，Python 简介

编程语言有“高低”之分，而高级语言又有很多种，比如 C++、Java、C#、PHP、JavaScript 等，Python 也是其中之一。从本节开始，我们将正式开始学习 Python 这门高级编程语言，但是在此之前，我们有必要先讨论一下“Python 是什么”。

Python 英文原意为“蟒蛇”，直到 1989 年荷兰人 Guido van Rossum（简称 Guido）发明了一种面向对象的解释型编程语言，并将其命名为 Python，才赋予了它表示一门编程语言的含义。

> 我们将在《编译型语言和解释型语言的区别》一文中讲解什么是解释型语言。

图 1 Python 的标志（Logo）

> 说道 Python，它的诞生是极具戏曲性的，据 Guido 的自述记载，Python 语言是他在圣诞节期间为了打发时间开发出来的，之所以会选择 Python 作为该编程语言的名字，是因为他是一个叫 Monty Python 戏剧团体的忠实粉丝。

Python 语言是在 ABC 教学语言的基础上发展来的；遗憾的是，ABC 语言虽然非常强大，但却没有普及应用，Guido 认为是它不开放导致的。

基于这个考虑，Guido 在开发 Python 时，不仅为其添加了很多 ABC 没有的功能，还为其设计了各种丰富而强大的库，利用这些 Python 库，程序员可以把使用其它语言制作的各种模块（尤其是 C 语言和 C++）很轻松地联结在一起，因此 Python 又常被称为“胶水”语言。

这里的库和模块，简单理解就是一个个的源文件，每个文件中都包含可实现各种功能的方法（也可称为函数）。

从整体上看，Python 语言最大的特点就是简单，该特点主要体现在以下 2 个方面：

- Python 语言的语法非常简洁明了，即便是非软件专业的初学者，也很容易上手。
- 和其它编程语言相比，实现同一个功能，Python 语言的实现代码往往是最短的。

对于 Python，网络上流传着“人生苦短，我用 Python”的说法。

因此，看似 Python 是“不经意间”开发出来的，但丝毫不比其它编程语言差。事实也是如此，自 1991 年 Python 第一个公开发行版问世后：

- 2004 年起 Python 的使用率呈线性增长，不断受到编程者的欢迎和喜爱；
- 2010 年，Python 荣膺 TIOBE 2010 年度语言桂冠；
- 2017 年，IEEE Spectrum 发布的 2017 年度编程语言排行榜中，Python 位居第 1 位。

直至现在（2019 年 12 月份），根据 TIOBE 排行榜的显示，Python 也居于第 3 位，且有继续提升的态势（如表 2 所示）。

表 2 TIOBE 2019 年 12 月份编程语言排行榜（前 20 名）

2019 年 12 月	2018 年 12 月	编程语言	市场份额	变化
1	1	Java	17.253%	+1.32%
2	2	C	16.086%	+1.80%
3	3	Python	10.308%	+1.93%
4	4	C++	6.196%	-1.37%
5	6	C#	4.801%	+1.35%
6	5	Visual Basic .NET	4.743%	-2.38%
7	7	JavaScript	2.090%	-0.97%
8	8	PHP	2.048%	-0.39%
9	9	SQL	1.843%	-0.34%
10	14	Swift	1.490%	+0.27%
11	17	Ruby	1.314%	+0.21%
12	11	Delphi/Object Pascal	1.280%	-0.12%
13	10	Objective-C	1.204%	-0.27%
14	12	Assembly language	1.067%	-0.30%
15	15	Go	0.995%	-0.19%
16	16	R	0.995%	-0.12%
17	13	MATLAB	0.986%	-0.30%
18	25	D	0.930%	+0.42%
19	19	Visual Basic	0.929%	-0.05%
20	18	Perl	0.899%	-0.11%

显然，Python 已经将 C++ 语言甩在了后边，直逼 C 语言和 Java，而且未来有可能超越它们，成为编程语言排行榜冠军。

我们不妨再看一下 Python 历年来的市场份额变化曲线，Python 的未来大势可期。

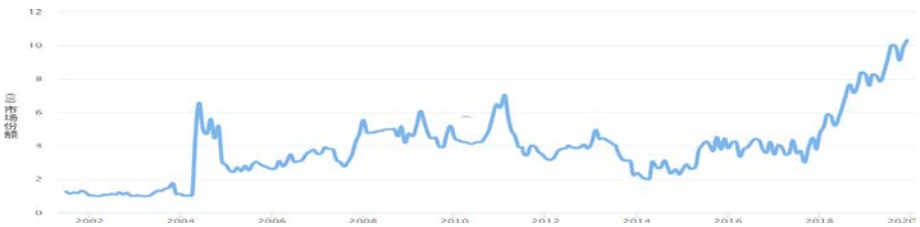


图 2 Python 历年来市场份额变化曲线

错过了 C/C++ 的 PC 时代，又错过了 Java 的互联网和智能手机时代，你还想错过 Python 的人工智能和大数据时代吗？Python 正位于软件产业的第四次风口之上，把握好风口，你就能飞起来。

1.4 Python 的特点（优点和缺点）

Python 是一种面向对象的、解释型的、通用的、开源的脚本编程语言，它之所以非常流行，我认为主要有三点原因：

- Python 简单易用，学习成本低，看起来非常优雅干净；
- Python 标准库和第三库众多，功能强大，既可以开发小工具，也可以开发企业级应用；
- Python 站在了人工智能和大数据的风口上，站在风口上，猪都能飞起来。

举个简单的例子来说明一下 Python 的简单。比如要实现某个功能，C 语言可能需要 100 行代码，而 Python 可能只需要几行代码，因为 C 语言什么都要得从头开始，而 Python 已经内置了很多常见功能，我们只需要导入包，然后调用一个函数即可。

简单就是 Python 的巨大魅力之一，是它的杀手锏，用惯了 Python 再用 C 语言简直不能忍受。

本文就来汇总一下 Python 的特性，综合对比一下它的优缺点。

Python 的优点

1) 语法简单

和传统的 C/C++、Java、C# 等语言相比，Python 对代码格式的要求没有那么严格，这种宽松使得用户在编写代码时比较舒服，不用在细枝末节上花费太多精力。我来举两个典型的例子：

- Python 不要求在每个语句的最后写分号，当然写上也没错；
- 定义变量时不需要指明类型，甚至可以给同一个变量赋值不同类型的数据。

这两点也是 PHP、JavaScript、MATLAB 等常见脚本语言都具备的特性。

Python 是一种代表极简主义的编程语言，阅读一段排版优美的 Python 代码，就像在阅读一个英文段落，非常贴近人类语言，所以人们常说，Python 是一种具有伪代码特质的编程语言。

> 伪代码（Pseudo Code）是一种算法描述语言，它介于自然语言和编程语言之间，使用伪代码的目的是为了使被描述的算法可以容易地以任何一种编程语言（Pascal，C，Java，etc）实现。因此，伪代码必须结构清晰、代码简单、可读性好，并且类似自然语言。
>
> 如果你学过数据结构，阅读过严蔚敏的书籍，那你一定知道什么是伪代码。

为什么说简单就是杀手锏？一旦简单了，一件事情就会变得很纯粹；我们在开发 Python 程序时，可以专注于解决问题本身，而不用顾虑语法的细枝末节。在简单的环境中做一件纯粹的事情，那简直是一种享受。

2) Python 是开源的

开源，也即开放源代码，意思是所有用户都可以看到源代码。

Python 的开源体现在两方面：

① 程序员使用 Python 编写的代码是开源的。

比如我们开发了一个 BBS 系统，放在互联网上让用户下载，那么用户下载到的就是该系统的所有源代码，并且可以随意修改。这也是解释型语言本身的特性，想要运行程序就必须有源代码。

② Python 解释器和模块是开源的。

官方将 Python 解释器和模块的代码开源，是希望所有 Python 用户都参与进来，一起改进 Python 的性能，弥补 Python 的漏洞，代码被研究的越多就越健壮。

这个世界上总有那么一小撮人，他们或者不慕名利，或者为了达到某种目的，会不断地加强和改善 Python。千万不要认为所有人都是只图眼前利益的，总有一些精英会放长线钓大鱼，总有一些极客会做一些炫酷的事情。

3) Python 是免费的

开源并不等于免费，开源软件和免费软件是两个概念，只不过大多数的开源软件也是免费软件；Python 就是这样一种语言，它既开源又免费。

> 如果你想区分开源和免费的概念，请猛击：开源就等于免费吗？用事实来说话

用户使用 Python 进行开发或者发布自己的程序，不需要支付任何费用，也不用担心版权问题，即使作为商业用途，Python 也是免费的。

4) Python 是高级语言

这里所说的高级，是指 Python 封装较深，屏蔽了很多底层细节，比如 Python 会自动管理内存（需要时自动分配，不需要时自动释放）。

高级语言的优点是使用方便，不用顾虑细枝末节；缺点是容易让人浅尝辄止，知其然不知其所以然。

5) Python 是解释型语言，能跨平台

解释型语言一般都是跨平台的（可移植性好），Python 也不例外，我们已经在《编译型语言和解释型语言的区别》中进行了讲解，这里不再赘述。

5) Python 是面向对象的编程语言

面向对象是现代编程语言一般都具备的特性，否则在开发中大型程序时会捉襟见肘。

Python 支持面向对象，但它不强制使用面向对象。Java 是典型的面向对象的编程语言，但是它强制必须以类和对象的形式来组织代码。

6) Python 功能强大（模块众多）

Python 的模块众多，基本实现了所有的常见的功能，从简单的字符串处理，到复杂的 3D 图形绘制，借助 Python 模块都可以轻松完成。

Python 社区发展良好，除了 Python 官方提供的核心模块，很多第三方机构也会参与进来开发模块，这其中就有 Google、Facebook、Microsoft 等软件巨头。即使是一些小众的功能，Python 往往也有对应的开源模块，甚至有可能不止一个模块。

7) Python 可扩展性强

Python 的可扩展性体现在它的模块，Python 具有脚本语言中最丰富和强大的类库，这些类库覆盖了文件 I/O、GUI、网络编程、数据库访问、文本操作等绝大部分应用场景。

这些类库的底层代码不一定都是 Python，还有很多 C/C++ 的身影。当需要一段关键代码运行速度更快时，就可以使用 C/C++ 语言实现，然后在 Python 中调用它们。Python 能把其它语言“粘”在一起，所以被称为“胶水语言”。

Python 依靠其良好的扩展性，在一定程度上弥补了运行效率慢的缺点。

Python 的缺点

除了上面提到的各种优点，Python 也是有缺点的。

1) 运行速度慢

运行速度慢是解释型语言的通病，Python 也不例外。

Python 速度慢不仅仅是因为一边运行一边“翻译”源代码，还因为 Python 是高级语言，屏蔽了很多底层细节。这个代价也是很大的，Python 要多做很多工作，有些工作是很消耗资源的，比如管理内存。

Python 的运行速度几乎是最慢的，不但远远慢于 C/C++，还慢于 Java。

但是速度慢的缺点往往也不会带来什么大问题。首先是计算机的硬件速度运来越快，多花钱就可以堆出高性能的硬件，硬件性能的提升可以弥补软件性能的不足。

其次是有些应用场景可以容忍速度慢，比如网站，用户打开一个网页的大部分时间是在等待网络请求，而不是等待服务器执行网页程序。服务器花 1ms 执行程序，和花 20ms 执行程序，对用户来说是毫无感觉的，因为网络连接时间往往需要 500ms 甚至 2000ms。

2) 代码加密困难

不像编译型语言的源代码会被编译成可执行程序，Python 是直接运行源代码，因此对源代码加密比较困难。

开源是软件产业的大趋势，传统程序员需要转变观念。

1.5 学 Python，不需要有编程基础！

“编程零基础，可以学习 Python 吗”，这是很多初学者经常问我的一个问题。当然，在计算机方面的基础越好，对学习任何一门新的编程语言越有利。但如果你在编程语言的学习上属于零基础，也不用担心，因为无论用哪门语言作为学习编程的入门语言，总是要有一个开始。

就我个人的观点，Python 作为学习编程的入门语言是再合适不过的。凡是在大学计算机专业学习过 C 语言的同学都感同身受，认为 C 语言不是很好的入门语言，很多曾经立志学习编程的读者，在学习了 C 语言之后，就决心不再学习编程。因此，是否学会 C 语言，好像成为了进入编程行业的筛选标准。

但是，如果将 Python 作为编程入门语言，就不会出现类似 C 语言的那些窘境问题。目前，逐渐有高校开始使用 Python 作为软件专业大学生（甚至也包含非软件专业）的入门编程语言。

本教程始终贯彻的思想就是，零基础也能学 Python，教程的目标就是和初学者一起，从零基础开始学习 Python。因此，编程零基础的你，无需犹豫，尽管放胆来学。

除此之外，很多初学者还面临这样一个问题，那就是教程已经学完啦，教程中的代码也都已经亲自编写并运行通过了，但还是不知道如何开发一个真正的应用程序，面对问题还是不知道如何下手解决。

如果你深有同感，只能说明你缺乏练习，代码编辑量太少。从编程零基础到专业程序员的过程，除了学习更多的基础知识，更要刻意地去培养自己的编程思维，这没有捷径，只有靠不断积累自己的代码量。

当然，增加代码量并不是要我们去盲目地编写代码，如果找不到增加代码量的方向，可以从阅读别人的代码开始。需要注意的是，在阅读他人编写的代码时，要边阅读边思考，多问几个为什么，例如代码为什么要这么写，有什么意图，有没有更简单的方法可以实现等等，必要情况下还可以给代码进行必要的注释。不仅如此，在完全理解他人代码的前提下，还可以试图对代码做修改，实现一些自己的想法。做到这些，才能说明你将别人的代码消化吸收了。

初学者在写代码或者调试代码的过程中，肯定会遇到很多问题，怎么办呢？最好的方法就是借助网络寻找答案，看看类似的问题别人是如何解决的，千万不要总是局限在自己的思维范围中。在这里，给大家推荐一个专门针对编程答疑解惑的网站 Stack OverFlow。

1.6 Python 能干什么，Python 的应用领域

Python 作为一种功能强大的编程语言，因其简单易学而受到很多开发者的青睐。那么，Python 的应用领域有哪些呢？

Python 的应用领域非常广泛，几乎所有大中型互联网企业都在使用 Python 完成各种各样的任务，例如国外的 Google、Youtube、Dropbox，国内的百度、新浪、搜狐、腾讯、阿里、网易、淘宝、知乎、豆瓣、汽车之家、美团等等。

概括起来，Python 的应用领域主要有如下几个。

Web 应用开发

Python 经常被用于 Web 开发，尽管目前 PHP、JS 依然是 Web 开发的主流语言，但 Python 上升势头更猛劲。尤其随着 Python 的 Web 开发框架逐渐成熟（比如 Django、flask、TurboGears、web2py 等等），程序员可以更轻松地开发和管理复杂的 Web 程序。

例如，通过 mod_wsgi 模块，Apache 可以运行用 Python 编写的 Web 程序。Python 定义了 WSGI 标准应用接口来协调 HTTP 服务器与基于 Python 的 Web 程序之间的通信。

举个最直观的例子，全球最大的搜索引擎 Google，在其网络搜索系统中就广泛使用 Python 语言。另外，我们经常访问的集电影、读书、音乐于一体的豆瓣网（如图 1 所示），也是使用 Python 实现的。

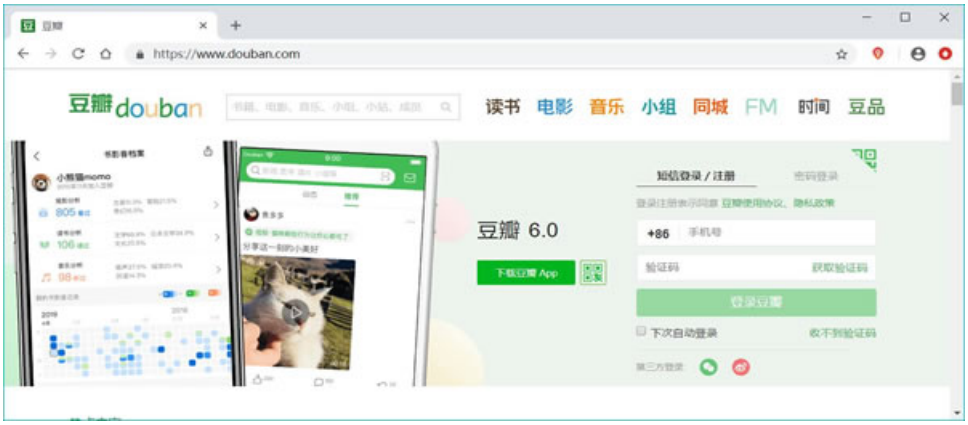


图 1 用 Python 实现的豆瓣网

不仅如此，全球最大的视频网站 Youtube 以及 Dropbox（一款网络文件同步工具）也都是用 Python 开发的。

自动化运维

很多操作系统中，Python 是标准的系统组件，大多数 Linux 发行版以及 NetBSD、OpenBSD 和 Mac OS X 都集成了 Python，可以在终端下直接运行 Python。

有一些 Linux 发行版的安装器使用 Python 语言编写，例如 Ubuntu 的 Ubiquity 安装器、Red Hat Linux 和 Fedora 的 Anaconda 安装器等等。

另外，Python 标准库中包含了多个可用来调用操作系统功能的库。例如，通过 pywin32 这个软件包，我们能访问 Windows 的 COM 服务以及其他 Windows API；使用 IronPython，我们能够直接调用 .Net Framework。

通常情况下，Python 编写的系统管理脚本，无论是可读性，还是性能、代码重用度以及扩展性方面，都优于普通的 shell 脚本。

人工智能领域

人工智能是项目非常火的一个研究方向，如果要评选当前最热、工资最高的 IT 职位，那么人工智能领域的工程师最有话语权。而 Python 在人工智能领域内的机器学习、神经网络、深度学习等方面，都是主流的编程语言。

可以这么说，基于大数据分析和深度学习发展而来的人工智能，其本质上已经无法离开 Python 的支持了，原因至少有以下几点：

1. 目前世界上优秀的人工智能学习框架，比如 Google 的 TransorFlow（神经网络框架）、FaceBook 的 PyTorch（神经网络框架）以及开源社区的 Karas 神经网络库等，都是用 Python 实现的；
2. 微软的 CNTK（认知工具包）也完全支持 Python，并且该公司开发的 VS Code，也已经把 Python 作为第一级语言进行支持。
3. Python 擅长进行科学计算和数据分析，支持各种数学运算，可以绘制出更高质量的 2D 和 3D 图像。

VS Code 是微软推出的一款代码编辑工具（IDE），有关它的下载、安装和使用，后续章节会做详细介绍。
总之，AI 时代的来临，使得 Python 从众多编程语言中脱颖而出，Python 作为 AI 时代头牌语言的位置，基本无人可撼动！

网路爬虫

Python 语言很早就用来编写网络爬虫。Google 等搜索引擎公司大量地使用 Python 语言编写网络爬虫。

从技术层面上将，Python 提供有很多服务于编写网络爬虫的工具，例如 urllib、Selenium 和 BeautifulSoup 等，还提供了一个网络爬虫框架 Scrapy。

科学计算

自 1997 年，NASA 就大量使用 Python 进行各种复杂的科学运算。

并且，和其它解释型语言（如 shell、js、PHP）相比，Python 在数据分析、可视化方面有相当完善和优秀的

库，例如 NumPy、SciPy、Matplotlib、pandas 等，这可以满足 Python 程序员编写科学计算程序。

游戏开发

很多游戏使用 C++ 编写图形显示等高性能模块，而使用 Python 或 Lua 编写游戏的逻辑。和 Python 相比，Lua 的功能更简单，体积更小；而 Python 则支持更多的特性和数据类型。

比如说，国际上指明的游戏 Sid Meier's Civilization（文明，如图 2 所示）就是使用 Python 实现的。

图 2 Python 开发的游戏

除此之外，Python 可以直接调用 Open GL 实现 3D 绘制，这是高性能游戏引擎的技术基础。事实上，有很多 Python 语言实现的游戏引擎，例如 Pygame、Pyglet 以及 Cocos 2d 等。

以上也仅是介绍了 Python 应用领域的“冰山一角”，例如，还可以利用 Pygame 进行游戏编程；用 PIL 和其他的一些工具进行图像处理；用 PyRo 工具包进行机器人控制编程，等等。有兴趣的读者，可自行搜索资料进行详细了解。

1.7 怎样学习 Python 才能成为高手？

在学习过程中，很多小伙伴经常抱怨，计算机编程语言种类太多，根本学不过来，可能用了 Java 很多年，突然最近的项目需要用 Python，就感到不知所措，压力山大。

再举个例子，Facebook 的主流语言是 Hack（PHP 的进化版本），但对于刚刚入职的工程师而言，100 个里至少有 95 个之前根本没有用过 Hack 或者 PHP。不过，这些人上手都特别快，基本上只需要 1~2 周，日常编程就变得毫无压力。

那么，他们是怎么做到的呢？

编程语言都是相通的

首先，如果你具有一定的编程基础，接触一门新的编程语言时会发现，不同的编程语言之间其实是相通的，因为编程语言本就是人类控制计算机的一系列指令，即便是不同的编程语言，它们在语法规则方面也大同小异。

因此，在原有编程基础上再学习一门新的编程语言，并没有那么难。学习过程中，首先要做到的就是明确区别。例如，学习 Python 的条件和循环语句时，可以比对 C 语言或者 C++ 语言的语法是怎样的；学习 Python 中的字符串相加时，可以对比 Java 语言中字符串相加的语法。

除了能够明确区分各编程语言的不同点，还要能将它们联系起来灵活运用。比如说，Python 语言的优势是擅长数据分析，因此它广泛应用于人工智能（AI）、机器学习等领域（例如机器学习用到的 TensorFlow 框架就是用 Python 写的），但是涉及到底层的矩阵运算等操作时，就需要依赖 C++ 语言，因为它的速度快，运行效率更高。

事实上，很多公司都是根据不同的需求选择不同的编程语言进行开发。毕竟，哪怕只是几十到几百毫秒的速度差距，对用户体验来说都是决定性的。

Python 对初学者很友好

如果 Python 是你接触的第一门编程语言，那也不必担心。我们知道，虽然同为人机交互的桥梁，但 Python 比 C++、Java 等，语言更简洁，也更接近英语，对初学者很友好，这也是 Python 语言的一个显著特点。

对于初学者来说，要做的就是专注于 Python 这一门语言，明确学习的重点，把握好学习节奏，由浅入深循序渐进地学习。根据本人多年的学习工作经验，把编程语言的学习分为以下 3 步，无论你是否有编程基础，都可以对照着来做。

1) 多实践，积累代码量

任何一门编程语言，其囊括的知识面都是非常广泛的，从基本的变量赋值、条件循环、到文件操作、并发编程等，千万不要等到把所有知识点都学完了才开始练习，因为到那时你会发现，前面好不容易记住的知识都忘记了。

学习编程，是十分讲究实战的，没有捷径可走，越早练习，练得越多越勤，学习效果就越好。

2) 时刻注意代码规范

学习编程语言，在追求快和高效的同时，每一种编程语言都有必要的编码规范，一定不能忽略。

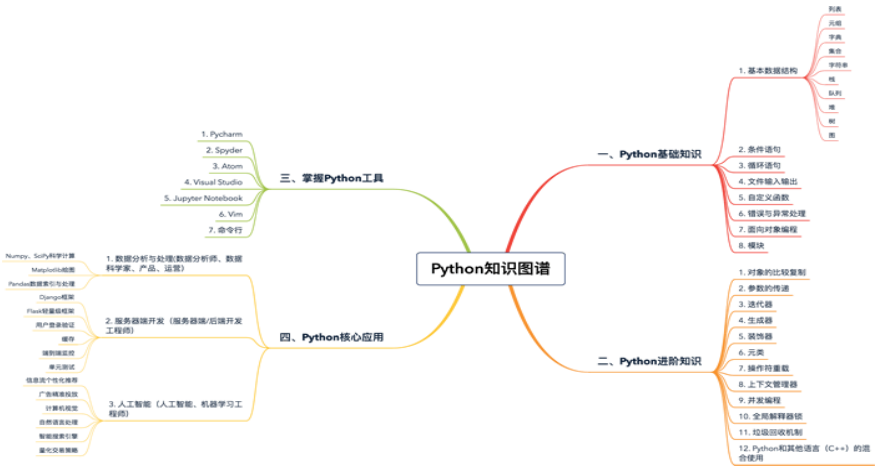
要想成为一名优秀的程序员，从起步阶段就要刻意地去培养自己的编程规范。例如，在刚开始编写代码时，不要将原本多行的代码全部写到一行，也不要随便用 a、b、c 等字母作为变量的名称。

3) 开发经验是必不可少的

要想真正熟练地掌握 Python（或者其它编程语言），拥有大中型产品的开发经验是必不可少的，它会让你站得更高，望得更远。

例如，我们几乎每天都会用搜索引擎，但你了解一个搜索引擎的服务器端实现吗？这是一个典型的面向对象设计，需要定义一系列相关的类和函数，还要从产品需求、代码复杂度、效率以及可读性等多个方面综合考量，同时在上线后还要进行各种优化等。

本教程中无法让你完成一个上亿用户级产品的实践，但设置有一些小项目，可以帮你掌握必要的开发知识。最后，这里为你准备了一章 Python 学习的知识图谱，涵盖了 Python 的核心知识，其中的大部分内容本教程都会做详细讲解。



Python 知识体系框架

1.8 Python 版本区别，Python 3 和 Python 2 区别详解

和 Python 2.x 版本相比，Python 3.x 版本在语句输出、编码、运算和异常等方面做出了一些调整，本节就对这些调整一一做一下简单介绍。

本节适合有 Python 基础的学员阅读，初学者可先跳过本节，整体学完 Python 之后，再回过头来阅读。

Python 3.x print 函数代替了 print 语句

在 Python2.x 中，输出数据使用的是 Print 语句，例如：

```
>>> print "3,4"
3,4
或者
>>> print(3,4)
(3,4)
```

但是在 Python 3.x 中，print 语句没有了，取而代之的是 print 函数，例如：

```
>>> print(3,4)
3 4
```

如果还像 Python 2.x 中那样使用 print 语句，Python 编译器就会报错，例如：

```
>>> print "3,4"
  File "<stdin>", line 1
    print "3,4"
            ^
SyntaxError: Missing parentheses in call to 'print'
```

Python 3.x 默认使用 UTF-8 编码

相比 Python 2.x 默认采用的 ASCII 编码，Python 3.x 默认使用 UTF-8 编码，可以很好地支持中文或其它非英文字符。

例如，输出一句中文，使用 Python 2.x 和 Python 3.x 的区别如下：

```
#Python 2.x
>>>str ="C 语言中文网"
```

```
>>>str
'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x91'

#Python 3.x
>>>str ="C 语言中文网"
>>>str
'C 语言中文网'
```

不仅如此，在 Python 3.x 中，下面的代码也是合法的：

```
>>>中国="China"
>>>print(中国)
China
```

Python 3.x 除法运算

和其他语言相比，Python 的除法运算要高端很多，它的除法运算包含 2 个运算符，分别是 / 和 //，这 2 个运算符在 Python 2.x 和 Python 3.x 的使用方法如下：

/ 运算符

在 Python 2.x 中，使用运算符 / 进行除法运算的方式和 Java、C 语言类似，整数相除的结果仍是一个整数，浮点数除法会保留小数点部分，例如：

```
>>>1/2
0
>>>1.0/2
0.5
```

但是在 Python 3.x 中使用 / 运算符，整数之间做除法运算，结果也会是浮点数。例如：

```
>>>1/2
0.5
```

运算符 //

使用运算符 // 进行的除法运算叫做 floor 除法，也就是输出不大于结果值的一个最大的整数（向下取整）。此运算符的用法在 Python 2.x 和 Python 3.x 中是一样的，举个例子：

```
#Python 2.x
>>> -1//2
-1

#Python 3.x
```

```
>>> -1//2
-1
```

Python 3.x 异常

在 Python 3.x 版本中，异常处理改变的地方主要在以下几个方面：

1. 在 Python 2.x 版本中，所有类型的对象都是直接被抛出的，但是在 Python 3.x 版本中，只有继承 BaseException 的对象才可以被抛出。
2. 在 Python 2.x 版本中，捕获异常的语法是“except Exception，var:”；但在 Python 3.x 版本中，引入了 as 关键字，捕获异常的语法变更为 “except Exception as var:”。
3. 在 Python 3.x 版本中，处理异常用 “raise Exception(args)”代替了“raise Exception，args”。
4. Python 3.x 版本中，取消了异常类的序列行为和 .message 属性。

有关 Python 2.x 版本和 Python 3.x 版本处理异常的示例代码如下所示：

```
#Python 2.x
>>> try:
...  raise TypeError,"类型错误"
... except TypeError,err:
...  print err.message
...
类型错误

#Python 3.x
>>> try:
...     raise TypeError("类型错误")
... except TypeError as err:
...     print(err)
...
类型错误
```

Python 3.x 八进制字面量表示

在 Python 3.x 中，表示八进制字面量的方式只有一种，并且必须写成“0o1000”这样的方式，原来“01000”的方式不能使用了。举个例子：

```
#Python 2.x
>>> 0o1000
512
>>> 01000
512
```

```
#Python 3.x
>>> 01000
  File "<stdin>", line 1
    01000
        ^
SyntaxError: invalid token
>>> 0o1000
512
```

Python 3.x 不等于运算符

Python 2.x 中的不等于运算符有 2 种写法，分别为 != 和 <>，但在 Python 3.x 中去掉了 <>，只有 ！= 这一种写法，例如：

```
#Python 2.x
>>> 1!=2
True
>>> 1<>2
True

#Python 3.x
>>> 1!=2
True
>>> 1<>2
  File "<stdin>", line 1
    1<>2
      ^
SyntaxError: invalid syntax
```

Python 3.x 数据类型

Python 3.x 中对数据类型也做了改动，比如说：

- Python 3.x 去除了 long 类型，现在只有一种整形 int，但它的行为就像是 Python 2.x 版本中的 long。
- Python 3.x 新增了 bytes 类型，对应 Python 2.x 版本的八位串，定义 bytes 字面量的方法如下所示：

```
>>>b=b'China'
>>>type(b)
<type 'bytes'>
```

字符串对象和 bytes 对象可以使用 .encode() 或者 .decode()方法相互转化，例如：

```
>>>s=b.decode()
>>>s
'China'
>>>b1=s.encode()
>>>b1
b'China'
```

- Python 3.x 中，字典的 keys()、items() 和 values() 方法用返回迭代器，且之前的 iterkeys() 等函数都被废弃。同时去掉的还有 dict.has_key()，改为用 in 替代。

> 本节所介绍的只是 Python 3.x 的一部分改动，由于篇幅有限，这里不再具体指出，教程中涉及到时再给大家详细介绍。

1.9 Python 2.x 和 Python 3.x，初学者应如何选择？

Python 自发布以来，主要有 3 个版本，分别是：

1. 1994 年发布的 Python 1.0 版本（已过时）；
2. 2000 年发布的 Python 2.0 版本，截止到 2019 年 3 月份，已经更新到 2.7.16；
3. 2008 年发布的 Python 3.0 版本，截止到 2019 年 3 月份，已经更新到 3.7.3；

3 个版本中，Python 3.0 是一次重大的升级，为了避免引入历史包袱，Python 3.0 没有考虑与 Python 2.x 的兼容，这也就导致很长时间以来，Python 2.x 的用户不愿意升级到 Python 3.0。

除此之外，造成目前这种状况的另一个原因是，将现有应用从 Python 2.x 迁移到 Python 3.x 是一项不小的挑战。虽然有 2to3（后续会介绍）之类的工具可以进行代码的自动转换，但无法保证转换后的代码 100% 正确。而且，如果不做人工修改的话，转换后的代码性能可能还不如转换前。因此，将现有的复杂代码库迁移到 Python 3.x 上可能需要付出巨大的精力和成本，某些公司无法负担这些成本。

目前，根据统计显示，使用 Python 2.x 的开发者仍占 63.7%，而 Python 3.x 的用户占 36.3%，由此可见，使用 Python 2.x 的用户还是占多数。在 2014 年，Python 创始人宣布，将 Python 2.7 支持时间延长到 2020 。

那么，初学者应该选择什么版本呢？本教程建议大家选择 Python 3.x 版本，理由有以下几点：

- 使用 Python 3.x 已经是大势所趋
  目前，虽然使用 Python 2.x 的开发者居多，但使用 Python 3.x 的开发者正在迅速增加，如图 1 所示：

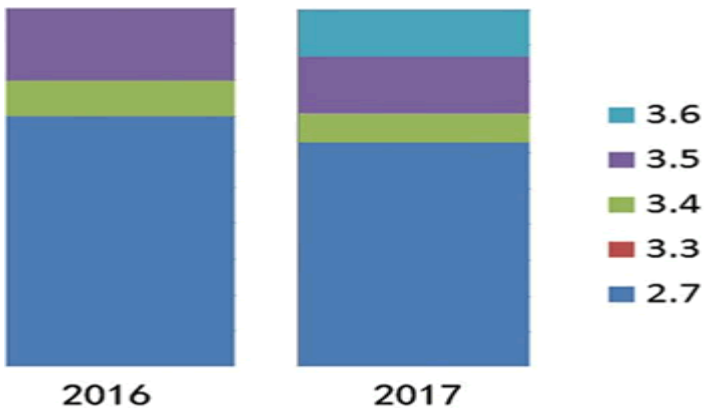


图 1 Python 3.x 是大势所趋

另外，根据 PEP-404 这份文档，Python 2.x 版本将不会再发布 2.8 版本，且未来非常重要的项目（如 Django、Flask 和 Numpy）可能都将放弃对 2.x 的支持，仅支持 Python 3.x。因此，从长远来看，学习 Python 3.x 只有好处。

本教程也是以 Python 3.x 来介绍 Python 编程，因此，为了同步，强烈建议初学者选择 Python 3.x。

- Python 3.x 在 Python 2.x 的基础上做了功能升级
  Python 3.x 对 Python 2.x 的标准库进行了一定程序的重新拆分和整合，比 Python 2.x 更容易理解，特别是在字符编码方面。Python 2.x 中对于中文字符串的支持性能不够好，需要编写单独的代码对中文进行处理，否则不能正确显示中文，而 Python 3.x 已经将该问题成功解决了。
- Python 3.x 和 Python 2.x 思想基本是共通的
  Python 3.x 和 Python 2.x 思想基本上是共通的，只有少量的语法差别，学会的 Python 3.x，只要稍微花点时间学习 Python 2.x 的语法，即可灵活运用这两个不同版本了。

注意，选择 Python 3.x 也不是没有弊端，很多扩展库的发行总是会滞后于 Python 的发行版本，甚至目前还有很多库不支持 Python 3.x。

因此，在选择 Python 时，一定要先考虑清楚自己的学习目的，比如说，打算做哪方面的开发，此方向需要用法哪些扩展库，以及这些扩展库支持的最高 Python 版本等，明确这些问题后，再选择适合自己的版本。

> 关于 Python 3.x 和 Python 2.x 具体的区别，可阅读《Python 3 和 Python 2 区别》一节。

1.10 Python 2to3：自动将 Python 2.x 代码转换成 Python3.x 代码

由于 Python 2.x 和 Python 3.x 的差别较大，因此 Python 2.x 的多数代码无法直接在 Python 3.x 环境中运行。而由于兼容性的原因，我们在网络上查找的资源多数是 Python 2.x 的代码，如果想要在 Python 3.x 环境下运行，就需要修改源代码。

针对这一问题，Python 官方提供了一个将 Python 2.x 代码自动转换为 Python 3.x 代码的小工具，它就是 2to3.py，通过该工具可以将大部分 Python 2.x 代码转换为 Python 3.x 代码。

接下来，就给大家详细介绍一下，如何使用 2to3.py 将 Python 2.x 代码转换成 Python 3.x 代码。

假设我们现在有 Python 2.x 的代码，要将其转换成 Python 3.x 代码，需要按照一下几个步骤进行操作：

1. 找到 2to3.py 文件，该文件保存在 Python 安装路径下的 “Tools\scripts” 目录中。比如说，我们将 Python 安装在了“G:\Python\Python36”目录中，那么 2to3.py 文件则保存在“G:\Python\Python36\Tools\scripts ”目录中，如图 1 所示：

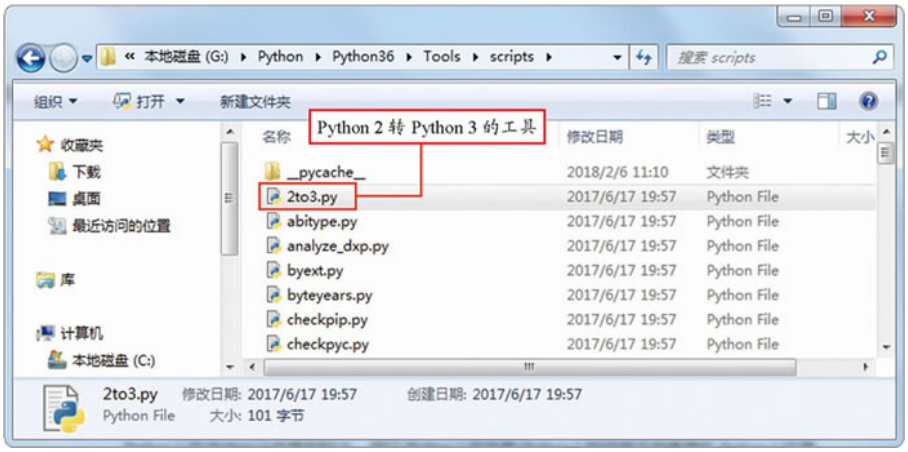


图 1 Python 2.x 转 Python 3.x 的工具

2. 将 2to3.py 文件复制到要转换代码所在的目录中。
3. 打开命令行窗口，并进入要转换代码文件所在的目录。例如，该文件保存在“E:\change”目录下，则在命令行窗口中可输入如下命令：

```
C:\users\Demo>E:
E:\>cd change
E:\change>
```

4. 调用 2to3.py 工具转化代码。例如，要转化的文件名称为 demo.py 文件，可以使用下面的代码：

```
Python 2to3.py -w demo.py
```

执行此行代码之后，将会在 “E:\change” 目录下创建一个 demo.py 文件的备份文件，名称为 demo.py.bak，同时，原 demo.py 文件的内容被转换为 Python 3.x 对应的代码。

注意，在使用 2to3.py 转换 python 2.x 代码前，尽量不要把要转换的代码保存在 C 盘中，因此如果保存在 C 盘，可能会因权限问题导致转换不能正常完成。

1.11 Python PEP 文档：及时追踪 Python 最新变化

通过前面章节对 Python 语言的介绍，到目前为止，Python 已经历经了 3 个版本的迭代，读者不禁要问，Python 为什么要不断的升级呢？作为程序猿的我们，怎样才能及时了解 Python 的最新变化呢？

首先解决第一个问题，Python 不断升级的原因很简单，因为用户有了更高的需求。人们之所以设计新的编程语言，是因为他们发现现有的语言已经无法以最佳方式来解决问题。此外，Python 的使用范围越来越广，人们发现它有许多可以改进的地方，应该做出这样的改进。

Python 的很多改进都是有特定应用领域的需求驱动的，其中最重要的领域是 Web 开发，这一领域需要 Python 改进对并发的处理。
还有一些是由于 Python 的历史原因导致的，随着使用 Python 的不断深入，发现了 Python 的一些不合理之处。比如，有些是标准库模块结构混乱或冗余，有些是程序设计缺陷。

那么，怎样才能及时追踪 Python 的更新动态呢？这就需要借助 Python PEP 文档。

Python PEP 文档

要知道，虽然各种各样的 Python 语句修改方案会以邮件列表的形式进行讨论，但 Python 社区有一种应对变化的固定方法，即只有发布了名为 PEP 的新文档，新的变化才会生效。

PEP（Python Enhancement Proposal），全称是 Python 改进方案。它是提交 Python 变化的书面文档，也是社区对这一变化进行讨论的出发点。值得一提得是，PEP 文档的整个目的，格式以及工作流程的标准格式，都包含 PEP 1 文档中。

PEP 文档对 Python 十分重要，它主要有以下 3 个用途：

1. 通知：汇总 Python 核心开发者重要的信息，并通过 Python 发布日程；
2. 标准化：提供代码风格、文档或者其他指导意见；
3. 设计：对提交的功能进行说明。

所有提交过的 PEP 都被汇总在 PEP 0 文档中。

需要注意的是，如果读者对 Python 语言的未来发展方向感兴趣，但苦于没有时间追踪 Python 邮件列表中的讨论，那么 PEP 0 是信息来源的不错选择，它会告诉你哪些文档已被接受但尚未实施，哪些文档仍在审议中。

不仅如此，PEP 还有其他的用途，比如说，人们常常会问以下类似的问题：

- A 功能为什么要以这样的方式运行？
- Python 为什么没有 B 功能？

多数情况下，关于该功能的某个 PEP 文档已经给出了上述问题的详细回答。
另外，还有关于 Python 语言功能的 PEP 文档并没有通过，这些文档可作为历史资料来参考。

1.12 Python 底层是用什么语言实现的？

确切地说，本教程介绍的是用 C 语言编写实现的 Python，又称为 CPython。平时我们所讨论的 Python，指的其实就是 CPython。

随着编程语言的不断发展，Python 的实现方式也发生了变化，除了用 C 语言实现外，Python 还有其他的实现方式。例如，用 Java 语言实现的 Python 称为 JPython，用 .net 实现的 Python 称为 IronPython 等等。

Python 的这些实现方式虽然诞生比 CPython 晚，但一直在努力地跟上主流，并在不同的生产环境中不断地使用并推广 Python。
Python 的实现方式有很多种，Python 官网上介绍了 20 多种语言变体、方言或 C 语言之外的 Python 解释器实现。其中一些只是实现了语言核心语法、功能和内置扩展的一个子集，但至少有几个与 CPython 几乎完全兼容。更重要的是，在这些不同的实现方式中，虽然有些只是玩具项目或实验，但大部分都是为了解决某些实际问题而创建的，这些问题要么使用 CPython 无法解决，要么需要开发人员花费巨大的精力，这里举几个例子：

- 在嵌入式系统中运行 Python 代码。
- 与运行框架（如 Java 或 .NET）或其他语言做代码集成。
- 在 Web 浏览器中运行 Python 代码。

由于受到篇幅的限制，本节仅给大家介绍几种 Python 开发人员最常用的几种 Python 实现方式。

Stackless Python

Stackless Python 自称 Python 增强版。之所以名为 Stackless（无栈），是因为它没有依赖 C 语言的调用栈，实际上就是对 CPython 做了一些修改，添加了一些新的功能。

在新添加的功能中，最重要就是由解释器管理的微线程，用来替代依赖系统内核上下文切换和任务调度的普通线程，既轻量化又节约资源。

Stackless Python 最新可用的版本是 2.7.9 和 3.3.5，分别实现的是 Python 2.7 和 3.3。在 Stackless Python 中，所有的额外功能都是内置 stackless 模块内的框架。

Stackless Python 并不是最有名的 Python 实现，但很值得一提，因为它引入的思想对编程语言社区有很大的影响。例如，将 Stackless Python 中的内核切换功能提取出来并作为一个独立包发布，名为 greenlet，是许多有用的库和框架的基础。

此外，Stackless Python 的大部分功能都在 PyPy 中重新实现，PyPy 是另一个 Python 实现，我们将稍后介绍。

JPython

Jython 是 Python 语言的 Java 实现。它将代码编译为 Java 字节代码，开发人员在 Python 模块中可以无缝使用 Java 类。

Jython 允许人们在复杂应用系统（例如 J2EE）中使用 Python 作为顶层脚本语言，它还将 Java 应用引入到 Python 中，一个很好的例子就是，在 Python 程序中可以使用 Apache Jackrabbit（这是一个基于 JCR 的文档仓库 API）。

Jython 最新可用的版本是 Jython 2.7，对应的是 Python 2.7 版。它宣称几乎实现了 Python 所有的核心标准库，并使用相同的回归测试套件。Jython 3.x 版正在开发中。

Jython 与 CPython 实现的主要区别如下所示：

- 真正的 Java 垃圾回收，而不是引用计数。
- 没有全局解释器锁（GlobakInterpreter Lock，GIL），在多线程应用中可以充分利用多个内核。

这一语言实现的主要缺点是缺少对 C/Python 扩展 API 的支持，因此用 C 语言编写的 Python 扩展在 Jython 中无法运行。这种情况未来可能会发生改变，因为 Jython 3.x 计划支持 C/Python 扩展 API。

某些 Python Web 框架（例如 Pylons）被认为是促进 Jython 的开发，使其可用于 Java 世界。

IronPython

IronPython 将 Python 引入 .NET 框架中，这个项目受到微软的支持，因为 IronPython 的主要开发人员都在微软工作。可以这么说，IronPython 是推广语言的一种重要实现。

除了 Java，.NET 社区是最大的开发者社区之一。
值得一提的是，微软提供了一套免费开发工具，名为 PTVS（Python Tools for Visual Studio，用于 Visual Studio 的 Python 工具），可以将 Visual Studio 转换为成熟的 Python IDE。这是作为 Visual Studio 的插件发布的，在 GitHub 可以找到其开源代码。

IronPython 最新的稳定版本是 2.7.5，与 Python 2.7 兼容。与 Jython 类似，Python 3.x 的实现也在开发中，但还没有可用的稳定版本。

虽然 .NET 主要在微软 Windows 系统上运行，但是 IronPython 也可以在 Mac OS X 和 Linux 系统上运行，这一点要感谢 Mono，一个跨平台的开源 .NET 实现。

与 CPython 相比，IronPython 的主要区别或优点如下：

- 与 Jython 类似，没有全局解释器锁（Global Interpreter Lock，GIL），在多线程应用中可以充分利用多个内核。

- 用 C# 和其他 .NET 语言编写的代码可以轻松集成到 IronPython 中，反之亦然。
- 通过 Silverlight，在所有主流 Web 浏览器中都可以运行。

说到弱点，IronPython 也与 Jython 非常类似，因为它也不支持 C/Python 扩展 API。对于想要使用主要基于 C 扩展的 Python 包（例如 NumPy）的开发人员来说，这一点很重要。

有一个叫作 ironclad 的项目，其目的是在 IronPython 中无缝使用这些扩展，其最新支持的版本是 2.6，开发已经停止。

PyPy

PyPy 可能是最令人兴奋的 Python 实现，因为其目标就是将 Python 重写为 Python。在 PyPy 中，Python 解释器本身是用 Python 编写的。

在 Python 的 CPython 实现中，有一个 C 代码层来实现具体细节。但在 PyPy 实现中，这个 C 代码层用 Python 完全重写。这样，你可以在代码运行期间改变解释器的行为，并实现 CPython 难以实现的代码模式。

目前 PyPy 的目的是与 Python 2.7 完全兼容，而 PyPy3 则与 Python 3.2.5 版兼容。

以前对 PyPy 感兴趣主要是理论上的原因，只有喜欢深入钻研语言细节的人才会对它感兴趣。PyPy 通常不用于生产环境，但这些年来这种状况已经发生改变，PyPy 通常比 CPython 实现要快得多。基于这一特性，使得越来越多的开发人员决定在生产环境中切换到 PyPy。

PyPy 与 CPython 实现的主要区别在于以下几个方面：

- 使用垃圾回收，而不是引用计数。
- 集成跟踪 JIT 编译器，可以显著提高性能。
- 借鉴 Stackless Python 在应用层的无栈特性。

与几乎所有其他的 Python 实现类似，PyPy 也缺乏对 C/Python 扩展 API 的完全官方支持。但它至少通过 CPyExt 子系统为 C 扩展提供了某种程度的支持，虽然文档不完整，功能也尚未完善。此外，社区正在努力将 NumPy 迁移到 PyPy 中，因为这是最需要的功能。

1.13 了解 Jupyter Notebook，你已然超越了 90% 的 Python 程序员

在 2019 年 8 月份 TOIBE 编程语言社区公布的编程语言排行榜中，Python 已经超过了 C++，稳居排行榜第 3 名（如图 1 所示）。

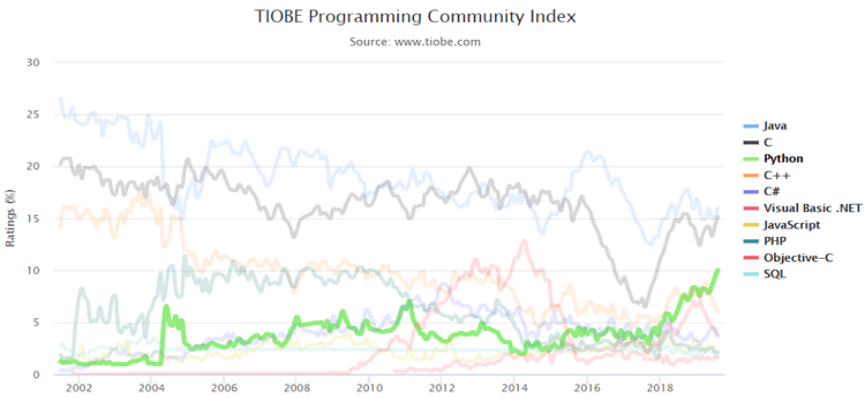


图 1 TOIBE 编程社区排行榜

我们应该知道，Python 之所有在 14 年后“崛起”，得益于机器学习和数学统计应用的兴起。至于 Python 适合数学统计和机器学习的原因，很多读者可能并不会想到，Jupyter Notebook 功不可没。可以好不夸张地说，作为一名 Python 工程师，如果不会使用 Jupyter Notebook，可能会真的太落伍了。

本节，就带领大家学习一下 Jupyter Notebook。

什么是 Jupyter Notebook

说了这么多，到底什么是 Jupyter Notebook 呢？按照 Jupyter 创始人的说法，起初他是想做一个综合 Ju（Julia）、Py（Python）和 R 三种科学运行语言的计算工具平台，所以将其命名为 Jupyter。

Jupyter 发展到现在，已经成为了一个几乎支持所有语言，能够把软件代码、计算输出、解释文档、多媒体资源整合在一起的多功能科学运行平台。Jupyter Notebook 的工作界面如图 2 所示。

图 2 Jupyter Notebook 界面（点此查看高清大图）

如图 2 所示，只要直接输入代码并运行，它就会直接在代码下面显示输出结果。那么，Jupyter Notebook 对 Python 的影响真的有那么大吗？

Jupyter Notebook 的影响力

衡量一个技术的影响力，或者说想利用自己的技术影响世界时，必定绕不开技术对教育界的影响力。

以微软的 Word 文本处理系统为例，从纯技术角度来讲，它的单机设计理念早已过时，但以 Google Doc 为代表的在线文档系统，并没有想象中实现对 Word 的降维打击。最直接的原因归咎于用户习惯，多数用户已经习惯使用 Word 来编辑、修改文档啦，之所以会这样，是因为我们从小学、中学，一直到大学期间，都在学习使用 Word。而且到了工作中，老员工还会带着新员工继续使用 Word。

从 2017 年开始，已经有大量的北美顶尖计算机课程开始使用 Jupyter Notebook 作为教学工具，而在这之前，一直使用的都是 Python 命令行的形式。不仅如此，Jupyter Notebook 在工业界的影响力更大。

例如，在 Facebook 中，几乎所有的中小型程序（比如内部的线下分析软件，机器学习模块的训练等）都是借助与 Jupyter Notebook 完成。同时，在硅谷其他的一线大厂中，也全部使用 Jupyter Notebook（他们用的是改进定制型的 Jupyter Notebook，名称 Google Colab）。

了解了以上这些，相信你已经认可了 Jupyter Notebook 的江湖地位。

但需要注意的是，不是说那个技术流行，就盲目跟从。对于技术的选择，要学习独立的思考，切勿人云亦云。以 Facebook 为例，之所以它选择 Jupyter Notebook 等技术，很大程序上因为它有几百个产品线，几万个工程师，而如果是几个人的团队，使用同样的技术反倒成了拖累。

Jupyter Notebook 的优势

这里给大家总结了几点 Jupyter Notebook 的优势：

1) 整合了所有资源

在软件开发过程中，频繁地进行上下文切换，会影响生产效率。举个例子，假设你需要切换窗口去看一些文档，再切换窗口去用另一个工具画图，不断地切换窗口就会成为影响效率的因素。

而 Jupyter Notebook 则不同，它会将所有和软件编写的资源全部放在一个地方，无需切换窗口就可以轻松找到。

2) 交互性编程体验

在机器学习和数据统计领域，Python 编程的实验性特别强，比如为了测试 100 种不同的方法，有时就需要将一小块代码重写 100 遍，在这种情况下，如果使用传统的 Python 开发流程，每一次测试都要将所有代码重新

跑一遍，会花费开发者很多时间。

Jupyter Notebook 引进了 Cell 的概念。每次测试可以只跑一小块的代码，并且在代码下方立刻就能看到运行结果。

如此强的交互性，满足了 Python 程序员可以专注于问题本身，不会被频繁的工具链拖累，也不用在命令行之间来回切换，所有工作都能在 Jupyter Notebook 上完成。

3) 轻松运行他人编写的代码

同样是在机器学习和数学统计领域，我们可能会借鉴他人分享的代码，但当我们拷贝过来想要运行时，却需要使用 pip 安装一大堆依赖的库，足以让人抓狂。而 Jupyter Notebook 就可以解决这个问题。

例如，Jupyter 官方的 Binder 平台以及 Google 提供的 Google Colab 环境，它们可以让 Jupyter Notebook 变得和 Google Doc 在线文档一样。比如用 Binder 打开一份 GitHub 上的 Jupyter Notebook 时，就不需要安装任何 Python 库，直接在打开代码就能运行。

本节，仅是为了让初学者对 Jupyter Notebook 有一个初步的了解，具体 Jupyter Notebook 的安装和使用，网络上有很多详细的教程，这里不再做过多描述。

注意，本教程仍旧使用 Python 内置的 IDLE 作为教学工具。

第 2 章　Python 编程环境搭建

2.1Windows 安装 Python（图解）

在 Windows 上安装 Python 和安装普通软件一样简单，下载安装包以后猛击“下一步”即可。
Python 安装包下载地址：https://www.python.org/downloads/
打开该链接，可以看到有两个版本的 Python，分别是 Python 3.x 和 Python 2.x，如下图所示：

Release version	Release date		Click for more
Python 3.8.1	Dec. 18, 2019	Download	Release Notes
Python 2.7.17	Oct. 19, 2019	Download	Release Notes
Python 3.7.5	Oct. 15, 2019	Download	Release Notes
Python 3.8.0	Oct. 14, 2019	Download	Release Notes
Python 3.7.0	June 27, 2018	Download	Release Notes
Python 3.6.5	March 28, 2018	Download	Release Notes
Python 3.5.5	Feb. 5, 2018	Download	Release Notes
Python 2.7.14	Sept. 16, 2017	Download	Release Notes
Python 3.4.7	Aug. 9, 2017	Download	Release Notes

图 1 Python 下载页面截图（包含 Python 2.x 和 Python 3.x 两个版本）

在《Python 2.x 和 Python 3.x，初学者应如何选择？》一文中提到，Python 3.x 是一次重大升级，为了避免引入历史包袱，Python 3.x 没有考虑与 Python 2.x 的兼容性，这导致很多已有的项目无法顺利升级 Python 3.x，只能继续使用 Python 2.x，而大部分刚刚起步的新项目又使用了 Python 3.x，所以目前官方还需要维护这两个版本的 Python。

我建议初学者直接使用 Python 3.x。截止到目前（2020-01-02），Python 的最新版本是 3.8.x，我们就以该版本为例演示 Windows 下的 Python 安装过程。

点击上图中的版本号或者“Download”按钮进入对应版本的下载页面，滚动到最后即可看到各个平台的 Python 安装包。

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		f215fa2f55a78de739c1787ec56b2bcd	23978360	SIG
XZ compressed source tarball	Source release		b3fb85fd479c0bf950c626ef80cacb57	17828408	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	d1b09665312b6b1f4e11b03b6a4510a3	29051411	SIG
Windows help file	Windows		f6bbf64cc36f1de38fbf61f625ea6cf2	8480993	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	4d091857a2153d9406bb5c522b211061	8013540	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	3e4c42f5ff8fcdbe6a828c912b7afdb1	27543360	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	662961733cc947839a73302789df6145	1363800	SIG
Windows x86 embeddable zip file	Windows		980d5745a7e525be5abf4b443a00f734	7143308	SIG
Windows x86 executable installer	Windows		2d4c7de97d6fcd8231fc3decbf8abf79	26446128	SIG
Windows x86 web-based installer	Windows		d21706bdac544e7a968e32bbb0520f51	1325432	SIG

图 2 各个平台的 Python 安装包

对前缀的说明：

- 以 `Windows x86-64` 开头的是 64 位的 Python 安装程序；
- 以 `Windows x86` 开头的是 32 位的 Python 安装程序。

对后缀的说明：

- `embeddable zip file` 表示 `.zip` 格式的绿色免安装版本，可以直接嵌入（集成）到其它的应用程序中；
- `executable installer` 表示 `.exe` 格式的可执行程序，这是完整的离线安装包，一般选择这个即可；
- `web-based installer` 表示通过网络安装的，也就是说下载到的是一个空壳，安装过程中还需要联网下载真正的 Python 安装包。

这里我选择的是“Windows x86-64 executable installer”，也即 64 位的完整的离线安装包。

双击下载得到的 python-3.8.1-amd64.exe，就可以正式开始安装 Python 了，如图 3 所示。

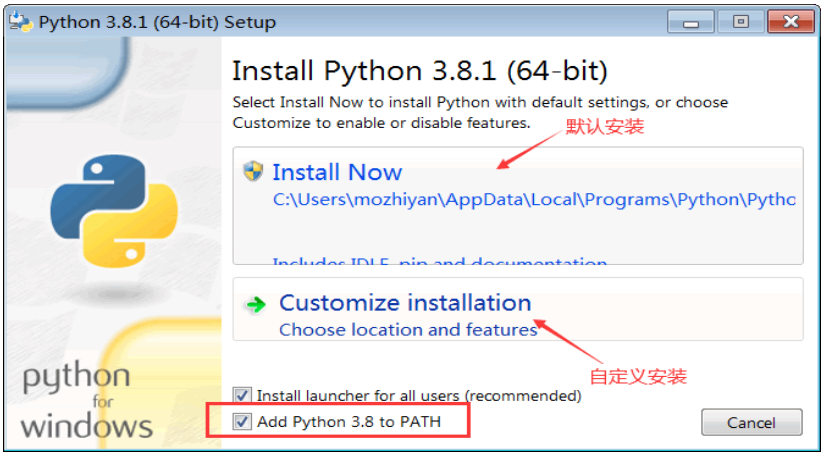


图 3 Python 安装向导

请尽量勾选 `Add Python 3.8 to PATH`，这样可以将 Python 命令工具所在目录添加到系统 Path 环境变量中，以后开发程序或者运行 Python 命令会非常方便。

Python 支持两种安装方式，默认安装和自定义安装：

- 默认安装会勾选所有组件，并安装在 C 盘；
- 自定义安装可以手动选择要安装的组件，并安装到其它盘符。

这里我们选择自定义安装，将 Python 安装到常用的目录，避免 C 盘文件过多。点击“Customize installation”进行入下一步，选择要安装的 Python 组件。

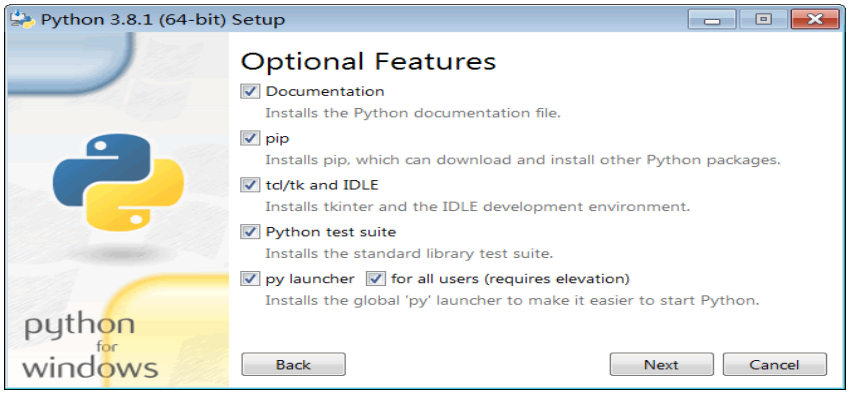


图 4 选择要安装的 Python 组件

没有特殊要求的话，保持默认即可，也就是全部勾选。

点击“Next”继续，选择安装目录。

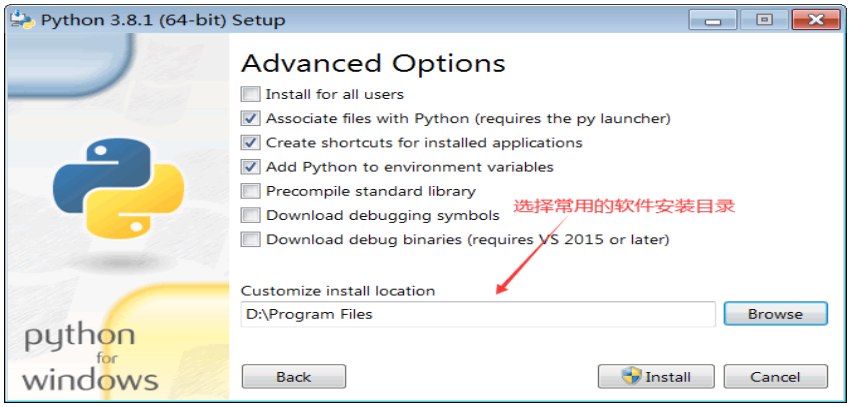


图 5 选择安装目录

选择好你常用的安装目录，点击“Install”，等待几分钟就可以完成安装。

安装完成以后，打开 Windows 的命令行程序（命令提示符），在窗口中输入 `python` 命令（注意字母 `p` 是小写的），如果出现 Python 的版本信息，并看到命令提示符 `>>>`，就说明安装成功了，如下图所示。

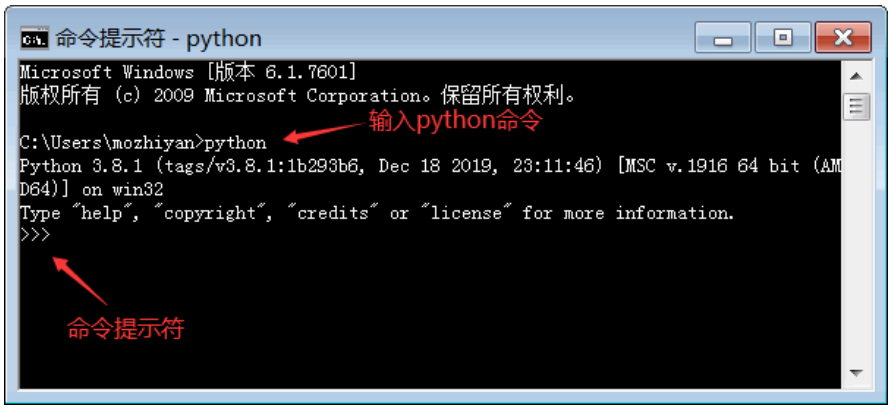


图 6 运行 python 命令

运行 python 命令启动的是 python 交互式编程环境，我们可以在`>>>`后面输入代码，并立即看到执行结果，请看下面的例子。

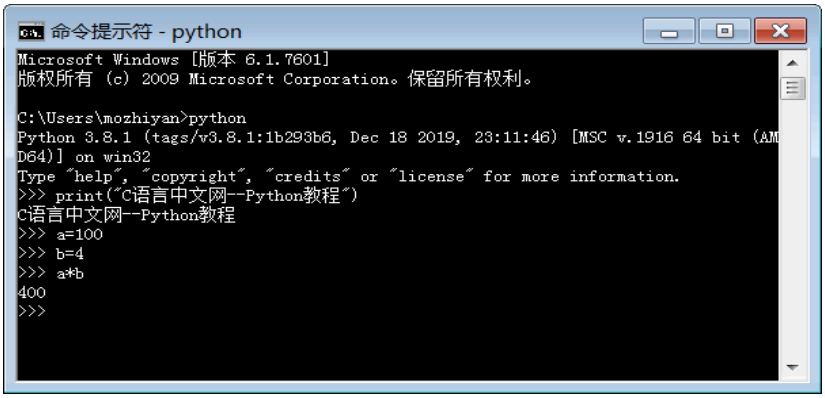


图 7 在 Python 交互式环境中编写代码

按下`Ctrl+Z`快捷键，或者输入 exit() 命令即可退出交互式编程环境，回到 Windows 命令行程序。

关于 IDLE

IDLE 是 Python 自带的简易开发环境，安装完成以后，在 Windows 开始菜单中找到`Python 3.8`文件夹，在这里可以看到 IDLE 工具，我们将在《Python IDLE 使用方法详解》一节中详细介绍。

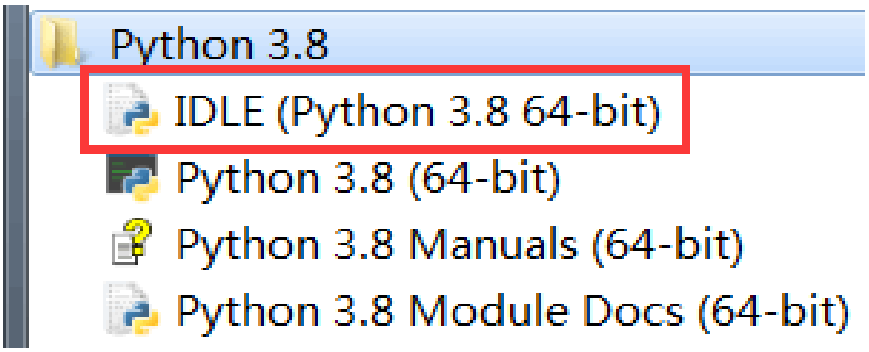


图 8 Python IDLE 简易开发环境

2.2 Linux（Ubuntu）系统安装 Python

Linux 系统是为编程而生的，因此绝大多数的 Linux 发行版（Ubuntu、CentOS 等）都默认自带了 Python。有的 Linux 发行版甚至还会自带两个版本的 Python，例如最新版的 Ubuntu 会自带 Python 2.x 和 Python 3.x。

打开 Linux 发行版内置的终端（Terminal），输入 `python` 命令就可以检测是否安装了 Python，以及安装了哪个版本，如下所示：

```
[c.biancheng.net@localhost ~]$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

可以看到，`python` 命令能够正常运行，并输出了 Python 的版本信息，这表明当前的 Linux 发行版已经自带了 Python 2.7.5。

另外，执行结果最后出现了 Python 命令提示符`>>>`，这意味着我们进入了 Python 交互式编程环境，可以在这里直接输入代码并查看运行结果，如下所示：

```
[c.biancheng.net@localhost ~]$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("C 语言中文网的网址是：http://c.biancheng.net")
C 语言中文网的网址是：http://c.biancheng.net
>>> a=100
>>> b=4
>>> a*b
400
>>> exit()
[c.biancheng.net@localhost ~]$
```

exit() 用来退出 Python 编程环境，回到 Linux 命令行。

大部分的 Linux 发行版会自带 Python 2.x，但是不一定自带 Python 3.x，要想检测当前 Linux 发行版是否安装

了 Python 3.x，可以在终端（Terminal）输入 python3 命令，如下所示：

```
[c.biancheng.net@localhost ~]$ Python3
Python 3.6.4 (default , Nov 18 2018 , 13:02:36)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help","copyright","credits" or "license" for more information.
>>>
```

如果 python3 命令运行成功，并出现 Python 提示符>>>，则表明当前 Linux 发行版已经安装了 Python 3 开发环境，只需执行 python3 命令就可以启动 Python 3 开发环境。

如果当前 Linux 发行版没有安装 Python 3，或者你觉得现有的 Python 3 版本不够新，那么就需要更新 Python 版本。本节我们以 Ubuntu 为例来进行演示。

更新 Python 版本

在 Ubuntu 终端执行以下两条命令即可更新 Python 版本：

```
$sudo apt-get update
$sudo apt-get install python3.8
```

对命令的说明：

- 第一条命令用来指定更新 /etc/apt/sources.list 和 /etc/apt/sources.list.d 所列出的源地址，这样能够保证获得最新的安装包。
- 第二条命令用来指定安装 Python 3.8，这是目前最新的 Python 版本。

等待以上两条命令执行完成，再次在终端输入 python3 命令，就可以看到 Python 交互式编程环境已经更新到 Python 3.8。

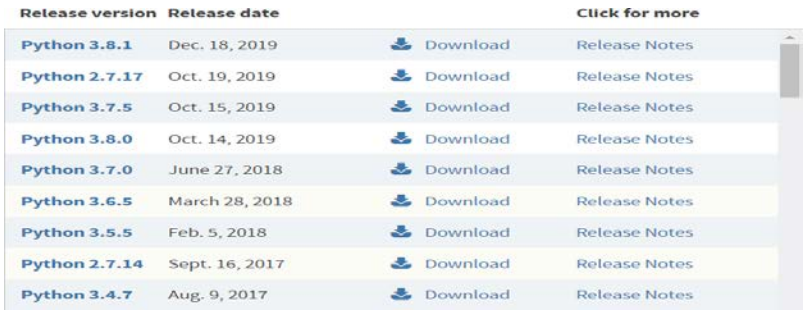
重新安装 Python

以上更新方法仅在 Ubuntu 已经安装 Python 的情况下才有效，如果你的 Ubuntu 中没有 Python 环境，或者你想重新安装，那么就得到官网下载源代码，然后自己编译。

1) 下载源代码

Python 官方下载地址：https://www.python.org/downloads/

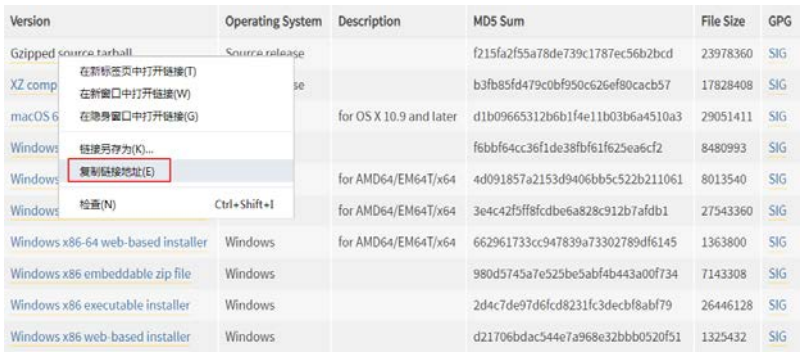
打开链接，可以看到各个版本的 Python：



Release version	Release date		Click for more
Python 3.8.1	Dec. 18, 2019	Download	Release Notes
Python 2.7.17	Oct. 19, 2019	Download	Release Notes
Python 3.7.5	Oct. 15, 2019	Download	Release Notes
Python 3.8.0	Oct. 14, 2019	Download	Release Notes
Python 3.7.0	June 27, 2018	Download	Release Notes
Python 3.6.5	March 28, 2018	Download	Release Notes
Python 3.5.5	Feb. 5, 2018	Download	Release Notes
Python 2.7.14	Sept. 16, 2017	Download	Release Notes
Python 3.4.7	Aug. 9, 2017	Download	Release Notes

图 1 Python 下载页面截图

点击上图中的版本号或者“Download”按钮进入对应版本的下载页面，滚动到最后即可看到各个平台的 Python 安装包。



Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		f215fa2f55a78de739c1787ec56b2bcd	23978360	SIG
XZ comp	se		b3fb85fd479c0bf950c626ef80cacb57	17828408	SIG
macOS 6		for OS X 10.9 and later	d1b09665312b6b1f4e11b03b6a4510a3	29051411	SIG
Windows			f6bbf64cc36f1de38fbf61f625ea6cf2	8480993	SIG
Windows		for AMD64/EM64T/x64	4d091857a2153d9406bb5c522b211061	8013540	SIG
Windows		for AMD64/EM64T/x64	3e4c42f5ff8fcdbe6a828c912b7afdb1	27543360	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	662961733cc947839a73302789df6145	1363800	SIG
Windows x86 embeddable zip file	Windows		980d5745a7e525be5abf4b443a00f734	7143308	SIG
Windows x86 executable installer	Windows		2d4c7de97d6fcd8231fc3decbf8abf79	26446128	SIG
Windows x86 web-based installer	Windows		d21706bdac544e7a968e32bbb0520f51	1325432	SIG

图 2 找到源码包地址

在“Gzipped source tarball”处单击鼠标右键，从弹出菜单中选择“复制链接地址”，即可得到`.tgz`格式的源码压缩包地址。

然后执行以下命令：

```
$ wget https://www.python.org/ftp/python/3.8.1/Python-3.8.1.tgz
```

解压源码包：

```
$ tar -zxvf Python-3.8.1.tgz
```

2) 编译

使用 make 工具进行编译：

```
$ ./configure  --prefix=/usr/local
$ make&&sudo make install
```

这里的`--prefix=/usr/local`用于指定安装目录（建议指定）。如果不指定，就会使用默认的安装目录。

经过以上几个命令，我们就安装好了 Python，这时就可以进入终端，输入 Python 指令，验证是否已安装成功。

小技巧

`python` 命令默认调用的是 Python 2.x 开发环境，如果你习惯使用 Python 3.x，感觉每次输入 `python3` 命令有点麻烦，那么你可以修改配置，让 `python` 命令转而调用 Python 3.x 开发环境。具体命令如下：

```
$sudo unlink /usr/bin/python
$sudo ln -s /usr/bin/python3.8 /usr/bin/python
```

注意，第二条命令中 Python 3.x 的路径和版本一定要正确。

上述命令执行完成以后，再次在终端输入 `python` 命令，进入的就是 Python 3.8 的交互式开发环境了。

2.3 Mac OS 安装 Python 环境

和 Linux 发行版类似，最新版的 Mac OS X 也会默认自带 Python 2.x。

我们可以在终端（Terminal）窗口中输入 python 命令来检测是否安装了 Python 开发环境，以及安装了哪个版本，如下所示：

```
c.biancheng.net:~ mozhiyan$ python
Python 2.7.10 (default, Jul 30 2016, 18:31:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

可以看到，python 命令能够正常运行，并输出了 Python 的版本信息，这表明当前的 Mac OS X 系统已经自带了 Python 2.7.10。

python 命令默认指向 Python 2.x 开发环境，如果想检测当前 Mac OS X 是否安装了 Python 3.x，可以在终端（Terminal）窗口中输入 python3 命令：

- 如果系统提示 command not found，则说明没有安装 Python 3.x；
- 如果 python3 命令运行成功，并显示出版本信息，则说明已经安装了 Python 3.x。

对于没有安装 Python 3.x 的 Mac OS X，想要安装也非常简单，用户只需要下载安装包，然后一直“下一步”即可，这和 Windows 安装 Python 的过程是非常类似的。

Mac OS X 安装 Python 3.x

Python 官方下载地址：https://www.python.org/downloads/

打开链接，可以看到各个版本的 Python：

Release version	Release date		Click for more
Python 3.8.1	Dec. 18, 2019	Download	Release Notes
Python 2.7.17	Oct. 19, 2019	Download	Release Notes
Python 3.7.5	Oct. 15, 2019	Download	Release Notes
Python 3.8.0	Oct. 14, 2019	Download	Release Notes
Python 3.7.0	June 27, 2018	Download	Release Notes
Python 3.6.5	March 28, 2018	Download	Release Notes
Python 3.5.5	Feb. 5, 2018	Download	Release Notes
Python 2.7.14	Sept. 16, 2017	Download	Release Notes
Python 3.4.7	Aug. 9, 2017	Download	Release Notes

图 1 Python 下载页面截图

点击上图中的版本号或者“Download”按钮进入对应版本的下载页面，滚动到最后即可看到各个平台的 Python 安装包。

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		f215fa2f55a78de739c1787ec56b2bcd	23978360	SIG
XZ compressed source tarball	Source release		b3fb85fd479c0bf950c626ef80cacb57	17828408	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	d1b09665312b6b1f4e11b03b6a4510a3	29051411	SIG
Windows help file	Windows		f6bbf64cc36f1de38fbf61f625ea6cf2	8480993	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	4d091857a2153d9406bb5c522b211061	8013540	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	3e4c42f5ff8fcdbe6a828c912b7afdb1	27543360	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	662961733cc947839a73302789df6145	1363800	SIG
Windows x86 embeddable zip file	Windows		980d5745a7e525be5abf4b443a00f734	7143308	SIG
Windows x86 executable installer	Windows		2d4c7de97d6fcd8231fc3decbf8abf79	26446128	SIG
Windows x86 web-based installer	Windows		d21706bdac544e7a968e32bbb0520f51	1325432	SIG

图 2 各个平台的 Python 安装包

macOS 64-bit installer 即为 Mac OS X 系统的 Python 安装包。点击该链接，下载完成后得到一个 python-3.8.1-macosx10.9.pkg 安装包。

双击 python-3.8.1-macosx10.9.pkg 就进入了 Python 安装向导，然后按照向导一步一步向下安装，一切保持默认即可。

图 1 Python 安装向导

安装完成以后，你的 Mac OS X 上将同时存在 Python 3.x 和 Python 2.x 的运行环境，在终端（Terminal）输入 python 命令将进入 Python 2.x 开发环境，在终端（Terminal）输入 python3 命令将进入 Python 3.x 开发环境。

```
c.biancheng.net:~ mozhiyan$ python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

与 Windows 系统类似，Mac OS 下的 Python 3.x 也会自带 IDLE 简易开发工具，你可以在程序列表中找到它。

图 2 安装完成以后的程序列表

2.4 python 不是内部或外部命令的解决方法

有些读者，在命令行窗口（Linux 称为终端）中输出 python 命令后，却显示“‘python’不是内部或外部命令，也不是可运行的程序或批处理文件”，如图 1 所示：

图 1 输出 python 命令后出错

出现此问题，是因为在当前的路径中找不到 Python.exe 可执行程序，最直接的解决方法就是手动配置环境变量，具体方法如下：

1. 在“计算机”图标上单击右键，然后在弹出的快捷菜单中选择“属性”，在属性对话框中单击“高级系统设置”超链接，会出现如图 2 所示的“系统属性”对话框。

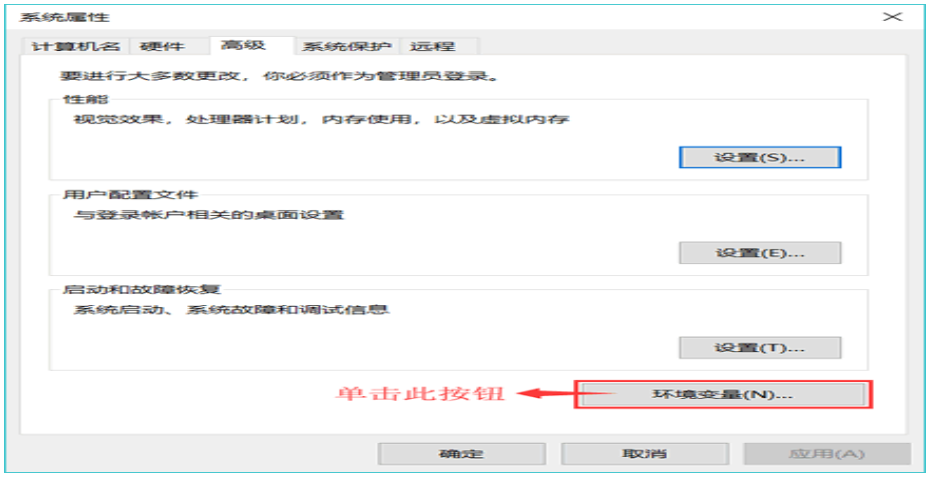


图 2 “系统属性”对话框示意图

2. 如图 2 所示，单击“环境变量”按钮，将弹出“环境变量”对话框，如图 3 所示：

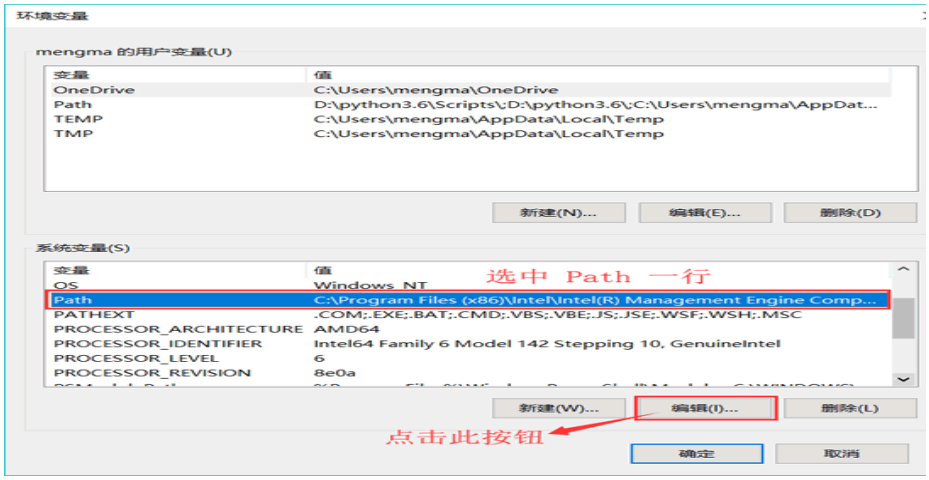


图 3 “环境变量”对话框

3. 在图 3 所示的对话框中，选中“系统变量”栏中的 Path 变量，然后单击“编辑”按钮。此时会弹出“编辑系统变量”的对话框，如图 4 所示：

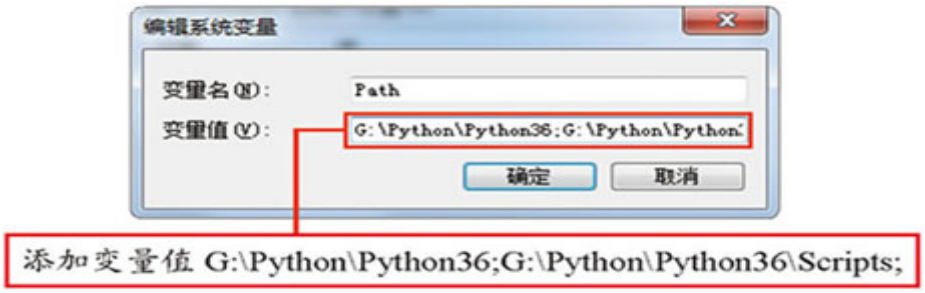


图 4 “编辑系统变量”对话框

4. 正如图 4 所示的这样，我们需要在现有 Path 变量值的最前端，手动添加“G:Python\Python36;G:Python\Python36/Scripts;”，注意，中间和最后的分号不要丢掉，另外将 Python 安装路径修改成自己的，然后单击确定按钮，就完成了对环境变量的设置。

> 再次强调，不要删除系统变量 Path 中原本存在的变量值，并且其中的分号是英文状态下输出的，否则会产生错误。

修改完成后，再在命令行窗口中输入 python 命令，就应该可以成功进入 Python 交互式解释器。

> 本节所介绍的解决方法，是以 Windows 操作系统为例进行说明的，但此方式同样适用于其他的操作系统，只是修改过程略有差异，使用其他操作系统的读者，需要自己找到修改 Path 变量的入口，然后按照此方式修改即可。

2.5 如何运行 Python 程序？

Python 是一种解释型的脚本编程语言，这样的编程语言一般支持两种代码运行方式：

1) 交互式编程

在命令行窗口中直接输入代码，按下回车键就可以运行代码，并立即看到输出结果；执行完一行代码，你还可以继续输入下一行代码，再次回车并查看结果……整个过程就好像我们在和计算机对话，所以称为交互式编程。

2) 编写源文件

创建一个源文件，将所有代码放在源文件中，让解释器逐行读取并执行源文件中的代码，直到文件末尾，也就是批量执行代码。这是最常见的编程方式，也是我们要重点学习的。

本节我们将详细介绍以上两种编程方式。

Python 交互式编程

一般有两种方法进入 Python 交互式编程环境，第一种方法是在命令行工具或者终端（Terminal）窗口中输入 `python` 命令，看到 `>>>` 提示符就可以开始输入代码了，如下所示：

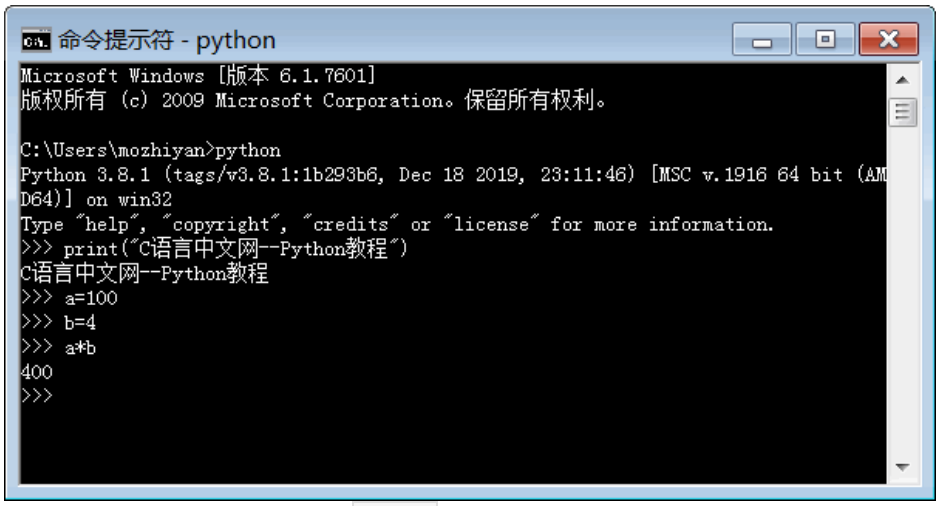


图 1 使用 `python` 命令进入交互式编程环境

第二种进入 Python 交互式编程环境的方法是，打开 Python 自带的 IDLE 工具，默认就会进入交互式编程环境，如下所示：

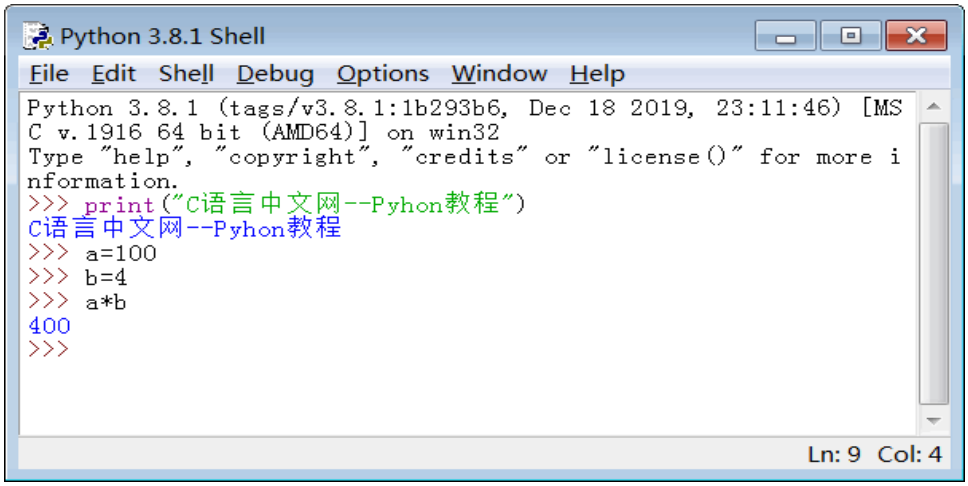


图 2 打开 IDLE 工具进入交互式编程环境

IDLE 支持代码高亮，看起来更加清爽，所以推荐使用 IDLE 编程。

实际上，你可以在交互式编程环境中输入任何复杂的表达式（包括数学计算、逻辑运算、循环语句、函数调用等），Python 总能帮你得到正确的结果。这也是很多非专业程序员喜欢 Python 的一个原因：即使你不是程序员，但只要输入想执行的运算，Python 就能告诉你正确的答案。

从这个角度来看，Python 的交互式编程环境相当于一个功能无比强大的“计算器”，比 Windows、Mac OS X 系统自带的计算器的功能强大多了。

编写 Python 源文件

交互式编程只是用来玩玩而已，真正的项目开发还是需要编写源文件的。

Python 源文件是一种纯文本文件，内部没有任何特殊格式，你可以使用任何文本编辑器打开它，比如：

- Windows 下的记事本程序；
- Linux 下的 Vim、gedit 等；
- Mac OS 下的 TextEdit 工具；
- 跨平台的 Notepad++、EditPlus、UltraEdit 等；
- 更加专业和现代化的 VS Code 和 Sublime Text（也支持多种平台）。

注意，不能使用写字板、Word、WPS 等排版工具编写 Python 源文件，因为排版工具一般都有内置的特殊格式或者特殊字符，这些会让代码变得“乱七八糟”，不能被 Python 解释器识别。

源文件的后缀

Python 源文件的后缀为.py。任何编程语言的源文件都有特定的后缀，例如：

- C 语言源文件的后缀是.c；
- C++源文件的后缀是.cpp；

- JavaScript 源文件的后缀是.js；
- C# 源文件的后缀是.cs；
- Java 源文件的后缀是.java。

后缀只是用来区分不同的编程语言，并不会导致源文件的内部格式发生变化，源文件还是纯文本的。编译器（解释器）、编辑器和用户（程序员）都依赖后缀区分当前源文件属于哪种编程语言。

源文件的编码格式

Python 源文件是一种纯文本文件，会涉及编码格式的问题，也就是使用哪种编码来存储源代码。

Python 3.x 已经将 UTF-8 作为默认的源文件编码格式，所以推荐大家使用专业的文本编辑器，比如 Sublime Text、VS Code、Vim、Notepad++ 等，它们都默认支持 UTF-8 编码。

UTF-8 是跨平台的，国际化的，编程语言使用 UTF-8 是大势所趋。

如果你对编码格式不了了解，请猛击下面的链接学习：

- ASCII 编码，将英文存储到计算机
- GB2312 编码和 GBK 编码，将中文存储到计算机
- Unicode 字符集，将全世界的文字存储到计算机

运行源文件

使用编辑器（我习惯使用 Sublime Text）创建一个源文件，命名为 demo.py，并输入下面的代码：

```
1.  print("Python 教程：http://c.biancheng.net/python/")
2.  a = 100
3.  b = 4
4.  print(a*b)
```

输入完成以后注意保存。

运行 Python 源文件有两种方法：

1) 使用 Python 自带的 IDLE 工具运行源文件。

通过 file -> open 菜单打开 demo.py 源文件，然后在源文件中的菜单栏中选择 Run->Run Module，或者按下 F5 快捷键，就可以执行源文件中的代码了。

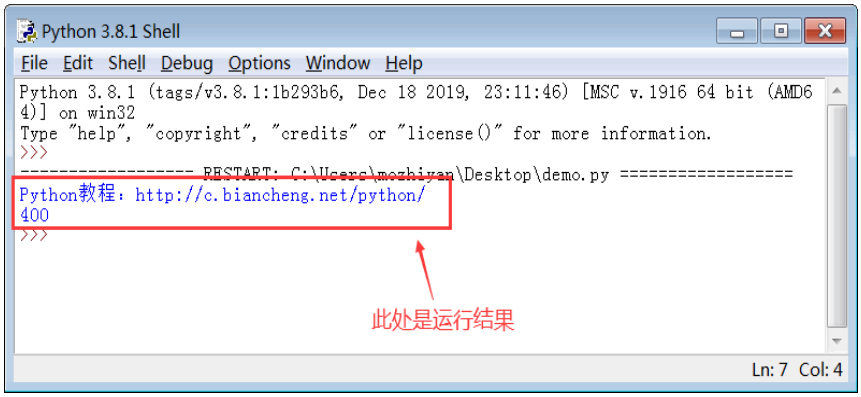


图 3 运行效果截图

更多关于 IDLE 的使用方法，请转到《Python IDLE 使用方法》。

2) 在命令行工具或者终端（Terminal）中运行源文件。

进入命令行工具或者终端（Terminal），切换到 demo.txt 所在的目录，然后输入下面的命令就可以运行源文件：

```
python demo.py
```

运行完该命令，可以立即看到输出结果，如下图所示。

图 4 在 WIndows 命令行工具中运行 Python 源文件

这里简单介绍一下 python 命令，它的语法非常简单，其基本格式如下：

```
python <源文件路径>
```

这里的源文件路径，可以是自盘符（C 盘、D 盘）开始的绝对路径，比如 D:\PythonDemo\demo.py；也可以在执行 python 命令之前，先进入源文件所在的目录，然后只写文件名，也就是使用相对路径。

图 4 演示的是使用相对路径，下面我们再演示一下使用绝对路径：

图 5 python 命令使用绝对路径

> 需要注意的是，Windows 系统不区分大小写，在 Windows 平台上输入源文件路径时可以不用注意大小写。但是类 Unix 系统（Mac OS X、Linux 等）都是区分大小写，在这些平台上输入 Python 源文件路径时一定要注意大小写问题。

2.6 第一个 Python 程序——在屏幕上输出文本

本节我将给大家介绍最简单、最常用的 Python 程序——在屏幕上输出一段文本，包括字符串和数字。

Python 使用 print 函数在屏幕上输出一段文本，输出结束后会自动换行。

在屏幕上输出字符串

字符串就是多个字符的集合，由双引号" "或者单引号' '包围，例如：

```
"Hello World"
"Number is 198"
'Pyhon 教程：http://c.biancheng.net/python/'
```

字符串中可以包含英文、数字、中文以及各种符号。

print 输出字符串的格式如下：

```
print("字符串内容")
```

或者

```
print('字符串内容')
```

字符串要放在小括号()中传递给 print，让 print 把字符串显示到屏幕上，这种写法在 Python 中被称为函数（Function）。

需要注意的是，引号和小括号都必须在英文半角状态下输入，而且 print 的所有字符都是小写。Python 是严格区分大小写的，print 和 Print 代表不同的含义。

print 用法举例：

```
1. print("Hello World!")  #输出英文
2. print("Number is 198")  #输出数字
3. print("Pyhon 教程：http://c.biancheng.net/python/")  #输出中文
```

在 IDLE 下的演示效果：

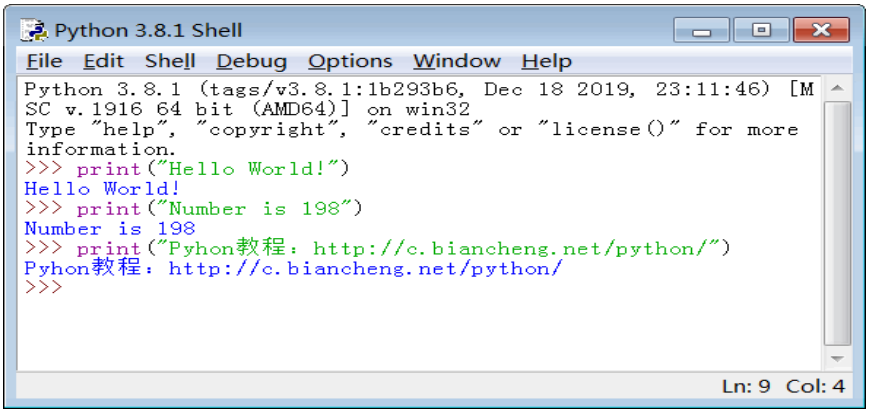


图 1 print 函数用法举例

也可以将多段文本放在一个 print 函数中：

```
print(
    "Hello World!"
    "Number is 198"
    "http://c.biancheng.net/python/"
);

print("Hello World!" "Python is great!" "Number is 198.")

print(
    "Hello World!\n"
    "Number is 198\n"
    "http://c.biancheng.net/python/"
);
```

注意，同一个 print 函数的字符串之间不会自动换行，加上\n 才能看到换行效果。

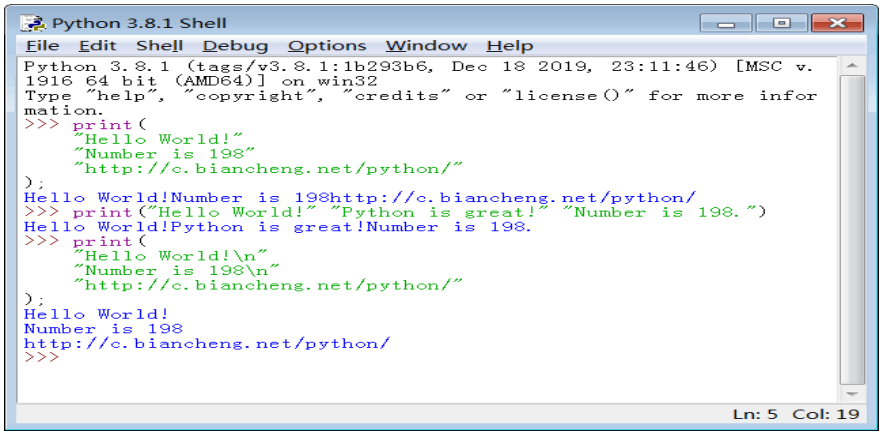


图 2 将多个字符串放在一个 print 中

对分号的说明

有编程经验的读者应该知道，很多编程语言（比如 C 语言、C++、Java 等）都要求在语句的最后加上分号;，用来表示一个语句的结束。但是 Python 比较灵活，它不要求语句使用分号结尾；当然也可以使用分号，但并没有实质的作用（除非同一行有更多的代码），而且这种做法也不是 Python 推荐的。

修改上面的代码，加上分号：

```
print(198);
print("Hello World!"); print("Python is good!");
print("Pyhon 教程：http://c.biancheng.net/python/");
```

运行结果：

```
198
Hello World!
Python is good!
Pyhon 教程：http://c.biancheng.net/python/
```

注意第 2 行代码，我们将两个 print 语句放在同一行，此时必须在第一个 print 语句最后加分号，否则会导致语法错误。

对 Python 2.x 的说明

Python 3.x 要求在使用函数时加上小括号()，但是以前的 Python 2.x 版本可以省略小括号，也即是写成下面的样子：

```
print 198
print "Hello World!";  #末尾也可以加上分号
print "Pyhon 教程：http://c.biancheng.net/python/"
```

我建议大家加上小括号，这样写比较容易理解，而且兼容性好。

在屏幕上输出数字

print 除了能输出字符串，还能输出数字，将数字或者数学表达式直接放在 print 中就可以输出，如下所示：

```
print( 100 )
print( 65 )
print( 100 + 12 )
print( 8 * (4 + 6) )
```

注意，输出数字时不能用引号包围，否则就变成了字符串。下面的写法就是一个反面教材，数学表达式会原样输出：

```
print("100 + 12")
```

运行结果是 `100 + 12`，而不是 112。

另外，和输出字符串不同，不能将多个数字放在一个 print 函数中。例如，下面的写法就是错误的：

```
print( 100 12 95 );
print(
    80
    26
    205
);
```

总结

Python 程序的写法比较简单，直接书写功能代码即可，不用给它套上“外壳”。下面我们分别使用 C 语言、Java 和 Python 输出 C 语言中文网的网址，让大家对比感受一下。

使用 C 语言：

```
#include <stdio.h>
int main()
{
    puts("http://c.biancheng.net/");
    return 0;
}
```

使用 Java：

```
public class HelloJava {
    public static void main(String[] args) {
        System.out.println("http://c.biancheng.net/");
    }
}
```

使用 Python：

```
print("http://c.biancheng.net/")
```

2.7 IDE（集成开发环境）是什么

IDE 是 Intergreated Development Environment 的缩写，中文称为集成开发环境，用来表示辅助程序员开发的应用软件，是它们的一个总称。

通过前面章节的学习我们知道，运行 C 语言（或 Java 语言）程序必须有编译器，而运行 Python 语言程序必须有解释器。在实际开发中，除了运行程序必须的工具外，我们往往还需要很多其他辅助软件，例如语言编辑器、自动建立工具、除错器等等。这些工具通常被打包在一起，统一发布和安装，例如 PythonWin、MacPython、PyCharm 等，它们统称为集成开发环境（IDE）。

因此可以这么说，集成开发环境就是一系列开发工具的组合套装。这就好比台式机，一个台式机的核心部件是主机，有了主机就能独立工作了，但是我们在购买台式机时，往往还要附带上显示器、键盘、鼠标、U 盘、摄像头等外围设备，因为只有主机太不方便了，必须有外设才能玩的爽。

> 需要注意的是，虽然有一些 IDE 支持多种程序语言的开发（如 Eclipse、NetBeans、VS），但通常来说，IDE 主要还是针对某一特定的程序语言而量身打造的（如 VB）。

一般情况下，程序员可选择的 IDE 类别是很多的，比如说，用 Python 语言进行程序开发，既可以选用 Python 自带的 IDLE，也可以选择使用 PyCharm 和 Notepad++ 作为 IDE。并且，为了称呼方便，人们也常常会将集成开发环境称为编译器或编程软件，对此读者没必要较真儿，就把它当做“乡间俗语”吧。

2.8 Python IDE 有哪些，哪款适合初学者？

百度搜索“Python IDE”会发现支持 Python 编程的 IDE 有很多，那么对于零基础的初学者，应该使用哪款 IDE 呢？

我个人推荐初学者使用 Python 自带的 IDLE。因为 IDLE 的使用方法很简单，非常适合初学者入门。本教程中使用的也是 Python 自带的 IDLE。

当然，除了 IDLE，还有很多其他的 IDE 供大家选择，这里列出常用的几个，喜欢探索的读者可自行安装使用。

PyCharm

这是由 JetBrains 公司开发的一款 Python 开发工具，在 Windows、Mac OS 和 Linux 操作系统中都可以使用。

PyCharm 具有语法高亮显示、Project（项目）管理代码跳转、智能提示、自动完成、调试、单元测试以及版本控制等一般开发工具都具有的功能，除此之外，它还支持 Django（Python 的 Web 开发框架）框架下进行 Web 开发。

PyCharm 的主窗口如图 1 所示。

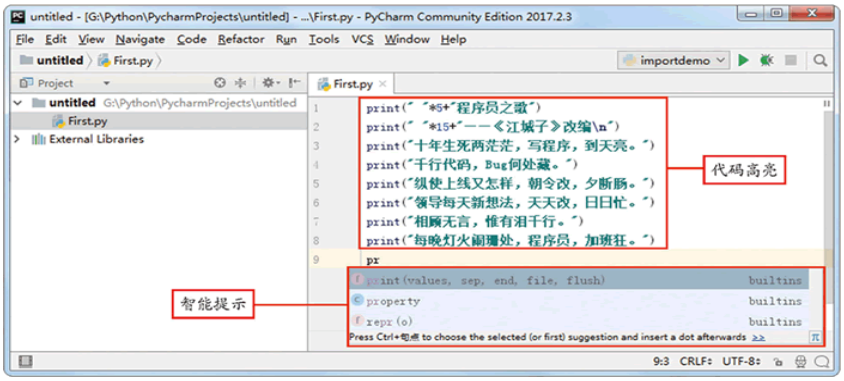


图 1 PyCharm 主窗口示意图

PyCharm 开发工具可通过其官方网站（http://www.jetbrains.com/pycharm/）下载获取。需要注意的是，该网站提供了 2 个版本，一个是社区版（免费并且提供源代码，适合多数读者），另一个是专业版（免费试用）。

有关 PyCharm 下载和安装，可阅读《PyCharm 下载和安装教程》一节，至于如何使用 PyCharm 运行 Python 程序，可阅读《PyCharm 运行 Python 程序》一节。

Eclipse+PyDev

Eclipse 是一个开源的、基于 Java 的可扩展开发平台，最初主要用于 Java 语言的开发。该平台可通过安装不同的插件，进行不同语言的开发。

PyDev 是一款功能强大的 Eclipse 插件，它可以提供语法高亮、语法分析、语法错误提示，以及大纲视图显示

导入的类、库和函数、源代码内部的超链接、运行和调试等功能。

当 Eclipse 在安装 PyDev 插件后，就可以进行 Python 应用开发。其开发界面如图 2 所示。

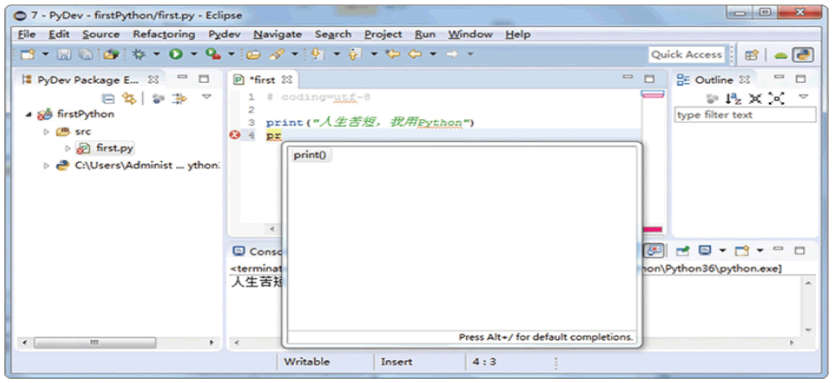


图 2 Eclipse+PyDev 开发界面

除此之外，还有 PythonWin（只针对 Win32 平台）、MacPython IDE（PythonWin 对应的 Mac 版本）、Emacs 和 Vim（功能强大的文本编辑器，可以用来编写 Python 程序）等，都可以作为执行 Python 程序的 IDE。

有关 Eclipse+PyDev 下载和安装，可阅读《Eclipse+PyDec 下载和安装教程》一节，至于如何使用安装有 PyDev 插件的 Eclipse 编写并运行 Python 程序，可阅读《Eclipse+PyDec 运行 Python 程序》一节。

Visual Studio Code

Visual Studio Code，简称 VS Code，是微软公司开发的一款轻量级 IDE。和 PyCharm 一样，它也支持在 Windows、Linux 和 macOS 平台上运行。

VS Code 支持几乎所有主流开发语言的语法高亮、智能代码补全、自定义热键、括号匹配等功能，支持使用插件进行功能扩展，还针对网页开发和云端应用开发做了优化。

值得一提的是，使用 VS Code 编写 Python 代码，无需向其它编译器那样，通过创建项目来管理源代码文件，在 VS Code 中可以直接创建 Python 源代码文件。VS code 的开发界面如图 3 所示。

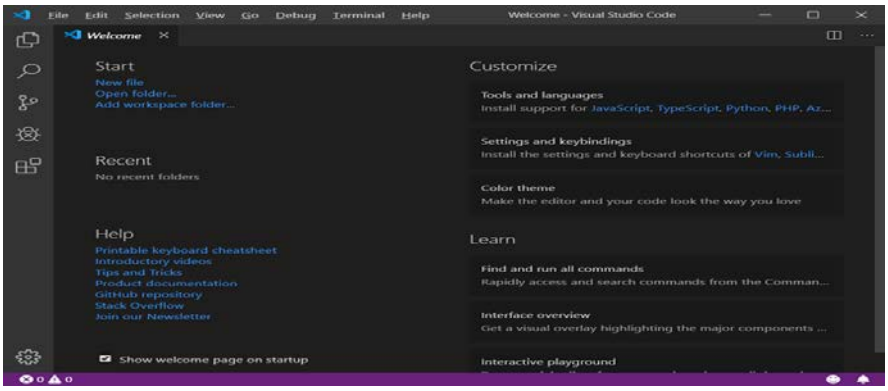


图 3 VS Code 开发界面

有关 VS Code 下载和安装，可阅读《VS Code 下载和安装教程》一节，至于如何使用 VS Code 编写并运行 Python 程序，可阅读《VS Code 运行 Python 程序》一节。

Microsoft Visual Studio

Microsoft Visual Studio，简称 VS，也是 Microsoft（微软）公司开发的一款 IDE。它可用于进行 C# 和 ASP.NET 等应用的开发，也可以作为 Python 的开发工具，只需要在安装时，选择安装 PTVS 插件即可。

PTVS 插件是一个开源插件，它支持编辑、浏览、智能感知、混合 Python/C++ 调试、Django 等，适用于 Windows、Linux 和 Mac OS 客户端的云计算。

当 VS 安装 PTVS 插件之后，就可以进行 Python 应用开发了，其开发界面如图 4 所示。

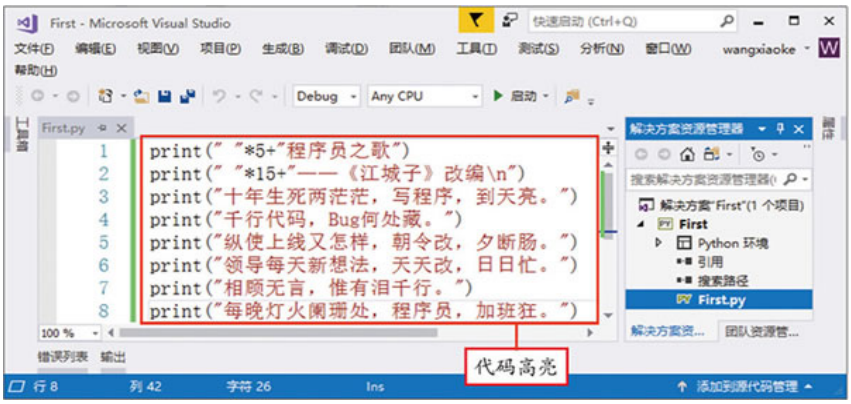


图 4 应用 VS 开发 Python 项目

有关 VS 下载和安装教程，可阅读《Visual Studio 下载和安装教程》一节，至于如何使用 VS 编写并运行 Python 程序，可阅读《Visual Studio 运行 Python 程序》一节。

2.9 Python IDLE 使用方法详解（包含常用快捷键）

在安装 Python 后，会自动安装一个 IDLE，它是一个 Python Shell (可以在打开的 IDLE 窗口的标题栏上看到），程序开发人员可以利用 Python Shell 与 Python 交互。

本节将以 Windows7 系统中的 IDLE 为例，详细介绍如何使用 IDLE 开发 Python 程序。

单击系统的开始菜单，然后依次选择“所有程序 -> Python 3.6 -> IDLE (Python 3.6 64-bit)”菜单项，即可打开 IDLE 窗口，如图 1 所示。

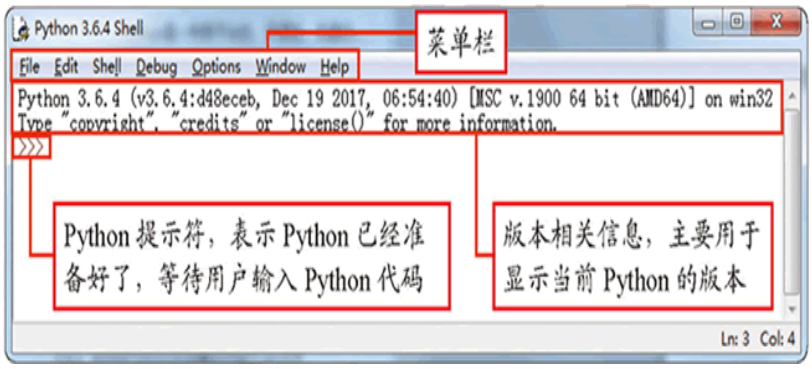


图 1 IDLE 主窗口

前面我们已经应用 IDLE 输出了简单的语句，但在实际开发中，通常不能只包含一行代码，当需要编写多行代码时，可以单独创建一个文件保存这些代码，在全部编写完成后一起执行。具体方法如下：

1. 在 IDLE 主窗口的菜单栏上，选择“File -> New File”菜单项，将打开一个新窗口，在该窗口中，可以直接编写 Python 代码。
   在输入一行代码后再按下 <Enter> 键，将自动换到下一行，等待继续输入，如图 2 所示。

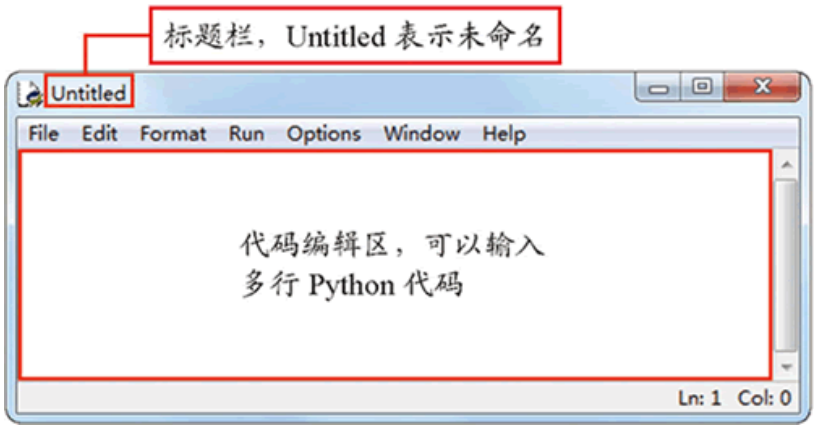


图 2 新创建的 Python 文件窗口

2. 在代码编辑区中，编写多行代码。例如，输出由宋词《江城子》改编而成的《程序员之歌》，代码如下：

```
1. print(" "*5+"程序员之歌")
```

```
print(" "*15+"——《江城子》改编\n")
print("十年生死两茫茫，写程序，到天亮，")
print("千行代码，Bug 何处藏。")
print("纵使上线又怎样，朝令改，夕断肠。")
print("领导每天新想法，天天改，日日忙。")
print("相顾无言，惟有泪千行")
print("每晚灯火阑珊处，程序员，加班狂。")
```

编写代码后的 Python 文件窗口如图 3 所示。

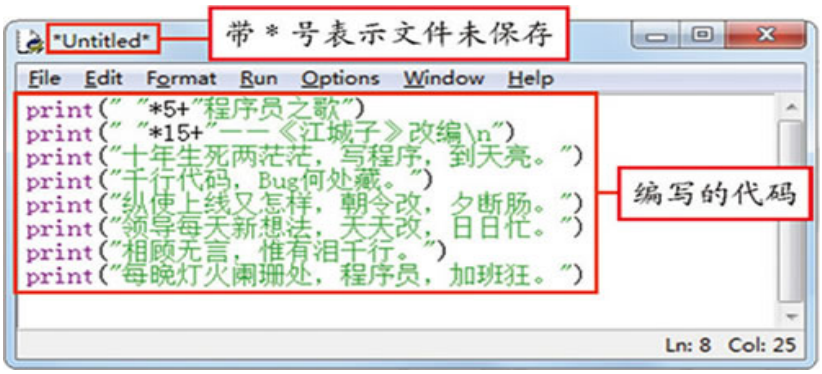


图 3 编写代码后的 Python 文件窗口

3. 按下快捷键 <Ctrl+S> 保存文件，这里将文件名称设置为 demo.py。其中，.py 是 Python 文件的扩展名。在菜单栏中选择“Run -> Run Module”菜单项（也可以直接按下快捷键 <F5>），运行程序，如图 4 所示。

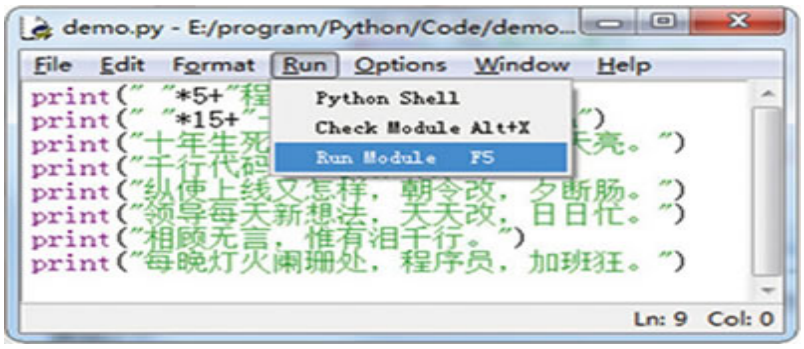


图 4 运行程序

4. 运行程序后，将打开 Python Shell 窗口显示运行结果，如图 5 所示。

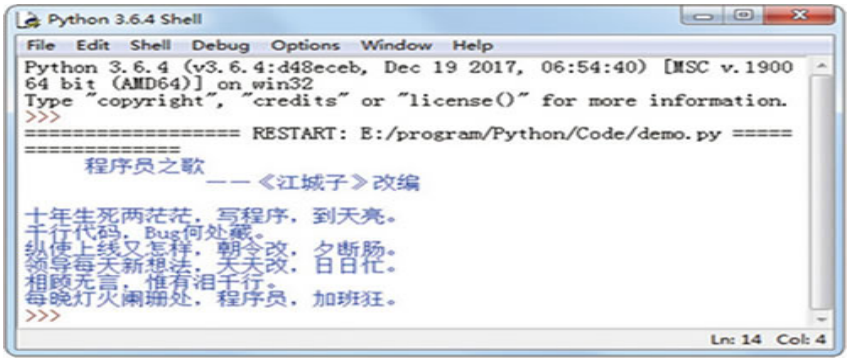


图 5 运行结果

Python IDLE 常用快捷键

在程序开发过程中，合理使用快捷键不但可以减少代码的错误率，而且可以提高开发效率。在 IDLE 中，可通过选择“Options -> Configure IDLE”菜单项，在打开的“Settings”对话框的‘Keys”选项卡中查看，但是该界面是英文的，不便于查看。为方便读者学习，表 6 列出了 IDLE 中一些常用的快捷键。

表 6 IDLE 提供的常用快捷键

快提键	说 明	适用范围
F1	打开 Python 帮助文档	Python 文件窗口和 Shell 均可用
Alt+P	浏览历史命令（上一条）	仅 Python Shell 窗口可用
Alt+N	浏览历史命令（下一条）	仅 Python Shell 窗口可用
Alt+/	自动补全前面曾经出现过的单词，如果之前有多个单词具有相同前缀，可以连续按下该快捷键，在多个单词中间循环选择	Python 文件窗口和 Shell 窗口均可用
Alt+3	注释代码块	仅 Python 文件窗口可用
Alt+4	取消代码块注释	仅 Python 文件窗口可用
Alt+g	转到某一行	仅 Python 文件窗口可用
Ctrl+Z	撤销一步操作	Python 文件窗口和 Shell 窗口均可用
Ctrl+Shift+Z	恢复上一次的撤销操作	Python 文件窗口和 Shell 窗口均可用
Ctrl+S	保存文件	Python 文件窗口和

		Shell 窗口均可用
Ctrl+]	缩进代码块	仅 Python 文件窗口可用
Ctrl+[	取消代码块缩进	仅 Python 文件窗口可用
Ctrl+F6	重新启动 Python Shell	仅 Python Shell 窗口可用

由于 IDLE 简单、方便，很适合联系，因此本教程如果没有特殊说明，均使用 IDLE 作为开发工具。

2.10 PyCharm 下载和安装教程（包含配置 Python 解释器）

PyCharm 是 JetBrains 公司（www.jetbrains.com）研发，用于开发 Python 的 IDE 开发工具。图 1 所示为 JetBrains 公司开发的多款开发工具，其中很多工具都好评如潮，这些工具可以编写 Python、C/C++、C#、DSL、Go、Groovy、Java、JavaScript、Objective-C、PHP 等编程语言。

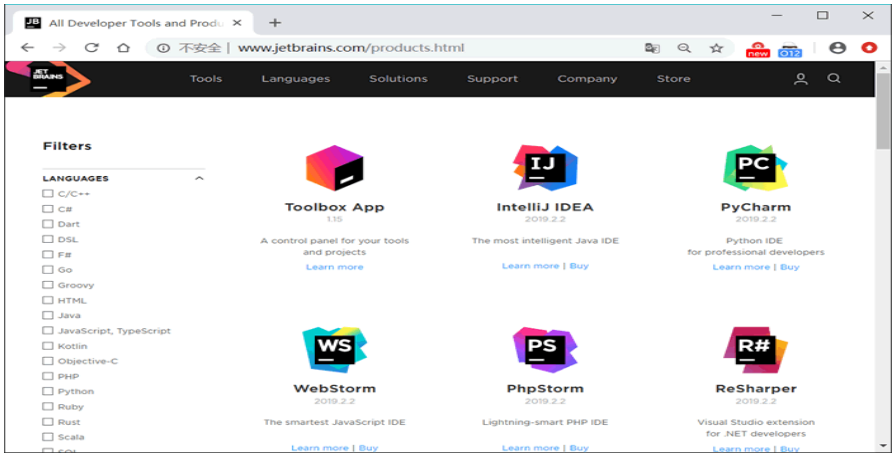


图 1 JetBrains 开发工具

PyCharm 下载和安装

进入 PyCharm 官方下载页面（如图 2 所示），可以看到 PyCharm 有 2 个版本，分别是 Professional（专业版）和 Community（社区版）。其中，专业版是收费的，可以免费试用 30 天；而社区版是完全免费的。

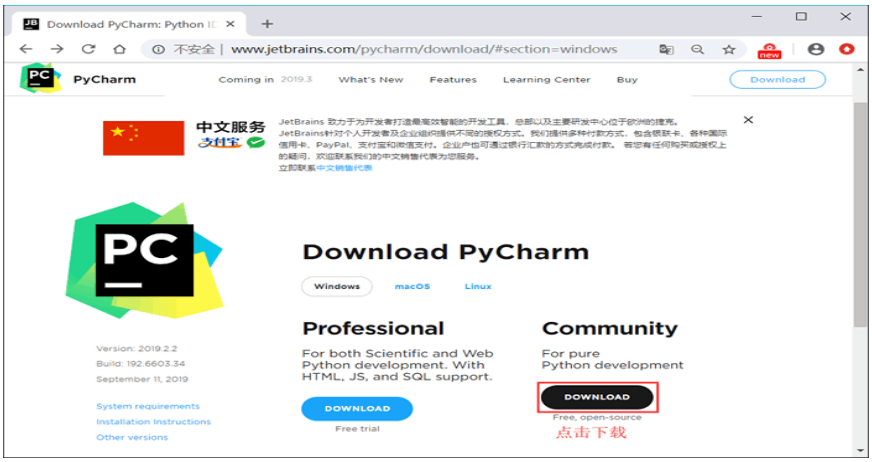


图 2 PyCharm 官方下载页面

强烈建议初学者使用社区版，更重要的是，该版本不会对学习 Python 产生任何影响。

根据图 2 所示点击“下载”按钮，等待下载完成。下载完成后，读者会得到一个 PyCharm 安装包（本节下载的是 pycharm-community-2019.2.2 版本）。双击打开下载的安装包，正式开始安装（如图 3 所示）。

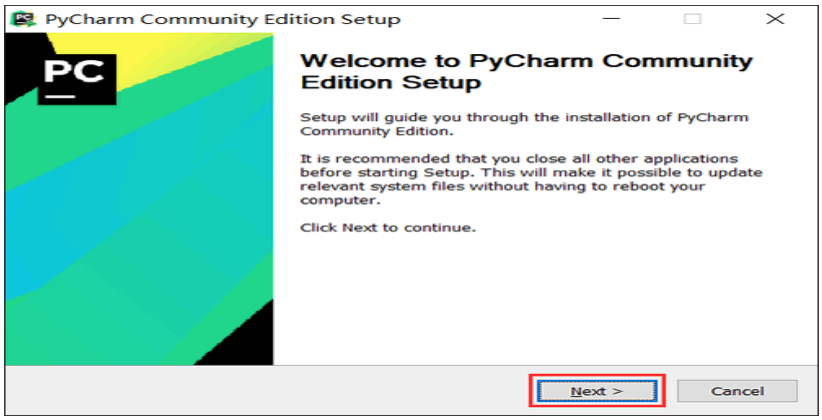


图 3 开始安装界面

直接选择“Next”，可以看到如图 4 所示的对话框，这里是设置 PyCharm 的安装路径，建议不要安装在系统盘（通常 C 盘是系统盘），这里选择安装到 E 盘。

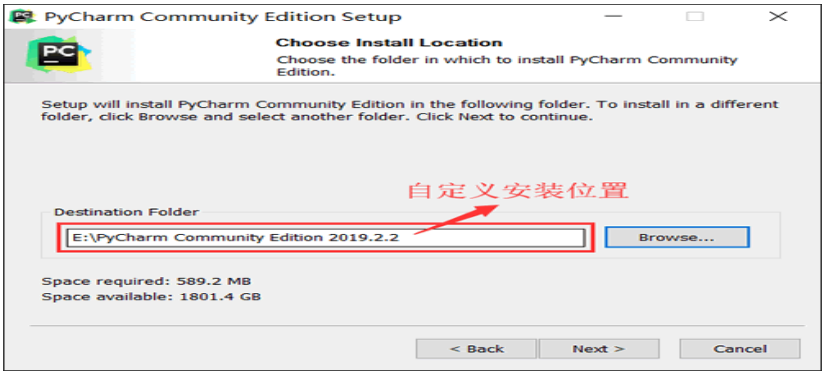


图 4 设置 PyCharm 安装路径

继续点击“Next”，这里需要进行一些设置，可根据图 5 所示，自行选择需要的功能，若无特殊需求，按图中勾选即可；

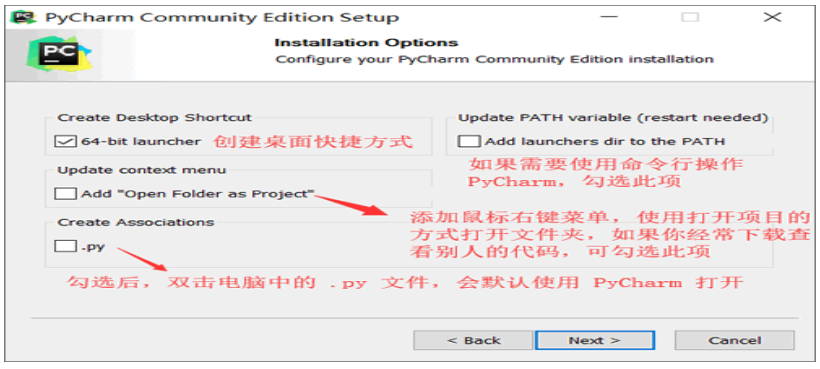


图 5 安装设置对话框

继续点击“Next”，达到图 6 所示的对话框，这里选择默认即可，点击 “Install”，并等待安装进度条达到 100%，PyCharm 就安装完成了。

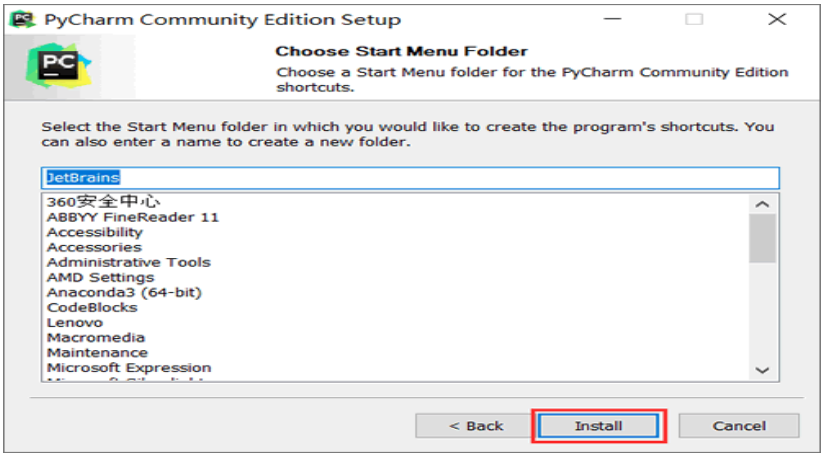


图 6 选择开始菜单文件

需要注意的是，首次启动 PyCharm，会自动进行配置 PyCharm 的过程（选择 PyCharm 界面显式风格等等），读者可根据自己的喜好进行配置，由于配置过程非常简单，这里不再给出具体图示。读者也可以直接退出，即表示全部选择默认配置。

PyCharm 配置 Python 解释器

首先安装 PyCharm 完成之后，打开它会显示如下所示的界面：

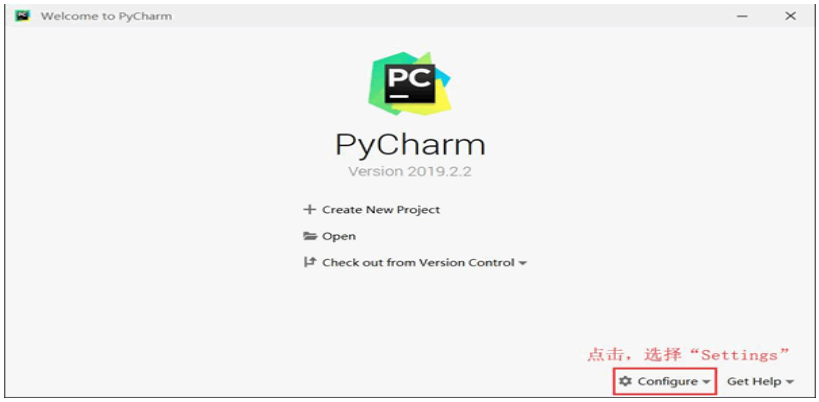


图 8 PyCharm 初始化界面

在此界面中，可以手动给 PyCharm 设置 Python 解释器。点击图 8 所示的 Configure 选项，选择“Settings”，进入图 9 所示的界面。

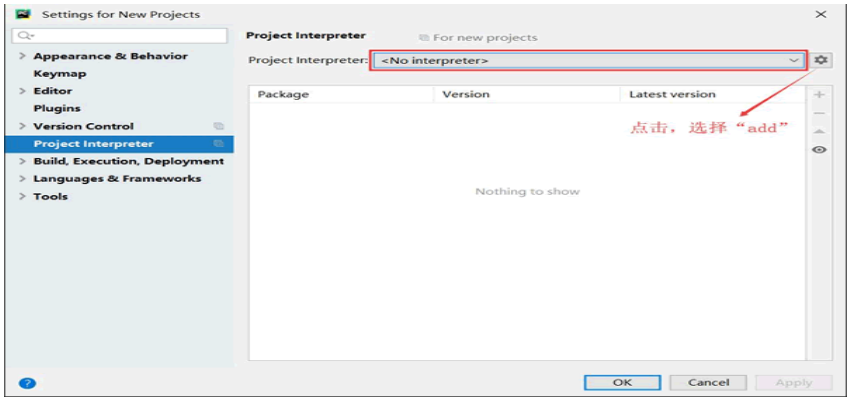


图 9 设置 Python 解释器界面

可以看到，"No interpreter" 表示未设置 Python 解释器，这种情况下，可以按图 9 所示，点击设置按钮，选择"add"，此时会弹出图 10 所示的窗口。

图 10 添加 Python 解释器界面

按照图 10 所示，选择“System Interpreter”（使用当前系统中的 Python 解释器），右侧找到你安装的 Python 目录，并找到 python.exe，然后选择“OK”。此时显式界面会自动跳到图 9 所示的界面，并显示出可用的解释器，如图 11 所示，再次点击“OK”。

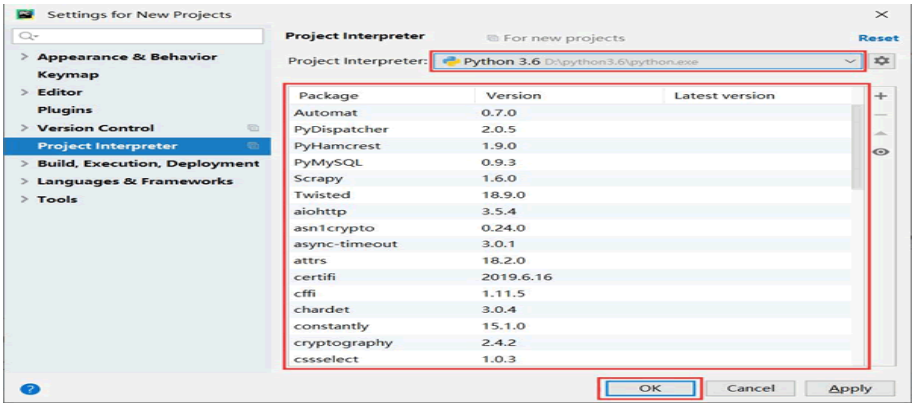


图 11 添加 Python 解释器界面

等待 PyCharm 配置成功，它会再次回到图 8 所示的界面，由此就成功的给 PyCharm 设置好了 Python 解释器。

关于如何使用 PyCharm 运行 Python 程序，可猛击《PyCharm 运行 Python 程序》一文详细了解。

2.11 PyCharm 运行 Python 程序

《第一个 Python 程序》一节中，分别介绍了如何使用 Python IDLE、Python Shell 以及 Sublime Text 编辑并运行 Python 程序。但是，如果要开发复杂的项目，使用 PyCharm、VS Code 等这些强大的 IDE 应该是更不错的选择。

本节仍以第一个 Python 程序为例，给大家介绍如何使用 PyCharm 创建 Python 项目，以及如何编写并运行 Python 程序。

PyCharm 创建 Python 项目

PyCharm 中，往往是通过项目来管理 Python 源代码文件的。虽然对于第一个 Python 程序来说，创建项目来管理似乎有些“大材小用”，但对于初学者来说，学会创建 Python 项目是非常有必要的。

PyCharm 创建项目的步骤是这样的，首先打开 PyCharm，会显示出如图 1 所示的欢迎界面。

图 1 PyCharm 欢迎界面

在该界面中点击“Create New Project”（创建一个新项目），打开如图 2 所示的 Location 对话框，在该对话框中输入项目名称（例如 demo）。

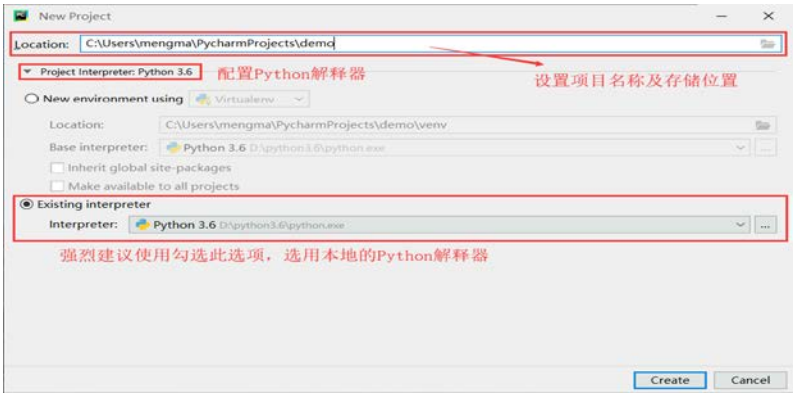


图 2 设置项目名称并配置 Python 解释器

另外，强烈建议初学者单击图中所示的三角按钮，展开 Python 解释器设置界面，按图中所示选用本地的

Python 解释器。默认情况下，PyCharm 会选用第一种配置方式，它会自动为项目配置虚拟环境，即向项目中添加运行 Python 程序所必备的文件（例如 Python 解释器和标准库文件），但这些文件对于初学者来说，是晦涩难懂的，对 Python 入门没有任何帮助。

输入好项目名称，并配置好 Python 解释器之后，就可以单击“Create”按钮创建项目。创建好的项目如图 3 所示。

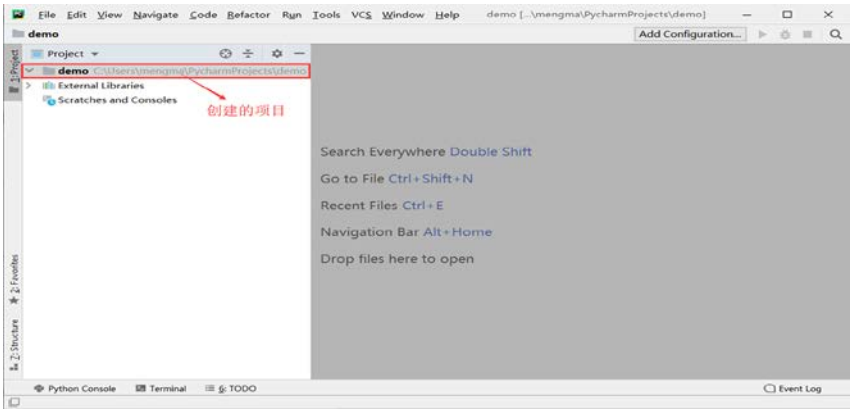


图 3 创建好的 Python 项目

可以看到，使用本地 Python 解释器，创建的项目是空的。

PyCharm 项目中创建 Python 文件

项目创建完成之后，就可以创建一个 Python 代码文件了，具体操作如下。

首先，在创建好项目的基础上，右键选中该项目，并依次选择“New->Python File” 菜单，如图 4 所示。

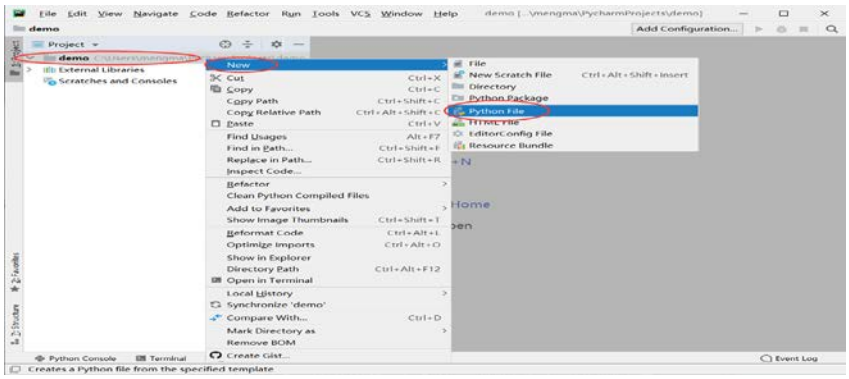


图 4 新建 Python 文件流程

此时，就会打开新建 Python 文件的对话框，输入要新建 Python 文件的名字（如 hello），如图 5 所示。

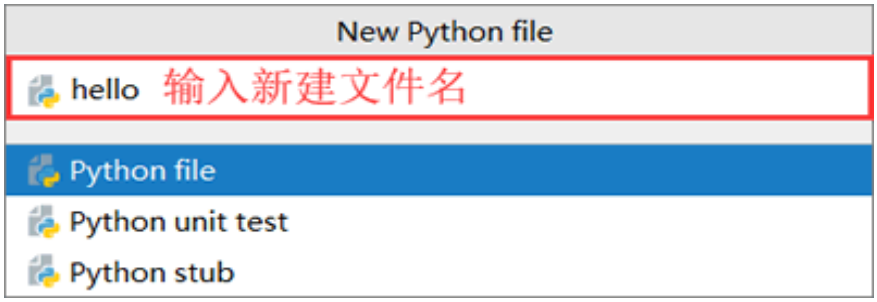


图 5 新建 Python 文件对话框

然后点击“OK”按钮或按“Enter”回车键，即可成功创建一个 Python 文件，如图 6 所示。

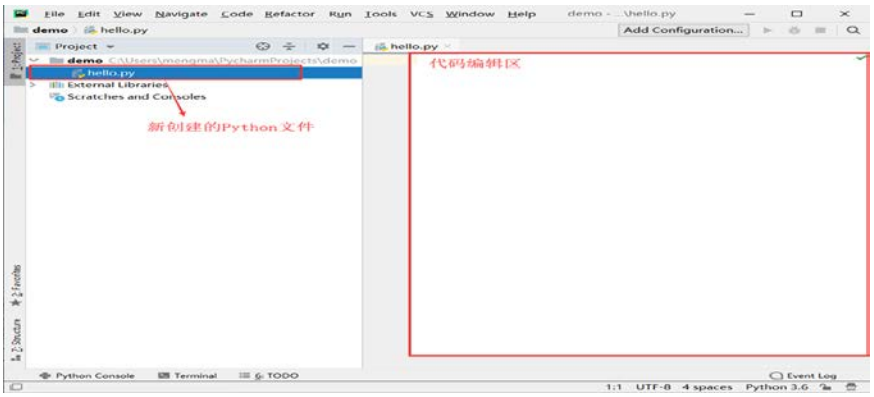


图 6 成功创建 Python 文件

Python 文件创建成功之后，就可以向该文件中编写 Python 程序。例如，在新创建的 hello 文件中编写第一个 Python 程序，如图 7 所示。

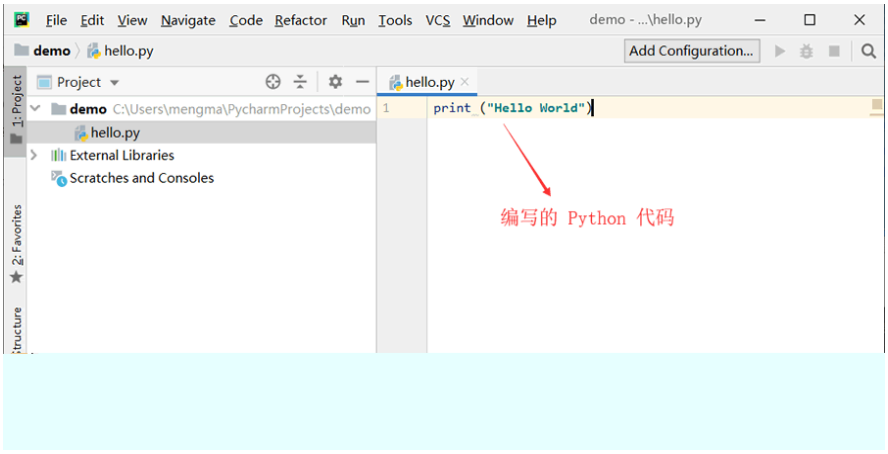


图 7 向 Python 文件中编写代码

运行此 Python 程序也很简单，只需选择左侧工程目录中的 hello.py 文件，并右键选择“Run'hello'”，如图 8 所示。

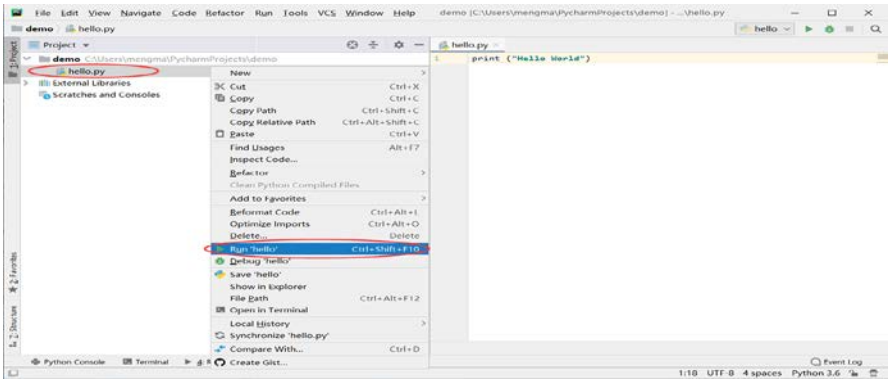


图 8 PyCharm 运行 Python 程序

运行结果如图 9 所示。

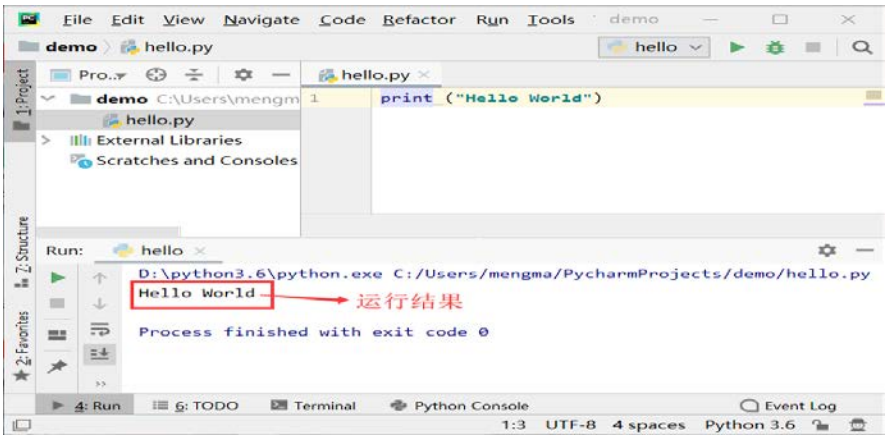


图 9 程序运行结果示意图

提示：如果该程序已经运行过一次，当再次运行时，可直接点击工具栏中的 Run 箭头按钮，或者使用快捷键“Shift + F10”，都可以运行上次的程序。

2.12 Python Eclipse+PyDec 下载和安装教程（超级详细）

Eclipse 是著名的跨平台 IDE 工具，最初 Eclipse 是 IBM 支持开发的免费 Java 开发工具，2001 年 11 月贡献给开源社区，目前它由非盈利软件供应商联盟 Eclipse 基金会管理。

Eclipse 本身也是一个框架平台，它有着丰富的插件，例如 C++、Python、PHP 等开发其他语言的插件。除此之外，Eclipse 是绿色软件，不需要写注册表，卸载非常方便。

安装 Eclipse+PyDev 要比 PyCharm 复杂，大致分为以下 3 个步骤：

1. 安装 JRE（Java 运行环境）或 JDK（Java 开发工具包），Eclipse 是基于 Java 的开发工具，必须有 Java 运行环境才能运行；
2. 下载和安装 Eclipse；
3. 安装 PyDev 插件。

安装 Eclipse 运行环境（JDK）

如果本机中以安装有 JDK，可直接跳过此步，直接安装 Eclipse。
进入 JavaSE 下载界面，这里下载的是 Java SE 13 最新版本（如图 1 所示）。

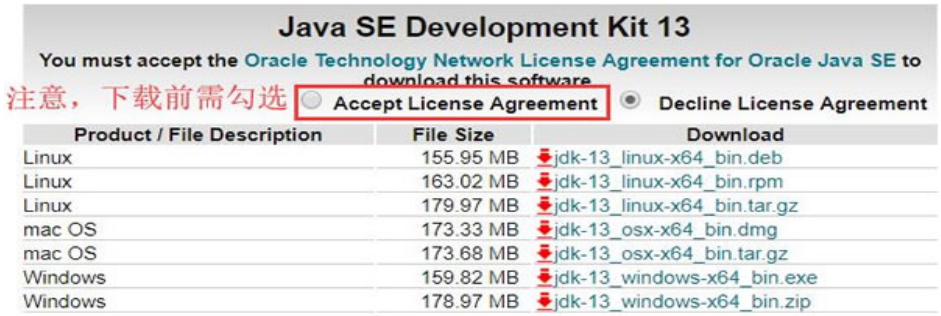


图 1 Java SE 13 下载方式

从图 1 中可以看到，针对不同的平台和操作系统，官方提供了多个版本。读者可根据自己机器的情况，下载合适的版本。由于本机是 Windows 系统，因此可以下载最后两种版本中的任意一个。

注意，.zip 格式是压缩包，下载后无法进行安装，而 .exe 格式是可执行文件，下载后需运行此文件，才能成功安装。本节以下载 exe 格式安装包为例。
下载后，会得到一个 jdk-13_windows-x64_bin.exe 文件，打开此文件，即可看到如图 2 所示的安装界面。

图 2 Java JDK 安装初始界面

直接点击“下一步”，进入图 3 所示的界面，通过点击“更改”按钮，可修改 JDK 的安装路径。

图 3 JDK 修改安装路径界面

继续点击“下一步”，将开始安装 JDK。安装完成后，会转到图 4 所示的界面。由此，JDK 就成功安装啦。

图 4 JDK 成功安装界面

Eclipse 下载和安装

Eclipse 官网提供有多个版本的下载地址，本节将以 2019 年 9 月份推出的最新版为例（如图 5 所示）。

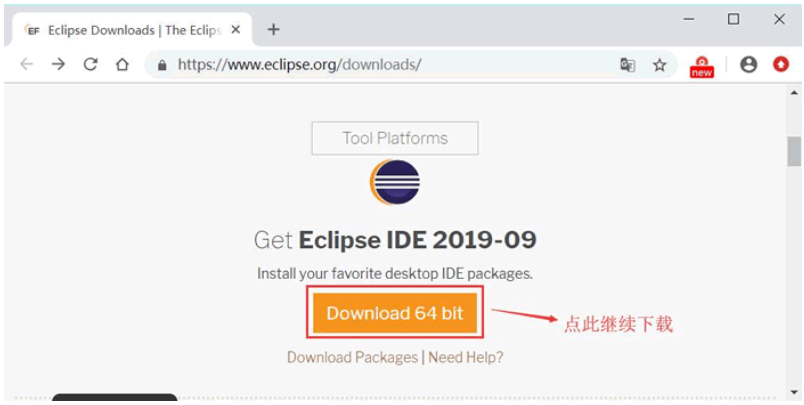


图 5 Eclipse 最新版下载地址

注意，Eclipse 4.9 版本以及之前的版本，都提供有 32 位和 64 位的安装包，而 4.9 之后的版本仅提供 64 位的安装包。如果读者想下载 Eclipse 4.9 以及之前的版本，可访问 https://archive.eclipse.org/eclipse/downloads/。

点击图 5 所示的下载按钮，会转到如图 6 所示的页面，再次点击下载安装即可开始下载。

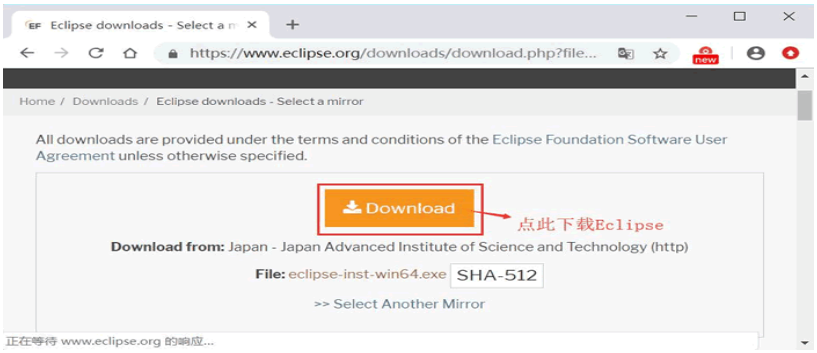


图 6 下载 Eclipse

下载完成后，会得到名为 eclipse-inst-win64.exe 的可执行文件，双击打开后会看到图 7 所示的界面。

图 7 Eclipse 开始安装界面

选择“Eclipse IDE for Java Developers”，进入图 8 所示的界面。

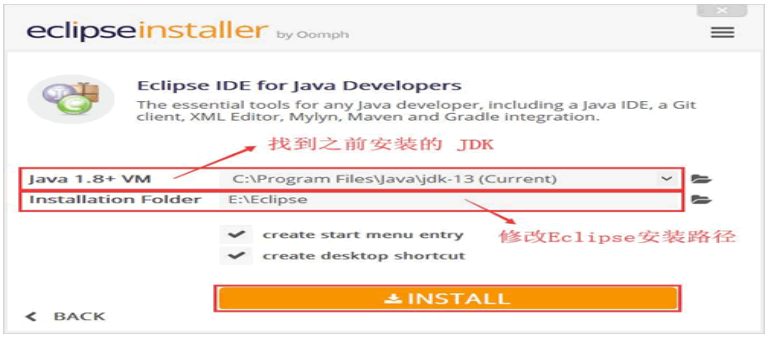


图 8 自定义 Eclipse 安装路径

在图 8 中，我们需要将之前安装的 JDK 导入，并修改 Eclipse 的默认安装路径（建议安装到除系统盘之前的其它盘），之后点击 "INSTALLING"，即可开始安装 Eclipse（如图 9 所示）。

图 9 Eclipse 安装过程示意图

注意，安装期间，可能会弹出选项框，选择“ACCEPT”即可。

安装完成，会出现如下界面，点击 LAUNCH 按钮，即可启动 Eclipse。

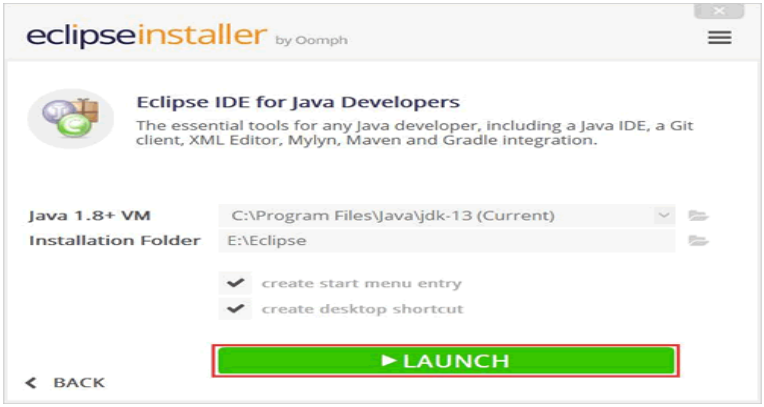


图 10 Eclipse 安装完成示意图

在 Eclipse 启动过程中，会弹出如图 11 所示的选择工作空间对话框。

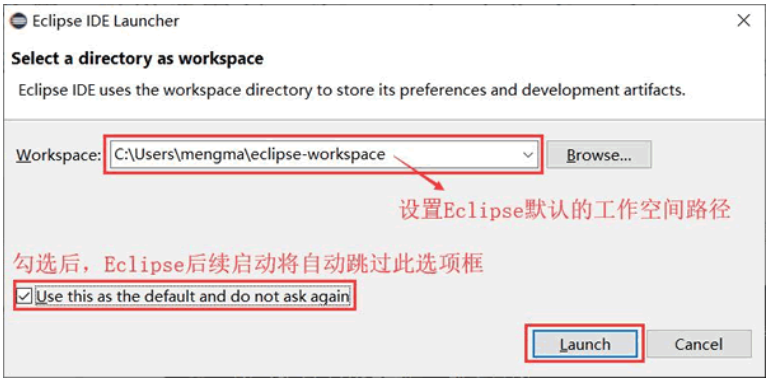


图 11 Eclipse 工作空间对话框

工作空间是用来保存工程的文件夹，默认情况下每次 Eclipse 启动时都需要选择工作空间，当然如果觉得每次启动都选择工作空间比较麻烦，可以勾选“Use this as the default and to not ask again”选项。

设置好工作空间之后，初次启动 Eclipse 会进入图 12 所示的欢迎界面。由此，Eclipse 就成功安装了。

图 12 Eclipse 欢迎界面

Eclipse 安装 PyDev 插件

PyDev 插件不需要我们手动去下载，借助 Eclipse 工具可实现在线安装，具体的安装过程如下。

首先启动 Eclipse，依次选择菜单“Help->Install New Software”，会弹出如图 13 所示的对话框。

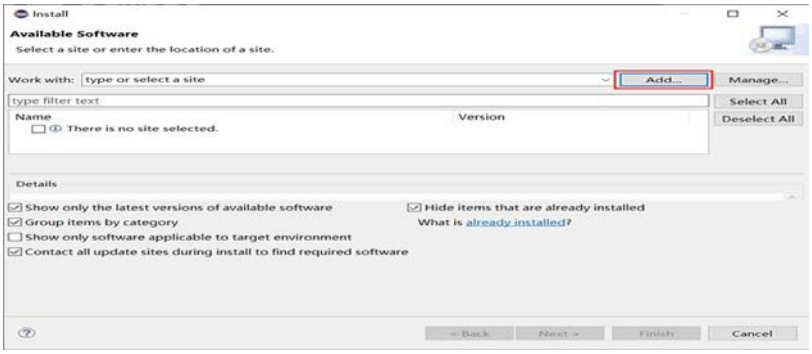


图 13

点击“ADD”，出现如图 14 所示的对话框，在此对话框的 Location 文本框中，输出 PyDev 插件的下载地址（http://pydev.org/updates），然后点击“OK”按钮，Eclipse 就会通过输入的网址查找插件，如果能够找到插件，会出现如图 14 所示的对话框。

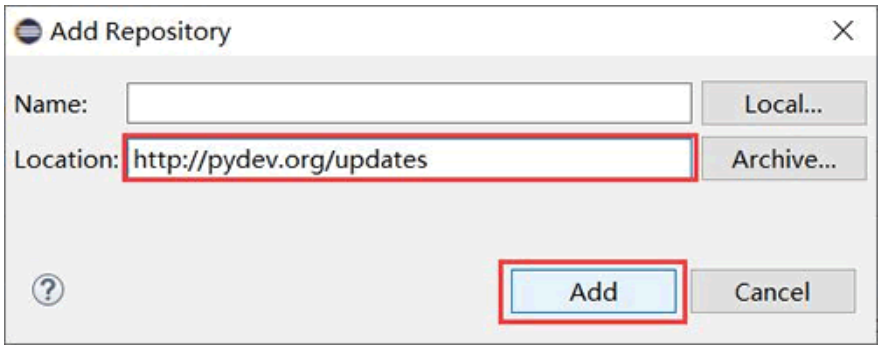


图 14 填写 PyDev 下载地址

选择“PyDev”，点击“Next”，即可开始安装 PyDev 插件（如图 15 所示）。安装完成后，需选择"restart"重启 Eclipse 才能生效。

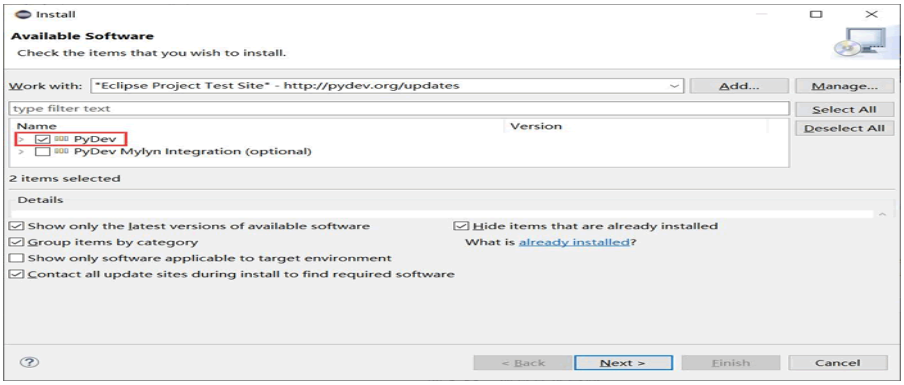


图 15 PyDev 安装示意图

Eclipse 设置 Python 解释器

PyDev 插件安装成功后，还需要设置 Python 解释器。具体步骤为：

- 打开 Eclipse，选择菜单“Window->Preferences”，弹回设置对话框；
- 选择“PyDev->Interpreters->Python Interpreter”，这里可以通过点击"Config first in PATH"按钮，通过在 Path 路径中找到 Python 解释器，也可以点击“Browse for Python/pypy exe”按钮，手动找到 Python 解释器。添加完成后，点击“Apply and Close”按钮即可（如图 16 所示）。

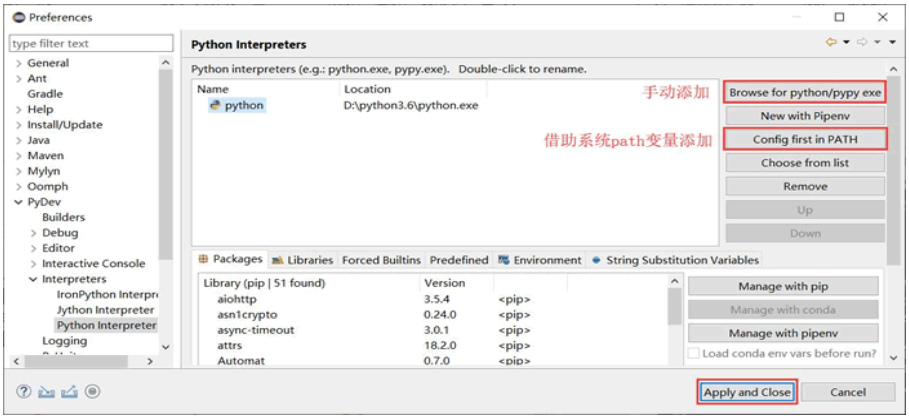


图 16 设置 Python 解释器

关于如何使用 Eclipse 运行 Python 程序，可猛击《Eclipse+PyDec 运行 Python 程序》一文详细了解。

2.13 Eclipse+PyDec 运行 Python 程序

本节仍以第一个 Python 程序为例，继续讲解如何通过 Eclipse + PyDev 实现编写和运行 Python 程序。

Eclipse 创建 Python 项目

和 PyCharm 一样，在 Eclipse 中也是通过项目来管理 Python 源代码文件的，因此需要先创建一个 Python 项目，然后在项目中创建一个 Python 源代码文件。

Eclipse 创建项目的过程是这样的，首先打开 Eclipse，依次选择菜单中“File -> New -> Project...”，如图 1 所示。

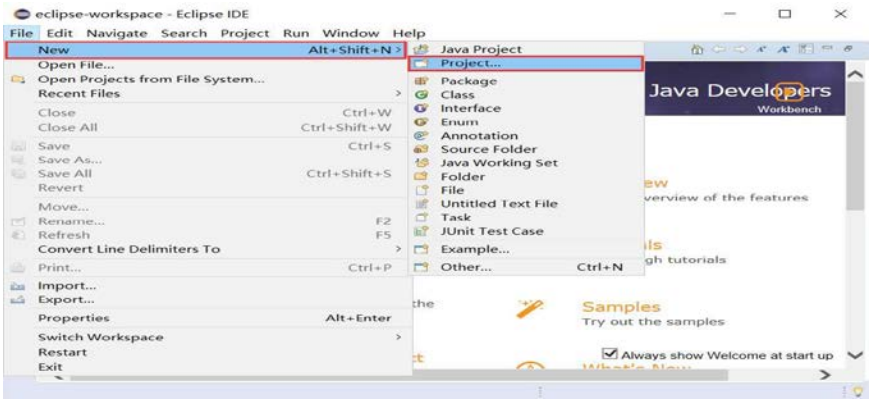


图 1 Eclipse 创建项目

此时会弹出如图 2 所示的对话框，选择“PyDev -> PyDev Project”，然后点击 “Next” 按钮。

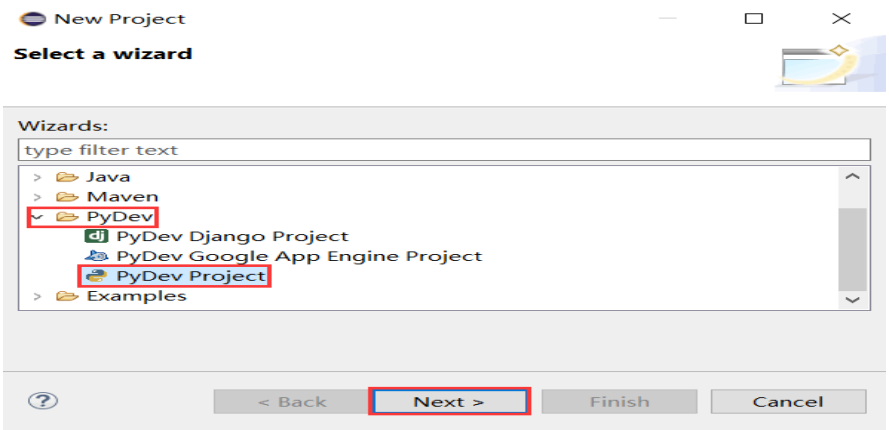


图 2 Select a wizard

弹出如图 3 所示的对话框，这里可以输入项目（例如 Demo），另外建议大家选中“Create 'src' folder and

add it to the PYTHONPATH”选项，这会在项目中增加 src 文件夹，代码文件会放到这个文件夹中，同时会将 src 文件夹添加到 PYTHONPATH 环境变量中。

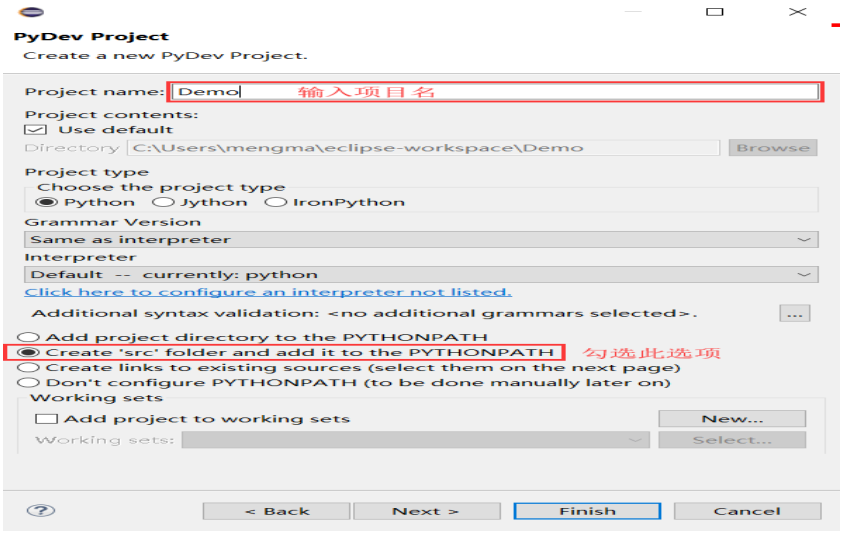


图 3 输入项目名称

如果大家不选择“Create 'src' folder and add it to the PYTHONPATH”选项，则成功创建项目之后，还需要手动创建一个源代码文件夹。

其他保持默认值即可，然后点击“Finish”按钮，即可成功创建一个 Python 项目，如图 4 所示。

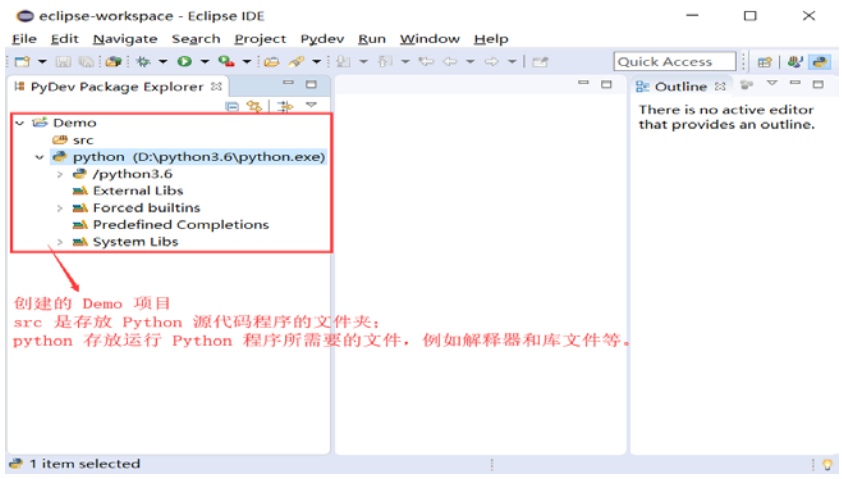


图 4 Eclipse 成功创建 Python 项目

Eclipse 创建 Python 源代码文件

在创建完成项目的基础上，接下来就可以创建 Python 源代码文件了。

选择刚刚创建的 Python 项目，右键选中 src 文件夹，在菜单中依次选择 "New -> PyDev Module"，打开创建文件对话框，如图 5 所示。

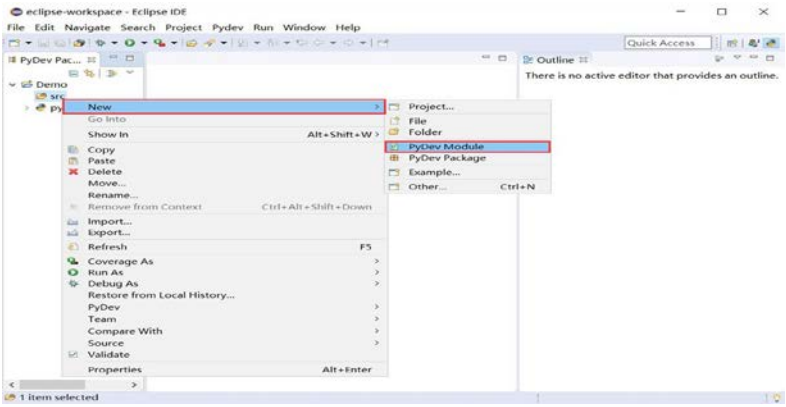


图 5 创建 Python 源代码文件

Module 是模块的意思，在 Python 中，一个模块指的就是一个 Python 源文件。

这里会弹出如图 6 所示的对话框，其中，Name 文本框中输入 Python 源代码文件的名称（例如 demo），然后点击“Finish”按钮。

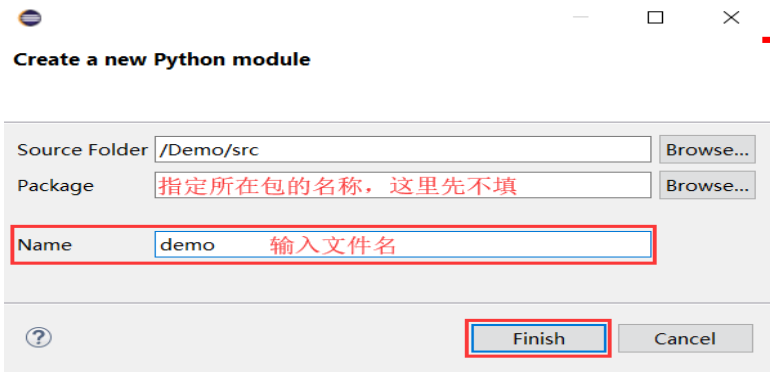


图 6 指定文件名对话框

图中，Package 文本框用来为该文件指定其所在的包，有关包的概念后续章节会进行详细介绍，本节先不涉及。

此时会弹出如图 7 所示的文件模板选择对话框，本节选择 <Empty>（即空模板）即可，然后单击 OK 按钮。

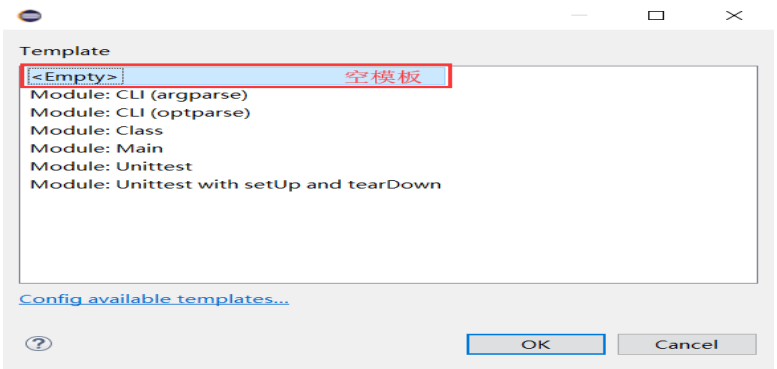


图 7 文件模板选择对话框

如此，即可成功创建一个 Python 源代码文件，如图 8 所示。

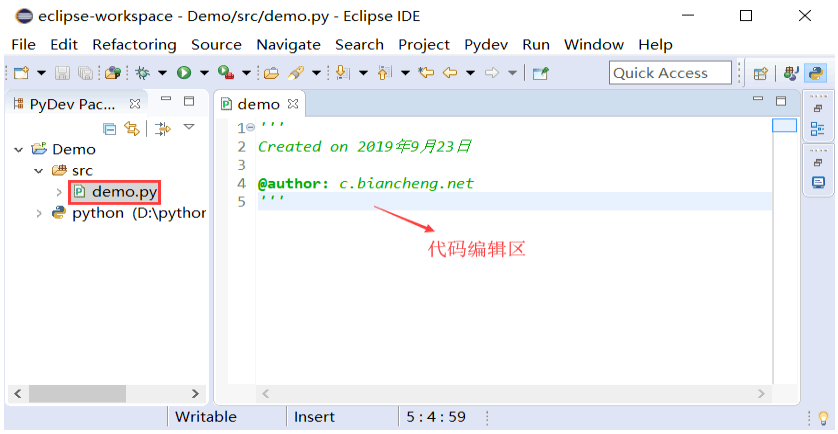


图 8 成功创建 Python 源代码文件

Python 文件创建成功之后，就可以向该文件中编写 Python 程序。例如，在新创建的 demo 文件中编写第一个 Python 程序，如图 9 所示。

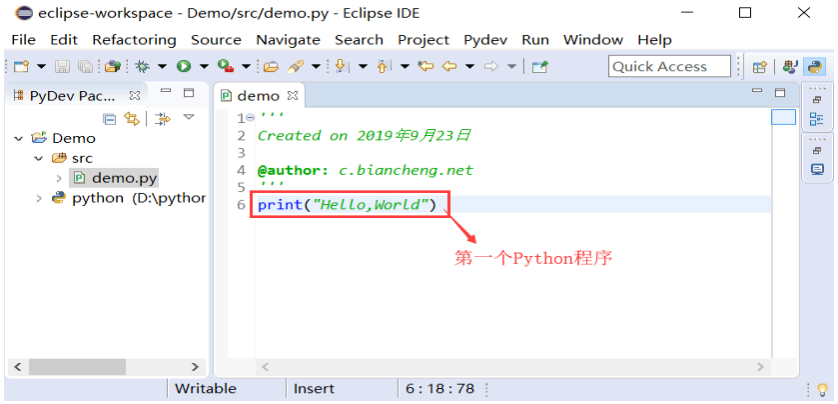


图 9 Eclipse 编写 Python 代码

Eclipse 运行程序的方式也很简单，右键选中 demo 文件，并依次选择“Run As -> Python Run”，如图 10 所示。

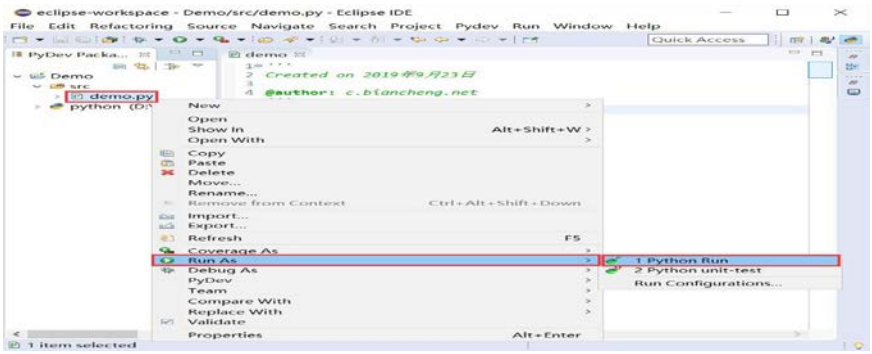


图 10 Eclipse 运行 Python 程序

这样即可成功运行 demo 文件中的程序，运行结果如图 11 所示。

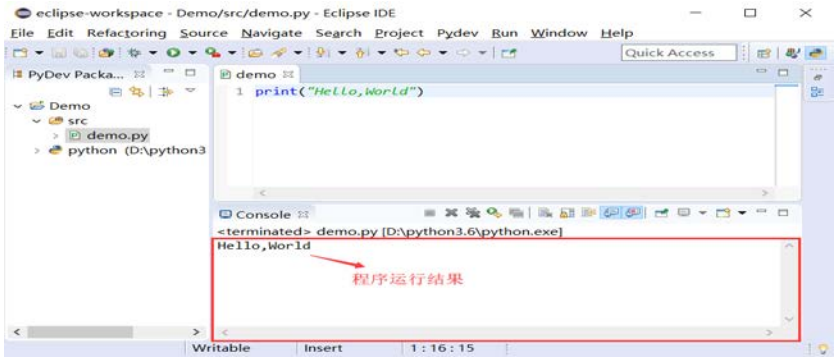


图 11 运行结果示意图

2.16 Python VS Code 下载和安装教程

Visual Studio Code，简称 VS Code，是由微软公司开发的 IDE 工具。与微软其他 IDE（如 Visual Studio）不同的是，Visual Studio Code 是跨平台的，可以安装在 Windows、Linux 和 macOS 平台上运行。

不仅如此，Visual Studio Code 没有限定只能开发特定语言程序，事实上只要安装了合适的扩展插件，它可以开发任何编程语言程序，包括 Python。因此，本节就来讲解如何下载并安装 VS Code，使其能够支持 Python 编程。

VS Code 下载和安装

VS Code 官网提供了 VS Code 下载地址 ，如图 1 所示。

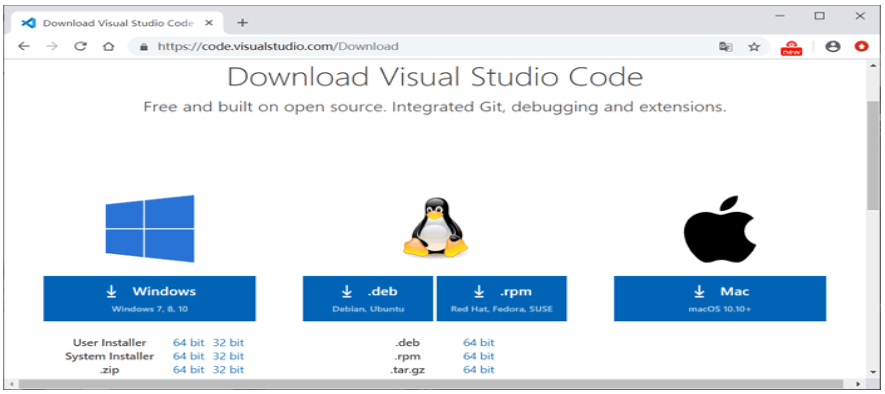


图 1 VS Code 下载界面

可以看到，考虑到不同的操作系统平台，官方准备了分别适用于 Windows、Linux 和 macOS 操作系统的安装包，读者可根据实际情况，选择适合自己电脑的安装包。

值得一提得是，针对 Windows 系统提供的安装包中，还被细分为 User Installer、System Installer 以及 .zip 版，它们之间的区别是：

- User Installer：表示 VS Code 会安装到计算机当前账户目录中，意味着使用其他账号登陆计算机的用户将无法使用 VS Code；
- System Installer：和 User Installer 正好相反，即一人安装，所有账户都可以使用。
- .zip：这是一个 VS Code 的压缩包，下载后只需解压，不需要安装。也就是说，解压此压缩包之后，直接双击包含的 "code.exe" 文件，即可运行 VS Code。

默认情况下，VS Code 提供的是 User Installer 64 位的版本。
由于笔者电脑使用的 Windows 10 系统，所以有 2 种安装 VS Code 的方式，但由于 .zip 版下载即可使用，无需安装，因此本节选择以 User installer 版本为例，给大家演示如何通过安装包安装 VS Code。

注意，.zip 压缩包中包含的 VS Code 和使用安装包安装，没有任何区别。

下载完成安装包之后，大家会得到一个类似名为 VSCodeUserSetup-x64-1.38.1.exe （笔者下载的是此版本）的文件，双击打开，看到如图 2 所示的安装界面。

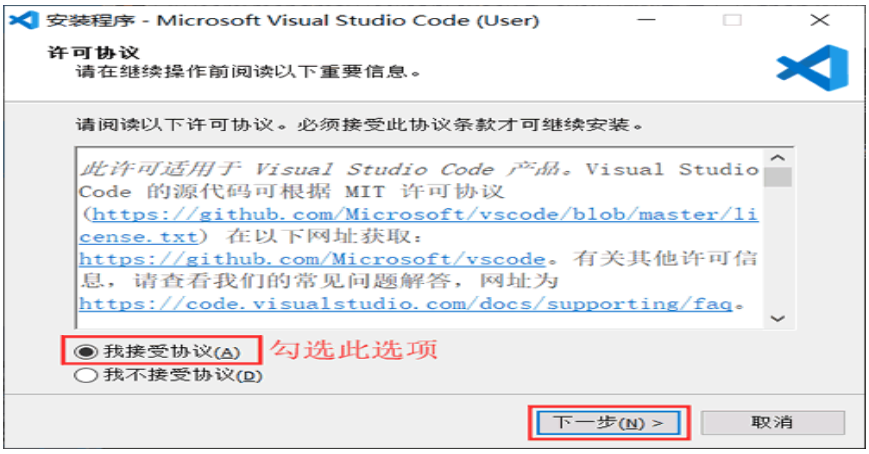


图 2 VS Code 安装界面

勾选“我接受协议”，然后点击“下一步”，进入图 3 所示的界面。

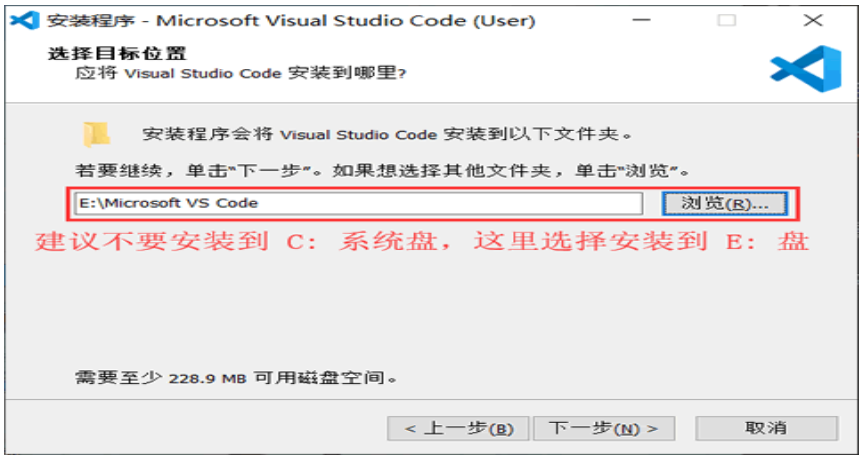


图 3 选择安装位置

如图 3 所示，建议读者不要将 VS Code 安装到系统盘（通常系统盘是 C 盘），可以安装到其它磁盘中。选择好安装位置后，继续点击“下一步”，进入图 4 所示的界面。

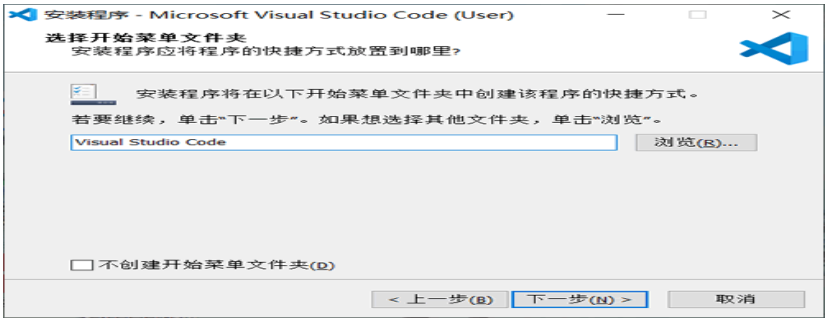


图 4 选择开始菜单文件夹

这里不需要改动，默认即可，直接点击“下一步”，进入图 5 所示的界面。

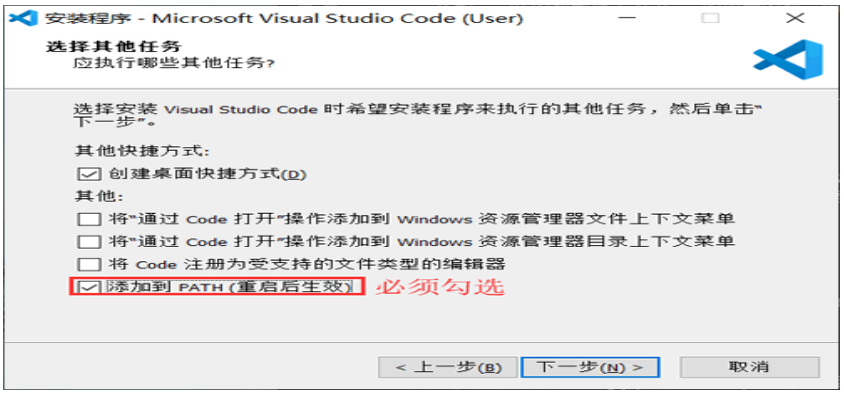


图 5 选择其他任务

读者可根据自己的操作习惯，勾选适合自己的选项即可，需要注意的是，"添加到 PATH"选项一定要勾选。选择完成后，点击“下一步”，进入图 6 所示的界面。

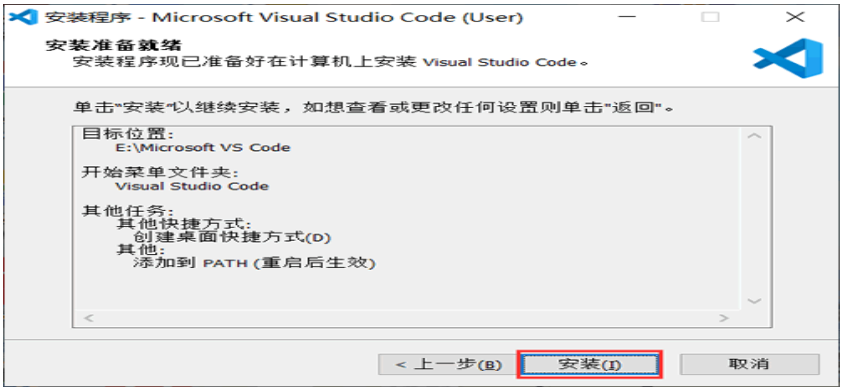


图 6 安装准备就绪

如图 6 所示，显示的是前面选择对 VS Code 做的配置，确认无误后点击“安装”，即可正式安装 VS Code。安装成功后，会出现如图 7 所示的界面，表示安装成功。

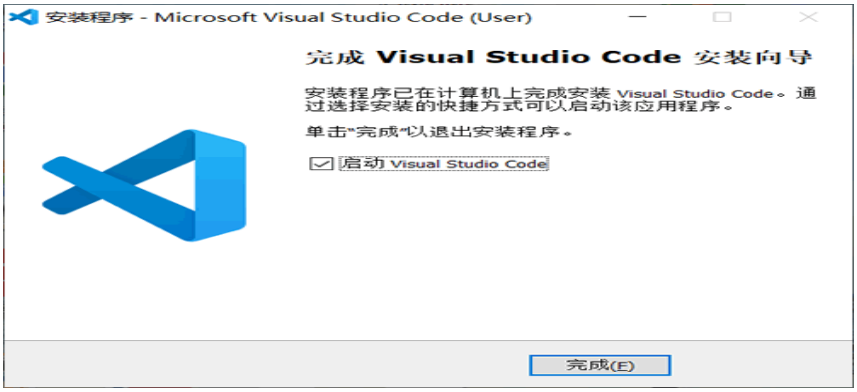


图 7 安装成功界面

点击“完成”，即可启动 VS Code。

VS Code 安装 Python 扩展插件

注意，刚刚安装成功的 VS Code 是没有 Python 扩展的，安装此插件的方法也很简单。打开 VS Code，会进入图 8 所示的欢迎界面。

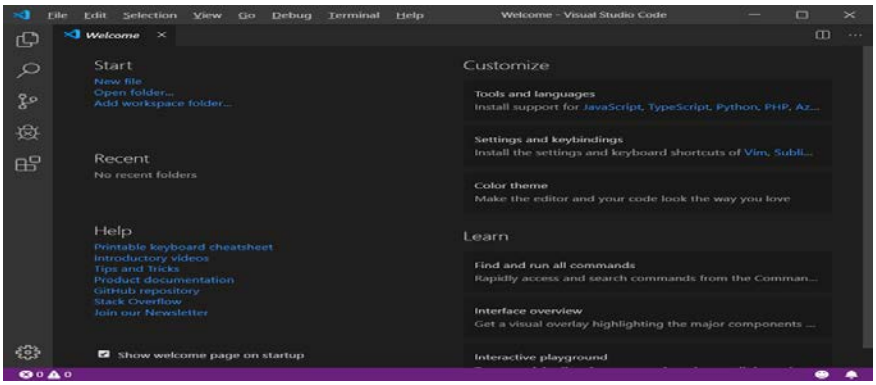


图 8 VS Code 界面

VS Code 安装 Python 扩展的方法有 2 种，分别是：

- 按照图 9 所示，在欢迎界面中选择 "Python"，下方会弹出一个对话框，选择 "OK"，即可完成 Python 扩展的安装；

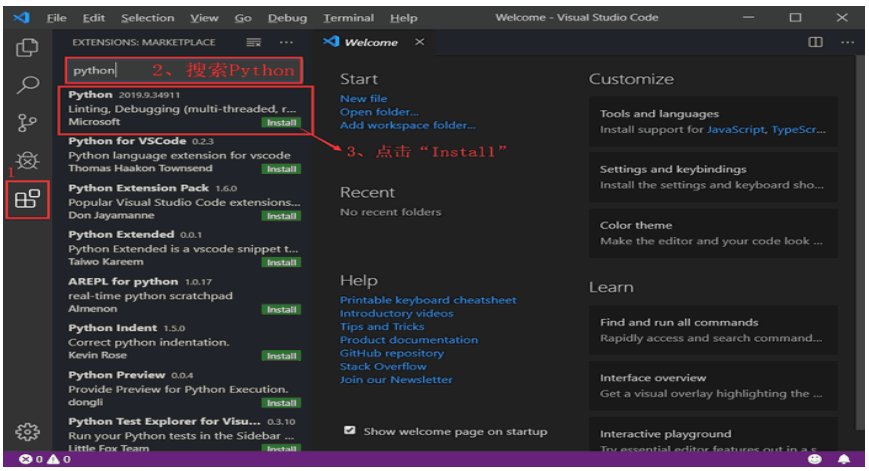


图 9 安装 Python 扩展插件

- 安装图 10 所示，点击“扩展”按钮，并搜索 Python 扩展插件，找到合适的扩展（这里选择的是第一个，这是 Python 的调试工具），选择“Install”即可安装成功。

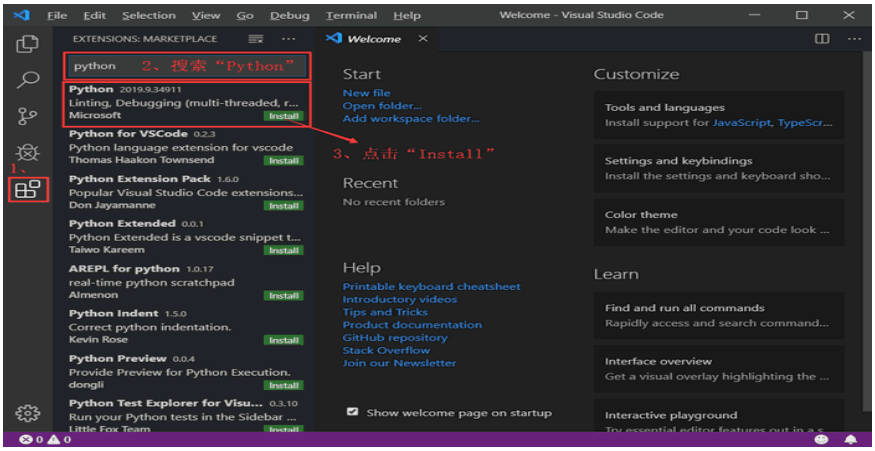


图 10 安装 Python 扩展插件

有关如何使用 VS Code 运行 Python 程序，可猛击《VS Code 运行 Python 程序》一文详细了解。

2.15 VS Code 运行 Python 程序

本节介绍如何使用 VS Code（Visual Studio Code 的简称）编写并运行 Python 程序。值得一提的是，相比 PyCharm、Eclipse+PyDev，使用 VS Code 编写 Python 程序，不用创建项目，直接创建 Python 文件即可。

VS Code 创建 Python 文件

首先，打开 VS Code，会看到如图 1 所示的欢迎界面。

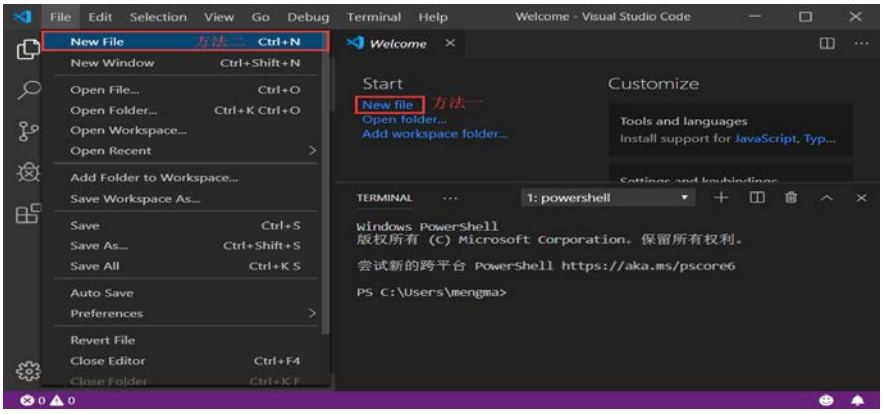


图 1 VS Code 欢迎界面

点击“New File”，或者在菜单栏中依次选择“File -> New File”，如图 2 所示。

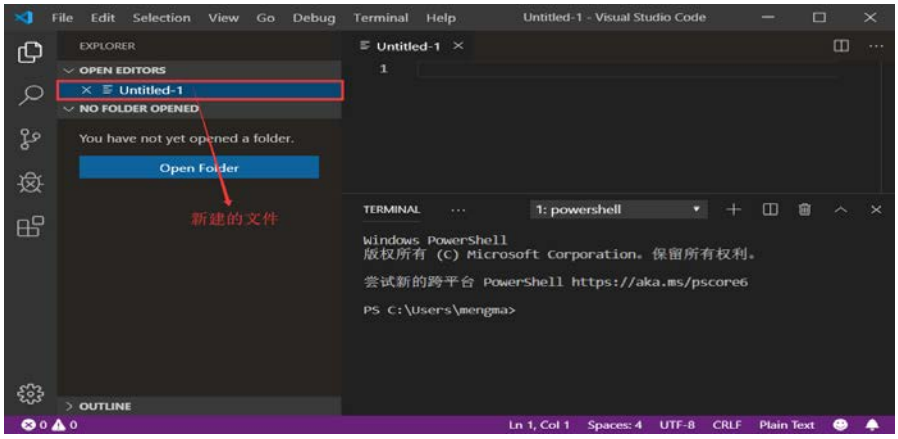


图 2 VS Code 创建文件

此时可以看到，新建了一个名为“Untitled-1”的文件，此文件是 VS Code 默认创建的文件，没有文件类型，所以在编写 Python 代码前，需要手动将其另存为后缀名为 .py 的文件。

另存为的方式也很简单，使用快捷键“Ctrl+S”，或者在菜单栏中依次选择“File -> Save”，都会弹出如图 3 所示的窗口。

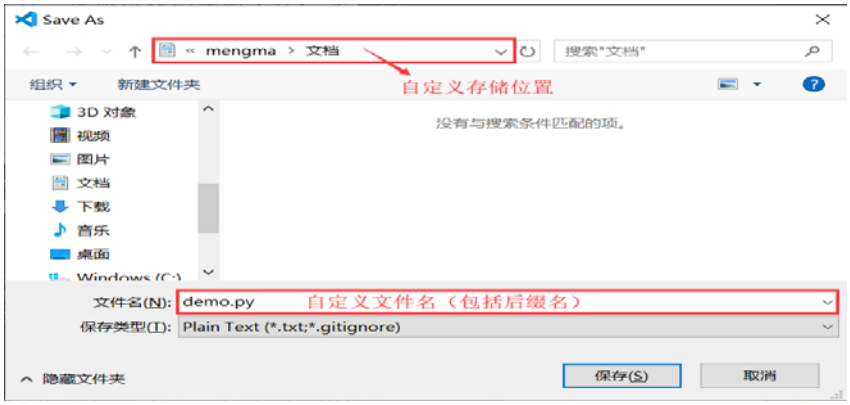


图 3 另存为 .py 文件

在此窗口中，我们可以设置该新建文件的文件名（包括后缀名），还可以自定义该文件的存储位置。

只有重新将新建文件保存为以 .py 为后缀名的文件，VS Code 才能够识别出来是 Python 文件，后期在此文件中编写 Python 代码时，才能高亮显示。

由此，我们就成功的创建了一个名为“demo.py”的 python 源文件，如图 4 所示。

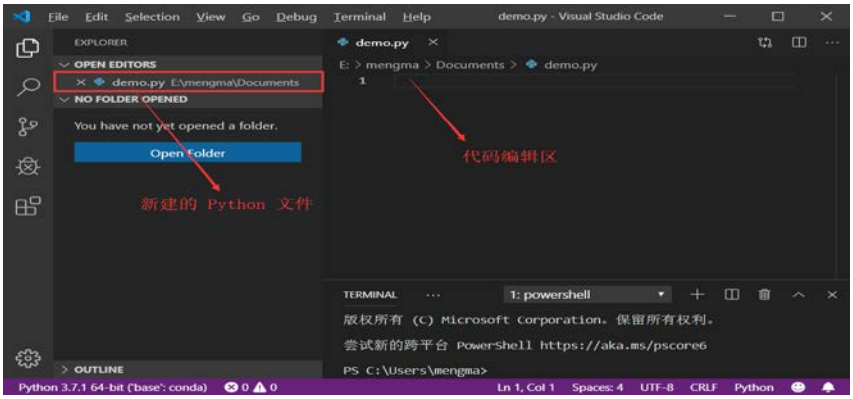


图 4 新建的 Python 文件

创建 Python 文件成功后，就可以在该文件中编写 Python 代码了，这里仍以第一个 Python 程序为例，即向“demo.py”文件中编写如下代码：

```
1.  print("Hello,World")
```

代码编写完成后，就可以运行了，具体步骤是，使用组合键“Ctrl+F5”，或者在菜单栏中选择“Debug -> Start Without Debugging”，即可看到如图 5 所示的输出结果。

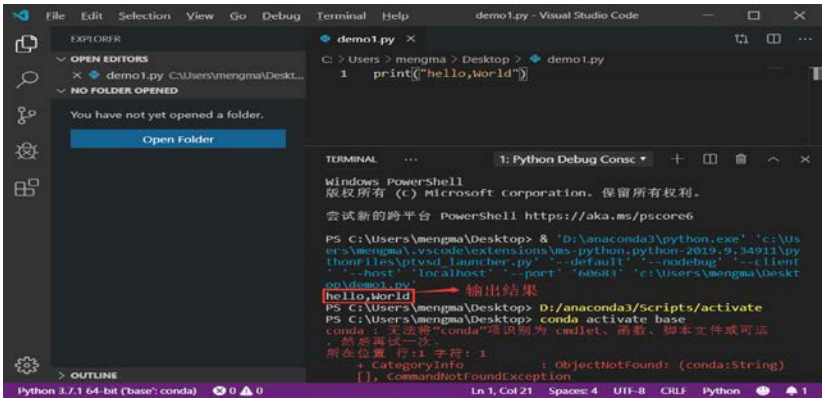


图 5 运行结果

图 5 显示的信息中，除了运行结果，还有代码执行过程中产生的信息。有读者可能会想，能否只显示运行结果呢？办法是有的，只需进行如下操作。

1) 手动将我们创建的 Python 文件放到一个文件夹中，然后将此文件夹引入到 VS Code。例如，这里将前面创建的 demo.py 文件放到了一个 Demo 文件夹（新建的），将此文件夹引入到 VS Code 的方法是，在菜单栏中依次选择“File -> Open Folder”（如图 6 所示），然后找到 Demo 文件夹，点击“选择文件夹”，就可以成功将指定文件夹引入到 VS Code 中。

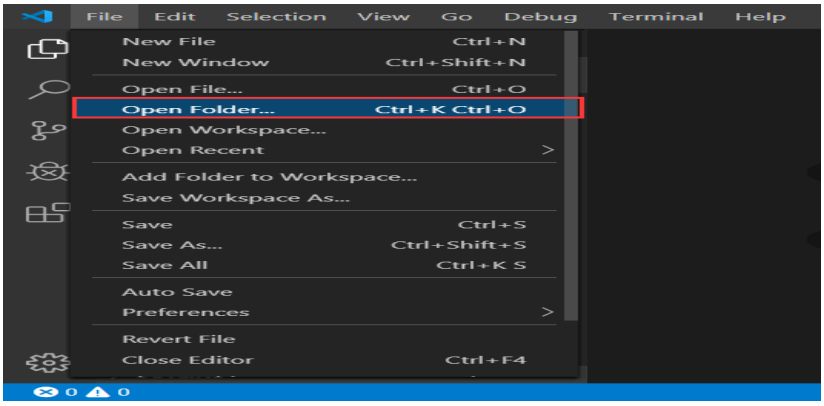


图 6 向 VS Code 引入文件夹

2) 引入成功，资源管理器（EXPLORER）中如图 7 所示。

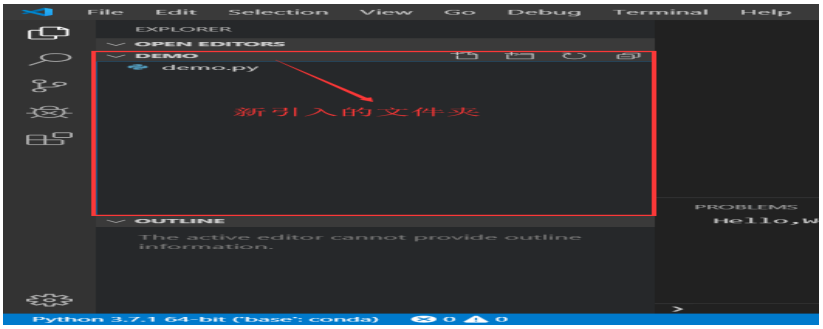


图 7 VS Code 资源管理器

3) 在图 7 的基础上，先点击“demo.py”文件（这一步很重要），使 VS Code 右侧显示该文件，然后点击左侧的 Debug 按钮（小甲壳虫图标），再点击“设置”按钮，如图 8 所示。

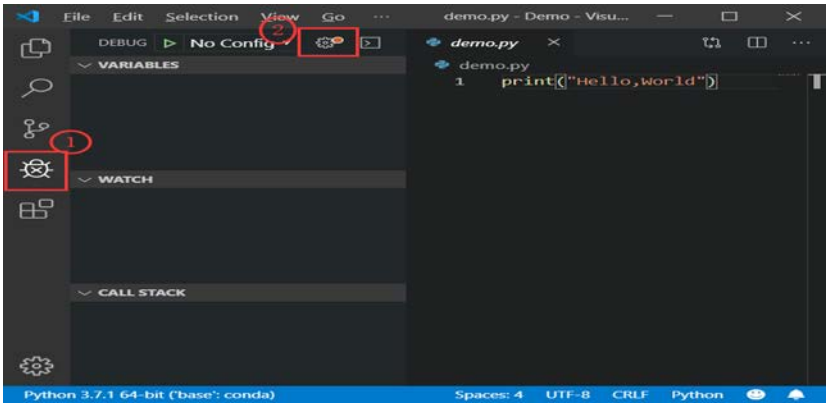


图 8 跳转到 Debug 界面

注意，VS Code 的右侧一定要显示有建立的 Python 源文件，才能执行此步，否则会出错。

4) 此时会弹出一个对话框，选择“Python File”，如图 9 所示。

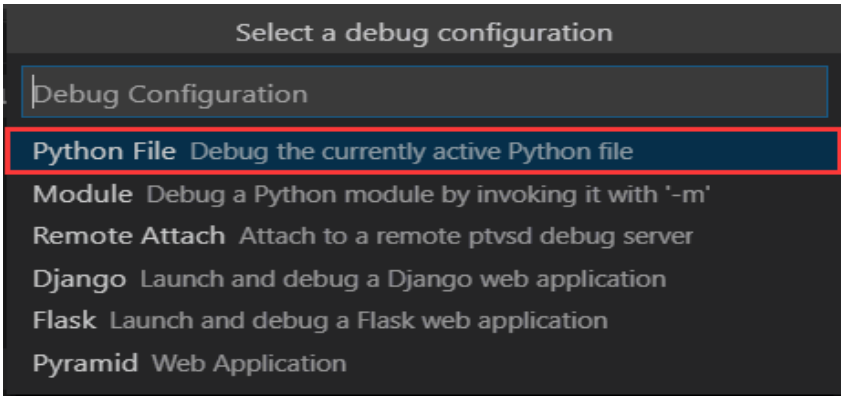


图 9 编译配置

5) 可以看到，显示出了一个名为“launch.json”的文件，我们需要做的就是将它所包含代码中的 "console":"integratedTerminal" 改为 "console": "none"，如图 10 所示。

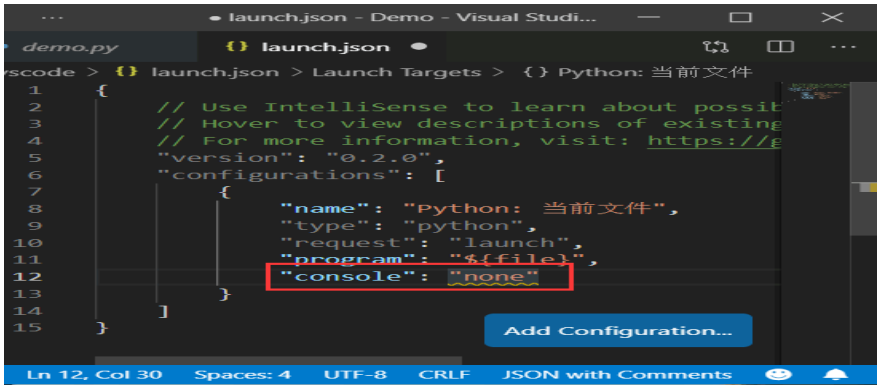


图 10 修改 launch.json 文件

6) 然后再次运行 demo.py，可以看出如图 11 所示的运行结果。

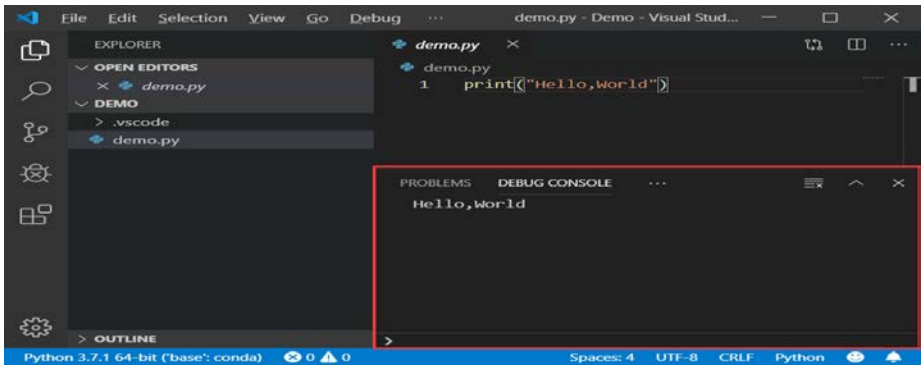


图 11 只显示程序运行结果

2.16 Python Visual Studio 下载和安装教程

Visual Studio（简称 VS）是微软推出的一款功能强大的开发工具，它支持 C#、C++、Python、Visual Basic、Node.js、HTML、JavaScript 等各大编程语言，还能开发 iOS、Android 的移动平台应用，VS 2017 甚至还自带了 iOS 模拟器（之前为 MAC 独享的开发环境）。

到目前为止，VS 最新版本为 2019 版，不过本节并不以最新版进行演示，而是以 VS 2017 版为例给大家讲解。不过，VS 所有版本的下载和安装过程都大同小异，因此打算安装其他版本的读者，也可以依照本节内容进行操作。

VS 2017 版本可细分为三个版本，分别是：

- 社区版（Community）：免费提供给单个开发人员，给予初学者及大部分程序员支持，可以无任何经济负担、合法地使用。
- 企业版：为正规企业量身定做，能够提供点对点的解决方案，充分满足企业的需求。企业版官方售价 2999 美元/年 或者 250 美元/月。
- 专业版：适用于专业用户或者小团体。虽没有企业版全面的功能，但相比于免费的社区版，有更强大的功能。专业版官方售价 539 美元/年 或者 45 美元/月。

对于大部分程序开发，以上版本区别不大，免费的社区版一样可以满足程序员需求，所以我推荐大家使用社区版，无需破解，轻松安装，快速使用。

下载 VS 2017

VS 2017 社区版（Community）下载地址：

- 迅雷下载：

  ed2k://|file|en_visual_studio_community_2015_x86_dvd_6847364.iso|3965825024|6A7D8489BB2877E6BB8ACB2DD187B637|/

- 百度网盘：

  链接: https://pan.baidu.com/s/1jJXyRMA 密码: ub6c

下载的文件，其大小不足 1 MB，只是 VS 2017 Community 简体中文版的一个安装引导程序。启动后勾选需要的组建即可进行在线下载安装。

.NET Framework 安装

下载 VS 2017 的安装引导程序后，双击运行，如果出现下面的 Visual Studio 提示（如不出现此提示，可直接省略此环节）：

则在安装 VS 2017 之前，需要首先自行安装版本较高的 .Net Framework 。建议直接下载. Net Framework 4.6 安装包进行安装，速度较快。

.Net Framework 4.6 版的下载地址:

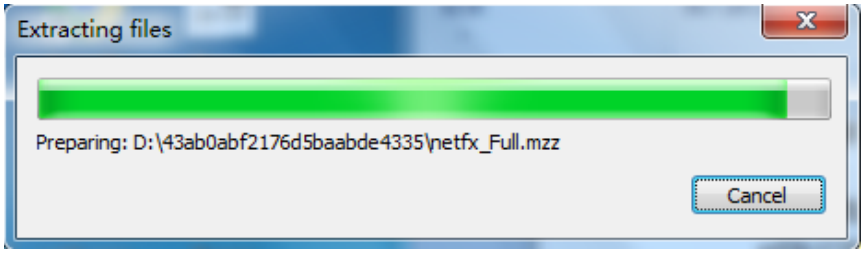
- 迅雷下载(较快):

  ed2k://|file|mu_.net_fx_4_6_2_for_win_7sp1_8dot1_10_win_server_2008sp2_2008r2sp1_2012_2012r2_x86_x64_9058211.exe|62008080|D36FDF083FF2970FD8B0080664AD32C6|/

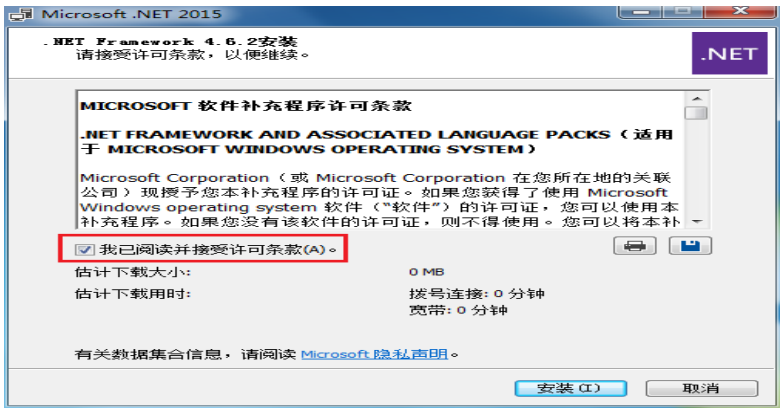
- 百度网盘下载（较慢）：

  链接: https://pan.baidu.com/s/1mj2mGgo 密码: bhf7

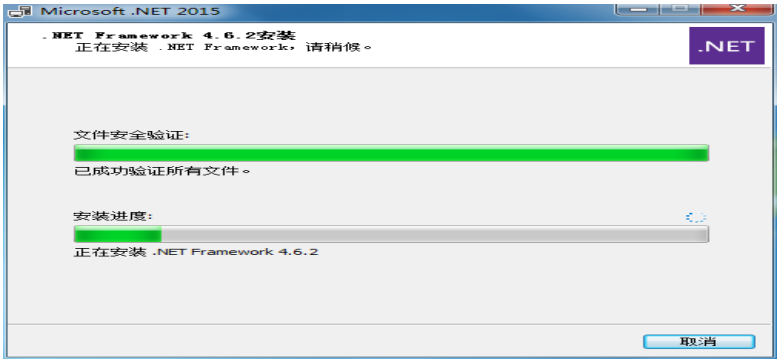
  下载完成后，会得到一个 .net framework 的安装包，双击打开，会出现下面的一个安装进度条：



进度条达到 100% 后，会自动跳到下面的页面：



勾选 “我已阅读并接受许可条款” ，然后点击 “安装” 按钮，进行安装：



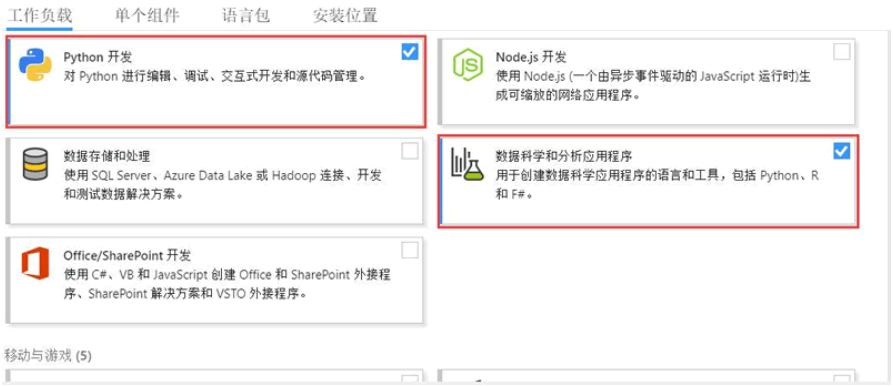
待 “文件安全验证" 进度条和 “安装进度" 进度条全部达到 100%，程序会提示你安装完成，点击关闭即可（此时可能需要重启计算机，没关系，重启即可）。

安装 VS 2017

VS 2017 下载完成后，会得到一个用于引导用户安装的可执行文件，双击该文件，在 .Net Framework 版本没有问题的前提下，会进入安装页面：

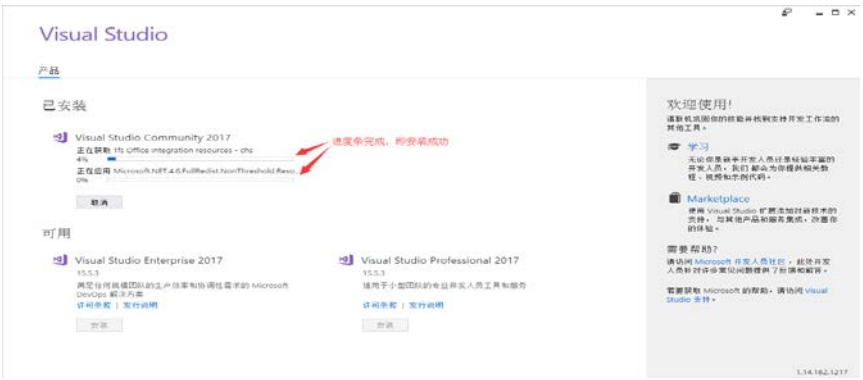
直接点击 “继续” 按钮，此时会弹出一个进度条：

等 Visual Studio 准备完成后，会直接跳到下面的页面：



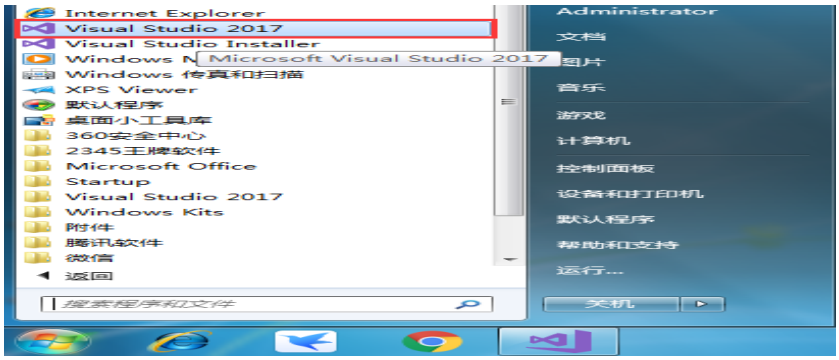
VS 2017 除了支持 Python 开发，还支持 C、C++、C#、F#、Visual Basic 等开发语言，不过我们没有必要安装所有的组件，只需要安装上图所示的 2 个模块即可。

同时在这个页面，还可以选择 VS 2017 的存储位置，建议不要安装在 C 盘，可选择其他盘。然后直接点击安装，安装过程可能需要一段时间，大家耐心等待。



安装完成后，VS 2017 会要求重启计算机，该保存的保存，按要求重启即可。

重启完成后，打开 “开始菜单” ，会发现多了一个叫 “Visual Studio 2017” 的图标，证明你安装成功啦。



注意，安装成功后，首次使用 VS 2017 还需要对其进行简单的配置（例如软件本身的主题风格），读者可根据自己的喜好进行选择，因为非常简单，这里不再给出配置过程的具体图示。

关于如何使用 VS 运行 Python 程序，可猛击《Visual Studio 运行 Python 程序》一文详细了解。

2.17 Visual Studio 运行 Python 程序（超级详细）

本节仍以第一个 Python 程序为例，继续讲解如何通过 VS（Visual Studio 的简写）实现编写和运行 Python 程序。

VS 创建 Python 项目

VS 和 PyCharm、Eclipse 一样，也是通过项目来管理 Python 源代码程序文件的。VS 创建 Python 项目的过程如下。

1) 首先打开 VS，在菜单栏中依次选择“文件 -> 新建 -> 项目”（如图 1 所示），打开新建项目对话框。

图 1 VS 打开新建项目窗口示意图

2) 新建项目对话框如图 2 所示，依次选择“Python -> Python 应用程序”，并为新建项目起名（比如为 Demo），其他选择默认即可。

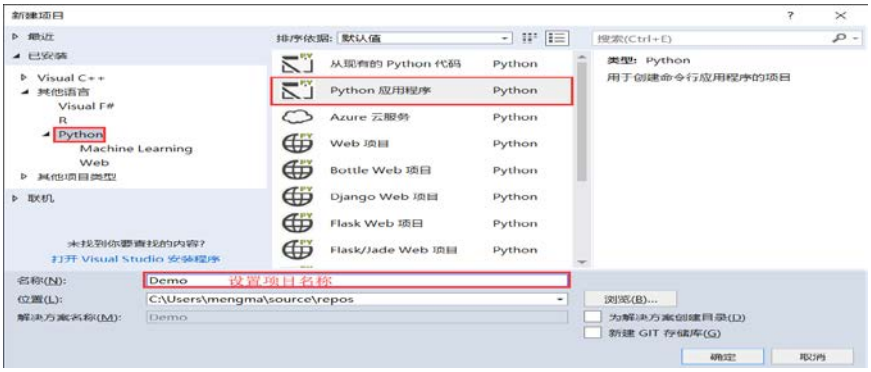


图 2 设置项目名称

直接使用快捷键“Ctrl+Shift+N”，也可以打开此窗口。

3) 点击“确定”之后，会回到 VS 主界面，此时可以看到，VS 资源管理器中已经出现了 Demo 项目，如图 3 所示。

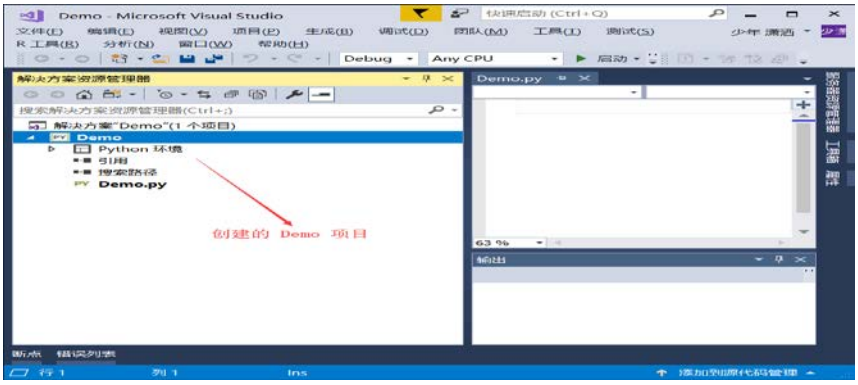


图 3 创建完成的 Python 项目

不仅如此，项目中还自动创建了一个和 Demo 项目同名的 Demo 源代码文件，我们可以在此文件中直接编写 Python 代码。但这里还是给大家介绍一下，VS 创建好项目之后，如何向项目中手动添加源代码文件。

以上面创建的 Demo 项目为例，下面手动向项目中创建一个名为 test 的源代码文件。

1) 首先，右键资源管理器中的 Demo 项目，并依次选择“添加 -> 新建项”，可以打开“添加文件”对话框。如图 4 所示。

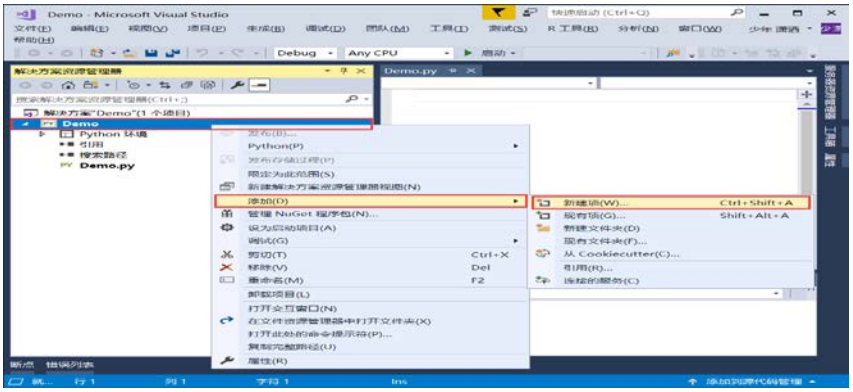


图 4 VS 添加文件具体操作

2) 打开的“添加文件”对话框，如图 5 所示，这里可以选择想要创建的 Python 文件。由于本节以简单的“Hello，World”程序为例，因此选择创建空 Python 文件即可。最后，还要记得给要添加的文件命名（比如“test”）。

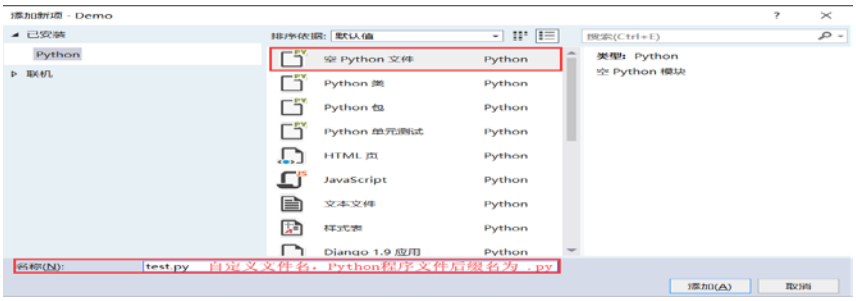


图 5 添加文件

直接使用快捷键“Ctrl+Shift+A”，也可以打开此窗口。

3) 点击“添加”按钮后，VS 会回到主界面，此时再次观察 Demo 项目，发现其多了一个 test.py 文件，这就是我们手动添加的文件，如图 6 所示。

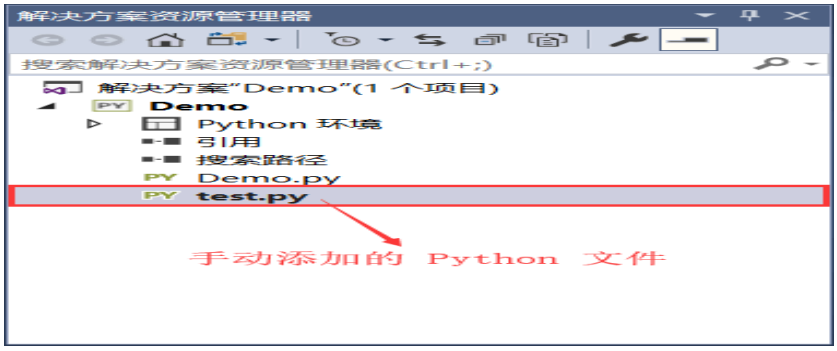


图 6 成功手动添加 Python 文件

由此，我们就可以在现有的 Demo.py 或者 test.py 文件中编写第一个 Python 程序。例如，我们在 Demo.py 文件中编写如下代码：

```
1.  print("Hello, Demo")
```

在 test.py 文件中编写如下代码：

```
1.  print("hello, test")
```

编写完成之后，VS 运行程序也很简单，只需要点击“启动”按钮，或者按“Ctrl+F5”，即可运行程序。点击之后，可以看到运行结果如图 7 所示。

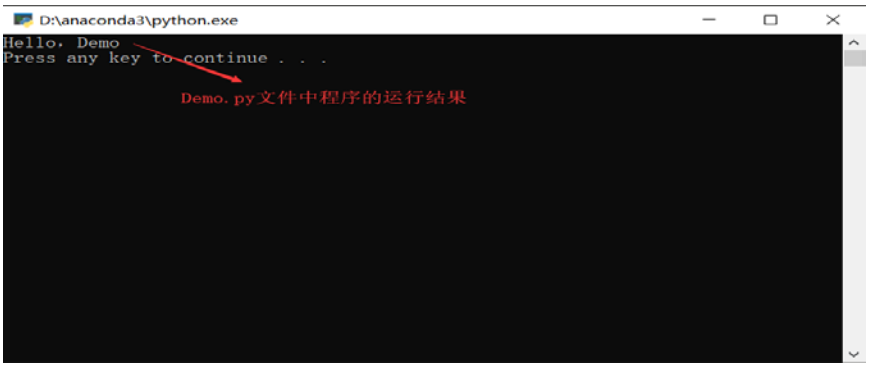


图 7 Demo.py 文件中代码的运行结果

有读者可能会问，为什么 VS 不运行 test.py 文件中的代码呢？这是因为，在默认情况下，VS 会将和项目同名的 Python 文件（本例中为 Demo 文件）设为启动文件，运行时也只会运行和启动文件相关的程序代码。

所以，如果想运行 test.py 文件中的程序，需要提前将 test.py 文件设置为启动文件，设置方式也很简单，右键点击 test.py 并选择“设置为启动文件”。此时，再次点击“启动”按钮，其运行结果如图 8 所示。

图 8 test.py 文件中代码的运行结果

2.18 Python 注释（多行注释和单行注释）用法详解

注释（Comments）用来向用户提示或解释某些代码的作用和功能，它可以出现在代码中的任何位置。Python 解释器在执行代码时会忽略注释，不做任何处理，就好像它不存在一样。

在调试（Debug）程序的过程中，注释还可以用来临时移除无用的代码。

注释的最大作用是提高程序的可读性，没有注释的程序简直就是天书，让人吐血！

千万不要认为你自己写的代码规范就可以不加注释，甩给别人一段没有注释的代码是对别人的不尊重，是非常自私的行为；你可以喜欢自虐，但请不要虐待别人。

很多程序员宁愿自己去开发一个应用，也不愿意去修改别人的代码，没有合理的注释是一个重要的原因。虽然良好的代码可以自成文挡，但我们永远不清楚今后阅读这段代码的人是谁，他是否和你有相同的思路；或者一段时间以后，你自己也不清楚当时写这段代码的目的了。

一般情况下，合理的代码注释应该占源代码的 1/3 左右。

Python 支持两种类型的注释，分别是单行注释和多行注释。

Python 单行注释

Python 使用井号#作为单行注释的符号，语法格式为：

```
# 注释内容
```

从井号#开始，直到这行结束为止的所有内容都是注释。Python 解释器遇到#时，会忽略它后面的整行内容。

说明多行代码的功能时一般将注释放在代码的上一行，例如：

```
#使用 print 输出字符串
print("Hello World!")
print("C 语言中文网")
print("http://c.biancheng.net/python/")

#使用 print 输出数字
print(100)
print( 3 + 100 * 2)
print( (3 + 100) * 2 )
```

说明单行代码的功能时一般将注释放在代码的右侧，例如：

```
print("http://c.biancheng.net/python/")  #输出 Python 教程的地址
print( 36.7 * 14.5 )  #输出乘积
print( 100 % 7 )  #输出余数
```

Python 多行注释

多行注释指的是一次性注释程序中多行的内容（包含一行）。

Python 使用三个连续的单引号'''或者三个连续的双引号"""注释多行内容，具体格式如下：

```
'''
使用 3 个单引号分别作为注释的开头和结尾
可以一次性注释多行内容
这里面的内容全部是注释内容
'''
```

或者

```
"""
使用 3 个双引号分别作为注释的开头和结尾
可以一次性注释多行内容
这里面的内容全部是注释内容
"""
```

多行注释通常用来为 Python 文件、模块、类或者函数等添加版权或者功能描述信息。

注意事项

1) Python 多行注释不支持嵌套，所以下面的写法是错误的：

```
'''
外层注释
    '''
    内层注释
    '''
'''
```

2) 不管是多行注释还是单行注释，当注释符作为字符串的一部分出现时，就不能再将它们视为注释标记，而应该看做正常代码的一部分，例如：

```
print('''Hello,World!''')
print("""http://c.biancheng.net/cplus/""")
print("#是单行注释的开始")
```

运行结果：

```
Hello,World!
http://c.biancheng.net/cplus/
#是单行注释的开始
```

对于前两行代码，Python 没有将这里的三个引号看作是多行注释，而是将它们看作字符串的开始和结束标志。

对于第 3 行代码，Python 也没有将井号看作单行注释，而是将它看作字符串的一部分。

注释可以帮助调试程序

给代码添加说明是注释的基本作用，除此以外它还有另外一个实用的功能，就是用来调试程序。

举个例子，如果你觉得某段代码可能有问题，可以先把这段代码注释起来，让 Python 解释器忽略这段代码，然后再运行。如果程序可以正常执行，则可以说明错误就是由这段代码引起的；反之，如果依然出现相同的错误，则可以说明错误不是由这段代码引起的。

在调试程序的过程中使用注释可以缩小错误所在的范围，提高调试程序的效率。

2.19Python 缩进规则（包含快捷键）

和其它程序设计语言（如 Java、C 语言）采用大括号“{}”分隔代码块不同，Python 采用代码缩进和冒号（：）来区分代码块之间的层次。

在 Python 中，对于类定义、函数定义、流程控制语句、异常处理语句等，行尾的冒号和下一行的缩进，表示下一个代码块的开始，而缩进的结束则表示此代码块的结束。

注意，Python 中实现对代码的缩进，可以使用空格或者 Tab 键实现。但无论是手动敲空格，还是使用 Tab 键，通常情况下都是采用 4 个空格长度作为一个缩进量（默认情况下，一个 Tab 键就表示 4 个空格）。

例如，下面这段 Python 代码中（涉及到了目前尚未学到的知识，初学者无需理解代码含义，只需体会代码块的缩进规则即可）：

```
height=float(input("输入身高："))  #输入身高
weight=float(input("输入体重："))  #输入体重
bmi=weight/(height*height)        #计算 BMI 指数
```

```
4.
5.  #判断身材是否合理
6.  if bmi<18.5:
7.      #下面 2 行同属于 if 分支语句中包含的代码，因此属于同一作用域
8.      print("BMI 指数为："+str(bmi)) #输出 BMI 指数
9.      print("体重过轻")
10. if bmi>=18.5 and bmi<24.9:
11.     print("BMI 指数为："+str(bmi)) #输出 BMI 指数
12.     print("正常范围，注意保持")
13. if bmi>=24.9 and bmi<29.9:
14.     print("BMI 指数为："+str(bmi)) #输出 BMI 指数
15.     print("体重过重")
16. if bmi>=29.9:
17.     print(BMI 指数为："+str(bmi)) #输出 BMI 指数
18.     print("肥胖")
```

Python 对代码的缩进要求非常严格，同一个级别代码块的缩进量必须一样，否则解释器会报 SyntaxError 异常错误。例如，对上面代码做错误改动，将位于同一作用域中的 2 行代码，它们的缩进量分别设置为 4 个空格和 3 个空格，如下所示：

```
1.  if bmi<18.5:
2.      print("BMI 指数为："+str(bmi)) #输出 BMI 指数
3.     print("体重过轻")
```

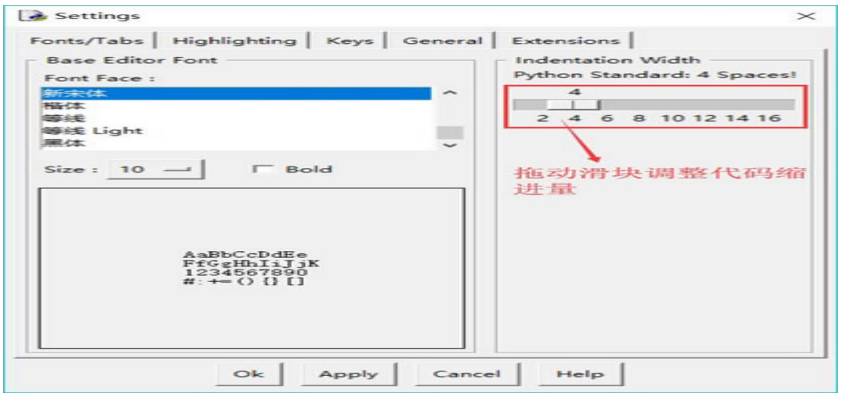
可以看到，第二行代码和第三航代码本来属于同一作用域，但我们手动修改了各自的缩进量，这会导致 SyntaxError 异常错误，如图 1 所示。

图 1 缩进不符合规范导致异常

对于 Python 缩进规则，初学者可以这样理解，Python 要求属于同一作用域中的各行代码，它们的缩进量必须一致，但具体缩进量为多少，并不做硬性规定。

IDLE 开发环境对缩进量的设置

在 IDLE 开发环境中，默认是以 4 个空格作为代码的基本缩进单位。不过，这个值是可以手动改变的，在菜单栏中选择 `Options -> Configure`，会弹出如下对话框：

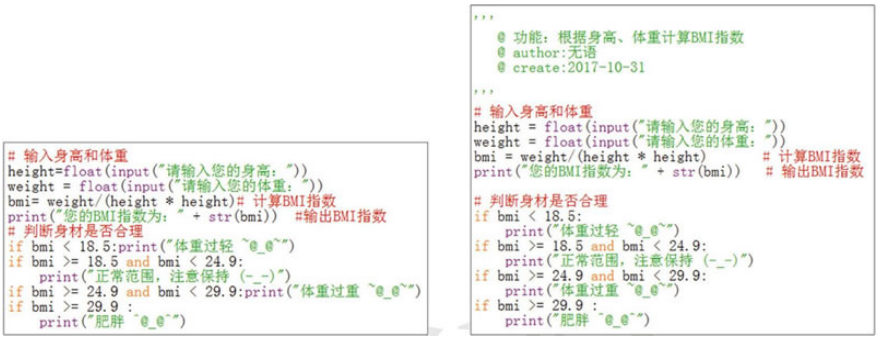


如图所示，通过拖动滑块，即可改变默认的代码缩进量，例如拖动至 2，则当你使用 Tab 键设置代码缩进量时，会发现按一次 Tab 键，代码缩进 2 个空格的长度。

不仅如此，在使用 IDLE 开发环境编写 Python 代码时，如果想通过设置多行代码的缩进量，可以使用 `Ctrl+]` 和 `Ctrl+[` 快捷键，此快捷键可以使所选中代码快速缩进（或反缩进）。

2.20 Python 编码规范（PEP 8）

在讲解具体的 Python 编码规范之前，先来看看图 1 中的代码：



```
# 输入身高和体重
height=float(input("请输入您的身高："))
weight = float(input("请输入您的体重："))
bmi= weight/(height * height)# 计算BMI指数
print("您的BMI指数为：" + str(bmi))  #输出BMI指数
# 判断身材是否合理
if bmi < 18.5:print("体重过轻 ^@_@^")
if bmi >= 18.5 and bmi < 24.9:
    print("正常范围，注意保持 (-_-)")
if bmi >= 24.9 and bmi < 29.9:print("体重过重 ^@_@^")
if bmi >= 29.9 :
    print("肥胖 ^@_@^")
```

```
'''
   @ 功能：根据身高、体重计算BMI指数
   @ author:无语
   @ create:2017-10-31
'''
# 输入身高和体重
height = float(input("请输入您的身高："))
weight = float(input("请输入您的体重："))
bmi = weight/(height * height)        # 计算BMI指数
print("您的BMI指数为：" + str(bmi))    # 输出BMI指数

# 判断身材是否合理
if bmi < 18.5:
    print("体重过轻 ^@_@^")
if bmi >= 18.5 and bmi < 24.9:
    print("正常范围，注意保持 (-_-)")
if bmi >= 24.9 and bmi < 29.9:
    print("体重过重 ^@_@^")
if bmi >= 29.9 :
    print("肥胖 ^@_@^")
```

图 1 两段功能相同的 Python 代码

对比图 1 中的两段代码你会发现，它们所包含的代码时完全相同的，但很明显，右侧的代码编写格式看上去比左侧的代码段更加规整，阅读起来也会比较轻松、畅快，因为它遵循了最基本的 Python 代码编写规范。

Python 采用 PEP 8 作为编码规范，其中 PEP 是 Python Enhancement Proposal（Python 增强建议书）的缩写，8 代表的是 Python 代码的样式指南。下面仅给大家列出 PEP 8 中初学者应严格遵守的一些编码规则：

1. 每个 import 语句只导入一个模块，尽量避免一次导入多个模块，例如：

```
#推荐
import os
import sys
#不推荐
import os,sys
```

关于 import 的含义和用法会在后续介绍，这里不必深究。

2. 不要在行尾添加分号，也不要用分号将两条命令放在同一行，例如：

```
#不推荐
height=float(input("输入身高："))  ;  weight=fioat(input("输入体重："))  ;
```

3. 建议每行不超过 80 个字符，如果超过，建议使用小括号将多行内容隐式的连接起来，而不推荐使用反斜杠 \ 进行连接。例如，如果一个字符串文本无法实现一行完全显示，则可以使用小括号将其分开显示，代码如下：

```
#推荐
s=("C 语言中文网是中国领先的 C 语言程序设计专业网站，"
"提供 C 语言入门经典教程、C 语言编译器、C 语言函数手册等。")
#不推荐
s="C 语言中文网是中国领先的 C 语言程序设计专业网站，\
提供 C 语言入门经典教程、C 语言编译器、C 语言函数手册等。"
```

注意，此编程规范适用于绝对大多数情况，但以下 2 种情况除外：

- 导入模块的语句过长。
- 注释里的 URL。

使用必要的空行可以增加代码的可读性，通常在顶级定义（如函数或类的定义）之间空两行，而方法定义之间空一行，另外在用于分隔某些功能的位置也可以空一行。比如说，在图 1 右侧这段代码中，if 判断语句同之前的代码多实现的功能不同，因此这里可以使用空行进行分隔。

通常情况下，在运算符两侧、函数参数之间以及逗号两侧，都建议使用空格进行分隔。

以上就是初学者应该遵循的部分 Python 编码规范，如果想了解更多 PEP 8 的详细信息，可访问 PEP 8 官方介绍。

2.21 Python 编码规范的重要性

很多去 Google 参观的人，在用完洗手间后都有这样的疑惑，马桶前面的门上怎么会贴着 Python 编码规范？要知道，Google 对编码规范的要求极其严格，这也能从侧面说明编码规范的重要性。

对于编码规范的认知，很多初学者还仅停留在初级阶段，即只知道编码规范有用，比如命名时使用驼峰式的格式（如 TheFirstDemo），而至于为什么要求这样严格，就不是很清楚了。

本节，将给读者扫除以下 2 个盲区：

1. Python 编码规范到底有多么重要，它对于业务开发来说，究竟有哪些帮助？
2. 有哪些流程和工具，可以强制你遵循规定好的编码规范呢？

注意，在讲解过程，会引用以下 2 个编码规范来举例，分别是：

- 《8 号 Python 增强规范》，通常称之为 PEP8；
- 《Google Python 风格规范》简称为 Google Style，这是源自 Google 内部公开发布的社区版本，其目的是为了让 Google 旗下所有 Python 开源项目的编程风格统一。

以上这 2 个编码规范，Google Style 比 PEP8 更为严格，因为 PEP8 的主要面向群体是个人和小团队开发者，而 Google Style 则能够胜任大团队甚至是企业。

Python 编码规范到底有多么重要

Python 编码规范重要性的原因用一句话来概括就是：统一的编码规范可以提高开发效率。

而影响开发效率的有 3 类对象，分别是阅读者、编程者和机器，它们的优先级是阅读者>>编程者>>机器（>>表示远远大于）。

阅读者>>编程者

写过代码的人应该深有体会，在实际工作中真正用来码代码的时间，远比阅读或者调试的时间要少。事实也是如此，有研究表明，软件工程中 80% 的时间都在阅读代码。

因此，如果想提高开发效率，首先要优化的不是码代码的速度，而是阅读代码的体验。

其实，很多编码规范本身就是为优化读者体验而存在的，拿命名原则来说，PEP8 第 38 条规定命名不能是无意义的单字母，有意义的名称可以很大程序提高阅读者的体验。

编程者>>机器

说完了阅读者的体验，再来聊聊编程者的体验。笔者常常见到的一个错误倾向就是过度简化自己的代码，这样做会大大降低代码的可阅读性，并且一旦出现 BUG，也不容易检查出来。

例如，阅读如下这行程序：

```
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]
```

上面这行代码还可以改写成如下这种形式：

```
result = []
for x in range(10):
  for y in range(5):
    if x * y > 10:
      result.append((x, y))
```

以上代码，涉及到了列表和判断循环结构的相关知识，由于还未学到，初学者不需要理解。
对比这 2 种写法，显然后者调理更清楚，更容易理解，编写起来也更轻松。

机器体验也很重要

每个人都希望自己编写的代码能正确、高效地在电脑上执行，但是一些危险的编程风格，不仅会影响程序的正确性，也容易成为代码效率的瓶颈。

例如，PEP8 和 Google Style 都特别强调了，何时使用 is， 何时使用 ==，何时使用隐式布尔转换。不仅如此，Google Style 2.8 还对遍历方式的选择作出了明确限制。

在编程过程中，只要严格遵守编码规范，编写出的代码通常都很健壮，可移植性也很高。

编码规范的自动化工具

既然编码规范的终极目标是提高开发效率。所以，如果每次写代码都需要在代码规范上额外花很多时间，就达不到我们的初衷了。

首先，你需要根据自己的具体工作环境，选择或者制定适合自己公司或团队的编码规范。市面上可以参考的规范，也就是在文章开头提到的 PEP8 和 Google Style。

要知道，没有放之四海而皆准的规范，我们必须要因地制宜。例如在 Google 中，因为历史原因 C++ 不使用异常，引入异常对整个代码库带来的风险已经远大于它的益处，所以在它的 C++ 代码规范中，禁止使用异常。

一旦确定了整个团队所遵从的编码规范，就一定要强制执行，有什么好的办法呢？靠强制代码评审和强制静态或者动态 linter。具体流程是：

1. 在代码评审工具里，添加必须的编码规范环节；
2. 把团队确定的代码规范写进 Pylint 里，能够在每份代码提交前自动检查，不通过的代码无法提交。

整合之后，你的团队工作流程就会变成图 1 所示的这样。

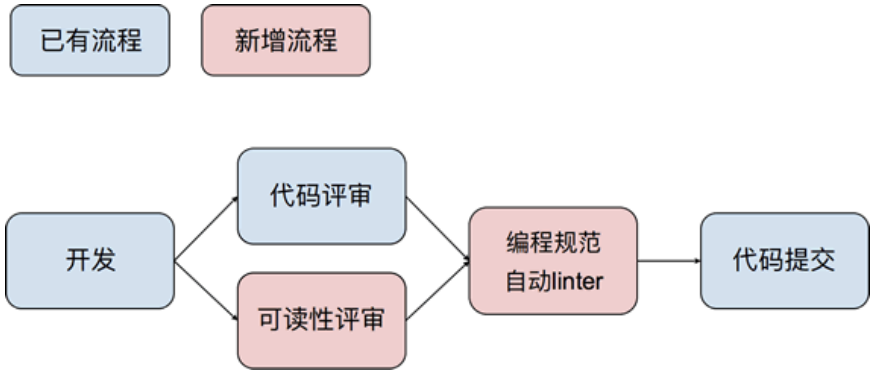


图 1 自动检查编码规范的工作流程

学到这里，相信你对代码风格的重要性有了全新的认识。

2.22 Python 标识符命名规范

简单地理解，标识符就是一个名字，就好像我们每个人都有属于自己的名字，它的主要作用就是作为变量、函数、类、模块以及其他对象的名称。

Python 中标识符的命名不是随意的，而是要遵守一定的命令规则，比如说：

1. 标识符是由字符（A~Z 和 a~z）、下划线和数字组成，但第一个字符不能是数字。
2. 标识符不能和 Python 中的保留字相同。有关保留字，后续章节会详细介绍。
3. Python 中的标识符中，不能包含空格、@、% 以及 $ 等特殊字符。
   例如，下面所列举的标识符是合法的：

```
UserID
name
mode12
user_age
```

以下命名的标识符不合法：

```
4word    #不能以数字开头
try      #try 是保留字，不能作为标识符
$money #不能包含特殊字符
```

4. 在 Python 中，标识符中的字母是严格区分大小写的，也就是说，两个同样的单词，如果大小格式不一样，多代表的意义也是完全不同的。比如说，下面这 3 个变量之间，就是完全独立、毫无关系的，它们彼此之间是相互独立的个体。

```
number = 0
Number = 0
NUMBER = 0
```

5. Python 语言中，以下划线开头的标识符有特殊含义，例如:

- 以单下划线开头的标识符（如 _width），表示不能直接访问的类属性，其无法通过 from...import* 的方式导入；
- 以双下划线开头的标识符（如__add）表示类的私有成员；
- 以双下划线作为开头和结尾的标识符（如 __init__），是专用标识符。

因此，除非特定场景需要，应避免使用以下划线开头的标识符。

另外需要注意的是，Python 允许使用汉字作为标识符，例如:

```
C 语言中文网 = "http://c.biancheng.net"
```

但我们应尽量避免使用汉字作为标识符，这会避免遇到很多奇葩的错误。

标识符的命名，除了要遵守以上这几条规则外，不同场景中的标识符，其名称也有一定的规范可循，例如：

- 当标识符用作模块名时，应尽量短小，并且全部使用小写字母，可以使用下划线分割多个字母，例如 game_mian、game_register 等。
- 当标识符用作包的名称时，应尽量短小，也全部使用小写字母，不推荐使用下划线，例如 com.mr、com.mr.book 等。
- 当标识符用作类名时，应采用单词首字母大写的形式。例如，定义一个图书类，可以命名为 Book。
- 模块内部的类名，可以采用 "下划线+首字母大写" 的形式，如 _Book;
- 函数名、类中的属性名和方法名，应全部使用小写字母，多个单词之间可以用下划线分割；
- 常量命名应全部使用大写字母，单词之间可以用下划线分割；

有读者可能会问，如果不遵守这些规范，会怎么样呢？答案是程序照样可以运行，但遵循以上规范的好处是，可以更加直观地了解代码所代表的含义，以 Book 类为例，我们可以很容易就猜到此类与书有关，虽然将类名改为 a（或其它）不会影响程序运行，但通常不这么做。

2.23 Python 关键字（保留字）一览表

保留字是 Python 语言中一些已经被赋予特定意义的单词，这就要求开发者在开发程序时，不能用这些保留字作为标识符给变量、函数、类、模板以及其他对象命名。

Python 包含的保留字可以执行如下命令进行查看：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

所有的保留字，如下表所示：

表 1 Python 保留字一览表

and	as	assert	break	class	continue
def	del	elif	else	except	finally
for	from	False	global	if	import
in	is	lambda	nonlocal	not	None
or	pass	raise	return	try	True
while	with	yield			

需要注意的是，由于 Python 是严格区分大小写的，保留字也不例外。所以，我们可以说 if 是保留字，但 IF 就不是保留字。

在实际开发中，如果使用 Python 中的保留字作为标识符，则解释器会提示“invalid syntax” 的错误信息，如图 2 所示。

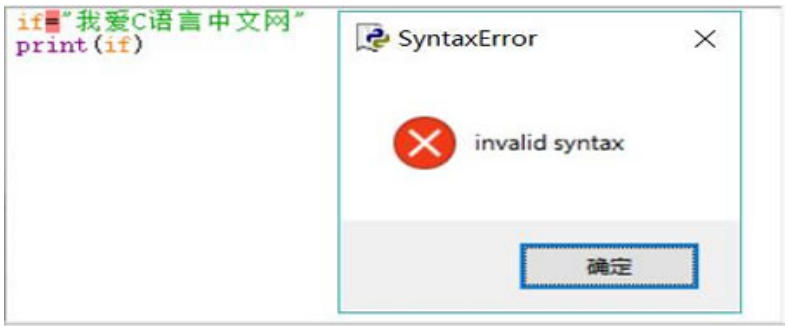


图 2 保留字作标识符报错信息示意图

2.24 Python 内置函数一览表

Python 解释器自带的函数叫做内置函数，这些函数可以直接使用，不需要导入某个模块。

如果你熟悉 Shell 编程，了解什么是 Shell 内置命令，那么你也很容易理解什么是 Python 内置函数，它们的概念是类似的。

> 将使用频繁的代码段封装起来，并给它起一个名字，以后使用的时候只要知道名字就可以，这就是函数。函数就是一段封装好的、可以重复使用的代码，它使得我们的程序更加模块化，不需要编写大量重复的代码。

内置函数和标准库函数是不一样的。

Python 解释器也是一个程序，它给用户提供了一些常用功能，并给它们起了独一无二的名字，这些常用功能就是内置函数。Python 解释器启动以后，内置函数也生效了，可以直接拿来使用。

Python 标准库相当于解释器的外部扩展，它并不会随着解释器的启动而启动，要想使用这些外部扩展，必须提前导入。Python 标准库非常庞大，包含了很多模块，要想使用某个函数，必须提前导入对应的模块，否则函数是无效的。

内置函数是解释器的一部分，它随着解释器的启动而生效；标准库函数是解释器的外部扩展，导入模块以后才能生效。一般来说，内置函数的执行效率要高于标准库函数。

Python 解释器一旦启动，所有的内置函数都生效了；而导入标准库的某个模块，只是该模块下的函数生效，并不是所有的标准库函数都生效。

内置函数的数量必须被严格控制，否则 Python 解释器会变得庞大和臃肿。一般来说，只有那些使用频繁或者和语言本身绑定比较紧密的函数，才会被提升为内置函数。

例如，在屏幕上输出文本就是使用最频繁的功能之一，所以 print() 是 Python 的内置函数。

> 在 Python 2.x 中，print 是一个关键字；到了 Python 3.x 中，print 变成了内置函数。

除了 print() 函数，Python 解释器还提供了更多内置函数，下表列出了 Python 3.x 中的所有内置函数。

表 1 Python 3.x 内置函数

内置函数				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slicea()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()

bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

表 1 中各个内置函数的具体功能和用法，可通过访问 https://docs.python.org/zh-cn/3/library/functions.html 进行查看。

注意，不要使用内置函数的名字作为标识符使用（例如变量名、函数名、类名、模板名、对象名等），虽然这样做 Python 解释器不会报错，但这会导致同名的内置函数被覆盖，从而无法使用。例如：

```
>>> print = "http://c.biancheng.net/python/"  #将 print 作为变量名
>>> print("Hello World!")  #print 函数被覆盖，失效
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("Hello World!")
TypeError: 'str' object is not callable
```

第 3 章：Python 变量类型和运算符

3.1 Python 变量的定义和使用

任何编程语言都需要处理数据，比如数字、字符串、字符等，我们可以直接使用数据，也可以将数据保存到变量中，方便以后使用。

变量（Variable）可以看成一个小箱子，专门用来“盛装”程序中的数据。每个变量都拥有独一无二的名字，通过变量的名字就能找到变量中的数据。

从底层看，程序中的数据最终都要放到内存（内存条）中，变量其实就是这块内存的名字。

和变量相对应的是常量（Constant），它们都是用来“盛装”数据的小箱子，不同的是：变量保存的数据可以被多次修改，而常量一旦保存某个数据之后就不能修改了。

Python 变量的赋值

在编程语言中，将数据放入变量的过程叫做赋值（Assignment）。Python 使用等号=作为赋值运算符，具体格式为：

```
name = value
```

name 表示变量名；value 表示值，也就是要存储的数据。

注意，变量是标识符的一种，它的名字不能随便起，要遵守 Python 标识符命名规范，还要避免和 Python 内置函数以及 Python 保留字重名。

例如，下面的语句将整数 10 赋值给变量 n：

```
n = 10
```

从此以后，n 就代表整数 10，使用 n 也就是使用 10。

更多赋值的例子：

```
1.  pi = 3.1415926  #将圆周率赋值给变量 pi
2.  url = "http://c.biancheng.net/python/"  #将 Python 教程的地址赋值给变量 str
3.  real = True  #将布尔值赋值给变量 real
```

变量的值不是一成不变的，它可以随时被修改，只要重新赋值即可；另外你也不用关心数据的类型，可以将不同类型的数据赋值给同一个变量。请看下面的演示：

```
1.  n = 10  #将 10 赋值给变量 n
```

```
n = 95  #将 95 赋值给变量 n
n = 200  #将 200 赋值给变量 n

abc = 12.5  #将小数赋值给变量 abc
abc = 85  #将整数赋值给变量 abc
abc = "http://c.biancheng.net/"  #将字符串赋值给变量 abc
```

注意，变量的值一旦被修改，之前的值就被覆盖了，不复存在了，再也找不回了。换句话说，变量只能容纳一个值。除了赋值单个数据，你也可以将表达式的运行结果赋值给变量，例如：

```
sum = 100 + 20  #将加法的结果赋值给变量
rem = 25 * 30 % 7  #将余数赋值给变量
str = "C 语言中文网" + "http://c.biancheng.net/"  #将字符串拼接的结果赋值给变量
```

Python 变量的使用

使用 Python 变量时，只要知道变量的名字即可。

几乎在 Python 代码的任何地方都能使用变量，请看下面的演示：

```
>>> n = 10
>>> print(n)  #将变量传递给函数
10
>>> m = n * 10 + 5  #将变量作为四则运算的一部分
>>> print(m)
105
>>> print(m-30)  #将由变量构成的表达式作为参数传递给函数
75
>>> m = m * 2  #将变量本身的值翻倍
>>> print(m)
210
>>> url = "http://c.biancheng.net/cplus/"
>>> str = "C++教程：" + url  #字符串拼接
>>> print(str)
```

```
C++教程：http://c.biancheng.net/cplus/
```

Python 是弱类型的语言

在强类型的编程语言中，定义变量时要指明变量的类型，而且赋值的数据也必须是相同类型的，C 语言、C++、Java 是强类型语言的代表。

下面我们以 C++ 为例来演示强类型语言中变量的使用：

```
int n = 10;  //int 表示整数类型
n = 100;
n = "http://c.biancheng.net/socket/";  //错误：不能将字符串赋值给整数类型

url = "http://c.biancheng.net/java/";  //错误：没有指明类型的变量是没有定义的，不能使用。
```

和强类型语言相对应的是弱类型语言，Python、JavaScript、PHP 等脚本语言一般都是弱类型的。

弱类型语言有两个特点：

- 变量无须声明就可以直接赋值，对一个不存在的变量赋值就相当于定义了一个新变量。
- 变量的数据类型可以随时改变，比如，同一个变量可以一会儿被赋值为整数，一会儿被赋值为字符串。

注意，弱类型并不等于没有类型！弱类型是说在书写代码时不用刻意关注类型，但是在编程语言的内部仍然是有类型的。我们可以使用 type() 内置函数类检测某个变量或者表达式的类型，例如：

```
>>> num = 10
>>> type(num)
<class 'int'>
>>> num = 15.8
>>> type(num)
<class 'float'>
>>> num = 20 + 15j
>>> type(num)
<class 'complex'>
>>> type(3*15.6)
<class 'float'>
```

3.2 Python 整数类型（int）详解

整数就是没有小数部分的数字，Python 中的整数包括正整数、0 和负整数。

有些强类型的编程语言会提供多种整数类型，每种类型的长度都不同，能容纳的整数的大小也不同，开发者要根据实际数字的大小选用不同的类型。例如 C 语言提供了 short、int、long、long long 四种类型的整数，它们的长度依次递增，初学者在选择整数类型时往往比较迷惑，有时候还会导致数值溢出。

而 Python 则不同，它的整数不分类型，或者说它只有一种类型的整数。Python 整数的取值范围是无限的，不管多大或者多小的数字，Python 都能轻松处理。

> 当所用数值超过计算机自身的计算能力时，Python 会自动转用高精度计算（大数计算）。

请看下面的代码：

```
#将 78 赋值给变量 n
n = 78
print(n)
print( type(n) )

#给 x 赋值一个很大的整数
x = 888888888888888888888
print(x)
print( type(x) )

#给 y 赋值一个很小的整数
y = -77777777777777777777
print(y)
print( type(y) )
```

运行结果：

```
78
<class 'int'>
888888888888888888888
<class 'int'>
-77777777777777777777
<class 'int'>
```

x 是一个极大的数字，y 是一个很小的数字，Python 都能正确输出，不会发生溢出，这说明 Python 对整数的处理能力非常强大。

不管对于多大或者多小的整数，Python 只用一种类型存储，就是 int。

关于 Python 2.x

Python 3.x 只用 int 一种类型存储整数，但是 Python 2.x 会使用 long 类型来存储较大的整数。以上代码在 Python 2.x 下的运行结果为：

```
78
<type 'int'>
888888888888888888888
<type 'long'>
-777777777777777777777
<type 'long'>
```

但是不管哪个版本的 Python，都能轻松处理极大和极小的数字，而且程序员也不用操心底层到底使用了 int 还是 long 类型。

整数的不同进制

在 Python 中，可以使用多种进制来表示整数：

1) 十进制形式

我们平时常见的整数就是十进制形式，它由 0~9 共十个数字排列组合而成。

注意，使用十进制形式的整数不能以 0 作为开头，除非这个数值本身就是 0。

2) 二进制形式

由 0 和 1 两个数字组成，书写时以 `0b` 或 `0B` 开头。例如，101 对应十进制数是 5。

3) 八进制形式

八进制整数由 0~7 共八个数字组成，以 `0o` 或 `0O` 开头。注意，第一个符号是数字 0，第二个符号是大写或小写的字母 O。

在 Python 2.x 中，八进制数字还可以直接以 `0`（数字零）开头。

4) 十六进制形式

由 0~9 十个数字以及 A~F（或 a~f）六个字母组成，书写时以 `0x` 或 `0X` 开头，

如果你对不同进制以及它们之间的转换方法不了解，请猛击下面的链接：

- 进制详解：二进制、八进制和十六进制
- 进制转换：二进制、八进制、十六进制、十进制之间的转换

【实例】不同进制整数在 Python 中的使用：

```
#十六进制
hex1 = 0x45
hex2 = 0x4Af
print("hex1Value: ", hex1)
print("hex2Value: ", hex2)

#二进制
bin1 = 0b101
print('bin1Value: ', bin1)
bin2 = 0B110
print('bin2Value: ', bin2)

#八进制
oct1 = 0o26
print('oct1Value: ', oct1)
oct2 = 0041
print('oct2Value: ', oct2)
```

运行结果：

```
hex1Value:  69
hex2Value:  1199
bin1Value:  5
bin2Value:  6
oct1Value:  22
oct2Value:  33
```

本例的输出结果都是十进制整数。

数字分隔符

为了提高数字的的可读性，Python 3.x 允许使用下划线_作为数字（包括整数和小数）的分隔符。通常每隔三个数字添加一个下划线，类似于英文数字中的逗号。下划线不会影响数字本身的值。

【实例】使用下划线书写数字：

```
click = 1_301_547
```

```
2.  distance = 384_000_000
3.  print("Python 教程阅读量：", click)
4.  print("地球和月球的距离：", distance)
```

运行结果：

```
Python 教程阅读量：1301547
地球和月球的距离：384000000
```

3.3 Python 小数/浮点数（float）类型详解

在编程语言中，小数通常以浮点数的形式存储。浮点数和定点数是相对的：小数在存储过程中如果小数点发生移动，就称为浮点数；如果小数点不动，就称为定点数。

> 如果你对浮点数的底层存储格式不了解，请猛击：小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）

Python 中的小数有两种书写形式：

1) 十进制形式

这种就是我们平时看到的小数形式，例如 34.6、346.0、0.346。

书写小数时必须包含一个小数点，否则会被 Python 当作整数处理。

2) 指数形式

Python 小数的指数形式的写法为：

```
aEn 或 aen
```

a 为尾数部分，是一个十进制数；n 为指数部分，是一个十进制整数；`E` 或 `e` 是固定的字符，用于分割尾数部分和指数部分。整个表达式等价于 $a\times10^n$。

指数形式的小数举例：

- 2.1E5 = 2.1×10^5，其中 2.1 是尾数，5 是指数。
- 3.7E-2 = 3.7×10^{-2}，其中 3.7 是尾数，-2 是指数。
- 0.5E7 = 0.5×10^7，其中 0.5 是尾数，7 是指数。

注意，只要写成指数形式就是小数，即使它的最终值看起来像一个整数。例如 14E3 等价于 14000，但 14E3 是一个小数。
Python 只有一种小数类型，就是 float。C 语言有两种小数类型，分别是 float 和 double：float 能容纳的小数范围比较小，double 能容纳的小数范围比较大。

【实例】小数在 Python 中的使用：

```
f1 = 12.5
print("f1Value: ", f1)
print("f1Type: ", type(f1))

f2 = 0.34557808421257003
print("f2Value: ", f2)
print("f2Type: ", type(f2))

```

```
f3 = 0.00000000000000000000000000847
print("f3Value: ", f3)
print("f3Type: ", type(f3))

f4 = 345679745132456787324523453.45006
print("f4Value: ", f4)
print("f4Type: ", type(f4))

f5 = 12e4
print("f5Value: ", f5)
print("f5Type: ", type(f5))

f6 = 12.3 * 0.1
print("f6Value: ", f6)
print("f6Type: ", type(f6))
```

运行结果：

```
f1Value:  12.5
f1Type:  <class 'float'>
f2Value:  0.34557808421257
f2Type:  <class 'float'>
f3Value:  8.47e-26
f3Type:  <class 'float'>
f4Value:  3.456797451324568e+26
f4Type:  <class 'float'>
f5Value:  120000.0
f5Type:  <class 'float'>
f6Value:  1.2300000000000002
f6Type:  <class 'float'>
```

从运行结果可以看出，Python 能容纳极小和极大的浮点数。print 在输出浮点数时，会根据浮点数的长度和大小适当的舍去一部分数字，或者采用科学计数法。

f5 的值是 120000，但是它依然是小数类型，而不是整数类型。

让人奇怪的是 f6，`12.3*0.1` 的计算结果很明显是 1.23，但是 print 的输出却不精确。这是因为小数在内存中是以二进制形式存储的，小数点后面的部分在转换成二进制时很有可能是一串无限循环的数字，无论如何都不能精确表示，所以小数的计算结果一般都是不精确的。有兴趣的读者请猛击下面的链接深入学习：

- 进制转换：二进制、八进制、十六进制、十进制之间的转换
- 小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）

3.4 Python 复数类型（complex）详解

复数（Complex）是 Python 的内置类型，直接书写即可。换句话说，Python 语言本身就支持复数，而不依赖于标准库或者第三方库。

复数由实部（real）和虚部（imag）构成，在 Python 中，复数的虚部以 j 或者 J 作为后缀，具体格式为：

```
a + bj
```

a 表示实部，b 表示虚部。

【实例】Python 复数的使用：

```
1.  c1 = 12 + 0.2j
2.  print("c1Value: ", c1)
3.  print("c1Type", type(c1))
4.
5.  c2 = 6 - 1.2j
6.  print("c2Value: ", c2)
7.
8.  #对复数进行简单计算
9.  print("c1+c2: ", c1+c2)
10. print("c1*c2: ", c1*c2)
```

运行结果：

```
c1Value:  (12+0.2j)
c1Type <class 'complex'>
c2Value:  (6-1.2j)
c1+c2:  (18-1j)
c1*c2:  (72.24-13.2j)
```

可以发现，复数在 Python 内部的类型是 complex，Python 默认支持对复数的简单计算。

3.5 Python 浮点数精度问题（包含解决方案）

前面章节提到，Python 中浮点类型之间的运算，其结果并不像我们想象的那样，例如：

```
>>> 0.1+0.2
0.30000000000000004
>>> 0.1+0.1-0.2
0.0
>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17
>>> 0.1+0.1+0.1-0.2
0.10000000000000003
```

为什么在计算这么简单的问题上，计算机会出现这样的低级错误呢？真正的原因在于十进制和数和二进制数的转换。

我们知道，计算机其实是不认识十进制数，它只认识二进制数，也就是说，当我们以十进制数进行运算的时候，计算机需要将各个十进制数转换成二进制数，然后进行二进制间的计算。

以类似 0.1 这样的浮点数为例，如果手动将其转换成二进制，其结果为：

$$0.1_{(10)}=0.00011001100110011\ldots_{(2)}$$

可以看到，结果是无限循环的，也就是说，0.1 转换成二进制数后，无法精确到等于十进制数的 0.1。同时，由于计算机存储的位数是有限制的，所以如果要存储的二进制位数超过了计算机存储位数的最大值，其后续位数会被舍弃（舍弃的原则是“0 舍 1 入”）。

这种问题不仅在 Python 中存在，在所有支持浮点数运算的编程语言中都会遇到，它不光是 Python 的 Bug。明白了问题产生的原因之后，那么该如何解决呢？就 Python 的浮点数运算而言，大多数计算机每次计算误差不会超过 2^{53}，这对于大多数任务来说已经足够了。

如果需要非常精确的结果，可以使用 decimal 模块（其实就是别人开发好的程序，我们可以直接拿来用），它实现的十进制数运算适合会计方面的应用和有高精度要求的应用。例如：

```
#使用模块前，需要使用 import 引入
import decimal
a = decimal.Decimal("10.0")
b = decimal.Decimal("3")
print(10.0/3)
print(a/b)
```

运行结果为：

```
3.3333333333333335
3.333333333333333333333333333
```

可以看到，相比普通运算的结果，使用 decimal 模块得到的结果更精确。

如果 decimal 模块还是无法满足需求，还可以使用 fractions 模块，例如：

```
#引入 decimal 模块
from fractions import Fraction
print(10/3)
print(Fraction(10,3))
```

运行结果为：

```
3.3333333333333335
10/3
```

可以看到，通过 fractions 模块能很好地解决浮点类型数之间运算的问题。

3.6 Python 字符串详解（包含长字符串和原始字符串）

若干个字符的集合就是一个字符串（String）。Python 中的字符串必须由双引号" "或者单引号' '包围，具体格式为：

```
"字符串内容"
'字符串内容'
```

字符串的内容可以包含字母、标点、特殊符号、中文、日文等全世界的所有文字。

下面都是合法的字符串：

- "123789"
- "123abc"
- "http://c.biancheng.net/python/"
- "C 语言中文网成立 8 年了"

Python 字符串中的双引号和单引号没有任何区别。而有些编程语言的双引号字符串可以解析变量，单引号字符串一律原样输出，例如 PHP 和 JavaScript。

处理字符串中的引号的

当字符串内容中出现引号时，我们需要进行特殊处理，否则 Python 会解析出错，例如：

```
'I'm a great coder!'
```

由于上面字符串中包含了单引号，此时 Python 会将字符串中的单引号与第一个单引号配对，这样就会把'I'当成字符串，而后面的 m a great coder!'就变成了多余的内容，从而导致语法错误。

对于这种情况，我们有两种处理方案：

1) 对引号进行转义

在引号前面添加反斜杠\就可以对引号进行转义，让 Python 把它作为普通文本对待，例如：

```
1.  str1 = 'I\'m a great coder!'
2.  str2 = "引文双引号是\", 中文双引号是“”
3.  print(str1)
4.  print(str2)
```

运行结果：

```
I'm a great coder!
引文双引号是"，中文双引号是 “
```

2) 使用不同的引号包围字符串

如果字符串内容中出现了单引号，那么我们可以使用双引号包围字符串，反之亦然。例如：

```
str1 = "I'm a great coder!"  #使用双引号包围含有单引号的字符串
str2 = '引文双引号是"，中文双引号是 “'  #使用单引号包围含有双引号的字符串
print(str1)
print(str2)
```

运行结果和上面相同。

字符串的换行

Python 不是格式自由的语言，它对程序的换行、缩进都有严格的语法要求。要想换行书写一个比较长的字符串，必须在行尾添加反斜杠\，请看下面的例子：

```
s2 = 'It took me six months to write this Python tutorial. \
    Please give me more support. \
    I will keep it updated.'
```

上面 s2 字符串的比较长，所以使用了转义字符\对字符串内容进行了换行，这样就可以把一个长字符串写成多行。

另外，Python 也支持表达式的换行，例如：

```
num = 20 + 3 / 4 + \
    2 * 3
print(num)
```

Python 长字符串

在《Python 注释》一节中我们提到，使用三个单引号或者双引号可以对多行内容进行注释，这其实是 Python 长字符串的写法。所谓长字符串，就是可以直接换行（不用加反斜杠\）书写的字符串。

Python 长字符串由三个双引号"""或者三个单引号'''包围，语法格式如下：

```
"""长字符串内容"""
'''长字符串内容'''
```

在长字符串中放置单引号或者双引号不会导致解析错误。

如果长字符串没有赋值给任何变量，那么这个长字符串就不会起到任何作用，和一段普通的文本无异，相当于被注释掉了。

> 注意，此时 Python 解释器并不会忽略长字符串，也会按照语法解析，只是长字符串起不到实际作用而已。

当程序中有大段文本内容需要定义成字符串时，优先推荐使用长字符串形式，因为这种形式非常强大，可以在字符串中放置任何内容，包括单引号和双引号。

【实例】将长字符串赋值给变量：

```
longstr = '''It took me 6 months to write this Python tutorial.
Please give me a to 'thumb' to keep it updated.
The Python tutorial is available at http://c.biancheng.net/python/.'''
print(longstr)
```

长字符串中的换行、空格、缩进等空白符都会原样输出，所以你不能写成下面的样子：

```
longstr = '''
    It took me 6 months to write this Python tutorial.
    Please give me a to 'thumb' to keep it updated.
    The Python tutorial is available at http://c.biancheng.net/python/.
'''
print(longstr)
```

虽然这样写格式优美，但是输出结果将变成：

```
It took me 6 months to write this Python tutorial.
Please give me a to 'thumb' to keep it updated.
The Python tutorial is available at http://c.biancheng.net/python/.
```

字符串内容前后多出了两个空行，并且每一行的前面会多出四个空格。

Python 原始字符串

Python 字符串中的反斜杠`\`有着特殊的作用，就是转义字符，例如上面提到的`\'`和`\"`，我们将在《Python 转义字符》一节中详细讲解，这里大家先简单了解。

转义字符有时候会带来一些麻烦，例如我要表示一个包含 Windows 路径 `D:\Program Files\Python 3.8\python.exe` 这样的字符串，在 Python 程序中直接这样写肯定是不行的，不管是普通字符串还是长字符串。

因为\的特殊性，我们需要对字符串中的每个\都进行转义，也就是写成 D:\\Program Files\\Python 3.8\\python.exe 这种形式才行。

这种写法需要特别谨慎，稍有疏忽就会出错。为了解决转义字符的问题，Python 支持原始字符串。在原始字符串中，\不会被当作转义字符，所有的内容都保持“原汁原味”的样子。

在普通字符串或者长字符串的开头加上 r 前缀，就变成了原始字符串，具体格式为：

```
str1 = r'原始字符串内容'
str2 = r"""原始字符串内容"""
```

将上面的 Windows 路径改写成原始字符串的形式：

```
1.  rstr = r'D:\Program Files\Python 3.8\python.exe'
2.  print(rstr)
```

原始字符串中的引号

如果普通格式的原始字符串中出现引号，程序同样需要对引号进行转义，否则 Python 照样无法对字符串的引号精确配对；但是和普通字符串不同的是，此时用于转义的反斜杠会变成字符串内容的一部分。

请看下面的代码：

```
1.  str1 = r'I\'m a great coder!'
2.  print(str1)
```

输出结果：

```
I\'m a great coder!
```

需要注意的是，Python 原始字符串中的反斜杠仍然会对引号进行转义，因此原始字符串的结尾处不能是反斜杠，否则字符串结尾处的引号会被转义，导致字符串不能正确结束。

在 Python 中有两种方式解决这个问题：一种方式是改用长字符串的写法，不要使用原始字符串；另一种方式是单独书写反斜杠，这是接下来要重点说明的。

例如想表示 D:\Program Files\Python 3.8\，可以这样写：

```
1.  str1 = r'D:\Program Files\Python 3.8' '\\'
2.  print(str1)
```

我们先写了一个原始字符串 r'D:\Program Files\Python 3.8'，紧接着又使用'\\'写了一个包含转义字符的普通字符串，Python 会自动将这两个字符串拼接在一起，所以上面代码的输出结果是：

```
D:\Program Files\Python 3.8\
```

由于这种写法涉及到了字符串拼接的相关知识，这里读者只需要了解即可，后续会对字符串拼接做详细介绍。

3.7 Python 字符串使用哪种编码格式？

在实践中，很多初学者都遇到过“文件显示乱码”的情况，其多数都是由于在打开文件时，没有选对编码格式导致的。因此，学习 Python 中的字符或字符串，了解其底层的编码格式是非常有必要的。

鉴于有些读者并不了解什么是编码格式，本节先从编码开始讲起。

什么是编码？

虽然很多教程中有关于编码的定义，但对初学者来说并不容易理解，这里先举一个例子。古代打仗，击鼓为号、鸣金收兵，即把要传达给士兵的命令对应为公认的其他形式，这就和编码有相似之处。

以发布进攻命令为例，相比用嗓子喊，敲鼓发出的声音传播的更远，并且士兵听到后也不会引起歧义，因此长官下达进攻命令后，传令员就将此命令转化为对应的鼓声，这个转化的过程称为编码；由于士兵都接受过训练，听到鼓声后，他们可以将其转化为对应的进攻命令，这个转化的过程称为解码。

需要说明的是，此例只是形象地描述了编码和解码的原理，真实的编码和解码过程比这要复杂的多。
了解了编码的含义之后，接下来再介绍一下字符编码。

什么是字符编码？

我们知道，计算机是以二进制的形式来存储数据的，即它只认识 0 和 1 两个数字。 20 世纪 60 年代，是计算机发展的早期，这时美国是计算机领域的老大，它制定了一套编码标准，解决了 128 个英文字符与二进制之间的对应关系，被称为 ASCII 字符编码（简称 ASCII 码）。

ASCII 码，全称为美国信息交换标准代码，是基于拉丁字母的一套字符编码，主要用于显示现代英语，因为万维网的出现，使得 ASCII 码广为使用，其直到 2007 年 12 月才逐渐被 Unicode 取代。
虽然英语用 128 个字符编码已经够用，但计算机不仅仅用于英语，如果想表示其他语言，128 个符号显然不够用，所以很多其他国家都在 ASCII 的基础上发明了很多别的编码，例如包含了汉语简体中文格式的 GB2312 编码格式（使用 2 个字节表示一个汉字）。

也正是由于出现了很多种编码格式，导致了“文件显示乱码”的情况。比如说，发送邮件时，如果发信人和收信人使用的编码格式不一样，则收信人很可能看到乱码的邮件。基于这个原因，Unicode 字符集应运而生。

Unicode 字符集又称万国码、国际码、统一码等。从名字就可以看出来，它是以统一符号为目标的字符集。Unicode 对世界上大部分的文字系统进行了整理、编码，使得电脑可以用更简单的方式来呈现和处理文字。

注意，在实际使用时，人们常常混淆字符集和字符编码这两个概念，我认为它们是不同的：

- 字符集定义了字符和二进制的对应关系，为每个字符分配了唯一的编号。可以将字符集理解成一个很大的表格，它列出了所有字符和二进制的对应关系，计算机显示文字或者存储文字，就是一个查表的过程；

- 而字符编码规定了如何将字符的编号存储到计算机中，要知道，有些字符编码（如 GB2312 和 GBK）规定，不同字符在存储时所占用的字节数是不一样的，因此为了区分一个字符到底使用了几个字节，就不能将字符的编号直接存储到计算机中，字符编号在存储之前必须要经过转换，在读取时还要再逆向转换一次，这套转换方案就叫做字符编码。

Unicode 字符集可以使用的编码方案有三种，分别是：

1. UTF-8：一种变长的编码方案，使用 1~6 个字节来存储；
2. UTF-32：一种固定长度的编码方案，不管字符编号大小，始终使用 4 个字节来存储；
3. UTF-16：介于 UTF-8 和 UTF-32 之间，使用 2 个或者 4 个字节来存储，长度既固定又可变。

其中，UTF-8 是目前使用最广的一种 Unicode 字符集的实现方式，可以说它几乎已经一统江湖了。

Python 使用哪种字符编码？

了解了什么是编码，以及什么是字符编码之后，最后解决“Python 使用哪种字符编码？”这个问题。

Python 3.x 中，字符串采用的是 Unicode 字符集，可以用如下代码来查看当前环境的编码格式：

```
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
```

同时，在 Python 3.x 中也可以用 ord() 和 chr() 函数实现字符和编码数字之间的转换，例如：

```
>>> ord('Q')
81
>>> chr(81)
'Q'
>>> ord("网")
32593
>>> chr(32593)
'网'
```

Python 2.x 中无法使用 ord() 得到指定字符对应的编码数字。
由此可以知道，在 Unicode 字符集中，字符‘Q’对应的编码数字为 81，而中文‘网’对应的编码数字为 32593。

值得一提的是，虽然 Python 默认采用 UTF-8 编码，但它也提供了 encode() 方法，可以轻松实现将 Unicode 编码格式的字符串转化为其它编码格式。有关 encode() 方法的用法，可阅读《Python encode()和 decode() 方法》一节。

3.8 Python bytes 类型及用法

Python bytes 类型用来表示一个字节串。“字节串“不是编程术语，是我自己“捏造”的一个词，用来和字符串相呼应。

> bytes 是 Python 3.x 新增的类型，在 Python 2.x 中是不存在的。

字节串（bytes）和字符串（string）的对比：

- 字符串由若干个字符组成，以字符为单位进行操作；字节串由若干个字节组成，以字节为单位进行操作。
- 字节串和字符串除了操作的数据单元不同之外，它们支持的所有方法都基本相同。
- 字节串和字符串都是不可变序列，不能随意增加和删除数据。

bytes 只负责以字节序列的形式（二进制形式）来存储数据，至于这些数据到底表示什么内容（字符串、数字、图片、音频等），完全由程序的解析方式决定。如果采用合适的字符编码方式（字符集），字节串可以恢复成字符串；反之亦然，字符串也可以转换成字节串。

说白了，bytes 只是简单地记录内存中的原始数据，至于如何使用这些数据，bytes 并不在意，你想怎么使用就怎么使用，bytes 并不约束你的行为。

bytes 类型的数据非常适合在互联网上传输，可以用于网络通信编程；bytes 也可以用来存储图片、音频、视频等二进制格式的文件。

字符串和 bytes 存在着千丝万缕的联系，我们可以通过字符串来创建 bytes 对象，或者说将字符串转换成 bytes 对象。有以下三种方法可以达到这个目的：

- 如果字符串的内容都是 ASCII 字符，那么直接在字符串前面添加 b 前缀就可以转换成 bytes。
- bytes 是一个类，调用它的构造方法，也就是 bytes()，可以将字符串按照指定的字符集转换成 bytes；如果不指定字符集，那么默认采用 UTF-8。
- 字符串本身有一个 encode() 方法，该方法专门用来将字符串按照指定的字符集转换成对应的字节串；如果不指定字符集，那么默认采用 UTF-8。

【实例】使用不同方式创建 bytes 对象：

```
#通过构造函数创建空 bytes
b1 = bytes()
#通过空字符串创建空 bytes
b2 = b''

#通过 b 前缀将字符串转换成 bytes
b3 = b'http://c.biancheng.net/python/'
print("b3: ", b3)
```

```
print(b3[3])
print(b3[7:22])

#为 bytes() 方法指定字符集
b4 = bytes('C 语言中文网 8 岁了', encoding='UTF-8')
print("b4: ", b4)

#通过 encode() 方法将字符串转换成 bytes
b5 = "C 语言中文网 8 岁了".encode('UTF-8')
print("b5: ", b5)
```

运行结果：

```
b3:  b'http://c.biancheng.net/python/'
112
b'c.biancheng.net'
b4:  b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x918\xe5\xb2\x81\xe4\xba\x86'
b5:  b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x918\xe5\xb2\x81\xe4\xba\x86'
```

从运行结果可以发现，对于非 ASCII 字符，print 输出的是它的字符编码值（十六进制形式），而不是字符本身。非 ASCII 字符一般占用两个字节以上的内存，而 bytes 是按照单个字节来处理数据的，所以不能一次处理多个字节。如果你对进制不了解，请猛击：

- 进制详解：二进制、八进制和十六进制
- 进制转换：二进制、八进制、十六进制、十进制之间的转换

如果你对字符集（字符编码）不了解，请猛击：

- ASCII 编码，将英文存储到计算机
- GB2312 编码和 GBK 编码，将中文存储到计算机
- Unicode 字符集，将全世界的文字存储到计算机

bytes 类也有一个 decode() 方法，通过该方法可以将 bytes 对象转换为字符串。紧接上面的程序，添加以下代码：

```
#通过 decode() 方法将 bytes 转换成字符串
str1 = b5.decode('UTF-8')
print("str1: ", str1)
```

输出结果：

```
str1:  C 语言中文网 8 岁了
```

3.9 Python bool 布尔类型

Python 提供了 bool 类型来表示真（对）或假（错），比如常见的 `5 > 3` 比较算式，这个是正确的，在程序世界里称之为真（对），Python 使用 True 来代表；再比如 `4 > 20` 比较算式，这个是错误的，在程序世界里称之为假（错），Python 使用 False 来代表。

> True 和 False 是 Python 中的关键字，当作为 Python 代码输入时，一定要注意字母的大小写，否则解释器会报错。

值得一提的是，布尔类型可以当做整数来对待，即 True 相当于整数值 1，False 相当于整数值 0。因此，下边这些运算都是可以的：

```
>>> False+1
1
>>> True+1
2
```

注意，这里只是为了说明 True 和 Flase 对应的整型值，在实际应用中是不妥的，不要这么用。

总的来说，bool 类型就是用于代表某个事情的真（对）或假（错），如果这个事情是正确的，用 True（或 1）代表；如果这个事情是错误的，用 False（或 0）代表。

【例 1】

```
>>> 5>3
True
>>> 4>20
False
```

在 Python 中，所有的对象都可以进行真假值的测试，包括字符串、元组、列表、字典、对象等，由于目前尚未学习，因此这里不做过多讲述，后续遇到时会做详细的介绍。

3.10 Python 缓存重用机制

Python 缓冲机制是为提高程序执行的效率服务的，实际上就是在 Python 解释器启动时从内存空间中开辟出一小部分，用来存储高频使用的数据，这样可以大大减少高频使用的数据创建时申请内存和销毁时撤销内存的开销。

Python 在存储数据时，会根据数据的读取频繁程度以及内存占用情况来考虑，是否按照一定的规则将数据存储缓存中。那么问题来了，内存重用机制适用于哪些基本数据类型呢？

表 1 罗列了 Python 是否将指定数据存入缓存中的规则：

表 1 Python 缓存重用规则

数据类型	是否可以重用	生效范围
范围在 [-5, 256] 之间的小整数	如果之前在程序中创建过，就直接存入缓存，后续不再创建。	全局
bool 类型		
字符串类型数据		
大于 256 的整数	只要在本代码块内创建过，就直接缓存，后续不再创建。	本代码块
大于 0 的浮点型小数		
小于 0 的浮点型小数	不进行缓存，每次都需要额外创建。	
小于 -5 的整数		

下面直接通过一段程序来演示 Python 缓存机制的规则。

```
#范围在 [-5, 256] 之间的小整数
int1 = -5
int2 = -5
print("[-5, 256] 情况下的两个变量：", id(int1), id(int2))

#bool 类型
bool1 = True
bool2 = True
print("bool 类型情况下的两个变量：",id(bool1),id(bool2))

#对于字符串
s1 = "3344"
s2 = "3344"
print("字符串情况下的两个交量", id(s1), id(s2))

```

```
#大于 256 的整数
int3 = 257
int4 = 257
print("大于 256 的整数情况下的两个变量", id(int3), id(int4))

#大于 0 的浮点数
f1 = 256.4
f2 = 256.4
print("大于 0 的浮点数情况下的两个变量", id(f1), id(f2))

#小于 0 的浮点数
f3 = -2.45
f4 = -2.45
print("小于 0 的浮点数情况下的两个变量", id(f3), id(f4))

#小于 -5 的整数
n1 = -6
n2 = -6
print("小于 -5 的整数情况下的两个变量", id(n1), id(n2))
```

注意，此程序中，大量使用 id() 内置函数，该函数的功能是获取变量（对象）所在的内存地址。运行该程序，其输出结果为：

```
[-5, 256] 情况下的两个变量： 1792722416 1792722416
bool 类型情况下的两个变量： 1792241888 1792241888
字符串情况下的两个交量 2912801330712 2912801330712
大于 256 的整数情况下的两个变量 2912801267920 2912801267920
大于 0 的浮点数情况下的两个变量 2912762210728 2912762210728
小于 0 的浮点数情况下的两个变量 2912762211016 2912762211040
小于 -5 的整数情况下的两个变量 2912801267952 2912801267984
```

以上输出结果中，每行都输出了 2 个相对应的变量所在的内存地址，如果相等，则表明 Python 内部对其使用了缓存机制，反之则没有。读者可对照以上输出结果来理解表 1 中有关变量缓存机制的规则。

另外，对于表 1 中所提到的代码块，Python 中的函数和类都被认为是在程序中开辟了一块新的代码块。以函数为例，函数内部的代码分属一个代码块，函数外部的代码属于另一个代码块。

有关函数的具有用法，后续章节会详细介绍，这里读者只需要知道函数中包含的代码，属于一个新的代码块即可。

由表 1 可以看到，Python 缓存机制在不同的代码块中也会有不同的表现。举一个例子，在上面例子代码的基础上，继续编写如下程序：

```
def fun():
    #[-5,256]
    int1 = -5
    print("fun 中 -5 的存储状态",id(int1), id(int2))

    #bool 类型
    bool3 = True
    print("fun 中 bool 类型的存储状态",id(bool3),id(bool2))

    #字符串类型
    s1 = "3344"
    print("fun 中 3344 字符串的存储状态", id(s1), id(s2))

    #大于 256
    int3 = 257
    print("fun 中 257 的存储状态", id(int3), id(int4))

    #浮点类型
    f1 = 256.4
    print("fun 中 256.4 的存储状态",id(f1), id(f2))

    #小于 -5
    n1 = -6
    print("fun 中 -6 的存储状态", id(n1), id(n2))

fun()
```

输出结果为：

```
fun 中 -5 的存储状态 1792722416 1792722416
fun 中 bool 类型的存储状态 1792241888 1792241888
fun 中 3344 字符串的存储状态 1976405206496 1976405206496
fun 中 257 的存储状态 1976405225648 1976405225680
fun 中 256.4 的存储状态 1976394459752 1976394459872
fun 中 -6 的存储状态 1976404744880 1976405225744
```

根据输出结果可以分析出：

1. 从 -5 、bool 类型以及字符串 "3344" 的输出结果可以得知，无论是在同一代码块，还是不同的代码块，它们都使用相同的缓存内容；

2. 从 257 和 256.4 的输出结果可以得知，如果位于同一代码块，则使用相同的缓存内容；反之，则不使用；
3. 从 -6 的输出结果得知，Python 没有对其缓存进行操作。

3.11 Python input()函数：获取用户输入的字符串

input() 是 Python 的内置函数，用于从控制台读取用户输入的内容。input() 函数总是以字符串的形式来处理用户输入的内容，所以用户输入的内容可以包含任何字符。

input() 函数的用法为：

```
str = input(tipmsg)
```

说明：

- str 表示一个字符串类型的变量，input 会将读取到的字符串放入 str 中。
- tipmsg 表示提示信息，它会显示在控制台上，告诉用户应该输入什么样的内容；如果不写 tipmsg，就不会有任何提示信息。

【实例】input() 函数的简单使用：

```
a = input("Enter a number: ")
b = input("Enter another number: ")

print("aType: ", type(a))
print("bType: ", type(b))

result = a + b
print("resultValue: ", result)
print("resultType: ", type(result))
```

运行结果示例：

```
Enter a number: 100↙
Enter another number: 45↙
aType:  <class 'str'>
bType:  <class 'str'>
resultValue:  10045
resultType:  <class 'str'>
```

↙表示按下回车键，按下回车键后 input() 读取就结束了。

本例中我们输入了两个整数，希望计算出它们的和，但是事与愿违，Python 只是它们当成了字符串，+起到了拼接字符串的作用，而不是求和的作用。

我们可以使用 Python 内置函数将字符串转换成想要的类型，比如：

- int(string) 将字符串转换成 int 类型；

- float(string) 将字符串转换成 float 类型；
- bool(string) 将字符串转换成 bool 类型。

修改上面的代码，将用户输入的内容转换成数字：

```
a = input("Enter a number: ")
b = input("Enter another number: ")
a = float(a)
b = int(b)
print("aType: ", type(a))
print("bType: ", type(b))

result = a + b
print("resultValue: ", result)
print("resultType: ", type(result))
```

运行结果：

```
Enter a number: 12.5↙
Enter another number: 64↙
aType:  <class 'float'>
bType:  <class 'int'>
resultValue:  76.5
resultType:  <class 'float'>
```

关于 Python 2.x

上面讲解的是 Python 3.x 中 input() 的用法，但是在较老的 Python 2.x 中情况就不一样了。Python 2.x 共提供了两个输入函数，分别是 input() 和 raw_input()：

- Python 2.x raw_input() 和 Python 3.x input() 效果是一样的，都只能以字符串的形式读取用户输入的内容。
- Python 2.x input() 看起来有点奇怪，它要求用户输入的内容必须符合 Python 的语法，稍有疏忽就会出错，通常来说只能是整数、小数、复数、字符串等。

比较强迫的是，Python 2.x input() 要求用户在输入字符串时必须使用引号包围，这有违 Python 简单易用的原则，所以 Python 3.x 取消了这种输入方式。

修改本节第一段代码，去掉 print 后面的括号：

```
a = input("Enter a number: ")
```

```
2.  b = input("Enter another number: ")
3.
4.  print "aType: ", type(a)
5.  print "bType: ", type(b)
6.
7.  result = a + b
8.  print "resultValue: ", result
9.  print "resultType: ", type(result)
```

在 Python 2.x 下运行该代码：

```
Enter a number: 45↙
Enter another number: 100↙
aType:  <type 'int'>
bType:  <type 'int'>
resultValue:  145
resultType:  <type 'int'>
```

3.12 Python print()函数高级用法

前面使用 print() 函数时，都只输出了一个变量，但实际上 print() 函数完全可以同时输出多个变量，而且它具有更多丰富的功能。

print() 函数的详细语法格式如下：

```
print (value,...,sep='',end='\n',file=sys.stdout,flush=False)
```

从上面的语法格式可以看出，value 参数可以接受任意多个变量或值，因此 print() 函数完全可以输出多个值。例如如下代码：

```
1.  user_name = 'Charlie'
2.  user_age = 8
3.  #同时输出多个变量和字符串
4.  print("读者名：",user_name,"年龄：",user_age)
```

运行上面代码，可以看到如下输出结果：

```
读者名： Charlie 年龄： 8
```

从输出结果来看，使用 print() 函数输出多个变量时，print() 函数默认以空格隔开多个变量，如果读者希望改变默认的分隔符，可通过 sep 参数进行设置。例如输出语句：

```
1.  #同时输出多个变量和字符串，指定分隔符
2.  print("读者名：" ,user_name,"年龄：",user_age,sep='|')
```

运行上面代码，可以看到如下输出结果：

```
读者名：|Charlie|年龄：|8
```

在默认情况下，print() 函数输出之后总会换行，这是因为 print() 函数的 end 参数的默认值是“\n” ，这个“\n”就代表了换行。如果希望 print() 函数输出之后不会换行，则重设 end 参数即可，例如如下代码：

```
#设置 end 参数，指定输出之后不再换行
print(40,'\t',end="")
print(50,'\t',end="")
print(60,'\t',end="")
```

上面三条 print() 语句会执行三次输出，但由于它们都指定了 end = ""，因此每条 print() 语句的输出都不会换行，依然位于同一行。运行上面代码，可以看到如下输出结果：

```
40    50    60
```

file 参数指定 print() 函数的输出目标，file 参数的默认值为 sys.stdout，该默认值代表了系统标准输出，也就

是屏幕，因此 print() 函数默认输出到屏幕。实际上，完全可以通过改变该参数让 print() 函数输出到特定文件中，例如如下代码：

```
f = open("demo.txt","w")#打开文件以便写入
print('沧海月明珠有泪',file=f)
print('蓝回日暖玉生烟',file=f)
f.close()
```

上面程序中，open() 函数用于打开 demo.txt 文件，接连 2 个 print 函数会将这 2 段字符串依次写入此文件，最后调用 close() 函数关闭文件，教程后续章节还会详细介绍关于文件操作的内容。

print() 函数的 flush 参数用于控制输出缓存，该参数一般保持为 False 即可，这样可以获得较好的性能。

3.13 Python 格式化字符串（格式化输出）

我们在《第一个 Python 程序——在屏幕上输出文本》中讲到过 print() 函数的用法，这只是最简单最初级的形式，print() 还有很多高级的玩法，比如格式化输出，这就是本节要讲解的内容。

> 熟悉 C 语言 printf() 函数的读者能够轻而易举学会 Python print() 函数，它们是非常类似的。

print() 函数使用以%开头的转换说明符对各种类型的数据进行格式化输出，具体请看下表。

表 1 Python 转换说明符

转换说明符	解释
%d、%i	转换为带符号的十进制整数
%o	转换为带符号的八进制整数
%x、%X	转换为带符号的十六进制整数
%e	转化为科学计数法表示的浮点数（e 小写）
%E	转化为科学计数法表示的浮点数（E 大写）
%f、%F	转化为十进制浮点数
%g	智能选择使用 %f 或 %e 格式
%G	智能选择使用 %F 或 %E 格式
%c	格式化字符及其 ASCII 码
%r	使用 repr() 函数将表达式转换为字符串
%s	使用 str() 函数将表达式转换为字符串

转换说明符（Conversion Specifier）只是一个占位符，它会被后面表达式（变量、常量、数字、字符串、加减乘除等各种形式）的值代替。

【实例】输出一个整数：

```
1.  age = 8
2.  print("C 语言中文网已经%d 岁了！" % age)
```

运行结果：
C 语言中文网已经 8 岁了！

在 print() 函数中，由引号包围的是格式化字符串，它相当于一个字符串模板，可以放置一些转换说明符（占位符）。本例的格式化字符串中包含一个%d 说明符，它最终会被后面的 age 变量的值所替代。

中间的%是一个分隔符，它前面是格式化字符串，后面是要输出的表达式。

当然，格式化字符串中也可以包含多个转换说明符，这个时候也得提供多个表达式，用以替换对应的转换说明符；多个表达式必须使用小括号()包围起来。请看下面的例子：

```
name = "C 语言中文网"
age = 8
url = "http://c.biancheng.net/"
print("%s 已经%d 岁了，它的网址是%s。" % (name, age, url))
```

运行结果：

```
C 语言中文网已经 8 岁了，它的网址是 http://c.biancheng.net/。
```

总之，有几个占位符，后面就得跟着几个表达式。

指定最小输出宽度

当使用表 1 中的转换说明符时，可以使用下面的格式指定最小输出宽度（至少占用多少个字符的位置）：

- %10d 表示输出的整数宽度至少为 10；
- %20s 表示输出的字符串宽度至少为 20。

请看下面的演示：

```
n = 1234567
print("n(10):%10d." % n)
print("n(5):%5d." % n)

url = "http://c.biancheng.net/python/"
print("url(35):%35s." % url)
print("url(20):%20s." % url)
```

运行结果：

```
n(10):    1234567.
n(5):1234567.
url(35):       http://c.biancheng.net/python/.
url(20):http://c.biancheng.net/python/.
```

从运行结果可以发现，对于整数和字符串，当数据的实际宽度小于指定宽度时，会在左侧以空格补齐；当数据的实际宽度大于指定宽度时，会按照数据的实际宽度输出。

你看，这里指定的只是最小宽度，当数据的实际宽度足够时，指定的宽度就没有实际意义了。

指定对齐方式

默认情况下，print() 输出的数据总是右对齐的。也就是说，当数据不够宽时，数据总是靠右边输出，而在左边补充空格以达到指定的宽度。Python 允许在最小宽度之前增加一个标志来改变对齐方式，Python 支持的标志如下：

Python 支持的标志

标志	说明
-	指定左对齐
+	表示输出的数字总要带着符号；正数带`+`，负数带`-`。
0	表示宽度不足时补充 0，而不是补充空格。

几点说明：

- 对于整数，指定左对齐时，在右边补 0 是没有效果的，因为这样会改变整数的值。
- 对于小数，以上三个标志可以同时存在。
- 对于字符串，只能使用`-`标志，因为符号对于字符串没有意义，而补 0 会改变字符串的值。

请看下面的代码：

```
n = 123456
# %09d 表示最小宽度为 9，左边补 0
print("n(09):%09d" % n)
# %+9d 表示最小宽度为 9，带上符号
print("n(+9):%+9d" % n)

f = 140.5
# %-+010f 表示最小宽度为 10，左对齐，带上符号
print("f(-+0):%-+010f" % f)

s = "Hello"
# %-10s 表示最小宽度为 10，左对齐
print("s(-10):%-10s." % s)
```

运行结果：

```
n(09):000123456

n(+9):  +123456

f(-+0):+140.500000

s(-10):Hello      .
```

指定小数精度

对于小数（浮点数），print() 还允许指定小数点后的数字位数，也即指定小数的输出精度。

精度值需要放在最小宽度之后，中间用点号.隔开；也可以不写最小宽度，只写精度。具体格式如下：

```
%m.nf
%.nf
```

m 表示最小宽度，n 表示输出精度，.是必须存在的。

请看下面的代码：

```
1.  f = 3.141592653
2.  # 最小宽度为 8，小数点后保留 3 位
3.  print("%8.3f" % f)
4.  # 最小宽度为 8，小数点后保留 3 位，左边补 0
5.  print("%08.3f" % f)
6.  # 最小宽度为 8，小数点后保留 3 位，左边补 0，带符号
7.  print("%+08.3f" % f)
```

运行结果：

```
   3.142

0003.142

+003.142
```

3.14 Python 转义字符及用法

在《Python 字符串》一节中我们曾提到过转义字符，就是那些以反斜杠\开头的字符。

ASCII 编码为每个字符都分配了唯一的编号，称为编码值。在 Python 中，一个 ASCII 字符除了可以用它的实体（也就是真正的字符）表示，还可以用它的编码值表示。这种使用编码值来间接地表示字符的方式称为转义字符（Escape Character）。

如果你对 ASCII 编码不了解，请猛击：

- ASCII 编码，将英文存储到计算机
- ASCII 码一览表，ASCII 码对照表（完整版）

转义字符以\0或者\x开头，以\0开头表示后跟八进制形式的编码值，以\x开头表示后跟十六进制形式的编码值，Python 中的转义字符只能使用八进制或者十六进制。具体格式如下：

```
\0dd
\xhh
```

dd 表示八进制数字，hh 表示十六进制数字。

ASCII 编码共收录了 128 个字符，\0 和\x 后面最多只能跟两位数字，所以八进制形式\0 并不能表示所有的 ASCII 字符，只有十六进制形式\x 才能表示所有 ASCII 字符。

我们一直在说 ASCII 编码，没有提及 Unicode、GBK、Big5 等其它编码（字符集），是因为 Python 转义字符只对 ASCII 编码（128 个字符）有效，超出范围的行为是不确定的。

字符 1、2、3、x、y、z 对应的 ASCII 码的八进制形式分别是 61、62、63、170、171、172，十六进制形式分别是 31、32、33、78、79、7A。下面的例子演示了转义字符的用法：

```
1.  str1 = "Oct: \061\062\063"
2.  str2 = "Hex: \x31\x32\x33\x78\x79\x7A"
3.  print(str1)
4.  print(str2)
```

运行结果：

```
Oct: 123
Hex: 123xyz
```

注意，使用八进制形式的转义字符没法表示 xyz，因为它们的编码值转换成八进制以后有三位。

对于 ASCII 编码，0~31（十进制）范围内的字符为控制字符，它们都是看不见的，不能在显示器上显示，甚至无法从键盘输入，只能用转义字符的形式来表示。不过，直接使用 ASCII 码记忆不方便，也不容易理解，所以，针对常用的控制字符，C 语言又定义了简写方式，完整的列表如下。

表 1 Python 支持的转义字符

转义字符	说明
\n	换行符，将光标位置移到下一行开头。
\r	回车符，将光标位置移到本行开头。
\t	水平制表符，也即 Tab 键，一般相当于四个空格。
\a	蜂鸣器响铃。注意不是喇叭发声，现在的计算机很多都不带蜂鸣器了，所以响铃不一定有效。
\b	退格（Backspace），将光标位置移到前一列。
\\	反斜线
\'	单引号
\"	双引号
\	在字符串行尾的续行符，即一行未完，转到下一行继续写。

转义字符在书写形式上由多个字符组成，但 Python 将它们看作是一个整体，表示一个字符。

Python 转义字符综合示例：

```
#使用\t 排版
str1 = '网站\t\t 域名\t\t\t 年龄\t\t 价值'
str2 = 'C 语言中文网\tc.biancheng.net\t\t8\t\t500W'
str3 = '百度\t\twww.baidu.com\t\t20\t\t500000W'
print(str1)
print(str2)
print(str3)

print("--------------------")

# \n 在输出时换行，\在书写字符串时换行
info = "Python 教程：http://c.biancheng.net/python/\n\
C++教程：http://c.biancheng.net/cplus/\n\
Linux 教程：http://c.biancheng.net/linux_tutorial/"
print(info)
```

运行结果：

```
网站          域名                    年龄      价值
C 语言中文网  c.biancheng.net       8         500W
```

```
百度          www.baidu.com        20       500000W

--------------------

Python 教程：http://c.biancheng.net/python/

C++教程：http://c.biancheng.net/cplus/

Linux 教程：http://c.biancheng.net/linux_tutorial/
```

3.15 Python 类型转换，Python 数据类型转换函数大全

虽然 Python 是弱类型编程语言，不需要像 Java 或 C 语言那样还要在使用变量前声明变量的类型，但在一些特定场景中，仍然需要用到类型转换。

比如说，我们想通过使用 print() 函数输出信息“您的身高：”以及浮点类型 height 的值，如果在交互式解释器中执行如下代码：

```
>>> height = 70.0
>>> print("您的身高"+height)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("您的身高"+height)
TypeError: must be str, not float
```

你会发现这是错误的，解释器提示我们字符串和浮点类型变量不能直接相连，需要提前将浮点类型变量 height 转换为字符串才可以。

庆幸的是，Python 已经为我们提供了多种可实现数据类型转换的函数，如表 1 所示。

表 1 常用数据类型转换函数

函 数	作 用
int(x)	将 x 转换成整数类型
float(x)	将 x 转换成浮点数类型
complex(real，[,imag])	创建一个复数
str(x)	将 x 转换为字符串
repr(x)	将 x 转换为表达式字符串
eval(str)	计算在字符串中的有效 Python 表达式，并返回一个对象
chr(x)	将整数 x 转换为一个字符
ord(x)	将一个字符 x 转换为它对应的整数值
hex(x)	将一个整数 x 转换为一个十六进制字符串
oct(x)	将一个整数 x 转换为一个八进制的字符串

需要注意的是，在使用类型转换函数时，提供给它的数据必须是有意义的。例如，int() 函数无法将一个非数字字符串转换成整数：

```
>>> int("123") #转换成功
123
>>> int("123 个") #转换失败
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int("123 个")
ValueError: invalid literal for int() with base 10: '123 个'
>>>
```

3.16 Python 算术运算符及用法详解

算术运算符也即数学运算符，用来对数字进行数学运算，比如加减乘除。下表列出了 Python 支持所有基本算术运算符。

表 1 Python 常用算术运算符

运算符	说明	实例	结果
+	加	12.45 + 15	27.45
-	减	4.56 - 0.26	4.3
*	乘	5 * 3.6	18.0
/	除法（和数学中的规则一样）	7 / 2	3.5
//	整除（只保留商的整数部分）	7 // 2	3
%	取余，即返回除法的余数	7 % 2	1
**	幂运算/次方运算，即返回 x 的 y 次方	2 ** 4	16，即 2^4

接下来将对表 1 中各个算术运算符的用法逐一讲解。

+ 加法运算符

加法运算符很简单，和数学中的规则一样，请看下面的代码：

```
m = 10
n = 97
sum1 = m + n

x = 7.2
y = 15.3
sum2 = x + y

print("sum1=%d, sum2=%.2f" % (sum1, sum2) )
```

运行结果：
sum1=107, sum2=22.50

拼接字符串

当`+`用于数字时表示加法，但是当`+`用于字符串时，它还有拼接字符串（将两个字符串连接为一个）的作用，请看代码：

```
name = "C 语言中文网"
url = "http://c.biancheng.net/"
age = 8
info = name + "的网址是" + url + "，它已经" + str(age) + "岁了。"
print(info)
```

运行结果：

```
C 语言中文网的网址是 http://c.biancheng.net/，它已经 8 岁了。
```

str() 函数用来将整数类型的 age 转换成字符串。

- 减法运算符

减法运算也和数学中的规则相同，请看代码：

```
n = 45
m = -n

x = -83.5
y = -x

print(m, ",", y)
```

运行结果：
-45 , 83.5

求负

-除了可以用作减法运算之外，还可以用作求负运算（正数变负数，负数变正数），请看下面的代码：

```
n = 45
n_neg = -n

f = -83.5
f_neg = -f

print(n_neg, ",", f_neg)
```

运行结果：
-45 , 83.5

注意，单独使用+是无效的，不会改变数字的值，例如：

```
n = 45
m = +n

x = -83.5
y = +x

print(m, ",", y)
```

运行结果：
45 , -83.5

* 乘法运算符

乘法运算也和数学中的规则相同，请看代码：

```
n = 4 * 25
f = 34.5 * 2
print(n, ",", f)
```

运行结果：
100 , 69.0

重复字符串

*除了可以用作乘法运算，还可以用来重复字符串，也即将 n 个同样的字符串连接起来，请看代码：

```
str1 = "hello "
print(str1 * 4)
```

运行结果：
hello hello hello hello

/ 和 // 除法运算符

Python 支持/和//两个除法运算符，但它们之间是有区别的：

- /表示普通除法，使用它计算出来的结果和数学中的计算结果相同。
- //表示整除，只保留结果的整数部分，舍弃小数部分；注意是直接丢掉小数部分，而不是四舍五入。

请看下面的例子：

```
#整数不能除尽
print("23/5 =", 23/5)
print("23//5 =", 23//5)
print("23.0//5 =", 23.0//5)
print("--------------------")

#整数能除尽
print("25/5 =", 25/5)
print("25//5 =", 25//5)
print("25.0//5 =", 25.0//5)
print("--------------------")

#小数除法
print("12.4/3.5 =", 12.4/3.5)
print("12.4//3.5 =", 12.4//3.5)
```

运行结果：

```
23/5 = 4.6
23//5 = 4
23.0//5 = 4.0
--------------------
25/5 = 5.0
25//5 = 5
25.0//5 = 5.0
--------------------
12.4/3.5 = 3.542857142857143
12.4//3.5 = 3.0
```

从运行结果可以发现：

- /的计算结果总是小数，不管是否能除尽，也不管参与运算的是整数还是小数。
- 当有小数参与运算时，//结果才是小数，否则就是整数。

需要注意的是，除数始终不能为 0，除以 0 是没有意义的，这将导致 ZeroDivisionError 错误。在某些编程语言中，除以 0 的结果是无穷大（包括正无穷大和负无穷大）。

Python 2.x 中的除法

Python 2.x 只提供了一种除法运算，就是/，它的行为和大部分编程语言中/的行为是一样的：

- 当/两边都是整数时，结果始终是整数；如果不能除尽，就直接舍弃小数部分。
- 当/两边有一个是小数时，结果始终是小数；如果恰好除尽，小数部分就是 0。

请看下面的代码：

```
#整数除法
print "18/6 =", 18/6
print "47/7 =", 47/7

print "----------------"

#小数除法
print "18.0/6 =", 18.0/6
print "47.0/7 =", 47.0/7
print "29.5/4.2 =", 29.5/4.2
```

运行结果：

```
18/6 = 3
47/7 = 6
----------------
18.0/6 = 3.0
47.0/7 = 6.71428571429
29.5/4.2 = 7.02380952381
```

你可以将 Python 2.x 中的/看作 Python 3.x 中/和//的结合体，因为 Python 2.x 中/的行为有点奇怪，所以 Python 3.x 增加了//运算符，用以规范除法运算的行为。

% 求余运算符

Python % 运算符用来求得两个数相除的余数，包括整数和小数。Python 使用第一个数字除以第二个数字，得到一个整数的商，剩下的值就是余数。对于小数，求余的结果一般也是小数。

注意，求余运算的本质是除法运算，所以第二个数字也不能是 0，否则会导致 ZeroDivisionError 错误。

Python % 使用示例：

```
print("-----整数求余-----")
print("15%6 =", 15%6)
print("-15%6 =", -15%6)
print("15%-6 =", 15%-6)
print("-15%-6 =", -15%-6)

print("-----小数求余-----")
print("7.7%2.2 =", 7.7%2.2)
print("-7.7%2.2 =", -7.7%2.2)
print("7.7%-2.2 =", 7.7%-2.2)
print("-7.7%-2.2 =", -7.7%-2.2)

print("---整数和小数运算---")
print("23.5%6 =", 23.5%6)
print("23%6.5 =", 23%6.5)
print("23.5%-6 =", 23.5%-6)
print("-23%6.5 =", -23%6.5)
print("-23%-6.5 =", -23%-6.5)
```

运行结果：

```
-----整数求余-----
15%6 = 3
-15%6 = 3
15%-6 = -3
-15%-6 = -3
-----小数求余-----
7.7%2.2 = 1.0999999999999996
-7.7%2.2 = 1.1000000000000005
7.7%-2.2 = -1.1000000000000005
-7.7%-2.2 = -1.0999999999999996
---整数和小数运算---
23.5%6 = 5.5
23%6.5 = 3.5
23.5%-6 = -0.5
-23%6.5 = 3.0
-23%-6.5 = -3.5
```

从运行结果可以发现两点：

- 只有当第二个数字是负数时，求余的结果才是负数。换句话说，求余结果的正负和第一个数字没有关系，只由第二个数字决定。

- %两边的数字都是整数时，求余的结果也是整数；但是只要有一个数字是小数，求余的结果就是小数。

本例中小数求余的四个结果都不精确，而是近似值，这和小数在底层的存储有关系，有兴趣的读者请猛击《小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）》了解更多。

** 次方（乘方）运算符

Python ** 运算符用来求一个 x 的 y 次方，也即次方（乘方）运算符。

由于开方是次方的逆运算，所以也可以使用 ** 运算符间接地实现开方运算。

Python ** 运算符示例：

```
print('----次方运算----')
print('3**4 =', 3**4)
print('2**5 =', 2**5)

print('----开方运算----')
print('81**(1/4) =', 81**(1/4))
print('32**(1/5) =', 32**(1/5))
```

运行结果：

```
----次方运算----
3**4 = 81
2**5 = 32
----开方运算----
81**(1/4) = 3.0
32**(1/5) = 2.0
```

3.17 Python 赋值运算符（入门必读）

赋值运算符用来把右侧的值传递给左侧的变量（或者常量）；可以直接将右侧的值交给左侧的变量，也可以进行某些运算后再交给左侧的变量，比如加减乘除、函数调用、逻辑运算等。

Python 中最基本的赋值运算符是等号=；结合其它运算符，=还能扩展出更强大的赋值运算符。

基本赋值运算符

=是 Python 中最常见、最基本的赋值运算符，用来将一个表达式的值赋给另一个变量，请看下面的例子：

```
#将字面量（直接量）赋值给变量
n1 = 100
f1 = 47.5
s1 = "http://c.biancheng.net/python/"

#将一个变量的值赋给另一个变量
n2 = n1
f2 = f1

#将某些运算的值赋给变量
sum1 = 25 + 46
sum2 = n1 % 6
s2 = str(1234)  #将数字转换成字符串
s3 = str(100) + "abc"
```

连续赋值

Python 中的赋值表达式也是有值的，它的值就是被赋的那个值，或者说是左侧变量的值；如果将赋值表达式的值再赋值给另外一个变量，这就构成了连续赋值。请看下面的例子：

```
a = b = c = 100
```

=具有右结合性，我们从右到左分析这个表达式：

- c = 100 表示将 100 赋值给 c，所以 c 的值是 100；同时，c = 100 这个子表达式的值也是 100。
- b = c = 100 表示将 c = 100 的值赋给 b，因此 b 的值也是 100。
- 以此类推，a 的值也是 100。

最终结果就是，a、b、c 三个变量的值都是 100。

= 和 ==

= 和 == 是两个不同的运算符，= 用来赋值，而 == 用来判断两边的值是否相等，千万不要混淆。

扩展后的赋值运算符

=还可与其他运算符（包括算术运算符、位运算符和逻辑运算符）相结合，扩展成为功能更加强大的赋值运算符，如表 1 所示。扩展后的赋值运算符将使得赋值表达式的书写更加优雅和方便。

表 1 Python 扩展赋值运算符

运算符	说 明	用法举例	等价形式
=	最基本的赋值运算	x = y	x = y
+=	加赋值	x += y	x = x + y
-=	减赋值	x -= y	x = x - y
*=	乘赋值	x *= y	x = x * y
/=	除赋值	x /= y	x = x / y
%=	取余数赋值	x %= y	x = x % y
**=	幂赋值	x **= y	x = x ** y
//=	取整数赋值	x //= y	x = x // y
&=	按位与赋值	x &= y	x = x & y
|=	按位或赋值	x |= y	x = x | y
^=	按位异或赋值	x ^= y	x = x ^ y
<<=	左移赋值	x <<= y	x = x << y，这里的 y 指的是左移的位数
>>=	右移赋值	x >>= y	x = x >> y，这里的 y 指的是右移的位数

这里举个简单的例子：

```
n1 = 100
f1 = 25.5

n1 -= 80  #等价于 n1=n1-80
f1 *= n1 - 10 #等价于 f1=f1*( n1 - 10 )

```

```
7.  print("n1=%d" % n1)
8.  print("f1=%.2f" % f1)
```

运行结果为：
n1=20
f1=255.00

通常情况下，只要能使用扩展后的赋值运算符，都推荐使用这种赋值运算符。

但是请注意，这种赋值运算符只能针对已经存在的变量赋值，因为赋值过程中需要变量本身参与运算，如果变量没有提前定义，它的值就是未知的，无法参与运算。例如，下面的写法就是错误的：

```
n += 10
```

该表达式等价于 n = n + 10，n 没有提前定义，所以它不能参与加法运算。

3.18 Python 位运算符详解

Python 位运算按照数据在内存中的二进制位（Bit）进行操作，它一般用于底层开发（算法设计、驱动、图像处理、单片机等），在应用层开发（Web 开发、Linux 运维等）中并不常见。想加快学习进度，或者不关注底层开发的读者可以先跳过本节，以后需要的话再来学习。

Python 位运算符只能用来操作整数类型，它按照整数在内存中的二进制形式进行计算。Python 支持的位运算符如表 1 所示。

表 1 Python 位运算符一览表

位运算符	说明	使用形式	举 例
&	按位与	a & b	4 & 5
|	按位或	a | b	4 | 5
^	按位异或	a ^ b	4 ^ 5
~	按位取反	~a	~4
<<	按位左移	a << b	4 << 2，表示整数 4 按位左移 2 位
>>	按位右移	a >> b	4 >> 2，表示整数 4 按位右移 2 位

& 按位与运算符

按位与运算符`&`的运算规则是：只有参与`&`运算的两个位都为 1 时，结果才为 1，否则为 0。例如`1&1`为 1，`0&0`为 0，`1&0`也为 0，这和逻辑运算符`&&`非常类似。

表 2 Python & 运算符的规则

第一个 Bit 位	第二个 Bit 位	结果
0	0	0
0	1	0
1	0	0
1	1	1

例如，`9&5`可以转换成如下的运算：

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
& 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
```

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0001   (1 在内存中的存储)
```

&运算符会对参与运算的两个整数的所有二进制位进行&运算，9&5 的结果为 1。

又如，-9&5 可以转换成如下的运算：

```
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
& 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
```

-9&5 的结果是 5。

> 不了解整数在内存中如何存储的读者，请猛击：整数在内存中是如何存储的，为什么它堪称天才般的设计？

再强调一遍，&运算符操作的是数据在内存中存储的原始二进制位，而不是数据本身的二进制形式；其他位运算符也一样。以-9&5 为例，-9 的在内存中的存储和 -9 的二进制形式截然不同：

```
 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
-0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (-9 的二进制形式，前面多余的 0 可以抹掉)
```

按位与运算通常用来对某些位清 0，或者保留某些位。例如要把 n 的高 16 位清 0 ，保留低 16 位，可以进行 n & 0XFFFF 运算（0XFFFF 在内存中的存储形式为 0000 0000 -- 0000 0000 -- 1111 1111 -- 1111 1111）。

使用 Python 代码对上面的分析进行验证：

```
1.  n = 0X8FA6002D
2.  print("%X" % (9&5) )
3.  print("%X" % (-9&5) )
4.  print("%X" % (n&0XFFFF) )
```

运行结果：

```
1
5
2D
```

| 按位或运算符

按位或运算符|的运算规则是：两个二进制位有一个为 1 时，结果就为 1，两个都为 0 时结果才为 0。例如 1|1 为 1，0|0 为 0，1|0 为 1，这和逻辑运算中的||非常类似。

表 3 Python | 运算符的规则

第一个 Bit 位	第二个 Bit 位	结果
0	0	0
0	1	1
1	0	1
1	1	1

例如，9 | 5 可以转换成如下的运算：

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
| 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
-------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1101   (13 在内存中的存储)
```

9 | 5 的结果为 13。

又如，-9 | 5 可以转换成如下的运算：

```
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
| 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
-------------------------------------------------------------------------------
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
```

-9 | 5 的结果是 -9。

按位或运算可以用来将某些位置 1，或者保留某些位。例如要把 n 的高 16 位置 1，保留低 16 位，可以进行 n | 0XFFFF0000 运算（0XFFFF0000 在内存中的存储形式为 1111 1111 -- 1111 1111 -- 0000 0000 -- 0000 0000）。

使用 Python 代码对上面的分析进行验证：

```
1.  n = 0X2D
```

```
print("%X" % (9|5) )
print("%X" % (-9|5) )
print("%X" % (n|0XFFFF0000) )
```

运行结果：
D
-9
FFFF002D

^按位异或运算符

按位异或运算^的运算规则是：参与运算的两个二进制位不同时，结果为 1，相同时结果为 0。例如 0^1 为 1，0^0 为 0，1^1 为 0。

表 4 Python ^ 运算符的规则

第一个 Bit 位	第二个 Bit 位	结果
0	0	0
0	1	1
1	0	1
1	1	0

例如，9 ^ 5 可以转换成如下的运算：

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
^ 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
-------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1100   (12 在内存中的存储)
```

9 ^ 5 的结果为 12。

又如，-9 ^ 5 可以转换成如下的运算：

```
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
^ 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
-------------------------------------------------------------------------
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0010   (-14 在内存中的存储)
```

`-9 ^ 5` 的结果是 -14。

按位异或运算可以用来将某些二进制位反转。例如要把 n 的高 16 位反转，保留低 16 位，可以进行 `n ^ 0XFFFF0000` 运算（0XFFFF0000 在内存中的存储形式为 1111 1111 -- 1111 1111 -- 0000 0000 -- 0000 0000）。

使用 Python 代码对上面的分析进行验证：

```
n = 0X0A07002D
print("%X" % (9^5) )
print("%X" % (-9^5) )
print("%X" % (n^0XFFFF0000) )
```

运行结果：
C
-E
F5F8002D

~按位取反运算符

按位取反运算符`~`为单目运算符（只有一个操作数），右结合性，作用是对参与运算的二进制位取反。例如`~1`为 0，`~0` 为 1，这和逻辑运算中的`!`非常类似。

例如，`~9` 可以转换为如下的运算：

```
~ 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
-------------------------------------------------------------------------
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0110   (-10 在内存中的存储)
```

所以`~9` 的结果为 -10。

例如，`~-9` 可以转换为如下的运算：

```
~ 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
-------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1000   (8 在内存中的存储)
```

所以`~-9` 的结果为 8。

使用 Python 代码对上面的分析进行验证：

```
print("%X" % (~9) )
print("%X" % (~-9) )
```

运行结果：
-A
8

<<左移运算符

Python 左移运算符`<<`用来把操作数的各个二进制位全部左移若干位，高位丢弃，低位补 0。

例如，`9<<3`可以转换为如下的运算：

```
<< 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   （9 在内存中的存储）
--------------------------------------------------------------------------------
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0100 1000   （72 在内存中的存储）
```

所以`9<<3`的结果为 72。

又如，`(-9)<<3`可以转换为如下的运算：

```
<< 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   （-9 在内存中的存储）
--------------------------------------------------------------------------------
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1011 1000   （-72 在内存中的存储）
```

所以`(-9)<<3`的结果为 -72

如果数据较小，被丢弃的高位不包含 1，那么左移 n 位相当于乘以 2 的 n 次方。

使用 Python 代码对上面的分析进行验证：

```
print("%X" % (9<<3) )
print("%X" % ((-9)<<3) )
```

运行结果：
48
-48

>>右移运算符

Python 右移运算符`>>`用来把操作数的各个二进制位全部右移若干位，低位丢弃，高位补 0 或 1。如果数据的最高位是 0，那么就补 0；如果最高位是 1，那么就补 1。

例如，`9>>3`可以转换为如下的运算：

```
>> 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001  （9 在内存中的存储）
-------------------------------------------------------------------------------
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0001  （1 在内存中的存储）
```

所以`9>>3`的结果为 1。

又如，`(-9)>>3`可以转换为如下的运算：

```
>> 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111  （-9 在内存中的存储）
-------------------------------------------------------------------------------
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 1110  （-2 在内存中的存储）
```

所以`(-9)>>3`的结果为 -2

如果被丢弃的低位不包含 1，那么右移 n 位相当于除以 2 的 n 次方（但被移除的位中经常会包含 1）。

使用 Python 代码对上面的分析进行验证：

```
1.  print("%X" % (9>>3) )
2.  print("%X" % ((-9)>>3) )
```

运行结果：
1
-2

3.19 Python 比较运算符（关系运算符）

比较运算符，也称关系运算符，用于对常量、变量或表达式的结果进行大小比较。如果这种比较是成立的，则返回 True（真），反之则返回 False（假）。

> True 和 False 都是 bool 类型，它们专门用来表示一件事情的真假，或者一个表达式是否成立，我们将在《Python bool 布尔类型》中详细讲解。

Python 支持的比较运算符如表 1 所示。

表 1 Python 比较运算符汇总

比较运算符	说明
>	大于，如果>前面的值大于后面的值，则返回 True，否则返回 False。
<	小于，如果<前面的值小于后面的值，则返回 True，否则返回 False。
==	等于，如果==两边的值相等，则返回 True，否则返回 False。
>=	大于等于（等价于数学中的 ≥），如果>=前面的值大于或者等于后面的值，则返回 True，否则返回 False。
<=	小于等于（等价于数学中的 ≤），如果<=前面的值小于或者等于后面的值，则返回 True，否则返回 False。
!=	不等于（等价于数学中的 ≠），如果!=两边的值不相等，则返回 True，否则返回 False。
is	判断两个变量所引用的对象是否相同，如果相同则返回 True，否则返回 False。
is not	判断两个变量所引用的对象是否不相同，如果不相同则返回 True，否则返回 False。

Python 比较运算符的使用举例：

```
print("89 是否大于 100：", 89 > 100)
print("24*5 是否大于等于 76：", 24*5 >= 76)
print("86.5 是否等于 86.5：", 86.5 == 86.5)
print("34 是否等于 34.0：", 34 == 34.0)
print("False 是否小于 True：", False < True)
print("True 是否等于 True：", True < True)
```

运行结果：

```
89 是否大于 100： False
24*5 是否大于等于 76： True
86.5 是否等于 86.5： True
34 是否等于 34.0： True
```

```
False 是否小于 True： True
True 是否等于 True： False
```

== 和 is 的区别

初学 Python，大家可能对 is 比较陌生，很多人会误将它和 == 的功能混为一谈，但其实 is 与 == 有本质上的区别，完全不是一码事儿。

== 用来比较两个变量的值是否相等，而 is 则用来比对两个变量引用的是否是同一个对象，例如：

```
1.  import time  #引入 time 模块
2.
3.  t1 = time.gmtime() # gmtime()用来获取当前时间
4.  t2 =  time.gmtime()
5.
6.  print(t1 == t2) #输出 True
7.  print(t1 is t2) #输出 False
```

运行结果：

```
True
False
```

time 模块的 gmtime() 方法用来获取当前的系统时间，精确到秒级，因为程序运行非常快，所以 t1 和 t1 得到的时间是一样的。== 用来判断 t1 和 t2 的值是否相等，所以返回 True。

虽然 t1 和 t2 的值相等，但它们是两个不同的对象（每次调用 gmtime() 都返回不同的对象），所以 `t1 is t2` 返回 False。这就好像两个双胞胎姐妹，虽然她们的外貌是一样的，但它们是两个人。

那么，如何判断两个对象是否相同呢？答案是判断两个对象的内存地址。如果内存地址相同，说明两个对象使用的是同一块内存，当然就是同一个对象了；这就像两个名字使用了同一个身体，当然就是同一个人了。

3.20 Python 逻辑运算符及其用法

高中数学中我们就学过逻辑运算，例如 p 为真命题，q 为假命题，那么“p 且 q”为假，“p 或 q”为真，“非 q”为真。Python 也有类似的逻辑运算，请看下表：

表 1 Python 逻辑运算符及功能

逻辑运算符	含义	基本格式	说明
and	逻辑与运算，等价于数学中的“且”	a and b	当 a 和 b 两个表达式都为真时，a and b 的结果才为真，否则为假。
or	逻辑或运算，等价于数学中的“或”	a or b	当 a 和 b 两个表达式都为假时，a or b 的结果才是假，否则为真。
not	逻辑非运算，等价于数学中的“非”	not a	如果 a 为真，那么 not a 的结果为假；如果 a 为假，那么 not a 的结果为真。相当于对 a 取反。

逻辑运算符一般和关系运算符结合使用，例如：

```
14>6 and 45.6 < 90
```

14>6 结果为 True，成立，45.6<90 结果为 False，不成立，所以整个表达式的结果为 False，也即不成立。

再看一个比较实用的例子：

```
age = int(input("请输入年龄："))
height = int(input("请输入身高："))

if age>=18 and age<=30 and height >=170 and height <= 185 :
    print("恭喜，你符合报考飞行员的条件")
else:
    print("抱歉，你不符合报考飞行员的条件")
```

可能的运行结果：

```
请输入年龄：23↙
请输入身高：178↙
恭喜，你符合报考飞行员的条件
```

打脸某些 Python 教程

有些不负责任的 Python 教程说：Python 逻辑运算符用于操作 bool 类型的表达式，执行结果也是 bool 类型，这两点其实都是错误的！

Python 逻辑运算符可以用来操作任何类型的表达式，不管表达式是不是 bool 类型；同时，逻辑运算的结果也不一定是 bool 类型，它也可以是任意类型。请看下面的例子：

```
print(100 and 200)
print(45 and 0)
print("" or "http://c.biancheng.net/python/")
print(18.5 or "http://c.biancheng.net/python/")
```

运行结果：

```
200
0
http://c.biancheng.net/python/
18.5
```

你看，本例中 and 和 or 运算符操作的都不是 bool 类型表达式，操作的结果也不是 bool 值。

逻辑运算符的本质

在 Python 中，and 和 or 不一定会计算右边表达式的值，有时候只计算左边表达式的值就能得到最终结果。

另外，and 和 or 运算符会将其中一个表达式的值作为最终结果，而不是将 True 或者 False 作为最终结果。

以上两点极其重要，了解这两点不会让你在使用逻辑运算的过程中产生疑惑。

对于 and 运算符，两边的值都为真时最终结果才为真，但是只要其中有一个值为假，那么最终结果就是假，所以 Python 按照下面的规则执行 and 运算：

- 如果左边表达式的值为假，那么就不用计算右边表达式的值了，因为不管右边表达式的值是什么，都不会影响最终结果，最终结果都是假，此时 and 会把左边表达式的值作为最终结果。
- 如果左边表达式的值为真，那么最终值是不能确定的，and 会继续计算右边表达式的值，并将右边表达式的值作为最终结果。

对于 or 运算符，情况是类似的，两边的值都为假时最终结果才为假，只要其中有一个值为真，那么最终结果就是真，所以 Python 按照下面的规则执行 or 运算：

- 如果左边表达式的值为真，那么就不用计算右边表达式的值了，因为不管右边表达式的值是什么，都不会影响最终结果，最终结果都是真，此时 or 会把左边表达式的值作为最终结果。
- 如果左边表达式的值为假，那么最终值是不能确定的，or 会继续计算右边表达式的值，并将右边表达式的值作为最终结果。

使用代码验证上面的结论：

```
1.  url = "http://c.biancheng.net/cplus/"
2.
3.  print("----False and xxx-----")
4.  print( False and print(url) )
5.  print("----True and xxx-----")
6.  print( True and print(url) )
7.  print("----False or xxx-----")
8.  print( False or print(url) )
9.  print("----True or xxx-----")
10. print( True or print(url) )
```

运行结果：

```
----False and xxx-----
False
----True and xxx-----
http://c.biancheng.net/cplus/
None
----False or xxx-----
http://c.biancheng.net/cplus/
None
----True or xxx-----
True
```

第 4 行代码中，and 左边的值为假，不需要再执行右边的表达式了，所以 print(url) 没有任何输出。

第 6 行代码中，and 左边的值为真，还需要执行右边的表达式才能得到最终的结果，所以 print(url) 输出了一个网址。

第 8、10 行代码也是类似的。

3.21 Python 三目运算符（三元运算符）用法详解

我们从一个具体的例子切入本节内容。假设现在有两个数字，我们希望获得其中较大的一个，那么可以使用 if else 语句，例如：

```
if a>b:
    max = a;
else:
    max = b;
```

但是 Python 提供了一种更加简洁的写法，如下所示：

```
max = a if a>b else b
```

这是一种类似于其它编程语言中三目运算符?:的写法。Python 是一种极简主义的编程语言，它没有引入?:这个新的运算符，而是使用已有的 if else 关键字来实现相同的功能。

使用 if else 实现三目运算符（条件运算符）的格式如下：

```
exp1 if contion else exp2
```

condition 是判断条件，exp1 和 exp2 是两个表达式。如果 condition 成立（结果为真），就执行 exp1，并把 exp1 的结果作为整个表达式的结果；如果 condition 不成立（结果为假），就执行 exp2，并把 exp2 的结果作为整个表达式的结果。

前面的语句 `max = a if a>b else b` 的含义是：

- 如果 a>b 成立，就把 a 作为整个表达式的值，并赋给变量 max；
- 如果 a> b 不成立，就把 b 作为整个表达式的值，并赋给变量 max。

三目运算符的嵌套

Python 三目运算符支持嵌套，如此可以构成更加复杂的表达式。在嵌套时需要注意 if 和 else 的配对，例如：

```
a if a>b else c if c>d else d
```

应该理解为：

```
a if a>b else ( c if c>d else d )
```

【实例】使用 Python 三目运算符判断两个数字的关系：

```
a = int( input("Input a: ") )
b = int( input("Input b: ") )
print("a 大于 b") if a>b else ( print("a 小于 b") if a<b else print("a 等于 b") )
```

可能的运行结果：

```
Input a: 45↙
Input b: 100↙
a 小于 b
```

该程序是一个嵌套的三目运算符。程序先对 a>b 求值，如果该表达式为 True，程序就返回执行第一个表达式 print("a 大于 b")，否则将继续执行 else 后面的内容，也就是：

```
( print("a 小于 b") if a<b else print("a 等于 b") )
```

进入该表达式后，先判断 a<b 是否成立，如果 a<b 的结果为 True，将执行 print("a 小于 b")，否则执行 print("a 等于 b")。

3.22 Python 运算符优先级和结合性一览表

优先级和结合性是 Python 表达式中比较重要的两个概念，它们决定了先执行表达式中的哪一部分。

Python 运算符优先级

所谓优先级，就是当多个运算符同时出现在一个表达式中时，先执行哪个运算符。

例如对于表达式 `a + b * c`，Python 会先计算乘法再计算加法；`b * c` 的结果为 8，`a + 8` 的结果为 24，所以 d 最终的值也是 24。先计算`*`再计算`+`，说明`*`的优先级高于`+`。

Python 支持几十种运算符，被划分成将近二十个优先级，有的运算符优先级不同，有的运算符优先级相同，请看下表。

表 1 Python 运算符优先级和结合性一览表

运算符说明	Python 运算符	优先级	结合性	优先级顺序
小括号	( )	19	无	高 ^ | | | | | | | | | | | | | | | | | | | | 低
索引运算符	x[i] 或 x[i1: i2 [:i3]]	18	左	
属性访问	x.attribute	17	左	
乘方	**	16	左	
按位取反	~	15	右	
符号运算符	+（正号）、-（负号）	14	右	
乘除	*、/、//、%	13	左	
加减	+、-	12	左	
位移	>>、<<	11	左	
按位与	&	10	右	
按位异或	^	9	左	
按位或	|	8	左	
比较运算符	==、!=、>、>=、<、<=	7	左	
is 运算符	is、is not	6	左	
in 运算符	in、not in	5	左	
逻辑非	not	4	右	
逻辑与	and	3	左	
逻辑或	or	2	左	
逗号运算符	exp1, exp2	1	左	

结果表 1 中的运算符优先级，我们尝试分析下面表达式的结果：

```
4+4<<2
```

+的优先级是 12，<<的优先级是 11，+的优先级高于<<，所以先执行 4+4，得到结果 8，再执行 8<<2，得到结果 32，这也是整个表达式的最终结果。

像这种不好确定优先级的表达式，我们可以给子表达式加上( )，也就是写成下面的样子：

```
(4+4) << 2
```

这样看起来就一目了然了，不容易引起误解。

当然，我们也可以使用( )改变程序的执行顺序，比如：

```
4+(4<<2)
```

则先执行 4<<2，得到结果 16，再执行 4+16，得到结果 20。

虽然 Python 运算符存在优先级的关系，但我不推荐过度依赖运算符的优先级，这会导致程序的可读性降低。因此，我建议读者：

- 不要把一个表达式写得过于复杂，如果一个表达式过于复杂，可以尝试把它拆分来书写。
- 不要过多地依赖运算符的优先级来控制表达式的执行顺序，这样可读性太差，应尽量使用( )来控制表达式的执行顺序。

Python 运算符结合性

所谓结合性，就是当一个表达式中出现多个优先级相同的运算符时，先执行哪个运算符：先执行左边的叫左结合性，先执行右边的叫右结合性。

例如对于表达式对于 100 / 25 * 16，/和*的优先级相同，应该先执行哪一个呢？这个时候就不能只依赖运算符优先级决定了，还要参考运算符的结合性。/和*都具有左结合性，因此先执行左边的除法，再执行右边的乘法，最终结果是 64。

Python 中大部分运算符都具有左结合性，也就是从左到右执行；只有单目运算符（例如 not 逻辑非运算符）、赋值运算符和三目运算符例外，它们具有右结合性，也就是从右向左执行。表 1 中列出了所有 Python 运算符的结合性。

总结

当一个表达式中出现多个运算符时，Python 会先比较各个运算符的优先级，按照优先级从高到低的顺序依次执行；当遇到优先级相同的运算符时，再根据结合性决定先执行哪个运算符：如果是左结合性就先执行左边的运算符，如果是右结合性就先执行右边的运算符。

第 4 章：列表、元组、字典和集合

4.1 什么是序列，Python 序列详解（包括序列类型和常用操作）

所谓序列，指的是一块可存放多个值的连续内存空间，这些值按一定顺序排列，可通过每个值所在位置的编号（称为索引）访问它们。

为了更形象的认识序列，可以将它看做是一家旅店，那么店中的每个房间就如同序列存储数据的一个个内存空间，每个房间所特有的房间号就相当于索引值。也就是说，通过房间号（索引）我们可以找到这家旅店（序列）中的每个房间（内存空间）。

在 Python 中，序列类型包括字符串、列表、元组、集合和字典，这些序列支持以下几种通用的操作，但比较特殊的是，集合和字典不支持索引、切片、相加和相乘操作。

> 字符串也是一种常见的序列，它也可以直接通过索引访问字符串内的字符。

序列索引

序列中，每个元素都有属于自己的编号（索引）。从起始元素开始，索引值从 0 开始递增，如图 1 所示。

图 1 序列索引值示意图

除此之外，Python 还支持索引值是负数，此类索引是从右向左计数，换句话说，从最后一个元素开始计数，从索引值 -1 开始，如图 2 所示。

图 2 负值索引示意图

> 注意，在使用负值作为列序中各元素的索引值时，是从 -1 开始，而不是从 0 开始。

无论是采用正索引值，还是负索引值，都可以访问序列中的任何元素。以字符串为例，访问“C 语言中文网”的首元素和尾元素，可以使用如下的代码：

```
str="C 语言中文网"
print(str[0],"==",str[-6])
print(str[5],"==",str[-1])
```

输出结果为：

```
C == C
网 == 网
```

序列切片

切片操作是访问序列中元素的另一种方法，它可以访问一定范围内的元素，通过切片操作，可以生成一个新的序列。

序列实现切片操作的语法格式如下：

```
sname[start : end : step]
```

其中，各个参数的含义分别是：

- sname：表示序列的名称；
- start：表示切片的开始索引位置（包括该位置），此参数也可以不指定，会默认为 0，也就是从序列的开头进行切片；
- end：表示切片的结束索引位置（不包括该位置），如果不指定，则默认为序列的长度；
- step：表示在切片过程中，隔几个存储位置（包含当前位置）取一次元素，也就是说，如果 step 的值大于 1，则在进行切片去序列元素时，会“跳跃式”的取元素。如果省略设置 step 的值，则最后一个冒号就可以省略。

例如，对字符串“C 语言中文网”进行切片：

```
str="C 语言中文网"
#取索引区间为[0,2]之间（不包括索引 2 处的字符）的字符串
print(str[:2])
#隔 1 个字符取一个字符，区间是整个字符串
print(str[::2])
#取整个字符串，此时 [] 中只需一个冒号即可
print(str[:])
```

运行结果为：

```
C 语
C 言文
C 语言中文网
```

序列相加

Python 中，支持两种类型相同的序列使用“+”运算符做相加操作，它会将两个序列进行连接，但不会去除重复的元素。

这里所说的“类型相同”，指的是“+”运算符的两侧序列要么都是列表类型，要么都是元组类型，要么都是字符串。

例如，前面章节中我们已经实现用“+”运算符连接 2 个（甚至多个）字符串，如下所示：

```
1. str="c.biancheng.net"
2. print("C 语言"+"中文网:"+str)
```

输出结果为：

```
C 语言中文网：c.biancheng.net
```

序列相乘

Python 中，使用数字 n 乘以一个序列会生成新的序列，其内容为原来序列被重复 n 次的结果。例如：

```
1. str="C 语言中文网"
2. print(str*3)
```

输出结果为：

```
'C 语言中文网 C 语言中文网 C 语言中文网'
```

比较特殊的是，列表类型在进行乘法运算时，还可以实现初始化指定长度列表的功能。例如如下的代码，将创建一个长度为 5 的列表，列表中的每个元素都是 None，表示什么都没有。

```
1. #列表的创建用 []，后续讲解列表时会详细介绍
2. list = [None]*5
3. print(list)
```

输出结果为：

```
[None, None, None, None, None]
```

检查元素是否包含在序列中

Python 中，可以使用 in 关键字检查某元素是否为序列的成员，其语法格式为：

```
value in sequence
```

其中，value 表示要检查的元素，sequence 表示指定的序列。

例如，检查字符‘c’是否包含在字符串“c.biancheng.net”中，可以执行如下代码：

```
1.  str="c.biancheng.net"
2.  print('c'in str)
```

运行结果为：

```
True
```

和 in 关键字用法相同，但功能恰好相反的，还有 not in 关键字，它用来检查某个元素是否不包含在指定的序列中，比如说：

```
1.  str="c.biancheng.net"
2.  print('c' not in str)
```

输出结果为：

```
False
```

和序列相关的内置函数

Python 提供了几个内置函数（表 3 所示），可用于实现与序列相关的一些常用操作。

表 3 序列相关的内置函数

函数	功能
len()	计算序列的长度，即返回序列中包含多少个元素。
max()	找出序列中的最大元素。注意，对序列使用 sum() 函数时，做加和操作的必须都是数字，不能是字符或字符串，否则该函数将抛出异常，因为解释器无法判定是要做连接操作（ + 运算符可以连

	接两个序列），还是做加和操作。
min()	找出序列中的最小元素。
list()	将序列转换为列表。
str()	将序列转换为字符串。
sum()	计算元素和。
sorted()	对元素进行排序。
reversed()	反向序列中的元素。
enumerate()	将序列组合为一个索引序列，多用在 for 循环中。

这里给大家给几个例子：

```
str="c.biancheng.net"
#找出最大的字符
print(max(str))
#找出最小的字符
print(min(str))
#对字符串中的元素进行排序
print(sorted(str))
```

输出结果为：

```
t
.
['.', '.', 'a', 'b', 'c', 'c', 'e', 'e', 'g', 'h', 'i', 'n', 'n', 'n', 't']
```

4.2 Python list 列表详解

在实际开发中，经常需要将一组（不只一个）数据存储起来，以便后边的代码使用。说到这里，一些读者可能听说过数组（Array），它就可以把多个数据挨个存储到一起，通过数组下标可以访问数组中的每个元素。

需要明确的是，Python 中没有数组，但是加入了更加强大的列表。如果把数组看做是一个集装箱，那么 Python 的列表就是一个工厂的仓库。

> 大部分编程语言都支持数组，比如 C 语言、C++、Java、PHP、JavaScript 等。

从形式上看，列表会将所有元素都放在一对中括号[ ]里面，相邻元素之间用逗号,分隔，如下所示：

```
[element1, element2, element3, ..., elementn]
```

格式中，element1 ~ elementn 表示列表中的元素，个数没有限制，只要是 Python 支持的数据类型就可以。

从内容上看，列表可以存储整数、小数、字符串、列表、元组等任何类型的数据，并且同一个列表中元素的类型也可以不同。比如说：

```
["http://c.biancheng.net/python/", 1, [2,3,4] , 3.0]
```

可以看到，列表中同时包含字符串、整数、列表、浮点数这些数据类型。

> 注意，在使用列表时，虽然可以将不同类型的数据放入到同一个列表中，但通常情况下不这么做，同一列表中只放入同一类型的数据，这样可以提高程序的可读性。

另外，在其它 Python 教程中，经常用 list 代指列表，这是因为列表的数据类型就是 list，通过 type() 函数就可以知道，例如：

```
>>> type( ["http://c.biancheng.net/python/", 1, [2,3,4] , 3.0] )
<class 'list'>
```

可以看到，它的数据类型为 list，就表示它是一个列表。

Python 创建列表

在 Python 中，创建列表的方法可分为两种，下面分别进行介绍。

1) 使用 [ ] 直接创建列表

使用[ ]创建列表后，一般使用=将它赋值给某个变量，具体格式如下：

```
listname = [element1 , element2 , element3 , ... , elementn]
```

其中，listname 表示变量名，element1 ~ elementn 表示列表元素。

例如，下面定义的列表都是合法的：

```
num = [1, 2, 3, 4, 5, 6, 7]
name = ["C 语言中文网", "http://c.biancheng.net"]
program = ["C 语言", "Python", "Java"]
```

另外，使用此方式创建列表时，列表中元素可以有多个，也可以一个都没有，例如：

```
emptylist = [ ]
```

这表明，emptylist 是一个空列表。

2) 使用 list() 函数创建列表

除了使用[ ]创建列表外，Python 还提供了一个内置的函数 list()，使用它可以将其它数据类型转换为列表类型。例如：

```
#将字符串转换成列表
list1 = list("hello")
print(list1)

#将元组转换成列表
tuple1 = ('Python', 'Java', 'C++', 'JavaScript')
list2 = list(tuple1)
print(list2)

#将字典转换成列表
dict1 = {'a':100, 'b':42, 'c':9}
list3 = list(dict1)
print(list3)

#将区间转换成列表
range1 = range(1, 6)
list4 = list(range1)
print(list4)

```

```
#创建空列表
print(list())
```

运行结果：

```
['h', 'e', 'l', 'l', 'o']
['Python', 'Java', 'C++', 'JavaScript']
['a', 'b', 'c']
[1, 2, 3, 4, 5]
[]
```

访问列表元素

列表是 Python 序列的一种，我们可以使用索引（Index）访问列表中的某个元素（得到的是一个元素的值），也可以使用切片访问列表中的一组元素（得到的是一个新的子列表）。

使用索引访问列表元素的格式为：

```
listname[i]
```

其中，listname 表示列表名字，i 表示索引值。列表的索引可以是正数，也可以是负数。

使用切片访问列表元素的格式为：

```
listname[start : end : step]
```

其中，listname 表示列表名字，start 表示起始索引，end 表示结束索引，step 表示步长。

以上两种方式我们已在《Python 序列》中进行了讲解，这里就不再赘述了，仅作示例演示，请看下面代码：

```
url = list("http://c.biancheng.net/shell/")

#使用索引访问列表中的某个元素
print(url[3])  #使用正数索引
print(url[-4])  #使用负数索引

#使用切片访问列表中的一组元素
print(url[9: 18])  #使用正数切片
print(url[9: 18: 3])  #指定步长
print(url[-6: -1])  #使用负数切片
```

运行结果：

```
p
e
['b', 'i', 'a', 'n', 'c', 'h', 'e', 'n', 'g']
['b', 'n', 'e']
['s', 'h', 'e', 'l', 'l']
```

Python 删除列表

对于已经创建的列表，如果不再使用，可以使用 del 关键字将其删除。

实际开发中并不经常使用 del 来删除列表，因为 Python 自带的垃圾回收机制会自动销毁无用的列表，即使开发者不手动删除，Python 也会自动将其回收。

del 关键字的语法格式为：

```
del listname
```

其中，listname 表示要删除列表的名称。

Python 删除列表实例演示：

```
1. intlist = [1, 45, 8, 34]
2. print(intlist)
3. del intlist
4. print(intlist)
```

运行结果：

```
[1, 45, 8, 34]
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 4, in <module>
    print(intlist)
NameError: name 'intlist' is not defined
```

4.3 Python list 列表添加元素的 3 种方法

实际开发中，经常需要对 Python 列表进行更新，包括向列表中添加元素、修改表中元素以及删除元素。本节先来学习如何向列表中添加元素。

《Python 序列》一节告诉我们，使用+运算符可以将多个序列连接起来；列表是序列的一种，所以也可以使用+进行连接，这样就相当于在第一个列表的末尾添加了另一个列表。

请看下面的演示：

```
language = ["Python", "C++", "Java"]
birthday = [1991, 1998, 1995]
info = language + birthday

print("language =", language)
print("birthday =", birthday)
print("info =", info)
```

运行结果：

```
language = ['Python', 'C++', 'Java']
birthday = [1991, 1998, 1995]
info = ['Python', 'C++', 'Java', 1991, 1998, 1995]
```

从运行结果可以发现，使用+会生成一个新的列表，原有的列表不会被改变。

+更多的是用来拼接列表，而且执行效率并不高，如果想在列表中插入元素，应该使用下面几个专门的方法。

Python append()方法添加元素

append() 方法用于在列表的末尾追加元素，该方法的语法格式如下：

```
listname.append(obj)
```

其中，listname 表示要添加元素的列表；obj 表示到添加到列表末尾的数据，它可以是单个元素，也可以是列表、元组等。

请看下面的演示：

```
l = ['Python', 'C++', 'Java']
#追加元素
l.append('PHP')
```

```
print(l)

#追加元组，整个元组被当成一个元素
t = ('JavaScript', 'C#', 'Go')
l.append(t)
print(l)

#追加列表，整个列表也被当成一个元素
l.append(['Ruby', 'SQL'])
print(l)
```

运行结果为：

```
['Python', 'C++', 'Java', 'PHP']
['Python', 'C++', 'Java', 'PHP', ('JavaScript', 'C#', 'Go')]
['Python', 'C++', 'Java', 'PHP', ('JavaScript', 'C#', 'Go'), ['Ruby', 'SQL']]
```

可以看到，当给 append() 方法传递列表或者元组时，此方法会将它们视为一个整体，作为一个元素添加到列表中，从而形成包含列表和元组的新列表。

Python extend()方法添加元素

extend() 和 append() 的不同之处在于：extend() 不会把列表或者元祖视为一个整体，而是把它们包含的元素逐个添加到列表中。

extend() 方法的语法格式如下：

```
listname.extend(obj)
```

其中，listname 指的是要添加元素的列表；obj 表示到添加到列表末尾的数据，它可以是单个元素，也可以是列表、元组等，但不能是单个的数字。

请看下面的演示：

```
l = ['Python', 'C++', 'Java']
#追加元素
l.extend('C')
print(l)

#追加元组，元祖被拆分成多个元素
```

```
t = ('JavaScript', 'C#', 'Go')
l.extend(t)
print(l)

#追加列表，列表也被拆分成多个元素
l.extend(['Ruby', 'SQL'])
print(l)
```

运行结果：

```
['Python', 'C++', 'Java', 'C']
['Python', 'C++', 'Java', 'C', 'JavaScript', 'C#', 'Go']
['Python', 'C++', 'Java', 'C', 'JavaScript', 'C#', 'Go', 'Ruby', 'SQL']
```

Python insert()方法插入元素

append() 和 extend() 方法只能在列表末尾插入元素，如果希望在列表中间某个位置插入元素，那么可以使用 insert() 方法。

insert() 的语法格式如下：

```
listname.insert(index , obj)
```

其中，index 表示指定位置的索引值。insert() 会将 obj 插入到 listname 列表第 index 个元素的位置。

当插入列表或者元祖时，insert() 也会将它们视为一个整体，作为一个元素插入到列表中，这一点和 append() 是一样的。

请看下面的演示代码：

```
l = ['Python', 'C++', 'Java']
#插入元素
l.insert(1, 'C')
print(l)

#插入元组，整个元祖被当成一个元素
t = ('C#', 'Go')
l.insert(2, t)
print(l)
```

```
10.
11. #插入列表，整个列表被当成一个元素
12. l.insert(3, ['Ruby', 'SQL'])
13. print(l)
14.
15. #插入字符串，整个字符串被当成一个元素
16. l.insert(0, "http://c.biancheng.net")
17. print(l)
```

输出结果为：

```
['Python', 'C', 'C++', 'Java']
['Python', 'C', ('C#', 'Go'), 'C++', 'Java']
['Python', 'C', ('C#', 'Go'), ['Ruby', 'SQL'], 'C++', 'Java']
['http://c.biancheng.net', 'Python', 'C', ('C#', 'Go'), ['Ruby', 'SQL'], 'C++', 'Java']
```

提示，insert() 主要用来在列表的中间位置插入元素，如果你仅仅希望在列表的末尾追加元素，那我更建议使用 append() 和 extend()。

4.4 Python list 列表删除元素（4 种方法）

在 Python 列表中删除元素主要分为以下 3 种场景：

- 根据目标元素所在位置的索引进行删除，可以使用 del 关键字或者 pop() 方法；
- 根据元素本身的值进行删除，可使用列表（list 类型）提供的 remove() 方法；
- 将列表中所有元素全部删除，可使用列表（list 类型）提供的 clear() 方法。

del：根据索引值删除元素

del 是 Python 中的关键字，专门用来执行删除操作，它不仅可以删除整个列表，还可以删除列表中的某些元素。我们已经在《Python 列表》中讲解了如何删除整个列表，所以本节只讲解如何删除列表元素。

del 可以删除列表中的单个元素，格式为：

```
del listname[index]
```

其中，listname 表示列表名称，index 表示元素的索引值。

del 也可以删除中间一段连续的元素，格式为：

```
del listname[start : end]
```

其中，start 表示起始索引，end 表示结束索引。del 会删除从索引 start 到 end 之间的元素，不包括 end 位置的元素。

【示例】使用 del 删除单个列表元素：

```
lang = ["Python", "C++", "Java", "PHP", "Ruby", "MATLAB"]

#使用正数索引
del lang[2]
print(lang)

#使用负数索引
del lang[-2]
print(lang)
```

运行结果：
['Python', 'C++', 'PHP', 'Ruby', 'MATLAB']
['Python', 'C++', 'PHP', 'MATLAB']

【示例】使用 del 删除一段连续的元素：

```
lang = ["Python", "C++", "Java", "PHP", "Ruby", "MATLAB"]

del lang[1: 4]
print(lang)

lang.extend(["SQL", "C#", "Go"])
del lang[-5: -2]
print(lang)
```

运行结果：
['Python', 'Ruby', 'MATLAB']
['Python', 'C#', 'Go']

pop()：根据索引值删除元素

Python pop() 方法用来删除列表中指定索引处的元素，具体格式如下：

```
listname.pop(index)
```

其中，listname 表示列表名称，index 表示索引值。如果不写 index 参数，默认会删除列表中的最后一个元素，类似于数据结构中的“出栈”操作。

pop() 用法举例：

```
nums = [40, 36, 89, 2, 36, 100, 7]
nums.pop(3)
print(nums)
nums.pop()
print(nums)
```

运行结果：
[40, 36, 89, 36, 100, 7]
[40, 36, 89, 36, 100]

大部分编程语言都会提供和 pop() 相对应的方法，就是 push()，该方法用来将元素添加到列表的尾部，类似于数据结构中的“入栈”操作。但是 Python 是个例外，Python 并没有提供 push() 方法，因为完全可以使用 append() 来代替 push() 的功能。

remove()：根据元素值进行删除

除了 del 关键字，Python 还提供了 remove() 方法，该方法会根据元素本身的值来进行删除操作。需要注意的是，remove() 方法只会删除第一个和指定值相同的元素，而且必须保证该元素是存在的，否则会引发 ValueError 错误。

remove() 方法使用示例：

```
nums = [40, 36, 89, 2, 36, 100, 7]
#第一次删除 36
nums.remove(36)
print(nums)
#第二次删除 36
nums.remove(36)
print(nums)
#删除 78
nums.remove(78)
print(nums)
```

运行结果：

```
[40, 89, 2, 36, 100, 7]
[40, 89, 2, 100, 7]
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 9, in <module>
    nums.remove(78)
ValueError: list.remove(x): x not in list
```

最后一次删除，因为 78 不存在导致报错，所以我们在使用 remove() 删除元素时最好提前判断一下。

clear()：删除列表所有元素

Python clear() 用来删除列表的所有元素，也即清空列表，请看下面的代码：

```
url = list("http://c.biancheng.net/python/")
url.clear()
print(url)
```

运行结果：

```
[]
```

4.5 Python list 列表修改元素

Python 提供了两种修改列表（list）元素的方法，你可以每次修改单个元素，也可以每次修改一组元素（多个）。

修改单个元素

修改单个元素非常简单，直接对元素赋值即可。请看下面的例子：

```
nums = [40, 36, 89, 2, 36, 100, 7]
nums[2] = -26  #使用正数索引
nums[-3] = -66.2  #使用负数索引
print(nums)
```

运行结果：

```
[40, 36, -26, 2, -66.2, 100, 7]
```

使用索引得到列表元素后，通过=赋值就改变了元素的值。

修改一组元素

Python 支持通过切片语法给一组元素赋值。在进行这种操作时，如果不指定步长（step 参数），Python 就不要求新赋值的元素个数与原来的元素个数相同；这意味，该操作既可以为列表添加元素，也可以为列表删除元素。

下面的代码演示了如何修改一组元素的值：

```
nums = [40, 36, 89, 2, 36, 100, 7]

#修改第 1~4 个元素的值（不包括第 4 个元素）
nums[1: 4] = [45.25, -77, -52.5]
print(nums)
```

运行结果：
[40, 45.25, -77, -52.5, 36, 100, 7]

如果对空切片（slice）赋值，就相当于插入一组新的元素：

```
nums = [40, 36, 89, 2, 36, 100, 7]
#在 4 个位置插入元素
```

```
3.  nums[4: 4] = [-77, -52.5, 999]
4.  print(nums)
```

运行结果：
[40, 36, 89, 2, -77, -52.5, 999, 36, 100, 7]

使用切片语法赋值时，Python 不支持单个值，例如下面的写法就是错误的：

```
nums[4: 4] = -77
```

但是如果使用字符串赋值，Python 会自动把字符串转换成序列，其中的每个字符都是一个元素，请看下面的代码：

```
s = list("Hello")
s[2:4] = "XYZ"
print(s)
```

运行结果：
['H', 'e', 'X', 'Y', 'Z', 'o']

使用切片语法时也可以指定步长（step 参数），但这个时候就要求所赋值的新元素的个数与原有元素的个数相同，例如：

```
1.  nums = [40, 36, 89, 2, 36, 100, 7]
2.  #步长为 2，为第 1、3、5 个元素赋值
3.  nums[1: 6: 2] = [0.025, -99, 20.5]
4.  print(nums)
```

运行结果：
[40, 0.025, 89, -99, 36, 20.5, 7]

4.6 Python list 列表查找元素

Python 列表（list）提供了 index() 和 count() 方法，它们都可以用来查找元素。

index() 方法

index() 方法用来查找某个元素在列表中出现的位置（也就是索引），如果该元素不存在，则会导致 ValueError 错误，所以在查找之前最好使用 count() 方法判断一下。

index() 的语法格式为：

```
listname.index(obj, start, end)
```

其中，listname 表示列表名称，obj 表示要查找的元素，start 表示起始位置，end 表示结束位置。

start 和 end 参数用来指定检索范围：

- start 和 end 可以都不写，此时会检索整个列表；
- 如果只写 start 不写 end，那么表示检索从 start 到末尾的元素；
- 如果 start 和 end 都写，那么表示检索 start 和 end 之间的元素。

index() 方法会返回元素所在列表中的索引值。

index() 方法使用举例：

```
1.  nums = [40, 36, 89, 2, 36, 100, 7, -20.5, -999]
2.  #检索列表中的所有元素
3.  print( nums.index(2) )
4.  #检索 3~7 之间的元素
5.  print( nums.index(100, 3, 7) )
6.  #检索 4 之后的元素
7.  print( nums.index(7, 4) )
8.  #检索一个不存在的元素
9.  print( nums.index(55) )
```

运行结果：

```
3
5
6
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 9, in <module>
```

```
    print( nums.index(55) )
ValueError: 55 is not in list
```

count()方法

count() 方法用来统计某个元素在列表中出现的次数，基本语法格式为：

```
listname.count(obj)
```

其中，listname 代表列表名，obj 表示要统计的元素。

如果 count() 返回 0，就表示列表中不存在该元素，所以 count() 也可以用来判断列表中的某个元素是否存在。

count() 用法示例：

```
1.  nums = [40, 36, 89, 2, 36, 100, 7, -20.5, 36]
2.  #统计元素出现的次数
3.  print("36 出现了%d 次" % nums.count(36))
4.  #判断一个元素是否存在
5.  if nums.count(100):
6.      print("列表中存在 100 这个元素")
7.  else:
8.      print("列表中不存在 100 这个元素")
```

运行结果：

```
36 出现了 3 次
列表中存在 100 这个元素
```

4.7 Python list 列表使用技巧及注意事项

前面章节介绍了很多关于 list 列表的操作函数，细心的读者可能会发现，有很多操作函数的功能非常相似。例如，增加元素功能的函数有 append() 和 extend()，删除元素功能的有 clear()、 remove()、pop() 和 del 关键字。

本节将通过实例演示的方式来明确各个函数的用法，以及某些函数之间的区别和在使用时的一些注意事项。

Python list 添加元素的方法及区别

定义两个列表（分别是 list1 和 list3），并分别使用 +、extend()、append() 对这两个 list 进行操作，其操作的结果赋值给 l2。实例代码如下：

```
1.  tt = 'hello'
2.  #定义一个包含多个类型的 list
3.  list1 = [1,4,tt,3.4,"yes",[1,2]]
4.  print(list1,id(list1))
5.
6.  print("1.-----------------")
7.
8.  #比较 list 中添加元素的几种方法的用法和区别
9.  list3 = [6,7]
10. l2 = list1 + list3
11. print(l2,id(l2))
12.
13. print("2.-----------------")
14.
15. l2 = list1.extend(list3)
16. print(l2,id(l2))
17. print(list1,id(list1))
18.
19. print("3.-----------------")
20.
21. l2 = list1.append(list3)
22. print(l2,id(l2))
23. print(list1,id(list1))
```

输出结果为：

```
[1, 4, 'hello', 3.4, 'yes', [1, 2]] 2251638471496
1.----------------
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7] 2251645237064
2.----------------
None 1792287952
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7] 2251638471496
3.----------------
None 1792287952
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7, [6, 7]] 2251638471496
```

根据输出结果，可以分析出以下几个结论：

1. 使用“+”号连接的列表，是将 list3 中的元素放在 list 的后面得到的 l2。并且 l2 的内存地址值与 list1 并不一样，这表明 l2 是一个重新生成的列表。
2. 使用 extend 处理后得到的 l2 是 none。表明 extend 没有返回值，并不能使用链式表达式。即 extend 千万不能放在等式的右侧，这是编程时常犯的错误，一定要引起注意。
3. extend 处理之后， list1 的内容与使用“+”号生成的 l2 是一样的。但 list1 的地址在操作前后并没有变化，这表明 extend 的处理仅仅是改变了 list1，而没有重新创建一个 list。从这个角度来看，extend 的效率要高于“+”号。
4. 从 append 的结果可以看出，append 的作用是将 list3 整体当成一个元素追加到 list1 后面，这与 extend 和“+”号的功能完全不同，这一点也需要注意。

Python list 删除操作

接下来演示有关 del 的基本用法，实例代码如下：

```
tt = 'hello'
#定义一个包含多个类型的 list
list1 = [1,4,tt,3.4,"yes",[1,2]]
print(list1)
del list1[2:5]
print(list1)
del list1[2]
print(list1)
```

输出结果为：

```
[1, 4, 'hello', 3.4, 'yes', [1, 2]]
[1, 4, [1, 2]]
[1, 4]
```

这 3 行输出分别是 list1 的原始内容、删除一部分切片内容、删除指定索引内容。可以看到，del 关键字按照指定的位置删掉了指定的内容。

需要注意的是，在使用 del 关键字时，一定要搞清楚，删除的到底是变量还是数据。例如，下面代码演示和删除变量的方法：

```
tt = 'hello'
#定义一个包含多个类型的 list
list1 = [1,4,tt,3.4,"yes",[1,2]]
l2 = list1
print(id(l2),id(list1))
del list1
print(l2)
print(list1)
```

运行结果如下：

```
1765451922248 1765451922248
[1, 4, 'hello', 3.4, 'yes', [1, 2]]
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 8, in <module>
    print(list1)
NameError: name 'list1' is not defined
```

第一行输出的内容是 l2 和 list1 的地址，可以看到它们是相同的，说明 l2 和 list1 之间的赋值仅仅是传递内存地址。接下来将 list1 删掉，并打印 l2，可以看到，l2 所指向的内存数据还是存在的，这表明 del 删除 list1 时仅仅是销毁了变量 list1，并没有删除指定的数据。

除了删除变量，其他的删除都是删除数据，比如将列表中数据全部清空，实现代码如下：

```
tt = 'hello'
#定义一个包含多个类型的 list
list1 = [1,4,tt,3.4,"yes",[1,2]]
l2 = list1
l3 = l2
del l2[:]
print(l2)
print(l3)
```

输出结果为：

```
[]
[]
```

可以看到，l3 和 l2 执行同样的内存地址，当 l2 被清空之后，l3 的内容也被清空了。这表明内存中的数据真正改变了。

另外，在实际过程中，即便使用 del 关键字删除了指定变量，且该变量所占用的内存再没有其他变量使用，此内存空间也不会真正地被系统回收并进行二次使用，它只是会被标记为无效内存。

如果想让系统回收这些可用的内存，需要借助 gc 库中的 collect() 函数。例如：

```
#引入 gc 库
import gc
tt = 'hello'
#定义一个包含多个类型的 list
list1 = [1,4,tt,3.4,"yes",[1,2]]
del list1
#回收内存地址
gc.collect()
```

前面我们在《Python 缓存机制》一节讲过，系统为了提升性能，会将一部分变量驻留在内存中。这个机制对于，多线程并发时程序产生大量占用内存的变量无法得到释放，或者某些不再需要使用的全局变量占用着大的内存，导致后续运行中出现内存不足的情况，此时就可以使用 del 关键字来回收内存，使系统的性能得以提升。同时，它可以为团队省去扩充大量内存的成本。

4.8 Python range()快速初始化数字列表

注意：本节需具备最基本的 Python 循环结构的基础，初学者可先跳过本节。

实际场景中，经常需要存储一组数字。例如在游戏中，需要跟踪每个角色的位置，还可能需要跟踪玩家的几个最高得分。在数据可视化中，处理的几乎都是由数字（如温度、距离、人口数量、经度和纬度等）组成的集合。

列表非常适合用于存储数字集合，并且 Python 提供了 range() 函数，可帮助我们高效地处理数字列表，即便列表需要包含数百万个元素，也可以快速实现。

Python range() 函数能够轻松地生成一系列的数字。例如，可以像如下这样使用 range() 来打印一系列数字：

```
1.  for value in range(1,5):
2.      print(value)
```

输出结果为：

```
1
2
3
4
```

注意，在这个示例程序中，range() 只是打印数字 1~4，因为 range() 函数的用法是：让 Python 从指定的第一个值开始，一直数到指定的第二个值停止，但不包含第二个值（这里为 5）。

因此，如果想要上面程序打印数字 1~5，需要使用 range(1,6)。

另外需要指明的是，range() 函数的返回值并不直接是列表类型（list），例如：

```
>>> type([1,2,3,4,5])
<class 'list'>
>>> type(range(1,6))
<class 'range'>
```

可以看到，range() 函数的返回值类型为 range，而不是 list。而如果想要得到 range() 函数创建的数字列表，还需要借助 list() 函数，比如：

```
>>> list(range(1,6))
[1, 2, 3, 4, 5]
```

可以看到，如果将 range() 作为 list() 的参数，其输出就是一个数字列表。

不仅如此，在使用 range() 函数时，还可以指定步长。例如，下面的代码打印 1~10 内的偶数：

```
1.  even_numbers = list(range(2,11,2))
2.  print(even_numbers)
```

在这个示例中，函数 range() 从 2 开始数，然后不断地加 2，直到达到或超过终值，因此输出如下：

```
[2, 4, 6, 8, 10]
```

注意，即便 range() 第二个参数恰好符合条件，最终创建的数字列表中也不会包含它。

实际使用时，range() 函数常常和 Python 循环结构、推导式（后续会讲，这里先不涉及）一起使用，几乎能够创建任何需要的数字列表。

例如，创建这样一个列表，其中包含前 10 个整数（即 1~10）的平方，实现代码如下：

```
squares = []
for value in range(1,11):
    square = value**2
    squares.append(square)
print(squares)
```

运行结果为：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

4.9 Python list 列表实现栈和队列

队列和栈是两种数据结构，其内部都是按照固定顺序来存放变量的，二者的区别在于对数据的存取顺序：

- 队列是，先存入的数据最先取出，即“先进先出”。
- 栈是，最后存入的数据最先取出，即“后进先出”。

考虑到 list 类型数据本身的存放就是有顺序的，而且内部元素又可以是各不相同的类型，非常适合用于队列和栈的实现。本节将演示如何使用 list 类型变量来实现队列和栈。

Python list 实现队列

使用 list 列表模拟队列功能的实现方法是，定义一个 list 变量，存入数据时使用 insert() 方法，设置其第一个参数为 0，即表示每次都从最前面插入数据；读取数据时，使用 pop() 方法，即将队列的最后一个元素弹出。

如此 list 列表中数据的存取顺序就符合“先进先出”的特点。实现代码如下：

```
#定义一个空列表，当做队列
queue = []
#向列表中插入元素
queue.insert(0,1)
queue.insert(0,2)
queue.insert(0,"hello")
print(queue)
print("取一个元素：",queue.pop())
print("取一个元素：",queue.pop())
print("取一个元素：",queue.pop())
```

运行结果为：

```
['hello', 2, 1]
取一个元素： 1
取一个元素： 2
取一个元素： hello
```

Python list 实现栈

使用 list 列表模拟栈功能的实现方法是，使用 append() 方法存入数据；使用 pop() 方法读取数据。

append() 方法向 list 中存入数据时，每次都在最后面添加数据，这和前面程序中的 insert() 方法正好相反。
举个例子：

```
#定义一个空 list 当做栈
stack = []
stack.append(1)
stack.append(2)
stack.append("hello")
print(stack)
print("取一个元素：",stack.pop())
print("取一个元素：",stack.pop())
print("取一个元素：",stack.pop())
```

输出结果为：

```
[1, 2, 'hello']
取一个元素： hello
取一个元素： 2
取一个元素： 1
```

collections 模块实现栈和队列

前面使用 list 实现队列的例子中，插入数据的部分是通过 insert() 方法实现的，这种方法效率并不高，因为每次从列表的开头插入一个数据，列表中所有元素都得向后移动一个位置。

这里介绍一个相对更高效的方法，即使用标准库的 collections 模块中的 deque 结构体，它被设计成在两端存入和读取都很快的特殊 list，可以用来实现栈和队列的功能。

举个例子：


```
queueAndStack = deque()
queueAndStack.append(1)
queueAndStack.append(2)
queueAndStack.append("hello")
print(list(queueAndStack))

#实现队列功能，从队列中取一个元素，根据先进先出原则，这里应输出 1
print(queueAndStack.popleft())
```

```
#实现栈功能，从栈里取一个元素，根据后进先出原则，这里应输出 hello
print(queueAndStack.pop())
#再次打印列表
print(list(queueAndStack))
```

输出结果为：

```
[1, 2, 'hello']
1
hello
[2]
```

4.10 Python tuple 元组详解

元组（tuple）是 Python 中另一个重要的序列结构，和列表类似，元组也是由一系列按特定顺序排序的元素组成。

元组和列表（list）的不同之处在于：

- 列表的元素是可以更改的，包括修改元素值，删除和插入元素，所以列表是可变序列；
- 而元组一旦被创建，它的元素就不可更改了，所以元组是不可变序列。

元组也可以看做是不可变的列表，通常情况下，元组用于保存无需修改的内容。

从形式上看，元组的所有元素都放在一对小括号( )中，相邻元素之间用逗号,分隔，如下所示：

```
(element1, element2, ... , elementn)
```

其中 element1~elementn 表示元组中的各个元素，个数没有限制，只要是 Python 支持的数据类型就可以。

从存储内容上看，元组可以存储整数、实数、字符串、列表、元组等任何类型的数据，并且在同一个元组中，元素的类型可以不同，例如：

```
("c.biancheng.net", 1, [2,'a'], ("abc",3.0))
```

在这个元组中，有多种类型的数据，包括整形、字符串、列表、元组。

另外，我们都知道，列表的数据类型是 list，那么元组的数据类型是什么呢？我们不妨通过 type() 函数来查看一下：

```
>>> type( ("c.biancheng.net",1,[2,'a'],("abc",3.0)) )
<class 'tuple'>
```

可以看到，元组是 tuple 类型，这也是很多教程中用 tuple 指代元组的原因。

Python 创建元组

Python 提供了两种创建元组的方法，下面一一进行介绍。

1) 使用 ( ) 直接创建

通过( )创建元组后，一般使用=将它赋值给某个变量，具体格式为：

```
tuplename = (element1, element2, ..., elementn)
```

其中，tuplename 表示变量名，element1 ~ elementn 表示元组的元素。

例如，下面的元组都是合法的：

```
num = (7, 14, 21, 28, 35)
course = ("Python 教程", "http://c.biancheng.net/python/")
abc = ( "Python", 19, [1,2], ('c',2.0) )
```

在 Python 中，元组通常都是使用一对小括号将所有元素包围起来的，但小括号不是必须的，只要将各元素用逗号隔开，Python 就会将其视为元组，请看下面的例子：

```
course = "Python 教程", "http://c.biancheng.net/python/"
print(course)
```

运行结果为：

```
('Python 教程', 'http://c.biancheng.net/python/')
```

需要注意的一点是，当创建的元组中只有一个字符串类型的元素时，该元素后面必须要加一个逗号，，否则 Python 解释器会将它视为字符串。请看下面的代码：

```
#最后加上逗号
a =("http://c.biancheng.net/cplus/",)
print(type(a))
print(a)

#最后不加逗号
b = ("http://c.biancheng.net/socket/")
print(type(b))
print(b)
```

运行结果为：

```
<class 'tuple'>
('http://c.biancheng.net/cplus/',)
<class 'str'>
http://c.biancheng.net/socket/
```

你看，只有变量 a 才是元组，后面的变量 b 是一个字符串。

2) 使用 tuple()函数创建元组

除了使用( )创建元组外，Python 还提供了一个内置的函数 tuple()，用来将其它数据类型转换为元组类型。

tuple() 的语法格式如下：

```
tuple(data)
```

其中，data 表示可以转化为元组的数据，包括字符串、元组、range 对象等。

tuple() 使用示例：

```
#将字符串转换成元组
tup1 = tuple("hello")
print(tup1)

#将列表转换成元组
list1 = ['Python', 'Java', 'C++', 'JavaScript']
tup2 = tuple(list1)
print(tup2)

#将字典转换成元组
dict1 = {'a':100, 'b':42, 'c':9}
tup3 = tuple(dict1)
print(tup3)

#将区间转换成元组
range1 = range(1, 6)
tup4 = tuple(range1)
print(tup4)

#创建空元组
print(tuple())
```

运行结果为：

```
('h', 'e', 'l', 'l', 'o')
('Python', 'Java', 'C++', 'JavaScript')
('a', 'b', 'c')
(1, 2, 3, 4, 5)
()
```

Python 访问元组元素

和列表一样，我们可以使用索引（Index）访问元组中的某个元素（得到的是一个元素的值），也可以使用切片访问元组中的一组元素（得到的是一个新的子元组）。

使用索引访问元组元素的格式为：

```
tuplename[i]
```

其中，tuplename 表示元组名字，i 表示索引值。元组的索引可以是正数，也可以是负数。

使用切片访问元组元素的格式为：

```
tuplename[start : end : step]
```

其中，start 表示起始索引，end 表示结束索引，step 表示步长。

以上两种方式我们已在《Python 序列》中进行了讲解，这里就不再赘述了，仅作示例演示，请看下面代码：

```
1.  url = tuple("http://c.biancheng.net/shell/")
2.
3.  #使用索引访问元组中的某个元素
4.  print(url[3])  #使用正数索引
5.  print(url[-4])  #使用负数索引
6.
7.  #使用切片访问元组中的一组元素
8.  print(url[9: 18])  #使用正数切片
9.  print(url[9: 18: 3])  #指定步长
10. print(url[-6: -1])  #使用负数切片
```

运行结果：

```
p
e
('b', 'i', 'a', 'n', 'c', 'h', 'e', 'n', 'g')
```

```
('b', 'n', 'e')
('s', 'h', 'e', 'l', 'l')
```

Python 修改元组

前面我们已经说过，元组是不可变序列，元组中的元素不能被修改，所以我们只能创建一个新的元组去替代旧的元组。

例如，对元组变量进行重新赋值：

```
1.  tup = (100, 0.5, -36, 73)
2.  print(tup)
3.  #对元组进行重新赋值
4.  tup = ('Shell 脚本',"http://c.biancheng.net/shell/")
5.  print(tup)
```

运行结果为：

```
(100, 0.5, -36, 73)
('Shell 脚本', 'http://c.biancheng.net/shell/')
```

另外，还可以通过连接多个元组（使用`+`可以拼接元组）的方式向元组中添加新元素，例如：

```
1.  tup1 = (100, 0.5, -36, 73)
2.  tup2 = (3+12j, -54.6, 99)
3.  print(tup1+tup2)
4.  print(tup1)
5.  print(tup2)
```

运行结果为：

```
(100, 0.5, -36, 73, (3+12j), -54.6, 99)
(100, 0.5, -36, 73)
((3+12j), -54.6, 99)
```

你看，使用`+`拼接元组以后，tup1 和 tup2 的内容没法发生改变，这说明生成的是一个新的元组。

Python 删除元组

当创建的元组不再使用时，可以通过 del 关键字将其删除，例如：

```
tup = ('Java 教程',"http://c.biancheng.net/java/")
print(tup)
del tup
print(tup)
```

运行结果为：

```
('Java 教程', 'http://c.biancheng.net/java/')
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 4, in <module>
    print(tup)
NameError: name 'tup' is not defined
```

Python 自带垃圾回收功能，会自动销毁不用的元组，所以一般不需要通过 del 来手动删除。

4.11 Python 元组和列表的区别

元组和列表同属序列类型，且都可以按照特定顺序存放一组数据，数据类型不受限制，只要是 Python 支持的数据类型就可以。那么，元组和列表有哪些区别呢？

元组和列表最大的区别就是，列表中的元素可以进行任意修改，就好比是用铅笔在纸上写的字，写错了还可以擦除重写；而元组中的元素无法修改，除非将元组整体替换掉，就好比是用圆珠笔写的字，写了就擦不掉了，除非换一张纸。

可以理解为，tuple 元组是一个只读版本的 list 列表。

需要注意的是，这样的差异势必会影响两者的存储方式，我们来直接看下面的例子：

```
>>> listdemo = []
>>> listdemo.__sizeof__()
40
>>> tupleDemo = ()
>>> tupleDemo.__sizeof__()
24
```

可以看到，对于列表和元组来说，虽然它们都是空的，但元组却比列表少占用 16 个字节，这是为什么呢？

事实上，就是由于列表是动态的，它需要存储指针来指向对应的元素（占用 8 个字节）。另外，由于列表中元素可变，所以需要额外存储已经分配的长度大小（占用 8 个字节）。但是对于元组，情况就不同了，元组长度大小固定，且存储元素不可变，所以存储空间也是固定的。

读者可能会问题，既然列表这么强大，还要元组这种序列类型干什么？

通过对比列表和元组存储方式的差异，我们可以引申出这样的结论，即元组要比列表更加轻量级，所以从总体上来说，元组的性能速度要优于列表。

另外，Python 会在后台，对静态数据做一些资源缓存。通常来说，因为垃圾回收机制的存在，如果一些变量不被使用了，Python 就会回收它们所占用的内存，返还给操作系统，以便其他变量或其他应用使用。

但是对于一些静态变量（比如元组），如果它不被使用并且占用空间不大时，Python 会暂时缓存这部分内存。这样的话，当下次再创建同样大小的元组时，Python 就可以不用再向操作系统发出请求去寻找内存，而是可以直接分配之前缓存的内存空间，这样就能大大加快程序的运行速度。

下面的例子，是计算初始化一个相同元素的列表和元组分别所需的时间。我们可以看到，元组的初始化速度要比列表快 5 倍。

```
C:\Users\mengma>python -m timeit 'x=(1,2,3,4,5,6)'
20000000 loops, best of 5: 9.97 nsec per loop
C:\Users\mengma>python -m timeit 'x=[1,2,3,4,5,6]'
5000000 loops, best of 5: 50.1 nsec per loop
```

当然，如果你想要增加、删减或者改变元素，那么列表显然更优。因为对于元组来说，必须得通过新建一个元组来完成。

总的来说，元组确实没有列表那么多功能，但是元组依旧是很重要的序列类型之一，元组的不可替代性体现在以下这些场景中：

1. 元组作为很多内置函数和序列类型方法的返回值存在，也就是说，在使用某些函数或者方法时，它的返回值会元组类型，因此你必须对元组进行处理。
2. 元组比列表的访问和处理速度更快，因此，当需要对指定元素进行访问，且不涉及修改元素的操作时，建议使用元组。
3. 元组可以在映射（和集合的成员）中当做“键”使用，而列表不行。这会在后续章节中作详解介绍。

4.12 Python 列表和元组的底层实现

有关列表（list）和元组（tuple）的底层实现，本节分别从它们的源码来进行分析。

首先来分析 list 列表，它的具体结构如下所示：

```
typedef struct {
    PyObject_VAR_HEAD
    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;

    /* ob_item contains space for 'allocated' elements.  The number
     * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     * list.sort() temporarily sets allocated to -1 to detect mutations.
     *
     * Items must normally not be NULL, except during construction when
     * the list is not yet visible outside the function that builds it.
     */
    Py_ssize_t allocated;
} PyListObject;
```

有兴趣的读者，可直接阅读 list 列表实现的源码文件 listobject.h 和 listobject.c。
list 本质上是一个长度可变的连续数组。其中 ob_item 是一个指针列表，里边的每一个指针都指向列表中的元素，而 allocated 则用于存储该列表目前已被分配的空间大小。

需要注意的是，allocated 和列表的实际空间大小不同，列表实际空间大小，指的是 len(list) 返回的结果，也就是上边代码中注释中的 ob_size，表示该列表总共存储了多少个元素。而在实际情况中，为了优化存储结构，避免每次增加元素都要重新分配内存，列表预分配的空间 allocated 往往会大于 ob_size。

因此 allocated 和 ob_size 的关系是：`allocated >= len(list) = ob_size >= 0`。

如果当前列表分配的空间已满（即 allocated == len(list)），则会向系统请求更大的内存空间，并把原来的元素全部拷贝过去。

接下来再分析元组，如下所示为 Python 3.7 tuple 元组的具体结构：

```
typedef struct {
```

```
    PyObject_VAR_HEAD
    PyObject *ob_item[1];

    /* ob_item contains space for 'ob_size' elements.
     * Items must normally not be NULL, except during construction when
     * the tuple is not yet visible outside the function that builds it.
     */
} PyTupleObject;
```

有兴趣的读者，可阅读 tuple 元组实现的源码文件 tupleobject.h 和 tupleobject.c。

tuple 和 list 相似，本质也是一个数组，但是空间大小固定。不同于一般数组，Python 的 tuple 做了许多优化，来提升在程序中的效率。

举个例子，为了提高效率，避免频繁的调用系统函数 free 和 malloc 向操作系统申请和释放空间，tuple 源文件中定义了一个 free_list：

```
static PyTupleObject *free_list[PyTuple_MAXSAVESIZE];
```

所有申请过的，小于一定大小的元组，在释放的时候会被放进这个 free_list 中以供下次使用。也就是说，如果以后需要再去创建同样的 tuple，Python 就可以直接从缓存中载入。

4.13 Python dict 字典详解

Python 字典（dict）是一种无序的、可变的序列，它的元素以“键值对（key-value）”的形式存储。相对地，列表（list）和元组（tuple）都是有序的序列，它们的元素在底层是挨着存放的。

字典类型是 Python 中唯一的映射类型。“映射”是数学中的术语，简单理解，它指的是元素之间相互对应的关系，即通过一个元素，可以唯一找到另一个元素。如图 1 所示。

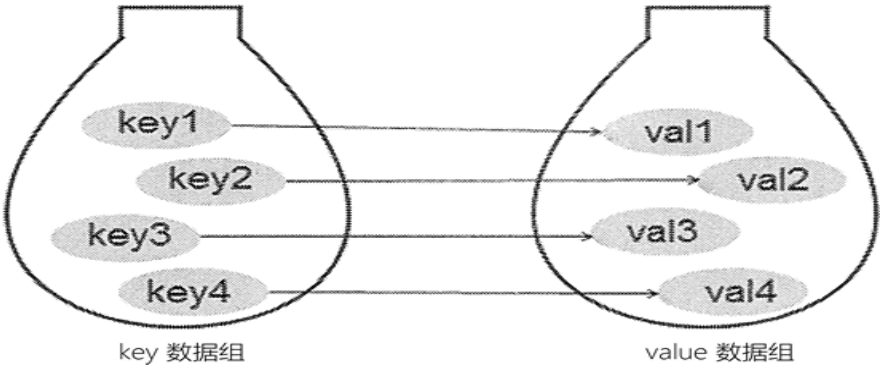


图 1 映射关系示意图

字典中，习惯将各元素对应的索引称为键（key），各个键对应的元素称为值（value），键及其关联的值称为“键值对”。

字典类型很像学生时代常用的新华字典。我们知道，通过新华字典中的音节表，可以快速找到想要查找的汉字。其中，字典里的音节表就相当于字典类型中的键，而键对应的汉字则相当于值。

总的来说，字典类型所具有的主要特征如表 1 所示。

表 1 Python 字典特征

主要特征	解释
通过键而不是通过索引来读取元素	字典类型有时也称为关联数组或者散列表（hash）。它是通过键将一系列的值联系起来的，这样就可以通过键从字典中获取指定项，但不能通过索引来获取。
字典是任意数据类型的无序集合	和列表、元组不同，通常会将索引值 0 对应的元素称为第一个元素，而字典中的元素是无序的。
字典是可变的，并且可以任意嵌套	字典可以在原处增长或者缩短（无需生成一个副本），并且它支持任意深度的嵌套，即字典存储的值也可以是列表或其它的字典。
字典中的键必须唯一	字典中，不支持同一个键出现多次，否则只会保留最后一个键值对。
字典中的键必须不可变	字典中每个键值对的键是不可变的，只能使用数字、字符串或者元组，不能使用列表。

> Python 中的字典类型相当于 Java 或者 C++ 中的 Map 对象。

和列表、元组一样，字典也有它自己的类型。Python 中，字典的数据类型为 dict，通过 type() 函数即可查看：

```
>>> a = {'one': 1, 'two': 2, 'three': 3}  #a 是一个字典类型
>>> type(a)
<class 'dict'>
```

Python 创建字典

创建字典的方式有很多，下面一一做介绍。

1) 使用 { } 创建字典

由于字典中每个元素都包含两部分，分别是键（key）和值（value），因此在创建字典时，键和值之间使用冒号:分隔，相邻元素之间使用逗号,分隔，所有元素放在大括号{ }中。

使用{ }创建字典的语法格式如下：

```
dictname = {'key':'value1', 'key2':'value2', ..., 'keyn':valuen}
```

其中 dictname 表示字典变量名，keyn : valuen 表示各个元素的键值对。需要注意的是，同一字典中的各个键必须唯一，不能重复。

如下代码示范了使用花括号语法创建字典：

```
#使用字符串作为 key
scores = {'数学': 95, '英语': 92, '语文': 84}
print(scores)

#使用元组和数字作为 key
dict1 = {(20, 30): 'great', 30: [1,2,3]}
print(dict1)

#创建空元组
dict2 = {}
print(dict2)
```

运行结果为：

```
{'数学': 95, '英语': 92, '语文': 84}
{(20, 30): 'great', 30: [1, 2, 3]}
{}
```

可以看到，字典的键可以是整数、字符串或者元组，只要符合唯一和不可变的特性就行；字典的值可以是 Python 支持的任意数据类型。

2) 通过 fromkeys() 方法创建字典

Python 中，还可以使用 dict 字典类型提供的 fromkeys() 方法创建带有默认值的字典，具体格式为：

```
dictname = dict.fromkeys(list，value=None)
```

其中，list 参数表示字典中所有键的列表（list）；value 参数表示默认值，如果不写，则为空值 None。

请看下面的例子：

```
knowledge = ['语文', '数学', '英语']
scores = dict.fromkeys(knowledge, 60)
print(scores)
```

运行结果为：

```
{'语文': 60, '英语': 60, '数学': 60}
```

可以看到，knowledge 列表中的元素全部作为了 scores 字典的键，而各个键对应的值都是 60。这种创建方式通常用于初始化字典，设置 value 的默认值。

3) 通过 dict() 映射函数创建字典

通过 dict() 函数创建字典的写法有多种，表 2 罗列出了常用的几种方式，它们创建的都是同一个字典 a。

表 2 dict() 函数创建字典

创建格式	注意事项
a = dict(str1=value1, str2=value2, str3=value3)	str 表示字符串类型的键，value 表示键对应的值。使用此方式创建字典时，字符串不能带引号。

#方式 1
demo = [('two',2), ('one',1), ('three',3)]
#方式 2
demo = [['two',2], ['one',1], ['three',3]]
#方式 3
demo = (('two',2), ('one',1), ('three',3))
#方式 4
demo = (['two',2], ['one',1],
	向 dict() 函数传入列表或元组，而它们中的元素又各自是包含 2 个元素的列表或元组，其中第一个元素作为键，第二个元素作为值。

['three',3]) a = dict(demo)	

keys = ['one', 'two', 'three'] #还可以是字符串或元组
values = [1, 2, 3] #还可以是字符串或元组
a = dict(zip(keys, values))
	通过应用 dict() 函数和 zip() 函数，可将前两个列表转换为对应的字典。

> 注意，无论采用以上哪种方式创建字典，字典中各元素的键都只能是字符串、元组或数字，不能是列表。列表是可变的，不能作为键。

如果不为 dict() 函数传入任何参数，则代表创建一个空的字典，例如：

```
# 创建空的字典
d = dict()
print(d)
```

运行结果为：

```
{}
```

Python 访问字典

列表和元组是通过下标来访问元素的，而字典不同，它通过键来访问对应的值。因为字典中的元素是无序的，每个元素的位置都不固定，所以字典也不能像列表和元组那样，采用切片的方式一次性访问多个元素。

Python 访问字典元素的具体格式为：

```
dictname[key]
```

其中，dictname 表示字典变量的名字，key 表示键名。注意，键必须是存在的，否则会抛出异常。

请看下面的例子：

```
tup = (['two',26], ['one',88], ['three',100], ['four',-59])
dic = dict(tup)
print(dic['one'])  #键存在
print(dic['five'])  #键不存在
```

运行结果：

```
88
Traceback (most recent call last):
```

```
  File "C:\Users\mozhiyan\Desktop\demo.py", line 4, in <module>
    print(dic['five']) #键不存在
KeyError: 'five'
```

除了上面这种方式外，Python 更推荐使用 dict 类型提供的 get() 方法来获取指定键对应的值。当指定的键不存在时，get() 方法不会抛出异常。

get() 方法的语法格式为：

```
dictname.get(key[,default])
```

其中，dictname 表示字典变量的名字；key 表示指定的键；default 用于指定要查询的键不存在时，此方法返回的默认值，如果不手动指定，会返回 None。

get() 使用示例：

```
a = dict(two=0.65, one=88, three=100, four=-59)
print( a.get('one') )
```

运行结果：

```
88
```

注意，当键不存在时，get() 返回空值 None，如果想明确地提示用户该键不存在，那么可以手动设置 get() 的第二个参数，例如：

```
a = dict(two=0.65, one=88, three=100, four=-59)
print( a.get('five', '该键不存在') )
```

运行结果：

```
该键不存在
```

Python 删除字典

和删除列表、元组一样，手动删除字典也可以使用 del 关键字，例如：

```
a = dict(two=0.65, one=88, three=100, four=-59)
print(a)
del a
print(a)
```

运行结果：

```
{'two': 0.65, 'one': 88, 'three': 100, 'four': -59}
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 4, in <module>
    print(a)
NameError: name 'a' is not defined
```

Python 自带垃圾回收功能，会自动销毁不用的字典，所以一般不需要通过 del 来手动删除。

4.14 Python dict 字典基本操作（包括添加、修改、删除键值对）

由于字典属于可变序列，所以我们可以任意操作字典中的键值对（key-value）。Python 中，常见的字典操作有以下几种：

- 向现有字典中添加新的键值对。
- 修改现有字典中的键值对。
- 从现有字典中删除指定的键值对。
- 判断现有字典中是否存在指定的键值对。

> 初学者要牢记，字典是由一个一个的 key-value 构成的，key 是找到数据的关键，Python 对字典的操作都是通过 key 来完成的。

Python 字典添加键值对

为字典添加新的键值对很简单，直接给不存在的 key 赋值即可，具体语法格式如下：

```
dictname[key] = value
```

对各个部分的说明：

- dictname 表示字典名称。
- key 表示新的键。
- value 表示新的值，只要是 Python 支持的数据类型都可以。

下面代码演示了在现有字典基础上添加新元素的过程：

```
1.  a = {'数学':95}
2.  print(a)
3.  #添加新键值对
4.  a['语文'] = 89
5.  print(a)
6.  #再次添加新键值对
7.  a['英语'] = 90
8.  print(a)
```

运行结果：

```
{'数学': 95}
{'数学': 95, '语文': 89}
{'数学': 95, '语文': 89, '英语': 90}
```

Python 字典修改键值对

Python 字典中键（key）的名字不能被修改，我们只能修改值（value）。

字典中各元素的键必须是唯一的，因此，如果新添加元素的键与已存在元素的键相同，那么键所对应的值就会被新的值替换掉，以此达到修改元素值的目的。请看下面的代码：

```
a = {'数学': 95, '语文': 89, '英语': 90}
print(a)
a['语文'] = 100
print(a)
```

运行结果：

```
{'数学': 95, '语文': 89, '英语': 90}
{'数学': 95, '语文': 100, '英语': 90}
```

可以看到，字典中没有再添加一个`{'语文':100}`键值对，而是对原有键值对`{'语文': 89}`中的 value 做了修改。

Python 字典删除键值对

如果要删除字典中的键值对，还是可以使用 del 语句。例如：

```
# 使用 del 语句删除键值对
a = {'数学': 95, '语文': 89, '英语': 90}
del a['语文']
del a['数学']
print(a)
```

运行结果为：

```
{'英语': 90}
```

判断字典中是否存在指定键值对

如果要判断字典中是否存在指定键值对，首先应判断字典中是否有对应的键。判断字典是否包含指定键值对的键，可以使用 in 或 not in 运算符。

> 需要指出的是，对于 dict 而言，in 或 not in 运算符都是基于 key 来判断的。

例如如下代码：

```
a = {'数学': 95, '语文': 89, '英语': 90}
# 判断 a 中是否包含名为'数学'的 key
```

```
print('数学' in a) # True
# 判断 a 是否包含名为'物理'的key
print('物理' in a) # False
```

运行结果为：

```
True
False
```

通过 in（或 not in）运算符，我们可以很轻易地判断出现有字典中是否包含某个键，如果存在，由于通过键可以很轻易的获取对应的值，因此很容易就能判断出字典中是否有指定的键值对。

4.15 Python dict 字典方法完全攻略（全）

我们知道，Python 字典的数据类型为 dict，我们可使用 dir(dict) 来查看该类型包含哪些方法，例如：

```
>>> dir(dict)
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

这些方法中，fromkeys() 和 get() 的用法已在《Python 字典》中进行了介绍，这里不再赘述，本节只给大家介绍剩下的方法。

keys()、values() 和 items() 方法

将这三个方法放在一起介绍，是因为它们都用来获取字典中的特定数据：

- keys() 方法用于返回字典中的所有键（key）；
- values() 方法用于返回字典中所有键对应的值（value）；
- items() 用于返回字典中所有的键值对（key-value）。

请看下面的例子：

```
scores = {'数学': 95, '语文': 89, '英语': 90}
print(scores.keys())
print(scores.values())
print(scores.items())
```

运行结果：

```
dict_keys(['数学', '语文', '英语'])
dict_values([95, 89, 90])
dict_items([('数学', 95), ('语文', 89), ('英语', 90)])
```

可以发现，keys()、values() 和 items() 返回值的类型分别为 dict_keys、dict_values 和 dict_items。

需要注意的是，在 Python 2.x 中，上面三个方法的返回值都是列表（list）类型。但在 Python 3.x 中，它们的返回值并不是我们常见的列表或者元组类型，因为 Python 3.x 不希望用户直接操作这几个方法的返回值。

在 Python 3.x 中如果想使用这三个方法返回的数据，一般有下面两种方案：

1) 使用 list() 函数，将它们返回的数据转换成列表，例如：

```
a = {'数学': 95, '语文': 89, '英语': 90}
b = list(a.keys())
```

```
print(b)
```

运行结果为：

```
['数学', '语文', '英语']
```

2) 使用 for in 循环遍历它们的返回值，例如：

```
a = {'数学': 95, '语文': 89, '英语': 90}
for k in a.keys():
    print(k,end=' ')
print("\n---------------")
for v in a.values():
    print(v,end=' ')
print("\n---------------")
for k,v in a.items():
    print("key:",k," value:",v)
```

运行结果为：

```
数学 语文 英语
---------------
95 89 90
---------------
key: 数学  value: 95
key: 语文  value: 89
key: 英语  value: 90
```

copy() 方法

copy() 方法返回一个字典的拷贝，也即返回一个具有相同键值对的新字典，例如：

```
a = {'one': 1, 'two': 2, 'three': [1,2,3]}
b = a.copy()
print(b)
```

运行结果为：

```
{'one': 1, 'two': 2, 'three': [1, 2, 3]}
```

可以看到，copy() 方法将字典 a 的数据全部拷贝给了字典 b。

注意，copy() 方法所遵循的拷贝原理，既有深拷贝，也有浅拷贝。拿拷贝字典 a 为例，copy() 方法只会对最表层的键值对进行深拷贝，也就是说，它会再申请一块内存用来存放 {'one': 1, 'two': 2, 'three': []}；而对于某些列表类型的值来说，此方法对其做的是浅拷贝，也就是说，b 中的 [1,2,3] 的值不是自己独有，而是和 a 共有。

请看下面的例子：

```
a = {'one': 1, 'two': 2, 'three': [1,2,3]}
b = a.copy()
#向 a 中添加新键值对，由于 b 已经提前将 a 所有键值对都深拷贝过来，因此 a 添加新键值对，不会影响 b。
a['four']=100
print(a)
print(b)
#由于 b 和 a 共享[1,2,3]（浅拷贝），因此移除 a 中列表中的元素，也会影响 b。
a['three'].remove(1)
print(a)
print(b)
```

运行结果为：

```
{'one': 1, 'two': 2, 'three': [1, 2, 3], 'four': 100}
{'one': 1, 'two': 2, 'three': [1, 2, 3]}
{'one': 1, 'two': 2, 'three': [2, 3], 'four': 100}
{'one': 1, 'two': 2, 'three': [2, 3]}
```

从运行结果不难看出，对 a 增加新键值对，b 不变；而修改 a 某键值对中列表内的元素，b 也会相应改变。

update() 方法

update() 方法可以使用一个字典所包含的键值对来更新已有的字典。

在执行 update() 方法时，如果被更新的字典中已包含对应的键值对，那么原 value 会被覆盖；如果被更新的字典中不包含对应的键值对，则该键值对被添加进去。

请看下面的代码：

```
a = {'one': 1, 'two': 2, 'three': 3}
a.update({'one':4.5, 'four': 9.3})
print(a)
```

运行结果为：

```
{'one': 4.5, 'two': 2, 'three': 3, 'four': 9.3}
```

从运行结果可以看出，由于被更新的字典中已包含 key 为“one”的键值对，因此更新时该键值对的 value 将被改写；而被更新的字典中不包含 key 为“four”的键值对，所以更新时会为原字典增加一个新的键值对。

pop() 和 popitem() 方法

pop() 和 popitem() 都用来删除字典中的键值对，不同的是，pop() 用来删除指定的键值对，而 popitem() 用来随机删除一个键值对，它们的语法格式如下：

```
dictname.pop(key)
dictname.popitem()
```

其中，dictname 表示字典名称，key 表示键。

下面的代码演示了两个函数的用法：

```
a = {'数学': 95, '语文': 89, '英语': 90, '化学': 83, '生物': 98, '物理': 89}
print(a)
a.pop('化学')
print(a)
a.popitem()
print(a)
```

运行结果：

```
{'数学': 95, '语文': 89, '英语': 90, '化学': 83, '生物': 98, '物理': 89}
{'数学': 95, '语文': 89, '英语': 90, '生物': 98, '物理': 89}
{'数学': 95, '语文': 89, '英语': 90, '生物': 98}
```

对 popitem() 的说明

其实，说 popitem() 随机删除字典中的一个键值对是不准确的，虽然字典是一种无须的列表，但键值对在底层也是有存储顺序的，popitem() 总是弹出底层中的最后一个 key-value，这和列表的 pop() 方法类似，都实现了数据结构中“出栈”的操作。

setdefault() 方法

setdefault() 方法用来返回某个 key 对应的 value，其语法格式如下：

```
dictname.setdefault(key, defaultvalue)
```

说明，dictname 表示字典名称，key 表示键，defaultvalue 表示默认值（可以不写，不写的话是 None）。

当指定的 key 不存在时，setdefault() 会先为这个不存在的 key 设置一个默认的 defaultvalue，然后再返回 defaultvalue。

也就是说，setdefault() 方法总能返回指定 key 对应的 value：

- 如果该 key 存在，那么直接返回该 key 对应的 value；
- 如果该 key 不存在，那么先为该 key 设置默认的 defaultvalue，然后再返回该 key 对应的 defaultvalue。

请看下面的代码：

```
a = {'数学': 95, '语文': 89, '英语': 90}
print(a)
#key 不存在，指定默认值
a.setdefault('物理', 94)
print(a)
#key 不存在，不指定默认值
a.setdefault('化学')
print(a)
#key 存在，指定默认值
a.setdefault('数学', 100)
print(a)
```

运行结果为：

```
{'数学': 95, '语文': 89, '英语': 90}
{'数学': 95, '语文': 89, '英语': 90, '物理': 94}
{'数学': 95, '语文': 89, '英语': 90, '物理': 94, '化学': None}
{'数学': 95, '语文': 89, '英语': 90, '物理': 94, '化学': None}
```

4.16Python 使用字典格式化字符串

在《Python 格式化输出》一节中，我们介绍了如何使用 print() 格式化输出各种类型的数据。我们知道，如果格式化字符串的模板中包含了多个转换说明符，后面就得按照顺序给出多个对应的变量；当字符串模板中只包含少量转换说明符时，这种写法还是比较合适的，但如果字符串模板中包含大量转换说明符，这种按顺序提供变量的方式就有些麻烦了。

这时，就可以使用字典对字符串进行格式化输出，具体方法是：在字符串模板中按 key 指定变量，然后通过字典为字符串模板中的 key 设置值。

请看下面的代码：

```
# 字符串模板中使用 key
temp = '教程是：%(name)s, 价格是：%(price).2f, 网址是：%(url)s'
course = {'name':'Python 教程', 'price': 9.9, 'url': 'http://c.biancheng.net/python/'}
# 使用字典为字符串模板中的 key 传入值
print(temp % course)
course = {'name':'C++教程', 'price':15.6, 'url': 'http://c.biancheng.net/cplus/'}
# 使用字典为字符串模板中的 key 传入值
print(temp % course)
```

运行上面程序，可以看到如下输出结果：

```
教程是：Python 教程, 价格是：9.90, 网址是：http://c.biancheng.net/python/
教程是：C++教程, 价格是：15.60, 网址是：http://c.biancheng.net/cplus/
```

4.17 Python set 集合详解

Python 中的集合，和数学中的集合概念一样，用来保存不重复的元素，即集合中的元素都是唯一的，互不相同。

从形式上看，和字典类似，Python 集合会将所有元素放在一对大括号 {} 中，相邻元素之间用“,”分隔，如下所示：

```
{element1,element2,...,elementn}
```

其中，elementn 表示集合中的元素，个数没有限制。

从内容上看，同一集合中，只能存储不可变的数据类型，包括整形、浮点型、字符串、元组，无法存储列表、字典、集合这些可变的数据类型，否则 Python 解释器会抛出 TypeError 错误。比如说：

```
>>> {{'a':1}}
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    {{'a':1}}
TypeError: unhashable type: 'dict'
>>> {[1,2,3]}
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    {[1,2,3]}
TypeError: unhashable type: 'list'
>>> {{1,2,3}}
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    {{1,2,3}}
TypeError: unhashable type: 'set'
```

并且需要注意的是，数据必须保证是唯一的，因为集合对于每种数据元素，只会保留一份。例如：

```
>>> {1,2,1,(1,2,3),'c','c'}
{1, 2, 'c', (1, 2, 3)}
```

由于 Python 中的 set 集合是无序的，所以每次输出时元素的排序顺序可能都不相同。

其实，Python 中有两种集合类型，一种是 set 类型的集合，另一种是 frozenset 类型的集合，它们唯一的区别是，set 类型集合可以做添加、删除元素的操作，而 forzenset 类型集合不行。本节先介绍 set 类型集合，后续章节再介绍 forzenset 类型集合。

Python 创建 set 集合

Python 提供了 2 种创建 set 集合的方法，分别是使用 {} 创建和使用 set() 函数将列表、元组等类型数据转换为集合。

1) 使用 {} 创建

在 Python 中，创建 set 集合可以像列表、元素和字典一样，直接将集合赋值给变量，从而实现创建集合的目的，其语法格式如下：

```
setname = {element1,element2,...,elementn}
```

其中，setname 表示集合的名称，起名时既要符合 Python 命名规范，也要避免与 Python 内置函数重名。

举个例子：

```
a = {1,'c',1,(1,2,3),'c'}
print(a)
```

运行结果为：

```
{1, 'c', (1, 2, 3)}
```

2) set()函数创建集合

set() 函数为 Python 的内置函数，其功能是将字符串、列表、元组、range 对象等可迭代对象转换成集合。该函数的语法格式如下：

```
setname = set(iteration)
```

其中，iteration 就表示字符串、列表、元组、range 对象等数据。

例如：

```
set1 = set("c.biancheng.net")
set2 = set([1,2,3,4,5])
set3 = set((1,2,3,4,5))
print("set1:",set1)
print("set2:",set2)
print("set3:",set3)
```

运行结果为：

```
set1: {'a', 'g', 'b', 'c', 'n', 'h', '.', 't', 'i', 'e'}
set2: {1, 2, 3, 4, 5}
set3: {1, 2, 3, 4, 5}
```

注意，如果要创建空集合，只能使用 set() 函数实现。因为直接使用一对 {}，Python 解释器会将其视为一个空字典。

Python 访问 set 集合元素

由于集合中的元素是无序的，因此无法向列表那样使用下标访问元素。Python 中，访问集合元素最常用的方法是使用循环结构，将集合中的数据逐一读取出来。

例如：

```
a = {1,'c',1,(1,2,3),'c'}
for ele in a:
    print(ele,end=' ')
```

运行结果为：

```
1 c (1, 2, 3)
```

由于目前尚未学习循环结构，以上代码初学者只需初步了解，后续学习循环结构后自然会明白。

Python 删除 set 集合

和其他序列类型一样，手动函数集合类型，也可以使用 del() 语句，例如：

```
a = {1,'c',1,(1,2,3),'c'}
print(a)
del(a)
print(a)
```

运行结果为：

```
{1, 'c', (1, 2, 3)}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(a)
NameError: name 'a' is not defined
```

Python set 集合最常用的操作是向集合中添加、删除元素，以及集合之间做交集、并集、差集等运算。受到篇幅的限制，这些知识会放到下节进行详细讲解。

4.18 Python set 集合基本操作（添加、删除、交集、并集、差集）

Python set 集合最常用的操作是向集合中添加、删除元素，以及集合之间做交集、并集、差集等运算，本节将一一讲解这些操作的具体实现。

向 set 集合中添加元素

set 集合中添加元素，可以使用 set 类型提供的 add() 方法实现，该方法的语法格式为：

```
setname.add(element)
```

其中，setname 表示要添加元素的集合，element 表示要添加的元素内容。

需要注意的是，使用 add() 方法添加的元素，只能是数字、字符串、元组或者布尔类型（True 和 False）值，不能添加列表、字典、集合这类可变的数据，否则 Python 解释器会报 TypeError 错误。例如：

```
1.  a = {1, 2, 3}
2.  a.add((1, 2))
3.  print(a)
4.  a.add([1, 2])
5.  print(a)
```

运行结果为：

```
{(1, 2), 1, 2, 3}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    a.add([1,2])
TypeError: unhashable type: 'list'
```

从 set 集合中删除元素

删除现有 set 集合中的指定元素，可以使用 remove() 方法，该方法的语法格式如下：

```
setname.remove(element)
```

使用此方法删除集合中元素，需要注意的是，如果被删除元素本就不包含在集合中，则此方法会抛出 KeyError 错误，例如：

```
1.  a = {1, 2, 3}
```

```
a.remove(1)
print(a)
a.remove(1)
print(a)
```

运行结果为：

```
{2, 3}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    a.remove(1)
KeyError: 1
```

上面程序中，由于集合中的元素 1 已被删除，因此当再次尝试使用 remove() 方法删除时，会引发 KeyError 错误。

如果我们不想在删除失败时令解释器提示 KeyError 错误，还可以使用 discard() 方法，此方法和 remove() 方法的用法完全相同，唯一的区别就是，当删除集合中元素失败时，此方法不会抛出任何错误。

例如：

```
a = {1,2,3}
a.remove(1)
print(a)
a.discard(1)
print(a)
```

运行结果为：

```
{2, 3}
{2, 3}
```

Python set 集合做交集、并集、差集运算

集合最常做的操作就是进行交集、并集、差集以及对称差集运算，首先有必要给大家普及一下各个运算的含义。

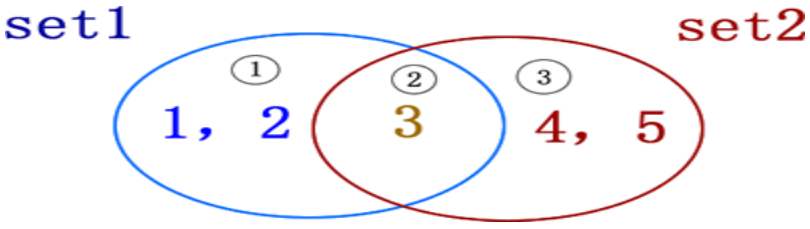


图 1 集合示意图

图 1 中，有 2 个集合，分别为 set1={1,2,3} 和 set2={3,4,5}，它们既有相同的元素，也有不同的元素。以这两个集合为例，分别做不同运算的结果如表 1 所示。

表 1 Python set 集合运算

运算操作	Python 运算符	含义	例子
交集	&	取两集合公共的元素	>>> set1 & set2 {3}
并集	|	取两集合全部的元素	>>> set1 | set2 {1,2,3,4,5}

差集	-	取一个集合中另一集合没有的元素	>>> set1 - set2
			{1,2}
			>>> set2 - set1
			{4,5}

对称差集	^	取集合 A 和 B 中不属于 A&B 的元素	>>> set1 ^ set2 {1,2,4,5}

4.19 Python set 集合方法详解（全）

前面学习了 set 集合，本节来一一学习 set 类型提供的方法。首先，通过 dir(set) 命令可以查看它有哪些方法：

```
>>> dir(set)
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update',
'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update']
```

各个方法的具体语法结构及功能如表 1 所示。

表 1 Python set 方法

方法名	语法格式	功能	实例

add()	set1.add()	向 set1 集合中添加数字、字符串、元组或者布尔类型	>>> set1 = {1,2,3}
			>>> set1.add((1,2))
			>>> set1
			{(1, 2), 1, 2, 3}

clear()	set1.clear()	清空 set1 集合中所有元素	>>> set1 = {1,2,3}
			>>> set1.clear()
			>>> set1
			set()
			
			set()才表示空集合，{}表示的是空字典

copy()	set2 = set1.copy()	拷贝 set1 集合给 set2	>>> set1 = {1,2,3}
			>>> set2 = set1.copy()
			>>> set1.add(4)
			>>> set1
			{1, 2, 3, 4}
			>>> set1
			{1, 2, 3}

difference()	set3 = set1.difference(set2)	将 set1 中有而	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.difference(set2)
			>>> set3

		set2没有的元素给set3	{1, 2}

difference_update()	set1.difference_update(set2)	从set1中删除与set2相同的元素	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.difference_update(set2)
			>>> set1
			{1, 2}

discard()	set1.discard(elem)	删除set1中的elem元素	>>> set1 = {1,2,3}
			>>> set1.discard(2)
			>>> set1
			{1, 3}
			>>> set1.discard(4)
			{1, 3}

intersection()	set3 = set1.intersection(set2)	取set1和set2的交集给set3	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.intersection(set2)
			>>> set3
			{3}

intersection_update()	set1.intersection_update(set2)	取set1和set2的交集，并更新给set1	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.intersection_update(set2)
			>>> set1
			{3}

isdisjoint()	set1.isdisjoint(set2)	判断set1和set2是否没有交集，有交集返	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.isdisjoint(set2)
			False

		回False；没有交集返回True	

issubset()	set1.issubset(set2)	判断set1是否是set2的子集	>>> set1 = {1,2,3}
			>>> set2 = {1,2}
			>>> set1.issubset(set2)
			False

issuperset()	set1.issuperset(set2)	判断set2是否是set1的子集	>>> set1 = {1,2,3}
			>>> set2 = {1,2}
			>>> set1.issuperset(set2)
			True

pop()	a = set1.pop()	取set1中一个元素，并赋值给a	>>> set1 = {1,2,3}
			>>> a = set1.pop()
			>>> set1
			{2,3}
			>>> a
			1

remove()	set1.remove(elem)	移除set1中的elem元素	>>> set1 = {1,2,3}
			>>> set1.remove(2)
			>>> set1
			{1, 3}
			>>> set1.remove(4)
			Traceback (most recent call last):
			File "<pyshell#90>", line 1, in <module>
			set1.remove(4)
			KeyError: 4

symmetric_difference()	set3 = set1.symmetric_difference(set2)	取set1和set2中互不相	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3 = set1.symmetric_difference(set2)
			>>> set3
			{1, 2, 4}

		同的元素，给set3	

symmetric_difference_update()	set1.symmetric_difference_update(set2)	取set1和set2中互不相同的元素，并更新给set1	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set1.symmetric_difference_update(set2)
			>>> set1
			{1, 2, 4}

union()	set3 = set1.union(set2)	取set1和set2的并集，赋给set3	>>> set1 = {1,2,3}
			>>> set2 = {3,4}
			>>> set3=set1.union(set2)
			>>> set3
			{1, 2, 3, 4}

update()	set1.update(elem)	添加列表或集合中的元素到set1	>>> set1 = {1,2,3}
			>>> set1.update([3,4])
			>>> set1
			{1,2,3,4}

4.20 Python frozenset 集合（set 集合的不可变版本）

set 集合是可变序列，程序可以改变序列中的元素；frozenset 集合是不可变序列，程序不能改变序列中的元素。set 集合中所有能改变集合本身的方法，比如 remove()、discard()、add() 等，frozenset 都不支持；set 集合中不改变集合本身的方法，fronzenset 都支持。

我们可以在交互式编程环境中输入 dir(frozenset)来查看 frozenset 集合支持的方法：

```
>>> dir(frozenset)
['copy', 'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

frozenset 集合的这些方法和 set 集合中同名方法的功能是一样的。

两种情况下可以使用 fronzenset：

- 当集合的元素不需要改变时，我们可以使用 fronzenset 替代 set，这样更加安全。
- 有时候程序要求必须是不可变对象，这个时候也要使用 fronzenset 替代 set。比如，字典（dict）的键（key）就要求是不可变对象。

下面程序演示了 frozenset 的用法：

```
1.  s = {'Python', 'C', 'C++'}
2.  fs = frozenset(['Java', 'Shell'])
3.  s_sub = {'PHP', 'C#'}
4.
5.  #向 set 集合中添加 frozenset
6.  s.add(fs)
7.  print('s =', s)
8.
9.
10. #向为 set 集合添加子 set 集合
11. s.add(s_sub)
12. print('s =', s)
```

运行结果：

```
s = {'Python', frozenset({'Java', 'Shell'}), 'C', 'C++'}
Traceback (most recent call last):
  File "C:\Users\mozhiyan\Desktop\demo.py", line 11, in <module>
```

```
        s.add(s_sub)
TypeError: unhashable type: 'set'
```

需要注意的是，set 集合本身的元素必须是不可变的， 所以 set 的元素不能是 set，只能是 frozenset。第 6 行代码向 set 中添加 frozenset 是没问题的，因为 frozenset 是不可变的；但是，第 10 行代码中尝试向 set 中添加子 set，这是不允许的，因为 set 是可变的。

4.21 深入底层了解 Python 字典和集合，一眼看穿他们的本质！

字典和集合是进行过性能高度优化的数据结构，特别是对于查找、添加和删除操作。本节将结合实例介绍它们在具体场景下的性能表现，以及与列表等其他数据结构的对比。

例如，有一个存储产品信息（产品 ID、名称和价格）的列表，现在的需求是，借助某件产品的 ID 找出其价格。则实现代码如下：

```
def find_product_price(products, product_id):
    for id, price in products:
        if id == product_id:
            return price
    return None
products = [
    (111, 100),
    (222, 30),
    (333, 150)
]
print('The price of product 222 is {}'.format(find_product_price(products, 222)))
```

运行结果为：

```
The price of product 222 is 30
```

在上面程序的基础上，如果列表有 n 个元素，因为查找的过程需要遍历列表，那么最坏情况下的时间复杂度就为 `O(n)`。即使先对列表进行排序，再使用二分查找算法，也需要 `O(logn)` 的时间复杂度，更何况列表的排序还需要 `O(nlogn)` 的时间。

但如果用字典来存储这些数据，那么查找就会非常便捷高效，只需 `O(1)` 的时间复杂度就可以完成，因为可以直接通过键的哈希值，找到其对应的值，而不需要对字典做遍历操作，实现代码如下：

```
products = {
  111: 100,
  222: 30,
  333: 150
}
print('The price of product 222 is {}'.format(products[222]))
```

运行结果为：

```
The price of product 222 is 30
```

有些读者可能对时间复杂度并没有直观的认识，没关系，再给大家列举一个实例。下面的代码中，初始化了含有 100,000 个元素的产品，并分别计算出了使用列表和集合来统计产品价格数量的运行时间：

```
#统计时间需要用到 time 模块中的函数，了解即可
import time

def find_unique_price_using_list(products):
    unique_price_list = []
    for _, price in products: # A
        if price not in unique_price_list: #B
            unique_price_list.append(price)
    return len(unique_price_list)

id = [x for x in range(0, 100000)]
price = [x for x in range(200000, 300000)]
products = list(zip(id, price))

# 计算列表版本的时间
start_using_list = time.perf_counter()
find_unique_price_using_list(products)
end_using_list = time.perf_counter()
print("time elapse using list: {}".format(end_using_list - start_using_list))

#使用集合完成同样的工作
def find_unique_price_using_set(products):
    unique_price_set = set()
    for _, price in products:
        unique_price_set.add(price)
    return len(unique_price_set)

# 计算集合版本的时间
start_using_set = time.perf_counter()
find_unique_price_using_set(products)
```

```
end_using_set = time.perf_counter()
print("time elapse using set: {}".format(end_using_set - start_using_set))
```

运行结果为：

```
time elapse using list: 68.78650900000001
time elapse using set: 0.010747099999989018
```

可以看到，仅仅十万的数据量，两者的速度差异就如此之大。而往往企业的后台数据都有上亿乃至十亿数量级，因此如果使用了不合适的数据结构，很容易造成服务器的崩溃，不但影响用户体验，并且会给公司带来巨大的财产损失。

那么，字典和集合为什么能如此高效，特别是查找、插入和删除操作呢？

字典和集合的工作原理

字典和集合能如此高效，和它们内部的数据结构密不可分。不同于其他数据结构，字典和集合的内部结构都是一张哈希表：

- 对于字典而言，这张表存储了哈希值（hash）、键和值这 3 个元素。
- 而对集合来说，哈希表内只存储单一的元素。

对于之前版本的 Python 来说，它的哈希表结构如下所示：

```
  | 哈希值 (hash)  键 (key)  值 (value)
. |          ...
0 |   hash0      key0    value0
. |          ...
1 |   hash1      key1    value1
. |          ...
2 |   hash2      key2    value2
. |          ...
```

这种结构的弊端是，随着哈希表的扩张，它会变得越来越稀疏。比如，有这样一个字典：

```
{'name': 'mike', 'dob': '1999-01-01', 'gender': 'male'}
```

那么它会存储为类似下面的形式：

```
entries = [
['--', '--', '--']
[-230273521, 'dob', '1999-01-01'],
['--', '--', '--'],
['--', '--', '--'],
[1231236123, 'name', 'mike'],
['--', '--', '--'],
[9371539127, 'gender', 'male']
]
```

显然，这样非常浪费存储空间。为了提高存储空间的利用率，现在的哈希表除了字典本身的结构，会把索引和哈希值、键、值单独分开，也就是采用如下这种结构：

```
Indices
----------------------------------------------------
None | index | None | None | index | None | index ...
----------------------------------------------------


Entries
--------------------
hash0   key0  value0
---------------------
hash1   key1  value1
---------------------
hash2   key2  value2
---------------------
        ...
--------------------
```

在此基础上，上面的字典在新哈希表结构下的存储形式为：

```
indices = [None, 1, None, None, 0, None, 2]
```

```
entries = [
[1231236123, 'name', 'mike'],
[-230273521, 'dob', '1999-01-01'],
[9371539127, 'gender', 'male']
]
```

通过对比可以发现，空间利用率得到很大的提高。

清楚了具体的设计结构，接下来再分析一下如何使用哈希表完成对数据的插入、查找和删除操作。

哈希表插入数据

当向字典中插入数据时，Python 会首先根据键（key）计算出对应的哈希值（通过 hash(key) 函数），而向集合中插入数据时，Python 会根据该元素本身计算对应的哈希值（通过 hash(valuse) 函数）。

例如：

```
dic = {"name":1}
print(hash("name"))
setDemo = {1}
print(hash(1))
```

运行结果为：

```
8230115042008314683
1
```

得到哈希值（例如为 hash）之后，再结合字典或集合要存储数据的个数（例如 n），就可以得到该元素应该插入到哈希表中的位置（比如，可以用 hash%n 的方式）。

如果哈希表中此位置是空的，那么此元素就可以直接插入其中；反之，如果此位置已被其他元素占用，那么 Python 会比较这两个元素的哈希值和键是否相等：

- 如果相等，则表明该元素已经存在，再比较他们的值，不相等就进行更新；
- 如果不相等，这种情况称为哈希冲突（即两个元素的键不同，但求得的哈希值相同）。这种情况下，Python 会使用开放定址法、再哈希法等继续寻找哈希表中空余的位置，直到找到位置。

具体遇到哈希冲突时，各解决方法的具体含义可阅读《哈希表详解》一节做详细了解。

哈希表查找数据

在哈希表中查找数据，和插入操作类似，Python 会根据哈希值，找到该元素应该存储到哈希表中的位置，然后和该位置的元素比较其哈希值和键（集合直接比较元素值）：

- 如果相等，则证明找到；
- 反之，则证明当初存储该元素时，遇到了哈希冲突，需要继续使用当初解决哈希冲突的方法进行查找，直到找到该元素或者找到空位为止。

> 这里的找到空位，表示哈希表中没有存储目标元素。

哈希表删除元素

对于删除操作，Python 会暂时对这个位置的元素赋于一个特殊的值，等到重新调整哈希表的大小时，再将其删除。

需要注意的是，哈希冲突的发生往往会降低字典和集合操作的速度。因此，为了保证其高效性，字典和集合内的哈希表，通常会保证其至少留有 1/3 的剩余空间。随着元素的不停插入，当剩余空间小于 1/3 时，Python 会重新获取更大的内存空间，扩充哈希表，与此同时，表内所有的元素位置都会被重新排放。

虽然哈希冲突和哈希表大小的调整，都会导致速度减缓，但是这种情况发生的次数极少。所以，平均情况下，仍能保证插入、查找和删除的时间复杂度为 `O(1)`。

4.22 Python 深拷贝和浅拷贝详解

对于浅拷贝（shallow copy）和深度拷贝（deep copy），本节并不打算一上来抛出它们的概念，而是先从它们的操作方法说起，通过代码来理解两者的不同。

Python 浅拷贝

常见的浅拷贝的方法，是使用数据类型本身的构造器，比如下面两个例子：

```
1.  list1 = [1, 2, 3]
2.  list2 = list(list1)
3.  print(list2)
4.  print("list1==list2 ?",list1==list2)
5.  print("list1 is list2 ?",list1 is list2)
6.
7.  set1= set([1, 2, 3])
8.  set2 = set(set1)
9.  print(set2)
10. print("set1==set2 ?",set1==set2)
11. print("set1 is set2 ?",set1 is set2)
```

运行结果为：

```
[1, 2, 3]
list1==list2 ? True
list1 is list2 ? False
{1, 2, 3}
set1==set2 ? True
set1 is set2 ? False
```

在上面程序中，list2 就是 list1 的浅拷贝，同理 set2 是 set1 的浅拷贝。

当然，对于可变的序列，还可以通过切片操作符 “：” 来完成浅拷贝，例如：

```
1.  list1 = [1, 2, 3]
2.  list2 = list1[:]
3.  print(list2)
4.  print("list1 == list2 ?",list1 == list2)
5.  print("list1 is list2 ?",list1 is list2)
```

运行结果为：

```
[1, 2, 3]
list1 == list2 ? True
list1 is list2 ? False
```

除此之外，Python 还提供了对应的函数 copy.copy() 函数，适用于任何数据类型。其用法如下：

```
import copy
list1 = [1, 2, 3]
list2 = copy.copy(list1)
print(list2)
print("list1 == list2 ?",list1 == list2)
print("list1 is list2 ?",list1 is list2)
```

运行结果为：

```
[1, 2, 3]
list1 == list2 ? True
list1 is list2 ? False
```

不过需要注意的是，对于元组，使用 tuple() 或者切片操作符 ':' 不会创建一份浅拷贝，相反它会返回一个指向相同元组的引用：

```
tuple1 = (1, 2, 3)
tuple2 = tuple(tuple1)
print(tuple2)
print("tuple1 == tuple2 ?",tuple1 == tuple2)
print("tuple1 is tuple2 ?",tuple1 is tuple2)
```

运行结果为：

```
(1, 2, 3)
tuple1 == tuple2 ? True
tuple1 is tuple2 ? True
```

此程序中，元组 (1, 2, 3) 只被创建一次，t1 和 t2 同时指向这个元组。

看到这里，也许你可能对浅拷贝有了初步的认识。浅拷贝，指的是重新分配一块内存，创建一个新的对象，但里面的元素是原对象中各个子对象的引用。

对数据采用浅拷贝的方式时，如果原对象中的元素不可变，那倒无所谓；但如果元素可变，浅拷贝通常会出现一些问题，例如：

```
list1 = [[1, 2], (30, 40)]
list2 = list(list1)

```

```
list1.append(100)
print("list1:",list1)
print("list2:",list2)

list1[0].append(3)
print("list1:",list1)
print("list2:",list2)

list1[1] += (50, 60)
print("list1:",list1)
print("list2:",list2)
```

运行结果为：

```
list1: [[1, 2], (30, 40), 100]
list2: [[1, 2], (30, 40)]
list1: [[1, 2, 3], (30, 40), 100]
list2: [[1, 2, 3], (30, 40)]
list1: [[1, 2, 3], (30, 40, 50, 60), 100]
list2: [[1, 2, 3], (30, 40)]
```

此程序中，首先初始化了 list1 列表，包含一个列表和一个元组；然后对 list1 执行浅拷贝，赋予 list2。因为浅拷贝里的元素是对原对象元素的引用，因此 list2 中的元素和 list1 指向同一个列表和元组对象。

接着往下看，list1.append(100) 表示对 list1 的列表新增元素 100。这个操作不会对 list2 产生任何影响，因为 list2 和 list1 作为整体是两个不同的对象，并不共享内存地址。操作过后 list2 不变，list1 会发生改变。

再来看，list1[0].append(3) 表示对 list1 中的第一个列表新增元素 3。因为 list2 是 list1 的浅拷贝，list2 中的第一个元素和 list1 中的第一个元素，共同指向同一个列表，因此 list2 中的第一个列表也会相对应的新增元素 3。

最后是 list1[1] += (50, 60)，因为元组是不可变的，这里表示对 list1 中的第二个元组拼接，然后重新创建了一个新元组作为 list1 中的第二个元素，而 list2 中没有引用新元组，因此 list2 并不受影响。

通过这个例子，你可以很清楚地看到使用浅拷贝可能带来的副作用。如果想避免这种副作用，完整地拷贝一个对象，就需要使用深拷贝。所谓深拷贝，是指重新分配一块内存，创建一个新的对象，并且将原对象中的元素，以递归的方式，通过创建新的子对象拷贝到新对象中。因此，新对象和原对象没有任何关联。

Python 中以 copy.deepcopy() 来实现对象的深度拷贝。比如上述例子写成下面的形式，就是深度拷贝：

```
import copy
list1 = [[1, 2], (30, 40)]
list2 = copy.deepcopy(list1)
```

```
4.
5.  list1.append(100)
6.  print("list1:",list1)
7.  print("list2:",list2)
8.
9.  list1[0].append(3)
10. print("list1:",list1)
11. print("list2:",list2)
12.
13. list1[1] += (50, 60)
14. print("list1:",list1)
15. print("list2:",list2)
```

运行结果为：

```
list1: [[1, 2], (30, 40), 100]
list2: [[1, 2], (30, 40)]
list1: [[1, 2, 3], (30, 40), 100]
list2: [[1, 2], (30, 40)]
list1: [[1, 2, 3], (30, 40, 50, 60), 100]
list2: [[1, 2], (30, 40)]
```

可以看到，无论 list1 如何变化，list2 都不变。因为此时的 list1 和 list2 完全独立，没有任何联系。

不过，深度拷贝也不是完美的，往往也会带来一系列问题。如果被拷贝对象中存在指向自身的引用，那么程序很容易陷入无限循环，例如：

```
1.  import copy
2.  list1 = [1]
3.  list1.append(list1)
4.  print(list1)
5.
6.  list2 = copy.deepcopy(list1)
7.  print(list2)
```

运行结果为：

```
[1, [...]]
[1, [...]]
```

此例子中，列表 x 中有指向自身的引用，因此 x 是一个无限嵌套的列表。但是当深度拷贝 x 到 y 后，程序并没有出现栈溢出的现象。这是为什么呢？

其实，这是因为深度拷贝函数 deepcopy 中会维护一个字典，记录已经拷贝的对象与其 ID。拷贝过程中，如果字典里已经存储了将要拷贝的对象，则会从字典直接返回。通过查看 deepcopy 函数实现的源码就会明白：

纯文本复制

```
def deepcopy(x, memo=None, _nil=[]):
    """Deep copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.
    """

    if memo is None:
        memo = {}
    d = id(x) # 查询被拷贝对象 x 的 id
    y = memo.get(d, _nil) # 查询字典里是否已经存储了该对象
    if y is not _nil:
        return y # 如果字典里已经存储了将要拷贝的对象，则直接返回
```

第 5 章： Python 字符串常用方法详解

5.1 Python 字符串拼接（包含字符串拼接数字）

在 Python 中拼接（连接）字符串很简单，可以直接将两个字符串紧挨着写在一起，具体格式为：

```
strname = "str1" "str2"
```

strname 表示拼接以后的字符串变量名，str1 和 str2 是要拼接的字符串内容。使用这种写法，Python 会自动将两个字符串拼接在一起。

【示例】以连续书写的形式拼接字符串：

```
1. str1 = "Python 教程" "http://c.biancheng.net/python/"
2. print(str1)
3. str2 = "Java" "Python" "C++" "PHP"
4. print(str2)
```

运行结果：

```
Python 教程 http://c.biancheng.net/python/
JavaPythonC++PHP
```

需要注意的是，这种写法只能拼接字符串常量。

如果需要使用变量，就得借助+运算符来拼接，具体格式为：

```
strname = str1 + str2
```

当然，+运算符也能拼接字符串常量。

【示例】使用+运算符拼接字符串：

```
1. name = "C++教程"
2. url = "http://c.biancheng.net/cplus/"
3. info = name + "的网址是：" + url
4. print(info)
```

运行结果：

```
C++教程的网址是：http://c.biancheng.net/cplus/
```

Python 字符串和数字的拼接

在很多应用场景中，我们需要将字符串和数字拼接在一起，而 Python 不允许直接拼接数字和字符串，所以我们必须先将数字转换成字符串。可以借助 str() 和 repr() 函数将数字转换为字符串，它们的使用格式为：

```
str(obj)
repr(obj)
```

obj 表示要转换的对象，它可以是数字、列表、元组、字典等多种类型的数据。

请看下面的代码：

```
name = "C 语言中文网"
age = 8
course = 30
info = name + "已经" + str(age) + "岁了，共发布了" + repr(course) + "套教程。"
print(info)
```

运行结果：

```
C 语言中文网已经 8 岁了，共发布了 30 套教程。
```

str() 和 repr() 的区别

str() 和 repr() 函数虽然都可以将数字转换成字符串，但它们之间是有区别的：

- str() 用于将数据转换成适合人类阅读的字符串形式。
- repr() 用于将数据转换成适合解释器阅读的字符串形式（Python 表达式的形式），适合在开发和调试阶段使用；如果没有等价的语法，则会发生 SyntaxError 异常。

请看下面的例子：

```
s = "http://c.biancheng.net/shell/"
s_str = str(s)
s_repr = repr(s)

print( type(s_str) )
print (s_str)
print( type(s_repr) )
print (s_repr)
```

运行结果：

```
<class 'str'>
http://c.biancheng.net/shell/
<class 'str'>
'http://c.biancheng.net/shell/'
```

本例中，s 本身就是一个字符串，但是我们依然使用 str() 和 repr() 对它进行了转换。从运行结果可以看出，str() 保留了字符串最原始的样子，而 repr() 使用引号将字符串包围起来，这就是 Python 字符串的表达式形式。

另外，在 Python 交互式编程环境中输入一个表达式（变量、加减乘除、逻辑运算等）时，Python 会自动使用 repr() 函数处理该表达式。

5.2 Python 截取字符串（字符串切片）方法详解

从本质上讲，字符串是由多个字符构成的，字符之间是有顺序的，这个顺序号就称为索引（index）。

Python 允许通过索引来操作字符串中的单个或者多个字符，比如获取指定索引处的字符，返回指定字符的索引值等。

获取单个字符

知道字符串名字以后，在方括号[ ]中使用索引即可访问对应的字符，具体的语法格式为：

```
strname[index]
```

strname 表示字符串名字，index 表示索引值。

Python 允许从字符串的两端使用索引：

- 当以字符串的左端（字符串的开头）为起点时，索引是从 0 开始计数的；字符串的第一个字符的索引为 0，第二个字符的索引为 1，第三个字符串的索引为 2 ……
- 当以字符串的右端（字符串的末尾）为起点时，索引是从 -1 开始计数的；字符串的倒数第一个字符的索引为 -1，倒数第二个字符的索引为 -2，倒数第三个字符的索引为 -3 ……

请看下面的实例演示：

```
1. url = 'http://c.biancheng.net/python/'
2. #获取索引为 10 的字符
3. print(url[10])
4. #获取索引为 6 的字符
5. print(url[-6])
```

运行结果：

```
i
y
```

获取多个字符（字符串截去/字符串切片）

使用[ ]除了可以获取单个字符外，还可以指定一个范围来获取多个字符，也就是一个子串或者片段，具体格式为：

```
strname[start : end : step]
```

对各个部分的说明：

- strname：要截取的字符串；
- start：表示要截取的第一个字符所在的索引（截取时包含该字符）。如果不指定，默认为 0，也就是从字符串的开头截取；
- end：表示要截取的最后一个字符所在的索引（截取时不包含该字符）。如果不指定，默认为字符串的长度；

- step：指的是从 start 索引处的字符开始，每 step 个距离获取一个字符，直至 end 索引出的字符。step 默认值为 1，当省略该值时，最后一个冒号也可以省略。

【实例 1】基本用法：

```
url = 'http://c.biancheng.net/java/'
#获取索引从 3 处 22（不包含 22）的子串
print(url[7: 22]) # 输出 zy
#获取索引从 7 处到-6 的子串
print(url[7: -6]) # 输出 zyit.org is very
#获取索引从-7 到 6 的子串
print(url[-21: -6])
#从索引 3 开始，每隔 4 个字符取出一个字符，直到索引 22 为止
print(url[3: 22: 4])
```

运行结果：

```
c.biancheng.net
c.biancheng.net
c.biancheng.net
pcaen
```

【实例 2】高级用法，start、end、step 三个参数都可以省略：

```
url = 'http://c.biancheng.net/java/'
#获取从索引 5 开始，直到末尾的子串
print(url[7: ])
#获取从索引-21 开始，直到末尾的子串
print(url[-21: ])
#从开头截取字符串，直到索引 22 为止
print(url[: 22])
#每隔 3 个字符取出一个字符
print(url[:: 3])
```

运行结果：

```
c.biancheng.net/java/
c.biancheng.net/java/
```

```
http://c.biancheng.net
hp/bne.ta/
```

5.3 Python len()函数详解：获取字符串长度或字节数

Python 中，要想知道一个字符串有多少个字符（获得字符串长度），或者一个字符串占用多少个字节，可以使用 len 函数。

len 函数的基本语法格式为：

```
len（string）
```

其中 string 用于指定要进行长度统计的字符串。

例如，定义一个字符串，内容为“http://c.biancheng.net”，然后用 len() 函数计算该字符串的长度，执行代码如下：

```
>>> a='http://c.biancheng.net'
>>> len(a)
22
```

在实际开发中，除了常常要获取字符串的长度外，有时还要获取字符串的字节数。

在 Python 中，不同的字符所占的字节数不同，数字、英文字母、小数点、下划线以及空格，各占一个字节，而一个汉字可能占 2~4 个字节，具体占多少个，取决于采用的编码方式。例如，汉字在 GBK/GB2312 编码中占用 2 个字节，而在 UTF-8 编码中一般占用 3 个字节。

以 UTF-8 编码为例，字符串“人生苦短，我用 Python”所占用的字节数如图 1 所示。

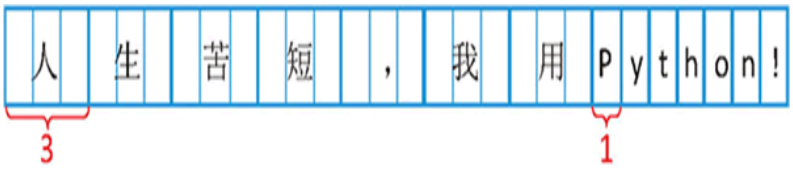


图 1 汉字和英文所占字节数

我们可以通过使用 encode() 方法，将字符串进行编码后再获取它的字节数。例如，采用 UTF-8 编码方式，计算“人生苦短，我用 Python”的字节数，可以执行如下代码：

```
>>> str1 = "人生苦短，我用 Python"
>>> len(str1.encode())
27
```

因为汉字加中文标点符号共 7 个，占 21 个字节，而英文字母和英文的标点符号占 6 个字节，一共占用 27 个字节。

同理，如果要获取采用 GBK 编码的字符串的长度，可以执行如下代码：

```
>>> str1 = "人生苦短，我用 Python"
>>> len(str1.encode('gbk'))
20
```

5.4 Python split()方法详解：分割字符串

Python 中，除了可以使用一些内建函数获取字符串的相关信息外（例如 len() 函数获取字符串长度），字符串类型本身也拥有一些方法供我们使用。

> 注意，这里所说的方法，指的是字符串类型 str 本身所提供的，由于涉及到类和对象的知识，初学者不必深究，只需要知道方法的具体用法即可。

从本节开始，将给大家介绍一些常用的字符串类型方法，本节先介绍分割字符串的 split() 方法。

split() 方法可以实现将一个字符串按照指定的分隔符切分成多个子串，这些子串会被保存到列表中（不包含分隔符），作为方法的返回值反馈回来。该方法的基本语法格式如下：

```
str.split(sep,maxsplit)
```

此方法中各部分参数的含义分别是：

1. str：表示要进行分割的字符串；
2. sep：用于指定分隔符，可以包含多个字符。此参数默认为 None，表示所有空字符，包括空格、换行符“\n”、制表符“\t”等。
3. maxsplit：可选参数，用于指定分割的次数，最后列表中子串的个数最多为 maxsplit+1。如果不指定或者指定为 -1，则表示分割次数没有限制。

在 split 方法中，如果不指定 sep 参数，那么也不能指定 maxsplit 参数。

> 同内建函数（如 len）的使用方式不同，字符串变量所拥有的方法，只能采用“字符串.方法名()”的方式调用。这里不用纠结为什么，学完类和对象之后，自然会明白。

例如，定义一个保存 C 语言中文网网址的字符串，然后用 split() 方法根据不同的分隔符进行分隔，执行过程如下：

```
>>> str = "C 语言中文网 >>> c.biancheng.net"
>>> str
'C 语言中文网 >>> c.biancheng.net'
>>> list1 = str.split() #采用默认分隔符进行分割
>>> list1
['C 语言中文网', '>>>', 'c.biancheng.net']
>>> list2 = str.split('>>>') #采用多个字符进行分割
>>> list2
['C 语言中文网 ', ' c.biancheng.net']
>>> list3 = str.split('.') #采用 . 号进行分割
>>> list3
```

```
['C 语言中文网 >>> c', 'biancheng', 'net']
>>> list4 = str.split(' ',4) #采用空格进行分割，并规定最多只能分割成 4 个子串
>>> list4
['C 语言中文网', '>>>', 'c.biancheng.net']
>>> list5 = str.split('>') #采用 > 字符进行分割
>>> list5
['C 语言中文网 ', '', '', ' c.biancheng.net']
>>>
```

需要注意的是，在未指定 sep 参数时，split() 方法默认采用空字符进行分割，但当字符串中有连续的空格或其他空字符时，都会被视为一个分隔符对字符串进行分割，例如：

```
>>> str = "C 语言中文网  >>>  c.biancheng.net"  #包含 3 个连续的空格
>>> list6 = str.split()
>>> list6
['C 语言中文网', '>>>', 'c.biancheng.net']
>>>
```

5.5 Python join()方法：合并字符串

join() 方法也是非常重要的字符串方法，它是 split() 方法的逆方法，用来将列表（或元组）中包含的多个字符串连接成一个字符串。

想详细了解 split() 方法的读者，可阅读《Python split()方法》一节。

使用 join() 方法合并字符串时，它会将列表（或元组）中多个字符串采用固定的分隔符连接在一起。例如，字符串“c.biancheng.net”就可以看做是通过分隔符“.”将 ['c','biancheng','net'] 列表合并为一个字符串的结果。

join() 方法的语法格式如下：

```
newstr = str.join(iterable)
```

此方法中各参数的含义如下：

1. newstr：表示合并后生成的新字符串；
2. str：用于指定合并时的分隔符；
3. iterable：做合并操作的源字符串数据，允许以列表、元组等形式提供。

【例 1】将列表中的字符串合并成一个字符串。

```
>>> list = ['c','biancheng','net']
>>> '.'.join(list)
'c.biancheng.net'
```

【例 2】将元组中的字符串合并成一个字符串。

```
>>> dir = '','usr','bin','env'
>>> type(dir)
<class 'tuple'>
>>> '/'.join(dir)
'/usr/bin/env'
```

5.6 Python count()方法：统计字符串出现的次数

count 方法用于检索指定字符串在另一字符串中出现的次数，如果检索的字符串不存在，则返回 0，否则返回出现的次数。

count 方法的语法格式如下：

```
str.count(sub[,start[,end]])
```

此方法中，各参数的具体含义如下：

1. str：表示原字符串；
2. sub：表示要检索的字符串；
3. start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
4. end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。

【例 1】检索字符串“c.biancheng.net”中“.”出现的次数。

```
>>> str = "c.biancheng.net"
>>> str.count('.')
2
```

【例 2】

```
>>> str = "c.biancheng.net"
>>> str.count('.',1)
2
>>> str.count('.',2)
1
```

前面讲过，字符串中各字符对应的检索值，从 0 开始，因此，本例中检索值 1 对应的是第 2 个字符‘.’，从输出结果可以分析出，从指定索引位置开始检索，其中也包含此索引位置。

【例 3】

```
>>> str = "c.biancheng.net"
>>> str.count('.',2,-3)
1
>>> str.count('.',2,-4)
0
```

5.7 Python find()方法：检测字符串中是否包含某子串

find() 方法用于检索字符串中是否包含目标字符串，如果包含，则返回第一次出现该字符串的索引；反之，则返回 -1。

find() 方法的语法格式如下：

```
str.find(sub[,start[,end]])
```

此格式中各参数的含义如下：

1. str：表示原字符串；
2. sub：表示要检索的目标字符串；
3. start：表示开始检索的起始位置。如果不指定，则默认从头开始检索；
4. end：表示结束检索的结束位置。如果不指定，则默认一直检索到结尾。

【例 1】用 find() 方法检索 “c.biancheng.net” 中首次出现 “.” 的位置索引。

```
>>> str = "c.biancheng.net"
>>> str.find('.')
1
```

【例 2】手动指定起始索引的位置。

```
>>> str = "c.biancheng.net"
>>> str.find('.',2)
11
```

【例 3】手动指定起始索引和结束索引的位置。

```
>>> str = "c.biancheng.net"
>>> str.find('.',2,-4)
-1
```

位于索引（2，-4）之间的字符串为“biancheng”，由于其不包含“.”，因此 find() 方法的返回值为 -1。

注意，Python 还提供了 rfind() 方法，与 find() 方法最大的不同在于，rfind() 是从字符串右边开始检索。例如：

```
>>> str = "c.biancheng.net"
>>> str.rfind('.')
11
```

5.8 Python index()方法：检测字符串中是否包含某子串

同 find() 方法类似，index() 方法也可以用于检索是否包含指定的字符串，不同之处在于，当指定的字符串不存在时，index() 方法会抛出异常。

index() 方法的语法格式如下：

```
str.index(sub[,start[,end]])
```

此格式中各参数的含义分别是：

1. str：表示原字符串；
2. sub：表示要检索的子字符串；
3. start：表示检索开始的起始位置，如果不指定，默认从头开始检索；
4. end：表示检索的结束位置，如果不指定，默认一直检索到结尾。

【例 1】用 index() 方法检索“c.biancheng.net”中首次出现“.”的位置索引。

```
>>> str = "c.biancheng.net"
>>> str.index('.')
1
```

【例 2】当检索失败时，index()会抛出异常。

```
>>> str = "c.biancheng.net"
>>> str.index('z')
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    str.index('z')
ValueError: substring not found
```

同 find() 和 rfind() 一样，字符串变量还具有 rindex() 方法，其作用和 index() 方法类似，不同之处在于它是从右边开始检索，例如：

```
>>> str = "c.biancheng.net"
>>> str.rindex('.')
11
```

5.9 Python 字符串对齐方法（ljust()、rjust()和center()）详解

Python str 提供了 3 种可用来进行文本对齐的方法，分别是 ljust()、rjust() 和 center() 方法，本节就来一一介绍它们的用法。

Python ljust()方法

ljust() 方法的功能是向指定字符串的右侧填充指定字符，从而达到左对齐文本的目的。

ljust() 方法的基本格式如下：

```
S.ljust(width[, fillchar])
```

其中各个参数的含义如下：

- S：表示要进行填充的字符串；
- width：表示包括 S 本身长度在内，字符串要占的总长度；
- fillchar：作为可选参数，用来指定填充字符串时所用的字符，默认情况使用空格。

【例 1】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.ljust(35))
print(addr.ljust(35))
```

输出结果为：

```
http://c.biancheng.net/python/
http://c.biancheng.net
```

注意，该输出结果中除了明显可见的网址字符串外，其后还有空格字符存在，每行一共 35 个字符长度。

【例 2】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.ljust(35,'-'))
```

```
4.  print(addr.ljust(35,'-'))
```

输出结果为：

```
http://c.biancheng.net/python/-----

http://c.biancheng.net----------
```

此程序和例 1 的唯一区别是，填充字符从空格改为 '-' 。

Python rjust()方法

rjust() 和 ljust() 方法类似，唯一的不同在于，rjust() 方法是向字符串的左侧填充指定字符，从而达到右对齐文本的目的。

rjust() 方法的基本格式如下：

```
S.rjust(width[, fillchar])
```

其中各个参数的含义和 ljust() 完全相同，所以这里不再重复描述。

【例 3】

```
1.  S = 'http://c.biancheng.net/python/'
2.  addr = 'http://c.biancheng.net'
3.  print(S.rjust(35))
4.  print(addr.rjust(35))
```

输出结果为：

```
    http://c.biancheng.net/python/
            http://c.biancheng.net
```

可以看到，每行字符串都占用 35 个字节的位置，实现了整体的右对齐效果。

【例 4】

```
1.  S = 'http://c.biancheng.net/python/'
2.  addr = 'http://c.biancheng.net'
3.  print(S.rjust(35,'-'))
4.  print(addr.rjust(35,'-'))
```

输出结果为：

```
-----http://c.biancheng.net/python/

-------------http://c.biancheng.net
```

Python center()方法

center() 字符串方法与 ljust() 和 rjust() 的用法类似，但它让文本居中，而不是左对齐或右对齐。

center() 方法的基本格式如下：

```
S.center(width[, fillchar])
```

其中各个参数的含义和 ljust()、rjust() 方法相同。

【例 5】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.center(35,))
print(addr.center(35,))
```

输出结果为：

```
   http://c.biancheng.net/python/

       http://c.biancheng.net
```

【例 6】

```
S = 'http://c.biancheng.net/python/'
addr = 'http://c.biancheng.net'
print(S.center(35,'-'))
print(addr.center(35,'-'))
```

输出结果为：

```
---http://c.biancheng.net/python/--

-------http://c.biancheng.net------
```

5.10 Python startswith()和 endswith()方法

除了前面介绍的几个方法外，Python 字符串变量还可以使用 startswith() 和 endswith() 方法。

startswith()方法

startswith() 方法用于检索字符串是否以指定字符串开头，如果是返回 True；反之返回 False。此方法的语法格式如下：

```
str.startswith(sub[,start[,end]])
```

此格式中各个参数的具体含义如下：

1. str：表示原字符串；
2. sub：要检索的子串；
3. start：指定检索开始的起始位置索引，如果不指定，则默认从头开始检索；
4. end：指定检索的结束位置索引，如果不指定，则默认一直检索在结束。

【例 1】判断“c.biancheng.net”是否以“c”子串开头。

```
>>> str = "c.biancheng.net"
>>> str.startswith("c")
True
```

【例 2】

```
>>> str = "c.biancheng.net"
>>> str.startswith("http")
False
```

【例 3】从指定位置开始检索。

```
>>> str = "c.biancheng.net"
>>> str.startswith("b",2)
True
```

endswith()方法

endswith() 方法用于检索字符串是否以指定字符串结尾，如果是则返回 True；反之则返回 False。该方法的语法格式如下：

```
str.endswith(sub[,start[,end]])
```

此格式中各参数的含义如下：

1. str：表示原字符串；
2. sub：表示要检索的字符串；
3. start：指定检索开始时的起始位置索引（字符串第一个字符对应的索引值为 0），如果不指定，默认从头开始检索。
4. end：指定检索的结束位置索引，如果不指定，默认一直检索到结束。

【例 4】检索“c.biancheng.net”是否以“net”结束。

```
>>> str = "c.biancheng.net"
>>> str.endswith("net")
True
```

5.11Python 字符串大小写转换（3 种）函数及用法

Python 中，为了方便对字符串中的字母进行大小写转换，字符串变量提供了 3 种方法，分别是 title()、lower() 和 upper()。

Python title()方法

title() 方法用于将字符串中每个单词的首字母转为大写，其他字母全部转为小写，转换完成后，此方法会返回转换得到的字符串。如果字符串中没有需要被转换的字符，此方法会将字符串原封不动地返回。

title() 方法的语法格式如下：

```
str.title()
```

其中，str 表示要进行转换的字符串。

【例 1】

```
>>> str = "c.biancheng.net"
>>> str.title()
'C.Biancheng.Net'
>>> str = "I LIKE C"
>>> str.title()
'I Like C'
```

Python lower()方法

lower() 方法用于将字符串中的所有大写字母转换为小写字母，转换完成后，该方法会返回新得到的字符串。如果字符串中原本就都是小写字母，则该方法会返回原字符串。

lower() 方法的语法格式如下：

```
str.lower()
```

其中，str 表示要进行转换的字符串。

【例 2】

```
>>> str = "I LIKE C"
>>> str.lower()
'i like c'
```

Python upper()方法

upper() 的功能和 lower() 方法恰好相反，它用于将字符串中的所有小写字母转换为大写字母，和以上两种方法的返回方式相同，即如果转换成功，则返回新字符串；反之，则返回原字符串。

upper() 方法的语法格式如下：

```
str.upper()
```

其中，str 表示要进行转换的字符串。

【例 3】

```
>>> str = "i like C"
>>> str.upper()
'I LIKE C'
```

需要注意的是，以上 3 个方法都仅限于将转换后的新字符串返回，而不会修改原字符串。

5.12Python 去除字符串中空格（删除指定字符）的 3 种方法

用户输入数据时，很有可能会无意中输入多余的空格，或者在一些场景中，字符串前后不允许出现空格和特殊字符，此时就需要去除字符串中的空格和特殊字符。

> 这里的特殊字符，指的是制表符（\t）、回车符（\r）、换行符（\n）等。

Python 中，字符串变量提供了 3 种方法来删除字符串中多余的空格和特殊字符，它们分别是：

1. strip()：删除字符串前后（左右两侧）的空格或特殊字符。
2. lstrip()：删除字符串前面（左边）的空格或特殊字符。
3. rstrip()：删除字符串后面（右边）的空格或特殊字符。

注意，Python 的 str 是不可变的（不可变的意思是指，字符串一旦形成，它所包含的字符序列就不能发生任何改变），因此这三个方法只是返回字符串前面或后面空白被删除之后的副本，并不会改变字符串本身。

Python strip()方法

strip() 方法用于删除字符串左右两个的空格和特殊字符，该方法的语法格式为：

```
str.strip([chars])
```

其中，str 表示原字符串，[chars] 用来指定要删除的字符，可以同时指定多个，如果不手动指定，则默认会删除空格以及制表符、回车符、换行符等特殊字符。

【例 1】

```
>>> str = "  c.biancheng.net \t\n\r"
>>> str.strip()
'c.biancheng.net'
>>> str.strip(" ,\r")
'c.biancheng.net \t\n'
>>> str
'  c.biancheng.net \t\n\r'
```

分析运行结果不难看出，通过 strip() 确实能够删除字符串左右两侧的空格和特殊字符，但并没有真正改变字符串本身。

Python lstrip()方法

lstrip() 方法用于去掉字符串左侧的空格和特殊字符。该方法的语法格式如下：

```
str.lstrip([chars])
```

其中，str 和 chars 参数的含义，分别同 strip() 语法格式中的 str 和 chars 完全相同。

【例 2】

```
>>> str = "  c.biancheng.net \t\n\r"
>>> str.lstrip()
'c.biancheng.net \t\n\r'
```

Python rstrip()方法

rstrip() 方法用于删除字符串右侧的空格和特殊字符，其语法格式为：

```
str.rstrip([chars])
```

str 和 chars 参数的含义和前面 2 种方法语法格式中的参数完全相同。

【例 3】

```
>>> str = "  c.biancheng.net \t\n\r"
>>> str.rstrip()
'  c.biancheng.net'
```

5.13 Python format()格式化输出方法详解

前面章节介绍了如何使用 % 操作符对各种类型的数据进行格式化输出，这是早期 Python 提供的方法。自 Python 2.6 版本开始，字符串类型（ str ）提供了 format() 方法对字符串进行格式化，本节就来学习此方法。

format() 方法的语法格式如下：

```
str.format(args)
```

此方法中，str 用于指定字符串的显示样式；args 用于指定要进行格式转换的项，如果有多项，之间有逗号进行分割。

学习 format() 方法的难点，在于搞清楚 str 显示样式的书写格式。在创建显示样式模板时，需要使用{}和：来指定占位符，其完整的语法格式为：

```
{ [index][ : [ [fill] align] [sign] [#] [width] [.precision] [type] ] }
```

注意，格式中用 [] 括起来的参数都是可选参数，即可以使用，也可以不使用。各个参数的含义如下：

- index：指定：后边设置的格式要作用到 args 中第几个数据，数据的索引值从 0 开始。如果省略此选项，则会根据 args 中数据的先后顺序自动分配。
- fill：指定空白处填充的字符。注意，当填充字符为逗号(,)且作用于整数或浮点数时，该整数（或浮点数）会以逗号分隔的形式输出，例如（1000000 会输出 1,000,000）。
- align：指定数据的对齐方式，具体的对齐方式如表 1 所示。

表 1 align 参数及含义

align	含义
<	数据左对齐。
>	数据右对齐。
=	数据右对齐，同时将符号放置在填充内容的最左侧，该选项只对数字类型有效。
^	数据居中，此选项需和 width 参数一起使用。

- sign：指定有无符号数，此参数的值以及对应的含义如表 2 所示。

表 2 sign 参数以含义

sign 参数	含义
+	正数前加正号，负数前加负号。

-	正数前不加正号，负数前加负号。
空格	正数前加空格，负数前加负号。
#	对于二进制数、八进制数和十六进制数，使用此参数，各进制数前会分别显示 0b、0o、0x 前缀；反之则不显示前缀。

- width：指定输出数据时所占的宽度。
- .precision：指定保留的小数位数。
- type：指定输出数据的具体类型，如表 3 所示。

表 3 type 占位符类型及含义

type 类型值	含义
s	对字符串类型格式化。
d	十进制整数。
c	将十进制整数自动转换成对应的 Unicode 字符。
e 或者 E	转换成科学计数法后，再格式化输出。
g 或 G	自动在 e 和 f（或 E 和 F）中切换。
b	将十进制数自动转换成二进制表示，再格式化输出。
o	将十进制数自动转换成八进制表示，再格式化输出。
x 或者 X	将十进制数自动转换成十六进制表示，再格式化输出。
f 或者 F	转换为浮点数（默认小数点后保留 6 位），再格式化输出。
%	显示百分比（默认显示小数点后 6 位）。

【例 1】

```
1.  str="网站名称：{:>9s}\t 网址：{:s}"
2.  print(str.format("C 语言中文网","c.biancheng.net"))
```

输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 2, in
    print(str.format("C 语言中文网","c.biancheng.net"))
ValueError: cannot switch from automatic field numbering to manual field specification
```

【例 2】

在实际开发中，数值类型有多种显示需求，比如货币形式、百分比形式等，使用 format() 方法可以将数值格式化为不同的形式。

```
#以货币形式显示
print("货币形式：{:,d}".format(1000000))
#科学计数法表示
print("科学计数法：{:E}".format(1200.12))
#以十六进制表示
print("100 的十六进制：{:#x}".format(100))
#输出百分比形式
print("0.01 的百分比表示：{:.0%}".format(0.01))
```

输出结果为：

```
货币形式：1,000,000
科学计数法：1.200120E+03
100 的十六进制：0x64
0.01 的百分比表示：1%
```

5.14Python encode()和 decode()方法：字符串编码转换

前面章节在介绍 bytes 类型时，已经对 encode() 和 decode() 方法的使用做了简单的介绍，本节将对这 2 个方法做详细地说明。

我们知道，最早的字符串编码是 ASCII 编码，它仅仅对 10 个数字、26 个大小写英文字母以及一些特殊字符进行了编码。ASCII 码做多只能表示 256 个符号，每个字符只需要占用 1 个字节。

随着信息技术的发展，各国的文字都需要进行编码，于是相继出现了 GBK、GB2312、UTF-8 编码等，其中 GBK 和 GB2312 是我国制定的中文编码标准，规定英文字符母占用 1 个字节，中文字符占用 2 个字节；而 UTF-8 是国际通过的编码格式，它包含了全世界所有国家需要用到的字符，其规定英文字符占用 1 个字节，中文字符占用 3 个字节。

Python 3.x 默认采用 UTF-8 编码格式，有效地解决了中文乱码的问题。

在 Python 中，有 2 种常用的字符串类型，分别为 str 和 bytes 类型，其中 str 用来表示 Unicode 字符，bytes 用来表示二进制数据。str 类型和 bytes 类型之间就需要使用 encode() 和 decode() 方法进行转换。

Python encode()方法

encode() 方法为字符串类型（ str ）提供的方法，用于将 str 类型转换成 bytes 类型，这个过程也称为“编码”。

encode() 方法的语法格式如下：

```
str.encode([encoding="utf-8"][,errors="strict"])
```

注意，格式中用 [] 括起来的参数为可选参数，也就是说，在使用此方法时，可以使用 [] 中的参数，也可以不使用。

该方法各个参数的含义如表 1 所示。

表 1 encode()参数及含义

参数	含义
str	表示要进行转换的字符串。
encoding = "utf-8"	指定进行编码时采用的字符编码，该选项默认采用 utf-8 编码。例如，如果想使用简体中文，可以设置 gb2312。 当方法中只使用这一个参数时，可以省略前边的“encoding=” ，直接写编码格式，例如 str.encode("UTF-8")。

errors = "strict"	指定错误处理方式，其可选择值可以是： strict：遇到非法字符就抛出异常。 ignore：忽略非法字符。 replace：用“？”替换非法字符。 xmlcharrefreplace：使用 xml 的字符引用。 该参数的默认值为 strict。

注意，使用 encode() 方法对原字符串进行编码，不会直接修改原字符串，如果想修改原字符串，需要重新赋值。

【例 1】将 str 类型字符串“C 语言中文网”转换成 bytes 类型。

```
>>> str = "C 语言中文网"
>>> str.encode()
b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x91'
```

此方式默认采用 UTF-8 编码，也可以手动指定其它编码格式，例如：

```
>>> str = "C 语言中文网"
>>> str.encode('GBK')
b'C\xd3\xef\xd1\xd4\xd6\xd0\xce\xc4\xcd\xf8'
```

Python decode()方法

和 encode() 方法正好相反，decode() 方法用于将 bytes 类型的二进制数据转换为 str 类型，这个过程也称为“解码”。

decode() 方法的语法格式如下：

```
bytes.decode([encoding="utf-8"][,errors="strict"])
```

该方法中各参数的含义如表 2 所示。

表 2 decode()参数及含义

参数	含义
bytes	表示要进行转换的二进制数据。
encoding="utf-8"	指定解码时采用的字符编码，默认采用 utf-8 格式。当方法中只使用这一个参数时，可以省略“encoding=”，直接写编码方式即可。

	注意，对 bytes 类型数据解码，要选择和当初编码时一样的格式。
errors = "strict"	指定错误处理方式，其可选择值可以是： strict：遇到非法字符就抛出异常。 ignore：忽略非法字符。 replace：用“？”替换非法字符。 xmlcharrefreplace：使用 xml 的字符引用。 该参数的默认值为 strict。

【例 2】

```
>>> str = "C 语言中文网"
>>> bytes=str.encode()
>>> bytes.decode()
'C 语言中文网'
```

注意，如果编码时采用的不是默认的 UTF-8 编码，则解码时要选择和编码时一样的格式，否则会抛出异常，例如：

```
>>> str = "C 语言中文网"
>>> bytes = str.encode("GBK")
>>> bytes.decode()  #默认使用 UTF-8 编码，会抛出以下异常
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    bytes.decode()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd3 in position 1: invalid continuation byte
>>> bytes.decode("GBK")
'C 语言中文网'
```

5.15 Python dir()和 help()帮助函数

前面我们已经学习了很多字符串提供的方法，包括 split()、join()、find()、index() 等，但这远远不是它的全部方法。由于篇幅有限，本章只能给大家列举一些最常用的方法，至于其他的方法，读者可通过本节介绍的 dir() 和 help() 函数自行查看。

Python dir() 函数用来列出某个类或者某个模块中的全部内容，包括变量、方法、函数和类等，它的用法为：

```
dir(obj)
```

obj 表示要查看的对象。obj 可以不写，此时 dir() 会列出当前范围内的变量、方法和定义的类型。

Python help() 函数用来查看某个函数或者模块的帮助文档，它的用法为：

```
help(obj)
```

obj 表示要查看的对象。obj 可以不写，此时 help() 会进入帮助子程序。

掌握了以上两个函数，我们就可以自行查阅 Python 中所有方法、函数、变量、类的用法和功能了。

【实例】使用 dir() 查看字符串类型（str）支持的所有方法：

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

在 Python 标准库中，以__开头和结尾的方法都是私有的，不能在类的外部调用。

【实例】使用 help() 查看 str 类型中 lower() 函数的用法：

```
>>> help(str.lower)
Help on method_descriptor:

lower(self, /)
    Return a copy of the string converted to lowercase.
```

可以看到，lower() 函数用来将字符串中的字母转换为小写形式，并返回一个新的字符串。

注意，使用 help() 查看某个函数的用法时，函数名后边不能带括号，例如将上面的命令写作 help(str.lower()) 就是错误的。

第 6 章：Python 流程控制

6.1 Python if else 条件语句详解

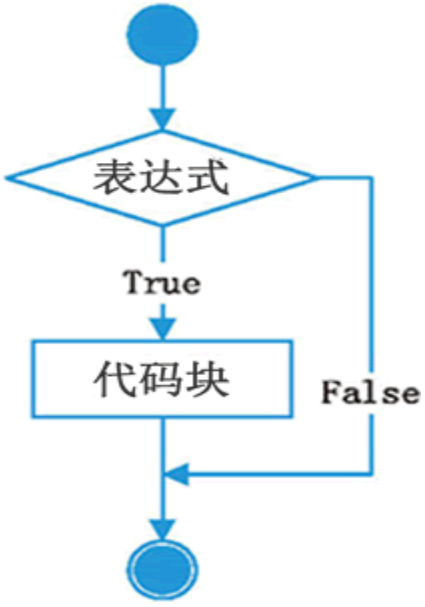
前面我们看到的代码都是顺序执行的，也就是先执行第 1 条语句，然后是第 2 条、第 3 条……一直到最后一条语句，这称为顺序结构。

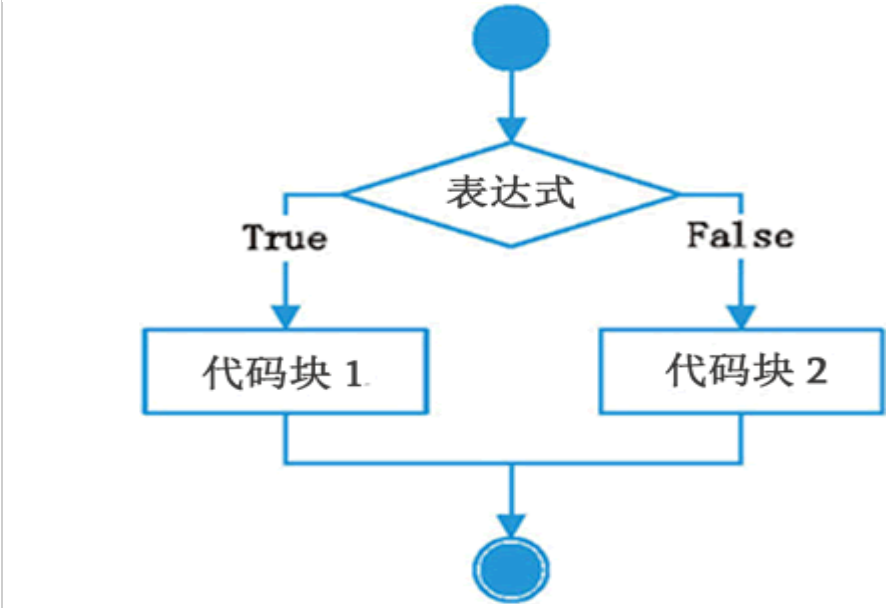
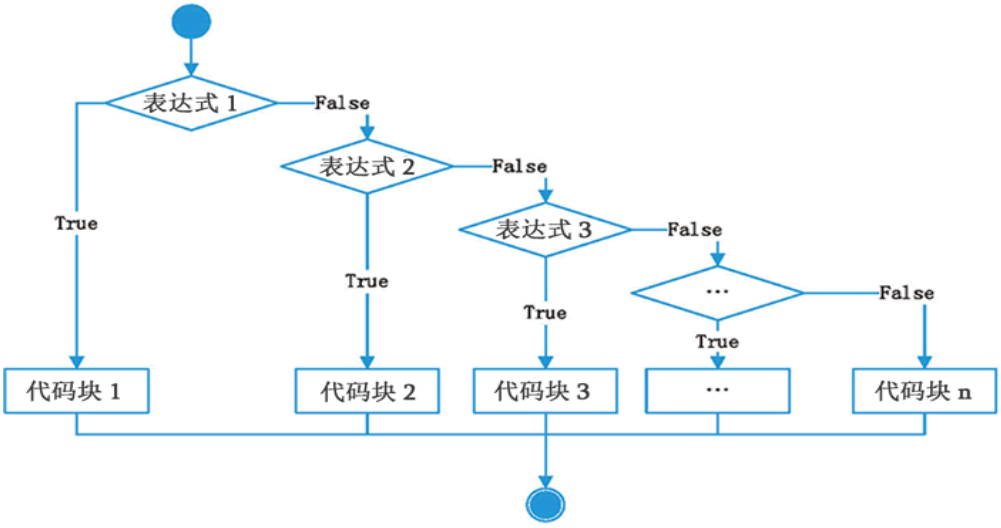
但是对于很多情况，顺序结构的代码是远远不够的，比如一个程序限制了只能成年人使用，儿童因为年龄不够，没有权限使用。这时候程序就需要做出判断，看用户是否是成年人，并给出提示。

在 Python 中，可以使用 if else 语句对条件进行判断，然后根据不同的结果执行不同的代码，这称为选择结构或者分支结构。

Python 中的 if else 语句可以细分为三种形式，分别是 if 语句、if else 语句和 if elif else 语句，它们的语法和执行流程如表 1 所示。

表 1 if else 分支语句的三种形式

语法格式	执行流程
if 表达式： 　代码块	

if 表达式： 　代码块 1 else： 　代码块 2	
if 表达式 1： 　代码块 1 elif 表达式 2： 　代码块 2 elif 表达式 3： 　代码块 3 ...//其它 elif 语句 else： 　代码块 n	

> 以上三种形式中，第二种和第三种形式是相通的，如果第三种形式中的 elif 块不出现，就变成了第二种形式。另外，elif 和 else 都不能单独使用，必须和 if 一起出现，并且要正确配对。

对语法格式的说明：

- “表达式”可以是一个单一的值或者变量，也可以是由运算符组成的复杂语句，形式不限，只要它能得到一个值就行。不管“表达式”的结果是什么类型，if else 都能判断它是否成立（真或者假）。
- “代码块”由具由相同缩进量的若干条语句组成。
- if、elif、else 语句的最后都有冒号:，不要忘记。

一旦某个表达式成立，Python 就会执行它后面对应的代码块；如果所有表达式都不成立，那就执行 else 后面的代码块；如果没有 else 部分，那就什么也不执行。

执行过程最简单的就是第一种形式——只有一个 if 部分。如果表达式成立（真），就执行后面的代码块；如果表达式不成立（假），就什么也不执行。

对于第二种形式，如果表达式成立，就执行 if 后面紧跟的代码块 1；如果表达式不成立，就执行 else 后面紧跟的代码块 2。

对于第三种形式，Python 会从上到下逐个判断表达式是否成立，一旦遇到某个成立的表达式，就执行后面紧跟的语句块；此时，剩下的代码就不再执行了，不管后面的表达式是否成立。如果所有的表达式都不成立，就执行 else 后面的代码块。

总起来说，不管有多少个分支，都只能执行一个分支，或者一个也不执行，不能同时执行多个分支。

【实例 1】使用第一种选择结构判断用户是否符合条件：

```
age = int( input("请输入你的年龄：") )

if age < 18 :
    print("你还未成年，建议在家人陪同下使用该软件！")
    print("如果你已经得到了家长的同意，请忽略以上提示。")

#该语句不属于 if 的代码块
print("软件正在使用中...")
```

运行结果 1：

```
请输入你的年龄：16↙
你还未成年，建议在家人陪同下使用该软件！
如果你已经得到了家长的同意，请忽略以上提示。
软件正在使用中...
```

运行结果 2：

```
请输入你的年龄：24↙
软件正在使用中...
```

从运行结果可以看出，如果输入的年龄小于 18，就执行 if 后面的语句块；如果输入的年龄大于等于 18，就不执行 if 后面的语句块。这里的语句块就是缩进四个空格的两个 print() 语句。

【实例 2】改进上面的代码，年龄不符合时退出程序：

```
import sys

age = int( input("请输入你的年龄：") )

if age < 18 :
    print("警告：你还未成年，不能使用该软件！")
    print("未成年人应该好好学习，读个好大学，报效祖国。")
    sys.exit()
else:
    print("你已经成年，可以使用该软件。")
    print("时间宝贵，请不要在该软件上浪费太多时间。")

print("软件正在使用中...")
```

运行结果 1：

```
请输入你的年龄：16↙
警告：你还未成年，不能使用该软件！
未成年人应该好好学习，读个好大学，报效祖国。
```

运行结果 2：

```
请输入你的年龄：20↙
你已经成年，可以使用该软件。
时间宝贵，请不要在该软件上浪费太多时间。
软件正在使用中...
```

sys 模块的 exit() 函数用于退出程序。

【实例 3】判断一个人的身材是否合理：

```
height = float(input("输入身高（米）："))
weight = float(input("输入体重（千克）："))
bmi = weight / (height * height)  #计算 BMI 指数

if bmi<18.5:
    print("BMI 指数为："+str(bmi))
```

```
7.      print("体重过轻")
8.  elif bmi>=18.5 and bmi<24.9:
9.      print("BMI 指数为：" +str(bmi))
10.     print("正常范围，注意保持")
11. elif bmi>=24.9 and bmi<29.9:
12.     print("BMI 指数为：" +str(bmi))
13.     print("体重过重")
14. else:
15.     print("BMI 指数为：" +str(bmi))
16.     print("肥胖")
```

运行结果：

```
输入身高（米）：1.7↙
输入体重（千克）：70↙
BMI 指数为：24.221453287197235
正常范围，注意保持
```

需要强调的是，Python 是一门非常独特的编程语言，它通过缩进来识别代码块，具有相同缩进量的若干行代码属于同一个代码块，所以你不能胡乱缩进，这样很容易导致语法错误。更多关于缩进的内容请转到《Python if else 对缩进的要求》。

> 在其他语言中（如 C 语言、C++、Java 等），选择结构还包括 switch 语句，也可以实现多重选择，但是在 Python 中没有 switch 语句，所以当要实现多重选择的功能时，只能使用 if else 分支语句。

if else 如何判断表达式是否成立

上面说过，if 和 elif 后面的“表达式”的形式是很自由的，只要表达式有一个结果，不管这个结果是什么类型，Python 都能判断它是“真”还是“假”。

布尔类型（bool）只有两个值，分别是 True 和 False，Python 会把 True 当做“真”，把 False 当做“假”。

对于数字，Python 会把 0 和 0.0 当做“假”，把其它值当做“真”。

对于其它类型，当对象为空或者为 None 时，Python 会把它们当做“假”，其它情况当做真。比如，下面的表达式都是不成立的：

```
"" #空字符串
[] #空列表
() #空元组
```

```
{} #空字典
None  #空值
```

【实例】if elif 判断各种类型的表达式：

```
b = False
if b:
    print('b 是 True')
else:
    print('b 是 False')

n = 0
if n:
    print('n 不是零值')
else:
    print('n 是零值')

s = ""
if s:
    print('s 不是空字符串')
else:
    print('s 是空字符串')

l = []
if l:
    print('l 不是空列表')
else:
    print('l 是空列表')

d = {}
if d:
    print('d 不是空字典')
else:
    print('d 是空字典')

def func():
    print("函数被调用")
```

```
33.
34. if func():
35.     print('func()返回值不是空')
36. else:
37.     print('func()返回值为空')
```

运行结果：

```
b 是 False
n 是零值
s 是空字符串
l 是空列表
d 是空字典
函数被调用
func()返回值为空
```

说明：对于没有 return 语句的函数，返回值为空，也即 None。

6.2 Python if else 对缩进的要求

前面的《Python if else》一节展示了选择结构的三种基本形式，并给出了实例演示，但是大家在编写代码过程中仍然要注意一些细节，尤其是代码块的缩进，这对 if else 选择结构极其重要。

Python 是以缩进来标记代码块的，代码块一定要有缩进，没有缩进的不是代码块。另外，同一个代码块的缩进量要相同，缩进量不同的不属于同一个代码块。

不要忘记缩进

if、elif 和 else 后面的代码块一定要缩进，而且缩进量要大于 if、elif 和 else 本身。例如，下面的代码就是一个反面教材：

```
age = int( input("请输入你的年龄：") )

if age < 18 :
print("警告：你还未成年，不能使用该软件！")
else:
print("你已经成年，可以使用该软件。")
```

本例中的 print() 函数和 if、else 语句是对齐的，在同一条竖线上，都没有缩进，所以 print() 就不是 if、else 的代码块了，这会导致 Python 解释器找不到 if、else 的代码块，从而报出如下错误：

```
SyntaxError: expected an indented block
```

翻译过来就是：

```
语法错误：需要一个缩进的代码块
```

总之，if、else 后面的代码一定要缩进，否则就不能构成 if、else 的执行体。

缩进多少合适？

Python 要求代码块必须缩进，但是却没有要求缩进量，你可以缩进 n 个空格，也可以缩进 n 个 Tab 键的位置。

但是从编程习惯的角度看，我建议缩进 1 个 Tab 键的位置，或者缩进 4 个空格；它们两者其实是等价的，很多编辑器都可以将 Tab 键设置为 4 个空格，比如，IDLE 中默认 Tab 键就是 4 个空格。

所有语句都要缩进

一个代码块的所有语句都要缩进，而且缩进量必须相同。如果某个语句忘记缩进了，Python 解释器并不一定会报错，但是程序的运行逻辑往往会有问题。请看下面的代码：

```
age = int( input("请输入你的年龄：") )

if age < 18 :
    print("你还未成年，建议在家人陪同下使用该软件！")
print("未成年人如果得到了家长的同意，请忽略以上提示。")  #忘记缩进
```

这段代码并没有语法错误，但是它的运行逻辑是不对的，比如，输入 16 的运行结果如下：

```
请输入你的年龄：24↙
未成年人如果得到了家长的同意，请忽略以上提示。
```

用户的年龄分明大于 18，但是却出现了“未成年人”的提示，画面非常尴尬，这是因为第二个 print() 语句没有缩进，if 没有把它和第一个 print() 语句当做同一个代码块，所以它不是 if 执行体的一部分。解决这个错误也很容易，让第二个 print() 缩进 4 个空格即可。

同一代码块缩进量要相同

Python 虽然不限制代码块的缩进量，你可以随意缩进 n 个空格，但是，同一个代码块内的所有语句都必须拥有相同的缩进量，不能一会缩进 2 个空格，一会缩进 4 个空格。下面的代码是一个反面教材：

```
age = int( input("请输入你的年龄：") )

if age < 18 :
    print("你还未成年，建议在家人陪同下使用该软件！")
      print("未成年人如果得到了家长的同意，请忽略以上提示。")  #缩进量不对
```

运行这段代码，Python 解释器会报出语法错误：

```
SyntaxError: unexpected indent
```

翻译过来就是：

```
语法错误：意外的缩进
```

这段代码中，第一个 print() 语句缩进了 4 个空格，第二个 print() 语句缩进了 6 个空格，缩进量不同导致它们不是同一个代码块。Python 会认为第一个 print() 语句是 if 的执行体，而第二个 print() 是一个意外的存在，

不知道该把它当做谁的代码块，所以解析失败，报错。

总之，位于同一个代码块中的所有语句必须拥有相同的缩进量，多一个空格或者少一个空格都不行。

不要随便缩进

另外需要注意的是，不需要使用代码块的地方千万不要缩进，一旦缩进就会产生一个代码块。下面的代码是一个反面教材：

```
1.  info = "Python 教程的网址是：http://c.biancheng.net/python/"
2.      print(info)
```

这两条简单的语句没有包含分支、循环、函数、类等结构，不应该使用缩进。

6.3Python if 语句嵌套（入门必读）

前面章节中，详细介绍了 3 种形式的条件语句，即 if、if else 和 if elif else，这 3 种条件语句之间可以相互嵌套。

例如，在最简单的 if 语句中嵌套 if else 语句，形式如下：

```
if 表达式 1：
    if 表示式 2：
        代码块 1
    else：
        代码块 2
```

再比如，在 if else 语句中嵌套 if else 语句，形式如下：

```
if 表示式 1：
    if 表达式 2：
        代码块 1
    else：
        代码块 2
else：
    if 表达式 3：
        代码块 3
    else：
        代码块 4
```

Python 中，if、if else 和 if elif else 之间可以相互嵌套。因此，在开发程序时，需要根据场景需要，选择合适的嵌套方案。需要注意的是，在相互嵌套时，一定要严格遵守不同级别代码块的缩进规范。

【实例】判断是否为酒后驾车
如果规定，车辆驾驶员的血液酒精含量小于 20mg/100ml 不构成酒驾；酒精含量大于或等于 20mg/100ml 为酒驾；酒精含量大于或等于 80mg/100ml 为醉驾。先编写 Python 程序判断是否为酒后驾车。

通过梳理思路，是否构成酒驾的界限值为 20mg/100ml；而在已确定为酒驾的范围（大于 20mg/100ml）中，是否构成醉驾的界限值为 80mg/100ml，整个代码执行流程应如图 1 所示。

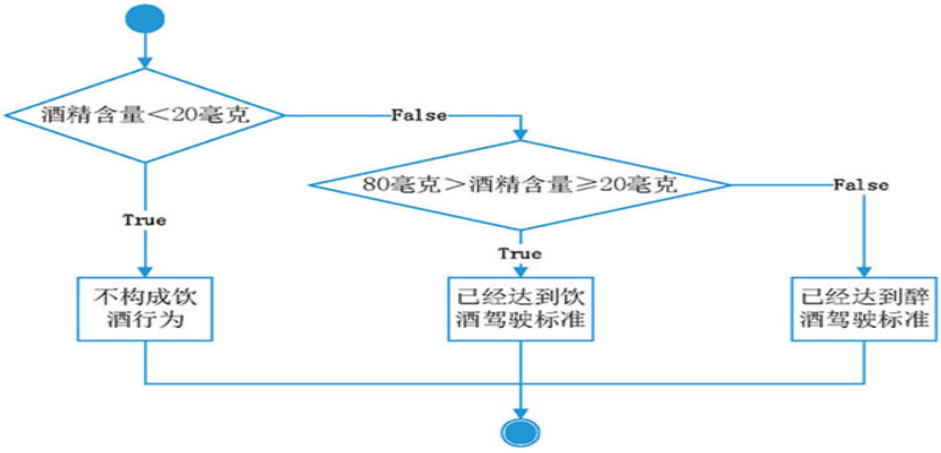


图 1 执行流程示意图

由此，我们可以使用两个 if else 语句嵌套来实现：

```
proof = int(input("输入驾驶员每 100ml 血液酒精的含量："))
if proof < 20:
    print("驾驶员不构成酒驾")
else:
    if proof < 80:
        print("驾驶员已构成酒驾")
    else:
        print("驾驶员已构成醉驾")
```

运行结果为：

```
输入驾驶员每 100ml 血液酒精的含量：10
驾驶员不构成酒驾
```

当然，这个例题单独使用 if elif else 也可以实现，这里只是为了让初学者熟悉 if 分支嵌套的用法而已。

除此之外，if 分支结构中还可以嵌套循环结构，同样，循环结构中也可以嵌套分支结构。不过，由于目前尚未系统学习循环结构，因此这部分知识会放到后续章节中作详细讲解。

6.4 Python pass 语句及其作用

在实际开发中，有时候我们会先搭建起程序的整体逻辑结构，但是暂时不去实现某些细节，而是在这些地方加一些注释，方面以后再添加代码，请看下面的例子：

```
age = int( input("请输入你的年龄：") )

if age < 12 :
    print("婴幼儿")
elif age >= 12 and age < 18:
    print("青少年")
elif age >= 18 and age < 30:
    print("成年人")
elif age >= 30 and age < 50:
    #TODO: 成年人
else:
    print("老年人")
```

当年龄大于等于 30 并且小于 50 时，我们没有使用 print() 语句，而是使用了一个注释，希望以后再处理成年人的情况。当 Python 执行到该 elif 分支时，会跳过注释，什么都不执行。

但是 Python 提供了一种更加专业的做法，就是空语句 pass。**pass** 是 Python 中的关键字，用来让解释器跳过此处，什么都不做。

就像上面的情况，有时候程序需要占一个位置，或者放一条语句，但又不希望这条语句做任何事情，此时就可以通过 pass 语句来实现。使用 pass 语句比使用注释更加优雅。

使用 pass 语句更改上面的代码：

```
age = int( input("请输入你的年龄：") )

if age < 12 :
    print("婴幼儿")
elif age >= 12 and age < 18:
    print("青少年")
elif age >= 18 and age < 30:
    print("成年人")
elif age >= 30 and age < 50:
    pass
else:
```

```
12.      print("老年人")
```

运行结果：

```
请输入你的年龄：40↙
```

从运行结果可以看出，程序虽然执行到第 10 行代码，但是并没有进行什么操作。

6.5 Python assert 断言函数及用法

Python assert 语句，又称断言语句，可以看做是功能缩小版的 if 语句，它用于判断某个表达式的值，如果值为真，则程序可以继续往下执行；反之，Python 解释器会报 AssertionError 错误。

assert 语句的语法结构为：

```
assert 表达式
```

assert 语句的执行流程可以用 if 判断语句表示，如下所示：

```
if 表达式==True:
    程序继续执行
else:
    程序报 AssertionError 错误
```

有读者可能会问，明明 assert 会令程序崩溃，为什么还要使用它呢？这是因为，与其让程序在晚些时候崩溃，不如在错误条件出现时，就直接让程序崩溃，这有利于我们对程序排错，提高程序的健壮性。

因此，assert 语句通常用于检查用户的输入是否符合规定，还经常用作程序初期测试和调试过程中的辅助工具。

下面的程序演示了 assert 语句的用法：

```
1. mathmark = int(input())
2. #断言数学考试分数是否位于正常范围内
3. assert 0 <= mathmark <= 100
4. #只有当 mathmark 位于 [0,100]范围内，程序才会继续执行
5. print("数学考试分数为：",mathmark)
```

运行该程序，测试数据如下：

```
90
数学考试分数为： 90
```

再次执行该程序，测试数据为：

```
159
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\file.py", line 3, in <module>
    assert 0 <= mathmark <= 100
AssertionError
```

可以看到，当 assert 语句后的表达式值为真时，程序继续执行；反之，程序停止执行，并报 AssertionError 错误。

6.6 Python 如何合理使用 assert（新手必读）

讲完了 assert 的基本语法之后，本节通过一些实际应用的例子，给大家演示一下 assert 在 Python 中的用法，并明确 assert 的使用场景。

第一个例子，假设 C 语言中文网想做 VIP 促销活动，准备进行打折，现需要写一个 apply_discount() 函数，要求是，向该函数传入原来的价格和折扣力度，该函数返回打折后的价格。

apply_discount() 大致应该写成如下这样：

```
#price 为原价，discount 为折扣力度
def apply_discount(price, discount):
    updated_price = price * (1 - discount)
    assert 0 <= updated_price <= price, '折扣价应在 0 和原价之间'
    return updated_price
```

可以看到，在计算新价格的后面，添加了一个 assert 语句，用来检查折后价格，这里要求新折扣价格必须大于等于 0、小于等于原来的价格，否则就抛出异常。

我们可以试着输入几组数，来验证一下这个功能：

```
print(apply_discount(100,0.2))
print(apply_discount(100,1.1))
```

运行结果为：

```
80.0
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print(apply_discount(100,1.1))
  File "C:\Users\mengma\Desktop\demo.py", line 4, in apply_discount
    assert 0 <= updated_price <= price, '折扣价应在 0 和原价之间'
AssertionError: 折扣价应在 0 和原价之间
```

可以看到，当 discount 是 0.2 时，输出 80 没有问题，但是当 discount 为 1.1 时，程序便抛出下面 AssertionError 异常。

这样一来，如果开发人员修改相关的代码，或者是加入新的功能，导致 discount 数值异常时，只要运行程序就很容易能发现问题，这也从侧面印证了前面多讲的，assert 的加入可以有效预防程序漏洞，提高程序的健壮性。

另外，在实际工作中，assert 还有一些很常见的用法，例如：

```
def func(input):
    assert isinstance(input, list), '输入内容必须是列表'
    # 下面的操作都是基于前提：input 必须是 list
```

```
    if len(input) == 1:
        ...
    elif len(input) == 2:
        ...
    else:
        ...
```

上面代码中，func() 函数中的所有操作都基于输入必须是列表这个前提。所以很有必要在开头加一句 assert 的检查，防止程序出错。

以上给大家介绍了 2 个有关 assert 的使用场景，很多读者可能觉得，assert 的作用和 if 语句非常接近，那么他们之间是否可以相互替代呢？

要注意，前面讲过，assert 的检查是可以被关闭的，比如在命令行模式下运行 Python 程序时，加入 -O 选项就可以使程序中的 assert 失效。一旦 assert 失效，其包含的语句也就不会被执行。

还是拿 C 语言中文网用户来说，只有 VIP 用户才可以阅读 VIP 文章，我们可以设计如下这个函数来模式判断用户身份的功能：

```
def login_user_identity(user_id):
    #凭借用户 id 判断该用户是否为 VIP 用户
    assert user_is_Vip(user_id) "用户必须是 VIP 用户，才能阅读 VIP 文章"
    read()
```

此代码从代码功能角度上看，并没有问题，但在实际场景中，基本上没人会这么写，因为一旦 assert 失效，则就造成任何用户都可以阅读 VIP 文章，这显然是不合理的。

所以正确的做法是，使用 if 条件语句替代 assert 语句进行相关的检查，并合理抛出异常：

```
def login_user_identity(user_id):
    #凭借用户 id 判断该用户是否为 VIP 用户
    if not user_is_Vip(user_id):
        raise Exception("用户必须是 VIP 用户，才能阅读 VIP 文章")
    read()
```

总之，不能滥用 assert，很多情况下，程序中出现的不同情况都是意料之中的，需要用不同的方案去处理，有时用条件语句进行判断更为合适，而对于程序中可能出现的一些异常，要记得用 try except 语句处理（后续章节会做详细介绍）。

6.7 Python while 循环语句详解

Python 中，while 循环和 if 条件分支语句类似，即在条件（表达式）为真的情况下，会执行相应的代码块。不同之处在于，只要条件为真，while 就会一直重复执行那段代码块。

while 语句的语法格式如下：

```
while 条件表达式：
    代码块
```

这里的代码块，指的是缩进格式相同的多行代码，不过在循环结构中，它又称为循环体。

while 语句执行的具体流程为：首先判断条件表达式的值，其值为真（True）时，则执行代码块中的语句，当执行完毕后，再回过头来重新判断条件表达式的值是否为真，若仍为真，则继续重新执行代码块...如此循环，直到条件表达式的值为假（False），才终止循环。

while 循环结构的执行流程如图 1 所示。

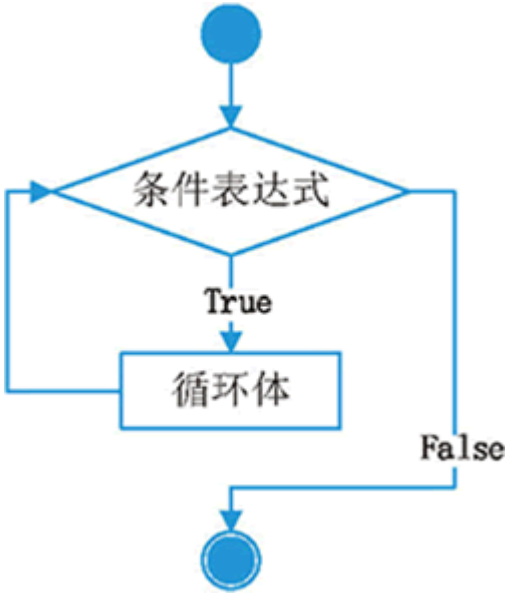


图 1 while 循环语句执行流程示意图

例如，打印 1~100 的所有数字，就可以使用 while 循环，实现代码如下：

```
# 循环的初始化条件
num = 1
# 当 num 小于 100 时，会一直执行循环体
while num < 100 :
    print("num=", num)
    # 迭代语句
```

```
    num += 1
print("循环结束!")
```

运行程序会发现，程序只输出了 1~99，却没有输出 100。这是因为，当循环至 num 的值为 100 时，此时条件表达式为假（100<100），当然就不会再去执行代码块中的语句，因此不会输出 100。

注意，在使用 while 循环时，一定要保证循环条件有变成假的时候，否则这个循环将成为一个死循环。所谓死循环，指的是无法结束循环的循环结构，例如将上面 while 循环中的 `num += 1` 代码注释掉，再运行程序你会发现，Python 解释器一直在输出"num= 1"，永远不会结束（因为 num<100 一直为 True），除非我们强制关闭解释器。

再次强调，只要位于 while 循环体中的代码，其必须使用相同的缩进格式（通常缩进 4 个空格），否则 Python 解释器会报 SyntaxError 错误（语法错误）。例如，将上面程序中 `num+=1` 语句前移一个空格，再次执行该程序，此时 Python 解释器就会报 SyntaxError 错误。

除此之外，while 循环还常用来遍历列表、元组和字符串，因为它们都支持通过下标索引获取指定位置的元素。例如，下面程序演示了如何使用 while 循环遍历一个字符串变量：

```
my_char="http://c.biancheng.net/python/"
i = 0;
while i<len(my_char):
    print(my_char[i],end="")
    i = i + 1
```

程序执行结果为：

```
http://c.biancheng.net/python/
```

6.8 Python for 循环及用法详解

Python 中的循环语句有 2 种，分别是 while 循环和 for 循环，前面章节已经对 while 做了详细的讲解，本节给大家介绍 for 循环，它常用于遍历字符串、列表、元组、字典、集合等序列类型，逐个获取序列中的各个元素。

for 循环的语法格式如下：

```
for 迭代变量 in 字符串|列表|元组|字典|集合：
    代码块
```

格式中，迭代变量用于存放从序列类型变量中读取出来的元素，所以一般不会在循环中对迭代变量手动赋值；代码块指的是具有相同缩进格式的多行代码（和 while 一样），由于和循环结构联用，因此代码块又称为循环体。

for 循环语句的执行流程如图 1 所示。

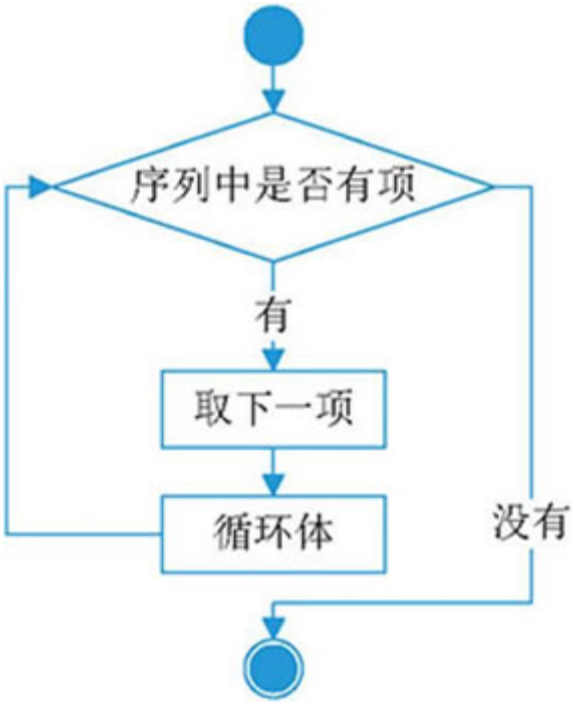


图 1 for 循环语句的执行流程图

下面的程序演示了 for 循环的具体用法：

```
1.  add = "http://c.biancheng.net/python/"
2.  #for 循环，遍历 add 字符串
3.  for ch in add:
4.      print(ch,end="")
```

运行结果为：

```
http://c.biancheng.net/python/
```

可以看到，使用 for 循环遍历 add 字符串的过程中，迭代变量 ch 会先后被赋值为 add 字符串中的每个字符，并代入循环体中使用。只不过例子中的循环体比较简单，只有一行输出语句。

Python for 循环的具体应用

for 循环进行数值循环

在使用 for 循环时，最基本的应用就是进行数值循环。比如说，想要实现从 1 到 100 的累加，可以执行如下代码：

```
print("计算 1+2+...+100 的结果为：")
#保存累加结果的变量
result = 0
#逐个获取从 1 到 100 这些值，并做累加操作
for i in range(101):
    result += i
print(result)
```

运行结果为：

```
计算 1+2+...+100 的结果为：
5050
```

上面代码中，使用了 range() 函数，此函数是 Python 内置函数，用于生成一系列连续整数，多用于 for 循环中。

有关 range() 函数的具体用法，可阅读《Python range()》一节，值得一提的是，Python 2.x 中除提供 range() 函数外，还提供了一个 xrange() 函数，它可以解决 range() 函数不经意间耗掉所有可用内存的问题。但在 Python 3.x 中，已经将 xrange() 更名为 range() 函数，并删除了老的 xrange() 函数。

for 循环遍历列表和元组

当用 for 循环遍历 list 列表或者 tuple 元组时，其迭代变量会先后被赋值为列表或元组中的每个元素并执行一次循环体。

下面程序使用 for 循环对列表进行了遍历：

```
my_list = [1,2,3,4,5]
for ele in my_list:
    print('ele =', ele)
```

程序执行结果为：

```
ele = 1
ele = 2
ele = 3
ele = 4
ele = 5
```

感兴趣的读者，可自行尝试用 for 循环遍历 tuple 元组，这里不再给出具体实例。

for 循环遍历字典

在使用 for 循环遍历字典时，经常会用到和字典相关的 3 个方法，即 items()、keys() 以及 values()，它们各自的用法已经在前面章节中讲过，这里不再赘述。当然，如果使用 for 循环直接遍历字典，则迭代变量会被先后赋值为每个键值对中的键。

例如：

```
my_dic = {'python 教程':"http://c.biancheng.net/python/",\
          'shell 教程':"http://c.biancheng.net/shell/",\
          'java 教程':"http://c.biancheng.net/java/"}
for ele in my_dic:
    print('ele =', ele)
```

程序执行结果为：

```
ele = python 教程
ele = shell 教程
ele = java 教程
```

因此，直接遍历字典，和遍历字典 keys() 方法的返回值是相同的。
除此之外，我们还可以遍历字典 values()、items() 方法的返回值。例如：

```
my_dic = {'python 教程':"http://c.biancheng.net/python/",\
          'shell 教程':"http://c.biancheng.net/shell/",\
          'java 教程':"http://c.biancheng.net/java/"}
for ele in my_dic.items():
    print('ele =', ele)
```

程序执行结果为：

```
ele = ('python 教程', 'http://c.biancheng.net/python/')
ele = ('shell 教程', 'http://c.biancheng.net/shell/')
ele = ('java 教程', 'http://c.biancheng.net/java/')
```

6.9 Python 循环结构中 else 用法（入门必读）

Python 中，无论是 while 循环还是 for 循环，其后都可以紧跟着一个 else 代码块，它的作用是当循环条件为 False 跳出循环时，程序会最先执行 else 代码块中的代码。

以 while 循环为例，下面程序演示了如何为 while 循环添加一个 else 代码块：

```
add = "http://c.biancheng.net/python/"
i = 0
while i < len(add):
    print(add[i],end="")
    i = i + 1
else:
    print("\n执行 else 代码块")
```

程序执行结果为：

```
http://c.biancheng.net/python/
执行 else 代码块
```

上面程序中，当 i==len(add) 结束循环时（确切的说，是在结束循环之前），Python 解释器会执行 while 循环后的 else 代码块。

有读者可能会觉得，else 代码块并没有什么具体作用，因为 while 循环之后的代码，即便不位于 else 代码块中，也会被执行。例如，修改上面程序，去掉 else 代码块：

```
add = "http://c.biancheng.net/python/"
i = 0
while i < len(add):
    print(add[i],end="")
    i = i + 1
#原本位于 else 代码块中的代码
print("\n执行 else 代码块")
```

程序执行结果为：

```
http://c.biancheng.net/python/
执行 else 代码块
```

那么，else 代码块真的没有用吗？当然不是。后续章节介绍 break 语句时，会具体介绍 else 代码块的用法。

当然，我们也可以为 for 循环添加一个 else 代码块，例如：

```
add = "http://c.biancheng.net/python/"
```

```
for i in  add:
    print(i,end="")
else:
    print("\n执行 else 代码块")
```

程序执行结果为：

```
http://c.biancheng.net/python/
执行 else 代码块
```

6.10 Python（for 和 while）循环嵌套及用法

Python 不仅支持 if 语句相互嵌套，while 和 for 循环结构也支持嵌套。所谓嵌套（Nest），就是一条语句里面还有另一条语句，例如 for 里面还有 for，while 里面还有 while，甚至 while 中有 for 或者 for 中有 while 也都是允许的。

当 2 个（甚至多个）循环结构相互嵌套时，位于外层的循环结构常简称为外层循环或外循环，位于内层的循环结构常简称为内层循环或内循环。

循环嵌套结构的代码，Python 解释器执行的流程为：

1. 当外层循环条件为 True 时，则执行外层循环结构中的循环体；
2. 外层循环体中包含了普通程序和内循环，当内层循环的循环条件为 True 时会执行此循环中的循环体，直到内层循环条件为 False，跳出内循环；
3. 如果此时外层循环的条件仍为 True，则返回第 2 步，继续执行外层循环体，直到外层循环的循环条件为 False；
4. 当内层循环的循环条件为 False，且外层循环的循环条件也为 False，则整个嵌套循环才算执行完毕。

循环嵌套的执行流程图如图 1 所示：

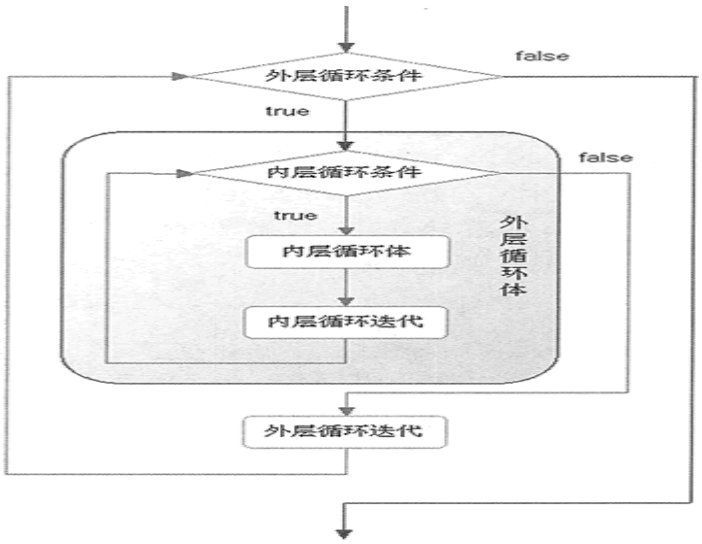


图 1 循环嵌套的执行流程图

下面程序演示了 while-for 嵌套结构：

```
i = 0
while i<10:
    for j in range(10):
        print("i=",i," j=",j)
    i=i+1
```

由于程序输出结果篇幅太长，读者可自行拷贝代码并执行，观察其执行结果。

可以看到，此程序中运用了嵌套循环结构，其中外循环使用的是 while 语句，而内循环使用的是 for 语句。程序执行的流程是：

- 一开始 i=0，循环条件 i<10 成立，进入 while 外循环执行其外层循环体；
- 从 j=0 开始，由于 j <10 成立，因此进入 for 内循环执行内层循环体，直到 j=10 不满足循环条件，跳出 for 循环体，继续执行 while 外循环的循环体；
- 执行 i=i+1 语句，如果 i<10 依旧成立，则从第 2 步继续执行。直到 i<10 不成立，则此循环嵌套结构才执行完毕。

根据上面的分析，此程序中外层循环将循环 9 次（从 i=1 到 i=9），而每次执行外层循环时，内层循环都从 j=0 循环执行到 j=9。因此，该嵌套循环结构将执行 9*10 = 90 次。

嵌套循环执行的总次数 = 外循环执行次数 * 内循环执行次数

事实上，if 语句和循环（while、for）结构之间，也可以相互嵌套，举个例子：

```
1.  i = 0
2.  if i<10:
3.      for j in range(5):
4.          print("i=", i, " j=", j)
```

程序执行结果为：

```
i= 0  j= 0
i= 0  j= 1
i= 0  j= 2
i= 0  j= 3
i= 0  j= 4
```

需要指明的是，上面程序演示的仅是 2 层嵌套结构，其实 if、while、for 之间完全支持多层（ ≥3 ）嵌套。例如：

```
if ...:
    while ...:
        for ...:
            if ...:
                ...
```

也就是说，只要场景需要，判断结构和循环结构之间完全可以相互嵌套，甚至可以多层嵌套。

6.11 Python 嵌套循环实现冒泡排序

冒泡排序是数据结构中的经典算法，手动实现冒泡排序，对初学者锻炼自己的编程逻辑有很大帮助，本节就带领大家使用循环结构实现冒泡排序算法。

冒泡排序算法的实现思想遵循以下几步：

1. 比较相邻的元素，如果第一个比第二个大，就交换它们两个。
2. 从最开始的第一对到结尾的最后一对，对每一对相邻元素做步骤 1 所描述的比较工作，并将最大的元素放在后面。这样，当从最开始的第一对到结尾的最后一对都执行完后，整个序列中的最后一个元素便是最大的数。
3. 将循环缩短，除去最后一个数（因为最后一个已经是最大的了），再重复步骤 2 的操作，得到倒数第二大的数。
4. 持续做步骤 3 的操作，每次将循环缩短一位，并得到本次循环中的最大数。直到循环个数缩短为 1，即没有任何一对数字需要比较，此时便得到了一个从小到大排序的序列。

通过分析冒泡排序算法的实现原理，要想实现该算法，需要借助循环结构，更确切地说，需要使用嵌套循环结构，使用 for 循环或者 while 循环都可以。

例如，使用 for 循环实现用冒泡排序算法对 [5,8,4,1] 进行排序：

```
data = [5,8,4,1]
#实现冒泡排序
for i in range(len(data)-1):
    for j in range(len(data)-i-1):
        if(data[j]>data[j+1]):
            data[j],data[j+1] = data[j+1],data[j]
print("排序后：",data)
```

运行结果为：

```
排序后： [1, 4, 5, 8]
```

可以看到，实现冒泡排序使用了 2 层循环，其中外层循环负责冒泡排序进行的次数，而内层循环负责将列表中相邻的两个元素进行比较，并调整顺序，即将较小的放在前面，较大的放在后面。

6.12 Python break 用法详解

我们知道，在执行 while 循环或者 for 循环时，只要循环条件满足，程序将会一直执行循环体，不停地转圈。但在某些场景，我们可能希望在循环结束前就强制结束循环，Python 提供了 2 种强制离开当前循环体的办法：

1. 使用 continue 语句，可以跳过执行本次循环体中剩余的代码，转而执行下一次的循环。
2. 只用 break 语句，可以完全终止当前循环。

本节先讲解 break 的用法，continue 语句放到下节做详细介绍。

break 语句可以立即终止当前循环的执行，跳出当前所在的循环结构。无论是 while 循环还是 for 循环，只要执行 break 语句，就会直接结束当前正在执行的循环体。

这就好比在操场上跑步，原计划跑 10 圈，可是当跑到第 2 圈的时候，突然想起有急事要办，于是果断停止跑步并离开操场，这就相当于使用了 break 语句提前终止了循环。

break 语句的语法非常简单，只需要在相应 while 或 for 语句中直接加入即可。例如如下程序：

```
add = "http://c.biancheng.net/python/,http://c.biancheng.net/shell/"
# 一个简单的 for 循环
for i in add:
    if i == ',' :
        #终止循环
        break
    print(i,end="")
print("\n 执行循环体外的代码")
```

运行结果为：

```
http://c.biancheng.net/python/
执行循环体外的代码
```

分析上面程序不难看出，当循环至 add 字符串中的逗号（，）时，程序执行 break 语句，其会直接终止当前的循环，跳出循环体。

break 语句一般会结合 if 语句进行搭配使用，表示在某种条件下跳出循环体。
注意，通过前面的学习我们知道，for 循环后也可以配备一个 else 语句。这种情况下，如果使用 break 语句跳出循环体，不会执行 else 中包含的代码。举个例子：

```
add = "http://c.biancheng.net/python/,http://c.biancheng.net/shell/"
for i in add:
    if i == ',' :
        #终止循环
```

```
        break
    print(i,end="")
else:
    print("执行 else 语句中的代码")
print("\n 执行循环体外的代码")
```

程序执行结果为：

```
http://c.biancheng.net/python/
执行循环体外的代码
```

从输出结果可以看出，使用 break 跳出当前循环体之后，该循环后的 else 代码块也不会被执行。但是，如果将 else 代码块中的代码直接放在循环体的后面，则该部分代码将会被执行。

另外，对于嵌套的循环结构来说，break 语句只会终止所在循环体的执行，而不会作用于所有的循环体。举个例子：

```
add = "http://c.biancheng.net/python/,http://c.biancheng.net/shell/"
for i in range(3):
    for j in add:
        if j == ',':
            break
        print(j,end="")
    print("\n 跳出内循环")
```

程序执行结果为：

```
http://c.biancheng.net/python/
跳出内循环
http://c.biancheng.net/python/
跳出内循环
http://c.biancheng.net/python/
跳出内循环
```

分析上面程序，每当执行内层循环时，只要循环至 add 字符串中的逗号（，）就会执行 break 语句，它会立即停止执行当前所在的内存循环体，转而继续执行外层循环。

那么读者可能会问，在嵌套循环结构中，如何同时跳出内层循环和外层循环呢？最简单的方法就是借用一个 bool 类型的变量。

修改上面的程序：

```
add = "http://c.biancheng.net/python/,http://c.biancheng.net/shell/"
```

```
#提前定义一个 bool 变量，并为其赋初值
flag = False
for i in range(3):
    for j in add:
        if j == ',':
            #在 break 前，修改 flag 的值
            flag = True
            break
        print(j,end="")
    print("\n跳出内循环")
    #在外层循环体中再次使用 break
    if flag == True:
        print("跳出外层循环")
        break
```

可以看到，通过借助一个 bool 类型的变量 flag，在跳出内循环时更改 flag 的值，同时在外层循环体中，判断 flag 的值是否发生改动，如有改动，则再次执行 break 跳出外层循环；反之，则继续执行外层循环。

因此，上面程序的执行结果为：

```
http://c.biancheng.net/python/
跳出内循环
跳出外层循环
```

当然，这里仅跳出了 2 层嵌套循环，此方法支持跳出多层嵌套循环。

6.13 Python continue 的用法

和 break 语句相比，continue 语句的作用则没有那么强大，它只会终止执行本次循环中剩下的代码，直接从下一次循环继续执行。

仍然以在操作跑步为例，原计划跑 10 圈，但当跑到 2 圈半的时候突然接到一个电话，此时停止了跑步，当挂断电话后，并没有继续跑剩下的半圈，而是直接从第 3 圈开始跑。

continue 语句的用法和 break 语句一样，只要 while 或 for 语句中的相应位置加入即可。例如：

```
1.  add = "http://c.biancheng.net/python/,http://c.biancheng.net/shell/"
2.  # 一个简单的 for 循环
3.  for i in add:
4.      if i == ',' :
5.          # 忽略本次循环的剩下语句
6.          print('\n')
7.          continue
8.      print(i,end="")
```

运行上面程序，将看到如下运行结果：

```
http://c.biancheng.net/python/
http://c.biancheng.net/shell/
```

可以看到，当遍历 add 字符串至逗号（，）时，会进入 if 判断语句执行 print() 语句和 continue 语句。其中，print() 语句起到换行的作用，而 continue 语句会使 Python 解释器忽略执行第 8 行代码，直接从下一次循环开始执行。

6.14 怎么避免 Python 程序出现死循环（无限循环）？

要知道，每个循环结构（while 或 for）都必须有停止运行的途径，这样才不会没完没了地执行下去。

例如，下面的循环从 1 数到 5：

```
1.  x = 1
2.  while x <= 5:
3.      print(x)
4.      x += 1
```

运行结果为：

```
1
2
3
4
5
```

但如果像下面这样不小心遗漏了代码行 x += 1，那么这个循环将没完没了地运行：

```
1.  #这个循环将没完没了地运行
2.  x = 1
3.  while x <= 5:
4.      print(x)
```

在这里，x 的初始值为 1，但根本不会变，因此条件测试 x <= 5 始终为 True，导致 while 循环没完没了地打印 1，运行结果如下所示：

```
1
1
1
1
--snip--
```

每个程序员都会偶尔因为不小心而编写出无限（死）循环，在循环的退出条件比较微妙时尤其如此。

如果程序陷入无限循环，可按组合键 Ctrl+C，也可关闭显示程序输出的终端窗口，如果前 2 种方式都无效，比如有些编辑器（如 Sublime Text）内嵌了输出窗口，这可能导致难以结束无限循环，因此不得不关闭编辑器来结束无限循环。

读者可能会问，怎样才能有效避免编写出包含无限循环的程序呢？

要避免编写无限（死）循环，应务必对每个 while 循环（或 for 循环）进行测试，确保它按预期那样结束。如果希望程序在用户输入特定值时结束，可运行程序并输入这样的值，如果此时程序没有结束，则说明该循环结构会无限循环，应检查程序处理这个值的方式。

总之，解决无限（死）循环的方法只有一个，即确认循环结构中至少有一个位置能让循环条件为 False 或让 break 语句得以执行。

6.15 Python 推导式（列表推导式、元组推导式、字典推导式和集合推导式）详解

推导式（又称解析器），是 Python 独有的一种特性。使用推导式可以快速生成列表、元组、字典以及集合类型的数据，因此推导式又可细分为列表推导式、元组推导式、字典推导式以及集合推导式。

Python 列表推导式

列表推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的列表。

列表推导式的语法格式如下：

```
[表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] ]
```

此格式中，[if 条件表达式] 不是必须的，可以使用，也可以省略。

通过列表推导式的语法格式，明显会感觉到它和 for 循环存在某些关联。其实，除去 [if 条件表达式] 部分，其余各部分的含义以及执行顺序和 for 循环是完全一样的（表达式其实就是 for 循环中的循环体），即它的执行顺序如下所示：

```
for 迭代变量 in 可迭代对象
    表达式
```

初学者可以这样认为，它只是对 for 循环语句的格式做了一下简单的变形，并用 [] 括起来而已，只不过最大的不同之处在于，列表推导式最终会将循环过程中，计算表达式得到的一系列值组成一个列表。

例如如下代码（程序一）：

```
1.  a_range = range(10)
2.  # 对 a_range 执行 for 表达式
3.  a_list = [x * x for x in a_range]
4.  # a_list 集合包含 10 个元素
5.  print(a_list)
```

上面代码的第 3 行会对 a_range 执行迭代，由于 a_range 相当于包含 10 个元素，因此程序生成的 a_list 同样包含 10 个元素，且每个元素都是 a_range 中每个元素的平方（由表达式 x*x 控制）。

运行上面代码，可以看到如下输出结果：

```
[0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64, 81]
```

不仅如此，我们还可以在列表推导式中添加 if 条件语句，这样列表推导式将只迭代那些符合条件的元素。例如如下代码：

```
b_list = [x * x for x in a_range if x % 2 == 0]
# a_list 集合包含 5 个元素
print(b_list)
```

第一行代码与程序一中第 3 行代码大致相同，只是为这里给列表推导式增加了 if 条件语句，这会导致推导式只处理 range 区间的偶数，因此程序生成的 b_list 只包含 5 个元素。

运行上面代码，可以看到如下输出结果：

```
[0 ,4 , 16, 36, 64]
```

另外，以上所看到的列表推导式都只有一个循环，实际上它可使用多个循环，就像嵌套循环一样。例如如下代码：

```
d_list = [(x, y) for x in range(5) for y in range(4)]
# d_list 列表包含 20 个元素
print(d_list)
```

上面代码中，x 是遍历 range(5) 的迭代变量（计数器），因此该 x 可迭代 5 次；y 是遍历 range(4) 的计数器，因此该 y 可迭代 4 次。因此，该（x,y）表达式一共会迭代 20 次。上面的 for 表达式相当于如下嵌套循环：

```
dd_list = []
for x in range(5):
    for y in range(4):
        dd_list.append((x, y))
```

运行上面代码，可以看到如下输出结果：

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3), (4, 0), (4, 1), (4, 2), (4, 3)]
```

当然，也支持类似于三层嵌套的 for 表达式，例如如下代码：

```
e_list = [[x, y, z] for x in range(5) for y in range(4) for z in range(6)]
# e_list 列表包含 120 个元素
print(e_list)
```

对于包含多个循环的 for 表达式，同样可指定 if 条件。假如我们有一个需求：程序要将两个列表中的数值按“能否整除”的关系配对在一起。比如 src_a 列表中包含 30，src_b 列表中包含 5，其中 30 可以整除 5，那么就将 30 和 5 配对在一起。对于上面的需求使用 for 表达式来实现非常简单，例如如下代码：

```
src_a = [30, 12, 66, 34, 39, 78, 36, 57, 121]
src_b = [3, 5, 7, 11]
# 只要 y 能整除 x，就将它们配对在一起
result = [(x, y) for x in src_b for y in src_a if y % x == 0]
print(result)
```

运行上面代码，可以看到如下输出结果：

```
[(3, 30), (3, 12), (3, 66), (3, 39), (3, 78), (3, 36), (3, 57), (5, 30), (11, 66), (11, 121)]
```

Python 元组推导式

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

元组推导式的语法格式如下：

```
(表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] )
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

通过和列表推导式做对比，你会发现，除了元组推导式是用 () 圆括号将各部分括起来，而列表推导式用的是 []，其它完全相同。不仅如此，元组推导式和列表推导式的用法也完全相同。

例如，我们可以使用下面的代码生成一个包含数字 1~9 的元组：

```
a = (x for x in range(1,10))
print(a)
```

运行结果为：

```
<generator object <genexpr> at 0x0000020BAD136620>
```

从上面的执行结果可以看出，使用元组推导式生成的结果并不是一个元组，而是一个生成器对象（后续会介绍），这一点和列表推导式是不同的。

如果我们想要使用元组推导式获得新元组或新元组中的元素，有以下三种方式：

1. 使用 tuple() 函数，可以直接将生成器对象转换成元组，例如：

```
a = (x for x in range(1,10))
```

```
print(tuple(a))
运行结果为：
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

2. 直接使用 for 循环遍历生成器对象，可以获得各个元素，例如：

```
a = (x for x in range(1,10))
for i in a:
    print(i,end=' ')
print(tuple(a))
```

运行结果为：

```
1 2 3 4 5 6 7 8 9 ()
```

3. 使用 __next__() 方法遍历生成器对象，也可以获得各个元素，例如：

```
a = (x for x in range(3))
print(a.__next__())
print(a.__next__())
print(a.__next__())
a = tuple(a)
print("转换后的元组：",a)
```

运行结果为：

```
0
1
2
转换后的元组： ()
```

注意，无论是使用 for 循环遍历生成器对象，还是使用 __next__() 方法遍历生成器对象，遍历后原生成器对象将不复存在，这就是遍历后转换原生成器对象却得到空元组的原因。

Python 字典推导式

Python 中，使用字典推导式可以借助列表、元组、字典、集合以及 range 区间，快速生成符合需求的字典。

字典推导式的语法格式如下：

```
{表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]}
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

可以看到，和其它推导式的语法格式相比，唯一不同在于，字典推导式用的是大括号{}。

【例 1】

```
listdemo = ['C 语言中文网','c.biancheng.net']
#将列表中各字符串值为键，各字符串的长度为值，组成键值对
newdict = {key:len(key) for key in listdemo}
print(newdict)
```

运行结果为：

```
{'C 语言中文网': 6, 'c.biancheng.net': 15}
```

【例 2】交换现有字典中各键值对的键和值。

```
olddict={'C 语言中文网': 6, 'c.biancheng.net': 15}
newdict = {v: k for k, v in olddict.items()}
print(newdict)
```

运行结果为：

```
{6: 'C 语言中文网', 15: 'c.biancheng.net'}
```

【例 3】使用 if 表达式筛选符合条件的键值对。

```
olddict={'C 语言中文网': 6, 'c.biancheng.net': 15}
newdict = {v: k for k, v in olddict.items() if v>10}
print(newdict)
```

运行结果为：

```
{15: 'c.biancheng.net'}
```

Python 集合推导式

Python 中，使用集合推导式可以借助列表、元组、字典、集合以及 range 区间，快速生成符合需求的集合。

集合推导式的语法格式和字典推导式完全相同，如下所示：

```
{ 表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] }
```

其中，用 [] 括起来的部分，可以使用，也可以省略。

有读者可能会问，集合推导式和字典推导式的格式完全相同，那么给定一个类似的推导式，如何判断是哪种推导式呢？最简单直接的方式，就是根据表达式进行判断，如果表达式以键值对（key：value）的形式，则证明此推导式是字典推导式；反之，则是集合推导式。

【例 1】

```
1. setnew = {i**2 for i in range(3)}
2. print(setnew)
```

运行结果为：

```
{0, 1, 4}
```

【例 2】既然生成的是集合，那么其保存的元素必须是唯一的。

```
1. tupledemo = (1, 1, 2, 3, 4, 5, 6, 6)
2. setnew = {x**2 for x in tupledemo if x%2==0}
3. print(setnew)
```

运行结果为:

```
{16, 4, 36}
```

【例 3】

```
1. dictdemo = {'1':1,'2':2,'3':3}
2. setnew = {x for x in dictdemo.keys()}
3. print(setnew)
```

运行结果为：

```
{'2', '1', '3'}
```

6.16 Python zip 函数及用法

zip() 函数是 Python 内置函数之一，它可以将多个序列（列表、元组、字典、集合、字符串以及 range() 区间构成的列表）“压缩”成一个 zip 对象。所谓“压缩”，其实就是将这些序列中对应位置的元素重新组合，生成一个个新的元组。

和 Python 3.x 版本不同，Python 2.x 版本中的 zip() 函数会直接返回列表，而不是返回 zip 对象。但是，返回的列表或者 zip 对象，其包含的元素（都是元组）是相同的。

zip() 函数的语法格式为：

```
zip(iterable, ...)
```

其中 iterable,... 表示多个列表、元组、字典、集合、字符串，甚至还可以为 range() 区间。

下面程序演示了 zip() 函数的基本用法：

```
1.  my_list = [11,12,13]
2.  my_tuple = (21,22,23)
3.
4.  print([x for x in zip(my_list,my_tuple)])
5.
6.  my_dic = {31:2,32:4,33:5}
7.  my_set = {41,42,43,44}
8.
9.  print([x for x in zip(my_dic)])
10.
11. my_pychar = "python"
12. my_shechar = "shell"
13.
14. print([x for x in zip(my_pychar,my_shechar)])
```

程序执行结果为：

```
[(11, 21), (12, 22), (13, 23)]
[(31,), (32,), (33,)]
[('p', 's'), ('y', 'h'), ('t', 'e'), ('h', 'l'), ('o', 'l')]
```

如果读者分析以上的程序和相应的输出结果不难发现，在使用 zip() 函数“压缩”多个序列时，它会分别取各序列中第 1 个元素、第 2 个元素、... 第 n 个元素，各自组成新的元组。需要注意的是，当多个序列中元素个数不一致时，会以最短的序列为准进行压缩。

另外，对于 zip() 函数返回的 zip 对象，既可以像上面程序那样，通过遍历提取其存储的元组，也可以向下面程序这样，通过调用 list() 函数将 zip() 对象强制转换成列表：

```
my_list = [11,12,13]
my_tuple = (21,22,23)

print(list(zip(my_list,my_tuple)))
```

程序执行结果为：

```
[(11, 21), (12, 22), (13, 23)]
```

6.17 Python reversed 函数及用法

reserved() 是 Pyton 内置函数之一，其功能是对于给定的序列（包括列表、元组、字符串以及 range(n) 区间），该函数可以返回一个逆序序列的迭代器（用于遍历该逆序序列）。

reserved() 函数的语法格式如下：

```
reversed(seq)
```

其中，seq 可以是列表，元素，字符串以及 range() 生成的区间列表。

下面程序演示了 reversed() 函数的基本用法：

```
#将列表进行逆序
print([x for x in reversed([1,2,3,4,5])])

#将元组进行逆序
print([x for x in reversed((1,2,3,4,5))])

#将字符串进行逆序
print([x for x in reversed("abcdefg")])

#将 range() 生成的区间列表进行逆序
print([x for x in reversed(range(10))])
```

程序执行结果为：

```
[5, 4, 3, 2, 1]
[5, 4, 3, 2, 1]
['g', 'f', 'e', 'd', 'c', 'b', 'a']
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

除了使用列表推导式的方式，还可以使用 list() 函数，将 reversed() 函数逆序返回的迭代器，直接转换成列表。例如：

```
#将列表进行逆序
print(list(reversed([1,2,3,4,5])))
```

程序执行结果为：

```
[5, 4, 3, 2, 1]
```

再次强调，使用 reversed() 函数进行逆序操作，并不会修改原来序列中元素的顺序，例如：

```
a = [1,2,3,4,5]
#将列表进行逆序
print(list(reversed(a)))
print("a=",a)
```

程序执行结果为：

```
[5, 4, 3, 2, 1]
a= [1, 2, 3, 4, 5]
```

6.18 Python sorted 函数及用法

sorted() 作为 Python 内置函数之一，其功能是对序列（列表、元组、字典、集合、还包括字符串）进行排序。

sorted() 函数的基本语法格式如下：

```
list = sorted(iterable, key=None, reverse=False)
```

其中，iterable 表示指定的序列，key 参数可以自定义排序规则；reverse 参数指定以升序（False，默认）还是降序（True）进行排序。sorted() 函数会返回一个排好序的列表。

注意，key 参数和 reverse 参数是可选参数，即可以使用，也可以忽略。
下面程序演示了 sorted() 函数的基本用法：

```
#对列表进行排序
a = [5,3,4,2,1]
print(sorted(a))

#对元组进行排序
a = (5,4,3,1,2)
print(sorted(a))

#字典默认按照 key 进行排序
a = {4:1,\
     5:2,\
     3:3,\
     2:6,\
     1:8}
print(sorted(a.items()))

#对集合进行排序
a = {1,5,3,2,4}
print(sorted(a))

#对字符串进行排序
a = "51423"
print(sorted(a))
```

程序执行结果为：

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[(1, 8), (2, 6), (3, 3), (4, 1), (5, 2)]
[1, 2, 3, 4, 5]
['1', '2', '3', '4', '5']
```

再次强调，使用 sorted() 函数对序列进行排序， 并不会在原序列的基础进行修改，而是会重新生成一个排好序的列表。例如：

```
#对列表进行排序
a = [5,3,4,2,1]
print(sorted(a))
#再次输出原来的列表 a
print(a)
```

程序执行结果为：

```
[1, 2, 3, 4, 5]
[5, 3, 4, 2, 1]
```

显然，sorted() 函数不会改变所传入的序列，而是返回一个新的、排序好的列表。

除此之外，sorted( ) 函数默认对序列中元素进行升序排序，通过手动将其 reverse 参数值改为 True，可实现降序排序。例如：

```
#对列表进行排序
a = [5,3,4,2,1]
print(sorted(a,reverse=True))
```

程序执行结果为：

```
[5, 4, 3, 2, 1]
```

另外在调用 sorted() 函数时，还可传入一个 key 参数，它可以接受一个函数，该函数的功能是指定 sorted() 函数按照什么标准进行排序。例如：

```
chars=['http://c.biancheng.net',\
       'http://c.biancheng.net/python/',\
       'http://c.biancheng.net/shell/',\
       'http://c.biancheng.net/java/',\
       'http://c.biancheng.net/golang/']
```

```
#默认排序
print(sorted(chars))

#自定义按照字符串长度排序
print(sorted(chars,key=lambda x:len(x)))
```

程序执行结果为：

```
['http://c.biancheng.net',
 'http://c.biancheng.net/golang/',
 'http://c.biancheng.net/java/',
 'http://c.biancheng.net/python/',
 'http://c.biancheng.net/shell/']
['http://c.biancheng.net',
 'http://c.biancheng.net/java/',
 'http://c.biancheng.net/shell/',
 'http://c.biancheng.net/python/',
 'http://c.biancheng.net/golang/']
```

此程序中，使用了 lambda 表示式，其用法会在后续章节进行详细介绍。

第 7 章：函数和 lambda 表达式

7.1 Python 函数（函数定义、函数调用）用法详解

Python 中函数的应用非常广泛，前面章节中我们已经接触过多个函数，比如 input() 、print()、range()、len() 函数等等，这些都是 Python 的内置函数，可以直接使用。

除了可以直接使用的内置函数外，Python 还支持自定义函数，即将一段有规律的、可重复使用的代码定义成函数，从而达到一次编写、多次调用的目的。

举个例子，前面学习了 len() 函数，通过它我们可以直接获得一个字符串的长度。我们不妨设想一下，如果没有 len() 函数，要想获取一个字符串的长度，该如何实现呢？请看下面的代码：

```
1.  n=0
2.  for c in "http://c.biancheng.net/python/":
3.      n = n + 1
4.  print(n)
```

程序执行结果为：

```
30
```

要知道，获取一个字符串长度是常用的功能，一个程序中就可能用到很多次，如果每次都写这样一段重复的代码，不但费时费力、容易出错，而且交给别人时也很麻烦。

所以 Python 提供了一个功能，即允许我们将常用的代码以固定的格式封装（包装）成一个独立的模块，只要知道这个模块的名字就可以重复使用它，这个模块就叫做函数（Function）。

比如，在程序中定义了一段代码，这段代码用于实现一个特定的功能。问题来了，如果下次需要实现同样的功能，难道要把前面定义的代码复制一次？如果这样做实在太傻了，这意味着每次当程序需要实现该功能时，都要将前面定义的代码复制一次。正确的做法是，将实现特定功能的代码定义成一个函数，每次当程序需要实现该功能时，只要执行（调用）该函数即可。

其实，函数的本质就是一段有特定功能、可以重复使用的代码，这段代码已经被提前编写好了，并且为其起一个“好听”的名字。在后续编写程序过程中，如果需要同样的功能，直接通过起好的名字就可以调用这段代码。

下面演示了如何将我们自己实现的 len() 函数封装成一个函数：

```
1.  #自定义 len() 函数
2.  def my_len(str):
3.      length = 0
4.      for c in str:
5.          length = length + 1
```

```
    return length
#调用自定义的 my_len() 函数
length = my_len("http://c.biancheng.net/python/")
print(length)

#再次调用 my_len() 函数
length = my_len("http://c.biancheng.net/shell/")
print(length)
```

程序执行结果为：

```
30
29
```

如果读者接触过其他编程语言中的函数，以上对于函数的描述，肯定不会陌生。但需要注意的一点是，和其他编程语言中函数相同的是，Python 函数支持接收多个（ ≥0 ）参数，不同之处在于，Python 函数还支持返回多个（ ≥0 ）值。

比如，上面程序中，我们自己封装的 my_len(str) 函数，在定义此函数时，我们为其设置了 1 个 str 参数，同时该函数经过内部处理，会返回给我们 1 个 length 值。

通过分析 my_len() 函数这个实例不难看出，函数的使用大致分为 2 步，分别是定义函数和调用函数。接下来一一为读者进行详细的讲解。

Python 函数的定义

定义函数，也就是创建一个函数，可以理解为创建一个具有某些用途的工具。定义函数需要用 def 关键字实现，具体的语法格式如下：

```
def 函数名(参数列表):
    //实现特定功能的多行代码
    [return [返回值]]
```

其中，用 [] 括起来的为可选择部分，即可以使用，也可以省略。

此格式中，各部分参数的含义如下：

- 函数名：其实就是一个符合 Python 语法的标识符，但不建议读者使用 a、b、c 这类简单的标识符作为函数名，函数名最好能够体现出该函数的功能（如上面的 my_len，即表示我们自定义的 len() 函数）。
- 形参列表：设置该函数可以接收多少个参数，多个参数之间用逗号（ , ）分隔。

- [return [返回值]]：整体作为函数的可选参参数，用于设置该函数的返回值。也就是说，一个函数，可以用返回值，也可以没有返回值，是否需要根据实际情况而定。

> 注意，在创建函数时，即使函数不需要参数，也必须保留一对空的“()”，否则 Python 解释器将提示“invaild syntax”错误。另外，如果想定义一个没有任何功能的空函数，可以使用 pass 语句作为占位符。

例如，下面定义了 2 个函数：

```
#定义个空函数，没有实际意义
def pass_dis():
    pass
#定义一个比较字符串大小的函数
def str_max(str1,str2):
    str = str1 if str1 > str2 else str2
    return str
```

虽然 Python 语言允许定义个空函数，但空函数本身并没有实际意义。

另外值得一提的是，函数中的 return 语句可以直接返回一个表达式的值，例如修改上面的 str_max() 函数：

```
def str_max(str1,str2):
    return str1 if str1 > str2 else str2
```

该函数的功能，和上面的 str_max() 函数是完全一样的，只是省略了创建 str 变量，因此函数代码更加简洁。

Python 函数的调用

调用函数也就是执行函数。如果把创建的函数理解为一个具有某种用途的工具，那么调用函数就相当于使用该工具。

函数调用的基本语法格式如下所示：

```
[返回值] = 函数名([形参值])
```

其中，函数名即指的是要调用的函数的名称；形参值指的是当初创建函数时要求传入的各个形参的值。如果该函数有返回值，我们可以通过一个变量来接收该值，当然也可以不接受。

需要注意的是，创建函数有多少个形参，那么调用时就需要传入多少个值，且顺序必须和创建函数时一致。即便该函数没有参数，函数名后的小括号也不能省略。
例如，我们可以调用上面创建的 pass_dis() 和 str_max() 函数：

```
pass_dis()
```

```
strmax = str_max("http://c.biancheng.net/python","http://c.biancheng.net/shell");
print(strmax)
```

首先，对于调用空函数来说，由于函数本身并不包含任何有价值的执行代码，也没有返回值，应该调用空函数不会有任何效果。

其次，对于上面程序中调用 str_max() 函数，由于当初定义该函数为其设置了 2 个参数，因此这里在调用该参数，就必须传入 2 个参数。同时，由于该函数内部还使用了 return 语句，因此我们可以使用 strmax 变量来接收该函数的返回值。

因此，程序执行结果为：

```
http://c.biancheng.net/shell
```

为函数提供说明文档

前面章节讲过，通过调用 Python 的 help() 内置函数或者 __doc__ 属性，我们可以查看某个函数的使用说明文档。事实上，无论是 Python 提供给我们的函数，还是自定义的函数，其说明文档都需要设计该函数的程序员自己编写。

其实，函数的说明文档，本质就是一段字符串，只不过作为说明文档，字符串的放置位置是有讲究的，函数的说明文档通常位于函数内部、所有代码的最前面。

以上面程序中的 str_max() 函数为例，下面演示了如何为其设置说明文档：

```
#定义一个比较字符串大小的函数
def str_max(str1,str2):
    '''
    比较 2 个字符串的大小
    '''
    str = str1 if str1 > str2 else str2
    return str
help(str_max)
#print(str_max.__doc__)
```

程序执行结果为：

```
Help on function str_max in module __main__:

str_max(str1, str2)
    比较 2 个字符串的大小
```

上面程序中，还可以使用 __doc__ 属性来获取 str_max() 函数的说明文档，即使用最后一行的输出语句，其输出结果为：

```
比较 2 个字符串的大小
```

7.2 Python 函数值传递和引用传递（包括形式参数和实际参数的区别）

通常情况下，定义函数时都会选择有参数的函数形式，函数参数的作用是传递数据给函数，令其对接收的数据做具体的操作处理。

在使用函数时，经常会用到形式参数（简称“形参”）和实际参数（简称“实参”），二者都叫参数，之间的区别是：

- 形式参数：在定义函数时，函数名后面括号中的参数就是形式参数，例如:

```
#定义函数时，这里的函数参数 obj 就是形式参数
def demo(obj)
    print(obj)
```

- 实际参数：在调用函数时，函数名后面括号中的参数称为实际参数，也就是函数的调用者给函数的参数。例如:

```
a = "C 语言中文网"
#调用已经定义好的 demo 函数，此时传入的函数参数 a 就是实际参数
demo(a)
```

> 实参和形参的区别，就如同剧本选主角，剧本中的角色相当于形参，而演角色的演员就相当于实参。

明白了什么是形参和实参后，再来想一个问题，那就是实参是如何传递给形参的呢？

Python 中，根据实际参数的类型不同，函数参数的传递方式可分为 2 种，分别为值传递和引用（地址）传递：

1. 值传递：适用于实参类型为不可变类型（字符串、数字、元组）；
2. 引用（地址）传递：适用于实参类型为可变类型（列表，字典）；

值传递和引用传递的区别是，函数参数进行值传递后，若形参的值发生改变，不会影响实参的值；而函数参数继续引用传递后，改变形参的值，实参的值也会一同改变。

例如，定义一个名为 demo 的函数，分别为传入一个字符串类型的变量（代表值传递）和列表类型的变量（代表引用传递）：

```
def demo(obj) :
    obj += obj
    print("形参值为：",obj)
print("-------值传递-----")
a = "C 语言中文网"
print("a 的值为：",a)
```

```
7.  demo(a)
8.  print("实参值为：",a)
9.  print("-----引用传递-----")
10. a = [1,2,3]
11. print("a 的值为：",a)
12. demo(a)
13. print("实参值为：",a)
```

运行结果为：

```
-------值传递-----
a 的值为： C 语言中文网
形参值为： C 语言中文网 C 语言中文网
实参值为： C 语言中文网
-----引用传递-----
a 的值为： [1, 2, 3]
形参值为： [1, 2, 3, 1, 2, 3]
实参值为： [1, 2, 3, 1, 2, 3]
```

分析运行结果不难看出，在执行值传递时，改变形式参数的值，实际参数并不会发生改变；而在进行引用传递时，改变形式参数的值，实际参数也会发生同样的改变。

对于初学者来说，本节只需要了解形参和实参，值传递和引用传递的区别即可。对于函数参数的传递方法，如果读者想深入探究其原因，可阅读《Python 函数参数传递机制》一节。

7.3 Python 函数参数传递机制（超级详细）

通过学习《Python 函数值传递和引用传递》一节我们知道，根据实际参数的类型不同，函数参数的传递方式分为值传递和引用传递（又称为地址传递），Python 底层是如何实现它们的呢？Python 中函数参数由实参传递给形参的过程，是由参数传递机制来控制的。

本节将围绕值传递和引用传递，深度剖析它们的底层实现。

Python 函数参数的值传递机制

Python 函数参数的值传递，其本质就是将实际参数值复制一份，将其副本传给形参。这意味着，采用值传递方式的函数中，无论其内部对参数值进行如何修改，都不会影响函数外部的实参。

> 值传递的方式，类似于《西游记》里的孙悟空，它复制一个假孙悟空，假孙悟空具有的能力和真孙悟空相同，可除妖或被砍头。但不管这个假孙悟空遇到什么事，真孙悟空都不会受到任何影响。与此类似，传入函数的是实际参数值的复制品，不管在函数中对这个复制品如何操作，实际参数值本身不会受到任何影响。

下面程序演示了函数参数进行值传递的效果：

```
1.  def swap(a , b) :
2.      '''下面代码实现 a、b 变量的值交换'''
3.      a, b = b, a
4.      print("swap 函数里，a =", a, " b =", b)
5.  a = 6
6.  b = 9
7.  swap(a , b)
8.  print("函数外部 a =", a ," b =", b)
```

运行上面程序，将看到如下运行结果：

```
swap 函数里，a = 9  b = 6
函数外部 a = 6  b = 9
```

从上面的运行结果来看，在 swap() 函数里，经过交换形参 a 和 b 的值，它们的值分别变成了 9 和 6，但函数外部变量 a 和 b 的值依然是 6 和 9。这也证实了，swap() 函数的参数传递机制，采用的是值传递，函数内部使用的形参 a 和 b，和实参 a、b 没有任何关系。

swap() 函数中形参 a 和 b，各自分别是实参 a、b 的复制品。
如果读者依旧不是很理解，下面通过示意图来说明上面程序的执行过程。

上面程序开始定义了 a、b 两个局部变量，这两个变量在内存中的存储示意图如图 1 所示。

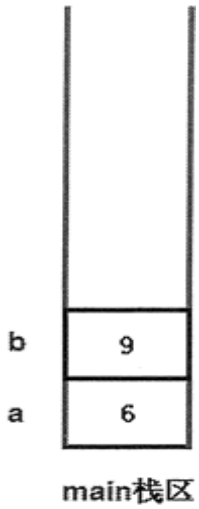


图 1 主栈区中 a、b 变量存储示意图

当程序执行 swap() 函数时，系统进入 swap() 函数，并将主程序中的 a、b 变量作为参数值传入 swap() 函数，但传入 swap() 函数的只是 a、b 的副本，而不是 a、b 本身。进入 swap() 函数后，系统中产生了 4 个变量，这 4 个变量在内存中的存储示意图如图 2 所示。

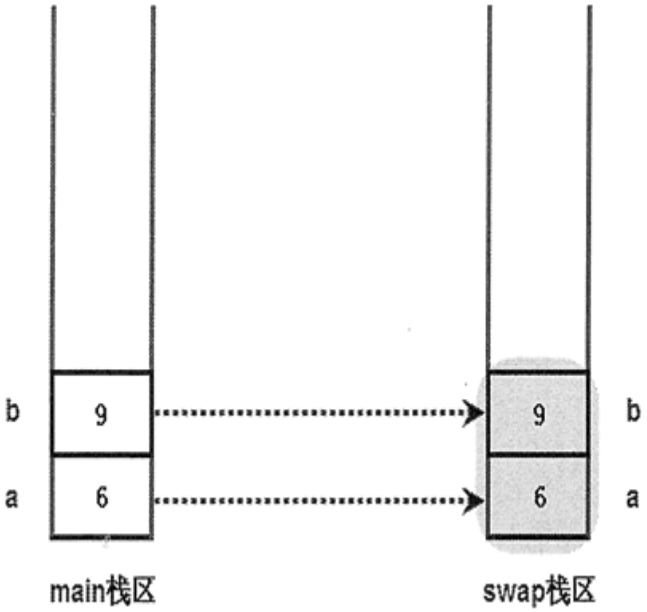


图 2 主栈区的变量作为参数值传入 swap() 函数后存储示意图

当在主程序中调用 swap() 函数时，系统分别为主程序和 swap() 函数分配两块栈区，用于保存它们的局部变量。将主程序中的 a、b 变量作为参数值传入 swap() 函数，实际上是在 swap() 函数栈区中重新产生了两个变量 a、b，并将主程序栈区中 a、b 变量的值分别赋值给 swap() 函数栈区中的 a、b 参数（就是对 swap() 函数的 a、b 两个变量进行初始化）。此时，系统存在两个 a 变量、两个 b 变量，只是存在于不同的栈区中而已。

程序在 swap() 函数中交换 a、b 两个变量的值，实际上是对图 2 中灰色区域的 a、b 变量进行交换。交换结束后，输出 swap() 函数中 a、b 变量的值，可以看到 a 的值为 9，b 的值为 6，此时在内存中的存储示意图如图 3 所示。

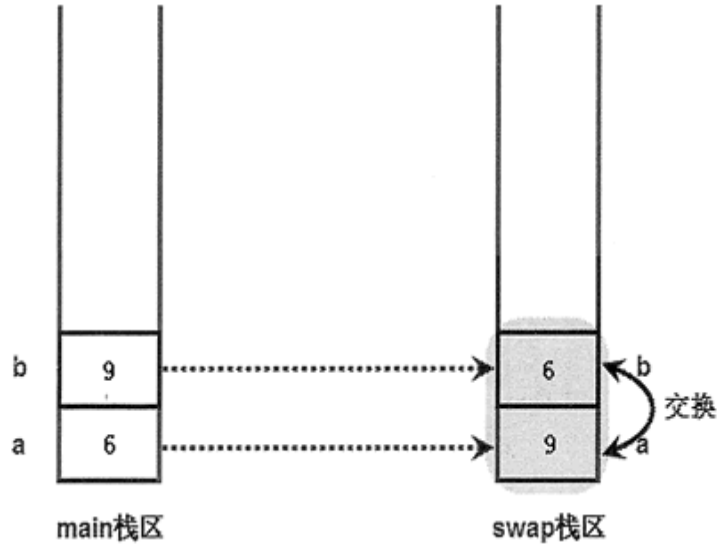


图 3 swap() 函数中 a、b 交换之后的存储示意图

对比图 3 与图 1，可以看到两个示意图中主程序栈区中 a、b 的值并未有任何改变，程序改变的只是 swap() 函数栈区中 a、b 的值。这就是值传递的实质：当系统开始执行函数时，系统对形参执行初始化，就是把实参变量的值赋给函数的形参变量，在函数中操作的并不是实际的实参变量。

Python 函数参数的引用传递

如果实际参数的数据类型是可变对象（列表、字典），则函数参数的传递方式将采用引用传递方式。

下面程序示范了引用传递参数的效果：

```
def swap(dw):
    # 下面代码实现 dw 的 a、b 两个元素的值交换
    dw['a'], dw['b'] = dw['b'], dw['a']
    print("swap 函数里，a =", dw['a'], " b =", dw['b'])
dw = {'a': 6, 'b': 9}
swap(dw)
print("外部 dw 字典中，a =", dw['a']," b =",dw['b'])
```

运行上面程序，将看到如下运行结果：

```
swap 函数里，a = 9  b = 6
外部 dw 字典中，a = 9  b = 6
```

从上面的运行结果来看，在 swap() 函数里，dw 字典的 a、b 两个元素的值被交换成功。不仅如此，当 swap() 函数执行结束后，主程序中 dw 字典的 a、b 两个元素的值也被交换了。

注意，这里这很容易造成一种错觉，读者可能认为，在此 swap() 函数中，使用 dw 字典，就是外界的 dw 字典

本身，而不是他的复制品。这只是一种错觉，实际上，引用传递的底层实现，依旧使用的是值传递的方式。下面还是结合示意图来说明程序的执行过程。

程序开始创建了一个字典对象，并定义了一个 dw 引用变量（其实就是一个指针）指向字典对象，这意味着此时内存中有两个东西：对象本身和指向该对象的引用变量。此时在系统内存中的存储示意图如图 4 所示：

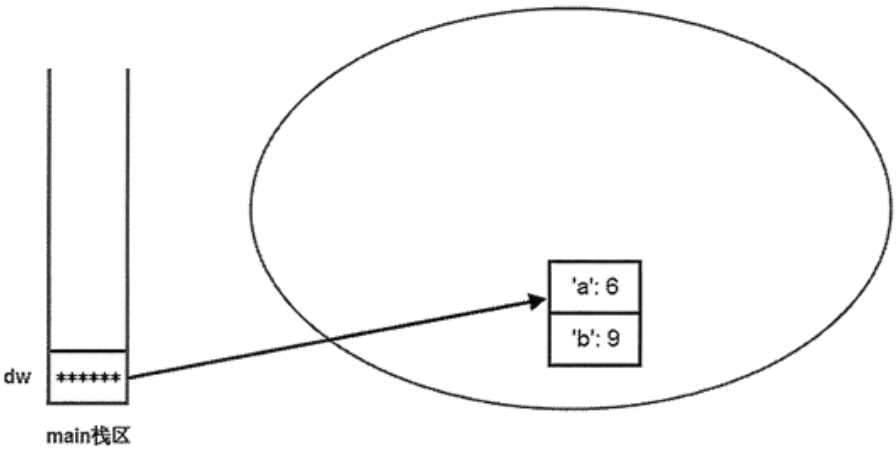


图 4 主程序创建了字典对象后存储示意图

接下来主程序开始调用 swap() 函数，在调用 swap() 函数时，dw 变量作为参数传入 swap() 函数，这里依然采用值传递方式：把主程序中 dw 变量的值赋给 swap() 函数的 dw 形参，从而完成 swap() 函数的 dw 参数的初始化。值得指出的是，主程序中的 dw 是一个引用变量（也就是一个指针），它保存了字典对象的地址值，当把 dw 的值赋给 swap() 函数的 dw 参数后，就是让 swap() 函数的 dw 参数也保存这个地址值，即也会引用到同一个字典对象。图 5 显示了 dw 字典传入 swap() 函数后的存储示意图。

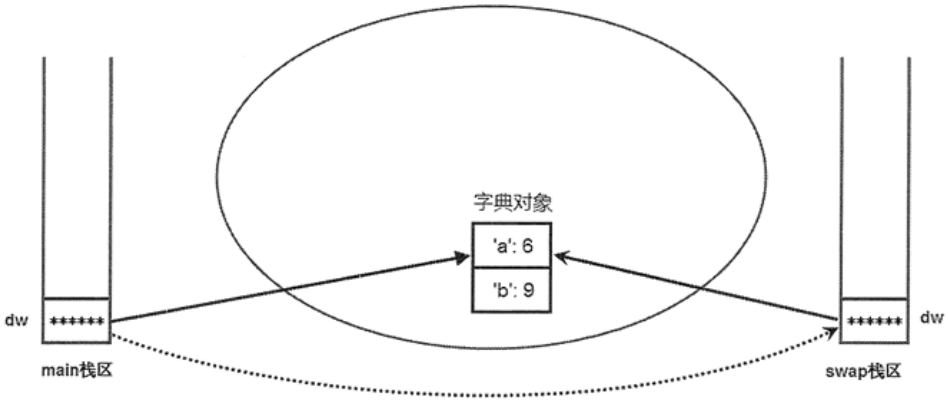


图 5 dw 字典传入 swap() 函数后存储示意图

从图 5 来看，这种参数传递方式是不折不扣的值传递方式，系统一样复制了 dw 的副本传入 swap() 函数。但由于 dw 只是一个引用变量，因此系统复制的是 dw 变量，并未复制字典本身。

当程序在 swap() 函数中操作 dw 参数时，由于 dw 只是一个引用变量，故实际操作的还是字典对象。此时，不管是操作主程序中的 dw 变量，还是操作 swap() 函数里的 dw 参数，其实操作的都是它们共同引用的字典对象，它们引用的是同一个字典对象。因此，当在 swap() 函数中交换 dw 参数所引用字典对象的 a、b 两个元素的值后，可以看到在主程序中 dw 变量所引用字典对象的 a、b 两个元素的值也被交换了。

为了更好地证明主程序中的 dw 和 swap() 函数中的 dw 是两个变量，在 swap() 函数的最后一行增加如下代码：

```
#把 dw 直接赋值为 None，让它不再指向任何对象
dw = None
```

运行上面代码，结果是 swap() 函数中的 dw 变量不再指向任何对象，程序其他地方没有任何改变。主程序调用 swap() 函数后，再次访问 dw 变量的 a、b 两个元素，依然可以输出 9、6。可见，主程序中的 dw 变量没有受到任何影响。实际上，当在 swap() 函数中增加“dw =None”代码后，在内存中的存储示意图如图 6 所示。

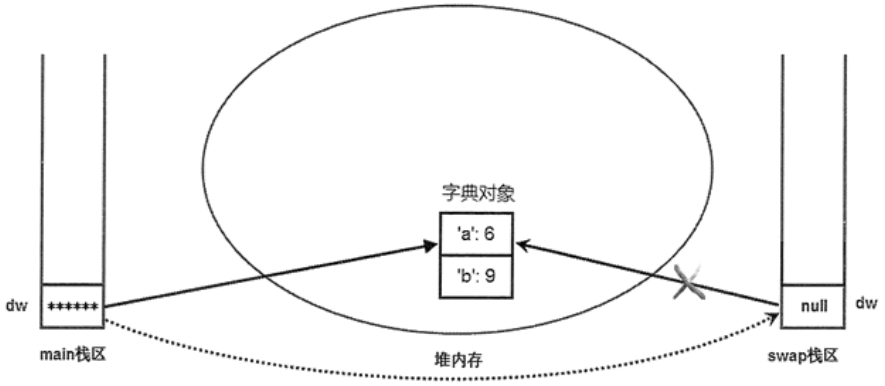


图 6 将 swap() 函数中的 dw 赋值为 None 后存储示意图

从图 6 来看，把 swap() 函数中的 dw 赋值为 None 后，在 swap() 函数中失去了对字典对象的引用，不可再访问该字典对象。但主程序中的 dw 变量不受任何影响，依然可以引用该字典对象，所以依然可以输出字典对象的 a、b 元素的值。

通过上面介绍可以得出如下两个结论：

1. 不管什么类型的参数，在 Python 函数中对参数直接使用“=”符号赋值是没用的，直接使用“=”符号赋值并不能改变参数。
2. 如果需要让函数修改某些数据，则可以通过把这些数据包装成列表、字典等可变对象，然后把列表、字典等可变对象作为参数传入函数，在函数中通过列表、字典的方法修改它们，这样才能改变这些数据。

7.4 什么是位置参数，Python 位置参数

位置参数，有时也称必备参数，指的是必须按照正确的顺序将实际参数传到函数中，换句话说，调用函数时传入实际参数的数量和位置都必须和定义函数时保持一致。

实参和形参数量必须一致

在调用函数，指定的实际参数的数量，必须和形式参数的数量一致（传多传少都不行），否则 Python 解释器会抛出 TypeError 异常，并提示缺少必要的位置参数。

例如：

```
1.  def girth(width , height):
2.      return 2 * (width + height)
3.  #调用函数时，必须传递 2 个参数，否则会引发错误
4.  print(girth(3))
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(girth(3))
TypeError: girth() missing 1 required positional argument: 'height'
```

可以看到，抛出的异常类型为 TypeError，具体是指 girth() 函数去烧一个必要的 height 参数。

同样，多传参数也会抛出异常：

```
1.  def girth(width , height):
2.      return 2 * (width + height)
3.  #调用函数时，必须传递 2 个参数，否则会引发错误
4.  print(girth(3,2,4))
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 4, in <module>
    print(girth(3,2,4))
TypeError: girth() takes 2 positional arguments but 3 were given
```

通过 TypeErroe 异常信息可以知道，girth() 函数本只需要 2 个参数，但是却传入了 3 个参数。

实参和形参位置必须一致

在调用函数时，传入实际参数的位置必须和形式参数位置一一对应，否则会产生以下 2 种结果：

1. 抛出 TypeError 异常
   当实际参数类型和形式参数类型不一致，并且在函数种，这两种类型之间不能正常转换，此时就会抛出 TypeError 异常。

   例如：

```
def area(height,width):
    return height*width/2
print(area("C 语言中文网",3))
```

   输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 3, in <module>
    print(area("C 语言中文网",3))
  File "C:\Users\mengma\Desktop\1.py", line 2, in area
    return height*width/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

   以上显示的异常信息，就是因为字符串类型和整形数值做除法运算。

2. 产生的结果和预期不符
   调用函数时，如果指定的实际参数和形式参数的位置不一致，但它们的数据类型相同，那么程序将不会抛出异常，只不过导致运行结果和预期不符。

   例如，设计一个求梯形面积的函数，并利用此函数求上底为 4cm，下底为 3cm，高为 5cm 的梯形的面积。但如果交互高和下低参数的传入位置，计算结果将导致错误：

```
def area(upper_base,lower_bottom,height):
    return (upper_base+lower_bottom)*height/2
print("正确结果为：",area(4,3,5))
print("错误结果为：",area(4,5,3))
```

   运行结果为:

```
正确结果为： 17.5
错误结果为： 13.5
```

   因此，在调用函数时，一定要确定好位置，否则很有可能产生类似示例中的这类错误，还不容易发现。

7.5 Python 函数关键字参数及用法

目前为止，我们使用函数时所用的参数都是位置参数，即传入函数的实际参数必须与形式参数的数量和位置对应。而本节将介绍的关键字参数，则可以避免牢记参数位置的麻烦，令函数的调用和参数传递更加灵活方便。

关键字参数是指使用形式参数的名字来确定输入的参数值。通过此方式指定函数实参时，不再需要与形参的位置完全一致，只要将参数名写正确即可。

> 因此，Python 函数的参数名应该具有更好的语义，这样程序可以立刻明确传入函数的每个参数的含义。

例如，在下面的程序中就使用到了关键字参数的形式给函数传参：

```
1.  def dis_str(str1,str2):
2.      print("str1:",str1)
3.      print("str2:",str2)
4.  #位置参数
5.  dis_str("http://c.biancheng.net/python/","http://c.biancheng.net/shell/")
6.  #关键字参数
7.  dis_str("http://c.biancheng.net/python/",str2="http://c.biancheng.net/shell/")
8.  dis_str(str2="http://c.biancheng.net/python/",str1="http://c.biancheng.net/shell/")
```

程序执行结果为：

```
str1: http://c.biancheng.net/python/
str2: http://c.biancheng.net/shell/
str1: http://c.biancheng.net/python/
str2: http://c.biancheng.net/shell/
str1: http://c.biancheng.net/shell/
str2: http://c.biancheng.net/python/
```

可以看到，在调用有参函数时，既可以根据位置参数来调用，也可以使用关键字参数（程序中第 8 行）来调用。在使用关键字参数调用时，可以任意调换参数传参的位置。

当然，还可以像第 7 行代码这样，使用位置参数和关键字参数混合传参的方式。但需要注意，混合传参时关键字参数必须位于所有的位置参数之后。也就是说，如下代码是错误的：

```
1.  # 位置参数必须放在关键字参数之前，下面代码错误
2.  dis_str(str1="http://c.biancheng.net/python/","http://c.biancheng.net/shell/")
```

Python 解释器会报如下错误：

```
SyntaxError: positional argument follows keyword argument
```

7.6 Python 函数默认参数设置（超级详细）

我们知道，在调用函数时如果不指定某个参数，Python 解释器会抛出异常。为了解决这个问题，Python 允许为参数设置默认值，即在定义函数时，直接给形式参数指定一个默认值。这样的话，即便调用函数时没有给拥有默认值的形参传递参数，该参数可以直接使用定义函数时设置的默认值。

Python 定义带有默认值参数的函数，其语法格式如下：

```
def 函数名(...，形参名，形参名=默认值)：
    代码块
```

注意，在使用此格式定义函数时，指定有默认值的形式参数必须在所有没默认值参数的最后，否则会产生语法错误。

下面程序演示了如何定义和调用有默认参数的函数：

```
1.  #str1 没有默认参数，str2 有默认参数
2.  def dis_str(str1,str2 = "http://c.biancheng.net/python/"):
3.      print("str1:",str1)
4.      print("str2:",str2)
5.
6.  dis_str("http://c.biancheng.net/shell/")
7.  dis_str("http://c.biancheng.net/java/","http://c.biancheng.net/golang/")
```

运行结果为：

```
str1: http://c.biancheng.net/shell/
str2: http://c.biancheng.net/python/
str1: http://c.biancheng.net/java/
str2: http://c.biancheng.net/golang/
```

上面程序中，dis_str() 函数有 2 个参数，其中第 2 个设有默认参数。这意味着，在调用 dis_str() 函数时，我们可以仅传入 1 个参数，此时该参数会传给 str1 参数，而 str2 会使用默认的参数，如程序中第 6 行代码所示。

当然在调用 dis_str() 函数时，也可以给所有的参数传值（如第 7 行代码所示），这时即便 str2 有默认值，它也会优先使用传递给它的新值。

同时，结合关键字参数，以下 3 种调用 dis_str() 函数的方式也是可以的：

```
1.  dis_str(str1 = "http://c.biancheng.net/shell/")
2.  dis_str("http://c.biancheng.net/java/",str2 = "http://c.biancheng.net/golang/")
3.  dis_str(str1 = "http://c.biancheng.net/java/",str2 = "http://c.biancheng.net/golang/")
```

再次强调，当定义一个有默认值参数的函数时，有默认值的参数必须位于所有没默认值参数的后面。因此，下面例子中定义的函数是不正确的：

```
#语法错误
def dis_str(str1="http://c.biancheng.net/python/",str2,str3):
    pass
```

显然，str1 设有默认值，而 str2 和 str3 没有默认值，因此 str1 必须位于 str2 和 str3 之后。

有读者可能会问，对于自己自定义的函数，可以轻易知道哪个参数有默认值，但如果使用 Python 提供的内置函数，又或者其它第三方提供的函数，怎么知道哪些参数有默认值呢？

Pyhton 中，可以使用“函数名.__defaults__”查看函数的默认值参数的当前值，其返回值是一个元组。以本节中的 dis_str() 函数为例，在其基础上，执行如下代码：

```
print(dis_str.__defaults__)
```

程序执行结果为：

('http://c.biancheng.net/python/',)

7.7 Python 函数可变参数（ *args,**kwargs ）详解

Python 在定义函数时也可以使用可变参数，即允许定义参数个数可变的函数。这样当调用该函数时，可以向其传入任意多个参数。

可变参数，又称不定长参数，即传入函数中的实际参数可以是任意多个。Python 定义可变参数，主要有以下 2 种形式。

1) 可变参数：形参前添加一个 '*'

此种形式的语法格式如下所示：

```
*args
```

args 表示创建一个名为 args 的空元组，该元组可接受任意多个外界传入的非关键字实参。

下面程序演示了如何定义一个参数可变的函数：

```
# 定义了支持参数收集的函数
def dis_str(home, *str) :
    print(home)
    # 输出 str 元组中的元素
    print("str=",str)
    for s in str :
        print(s)
#可传入任何多个参数
dis_str("http://c.biancheng.net","http://c.biancheng.net/python/","http://c.biancheng.net/shell/")
```

程序执行结果为：

```
http://c.biancheng.net
str= ('http://c.biancheng.net/python/', 'http://c.biancheng.net/shell/')
http://c.biancheng.net/python/
http://c.biancheng.net/shell/
```

上面程序中，dis_str() 函数的最后一个参数就是 str 元组，这样在调用该函数时，除了前面位置参数接收对应位置的实参外，其它非关键字参数都会由 str 元组接收。

当然，可变参数并不一定必须为最后一个函数参数，例如修改 dis_str() 函数为：

```
# 定义了支持参数收集的函数
```

```
def dis_str(*str,home) :
    print(home)
    # 输出 str 元组中的元素
    print("str=",str)
    for s in str :
        print(s)

dis_str("http://c.biancheng.net","http://c.biancheng.net/python/",home="http://c.biancheng.net/shell/")
```

可以看到，str 可变参数作为 dis_str() 函数的第一个参数。但需要注意的是，在调用该函数时，必须以关键字参数的形式给普通参数传值，否则 Python 解释器会把所有参数都优先传给可变参数，如果普通参数没有默认值，就会报错。

也就是说，下面代码调用上面的 dia_str() 函数，是不对的：

```
dis_str("http://c.biancheng.net","http://c.biancheng.net/python/","http://c.biancheng.net/shell/")
```

Python 解释器会提供如下报错信息：

```
TypeError: dis_str() missing 1 required keyword-only argument: 'home'
```

翻译过来就是我们没有给 home 参数传值。当然，如果 home 参数有默认参数，则此调用方式是可行的。

2) 可变参数：形参前添加两个'*'

这种形式的语法格式如下：

```
**kwargs
```

**kwargs 表示创建一个名为 kwargs 的空字典，该字典可以接收任意多个以关键字参数赋值的实际参数。

例如如下代码：

```
# 定义了支持参数收集的函数
def dis_str(home,*str,**course) :
    print(home)
    print(str)
    print(course)
#调用函数
```

```
dis_str("C 语言中文网",\
        "http://c.biancheng.net",\
        "http://c.biancheng.net/python/",\
        shell 教程="http://c.biancheng.net/shell/",\
        go 教程="http://c.biancheng.net/golang/",\
        java 教程="http://c.biancheng.net/java/")
```

程序输出结果为：

```
C 语言中文网
('http://c.biancheng.net', 'http://c.biancheng.net/python/')
{'shell 教程': 'http://c.biancheng.net/shell/', 'go 教程': 'http://c.biancheng.net/golang/', 'java 教程': 'http://c.biancheng.net/java/'}
```

上面程序在调用 dis_str() 函数时，第 1 个参数传递给 home 参数，第 2、3 个非关键字参数传递给 str 元组，最后 2 个关键字参数将由 course 字典接收。

注意，*args 可变参数的值默认是空元组，**kwargs 可变参数的值默认是空字典。因此，在调用具有可变参数的函数时，不一定非要给它们传值。以调用 dis_str(home, *str, **course) 为例，下面的调用方式也是正确的：

```
dis_str(home="http://c.biancheng.net/shell/")
```

程序执行结果为：

```
http://c.biancheng.net/shell/
()
{}
```

7.8 Python 逆向参数收集详解（进阶必读）

前面章节中介绍了，Python 支持定义具有可变参数的函数，即该函数可以接收任意多个参数，其中非关键字参数会集中存储到元组参数（*args）中，而关键字参数则集中存储到字典参数（**kwargs）中，这个过程可称为参数收集。

不仅如此，Python 还支持逆向参数收集，即直接将列表、元组、字典作为函数参数，Python 会将其进行拆分，把其中存储的元素按照次序分给函数中的各个形参。

在以逆向参数收集的方式向函数参数传值时，Pyhon 语法规定，当传入列表或元组时，其名称前要带一个 * 号，当传入字典时，其名称前要带有 2 个 * 号。

举个例子：

```
def dis_str(name,add) :
    print("name:",name)
    print("add",add)

data = ["Python 教程","http://c.biancheng.net/python/"]
#使用逆向参数收集方式传值
dis_str(*data)
```

程序执行结果为：

```
name: Python 教程
add http://c.biancheng.net/python/
```

再举个例子：

```
def dis_str(name,add) :
    print("name:",name)
    print("add:",add)

data = {'name':"Python 教程",'add':"http://c.biancheng.net/python/"}
#使用逆向参数收集方式传值
dis_str(**data)
```

程序执行结果为：

```
name: Python 教程
add: http://c.biancheng.net/python/
```

此外，以逆向参数收集的方式，还可以给拥有可变参数的函数传参，例如：

```
def dis_str(name,*add) :
    print("name:",name)
    print("add:",add)

data = ["http://c.biancheng.net/python/",\
        "http://c.biancheng.net/shell/",\
        "http://c.biancheng.net/golang/"]
#使用逆向参数收集方式传值
dis_str("Python 教程",*data)
```

程序执行结果为：

```
name: Python 教程
add: ('http://c.biancheng.net/python/', 'http://c.biancheng.net/shell/',
'http://c.biancheng.net/golang/')
```

上面程序中，也同样可以用逆向参数收集的方式给 name 参数传值，只需要将 "python 教程" 放到 data 列表中第一个位置即可。也就是说，上面程序中，以下面代码调用 dis_str() 函数的方式也是可行的：

```
data = ["Python 教程",\
        "http://c.biancheng.net/python/",\
        "http://c.biancheng.net/shell/",\
        "http://c.biancheng.net/golang/"]
#使用逆向参数收集方式传值
dis_str(*data)
```

执行此程序，会发现其输出结果和上面一致。

再次强调，如果使用逆向参数收集的方式，必须注意 * 号的添加。以逆向收集列表为例，如果传参时其列表名前不带 * 号，则 Python 解释器会将整个列表作为参数传递给一个参数。例如：

```
def dis_str(name,*add) :
    print("name:",name)
    print("add:",add)

data = ["Python 教程",\
        "http://c.biancheng.net/python/",\
        "http://c.biancheng.net/shell/",\
```

```
8.          "http://c.biancheng.net/golang/"]
9.  dis_str(data)
```

程序执行结果为：

```
name: ['Python 教程', 'http://c.biancheng.net/python/', 'http://c.biancheng.net/shell/',
'http://c.biancheng.net/golang/']
add: ()
```

7.9 Python None（空值）及用法

在 Python 中，有一个特殊的常量 None（N 必须大写）。和 False 不同，它不表示 0，也不表示空字符串，而表示没有值，也就是空值。

这里的空值并不代表空对象，即 None 和 []、 “” 不同：

```
>>> None is []
False
>>> None is ""
False
```

None 有自己的数据类型，我们可以在 IDLE 中使用 type() 函数查看它的类型，执行代码如下：

```
>>> type(None)
<class 'NoneType'>
```

可以看到，它属于 NoneType 类型。

需要注意的是，None 是 NoneType 数据类型的唯一值（其他编程语言可能称这个值为 null、nil 或 undefined），也就是说，我们不能再创建其它 NoneType 类型的变量，但是可以将 None 赋值给任何变量。如果希望变量中存储的东西不与任何其它值混淆，就可以使用 None。

除此之外，None 常用于 assert、判断以及函数无返回值的情况。举个例子，在前面章节中我们一直使用 print() 函数输出数据，其实该函数的返回值就是 None。因为它的功能是在屏幕上显示文本，根本不需要返回任何值，所以 print() 就返回 None。

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

另外，对于所有没有 return 语句的函数定义，Python 都会在末尾加上 return None，使用不带值的 return 语句（也就是只有 return 关键字本身），那么就返回 None。

7.10 Python return 函数返回值详解

到目前为止，我们创建的函数都只是对传入的数据进行了处理，处理完了就结束。但实际上，在某些场景中，我们还需函数将处理的结果反馈回来，就好像主管向下级员工下达命令，让其去打印文件，员工打印好文件后并没有完成任务，还需要将文件交给主管。

Python 中，用 def 语句创建函数时，可以用 return 语句指定应该返回的值，该返回值可以是任意类型。需要注意的是，return 语句在同一函数中可以出现多次，但只要有一个得到执行，就会直接结束函数的执行。

函数中，使用 return 语句的语法格式如下：

```
return [返回值]
```

其中，返回值参数可以指定，也可以省略不写（将返回空值 None）。

【例 1】

```
def add(a,b):
    c = a + b
    return c
#函数赋值给变量
c = add(3,4)
print(c)
#函数返回值作为其他函数的实际参数
print(add(3,4))
```

运行结果为：

```
7
7
```

本例中，add() 函数既可以用来计算两个数的和，也可以连接两个字符串，它会返回计算的结果。

通过 return 语句指定返回值后，我们在调用函数时，既可以将该函数赋值给一个变量，用变量保存函数的返回值，也可以将函数再作为某个函数的实际参数。

【例 2】

```
def isGreater0(x):
    if x > 0:
        return True
    else:
        return False
```

```
print(isGreater0(5))
print(isGreater0(0))
```

运行结果为：

```
True
False
```

可以看到，函数中可以同时包含多个 return 语句，但需要注意的是，最终真正执行的做多只有 1 个，且一旦执行，函数运行会立即结束。

以上实例中，我们通过 return 语句，都仅返回了一个值，但其实通过 return 语句，可以返回多个值，读者可以阅读《Python 函数返回多个值》一节做详细了解。

7.11 Python 函数返回多个值的方法（入门必读）

通常情况下，一个函数只有一个返回值，实际上 Python 也是如此，只不过 Python 函数能以返回列表或者元组的方式，将要返回的多个值保存到序列中，从而间接实现返回多个值的目的。

因此，实现 Python 函数返回多个值，有以下 2 种方式：

1. 在函数中，提前将要返回的多个值存储到一个列表或元组中，然后函数返回该列表或元组；
2. 函数直接返回多个值，之间用逗号（，）分隔，Python 会自动将多个值封装到一个元组中，其返回值仍是一个元组。

下面程序演示了以上 2 种实现方法：

```
def retu_list() :
    add = ["http://c.biancheng.net/python/",\
           "http://c.biancheng.net/shell/",\
           "http://c.biancheng.net/golang/"]
    return add

def retu_tuple() :
    return "http://c.biancheng.net/python/",\
          "http://c.biancheng.net/golang/",\
          "http://c.biancheng.net/golang/"

print("retu_list = ",retu_list())
print("retu_tuple = ",retu_tuple())
```

程序执行结果为：

```
retu_list =  ['http://c.biancheng.net/python/', 'http://c.biancheng.net/shell/',
'http://c.biancheng.net/golang/']
retu_tuple =  ('http://c.biancheng.net/python/', 'http://c.biancheng.net/golang/',
'http://c.biancheng.net/golang/')
```

在此基础上，我们可以利用 Python 提供的序列解包功能，之间使用对应数量的变量，直接接收函数返回列表或元组中的多个值。这里以 retu_list() 为例：

```
def retu_list() :
    add = ["http://c.biancheng.net/python/",\
           "http://c.biancheng.net/shell/",\
           "http://c.biancheng.net/golang/"]
```

```
5.      return add
6.
7.  pythonadd,shelladd,golangadd = retu_list()
8.
9.  print("pythonadd=",pythonadd)
10. print("shelladd=",shelladd)
11. print("golangadd=",golangadd)
```

程序执行结果为：

```
pythonadd= http://c.biancheng.net/python/
shelladd= http://c.biancheng.net/shell/
golangadd= http://c.biancheng.net/golang/
```

7.12 Python partial 偏函数及用法

简单的理解偏函数，它是对原始函数的二次封装，是将现有函数的部分参数预先绑定为指定值，从而得到一个新的函数，该函数就称为偏函数。相比原函数，偏函数具有较少的可变参数，从而降低了函数调用的难度。

定义偏函数，需使用 partial 关键字（位于 functools 模块中），其语法格式如下：

```
偏函数名 = partial(func, *args, **kwargs)
```

其中，func 指的是要封装的原函数，*args 和 **kwargs 分别用于接收无关键字实参和关键字实参。

有关 *args 和 **kwargs 作为函数形参更详细的解释，可阅读《Python 函数可变参数（*args, **kwargs）》一节。

下面举几个例子，让大家可以直观感受一下偏函数的用法和功能。

【例 1】

```
1.  from functools import partial
2.  #定义个原函数
3.  def display(name,age):
4.      print("name:",name,"age:",age)
5.  #定义偏函数，其封装了 display() 函数，并为 name 参数设置了默认参数
6.  GaryFun = partial(display,name = 'Gary')
7.  #由于 name 参数已经有默认值，因此调用偏函数时，可以不指定
8.  GaryFun(age = 13)
```

运行结果为：

```
name: Gary age: 13
```

注意，此程序的第 8 行代码中，必须采用关键字参数的形式给 age 形参传参，因为如果以无关键字参数的方式，该实参将试图传递给 name 形参，Python 解释器会报 TypeError 错误。

【例 2】

```
1.  from functools import partial
2.
3.  def mod( n, m ):
4.    return n % m
5.  #定义偏函数，并设置参数 n 对应的实参值为 100
6.  mod_by_100 = partial( mod, 100 )
7.
```

```
print(mod( 100, 7 ))
print(mod_by_100( 7 ))
```

运行结果为：

```
2
2
```

结合以上示例不难分析出，偏函数的本质是将函数式编程、默认参数和冗余参数结合在一起，通过偏函数传入的参数调用关系，与正常函数的参数调用关系是一致的。

偏函数通过将任意数量（顺序）的参数，转化为另一个带有剩余参数的函数对象，从而实现了截取函数功能（偏向）的效果。在实际应用中，可以使用一个原函数，然后将其封装多个偏函数，在调用函数时全部调用偏函数，一定程序上可以提高程序的可读性。

7.13 Python 函数递归（带实例演示）

一个函数在它的函数体内调用它自身称为递归调用，这种函数称为递归函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层，当最内层的函数执行完毕后，再一层一层地由里到外退出。

递归函数不是 Python 语言的专利，C/C++、Java、C#、JavaScript、PHP 等其他编程语言也都支持递归函数。

下面我们通过一个实例，看看递归函数到底是如何运作的。

有这样一个数学题。已知有一个数列：f(0) = 1，f(1) = 4，f(n + 2) = 2*f(n+ 1) +f(n)，其中 n 是大于 0 的整数，求 f(10) 的值。这道题可以使用递归来求得。下面程序将定义一个 fn() 函数，用于计算 f(10) 的值。

```
1.  def fn(n) :
2.      if n == 0 :
3.          return 1
4.      elif n == 1 :
5.          return 4
6.      else :
7.          # 函数中调用它自身，就是函数递归
8.          return 2 * fn(n - 1) + fn(n - 2)
9.  # 输出 fn(10)的结果
10. print("fn(10)的结果是:", fn(10))
```

在上面的 fn() 函数体中再次调用了 fn() 函数，这就是函数递归。注意在 fn() 函数体中调用 fn 的形式：

```
return 2 * fn(n - 1) + fn(n - 2)
```

对于 fn(10)，即等于 2*fn(9)+fn(8)，其中 fn(9) 又等于 2*fn(8)+fn(7)……依此类推，最终会计算到 fn(2) 等于 2*fn(1)+fn(0)，即 fn(2) 是可计算的，这样递归带来的隐式循环就有结束的时候，然后一路反算回去，最后就可以得到 fn(10) 的值。

仔细看上面递归的过程，当一个函数不断地调用它自身时，必须在某个时刻函数的返回值是确定的，即不再调用它自身：否则，这种递归就变成了无穷递归，类似于死循环。因此，在定义递归函数时有一条最重要的规定：递归一定要向已知方向进行。

例如，如果把上面数学题改为如此。已知有一个数列：f(20)=1，f(21)=4，f(n + 2)=2*f(n+1)+f(n)，其中 n 是大于 0 的整数，求 f(10) 的值。那么 f(10) 的函数体应该改为如下形式：

```
1.  def fn(n) :
2.      if n == 20 :
3.          return 1
4.      elif n == 21 :
5.          return 4
6.      else :
7.          # 函数中调用它自身，就是函数递归
```

```
8.          return fn(n + 2) - 2*fn(n + 1)
```

从上面的 fn() 函数来看，当程序要计算 fn(10) 的值时，fn(10) 等于 fn(12)-2*fn(11)，而 fn(11) 等于 fn(13)-2*fn(12)……依此类推，直到 fn(19) 等于 fn(21)-2*fn(20)，此时就可以得到 fn(19) 的值，然后依次反算到 fn(10) 的值。这就是递归的重要规则：对于求 fn(10) 而言，如果 fn(0) 和 fn(1) 是已知的，则应该采用 fn(n)=2*fn(n-1)+fn(n-2) 的形式递归，因为小的一端已知；如果 fn(20) 和 fn(21) 是已知的，则应该采用 fn(n)=fn(n+2)-2*fn(n+1) 的形式递归，因为大的一端已知。

递归是非常有用的，例如程序希望遍历某个路径下的所有文件，但这个路径下的文件夹的深度是未知的，那么就可以使用递归来实现这个需求。系统可定义一个函数，该函数接收一个文件路径作为参数，该函数可遍历出当前路径下的所有文件和文件路径，即在该函数的函数体中再次调用函数自身来处理该路径下的所有文件路径。

总之，只要在一个函数的函数体中调用了函数自身，就是函数递归。递归一定要向已知方向进行。

7.14 Python 变量作用域（全局变量和局部变量）

所谓作用域（Scope），就是变量的有效范围，就是变量可以在哪个范围以内使用。有些变量可以在整段代码的任意位置使用，有些变量只能在函数内部使用，有些变量只能在 for 循环内部使用。

变量的作用域由变量的定义位置决定，在不同位置定义的变量，它的作用域是不一样的。本节我们只讲解两种变量，局部变量和全局变量。

Python 局部变量

在函数内部定义的变量，它的作用域也仅限于函数内部，出了函数就不能使用了，我们将这样的变量称为局部变量（Local Variable）。

要知道，当函数被执行时，Python 会为其分配一块临时的存储空间，所有在函数内部定义的变量，都会存储在这块空间中。而在函数执行完毕后，这块临时存储空间随即会被释放并回收，该空间中存储的变量自然也就无法再被使用。

举个例子：

```
1.  def demo():
2.      add = "http://c.biancheng.net/python/"
3.      print("函数内部 add =",add)
4.
5.  demo()
6.  print("函数外部 add =",add)
```

程序执行结果为：

```
函数内部 add = http://c.biancheng.net/python/
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\file.py", line 6, in <module>
    print("函数外部 add =",add)
NameError: name 'add' is not defined
```

可以看到，如果试图在函数外部访问其内部定义的变量，Python 解释器会报 NameError 错误，并提示我们没有定义要访问的变量，这也证实了当函数执行完毕后，其内部定义的变量会被销毁并回收。

值得一提的是，函数的参数也属于局部变量，只能在函数内部使用。例如：

```
1.  def demo(name,add):
2.      print("函数内部 name =",name)
3.      print("函数内部 add =",add)
```

```
4.  demo("Python 教程","http://c.biancheng.net/python/")
5.
6.  print("函数外部 name =",name)
7.  print("函数外部 add =",add)
```

程序执行结果为：

```
函数内部 name = Python 教程
函数内部 add = http://c.biancheng.net/python/
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\file.py", line 7, in <module>
    print("函数外部 name =",name)
NameError: name 'name' is not defined
```

由于 Python 解释器是逐行运行程序代码，由此这里仅提示给我“name 没有定义”，实际上在函数外部访问 add 变量也会报同样的错误。

Python 全局变量

除了在函数内部定义变量，Python 还允许在所有函数的外部定义变量，这样的变量称为全局变量（Global Variable）。

和局部变量不同，全局变量的默认作用域是整个程序，即全局变量既可以在各个函数的外部使用，也可以在各函数内部使用。

定义全局变量的方式有以下 2 种：

- 在函数体外定义的变量，一定是全局变量，例如：

```
1.  add = "http://c.biancheng.net/shell/"
2.  def text():
3.      print("函数体内访问：",add)
4.  text()
5.  print('函数体外访问：',add)
```

运行结果为：

```
函数体内访问： http://c.biancheng.net/shell/
函数体外访问： http://c.biancheng.net/shell/
```

- 在函数体内定义全局变量。即使用 global 关键字对变量进行修饰后，该变量就会变为全局变量。例如：

```
1.  def text():
2.      global add
```

```
    add= "http://c.biancheng.net/java/"
    print("函数体内访问：",add)
text()
print('函数体外访问：',add)
```

运行结果为：

```
函数体内访问： http://c.biancheng.net/java/
函数体外访问： http://c.biancheng.net/java/
```

注意，在使用 global 关键字修饰变量名时，不能直接给变量赋初值，否则会引发语法错误。

获取指定作用域范围中的变量

在一些特定场景中，我们可能需要获取某个作用域内（全局范围内或者局部范围内）所有的变量，Python 提供了以下 3 种方式：

1) globals()函数

globals() 函数为 Python 的内置函数，它可以返回一个包含全局范围内所有变量的字典，该字典中的每个键值对，键为变量名，值为该变量的值。

举个例子：

```
#全局变量
Pyname = "Python 教程"
Pyadd = "http://c.biancheng.net/python/"
def text():
    #局部变量
    Shename = "shell 教程"
    Sheadd= "http://c.biancheng.net/shell/"
print(globals())
```

程序执行结果为：

```
{ ..... , 'Pyname': 'Python 教程', 'Pyadd': 'http://c.biancheng.net/python/', .....}
```

注意，globals() 函数返回的字典中，会默认包含有很多变量，这些都是 Python 主程序内置的，读者暂时不用理会它们。

可以看到，通过调用 globals() 函数，我们可以得到一个包含所有全局变量的字典。并且，通过该字典，我们还可以访问指定变量，甚至如果需要，还可以修改它的值。例如，在上面程序的基础上，添加如下语句：

```
print(globals()['Pyname'])
globals()['Pyname'] = "Python 入门教程"
print(Pyname)
```

程序执行结果为：

```
Python 教程
Python 入门教程
```

2) locals()函数

locals() 函数也是 Python 内置函数之一，通过调用该函数，我们可以得到一个包含当前作用域内所有变量的字典。这里所谓的“当前作用域”指的是，在函数内部调用 locals() 函数，会获得包含所有局部变量的字典；而在全局范文内调用 locals() 函数，其功能和 globals() 函数相同。

举个例子：

```
#全局变量
Pyname = "Python 教程"
Pyadd = "http://c.biancheng.net/python/"
def text():
    #局部变量
    Shename = "shell 教程"
    Sheadd= "http://c.biancheng.net/shell/"
    print("函数内部的 locals:")
    print(locals())
text()
print("函数外部的 locals:")
print(locals())
```

程序执行结果为：

```
函数内部的 locals:
{'Sheadd': 'http://c.biancheng.net/shell/', 'Shename': 'shell 教程'}
函数外部的 locals:
{..... , 'Pyname': 'Python 教程', 'Pyadd': 'http://c.biancheng.net/python/', ..... }
```

当使用 locals() 函数获取所有全局变量时，和 globals() 函数一样，其返回的字典中会默认包含有很多变量，这些都是 Python 主程序内置的，读者暂时不用理会它们。

注意，当使用 locals() 函数获得所有局部变量组成的字典时，可以向 globals() 函数那样，通过指定键访问对应的变量值，但无法对变量值做修改。例如：

```
#全局变量
Pyname = "Python 教程"
Pyadd = "http://c.biancheng.net/python/"
def text():
    #局部变量
    Shename = "shell 教程"
    Sheadd= "http://c.biancheng.net/shell/"
    print(locals()['Shename'])
    locals()['Shename'] = "shell 入门教程"
    print(Shename)
text()
```

程序执行结果为：

```
shell 教程
shell 教程
```

显然，locals() 返回的局部变量组成的字典，可以用来访问变量，但无法修改变量的值。

3) vars(object)

vars() 函数也是 Python 内置函数，其功能是返回一个指定 object 对象范围内所有变量组成的字典。如果不传入 object 参数，vars() 和 locals() 的作用完全相同。

由于目前读者还未学习 Python 类和对象，因此初学者可先跳过该函数的学习，等学完 Python 类和对象之后，再回过头来学习该函数。

举个例子：

```
 #全局变量
Pyname = "Python 教程"
Pyadd = "http://c.biancheng.net/python/"
class Demo:
    name = "Python 教程"
    add = "http://c.biancheng.net/python/"
print("有 object：")
print(vars(Demo))

print("无 object：")
print(vars())
```

程序执行结果为：

```
有 object：
{..... , 'name': 'Python 教程', 'add': 'http://c.biancheng.net/python/', .....}
无 object：
{..... , 'Pyname': 'Python 教程', 'Pyadd': 'http://c.biancheng.net/python/', ..... }
```

7.15 Python 如何在函数中使用同名的全局变量？

《Python 局部变量和全局变量》一节中提到，全局变量可以在程序中任何位置被访问甚至修改，但是，当函数中定义了和全局变量同名的局部变量时，那么在当前函数中，无论是访问还是修改该同名变量，操作的都是局部变量，而不再是全局变量。

当函数内部的局部变量和函数外部的全局变量同名时，在函数内部，局部变量会“遮蔽”同名的全局变量。

有读者可能并不能完全理解上面这段话，没关系，这里举个实例：

```
1.  name = "Python 教程"
2.  def demo ():
3.      #访问全局变量
4.      print(name)
5.  demo()
```

程序执行结果为：

```
Python 教程
```

上面程序中，第 4 行直接访问 name 变量，这是允许的。在上面程序的基础上，在函数内部添加一行代码，如下所示：

```
1.  name = "Python 教程"
2.  def demo ():
3.      #访问全局变量
4.      print(name)
5.      name = "shell 教程"
6.  demo()
```

执行此程序，Python 解释器报如下错误：

```
UnboundLocalError: local variable 'name' referenced before assignment
```

该错误直译过来的意思是：所访问的 name 变量还未定义。这是什么原因呢？就是我们添加第 5 行代码导致的。

Python 语法规定，在函数内部对不存在的变量赋值时，默认就是重新定义新的局部变量。上面程序中，第 5 行就定义了一个新的 name 局部变量，由于该局部变量名和全局变量名 name 同名，局部 name 变量就会“遮蔽”全局 name 变量，再加上局部变量 name 在 print(name) 后才被初始化，违反了“先定义后使用”的原则，因此程序会报错。

那么，如果就是想在函数中访问甚至修改被“遮蔽”的变量，怎么办呢？可以采取以下 2 中方法：

1. 直接访问被遮蔽的全局变量。如果希望程序依然能访问 name 全局变量，且在函数中可重新定义 name 局部变量，也就是在函数中可以访问被遮蔽的全局变量，此时可通过 globals() 函数来实现，将上面程序改为如下形式即可：

```
name = "Python 教程"
def demo ():
    #通过 globals() 函数访问甚至修改全局变量
    print(globals()['name'])
    globals()['name']="Java 教程"
    #定义局部变量
    name = "shell 教程"
demo()
print(name)
```

程序执行结果为：

```
Python 教程
Java 教程
```

2. 在函数中声明全局变量。为了避免在函数中对全局变量赋值（不是重新定义局部变量），可使用 global 语句来声明全局变量。因此，可将程序改为如下形式：

```
name = "Python 教程"
def demo ():
    global name
    #访问全局 name 变量
    print(name)
    #修改全局 name 变量的值
    name = "shell 教程"
demo()
print(name)
```

程序执行结果为：

```
Python 教程
shell 教程
```

增加了“global name”声明之后，程序会把 name 变量当成全局变量，这意味着 demo() 函数后面对 name 赋值的语句只是对全局变量赋值，而不是重新定义局部变量。

7.16 Python 局部函数及用法（包含 nonlocal 关键字）

通过前面的学习我们知道，Python 函数内部可以定义变量，这样就产生了局部变量，有读者可能会问，Python 函数内部能定义函数吗？答案是肯定的。Python 支持在函数内部定义函数，此类函数又称为局部函数。

那么，局部函数有哪些特征，在使用时需要注意什么呢？接下来就给读者详细介绍 Python 局部函数的用法。

首先，和局部变量一样，默认情况下局部函数只能在其所在函数的作用域内使用。举个例子：

```
#全局函数
def outdef ():
    #局部函数
    def indef():
        print("http://c.biancheng.net/python/")
    #调用局部函数
    indef()
#调用全局函数
outdef()
```

程序执行结果为：

```
http://c.biancheng.net/python/
```

就如同全局函数返回其局部变量，就可以扩大该变量的作用域一样，通过将局部函数作为所在函数的返回值，也可以扩大局部函数的使用范围。例如，修改上面程序为：

```
#全局函数
def outdef ():
    #局部函数
    def indef():
        print("调用局部函数")
    #调用局部函数
    return indef
#调用全局函数
new_indef = outdef()
调用全局函数中的局部函数
new_indef()
```

程序执行结果为：

```
调用局部函数
```

因此，对于局部函数的作用域，可以总结为：如果所在函数没有返回局部函数，则局部函数的可用范围仅限于所在函数内部；反之，如果所在函数将局部函数作为返回值，则局部函数的作用域就会扩大，既可以在所在函数内部使用，也可以在所在函数的作用域中使用。

以上面程序中的 outdef() 和 indef() 为例，如果 outdef() 不将 indef 作为返回值，则 indef() 只能在 outdef() 函数内部使用；反之，则 indef() 函数既可以在 outdef() 函数内部使用，也可以在 outdef() 函数的作用域，也就是全局范围内使用。

有关函数返回函数，更详细的讲解，可阅读《Python 函数高级方法》一节。

另外值得一提的是，如果局部函数中定义有和所在函数中变量同名的变量，也会发生“遮蔽”的问题。例如：

```
1.  #全局函数
2.  def outdef ():
3.      name = "所在函数中定义的 name 变量"
4.      #局部函数
5.      def indef():
6.          print(name)
7.          name = "局部函数中定义的 name 变量"
8.
9.      indef()
10. #调用全局函数
11. outdef()
```

执行此程序，Python 解释器会报如下错误：

```
UnboundLocalError: local variable 'name' referenced before assignment
```

此错误直译过来的意思是“局部变量 name 还没定义就使用”。导致该错误的原因就在于，局部函数 indef() 中定义的 name 变量遮蔽了所在函数 outdef() 中定义的 name 变量。再加上，indef() 函数中 name 变量的定义位于 print() 输出语句之后，导致 print(name) 语句在执行时找不到定义的 name 变量，因此程序报错。

由于这里的 name 变量也是局部变量，因此前面章节讲解的 globals() 函数或者 globals 关键字，并不适用于解决此问题。这里可以使用 Python 提供的 nonlocal 关键字。

例如，修改上面程序为：

```
1.  #全局函数
2.  def outdef ():
3.      name = "所在函数中定义的 name 变量"
```

```
    #局部函数
    def indef():
        nonlocal name
        print(name)
        #修改 name 变量的值
        name = "局部函数中定义的 name 变量"
    indef()
#调用全局函数
outdef()
```

程序执行结果为：

```
所在函数中定义的 name 变量
```

7.17 Python 函数使用方法（高级用法）

前面章节，已经介绍了 Python 函数的所有基本用法和使用注意事项。但是，Python 函数的用法还远不止此，Python 函数还支持赋值、作为其他函数的参数以及作为其他函数的返回值。

首先，Python 允许直接将函数赋值给其它变量，这样做的效果是，程序中也可以用其他变量来调用该函数，更加灵活。例如：

```
def my_def ():
    print("正在执行 my_def 函数")
#将函数赋值给其他变量
other = my_def
#间接调用 my_def() 函数
other()
```

程序执行结果为：

```
正在执行 my_def 函数
```

不仅如此，Python 还支持将函数以参数的形式传入其他函数中。例如：

```
def add (a,b):
    return a+b

def multi(a,b):
    return a*b

def my_def(a,b,dis):
    return dis(a,b)

#求 2 个数的和
print(my_def(3,4,add))
#求 2 个数的乘积
print(my_def(3,4,multi))
```

程序执行结果为：

```
7
12
```

通过分析上面程序不难看出，通过使用函数作为参数，可以在调用函数时动态传入函数，从而实现动态改变函数中的部分实现代码，在不同场景中赋予函数不同的作用。

与此同时，Python 还支持函数的返回值也为函数。例如：

```
def my_def ():
    #局部函数
    def indef():
        print("调用局部函数")
    #调用局部函数
    return indef
other_def = my_def()
#调用局部的 indef() 函数
other_def()
```

程序执行结果为：

```
调用局部函数
```

可以看到，通过返回值为函数的形式，可以扩大局部函数的作用域。

7.18 什么是闭包，Python 闭包（初学者必读）

前面章节中，已经对 Python 闭包做了初步的讲解，本节将详解介绍到底什么是闭包，以及使用闭包有哪些好处。

闭包，又称闭包函数或者闭合函数，其实和前面讲的嵌套函数类似，不同之处在于，闭包中外部函数返回的不是一个具体的值，而是一个函数。一般情况下，返回的函数会赋值给一个变量，这个变量可以在后面被继续执行调用。

例如，计算一个数的 n 次幂，用闭包可以写成下面的代码：

```
#闭包函数，其中 exponent 称为自由变量
def nth_power(exponent):
    def exponent_of(base):
        return base ** exponent
    return exponent_of # 返回值是 exponent_of 函数
square = nth_power(2) # 计算一个数的平方
cube = nth_power(3) # 计算一个数的立方

print(square(2))  # 计算 2 的平方
print(cube(2)) # 计算 2 的立方
```

运行结果为：

```
4
8
```

在上面程序中，外部函数 nth_power() 的返回值是函数 exponent_of()，而不是一个具体的数值。

需要注意的是，在执行完 square = nth_power(2) 和 cube = nth_power(3) 后，外部函数 nth_power() 的参数 exponent 会和内部函数 exponent_of 一起赋值给 squre 和 cube，这样在之后调用 square(2) 或者 cube(2) 时，程序就能顺利地输出结果，而不会报错说参数 exponent 没有定义。

看到这里，读者可能会问，为什么要闭包呢？上面的程序，完全可以写成下面的形式：

```
def nth_power_rewrite(base, exponent):
    return base ** exponent
```

上面程序确实可以实现相同的功能，不过使用闭包，可以让程序变得更简洁易读。设想一下，比如需要计算很多个数的平方，那么读者觉得写成下面哪一种形式更好呢？

```
# 不使用闭包
res1 = nth_power_rewrite(base1, 2)
```

```
res2 = nth_power_rewrite(base2, 2)
res3 = nth_power_rewrite(base3, 2)
# 使用闭包
square = nth_power(2)
res1 = square(base1)
res2 = square(base2)
res3 = square(base3)
```

显然第二种方式表达更为简洁，在每次调用函数时，都可以少输入一个参数。

其次，和缩减嵌套函数的优点类似，函数开头需要做一些额外工作，当需要多次调用该函数时，如果将那些额外工作的代码放在外部函数，就可以减少多次调用导致的不必要开销，提高程序的运行效率。

Python 闭包的__closure__属性

闭包比普通的函数多了一个 __closure__ 属性，该属性记录着自由变量的地址。当闭包被调用时，系统就会根据该地址找到对应的自由变量，完成整体的函数调用。

以 nth_power() 为例，当其被调用时，可以通过 __closure__ 属性获取自由变量（也就是程序中的 exponent 参数）存储的地址，例如：

```
def nth_power(exponent):
    def exponent_of(base):
        return base ** exponent
    return exponent_of
square = nth_power(2)
#查看 __closure__ 的值
print(square.__closure__)
```

输出结果为：

```
(<cell at 0x0000014454DFA948: int object at 0x00000000513CC6D0>,)
```

可以看到，显示的内容是一个 int 整数类型，这就是 square 中自由变量 exponent 的初始值。还可以看到，__closure__ 属性的类型是一个元组，这表明闭包可以支持多个自由变量的形式。

7.19 Python lambda 表达式（匿名函数）及用法

对于定义一个简单的函数，Python 还提供了另外一种方法，即使用本节介绍的 lambda 表达式。

lambda 表达式，又称匿名函数，常用来表示内部仅包含 1 行表达式的函数。如果一个函数的函数体仅有 1 行表达式，则该函数就可以用 lambda 表达式来代替。

lambda 表达式的语法格式如下：

```
name = lambda [list] : 表达式
```

其中，定义 lambda 表达式，必须使用 lambda 关键字；[list] 作为可选参数，等同于定义函数是指定的参数列表；value 为该表达式的名称。

该语法格式转换成普通函数的形式，如下所示：

```
def name(list):
    return 表达式
name(list)
```

显然，使用普通方法定义此函数，需要 3 行代码，而使用 lambda 表达式仅需 1 行。

举个例子，如果设计一个求 2 个数之和的函数，使用普通函数的方式，定义如下：

```
def add(x, y):
    return x+ y
print(add(3,4))
```

程序执行结果为：

```
7
```

由于上面程序中，add() 函数内部仅有 1 行表达式，因此该函数可以直接用 lambda 表达式表示：

```
add = lambda x,y:x+y
print(add(3,4))
```

程序输出结果为：

```
7
```

可以这样理解 lambda 表达式，其就是简单函数（函数体仅是单行的表达式）的简写版本。相比函数，lamba 表达式具有以下 2 个优势：

- 对于单行函数，使用 lambda 表达式可以省去定义函数的过程，让代码更加简洁；
- 对于不需要多次复用的函数，使用 lambda 表达式可以在用完之后立即释放，提高程序执行的性能。

7.20 Python eval()和 exec()函数详解

eval() 和 exec() 函数都属于 Python 的内置函数，由于这两个函数在功能和用法方面都有相似之处，所以将它们放到一节进行介绍。

eval() 和 exec() 函数的功能是相似的，都可以执行一个字符串形式的 Python 代码（代码以字符串的形式提供），相当于一个 Python 的解释器。二者不同之处在于，eval() 执行完要返回结果，而 exec() 执行完不返回结果（文章后续会给出详细示例）。

eval()和 exec()的用法

eval() 函数的语法格式为：

```
eval(source, globals=None, locals=None, /)
```

而 exec() 函数的语法格式如下：

```
exec(source, globals=None, locals=None, /)
```

可以看到，二者的语法格式除了函数名，其他都相同，其中各个参数的具体含义如下：

- expression：这个参数是一个字符串，代表要执行的语句 。该语句受后面两个字典类型参数 globals 和 locals 的限制，只有在 globals 字典和 locals 字典作用域内的函数和变量才能被执行。
- globals：这个参数管控的是一个全局的命名空间，即 expression 可以使用全局命名空间中的函数。如果只是提供了 globals 参数，而没有提供自定义的 __builtins__，则系统会将当前环境中的 __builtins__ 复制到自己提供的 globals 中，然后才会进行计算；如果连 globals 这个参数都没有被提供，则使用 Python 的全局命名空间。
- locals：这个参数管控的是一个局部的命名空间，和 globals 类似，当它和 globals 中有重复或冲突时，以 locals 的为准。如果 locals 没有被提供，则默认为 globals。

注意，__builtins__ 是 Python 的内建模块，平时使用的 int、str、abs 都在这个模块中。通过 print(dic["__builtins__"]) 语句可以查看 __builtins__ 所对应的 value。

首先，通过如下的例子来演示参数 globals 作用域的作用，注意观察它是何时将 __builtins__ 复制 globals 字典中去的：

```
1. dic={} #定义一个字
2. dic['b'] = 3 #在 dic 中加一条元素，key 为 b
3. print (dic.keys()) #先将 dic 的 key 打印出来，有一个元素 b
4. exec("a = 4", dic) #在 exec 执行的语句后面跟一个作用域 dic
5. print(dic.keys()) #exec 后，dic 的 key 多了一个
```

运行结果为：

```
dict_keys(['b'])
dict_keys(['b', '__builtins__', 'a'])
```

上面的代码是在作用域 dic 下执行了一句 a = 4 的代码。可以看出，exec() 之前 dic 中的 key 只有一个 b。执行完 exec() 之后，系统在 dic 中生成了两个新的 key，分别是 a 和 __builtins__。其中，a 为执行语句生成的变量，系统将其放到指定的作用域字典里；__builtins__ 是系统加入的内置 key。

locals 参数的用法就很简单了，举个例子：

```
1. a=10
2. b=20
3. c=30
4. g={'a':6, 'b':8} #定义一个字典
5. t={'b':100, 'c':10} #定义一个字典
6. print(eval('a+b+c', g, t)) #定义一个字典 116
```

输出结果为：

```
116
```

exec()和 eval()的区别

前面已经讲过，它们的区别在于，eval() 执行完会返回结果，而 exec() 执行完不返回结果。举个例子：

```
1. a = 1
2. exec("a = 2") #相当于直接执行 a=2
3. print(a)
4. a = exec("2+3") #相当于直接执行 2+3，但是并没有返回值，a 应为 None
5. print(a)
6. a = eval('2+3') #执行 2+3，并把结果返回给 a
7. print(a)
```

运行结果为：

```
2
None
5
```

可以看出，exec() 中最适合放置运行后没有结果的语句，而 eval() 中适合放置有结果返回的语句。

如果 eval() 里放置一个没有结果返回的语句会怎样呢？例如下面代码：

```
a= eval("a = 2")
```

这时 Python 解释器会报 SyntaxError 错误，提示 eval() 中不识别等号语法。

eval() 和 exec() 函数的应用场景

在使用 Python 开发服务端程序时，这两个函数应用得非常广泛。例如，客户端向服务端发送一段字符串代码，服务端无需关心具体的内容，直接跳过 eval() 或 exec() 来执行，这样的设计会使服务端与客户端的耦合度更低，系统更易扩展。

另外，如果读者以后接触 TensorFlow 框架，就会发现该框架中的静态图就是类似这个原理实现的：

- TensorFlow 中先将张量定义在一个静态图里，这就相当将键值对添加到字典里一样；
- TensorFlow 中通过 session 和张量的 eval() 函数来进行具体值的运算，就当于使用 eval() 函数进行具体值的运算一样。

需要注意的是，在使用 eval() 或是 exec() 来处理请求代码时，函数 eval() 和 exec() 常常会被黑客利用，成为可以执行系统级命令的入口点，进而来攻击网站。解决方法是：通过设置其命名空间里的可执行函数，来限制 eval() 和 exec() 的执行范围。

7.21 Python exec()和 eval()的使用注意事项

使用 exec() 和 eval() 函数时，一定要记住，它们的第一个参数是字符串，而字符串的内容一定要是可执行的代码。

以 eval() 函数为例，用代码演示常犯的错误：

```
1. s="hello"
2. print(eval(s))
```

输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 2, in <module>
    print(eval(s))
  File "<string>", line 1, in <module>
NameError: name 'hello' is not defined
```

上面例子出错的地方在于，字符串的内容是 hello，而 hello 并不是可执行的代码（除非定义了一个变量叫作 hello）。

如果要将字符串 hello 通过 print 函数打印出来，可以写成如下的样子：

```
1. s="hello"
2. print(eval('s'))
```

输出结果为：

```
hello
```

这种写法是要 eval() 执行 "hello" 这句代码。这个 hello 是有引号的，在代码中代表字符串的意思，所以可以执行。

同理，也可以写成这样：

```
1. s='"hello"' #s 是个字符串，字符串的内容是带引号的 hello
2. print(eval(s))
```

输出结果为：

```
hello
```

这种写法的意思是 s 是个字符串，并且其内容是个带引号的 hello。所以直接将 s 放入到函数 eval() 中也可以执行。

除了以上这种方式，还可以不去改变原有字符串 s 的写法，直接使用 repr() 函数来进行转化，也可以得到同样的效果。例如：

```
s="hello"
print(eval(repr(s))) #使用函数 repr() 进行转化
```

输出结果为：

```
hello
```

注意，虽然函数 eval() 与 str() 的返回值都是字符串。但是使用 str() 函数对 s 进行转化，程序同样会报错，例如：

```
s="hello"
print(eval(str(s)))
```

输出结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 2, in <module>
    print(eval(str(s)))
  File "<string>", line 1, in <module>
NameError: name 'hello' is not defined
```

为什么会有这个区别呢？同样对带字符串 s 的转化，使用 repr() 与 str() 得到的结果是有差别的，直接将二者的结果打印出来，就可以很明显地看出不同。见下面代码：

```
s="hello"
print(repr(s))
print(str(s))
```

输出结果为：

```
'hello'
hello
```

可见使用 repr() 返回的内容，输出后会在两边多一个单引号。

注意，在编写代码时，一般会使 repr() 函数来生成动态的字符串，再传入到 eval() 或 exec() 函数内，实现动态执行代码的功能。

7.22Python 函数式编程（ map()、filter()和 reduce() ）详解

所谓函数式编程，是指代码中每一块都是不可变的，都由纯函数的形式组成。这里的纯函数，是指函数本身相互独立、互不影响，对于相同的输入，总会有相同的输出。

除此之外，函数式编程还具有一个特点，即允许把函数本身作为参数传入另一个函数，还允许返回一个函数。

例如，想让列表中的元素值都变为原来的两倍，可以使用如下函数实现：

```
def multiply_2(list):
    for index in range(0, len(list)):
        list[index] *= 2
    return list
```

需要注意的是，这段代码不是一个纯函数的形式，因为列表中元素的值被改变了，如果多次调用 multiply_2() 函数，那么每次得到的结果都不一样。

而要想让 multiply_2() 成为一个纯函数的形式，就得重新创建一个新的列表并返回，也就是写成下面这种形式：

```
def multiply_2_pure(list):
    new_list = []
    for item in list:
        new_list.append(item * 2)
    return new_list
```

函数式编程的优点，主要在于其纯函数和不可变的特性使程序更加健壮，易于调试和测试；缺点主要在于限制多，难写。
注意，纯粹的函数式编程语言（比如 Scala ），其编写的函数中是没有变量的，因此可以保证，只要输入是确定的，输出就是确定的；而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出。

Python 允许使用变量，所以它并不是一门纯函数式编程语言。Python 仅对函数式编程提供了部分支持，主要包括 map()、filter() 和 reduce() 这 3 个函数，它们通常都结合 lambda 匿名函数一起使用。接下来就对这 3 个函数的用法做逐一介绍。

Python map()函数

map() 函数的基本语法格式如下：

```
map(function, iterable)
```

其中，function 参数表示要传入一个函数，其可以是内置函数、自定义函数或者 lambda 匿名函数；iterable 表示一个或多个可迭代对象，可以是列表、字符串等。

map() 函数的功能是对可迭代对象中的每个元素，都调用指定的函数，并返回一个 map 对象。

注意，该函数返回的是一个 map 对象，不能直接输出，可以通过 for 循环或者 list() 函数来显示。
【例 1】还是对列表中的每个元素乘以 2。

```
listDemo = [1, 2, 3, 4, 5]
new_list = map(lambda x: x * 2, listDemo)
print(list(new_list))
```

运行结果为：

```
[2 ， 4 ， 6 ， 8 ， 10]
```

【例 2】map() 函数可传入多个可迭代对象作为参数。

```
listDemo1 = [1, 2, 3, 4, 5]
listDemo2 = [3, 4, 5, 6, 7]
new_list = map(lambda x,y: x + y, listDemo1,listDemo2)
print(list(new_list))
```

运行结果为：

```
[4, 6, 8, 10, 12]
```

注意，由于 map() 函数是直接由用 C 语言写的，运行时不需要通过 Python 解释器间接调用，并且内部做了诸多优化，所以相比其他方法，此方法的运行效率最高。

Python filter()函数

filter()函数的基本语法格式如下：

```
filter(function, iterable)
```

此格式中，funcition 参数表示要传入一个函数，iterable 表示一个可迭代对象。

filter() 函数的功能是对 iterable 中的每个元素，都使用 function 函数判断，并返回 True 或者 False，最后将

返回 True 的元素组成一个新的可遍历的集合。

【例 3】返回一个列表中的所有偶数。

```
listDemo = [1, 2, 3, 4, 5]
new_list = filter(lambda x: x % 2 == 0, listDemo)
print(list(new_list))
```

运行结果为：

```
[2, 4]
```

【例 4】filter() 函数可以接受多个可迭代对象。

```
listDemo = [1, 2, 3, 4, 5]
new_list = map(lambda x,y: x-y>0,[3,5,6],[1,5,8] )
print(list(new_list))
```

运行结果为：

```
[True, False, False]
```

Python reduce()函数

reduce() 函数通常用来对一个集合做一些累积操作，其基本语法格式为：

```
reduce(function, iterable)
```

其中，function 规定必须是一个包含 2 个参数的函数；iterable 表示可迭代对象。

注意，由于 reduce() 函数在 Python 3.x 中已经被移除，放入了 functools 模块，因此在使用该函数之前，需先导入 functools 模块。

【例 5】计算某个列表元素的乘积。

```
import functools
listDemo = [1, 2, 3, 4, 5]
product = functools.reduce(lambda x, y: x * y, listDemo)
print(product)
```

运行结果为：

```
120
```

总结

通常来说，当对集合中的元素进行一些操作时，如果操作非常简单，比如相加、累积这种，那么应该优先考虑使用 map()、filter()、reduce() 实现。另外，在数据量非常多的情况下（比如机器学习的应用），一般更倾向于函数式编程的表示，因为效率更高。

当然，在数据量不多的情况下，使用 for 循环等方式也可以。不过，如果要对集合中的元素做一些比较复杂的操作，考虑到代码的可读性，通常会使用 for 循环。

7.23 Python 3 函数注解：为函数提供类型提示信息

函数注解是 Python 3 最独特的功能之一，关于它的介绍，官方文档是这么说的，“函数注解是关于用户自定义函数使用类型的完全可选的元信息”。也就是说，官方将函数注解的用途归结为：为函数中的形参和返回值提供类型提示信息。

下面是对 Python 官方文档中的示例稍作修改后的程序，可以很好的展示如何定义并获取函数注解：

```
1. def f(ham:str,egg:str='eggs')->str:
2.   pass
3. print(f.__annotations__)
```

输出结果为：

```
{'ham': <class 'str'>, 'egg': <class 'str'>, 'return': <class 'str'>}
```

如上所示，给函数中的参数做注解的方法是在形参后添加冒号“：”，后接需添加的注解（可以是类（如 str、int 等），也可以是字符串或者表示式）；给返回值做注解的方法是将注解添加到 def 语句结尾的冒号和 -> 之间。

注意，如果参数有默认值，参数注解位于冒号和等号之间。比如 eggs:str='eggs'，它表示 eggs 参数的默认值为 'eggs'，添加的注解为 str。
给函数定义好注解之后，可以通过函数对象的 __annotations__ 属性获取，它是一个字典，在应用运行期间可以获取。这里再举一个例子：

```
1. def square(number:"一个数字")->"返回 number 的平方":
2.   return number**2
3. print(square(10))
4. print(square.__annotations__)
```

运行结果为：

```
100
{'number': '一个数字', 'return': '返回 number 的平方'}
```

事实上，函数注解并不局限于类型提示，而且在 Python 及其标准库中也没有单个功能可以利用这种注解，这也是这个功能独特的原因。

注意，函数注解没有任何语法上的意义，只是为函数参数和返回值做注解，并在运行获取这些注解，仅此而已。换句话说，为函数做的注解，Python 不做检查，不做强制，不做验证，什么操作都不做，函数注解对 Python 解释器没任何意义。

函数注解可能的用法

PEP 3107 作为提议函数注解的官方文档，其中列出了以下可能的使用场景：

- 提供类型信息：包括类型检查、让 IDE 显示函数接受和返回的类型、适配、与其他语言的桥梁、数据库查询映射、RPC 参数编组等；
- 其他信息：函数参数和返回值的文档。

总之，虽然函数注解存在的时间和 Python 3 一样长，但目前仍未找到任一常见且积极维护的包，将函数注解用作类型检查之外的功能。Python 3 最初发布时包含函数注解的最初目的也仅是用于试验和玩耍。

7.24 提高代码可读性和颜值的几点建议（初学者必读）

学习过程中，我们经常会阅读他人写的代码，如果注意观察就会发现，好的代码本身就是一份文档，解决同样的问题，不同的人编写的代码，其可读性千差万别。

有些人的设计风格和代码风格犹如热刀切黄油，从顶层到底层的代码看下来酣畅淋漓，注释详尽又精简；深入到细节代码，无需注释也能理解清清楚楚。而有些人，代码勉勉强强能跑起来，遇到稍微复杂的情况就会出崩溃，且代码中处处都是堆积在一起的变量、函数和类，很难理清代码的实现思路。

Python 创始人 Guido van Rossum（吉多·范罗苏姆）说过，代码的阅读频率远高于编写代码的频率。毕竟是在编写代码的时候，我们自己也需要对代码进行反复阅读和调试，来确认代码能够按照期望运行。

本节，在读者学会如何使用 Puython 函数的基础上，教大家怎么才能合理分解代码，提高代码的可读性。

首先，大家在编程过程中，一定要围绕一个中心思想：不写重复性的代码。因为，重复代码往往是可以通过使用条件、循环、构造函数和类（后续章节会做详细介绍）来解决的。

例如，仔细观察下面的代码：

```
if i_am_rich:
    money = 100
    send(money)
else:
    money = 10
    send(money)
```

这段代码中，同样的 send 语句出现了两次，其实它完全可以进行合并，把代码改造成下面这样：

```
if i_am_rich:
    money = 100
else:
    money = 10
send(money)
```

与此同时，还要学会刻意地减少代码的迭代层数，尽可能让 Python 代码扁平化。例如：

```
def send(money):
    if is_server_dead:
        LOG('server dead')
        return
```

```
    else:
        if is_server_timed_out:
            LOG('server timed out')
            return
        else:
            result = get_result_from_server()
            if result == MONEY_IS_NOT_ENOUGH:
                LOG('you do not have enough money')
                return
            else:
                if result == TRANSACTION_SUCCEED:
                    LOG('OK')
                    return
                else:
                    LOG('something wrong')
                    return
```

上面这段代码层层缩进，如果我们没有比较强的逻辑分析能力，理清这段代码是比较困难。其实，这段代码完全可以改成如下这样：

```
def send(money):
    if is_server_dead:
        LOG('server dead')
        return

    if is_server_timed_out:
        LOG('server timed out')
        return

    result = get_result_from_server()

    if result == MONET_IS_NOT_ENOUGH:
        LOG('you do not have enough money')
        return

    if result == TRANSACTION_SUCCEED:
        LOG('OK')
```

```
        return

    LOG('something wrong')
```

可以看到，所有的判断语句都位于同一层级，同之前的代码格式相比，代码层次清晰了很多。

另外，在使用函数时，函数的粒度应该尽可能细，不要让一个函数做太多的事情。往往一个复杂的函数，我们要尽可能地把它拆分成几个功能简单的函数，然后合并起来。

如何拆分函数呢？这里，举一个二分搜索的例子。给定一个非递减整数数组，和一个 target 值，要求你找到数组中最小的一个数 x，满足 x*x > target，如果不存在，则返回 -1。

大家不妨先独立完成，写完后再对照着来看下面的代码，找出自己的问题：

```
def solve(arr, target):
    l, r = 0, len(arr) - 1
    ret = -1
    while l <= r:
        m = (l + r) // 2
        if arr[m] * arr[m] > target:
            ret = m
            r = m - 1
        else:
            l = m + 1
    if ret == -1:
        return -1
    else:
        return arr[ret]

print(solve([1, 2, 3, 4, 5, 6], 8))
print(solve([1, 2, 3, 4, 5, 6], 9))
print(solve([1, 2, 3, 4, 5, 6], 0))
print(solve([1, 2, 3, 4, 5, 6], 40))
```

对于上面这样的写法，应付算法比赛和面试已经绰绰有余。但如果从工程的角度考虑，还需要进行深度优化：

```
def comp(x, target):
    return x * x > target

```

```
def binary_search(arr, target):
    l, r = 0, len(arr) - 1
    ret = -1
    while l <= r:
        m = (l + r) // 2
        if comp(arr[m], target):
            ret = m
            r = m - 1
        else:
            l = m + 1
    return ret

def solve(arr, target):
    id = binary_search(arr, target)
    if id != -1:
        return arr[id]
    return -1

print(solve([1, 2, 3, 4, 5, 6], 8))
print(solve([1, 2, 3, 4, 5, 6], 9))
print(solve([1, 2, 3, 4, 5, 6], 0))
print(solve([1, 2, 3, 4, 5, 6], 40))
```

在这段代码中，我们把不同功能的代码单独提取出来作为独立的函数。其中，comp() 函数作为核心判断，提取出来之后，可以让整个程序更清晰；同时，还把二分搜索的主程序提取了出来，只负责二分搜索；最后的 solve() 函数拿到结果，决定返回不存在，还是返回值。这样一来，每个函数各司其职，阅读性也能得到一定提高。

第 8 章：Python 类和对象

8.1 什么是面向对象，Python 面向对象（一切皆对象）

读者肯定听过 Python 中“一切皆对象”的说法，但可能并不了解它的具体含义，只是在学习的时候听说 Python 是面向对象的编程语言，本节将向大家详细介绍 Python 面向对象的含义。

面向对象编程是在面向过程编程的基础上发展来的，它比面向过程编程具有更强的灵活性和扩展性。面向对象编程是程序员发展的分水岭，很多初学者会因无法理解面向对象而放弃学习编程。

面向对象编程（Object-oriented Programming，简称 OOP），是一种封装代码的方法。其实，在前面章节的学习中，我们已经接触了封装，比如说，将乱七八糟的数据扔进列表中，这就是一种简单的封装，是数据层面的封装；把常用的代码块打包成一个函数，这也是一种封装，是语句层面的封装。

> 代码封装，其实就是隐藏实现功能的具体代码，仅留给用户使用的接口，就好像使用计算机，用户只需要使用键盘、鼠标就可以实现一些功能，而根本不需要知道其内部是如何工作的。

本节所讲的面向对象编程，也是一种封装的思想，不过显然比以上两种封装更先进，它可以更好地模拟真实世界里的事物（将其视为对象），并把描述特征的数据和代码块（函数）封装到一起。

打个比方，若在某游戏中设计一个乌龟的角色，应该如何来实现呢？使用面向对象的思想会更简单，可以分为如下两个方面进行描述：

1. 从表面特征来描述，例如，绿色的、有 4 条腿、重 10 kg、有外壳等等。
2. 从所具有的的行为来描述，例如，它会爬、会吃东西、会睡觉、会将头和四肢缩到壳里，等等。

如果将乌龟用代码来表示，则其表面特征可以用变量来表示，其行为特征可以通过建立各种函数来表示。参考代码如下所示：

```
class tortoise:
    bodyColor = "绿色"
    footNum = 4
    weight = 10
    hasShell = True

    #会爬
    def crawl(self):
        print("乌龟会爬")
    #会吃东西
```

```
    def eat(self):
        print("乌龟吃东西")
    #会睡觉
    def sleep(self):
        print("乌龟在睡觉")
    #会缩到壳里
    def protect(self):
        print("乌龟缩进了壳里")
```

> 注意，以上代码仅是为了演示面向对象的编程思想，具体细节后续会做详细介绍。

因此，从某种程序上，相比较只用变量或只用函数，使用面向对象的思想可以更好地模拟现实生活中的事物。

不仅如此，在 Python 中，所有的变量其实也都是对象，包括整形（int）、浮点型（float）、字符串（str）、列表(list)、元组(tuple)、字典（dict）和集合（set）。以字典（dict）为例，它包含多个函数供我们使用，例如使用 keys() 获取字典中所有的键，使用 values() 获取字典中所有的值，使用 item() 获取字典中所有的键值对，等等。

面向对象相关术语

在系统学习面向对象编程之前，初学者要了解有关面向对象的一些术语。当和其他人讨论代码的时候，或者尝试查找我们遇到的问题的解决方案时，知道正确的术语会很有帮助。

面向对象中，常用术语包括：

- 类：可以理解是一个模板，通过它可以创建出无数个具体实例。比如，前面编写的 tortoise 表示的只是乌龟这个物种，通过它可以创建出无数个实例来代表各种不同特征的乌龟（这一过程又称为类的实例化）。
- 对象：类并不能直接使用，通过类创建出的实例（又称对象）才能使用。这有点像汽车图纸和汽车的关系，图纸本身（类）并不能为人们使用，通过图纸创建出的一辆辆车（对象）才能使用。
- 属性：类中的所有变量称为属性。例如，tortoise 这个类中，bodyColor、footNum、weight、hasShell 都是这个类拥有的属性。
- 方法：类中的所有函数通常称为方法。不过，和函数所有不同的是，类方法至少要包含一个 self 参数（后续会做详细介绍）。例如，tortoise 类中，crawl()、eat()、sleep()、protect() 都是这个类所拥有的方法，类方法无法单独使用，只能和类的对象一起使用。

8.2 Python class：定义类（入门必读）

前面章节中已经提到，类仅仅充当图纸的作用，本身并不能直接拿来用，而只有根据图纸造出的实际物品（对象）才能直接使用。因此，Python 程序中类的使用顺序是这样的：

1. 创建（定义）类，也就是制作图纸的过程；
2. 创建类的实例对象（根据图纸造出实际的物品），通过实例对象实现特定的功能。

本节先教大家如何创建（定义）一个类，如何使用定义好的类将放到后续章节进行讲解。

Python 类的定义

Python 中定义一个类使用 class 关键字实现，其基本语法格式如下：

```
class 类名：
    多个（≥0）类属性...
    多个（≥0）类方法...
```

注意，无论是类属性还是类方法，对于类来说，它们都不是必需的，可以有也可以没有。另外，Python 类中属性和方法所在的位置是任意的，即它们之间并没有固定的前后次序。

和变量名一样，类名本质上就是一个标识符，因此我们在给类起名字时，必须让其符合 Python 的语法。有读者可能会问，用 a、b、c 作为类的类名可以吗？从 Python 语法上讲，是完全没有问题的，但作为一名合格的程序员，我们必须还要考虑程序的可读性。

因此，在给类起名字时，最好使用能代表该类功能的单词，例如用“Student”作为学生类的类名；甚至如果必要，可以使用多个单词组合而成，例如初学者定义的第一个类的类名可以是“TheFirstDemo”。

注意，如果由单词构成类名，建议每个单词的首字母大写，其它字母小写。
给类起好名字之后，其后要跟有冒号（：），表示告诉 Python 解释器，下面要开始设计类的内部功能了，也就是编写类属性和类方法。

其实，类属性指的就是包含在类中的变量；而类方法指的是包含类中的函数。换句话说，类属性和类方法其实分别是包含类中的变量和函数的别称。需要注意的一点是，同属一个类的所有类属性和类方法，要保持统一的缩进格式，通常统一缩进 4 个空格。

Python 变量和函数的使用，前面章节中已经做了详细的介绍，这里不再重复赘述。
通过上面的分析，可以得出这样一个结论，即 Python 类是由类头（class 类名）和类体（统一缩进的变量和函数）构成。例如，下面程序定义一个 TheFirstDemo 类：

```
1.  class TheFirstDemo:
```

```
    '''这是一个学习 Python 定义的第一个类'''
    # 下面定义了一个类属性
    add = 'http://c.biancheng.net'
    # 下面定义了一个 say 方法
    def say(self, content):
        print(content)
```

和函数一样，我们也可以为类定义说明文档，其要放到类头之后，类体之前的位置，如上面程序中第二行的字符串，就是 TheFirstDemo 这个类的说明文档。

另外分析上面的代码可以看到，我们创建了一个名为 TheFirstDemo 的类，其包含了一个名为 add 的类属性。注意，根据定义属性位置的不同，在各个类方法之外定义的变量称为类属性或类变量（如 add 属性），而在类方法中定义的属性称为实例属性（或实例变量），它们的区别和用法可阅读《Python 类变量和实例变量》一节。

同时，TheFirstDemo 类中还包含一个 say() 类方法，细心的读者可能已经看到，该方法包含两个参数，分别是 self 和 content。可以肯定的是，content 参数就只是一个普通参数，没有特殊含义，但 self 比较特殊，并不是普通的参数，它的作用会在后续章节中详细介绍。

更确切地说，say() 是一个实例方法，除此之外，Python 类中还可以定义类方法和静态方法，这 3 种类方法的区别和具体用法，可阅读《Python 实例方法、静态方法和类方法》。

事实上，我们完全可以创建一个没有任何类属性和类方法的类，换句话说，Python 允许创建空类，例如：

```
class Empty:
    pass
```

可以看到，如果一个类没有任何类属性和类方法，那么可以直接用 pass 关键字作为类体即可。但在实际应用中，很少会创建空类，因为空类没有任何实际意义。

8.3 Python __init__()类构造方法

在创建类时，我们可以手动添加一个 __init__() 方法，该方法是一个特殊的类实例方法，称为构造方法（或构造函数）。

构造方法用于创建对象时使用，每当创建一个类的实例对象时，Python 解释器都会自动调用它。Python 类中，手动添加构造方法的语法格式如下：

```
def __init__(self,...):
    代码块
```

注意，此方法的方法名中，开头和结尾各有 2 个下划线，且中间不能有空格。Python 中很多这种以双下划线开头、双下划线结尾的方法，都具有特殊的意义，后续会一一为大家讲解。

另外，__init__() 方法可以包含多个参数，但必须包含一个名为 self 的参数，且必须作为第一个参数。也就是说，类的构造方法最少也要有一个 self 参数。例如，仍以 TheFirstDemo 类为例，添加构造方法的代码如下所示：

```
1.  class TheFirstDemo:
2.      '''这是一个学习 Python 定义的第一个类'''
3.      #构造方法
4.      def __init__(self):
5.          print("调用构造方法")
6.      # 下面定义了一个类属性
7.      add = 'http://c.biancheng.net'
8.      # 下面定义了一个 say 方法
9.      def say(self, content):
10.         print(content)
```

注意，即便不手动为类添加任何构造方法，Python 也会自动为类添加一个仅包含 self 参数的构造方法。

仅包含 self 参数的 __init__() 构造方法，又称为类的默认构造方法。

在上面代码的后面，顶头（不缩进）直接添加如下代码：

```
1.  zhangsan = TheFirstDemo()
```

这行代码的含义是创建一个名为 zhangsan 的 TheFirstDemo 类对象。运行代码可看到如下结果：

```
调用构造方法
```

显然，在创建 zhangsan 这个对象时，隐式调用了我们手动创建的 __init__() 构造方法。

不仅如此，在 __init__() 构造方法中，除了 self 参数外，还可以自定义一些参数，参数之间使用逗号“,”进行分割。例如，下面的代码在创建 __init__() 方法时，额外指定了 2 个参数：

```
class CLanguage:
    '''这是一个学习 Python 定义的一个类'''
    def __init__(self,name,add):
        print(name,"的网址为:",add)
#创建 add 对象，并传递参数给构造函数
add = CLanguage("C 语言中文网","http://c.biancheng.net")
```

注意，由于创建对象时会调用类的构造方法，如果构造函数有多个参数时，需要手动传递参数，传递方式如代码中所示（后续章节会做详细讲解）。

运行以上代码，执行结果为：

```
C 语言中文网 的网址为: http://c.biancheng.net
```

可以看到，虽然构造方法中有 self、name、add 3 个参数，但实际需要传参的仅有 name 和 add，也就是说，self 不需要手动传递参数。

关于 self 参数，后续章节会做详细介绍，这里只需要知道，在创建类对象时，无需给 self 传参即可。

8.4 Python 类对象的创建和使用

通过前面章节的学习，我们已经学会如何定义一个类，但要想使用它，必须创建该类的对象。

创建类对象的过程，又称为类的实例化。

Python 类的实例化

对已定义好的类进行实例化，其语法格式如下：

```
类名(参数)
```

定义类时，如果没有手动添加 __init__() 构造方法，又或者添加的 __init__() 中仅有一个 self 参数，则创建类对象时的参数可以省略不写。

例如，如下代码创建了名为 CLanguage 的类，并对其进行了实例化：

```
1.  class CLanguage :
2.      # 下面定义了 2 个类变量
3.      name = "C 语言中文网"
4.      add = "http://c.biancheng.net"
5.      def __init__(self,name,add):
6.          #下面定义 2 个实例变量
7.          self.name = name
8.          self.add = add
9.          print(name,"网址为：",add)
10.     # 下面定义了一个 say 实例方法
11.     def say(self, content):
12.         print(content)
13. # 将该 CLanguage 对象赋给 clanguage 变量
14. clanguage = CLanguage("C 语言中文网","http://c.biancheng.net")
```

在上面的程序中，由于构造方法除 self 参数外，还包含 2 个参数，且这 2 个参数没有设置默认参数，因此在实例化类对象时，需要传入相应的 name 值和 add 值（self 参数是特殊参数，不需要手动传值，Python 会自动传给它值）。

类变量和实例变量，简单地理解，定义在各个类方法之外（包含在类中）的变量为类变量（或者类属性），定义在类方法中的变量为实例变量（或者实例属性），二者的具体区别和用法可阅读《Python 类变量和实例变量》

Python 类对象的使用

定义的类只有进行实例化，也就是使用该类创建对象之后，才能得到利用。总的来说，实例化后的类对象可以执行以下操作：

- 访问或修改类对象具有的实例变量，甚至可以添加新的实例变量或者删除已有的实例变量；
- 调用类对象的方法，包括调用现有的方法，以及给类对象动态添加方法。

类对象访问变量或方法

使用已创建好的类对象访问类中实例变量的语法格式如下：

```
类对象名.变量名
```

使用类对象调用类中方法的语法格式如下：

```
对象名.方法名(参数)
```

注意，对象名和变量名以及方法名之间用点 "." 连接。

例如，下面代码演示了如何通过 clanguage 对象调用类中的实例变量和方法：

```
#输出 name 和 add 实例变量的值
print(clanguage.name,clanguage.add)
#修改实例变量的值
clanguage.name="Python 教程"
clanguage.add="http://c.biancheng.net/python"
#调用 clanguage 的 say()方法
clanguage.say("人生苦短，我用 Python")
#再次输出 name 和 add 的值
print(clanguage.name,clanguage.add)
```

程序运行结果为：

```
C 语言中文网 网址为： http://c.biancheng.net
C 语言中文网 http://c.biancheng.net
人生苦短，我用 Python
Python 教程 http://c.biancheng.net/python
```

给类对象动态添加/删除变量

Python 支持为已创建好的对象动态增加实例变量，方法也很简单，举个例子：

```
# 为 clanguage 对象增加一个 money 实例变量
clanguage.money= 159.9
print(clanguage.money)
```

运行结果为：

```
159.9
```

可以看到，通过直接增加一个新的实例变量并为其赋值，就成功地为 clanguage 对象添加了 money 变量。

既然能动态添加，那么是否能动态删除呢？答案是肯定的，使用 del 语句即可实现，例如：

```
#删除新添加的 money 实例变量
del clanguage.money
#再次尝试输出 money，此时会报错
print(clanguage.money)
```

运行程序会发现，结果显示 AttributeError 错误：

```
Traceback (most recent call last):
  File "C:/Users/mengma/Desktop/1.py", line 29, in <module>
    print(clanguage.money)
AttributeError: 'CLanguage' object has no attribute 'money'
```

给类对象动态添加方法

注意，初学者在理解下面内容之前，需明白 self 参数的含义和作用，可阅读《Python self 用法》详细了解。

Python 也允许为对象动态增加方法。以本节开头的 Clanguage 类为例，由于其内部只包含一个 say() 方法，因此该类实例化出的 clanguage 对象也只包含一个 say() 方法。但其实，我们还可以为 clanguage 对象动态添加其它方法。

需要注意的一点是，为 clanguage 对象动态增加的方法，Python 不会自动将调用者自动绑定到第一个参数（即使将第一个参数命名为 self 也没用）。例如如下代码：

```
# 先定义一个函数
def info(self):
    print("---info 函数---", self)
# 使用 info 对 clanguage 的 foo 方法赋值（动态绑定方法）
clanguage.foo = info
# Python 不会自动将调用者绑定到第一个参数，
# 因此程序需要手动将调用者绑定为第一个参数
clanguage.foo(clanguage)  # ①
```

```
9.
10. # 使用 lambda 表达式为 clanguage 对象的 bar 方法赋值（动态绑定方法）
11. clanguage.bar = lambda self: print('--lambda 表达式--', self)
12. clanguage.bar(clanguage) # ②
```

上面的第 5 行和第 11 行代码分别使用函数、lambda 表达式为 clanguage 对象动态增加了方法，但对于动态增加的方法，Python 不会自动将方法调用者绑定到它们的第一个参数，因此程序必须手动为第一个参数传入参数值，如上面程序中 ① 号、② 号代码所示。

有读者可能会问，有没有不用手动给 self 传值的方法呢？通过借助 types 模块下的 MethodType 可以实现，仍以上面的 info() 函数为例：

```
1. def info(self,content):
2.     print("C 语言中文网地址为：%s" % content)
3. # 导入 MethodType
4. from types import MethodType
5. clanguage.info = MethodType(info, clanguage)
6. # 第一个参数已经绑定了，无需传入
7. clanguage.info("http://c.biancheng.net")
```

可以看到，由于使用 MethodType 包装 info() 函数时，已经将该函数的 self 参数绑定为 clanguage，因此后续再使用 info() 函数时，就不用再给 self 参数绑定值了。

8.5 Python self 用法详解

在定义类的过程中，无论是显式创建类的构造方法，还是向类中添加实例方法，都要求将 self 参数作为方法的第一个参数。例如，定义一个 Person 类：

```
1.  class Person:
2.      def __init__(self):
3.          print("正在执行构造方法")
4.      # 定义一个 study()实例方法
5.      def study(self,name):
6.          print(name,"正在学 Python")
```

那么，self 到底扮演着什么样的角色呢？本节就对 self 参数做详细的介绍。

事实上，Python 只是规定，无论是构造方法还是实例方法，最少要包含一个参数，并没有规定该参数的具体名称。之所以将其命名为 self，只是程序员之间约定俗成的一种习惯，遵守这个约定，可以使我们编写的代码具有更好的可读性（大家一看到 self，就知道它的作用）。

那么，self 参数的具体作用是什么呢？打个比方，如果把类比作造房子的图纸，那么类实例化后的对象是真正可以住的房子。根据一张图纸（类），我们可以设计出成千上万的房子（类对象），每个房子长相都是类似的（都有相同的类变量和类方法），但它们都有各自的主人，那么如何对它们进行区分呢？

当然是通过 self 参数，它就相当于每个房子的门钥匙，可以保证每个房子的主人仅能进入自己的房子（每个类对象只能调用自己的类变量和类方法）。

> 如果你接触过其他面向对象的编程语言（例如 C++），其实 Python 类方法中的 self 参数就相当于 C++ 中的 this 指针。

也就是说，同一个类可以产生多个对象，当某个对象调用类方法时，该对象会把自身的引用作为第一个参数自动传给该方法，换句话说，Python 会自动绑定类方法的第一个参数指向调用该方法的对象。如此，Python 解释器就能知道到底要操作哪个对象的方法了。

因此，程序在调用实例方法和构造方法时，不需要手动为第一个参数传值。例如，更改前面的 Person 类，如下所示：

```
1.  class Person:
2.      def __init__(self):
3.          print("正在执行构造方法")
4.      # 定义一个 study()实例方法
5.      def study(self):
6.          print(self,"正在学 Python")
7.  zhangsan = Person()
```

```
zhangsan.study()
lisi = Person()
lisi.study()
```

上面代码中，study() 中的 self 代表该方法的调用者，即谁调用该方法，那么 self 就代表谁。因此，该程序的运行结果为：

```
正在执行构造方法
<__main__.Person object at 0x0000021ADD7D21D0> 正在学 Python
正在执行构造方法
<__main__.Person object at 0x0000021ADD7D2E48> 正在学 Python
```

另外，对于构造函数中的 self 参数，其代表的是当前正在初始化的类对象。举个例子：

```
class Person:
    name = "xxx"
    def __init__(self,name):
        self.name=name

zhangsan = Person("zhangsan")
print(zhangsan.name)
lisi = Person("lisi")
print(lisi.name)
```

运行结果为：

```
zhangsan
lisi
```

可以看到，zhangsan 在进行初始化时，调用的构造函数中 self 代表的是 zhangsan；而 lisi 在进行初始化时，调用的构造函数中 self 代表的是 lisi。

值得一提的是，除了类对象可以直接调用类方法，还有一种函数调用的方式，例如：

```
class Person:
    def who(self):
        print(self)
zhangsan = Person()
#第一种方式
zhangsan.who()
#第二种方式
```

```
8.  who = zhangsan.who
9.  who()#通过 who 变量调用 zhangsan 对象中的 who() 方法
```

运行结果为：

```
<__main__.Person object at 0x0000025C26F021D0>
<__main__.Person object at 0x0000025C26F021D0>
```

显然，无论采用哪种方法，self 所表示的都是实际调用该方法的对象。

总之，无论是类中的构造函数还是普通的类方法，实际调用它们的谁，则第一个参数 self 就代表谁。

8.6 Python 类变量和实例变量（类属性和实例属性）

无论是类属性还是类方法，都无法像普通变量或者函数那样，在类的外部直接使用它们。我们可以将类看做一个独立的空间，则类属性其实就是在类体中定义的变量，类方法是在类体中定义的函数。

前面章节提到过，在类体中，根据变量定义的位置不同，以及定义的方式不同，类属性又可细分为以下 3 种类型：

1. 类体中、所有函数之外：此范围定义的变量，称为类属性或类变量；
2. 类体中，所有函数内部：以“self.变量名”的方式定义的变量，称为实例属性或实例变量；
3. 类体中，所有函数内部：以“变量名=变量值”的方式定义的变量，称为局部变量。

不仅如此，类方法也可细分为实例方法、静态方法和类方法，后续章节会做详细介绍。
那么，类变量、实例变量以及局部变量之间有哪些不同呢？接下来就围绕此问题做详细地讲解。

类变量（类属性）

类变量指的是在类中，但在各个类方法外定义的变量。举个例子：

```
class CLanguage :
    # 下面定义了 2 个类变量
    name = "C 语言中文网"
    add = "http://c.biancheng.net"
    # 下面定义了一个 say 实例方法
    def say(self, content):
        print(content)
```

上面程序中，name 和 add 就属于类变量。

类变量的特点是，所有类的实例化对象都同时共享类变量，也就是说，类变量在所有实例化对象中是作为公用资源存在的。类方法的调用方式有 2 种，既可以使用类名直接调用，也可以使用类的实例化对象调用。

比如，在 CLanguage 类的外部，添加如下代码：

```
#使用类名直接调用
print(CLanguage.name)
print(CLanguage.add)
#修改类变量的值
CLanguage.name = "Python 教程"
CLanguage.add = "http://c.biancheng.net/python"
print(CLanguage.name)
```

```
print(CLanguage.add)
```

程序运行结果为：

```
C 语言中文网
http://c.biancheng.net
Python 教程
http://c.biancheng.net/python
```

可以看到，通过类名不仅可以调用类变量，也可以修改它的值。

当然，也可以使用类对象来调用所属类中的类变量（此方式不推荐使用，原因后续会讲）。例如，在 CLanguage 类的外部，添加如下代码：

```
clang = CLanguage()
print(clang.name)
print(clang.add)
```

运行程序，结果为：

```
C 语言中文网
http://c.biancheng.net
```

注意，因为类变量为所有实例化对象共有，通过类名修改类变量的值，会影响所有的实例化对象。例如，在 CLanguage 类体外部，添加如下代码：

```
print("修改前，各类对象中类变量的值：")
clang1 = CLanguage()
print(clang1.name)
print(clang1.add)
clang2 = CLanguage()
print(clang2.name)
print(clang2.add)

print("修改后，各类对象中类变量的值：")
CLanguage.name = "Python 教程"
CLanguage.add = "http://c.biancheng.net/python"
print(clang1.name)
print(clang1.add)
print(clang2.name)
print(clang2.add)
```

程序运行结果为：

```
修改前，各类对象中类变量的值：
C 语言中文网
http://c.biancheng.net
C 语言中文网
http://c.biancheng.net
修改后，各类对象中类变量的值：
Python 教程
http://c.biancheng.net/python
Python 教程
http://c.biancheng.net/python
```

显然，通过类名修改类变量，会作用到所有的实例化对象（例如这里的 clang1 和 clang2）。

注意，通过类对象是无法修改类变量的。通过类对象对类变量赋值，其本质将不再是修改类变量的值，而是在给该对象定义新的实例变量（在讲实例变量时会进行详细介绍）。

值得一提的是，除了可以通过类名访问类变量之外，还可以动态地为类和对象添加类变量。例如，在 CLanguage 类的基础上，添加以下代码：

```
clang = CLanguage()
CLanguage.catalog = 13
print(clang.catalog)
```

运行结果为：

```
13
```

实例变量（实例属性）

实例变量指的是在任意类方法内部，以“self.变量名”的方式定义的变量，其特点是只作用于调用方法的对象。另外，实例变量只能通过对象名访问，无法通过类名访问。

举个例子：

```
class CLanguage :
    def __init__(self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    # 下面定义了一个 say 实例方法
    def say(self):
        self.catalog = 13
```

此 CLanguage 类中，name、add 以及 catalog 都是实例变量。其中，由于 __init__() 函数在创建类对象时会自动调用，而 say() 方法需要类对象手动调用。因此，CLanguage 类的类对象都会包含 name 和 add 实例变量，而只有调用了 say() 方法的类对象，才包含 catalog 实例变量。

例如，在上面代码的基础上，添加如下语句：

```
clang = CLanguage()
print(clang.name)
print(clang.add)
#由于 clang 对象未调用 say() 方法，因此其没有 catalog 变量，下面这行代码会报错
#print(clang.catalog)

clang2 = CLanguage()
print(clang2.name)
print(clang2.add)
#只有调用 say()，才会拥有 catalog 实例变量
clang2.say()
print(clang2.catalog)
```

运行结果为：

```
C 语言中文网
http://c.biancheng.net
C 语言中文网
http://c.biancheng.net
13
```

前面讲过，通过类对象可以访问类变量，但无法修改类变量的值。这是因为，通过类对象修改类变量的值，不是在给“类变量赋值”，而是定义新的实例变量。例如，在 CLanguage 类体外，添加如下程序：

```
clang = CLanguage()
#clang 访问类变量
print(clang.name)
print(clang.add)

clang.name = "Python 教程"
clang.add = "http://c.biancheng.net/python"
#clang 实例变量的值
print(clang.name)
print(clang.add)
```

```
#类变量的值
print(CLanguage.name)
print(CLanguage.add)
```

程序运行结果为：

```
C 语言中文网
http://c.biancheng.net
Python 教程
http://c.biancheng.net/python
C 语言中文网
http://c.biancheng.net
```

显然，通过类对象是无法修改类变量的值的，本质其实是给 clang 对象新添加 name 和 add 这 2 个实例变量。

类中，实例变量和类变量可以同名，但这种情况下使用类对象将无法调用类变量，它会首选实例变量，这也是不推荐“类变量使用对象名调用”的原因。

另外，和类变量不同，通过某个对象修改实例变量的值，不会影响类的其它实例化对象，更不会影响同名的类变量。例如：

```
class CLanguage :
    name = "xxx"  #类变量
    add = "http://"  #类变量
    def __init__(self):
        self.name = "C 语言中文网"   #实例变量
        self.add = "http://c.biancheng.net"   #实例变量
    # 下面定义了一个 say 实例方法
    def say(self):
        self.catalog = 13  #实例变量
clang = CLanguage()
#修改 clang 对象的实例变量
clang.name = "python 教程"
clang.add = "http://c.biancheng.net/python"
print(clang.name)
print(clang.add)

clang2 = CLanguage()
print(clang2.name)
print(clang2.add)
#输出类变量的值
```

```
print(CLanguage.name)
print(CLanguage.add)
```

程序运行结果为：

```
python 教程
http://c.biancheng.net/python
C 语言中文网
http://c.biancheng.net
xxx
http://
```

不仅如此，Python 只支持为特定的对象添加实例变量。例如，在之前代码的基础上，为 clang 对象添加 money 实例变量，实现代码为：

```
clang.money = 30
print(clang.money)
```

局部变量

除了实例变量，类方法中还可以定义局部变量。和前者不同，局部变量直接以“变量名=值”的方式进行定义，例如：


```
class CLanguage :
    # 下面定义了一个 say 实例方法
    def count(self,money):
        sale = 0.8*money
        print("优惠后的价格为：",sale)
clang = CLanguage()
clang.count(100)
```

通常情况下，定义局部变量是为了所在类方法功能的实现。需要注意的一点是，局部变量只能用于所在函数中，函数执行完成后，局部变量也会被销毁。

8.7 Python 实例方法、静态方法和类方法详解（包含区别和用法）

和类属性一样，类方法也可以进行更细致的划分，具体可分为类方法、实例方法和静态方法。

和类属性的分类不同，对于初学者来说，区分这 3 种类方法是非常简单的，即采用 @classmethod 修饰的方法为类方法；采用 @staticmethod 修饰的方法为静态方法；不用任何修改的方法为实例方法。

其中 @classmethod 和 @staticmethod 都是函数装饰器，后续章节会对其做详细介绍。
接下来就给大家详细的介绍这 3 种类方法。

Python 类实例方法

通常情况下，在类中定义的方法默认都是实例方法。前面章节中，我们已经定义了不只一个实例方法。不仅如此，类的构造方法理论上也属于实例方法，只不过它比较特殊。

比如，下面的类中就用到了实例方法：

```
1.  class CLanguage:
2.      #类构造方法，也属于实例方法
3.      def __init__(self):
4.          self.name = "C 语言中文网"
5.          self.add = "http://c.biancheng.net"
6.      # 下面定义了一个 say 实例方法
7.      def say(self):
8.          print("正在调用 say() 实例方法")
```

实例方法最大的特点就是，它最少也要包含一个 self 参数，用于绑定调用此方法的实例对象（Python 会自动完成绑定）。实例方法通常会用类对象直接调用，例如：

```
1.  clang = CLanguage()
2.  clang.say()
```

运行结果：

```
正在调用 say() 实例方法
```

当然，Python 也支持使用类名调用实例方法，但此方式需要手动给 self 参数传值。例如：

```
1.  #类名调用实例方法，需手动给 self 参数传值
2.  clang = CLanguage()
```

```
3.  CLanguage.say(clang)
```

运行结果为：

```
正在调用 say() 实例方法
```

有关使用类名直接调用实例方法的更多介绍，可阅读《Python 类调用实例方法》一节。

Python 类方法

Python 类方法和实例方法相似，它最少也要包含一个参数，只不过类方法中通常将其命名为 cls，Python 会自动将类本身绑定给 cls 参数（注意，绑定的不是类对象）。也就是说，我们在调用类方法时，无需显式为 cls 参数传参。

和 self 一样，cls 参数的命名也不是规定的（可以随意命名），只是 Python 程序员约定俗称的习惯而已。

和实例方法最大的不同在于，类方法需要使用`@classmethod`修饰符进行修饰，例如：

```
1.  class CLanguage:
2.      #类构造方法，也属于实例方法
3.      def __init__(self):
4.          self.name = "C 语言中文网"
5.          self.add = "http://c.biancheng.net"
6.      #下面定义了一个类方法
7.      @classmethod
8.      def info(cls):
9.          print("正在调用类方法",cls)
```

注意，如果没有 @classmethod，则 Python 解释器会将 fly() 方法认定为实例方法，而不是类方法。
类方法推荐使用类名直接调用，当然也可以使用实例对象来调用（不推荐）。例如，在上面 CLanguage 类的基础上，在该类外部添加如下代码：

```
1.  #使用类名直接调用类方法
2.  CLanguage.info()
3.  #使用类对象调用类方法
4.  clang = CLanguage()
5.  clang.info()
```

运行结果为：

```
正在调用类方法 <class '__main__.CLanguage'>
正在调用类方法 <class '__main__.CLanguage'>
```

Python 类静态方法

静态方法，其实就是我们学过的函数，和函数唯一的区别是，静态方法定义在类这个空间（类命名空间）中，而函数则定义在程序所在的空间（全局命名空间）中。

静态方法没有类似 self、cls 这样的特殊参数，因此 Python 解释器不会对它包含的参数做任何类或对象的绑定。也正因为如此，类的静态方法中无法调用任何类属性和类方法。

静态方法需要使用`@staticmethod`修饰，例如：

```
class CLanguage:
    @staticmethod
    def info(name,add):
        print(name,add)
```

静态方法的调用，既可以使用类名，也可以使用类对象，例如：

```
#使用类名直接调用静态方法
CLanguage.info("C 语言中文网","http://c.biancheng.net")
#使用类对象调用静态方法
clang = CLanguage()
clang.info("Python 教程","http://c.biancheng.net/python")
```

运行结果为：

```
C 语言中文网 http://c.biancheng.net
Python 教程 http://c.biancheng.net/python
```

在实际编程中，几乎不会用到类方法和静态方法，因为我们完全可以使用函数代替它们实现想要的功能，但在一些特殊的场景中（例如工厂模式中），使用类方法和静态方法也是很不错的选择。

8.8 Python 类调用实例方法

通过前面的学习，类方法大体分为 3 类，分别是类方法、实例方法和静态方法，其中实例方法用的是最多的。我们知道，实例方法的调用方式其实有 2 种，既可以采用类对象调用，也可以直接通过类名调用。

通常情况下，我们习惯使用类对象调用类中的实例方法。但如果想用类调用实例方法，不能像如下这样：

```
class CLanguage:
    def info(self):
        print("我正在学 Python")
#通过类名直接调用实例方法
CLanguage.info()
```

运行上面代码，程序会报出如下错误：

```
Traceback (most recent call last):
  File "D:\python3.6\demo.py", line 5, in <module>
    CLanguage.info()
TypeError: info() missing 1 required positional argument: 'self'
```

其中，最后一行报错信息提示我们，调用 info() 类方式时缺少给 self 参数传参。这意味着，和使用类对象调用实例方法不同，通过类名直接调用实例方法时，Python 并不会自动给 self 参数传值。

读者想想也应该明白，self 参数需要的是方法的实际调用者（是类对象），而这里只提供了类名，当然无法自动传值。

因此，如果想通过类名直接调用实例方法，就必须手动为 self 参数传值。例如修改上面的代码为：

```
class CLanguage:
    def info(self):
        print("我正在学 Python")
clang = CLanguage()
#通过类名直接调用实例方法
CLanguage.info(clang)
```

再次运行程序，结果为：

```
我正在学 Python
```

可以看到，通过手动将 clang 这个类对象传给了 self 参数，使得程序得以正确执行。实际上，这里调用实例方法的形式完全是等价于 clang.info()。

值得一提的是，上面的报错信息只是让我们手动为 self 参数传值，但并没有规定必须传一个该类的对象，其实完全可以任意传入一个参数，例如：

```
class CLanguage:
```

```
    def info(self):
        print(self,"正在学 Python")
#通过类名直接调用实例方法
CLanguage.info("zhangsan")
```

运行结果为：

```
zhangsan 正在学 Python
```

可以看到，"zhangsan" 这个字符串传给了 info() 方法的 self 参数。显然，无论是 info() 方法中使用 self 参数调用其它类方法，还是使用 self 参数定义新的实例变量，胡乱的给 self 参数传参都将会导致程序运行崩溃。

总的来说，Python 中允许使用类名直接调用实例方法，但必须手动为该方法的第一个 self 参数传递参数，这种调用方法的方式被称为“非绑定方法”。

用类的实例对象访问类成员的方式称为绑定方法，而用类名调用类成员的方式称为非绑定方法。

8.9 浅谈 Python 类命名空间

前面章节已经不只一次提到，Python 类体中的代码位于独立的命名空间（称为类命名空间）中。换句话说，所有用 class 关键字修饰的代码块，都可以看做是位于独立的命名空间中。

和类命名空间相对的是全局命名空间，即整个 Python 程序默认都位于全局命名空间中。而类体则独立位于类命名空间中。

我们一开始学习类时就已经提到，类其实是由多个类属性和类方法构成，而类属性其实就是定义在类这个独立空间中的变量，而类方法其实就是定义在类空间中的函数，和定义在全局命名空间中的变量和函数相比，并没有明显的不同。

举个例子：

```
#全局空间定义变量
name = "C 语言中文网"
add = "http://c.biancheng.net"
# 全局空间定义函数
def say ():
    print("我在学习 Python--全局")

class CLanguage:
    # 定义 CLanguage 空间的 say 函数
    def say():
        print("我在学习 Python--CLanguage 独立空间")
    # 定义 CLanguage 空间的 catalog 变量
    name = "C 语言中文网"
    add = "http://c.biancheng.net"
#调用全局的变量和函数
print(name,add)
say()

#调用类独立空间的变量和函数
print(CLanguage.name,CLanguage.add)
CLanguage.say()
```

运行结果为：

```
C 语言中文网 http://c.biancheng.net
我在学习 Python--全局
C 语言中文网 http://c.biancheng.net
我在学习 Python--CLanguage 独立空间
```

可以看到，相比位于全局命名空间的变量和函数，位于类命名空间中的变量和函数在使用时，只需要标注 CLanguage 前缀即可。

甚至，Python 还允许直接在类命名空间中编写可执行程序（例如输出语句、分支语句等），例如：

```
class CLanguage:
    #直接编写可执行代码
    print('正在执行 CLanguage 类空间中的代码')
    for i in range(5):
        print(i)
```

运行结果为：

```
正在执行 CLanguage 类空间中的代码
0
1
2
3
4
```

显然，上面这些位于类命名空间的可执行程序，和位于全局命令空间相比，并没有什么不同。

但需要注意的一点是，当使用类对象调用类方法时，在传参方面是和外界的函数有区别的，因为 Python 会自动会第一个参数绑定方法的调用者，而位于全局空间中的函数，则必须显式为第一个参数传递参数。

8.10 什么是描述符，Python 描述符详解

Python 中，通过使用描述符，可以让程序员在引用一个对象属性时自定义要完成的工作。

本质上看，描述符就是一个类，只不过它定义了另一个类中属性的访问方式。换句话说，一个类可以将属性管理全权委托给描述符类。

描述符是 Python 中复杂属性访问的基础，它在内部被用于实现 property、方法、类方法、静态方法和 super 类型。
描述符类基于以下 3 个特殊方法，换句话说，这 3 个方法组成了描述符协议：

- __set__(self, obj, type=None)：在设置属性时将调用这一方法（本节后续用 setter 表示）；
- __get__(self, obj, value)：在读取属性时将调用这一方法（本节后续用 getter 表示）；
- __delete__(self, obj)：对属性调用 del 时将调用这一方法。

其中，实现了 setter 和 getter 方法的描述符类被称为数据描述符；反之，如果只实现了 getter 方法，则称为非数据描述符。

实际上，在每次查找属性时，描述符协议中的方法都由类对象的特殊方法 __getattribute__() 调用（注意不要和 __getattr__() 弄混）。也就是说，每次使用类对象.属性（或者 getattr(类对象，属性值)）的调用方式时，都会隐式地调用 __getattribute__()，它会按照下列顺序查找该属性：

1. 验证该属性是否为类实例对象的数据描述符；
2. 如果不是，就查看该属性是否能在类实例对象的 __dict__ 中找到；
3. 最后，查看该属性是否为类实例对象的非数据描述符。

为了表达清楚，这里举个例子：

```
#描述符类
class revealAccess:
    def __init__(self, initval = None, name = 'var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print("Retrieving",self.name)
        return self.val

    def __set__(self, obj, val):
        print("updating",self.name)
        self.val = val
```

```
class myClass:
    x = revealAccess(10,'var "x"')
    y = 5
m = myClass()
print(m.x)
m.x = 20
print(m.x)
print(m.y)
```

运行结果为：

```
Retrieving var "x"
10
updating var "x"
Retrieving var "x"
20
5
```

从这个例子可以看到，如果一个类的某个属性有数据描述符，那么每次查找这个属性时，都会调用描述符的 __get__() 方法，并返回它的值；同样，每次在对该属性赋值时，也会调用 __set__() 方法。

注意，虽然上面例子中没有使用 __del__() 方法，但也很容易理解，当每次使用 del 类对象.属性（或者 delattr(类对象，属性)）语句时，都会调用该方法。

除了使用描述符类自定义类属性被调用时做的操作外，还可以使用 property() 函数或者 @property 装饰器，它们会在后续章节做详细介绍。

8.11 Python property()函数：定义属性

前面章节中，我们一直在用“类对象.属性”的方式访问类中定义的属性，其实这种做法是欠妥的，因为它破坏了类的封装原则。正常情况下，类包含的属性应该是隐藏的，只允许通过类提供的方法来间接实现对类属性的访问和操作。

因此，在不破坏类封装原则的基础上，为了能够有效操作类中的属性，类中应包含读（或写）类属性的多个 getter（或 setter）方法，这样就可以通过“类对象.方法(参数)”的方式操作属性，例如：

```
class CLanguage:
    #构造函数
    def __init__(self,name):
        self.name = name
    #设置 name 属性值的函数
    def setname(self,name):
        self.name = name
    #访问 nema 属性值的函数
    def getname(self):
        return self.name
    #删除 name 属性值的函数
    def delname(self):
        self.name="xxx"
clang = CLanguage("C 语言中文网")
#获取 name 属性值
print(clang.getname())
#设置 name 属性值
clang.setname("Python 教程")
print(clang.getname())
#删除 name 属性值
clang.delname()
print(clang.getname())
```

运行结果为：

```
C 语言中文网
Python 教程
xxx
```

可能有读者觉得，这种操作类属性的方式比较麻烦，更习惯使用“类对象.属性”这种方式。

庆幸的是，Python 中提供了 property() 函数，可以实现在不破坏类封装原则的前提下，让开发者依旧使用

“类对象.属性”的方式操作类中的属性。

property() 函数的基本使用格式如下：

```
属性名=property(fget=None, fset=None, fdel=None, doc=None)
```

其中，fget 参数用于指定获取该属性值的类方法，fset 参数用于指定设置该属性值的方法，fdel 参数用于指定删除该属性值的方法，最后的 doc 是一个文档字符串，用于说明此函数的作用。

注意，在使用 property() 函数时，以上 4 个参数可以仅指定第 1 个、或者前 2 个、或者前 3 个，当前也可以全部指定。也就是说，property() 函数中参数的指定并不是完全随意的。

例如，修改上面的程序，为 name 属性配置 property() 函数：

```
class CLanguage:
    #构造函数
    def __init__(self,n):
        self.__name = n
    #设置 name 属性值的函数
    def setname(self,n):
        self.__name = n
    #访问 nema 属性值的函数
    def getname(self):
        return self.__name
    #删除 name 属性值的函数
    def delname(self):
        self.__name="xxx"
    #为 name 属性配置 property() 函数
    name = property(getname, setname, delname, '指明出处')
#调取说明文档的 2 种方式
#print(CLanguage.name.__doc__)
help(CLanguage.name)
clang = CLanguage("C 语言中文网")
#调用 getname() 方法
print(clang.name)
#调用 setname() 方法
clang.name="Python 教程"
print(clang.name)
#调用 delname() 方法
del clang.name
```

```
print(clang.name)
```

运行结果为：

```
Help on property:

    指明出处

C 语言中文网
Python 教程
xxx
```

注意，在此程序中，由于 getname() 方法中需要返回 name 属性，如果使用 self.name 的话，其本身又被调用 getname()，这将会先入无限死循环。为了避免这种情况的出现，程序中的 name 属性必须设置为私有属性，即使用 __name（前面有 2 个下划线）。

有关类属性和类方法的属性设置（分为共有属性、保护属性、私有属性），后续章节会做详细介绍。
当然，property() 函数也可以少传入几个参数。以上面的程序为例，我们可以修改 property() 函数如下所示

```
name = property(getname, setname)
```

这意味着，name 是一个可读写的属性，但不能删除，因为 property() 函数中并没有为 name 配置用于函数该属性的方法。也就是说，即便 CLanguage 类中设计有 delname() 函数，这种情况下也不能用来删除 name 属性。

同理，还可以像如下这样使用 property() 函数：

```
name = property(getname)     # name 属性可读，不可写，也不能删除
name = property(getname, setname,delname)     #name 属性可读、可写、也可删除，就是没有说明文档
```

8.12 Python @property 装饰器详解

既要保护类的封装特性，又要让开发者可以使用“对象.属性”的方式操作操作类属性，除了使用 property() 函数，Python 还提供了 @property 装饰器。通过 @property 装饰器，可以直接通过方法名来访问方法，不需要在方法名后添加一对“ ( ) ”小括号。

@property 的语法格式如下：

```
@property
def 方法名(self)
    代码块
```

例如，定义一个矩形类，并定义用 @property 修饰的方法操作类中的 area 私有属性，代码如下：

```
class Rect:
    def __init__(self,area):
        self.__area = area
    @property
    def area(self):
        return self.__area
rect = Rect(30)
#直接通过方法名来访问 area 方法
print("矩形的面积是：",rect.area)
```

运行结果为：

```
矩形的面积为： 30
```

上面程序中，使用 @property 修饰了 area() 方法，这样就使得该方法变成了 area 属性的 getter 方法。需要注意的是，如果类中只包含该方法，那么 area 属性将是一个只读属性。

也就是说，在使用 Rect 类时，无法对 area 属性重新赋值，即运行如下代码会报错：

```
rect.area = 90
print("修改后的面积：",rect.area)
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 10, in <module>
    rect.area = 90
AttributeError: can't set attribute
```

而要想实现修改 area 属性的值，还需要为 area 属性添加 setter 方法，就需要用到 setter 装饰器，它的语法格式如下：

```
@方法名.setter
def 方法名(self, value):
    代码块
```

例如，为 Rect 类中的 area 方法添加 setter 方法，代码如下：

```
@area.setter
def area(self, value):
    self.__area = value
```

再次运行如下代码：

```
rect.area = 90
print("修改后的面积：",rect.area)
```

运行结果为：

```
修改后的面积: 90
```

这样，area 属性就有了 getter 和 setter 方法，该属性就变成了具有读写功能的属性。

除此之外，还可以使用 deleter 装饰器来删除指定属性，其语法格式为：

```
@方法名.deleter
def 方法名(self):
    代码块
```

例如，在 Rect 类中，给 area() 方法添加 deleter 方法，实现代码如下：

```
@area.deleter
def area(self):
    self.__area = 0
```

然后运行如下代码：

```
del rect.area
print("删除后的 area 值为：",rect.area)
```

运行结果为：

```
删除后的 area 值为： 0
```

8.13 Python 封装机制及实现方法

不光是 Python，大多数面向对象编程语言（诸如 C++、Java 等）都具备 3 个典型特征，即封装、继承和多态。其中，本节重点讲解 Python 类的封装特性，继承和多态会在后续章节给大家做详细讲解。

简单的理解封装（Encapsulation），即在设计类时，刻意地将一些属性和方法隐藏在类的内部，这样在使用此类时，将无法直接以“类对象.属性名”（或者“类对象.方法名(参数)”）的形式调用这些属性（或方法），而只能用未隐藏的类方法间接操作这些隐藏的属性和方法。

就好比使用电脑，我们只需要学会如何使用键盘和鼠标就可以了，不用关心内部是怎么实现的，因为那是生产和设计人员该操心的。

注意，封装绝不是将类中所有的方法都隐藏起来，一定要留一些像键盘、鼠标这样可供外界使用的类方法。
那么，类为什么要进行封装，这样做有什么好处呢？

首先，封装机制保证了类内部数据结构的完整性，因为使用类的用户无法直接看到类中的数据结构，只能使用类允许公开的数据，很好地避免了外部对内部数据的影响，提高了程序的可维护性。

除此之外，对一个类实现良好的封装，用户只能借助暴露出来的类方法来访问数据，我们只需要在这些暴露的方法中加入适当的控制逻辑，即可轻松实现用户对类中属性或方法的不合理操作。

并且，对类进行良好的封装，还可以提高代码的复用性。

Python 类如何进行封装？

和其它面向对象的编程语言（如 C++、Java）不同，Python 类中的变量和函数，不是公有的（类似 public 属性），就是私有的（类似 private），这 2 种属性的区别如下：

- public：公有属性的类变量和类函数，在类的外部、类内部以及子类（后续讲继承特性时会做详细介绍）中，都可以正常访问；
- private：私有属性的类变量和类函数，只能在本类内部使用，类的外部以及子类都无法使用。

但是，Python 并没有提供 public、private 这些修饰符。为了实现类的封装，Python 采取了下面的方法：

- 默认情况下，Python 类中的变量和方法都是公有（public）的，它们的名称前都没有下划线（_）；
- 如果类中的变量和函数，其名称以双下划线“__”开头，则该变量（函数）为私有变量（私有函数），其属性等同于 private。

除此之外，还可以定义以单下划线“_”开头的类属性或者类方法（例如 _name、_display(self)），这种类属性和类方法通常被视为私有属性和私有方法，虽然它们也能通过类对象正常访问，但这是一种约定俗称的用法，初学者一定要遵守。

注意，Python 类中还有以双下划线开头和结尾的类方法（例如类的构造函数__init__(self)），这些都是 Python 内部定义的，用于 Python 内部调用。我们自己定义类属性或者类方法时，不要使用这种格式。

例如，如下程序示范了 Python 的封装机制：

```
class CLanguage :
    def setname(self, name):
        if len(name) < 3:
            raise ValueError('名称长度必须大于 3!')
        self.__name = name

    def getname(self):
        return self.__name
    #为 name 配置 setter 和 getter 方法
    name = property(getname, setname)
    def setadd(self, add):
        if add.startswith("http://"):
            self.__add = add
        else:
            raise ValueError('地址必须以 http:// 开头')

    def getadd(self):
        return self.__add

    #为 add 配置 setter 和 getter 方法
    add = property(getadd, setadd)

    #定义个私有方法
    def __display(self):
        print(self.__name,self.__add)

clang = CLanguage()
clang.name = "C 语言中文网"
clang.add = "http://c.biancheng.net"
print(clang.name)
print(clang.add)
```

程序运行结果为：

上面程序中，CLanguage 将 name 和 add 属性都隐藏了起来，但同时也提供了可操作它们的“窗口”，也就是各自的 setter 和 getter 方法，这些方法都是公有（public）的。

不仅如此，以 add 属性的 setadd() 方法为例，通过在该方法内部添加控制逻辑，即通过调用 startswith() 方法，控制用户输入的地址必须以“http://”开头，否则程序将会执行 raise 语句抛出 ValueError 异常。

有关 raise 的具体用法，后续章节会做详细的讲解，这里可简单理解成，如果用户输入不规范，程序将会报错。通过此程序的运行逻辑不难看出，通过对 CLanguage 类进行良好的封装，使得用户仅能通过暴露的 setter() 和 getter() 方法操作 name 和 add 属性，而通过对 setname() 和 setadd() 方法进行适当的设计，可以避免用户对类中属性的不合理操作，从而提高了类的可维护性和安全性。

细心的读者可能还发现，CLanguage 类中还有一个 __display() 方法，由于该类方法为私有（private）方法，且该类没有提供操作该私有方法的“窗口”，因此我们无法在类的外部使用它。换句话说，如下调用 __display() 方法是不可行的：

```
#尝试调用私有的 display() 方法
clang.__display()
```

这会导致如下错误：

```
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 33, in <module>
    clang.__display()
AttributeError: 'CLanguage' object has no attribute '__display'
```

那么，类似 __display() 这样的类方法，就没有办法调用了吗？并非如此，读者在了解《Python 封装实现原理》之后，就可以轻松搞定它。

8.14 Python 封装底层实现原理详解（通俗易懂）

事实上，Python 封装特性的实现纯属“投机取巧”，之所以类对象无法直接调用以双下划线开头命名的类属性和类方法，是因为其底层实现时，Python 偷偷改变了它们的名称。

前面章节中，我们定义了一个 CLanguage 类，定义如下：

```
class CLanguage :
    def setname(self, name):
        if len(name) < 3:
            raise ValueError('名称长度必须大于 3！')
        self.__name = name

    def getname(self):
        return self.__name
    #为 name 配置 setter 和 getter 方法
    name = property(getname, setname)
    def setadd(self, add):
        if add.startswith("http://"):
            self.__add = add
        else:
            raise ValueError('地址必须以 http:// 开头')

    def getadd(self):
        return self.__add

    #为 add 配置 setter 和 getter 方法
    add = property(getadd, setadd)

    #定义个私有方法
    def __display(self):
        print(self.__name,self.__add)
```

注意，在这个类中，有一个 __display() 方法，由于其是私有方法，且该类没有提供任何调用该方法的“接口”，因此在目前看来，此方法根本无法在类外部调用。也就是说，如下调用 __display() 方法是不可行的：

```
clang = CLanguage()
#尝试调用私有的 display() 方法
```

```
3.  clang.__display()
```

这会导致如下错误：

```
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 33, in <module>
    clang.__display()
AttributeError: 'CLanguage' object has no attribute '__display'
```

那么，是不是类似 display() 这种的私有方法，真的没有方法调用吗？如果你深入了解 Python 封装机制的底层实现原理，就可以调用它。

事实上，对于以双下划线开头命名的类属性或类方法，Python 在底层实现时，将它们的名称都偷偷改成了 "_类名__属性（方法）名" 的格式。

就以 CLanguage 类中的 __display() 为例，Python 在底层将其方法名偷偷改成了“_CLanguage__display()”。例如，在 CLanguage 类的基础上，执行如下代码：

```
1.  clang = CLanguage()
2.  #调用 name 的 setname()方法
3.  clang.name = "C 语言中文网"
4.  #调用 add 的 setadd()方法
5.  clang.add = "http://c.biancheng.net"
6.  #直接调用隐藏的 display()方法
7.  clang._CLanguage__display()
```

输出结果为：

```
C 语言中文网 http://c.biancheng.net
```

不仅如此，那些原本我们认为是私有的类属性（例如 __name 和 __add），其底层的名称也改成了“_类名__属性名”的这种格式。例如：

```
1.  clang = CLanguage()
2.  clang.name = "C 语言中文网"
3.  clang.add = "http://c.biancheng.net"
4.  #直接调用 name 和 add 私有属性
5.  print(clang._CLanguage__name,clang._CLanguage__add)
```

运行结果为：

C 语言中文网 http://c.biancheng.net

甚至于，我们还可以通过这种方式修改 clang 对象的私有属性，例如：

```
clang._CLanguage__name = "Python 教程"
clang._CLanguage__add = "http://c.biancheng.net/python"
print(clang._CLanguage__name,clang._CLanguage__add)
```

输出结果为：

Python 教程 http://c.biancheng.net/python

总结

Python 类中所有的属性和方法，都是公有（public）属性，如果希望 Python 底层修改类属性或者类方法的名称，以此将它们隐藏起来，只需将它们的名称前添加双下划线（“__”）即可。

8.15 Python 继承机制及其使用

Python 类的封装、继承、多态 3 大特性，前面章节已经详细介绍了 Python 类的封装，本节继续讲解 Python 类的继承机制。

继承机制经常用于创建和现有类功能类似的新类，又或是新类只需要在现有类基础上添加一些成员（属性和方法），但又不想直接将现有类代码复制给新类。也就是说，通过使用继承这种机制，可以轻松实现类的重复使用。

举个例子，假设现有一个 Shape 类，该类的 draw() 方法可以在屏幕上画出指定的形状，现在需要创建一个 Form 类，要求此类不但可以在屏幕上画出指定的形状，还可以计算出所画形状的面积。要创建这样的类，笨方法是将 draw() 方法直接复制到新类中，并添加计算面积的方法。实现代码如下所示：

```
1.  class Shape:
2.      def draw(self,content):
3.          print("画",content)
4.  class Form:
5.      def draw(self,content):
6.          print("画",content)
7.      def area(self):
8.          #....
9.          print("此图形的面积为...")
```

当然还有更简单的方法，就是使用类的继承机制。实现方法为：让 From 类继承 Shape 类，这样当 From 类对象调用 draw() 方法时，Python 解释器会先去 From 中找以 draw 为名的方法，如果找不到，它还会自动去 Shape 类中找。如此，我们只需在 From 类中添加计算面积的方法即可，示例代码如下：

```
1.  class Shape:
2.      def draw(self,content):
3.          print("画",content)
4.  class Form(Shape):
5.      def area(self):
6.          #....
7.          print("此图形的面积为...")
```

上面代码中，class From(Shape) 就表示 From 继承 Shape。

Python 中，实现继承的类称为子类，被继承的类称为父类（也可称为基类、超类）。因此在上面这个样例中，From 是子类，Shape 是父类。

子类继承父类时，只需在定义子类时，将父类（可以是多个）放在子类之后的圆括号里即可。语法格式如下：

```
class 类名(父类 1, 父类 2, ...)：
    #类定义部分
```

注意，如果该类没有显式指定继承自哪个类，则默认继承 object 类（object 类是 Python 中所有类的父类，即要么是直接父类，要么是间接父类）。另外，Python 的继承是多继承机制（和 C++ 一样），即一个子类可以同时拥有多个直接父类。

注意，有读者可能还听说过“派生”这个词汇，它和继承是一个意思，只是观察角度不同而已。换句话话，继承是相对子类来说的，即子类继承自父类；而派生是相对于父类来说的，即父类派生出子类。

了解了继承机制的含义和语法之后，下面代码演示了继承机制的用法：

```
1.  class People:
2.      def say(self):
3.          print("我是一个人，名字是：",self.name)
4.  class Animal:
5.      def display(self):
6.          print("人也是高级动物")
7.  #同时继承 People 和 Animal 类
8.  #其同时拥有 name 属性、say() 和 display() 方法
9.  class Person(People, Animal):
10.     pass
11. zhangsan = Person()
12. zhangsan.name = "张三"
13. zhangsan.say()
14. zhangsan.display()
```

运行结果，结果为：

```
我是一个人，名字是： 张三
人也是高级动物
```

可以看到，虽然 Person 类为空类，但由于其继承自 People 和 Animal 这 2 个类，因此实际上 Person 并不空，它同时拥有这 2 个类所有的属性和方法。

没错，子类拥有父类所有的属性和方法，即便该属性或方法是私有（private）的。至于为什么，可阅读《Python 封装实现原理》一节。

关于 Python 的多继承

事实上，大部分面向对象的编程语言，都只支持单继承，即子类有且只能有一个父类。而 Python 却支持多继承（C++也支持多继承）。

和单继承相比，多继承容易让代码逻辑复杂、思路混乱，一直备受争议，中小型项目中较少使用，后来的 Java、C#、PHP 等干脆取消了多继承。

使用多继承经常需要面临的问题是，多个父类中包含同名的类方法。对于这种情况，Python 的处置措施是：根据子类继承多个父类时这些父类的前后次序决定，即排在前面父类中的类方法会覆盖排在后面父类中的同名类方法。

举个例子：

```
1.  class People:
2.      def __init__(self):
3.          self.name = People
4.      def say(self):
5.          print("People 类",self.name)
6.
7.  class Animal:
8.      def __init__(self):
9.          self.name = Animal
10.     def say(self):
11.         print("Animal 类",self.name)
12. #People 中的 name 属性和 say() 会遮蔽 Animal 类中的
13. class Person(People, Animal):
14.     pass
15.
16. zhangsan = Person()
17. zhangsan.name = "张三"
18. zhangsan.say()
```

程序运行结果为：

```
People 类 张三
```

可以看到，当 Person 同时继承 People 类和 Animal 类时，People 类在前，因此如果 People 和 Animal 拥有同名的类方法，实际调用的是 People 类中的。

虽然 Python 在语法上支持多继承，但逼不得已，建议大家不要使用多继承。

8.16 Python MRO 方法解析顺序详解

我们知道，Python 类是支持（多）继承的，一个类的方法和属性可能定义在当前类，也可能定义在基类。针对这种情况，当调用类方法或类属性时，就需要对当前类以及它的基类进行搜索，以确定方法或属性的位置，而搜索的顺序就称为方法解析顺序。

方法解析顺序（Method Resolution Order），简称 MRO。对于只支持单继承的编程语言来说，MRO 很简单，就是从当前类开始，逐个搜索它的父类；而对于 Python，它支持多继承，MRO 相对会复杂一些。

实际上，Python 发展至今，经历了以下 3 种 MRO 算法，分别是：

1. 从左往右，采用深度优先搜索（DFS）的算法，称为旧式类的 MRO；
2. 自 Python 2.2 版本开始，新式类在采用深度优先搜索算法的基础上，对其做了优化；
3. 自 Python 2.3 版本，对新式类采用了 C3 算法。由于 Python 3.x 仅支持新式类，所以该版本只使用 C3 算法。

有关旧式类和新式类的讲解，可阅读《Python super()使用注意事项》一文。
有读者可能会好奇，为什么 MRO 弃用了前两种算法，而选择最终的 C3 算法呢？原因很简单，前 2 种算法都存在一定的问题。

旧式类 MRO 算法

在使用旧式类的 MRO 算法时，以下面代码为例（程序一）：

```
class A:
    def method(self):
      print("CommonA")
class B(A):
    pass
class C(A):
    def method(self):
      print("CommonC")
class D(B, C):
    pass
print(D().method())
```

通过分析可以想到，此程序中的 4 个类是一个“菱形”继承的关系，当使用 D 类对象访问 method() 方法时，根据深度优先算法，搜索顺序为 D->B->A->C->A。

旧式类的 MRO 可通过使用 inspect 模块中的 getmro(类名) 函数直接获取。例如 inspect.getmro(D) 表示获取 D 类的 MRO。

因此，使用旧式类的 MRO 算法最先搜索得到的是基类 A 中的 method() 方法，即在 Python 2.x 版本中，此程序的运行结果为：

```
CommonA
```

但是，这个结果显然不是想要的，我们希望搜索到的是 C 类中的 method() 方法。

新式类 MRO 算法

为解决旧式类 MRO 算法存在的问题，Python 2.2 版本推出了新的计算新式类 MRO 的方法，它仍然采用从左至右的深度优先遍历，但是如果遍历中出现重复的类，只保留最后一个。

仍以上面程序为例，通过深度优先遍历，其搜索顺序为 `D->B->A->C->A`，由于此顺序中有 2 个 A，因此仅保留后一个，简化后得到最终的搜索顺序为 `D->B->C->A`。

新式类可以直接通过 类名.__mro__ 的方式获取类的 MRO，也可以通过 类名.mro() 的形式，旧式类是没有 __mro__ 属性和 mro() 方法的。
可以看到，这种 MRO 方式已经能够解决“菱形”继承的问题，但是可能会违反单调性原则。所谓单调性原则，是指在类存在多继承时，子类不能改变基类的 MRO 搜索顺序，否则会导致程序发生异常。

例如，分析如下程序（程序二）：

```
1.  class X(object):
2.    pass
3.  class Y(object):
4.    pass
5.  class A(X,Y):
6.    pass
7.  class B(Y,X):
8.    pass
9.  class C(A, B):
10.   pass
```

通过进行深度遍历，得到搜索顺序为 `C->A->X->object->Y->object->B->Y->object->X->object`，再进行简化（相同取后者），得到 `C->A->B->Y->X->object`。

下面来分析这样的搜索顺序是否合理，我们来看下各个类中的 MRO：

- 对于 A，其搜索顺序为 A->X->Y->object；
- 对于 B，其搜索顺序为 B->Y->X->object；
- 对于 C，其搜索顺序为 C->A->B->X->Y->object。

可以看到，B 和 C 中，X、Y 的搜索顺序是相反的，也就是说，当 B 被继承时，它本身的搜索顺序发生了改变，这违反了单调性原则。

MRO C3

为解决 Python 2.2 中 MRO 所存在的问题，Python 2.3 采用了 C3 方法来确定方法解析顺序。多数情况下，如果某人提到 Python 中的 MRO，指的都是 C3 算法。

在 Python 2.3 及后续版本中，运行程序一，得到如下结果：

```
CommonC
```

运行程序二，会产生如下异常：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 9, in <module>
    class C(A, B):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases X, Y
```

由此可见，C3 可以有效解决前面 2 种算法的问题。那么，C3 算法是怎样实现的呢？

以程序一为主，C3 把各个类的 MRO 记为如下等式：

- 类 A：L[A] = merge(A , object)
- 类 B：L[B] = [B] + merge(L[A] , [A])
- 类 C：L[C] = [C] + merge(L[A] , [A])
- 类 D：L[D] = [D] + merge(L[B] , L[C] , [B] , [C])

注意，以类 A 等式为例，其中 merge 包含的 A 称为 L[A] 的头，剩余元素（这里仅有一个 object）称为尾。这里的关键在于 merge，它的运算方式如下：

1. 检查第一个列表的头元素（如 L[A] 的头），记作 H。
2. 若 H 未出现在 merge 中其它列表的尾部，则将其输出，并将其从所有列表中删除，然后回到步骤 1；否则，取出下一个列表的头部记作 H，继续该步骤。

重复上述步骤，直至列表为空或者不能再找出可以输出的元素。如果是前一种情况，则算法结束；如果是后一种情况，Python 会抛出异常。

由此，可以计算出类 B 的 MRO，其计算过程为：

```
L[B] = [B] + merge(L[A],[A])
```

```
    = [B] + merge([A,object],[A])
    = [B,A] + merge([object])
    = [B,A,object]
```

同理，其他类的 MRO 也可以轻松计算得出。这里不再赘述，有兴趣的读者可自行推算。

8.17 Python 父类方法重写（入门必读）

前面讲过在 Python 中，子类继承了父类，那么子类就拥有了父类所有的类属性和类方法。通常情况下，子类会在此基础上，扩展一些新的类属性和类方法。

但凡事都有例外，我们可能会遇到这样一种情况，即子类从父类继承得来的类方法中，大部分是适合子类使用的，但有个别的类方法，并不能直接照搬父类的，如果不对这部分类方法进行修改，子类对象无法使用。针对这种情况，我们就需要在子类中重复父类的方法。

举个例子，鸟通常是有翅膀的，也会飞，因此我们可以像如下这样定义个和鸟相关的类：

```
class Bird:
    #鸟有翅膀
    def isWing(self):
        print("鸟有翅膀")
    #鸟会飞
    def fly(self):
        print("鸟会飞")
```

但是，对于鸵鸟来说，它虽然也属于鸟类，也有翅膀，但是它只会奔跑，并不会飞。针对这种情况，可以这样定义鸵鸟类：

```
class Ostrich(Bird):
    # 重写 Bird 类的 fly()方法
    def fly(self):
        print("鸵鸟不会飞")
```

可以看到，因为 Ostrich 继承自 Bird，因此 Ostrich 类拥有 Bird 类的 isWing() 和 fly() 方法。其中，isWing() 方法同样适合 Ostrich，但 fly() 明显不适合，因此我们在 Ostrich 类中对 fly() 方法进行重写。

重写，有时又称覆盖，是一个意思，指的是对类中已有方法的内部实现进行修改。

在上面 2 段代码的基础上，添加如下代码并运行：

```
class Bird:
    #鸟有翅膀
    def isWing(self):
        print("鸟有翅膀")
    #鸟会飞
    def fly(self):
        print("鸟会飞")
class Ostrich(Bird):
```

```
    # 重写 Bird 类的 fly()方法
    def fly(self):
        print("鸵鸟不会飞")

# 创建 Ostrich 对象
ostrich = Ostrich()
#调用 Ostrich 类中重写的 fly() 类方法
ostrich.fly()
```

运行结果为：

```
鸵鸟不会飞
```

显然，ostrich 调用的是重写之后的 fly() 类方法。

如何调用被重写的方法

事实上，如果我们在子类中重写了从父类继承来的类方法，那么当在类的外部通过子类对象调用该方法时，Python 总是会执行子类中重写的方法。

这就产生一个新的问题，即如果想调用父类中被重写的这个方法，该怎么办呢？

很简单，前面讲过，Python 中的类可以看做是一个独立空间，而类方法其实就是出于该空间中的一个函数。而如果想要全局空间中，调用类空间中的函数，只需要在调用该函数是备注类名即可。举个例子：

```
class Bird:
    #鸟有翅膀
    def isWing(self):
        print("鸟有翅膀")
    #鸟会飞
    def fly(self):
        print("鸟会飞")
class Ostrich(Bird):
    # 重写 Bird 类的 fly()方法
    def fly(self):
        print("鸵鸟不会飞")

# 创建 Ostrich 对象
```

```
ostrich = Ostrich()
#调用 Bird 类中的 fly() 方法
Bird.fly(ostrich)
```

程序运行结果为：

```
鸟会飞
```

此程序中，需要大家注意的一点是，使用类名调用其类方法，Python 不会为该方法的第一个 self 参数自定绑定值，因此采用这种调用方法，需要手动为 self 参数赋值。

通过类名调用实例方法的这种方式，又被称为**未绑定方法**。

8.18 如何使用 Python 继承机制（子类化内置类型）

我们知道，Python 中内置有一个 object 类，它是所有内置类型的共同祖先，也是所有没有显式指定父类的类（包括用户自定义的）的共同祖先。因此在实际编程过程中，如果想实现与某个内置类型具有类似行为的类时，最好的方法就是将这个内置类型子类化。

内置类型子类化，其实就是自定义一个新类，使其继承有类似行为的内置类，通过重定义这个新类实现指定的功能。

举个例子，如下所示创建了一个名为 newDict 的类，其中 newDictError 是自定义的异常类：

```
1.  class newDictError(ValueError):
2.    """如果向 newDict 添加重复值，则引发此异常"""
3.
4.  class newDict(dict):
5.    """不接受重复值的字典"""
6.    def __setitem__(self,key,value):
7.      if value in self.values():
8.        if ((key in self and self[key]!=value) or (key not in self)):
9.          raise newDictError("这个值已经存在，并对应不同的键")
10.     super().__setitem__(key,value)
11. demoDict = newDict()
12. demoDict['key']='value'
13. demoDict['other_key']='value2'
14. print(demoDict)
15. demoDict['other_key']='value'
16. print(demoDict)
```

运行结果为：

```
{'key': 'value', 'other_key': 'value2'}
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 15, in <module>
    demoDict['other_key']='value'
  File "C:\Users\mengma\Desktop\demo.py", line 9, in __setitem__
    raise newDictError("这个值已经存在，并对应不同的键")
newDictError: 这个值已经存在，并对应不同的键
```

可以看到，newDict 是 Python 中 dict 类型的子类，所以其大部分行为都和 dict 内置类相同，唯一不同之处在于，newDict 不允许字典中多个键对应相同的值。如果用户试图添加具有相同值的新元素，则会引发 newDictError 异常，并给出提示信息。

由于目前尚未学习如何处理异常，因此这里没有 newDictError 做任何处理，异常处理会在后续章节做详细讲解。

另外，如果查看现有代码你会发现，其实很多类都是对 Python 内置类的部分实现，它们作为子类的速度更快，代码更整洁。

比如，list 类型用来管理序列，如果一个类需要在内部处理序列，那么就可以对 list 进行子类化，示例代码如下：

```
class myList(list):
  def __init__(self,name):
    self.name = name

  def dir(self,nesting = 0):
    offset = " " * nesting
    print("%s%s/" % (offset,self.name))

    for element in self:
      if hasattr(element , 'dir'):
        element.dir(nesting + 1)
      else:
        print("%s %s" % (offset,element))

demoList = myList('C 语言中文网')
demoList.append('http://c.biancheng.net')
print(demoList.dir())
```

运行结果如下：

```
C 语言中文网/
http://c.biancheng.net
None
```

其实，除了 Python 中常用的基本内置类型，collections 模块中还额外提供了很多有用的容器，这些容器可以满足大部分情况。

8.19 Python super()函数：调用父类的构造方法

前面不止一次讲过，Python 中子类会继承父类所有的类属性和类方法。严格来说，类的构造方法其实就是实例方法，因此毫无疑问，父类的构造方法，子类同样会继承。

但我们知道，Python 是一门支持多继承的面向对象编程语言，如果子类继承的多个父类中包含同名的类实例方法，则子类对象在调用该方法时，会优先选择排在最前面的父类中的实例方法。显然，构造方法也是如此。

举个例子：

```
1.  class People:
2.      def __init__(self,name):
3.          self.name = name
4.      def say(self):
5.          print("我是人，名字为：",self.name)
6.
7.  class Animal:
8.      def __init__(self,food):
9.          self.food = food
10.     def display(self):
11.         print("我是动物,我吃",self.food)
12. #People 中的 name 属性和 say() 会遮蔽 Animal 类中的
13. class Person(People, Animal):
14.     pass
15.
16. per = Person("zhangsan")
17. per.say()
18. #per.display()
```

运行结果，结果为：

```
我是人，名字为： zhangsan
```

上面程序中，Person 类同时继承 People 和 Animal，其中 People 在前。这意味着，在创建 per 对象时，其将会调用从 People 继承来的构造函数。因此我们看到，上面程序在创建 per 对象的同时，还要给 name 属性进行赋值。

但如果去掉最后一行的注释，运行此行代码，Python 解释器会报如下错误：

```
Traceback (most recent call last):
  File "D:\python3.6\Demo.py", line 18, in <module>
```

```
    per.display()
  File "D:\python3.6\Demo.py", line 11, in display
    print("我是动物,我吃",self.food)
AttributeError: 'Person' object has no attribute 'food'
```

这是因为，从 Animal 类中继承的 display() 方法中，需要用到 food 属性的值，但由于 People 类的构造方法“遮蔽”了 Animal 类的构造方法，使得在创建 per 对象时，Animal 类的构造方法未得到执行，所以程序出错。

反过来也是如此，如果将第 13 行代码改为如下形式：

```
class Person(Animal, People)
```

则在创建 per 对象时，会给 food 属性传值。这意味着，per.display() 能顺序执行，但 per.say() 将会报错。

针对这种情况，正确的做法是定义 Person 类自己的构造方法（等同于重写第一个直接父类的构造方法）。但需要注意，如果在子类中定义构造方法，则必须在该方法中调用父类的构造方法。

在子类中的构造方法中，调用父类构造方法的方式有 2 种，分别是：

1. 类可以看做一个独立空间，在类的外部调用其中的实例方法，可以向调用普通函数那样，只不过需要额外备注类名（此方式又称为未绑定方法）；
2. 使用 super() 函数。但如果涉及多继承，该函数只能调用第一个直接父类的构造方法。

也就是说，涉及到多继承时，在子类构造函数中，调用第一个父类构造方法的方式有以上 2 种，而调用其它父类构造方法的方式只能使用未绑定方法。
值得一提的是，Python 2.x 中，super() 函数的使用语法格式如下：

```
super(Class, obj).__init__(self,...)
```

其中，Class 值得是子类的类名，obj 通常指的就是 self。

但在 Python 3.x 中，super() 函数有一种更简单的语法格式，推荐大家使用这种格式：

```
super().__init__(self,...)
```

在掌握 super() 函数用法的基础上，我们可以尝试修改上面的程序：

```
1.  class People:
2.      def __init__(self,name):
3.          self.name = name
4.      def say(self):
5.          print("我是人，名字为：",self.name)
6.  class Animal:
7.      def __init__(self,food):
8.          self.food = food
```

```
    def display(self):
        print("我是动物,我吃",self.food)
class Person(People, Animal):
    #自定义构造方法
    def __init__(self,name,food):
        #调用 People 类的构造方法
        super().__init__(name)
        #super(Person,self).__init__(name) #执行效果和上一行相同
        #People.__init__(self,name)#使用未绑定方法调用 People 类构造方法
        #调用其它父类的构造方法，需手动给 self 传值
        Animal.__init__(self,food)
per = Person("zhangsan","熟食")
per.say()
per.display()
```

运行结果为：

```
我是人，名字为： zhangsan
我是动物,我吃 熟食
```

可以看到，Person 类自定义的构造方法中，调用 People 类构造方法，可以使用 super() 函数，也可以使用未绑定方法。但是调用 Animal 类的构造方法，只能使用未绑定方法。

8.20 Python super()使用注意事项（包含新式类和旧式类的区别）

前面已经讲解了 super() 函数的用法，值得一提的是，Python 2 中 super() 函数的用法和 Python 3 大致相同，唯一的区别在于，Python 2 中不能使用零参数形式的格式，必须提供至少一个参数。

对于想要编写跨版本兼容代码的程序员来说，还要注意一件事，即 Python 2 中的 super() 函数只适用于新式类，在旧式类中不能使用 super()。

那么，什么是旧式类和新式类呢？在早期版本的 Python 中，所有类并没有一个共同的祖先 object，如果定义一个类，但没有显式指定其祖先，那么就被解释为旧式类，例如：

```
class oldStyle1:
  pass
class oldStyle2:
  pass
```

其中，oldStyle1 和 oldStyle2 都属于旧式类。

Python 2.x 版本中，为了向后兼容保留了旧式类。该版本中的新式类必须显式继承 object 或者其他新式类：

```
class newStyleClass(object):
  pass
class newStyleClass(newStyleClass):
  pass
```

显然，以上两个类都属于新式类。

而在 Python 3.x 版本中，不再保留旧式类的概念。因此，没有继承任何其他类的类都隐式地继承自 object。

可以说，在 Python 3.x 中，显式声明某个类继承自 object 似乎是冗余的。但如果考虑跨版本兼容，那么就必须将 object 作为所有基类的祖先，因为如果不这么做的话，这些类将被解释为旧式类，最终会导致难以诊断的问题。

8.21 Python super()使用注意事项

Python 中，由于基类不会在 __init__() 中被隐式地调用，需要程序员显式调用它们。这种情况下，当程序中包含多重继承的类层次结构时，使用 super 是非常危险的，往往会在类的初始化过程中出现问题。

混用 super 与显式类调用

分析如下程序，C 类使用了 __init__() 方法调用它的基类，会造成 B 类被调用了 2 次：

```
1.  class A:
2.      def __init__(self):
3.          print("A",end=" ")
4.          super().__init__()
5.  class B:
6.      def __init__(self):
7.          print("B",end=" ")
8.          super().__init__()
9.  class C(A,B):
10.     def __init__(self):
11.         print("C",end=" ")
12.         A.__init__(self)
13.         B.__init__(self)
14. print("MRO:",[x.__name__ for x in C.__mro__])
15. C()
```

运行结果为：

```
MRO: ['C', 'A', 'B', 'object']
C A B B
```

出现以上这种情况的原因在于，C 的实例调用 A.__init__(self)，使得 super(A,self).__init__() 调用了 B.__init__() 方法。换句话说，super 应该被用到整个类的层次结构中。

但是，有时这种层次结构的一部分位于第三方代码中，我们无法确定外部包的这些代码中是否使用 super()，因此，当需要对某个第三方类进行子类化时，最好查看其内部代码以及 MRO 中其他类的内部代码。

不同种类的参数

使用 super 的另一个问题是初始化过程中的参数传递。如果没有相同的签名，一个类怎么能调用其基类的 __init__() 代码呢？这会导致下列问题：

```
class commonBase:
    def __init__(self):
        print("commonBase")
        super().__init__()

class base1(commonBase):
    def __init__(self):
        print("base1")
        super().__init__()

class base2(commonBase):
    def __init__(self):
        print("base2")
        super().__init__()

class myClass(base1,base2):
    def __init__(self,arg):
        print("my base")
        super().__init__(arg)
myClass(10)
```

运行结果为：

```
my base
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 20, in <module>
    myClass(10)
  File "C:\Users\mengma\Desktop\demo.py", line 19, in __init__
    super().__init__(arg)
TypeError: __init__() takes 1 positional argument but 2 were given
```

一种解决方法是使用 *args 和 **kwargs 包装的参数和关键字参数，这样即使不使用它们，所有的构造函数也会传递所有参数，如下所示：

```
class commonBase:
    def __init__(self,*args,**kwargs):
        print("commonBase")
        super().__init__()

class base1(commonBase):
    def __init__(self,*args,**kwargs):
        print("base1")
        super().__init__(*args,**kwargs)

class base2(commonBase):
    def __init__(self,*args,**kwargs):
        print("base2")
        super().__init__(*args,**kwargs)

class myClass(base1,base2):
    def __init__(self,arg):
        print("my base")
        super().__init__(arg)
myClass(10)
```

运行结果为：

```
my base
base1
base2
commonBase
```

不过，这是一种很糟糕的解决方法，由于任何参数都可以传入，所有构造函数都可以接受任何类型的参数，这会导致代码变得脆弱。另一种解决方法是在 MyClass 中显式地使用特定类的 __init__() 调用，但这无疑会导致第一种错误。

总结

如果想要避免程序中出现以上的这些问题，这里给出几点建议：

- 尽可能避免使用多继承，可以使用一些设计模式来替代它；
- super 的使用必须一致，即在类的层次结构中，要么全部使用 super，要么全不用。混用 super 和传统调用是一种混乱的写法；

- 如果代码需要兼容 Python 2.x，在 Python 3.x 中应该显式地继承自 object。在 Python 2.x 中，没有指定任何祖先地类都被认定为旧式类。
- 调用父类时应提前查看类的层次结构，也就是使用类的 __mro__ 属性或者 mro() 方法查看有关类的 MRO。

8.22 Python __slots__：限制类实例动态添加属性和方法

通过学习《Python 类变量和实例变量》一节，了解了如何动态的为单个实例对象添加属性，甚至如果必要的话，还可以为所有的类实例对象统一添加属性（通过给类添加属性）。

那么，Python 是否也允许动态地为类或实例对象添加方法呢？答案是肯定的。我们知道，类方法又可细分为实例方法、静态方法和类方法，Python 语言允许为类动态地添加这 3 种方法；但对于实例对象，则只允许动态地添加实例方法，不能添加类方法和静态方法。

为单个实例对象添加方法，不会影响该类的其它实例对象；而如果为类动态地添加方法，则所有的实例对象都可以使用。

举个例子：

```
class CLanguage:
    pass
#下面定义了一个实例方法
def info(self):
    print("正在调用实例方法")
#下面定义了一个类方法
@classmethod
def info2(cls):
    print("正在调用类方法")
#下面定义个静态方法
@staticmethod
def info3():
    print("正在调用静态方法")

#类可以动态添加以上 3 种方法，会影响所有实例对象
CLanguage.info = info
CLanguage.info2 = info2
CLanguage.info3 = info3

clang = CLanguage()
#如今，clang 具有以上 3 种方法
clang.info()
clang.info2()
clang.info3()

```

```
#类实例对象只能动态添加实例方法，不会影响其它实例对象
clang1 = CLanguage()
clang1.info = info
#必须手动为 self 传值
clang1.info(clang1)
```

程序输出结果为：

```
正在调用实例方法
正在调用类方法
正在调用静态方法
正在调用实例方法
```

显然，动态给类或者实例对象添加属性或方法，是非常灵活的。但与此同时，如果胡乱地使用，也会给程序带来一定的隐患，即程序中已经定义好的类，如果不做任何限制，是可以做动态的修改的。

庆幸的是，Python 提供了 __slots__ 属性，通过它可以避免用户频繁的给实例对象动态地添加属性或方法。

再次声明，__slots__ 只能限制为实例对象动态添加属性和方法，而无法限制动态地为类添加属性和方法。
__slots__ 属性值其实就是一个元组，只有其中指定的元素，才可以作为动态添加的属性或者方法的名称。举个例子：

```
class CLanguage:
    __slots__ = ('name','add','info')
```

可以看到， CLanguage 类中指定了 __slots__ 属性，这意味着，该类的实例对象仅限于动态添加 name、add、info 这 3 个属性以及 name()、add() 和 info() 这 3 个方法。

注意，对于动态添加的方法，__slots__ 限制的是其方法名，并不限制参数的个数。
比如，在 CLanguage 类的基础上，添加如下代码并运行：

```
def info(self,name):
    print("正在调用实例方法",self.name)
clang = CLanguage()
clang.name = "C 语言中文网"
#为 clang 对象动态添加 info 实例方法
clang.info = info
clang.info(clang,"Python 教程")
```

程序运行结果为：

```
正在调用实例方法 C 语言中文网
```

还是在 CLanguage 类的基础上，添加如下代码并运行：

```
def info(self,name):
    print("正在调用实例方法",self.name)
clang = CLanguage()
clang.name = "C 语言中文网"
clang.say = info
clang.say(clang,"Python 教程")
```

运行程序，显示如下信息：

```
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 9, in <module>
    clang.say = info
AttributeError: 'CLanguage' object has no attribute 'say'
```

显然，根据 __slots__ 属性的设置，CLanguage 类的实例对象是不能动态添加以 say 为名称的方法的。

另外本节前面提到，__slots__ 属性限制的对象是类的实例对象，而不是类，因此下面的代码是合法的：

```
def info(self):
    print("正在调用实例方法")

CLanguage.say = info
clang = CLanguage()
clang.say()
```

程序运行结果为：

```
正在调用实例方法
```

当然，还可以为类动态添加类方法和静态方法，这里不再给出具体实例，读者可自行编写代码尝试。

此外，__slots__ 属性对由该类派生出来的子类，也是不起作用的。例如如下代码：

```
class CLanguage:
    __slots__ = ('name','add','info')
#Clanguage 的空子类
class CLangs(CLanguage):
    pass
#定义的实例方法
```

```
7.  def info(self):
8.      print("正在调用实例方法")
9.  clang = CLangs()
10. #为子类对象动态添加 say() 方法
11. clang.say = info
12. clang.say(clang)
```

运行结果为：

```
正在调用实例方法
```

显然，__slots__ 属性只对当前所在的类起限制作用。

因此，如果子类也要限制外界为其实例对象动态地添加属性和方法，必须在子类中设置 __slots__ 属性。

注意，如果为子类也设置有 __slots__ 属性，那么子类实例对象允许动态添加的属性和方法，是子类中 __slots__ 属性和父类 __slots__ 属性的和。

8.23 Python type()函数：动态创建类

我们知道，type() 函数属于 Python 内置函数，通常用来查看某个变量的具体类型。其实，type() 函数还有一个更高级的用法，即创建一个自定义类型（也就是创建一个类）。

type() 函数的语法格式有 2 种，分别如下：

```
type(obj)
type(name, bases, dict)
```

以上这 2 种语法格式，各参数的含义及功能分别是：

- 第一种语法格式用来查看某个变量（类对象）的具体类型，obj 表示某个变量或者类对象。
- 第二种语法格式用来创建类，其中 name 表示类的名称；bases 表示一个元组，其中存储的是该类的父类；dict 表示一个字典，用于表示类内定义的属性或者方法。

对于使用 type() 函数查看某个变量或类对象的类型，由于很简单，这里不再做过多解释，直接给出一个样例：

```
#查看 3.4 的类型
print(type(3.4))
#查看类对象的类型
class CLanguage:
    pass
clangs = CLanguage()
print(type(clangs))
```

输出结果为：

```
<class 'float'>
<class '__main__.CLanguage'>
```

这里重点介绍 type() 函数的另一种用法，即创建一个新类，先来分析一个样例：

```
#定义一个实例方法
def say(self):
    print("我要学 Python！")
#使用 type() 函数创建类
CLanguage = type("CLanguage",(object,),dict(say = say, name = "C 语言中文网"))
#创建一个 CLanguage 实例对象
clangs = CLanguage()
#调用 say() 方法和 name 属性
```

```
clangs.say()
print(clangs.name)
```

注意，Python 元组语法规定，当 (object,) 元组中只有一个元素时，最后的逗号 (,) 不能省略。
可以看到，此程序中通过 type() 创建了类，其类名为 CLanguage，继承自 objects 类，且该类中还包含一个 say() 方法和一个 name 属性。

有读者可能会问，如何判断 dict 字典中添加的是方法还是属性？很简单，如果该键值对中，值为普通变量（如 "C 语言中文网"），则表示为类添加了一个类属性；反之，如果值为外部定义的函数（如 say() ），则表示为类添加了一个实例方法。
运行上面的程序，其输出结果为：

```
我要学 Python！
C 语言中文网
```

可以看到，使用 type() 函数创建的类，和直接使用 class 定义的类并无差别。事实上，我们在使用 class 定义类时，Python 解释器底层依然是用 type() 来创建这个类。

8.24 Python MetaClass 元类详解

MetaClass 元类，本质也是一个类，但和普通类的用法不同，它可以对类内部的定义（包括类属性和类方法）进行动态的修改。可以这么说，使用元类的主要目的就是为了实现在创建类时，能够动态地改变类中定义的属性或者方法。

不要从字面上去理解元类的含义，事实上 MetaClass 中的 Meta 这个词根，起源于希腊语词汇 meta，包含“超越”和“改变”的意思。

举个例子，根据实际场景的需要，我们要为多个类添加一个 name 属性和一个 say() 方法。显然有多种方法可以实现，但其中一种方法就是使用 MetaClass 元类。

如果在创建类时，想用 MetaClass 元类动态地修改内部的属性或者方法，则类的创建过程将变得复杂：先创建 MetaClass 元类，然后用元类去创建类，最后使用该类的实例化对象实现功能。

和前面章节创建的类不同，如果想把一个类设计成 MetaClass 元类，其必须符合以下条件：

1. 必须显式继承自 type 类；
2. 类中需要定义并实现 __new__() 方法，该方法一定要返回该类的一个实例对象，因为在使用元类创建类时，该 __new__() 方法会自动被执行，用来修改新建的类。

讲了这么多，读者可能对 MetaClass 元类的功能还是比较懵懂。没关系，我们先尝试定义一个 MetaClass 元类：

```
#定义一个元类
class FirstMetaClass(type):
    # cls 代表动态修改的类
    # name 代表动态修改的类名
    # bases 代表被动态修改的类的所有父类
    # attr 代表被动态修改的类的所有属性、方法组成的字典
    def __new__(cls, name, bases, attrs):
        # 动态为该类添加一个 name 属性
        attrs['name'] = "C 语言中文网"
        attrs['say'] = lambda self: print("调用 say() 实例方法")
        return super().__new__(cls,name,bases,attrs)
```

此程序中，首先可以断定 FirstMetaClass 是一个类。其次，由于该类继承自 type 类，并且内部实现了 __new__() 方法，因此可以断定 FirstMetaCLass 是一个元类。

有关 __new__() 的具体用法，可阅读《Python __new__()方法》一节。

可以看到，在这个元类的 __new__() 方法中，手动添加了一个 name 属性和 say() 方法。这意味着，通过 FirstMetaClass 元类创建的类，会额外添加 name 属性和 say() 方法。通过如下代码，可以验证这个结论：

```
#定义类时，指定元类
class CLanguage(object,metaclass=FirstMetaClass):
    pass

clangs = CLanguage()
print(clangs.name)
clangs.say()
```

可以看到，在创建类时，通过在标注父类的同时指定元类（格式为 `metaclass=元类名`），则当 Python 解释器在创建这该类时，FirstMetaClass 元类中的 __new__ 方法就会被调用，从而实现动态修改类属性或者类方法的目的。

运行上面的程序，输出结果为：

```
C 语言中文网
调用 say() 实例方法
```

显然，FirstMetaClass 元类的 __new__() 方法动态地为 Clanguage 类添加了 name 属性和 say() 方法，因此，即便该类在定义时是空类，它也依然有 name 属性和 say() 方法。

对于 MetaClass 元类，它多用于创建 API，因此我们几乎不会使用到它。

8.25 Python MetaClass 元类实现的底层原理

要理解 MetaClass 的底层原理，首先要深入理解 Python 类型模型。本节将从以下 2 点对 Python 类型模型做详细的介绍。

1) 所有的 Python 的用户定义类，都是 type 这个类的实例

事实上，类本身不过是一个名为 type 类的实例，可以通过如下代码进行验证：

```
class MyClass:
  pass

instance = MyClass()
print(type(instance))
print(type(MyClass))
```

输出结果为：

```
<class '__main__.MyClass'>
<class 'type'>
```

可以看到，instance 是 MyClass 的实例，而 MyClass 是 type 的实例。

2) 用户自定义类，只不过是 type 类的 __call__ 运算符重载。

当定义完成一个类时，真正发生的情况是 Python 会调用 type 类的 __call__ 运算符。

简单来说，当定义一个类时，例如下面语句：

```
class MyClass:
  data = 1
```

Python 底层执行的是下面这段代码：

```
class = type(classname, superclasses, attributedict)
```

其中等号右边的 type(classname, superclasses, attributedict) 就是 type 的 __call__ 运算符重载，它会进一步调用下面这 2 个函数：

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(class, classname, superclasses, attributedict)
```

以上整个过程，可以通过如下代码进行论证：

```
class MyClass:
    data = 1

instance = MyClass()
print(MyClass,instance)
print(instance.data)
MyClass = type('MyClass', (), {'data': 1})
instance = MyClass()

print(MyClass,instance)
print(instance.data)
```

运行结果为：

```
<class '__main__.MyClass'> <__main__.MyClass object at 0x000001CB469F7400>
1
<class '__main__.MyClass'> <__main__.MyClass object at 0x000001CB46A50828>
1
```

由此可见，正常的 MyClass 定义，和手工调用 type 运算符的结果是完全一样的。

总之，正是 Python 的类创建机制，给了 metaclass 大展身手的机会，即一旦把一个类型 MyClass 设置成元类 MyMeta，那么它就不再由原生的 type 创建，而是会调用 MyMeta 的 __call__ 运算符重载：

```
class = type(classname, superclasses, attributedict)
# 变为了
class = MyMeta(classname, superclasses, attributedict)
```

使用 metaclass 的风险

正如上面所看到的那样，metaclass 这样 “逆天” 的存在，会"扭曲变形"正常的 Python 类型模型，所以，如果使用不慎，对于整个代码库造成的风险是不可估量的。

换句话说，metaclass 仅仅是给小部分 Python 开发者，在开发框架层面的 Python 库时使用的。而在应用层，metaclass 往往不是很好的选择。

建议初学者不要轻易尝试使用 mateclass。

8.26 什么是多态，Python 多态及用法详解

在面向对象程序设计中，除了封装和继承特性外，多态也是一个非常重要的特性，本节就带领大家详细了解什么是多态。

我们都知道，Python 是弱类型语言，其最明显的特征是在使用变量时，无需为其指定具体的数据类型。这会导致一种情况，即同一变量可能会被先后赋值不同的类对象，例如：

```
class CLanguage:
    def say(self):
        print("赋值的是 CLanguage 类的实例对象")
class CPython:
    def say(self):
        print("赋值的是 CPython 类的实例对象")
a = CLanguage()
a.say()

a = CPython()
a.say()
```

运行结果为：

```
赋值的是 CLanguage 类的实例对象
赋值的是 CPython 类的实例对象
```

可以看到，a 可以被先后赋值为 CLanguage 类和 CPython 类的对象，但这并不是多态。类的多态特性，还要满足以下 2 个前提条件：

1. 继承：多态一定是发生在子类和父类之间；
2. 重写：子类重写了父类的方法。

下面程序是对上面代码的改写：

```
class CLanguage:
    def say(self):
        print("调用的是 Clanguage 类的 say 方法")
class CPython(CLanguage):
    def say(self):
        print("调用的是 CPython 类的 say 方法")
class CLinux(CLanguage):
    def say(self):
```

```
        print("调用的是 CLinux 类的 say 方法")
a = CLanguage()
a.say()

a = CPython()
a.say()

a = CLinux()
a.say()
```

程序执行结果为：

```
调用的是 Clanguage 类的 say 方法
调用的是 CPython 类的 say 方法
调用的是 CLinux 类的 say 方法
```

可以看到，CPython 和 CLinux 都继承自 CLanguage 类，且各自都重写了父类的 say() 方法。从运行结果可以看出，同一变量 a 在执行同一个 say() 方法时，由于 a 实际表示不同的类实例对象，因此 a.say() 调用的并不是同一个类中的 say() 方法，这就是多态。

但是，仅仅学到这里，读者还无法领略 Python 类使用多态特性的精髓。其实，Python 在多态的基础上，衍生出了一种更灵活的编程机制。

继续对上面的程序进行改写：

```
class WhoSay:
    def say(self,who):
        who.say()
class CLanguage:
    def say(self):
        print("调用的是 Clanguage 类的 say 方法")

class CPython(CLanguage):
    def say(self):
        print("调用的是 CPython 类的 say 方法")

class CLinux(CLanguage):
    def say(self):
        print("调用的是 CLinux 类的 say 方法")
```

```
a = WhoSay()
#调用 CLanguage 类的 say() 方法
a.say(CLanguage())
#调用 CPython 类的 say() 方法
a.say(CPython())
#调用 CLinux 类的 say() 方法
a.say(CLinux())
```

程序执行结果为：

```
调用的是 Clanguage 类的 say 方法
调用的是 CPython 类的 say 方法
调用的是 CLinux 类的 say 方法
```

此程序中，通过给 WhoSay 类中的 say() 函数添加一个 who 参数，其内部利用传入的 who 调用 say() 方法。这意味着，当调用 WhoSay 类中的 say() 方法时，我们传给 who 参数的是哪个类的实例对象，它就会调用那个类中的 say() 方法。

在其它教程中，Python 这种由多态衍生出的更灵活的编程机制，又称为“鸭子模型”或“鸭子类型”。

8.27 Python 枚举类定义和使用（详解版）

一些具有特殊含义的类，其实例化对象的个数往往是固定的，比如用一个类表示月份，则该类的实例对象最多有 12 个；再比如用一个类表示季节，则该类的实例化对象最多有 4 个。

针对这种特殊的类，Python 3.4 中新增加了 Enum 枚举类。也就是说，对于这些实例化对象个数固定的类，可以用枚举类来定义。

例如，下面程序演示了如何定义一个枚举类：

```
from enum import Enum
class Color(Enum):
    # 为序列值指定 value 值
    red = 1
    green = 2
    blue = 3
```

如果想将一个类定义为枚举类，只需要令其继承自 enum 模块中的 Enum 类即可。例如在上面程序中，Color 类继承自 Enum 类，则证明这是一个枚举类。

在 Color 枚举类中，red、green、blue 都是该类的成员（可以理解为是类变量）。注意，枚举类的每个成员都由 2 部分组成，分别为 name 和 value，其中 name 属性值为该枚举值的变量名（如 red），value 代表该枚举值的序号（序号通常从 1 开始）。

和普通类的用法不同，枚举类不能用来实例化对象，但这并不妨碍我们访问枚举类中的成员。访问枚举类成员的方式有多种，例如以 Color 枚举类为例，在其基础上添加如下代码：

```
#调用枚举成员的 3 种方式
print(Color.red)
print(Color['red'])
print(Color(1))
#调取枚举成员中的 value 和 name
print(Color.red.value)
print(Color.red.name)
#遍历枚举类中所有成员的 2 种方式
for color in Color:
    print(color)
```

程序输出结果为：

```
Color.red
Color.red
```

```
Color.red
1
red
Color.red
Color.green
Color.blue
```

枚举类成员之间不能比较大小，但可以用 == 或者 is 进行比较是否相等，例如：

```
print(Color.red == Color.green)
print(Color.red.name is Color.green.name)
```

输出结果为：

```
Flase
Flase
```

需要注意的是，枚举类中各个成员的值，不能在类的外部做任何修改，也就是说，下面语法的做法是错误的：

```
Color.red = 4
```

除此之外，该枚举类还提供了一个 __members__ 属性，该属性是一个包含枚举类中所有成员的字典，通过遍历该属性，也可以访问枚举类中的各个成员。例如：

```
for name,member in Color.__members__.items():
    print(name,"->",member)
```

输出结果为：

```
red -> Color.red
green -> Color.green
blue -> Color.blue
```

值得一提的是，Python 枚举类中各个成员必须保证 name 互不相同，但 value 可以相同，举个例子：

```
from enum import Enum

class Color(Enum):
    # 为序列值指定 value 值
    red = 1
    green = 1
```

```
    blue = 3
print(Color['green'])
```

输出结果为：

```
Color.red
```

可以看到，Color 枚举类中 red 和 green 具有相同的值（都是 1），Python 允许这种情况的发生，它会将 green 当做是 red 的别名，因此当访问 green 成员时，最终输出的是 red。

在实际编程过程中，如果想避免发生这种情况，可以借助 @unique 装饰器，这样当枚举类中出现相同值的成员时，程序会报 ValueError 错误。例如：

```
#引入 unique
from enum import Enum,unique
#添加 unique 装饰器
@unique
class Color(Enum):
    # 为序列值指定 value 值
    red = 1
    green = 1
    blue = 3
print(Color['green'])
```

运行程序会报错：

```
Traceback (most recent call last):
  File "D:\python3.6\demo.py", line 3, in <module>
    class Color(Enum):
  File "D:\python3.6\lib\enum.py", line 834, in unique
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'Color'>: green -> red
```

除了通过继承 Enum 类的方法创建枚举类，还可以使用 Enum() 函数创建枚举类。例如：

```
from enum import Enum
#创建一个枚举类
Color = Enum("Color",('red','green','blue'))

#调用枚举成员的 3 种方式
print(Color.red)
```

```
print(Color['red'])
print(Color(1))
#调取枚举成员中的 value 和 name
print(Color.red.value)
print(Color.red.name)
#遍历枚举类中所有成员的 2 种方式
for color in Color:
    print(color)
```

Enum() 函数可接受 2 个参数，第一个用于指定枚举类的类名，第二个参数用于指定枚举类中的多个成员。

如上所示，仅通过一行代码，即创建了一个和前面的 Color 类相同的枚举类。运行程序，其输出结果为：

```
Color.red
Color.red
Color.red
1
red
Color.red
Color.green
Color.blue
```

8.28 利用面向对象思想实现搜索引擎

要想实现一个搜索引擎，首先要了解什么是搜索引擎。简单地理解，搜索引擎是一个系统，它可以帮助用户去互联网上搜集与其检索内容相关的信息。

通常，一个搜索引擎由搜索器、索引器、检索器以及用户接口组成，其中各个部分的含义如下：

- 搜索器：其实就是我们常说的爬虫、它能够从互联网中搜集大量的信息，并将之传递给索引器；
- 索引器：理解搜索器搜索到的信息，并从中抽取出索引项，存储到内部的数据库中，等待检索；
- 检索器：根据用户查询的内容，在已经建立好的索引库中快速检索出与之相关的信息，并做相关度评价，以此进行排序。
- 用户接口：其作用就是提供给用户输入查询内容的窗口（例如百度、谷歌的搜索框），并将检索好的内容反馈给用户。

由于爬虫知识不是本节学习的重点，这里不再做深入介绍，我们假设搜索样本就存在于本地磁盘中。为了方便，这里只提供五个用于检索的文件，各文件存放的内容分别如下：

```
# 1.txt
C 语言中文网
# 2.txt
http://c.biancheng.net
# 3.txt
「C 语言中文网」是一个在线学习编程的网站，我们发布了多套文字教程，它们都通俗易懂，深入浅出。
# 4.txt
C 语言中文网成立于 2012 年初，目前已经运营了将近 7 年，我们致力于分享精品教程，帮助对编程感兴趣的读者。
# 5.txt
坚持做好一件事情，做到极致，让自己感动，让用户心动，这就是足以传世的作品！
```

下面，根据以上知识，我们先实现一个最基本的搜索引擎：

```
1.  class SearchEngineBase:
2.      def __init__(self):
3.          pass
4.      #搜索器
5.      def add_corpus(self, file_path):
6.          with open(file_path, 'rb') as fin:
7.              text = fin.read().decode('utf-8')
8.          self.process_corpus(file_path, text)
9.      #索引器
10.     def process_corpus(self, id, text):
11.         raise Exception('process_corpus not implemented.')
12.     #检索器
```

```
    def search(self, query):
        raise Exception('search not implemented.')
#用户接口
def main(search_engine):
    for file_path in ['1.txt', '2.txt', '3.txt', '4.txt', '5.txt']:
        search_engine.add_corpus(file_path)
    while True:
        query = input()
        results = search_engine.search(query)
        print('found {} result(s):'.format(len(results)))
        for result in results:
            print(result)
```

以上代码仅是建立了搜索引擎的一个基本框架，它可以作为基类被其他类继承，那么继承自此类的类将分别代表不同的搜索引擎，它们应该各自实现基类中的 process_corpus() 和 search() 方法。

整个代码的运行过程是这样的，首先将各个检索文件中包含的内容连同该文件所在的路径一起传递给索引器，索引器会以该文件的路径建立索引，等待用户检索。

在 SearchEngineBase 类的基础上，下面实现了一个基本可以工作的搜索引擎：

```
#继承 SearchEngineBase 类，并重写了 process_corpus 和 search 方法
class SimpleEngine(SearchEngineBase):
    def __init__(self):
        super(SimpleEngine, self).__init__()
        #建立索引时使用
        self.__id_to_texts = {}

    def process_corpus(self, id, text):
        #以文件路径为键，文件内容为值，形成键值对，存储在字典中，由此建立索引
        self.__id_to_texts[id] = text

    def search(self, query):
        results = []
        #依次检索字典中的键值对，如果文件内容中包含用户要搜索的信息，则将此文件的文件路径存储在 results 列表中
        for id, text in self.__id_to_texts.items():
            if query in text:
```

```
                results.append(id)
        return results

search_engine = SimpleEngine()
main(search_engine)
```

运行结果为：

```
C 语言中文网
found 3 result(s):
1.txt
3.txt
4.txt
```

可以看到，用户搜索与“C 语言中文网”有关的内容，最终检索到了 1.txt、3.txt 和 4.txt 文件中包含与之相关的内容。由此，只需要短短十来行代码就可以实现一个基础的搜索引擎。

第 9 章：Python 类特殊成员（属性和方法）

9.1 Python __new__()方法详解

__new__() 是一种负责创建类实例的静态方法，它无需使用 staticmethod 装饰器修饰，且该方法会优先 __init__() 初始化方法被调用。

一般情况下，覆写 __new__() 的实现将会使用合适的参数调用其超类的 super().__new__()，并在返回之前修改实例。例如：

```
class demoClass:
    instances_created = 0
    def __new__(cls,*args,**kwargs):
        print("__new__():",cls,args,kwargs)
        instance = super().__new__(cls)
        instance.number = cls.instances_created
        cls.instances_created += 1
        return instance
    def __init__(self,attribute):
        print("__init__():",self,attribute)
        self.attribute = attribute
test1 = demoClass("abc")
test2 = demoClass("xyz")
print(test1.number,test1.instances_created)
print(test2.number,test2.instances_created)
```

输出结果为：

```
__new__(): <class '__main__.demoClass'> ('abc',) {}
__init__(): <__main__.demoClass object at 0x0000026FC0DF8080> abc
__new__(): <class '__main__.demoClass'> ('xyz',) {}
__init__(): <__main__.demoClass object at 0x0000026FC0DED358> xyz
0 2
1 2
```

__new__() 通常会返回该类的一个实例，但有时也可能会返回其他类的实例，如果发生了这种情况，则会跳过对 __init__() 方法的调用。而在某些情况下（比如需要修改不可变类实例（Python 的某些内置类型）的创建行为），利用这一点会事半功倍。比如：

```
class nonZero(int):
    def __new__(cls,value):
        return super().__new__(cls,value) if value != 0 else None
    def __init__(self,skipped_value):
        #此例中会跳过此方法
        print("__init__()")
```

```
        super().__init__()
print(type(nonZero(-12)))
print(type(nonZero(0)))
```

运行结果为：

```
__init__()
<class '__main__.nonZero'>
<class 'NoneType'>
```

那么，什么情况下使用 __new__() 呢？答案很简单，在 __init__() 不够用的时候。

例如，前面例子中对 Python 不可变的内置类型（如 int、str、float 等）进行了子类化，这是因为一旦创建了这样不可变的对象实例，就无法在 __init__() 方法中对其进行修改。

有些读者可能会认为，__new__() 对执行重要的对象初始化很有用，如果用户忘记使用 super()，可能会漏掉这一初始化。虽然这听上去很合理，但有一个主要的缺点，即如果使用这样的方法，那么即便初始化过程已经是预期的行为，程序员明确跳过初始化步骤也会变得更加困难。不仅如此，它还破坏了“__init__() 中执行所有初始化工作”的潜规则。

注意，由于 __new__() 不限于返回同一个类的实例，所以很容易被滥用，不负责任地使用这种方法可能会对代码有害，所以要谨慎使用。一般来说，对于特定问题，最好搜索其他可用的解决方案，最好不要影响对象的创建过程，使其违背程序员的预期。比如说，前面提到的覆写不可变类型初始化的例子，完全可以用工厂方法（一种设计模式）来替代。

Python 中大量使用 __new__() 方法且合理的，就是 MetaClass 元类。有关元类的介绍，可阅读《Python MetaClass 元类》一节。

9.2 Python __repr__()方法：显示属性

前面章节中，我们经常会直接输出类的实例化对象，例如：

```
class CLanguage:
    pass
clangs = CLanguage()
print(clangs)
```

程序运行结果为：

```
<__main__.CLanguage object at 0x000001A7275221D0>
```

通常情况下，直接输出某个实例化对象，本意往往是想了解该对象的基本信息，例如该对象有哪些属性，它们的值各是多少等等。但默认情况下，我们得到的信息只会是“类名+object at+内存地址”，对我们了解该实例化对象帮助不大。

那么，有没有可能自定义输出实例化对象时的信息呢？答案是肯定，通过重写类的 __repr__() 方法即可。事实上，当我们输出某个实例化对象时，其调用的就是该对象的 __repr__() 方法，输出的是该方法的返回值。

以本节开头的程序为例，执行 print(clangs) 等同于执行 print(clangs.__repr__())，程序的输出结果是一样的（输出的内存地址可能不同）。

和 __init__(self) 的性质一样，Python 中的每个类都包含 __repr__() 方法，因为 object 类包含 __reper__() 方法，而 Python 中所有的类都直接或间接继承自 object 类。

默认情况下，__repr__() 会返回和调用者有关的 “类名+object at+内存地址”信息。当然，我们还可以通过在类中重写这个方法，从而实现当输出实例化对象时，输出我们想要的信息。

举个例子：

```
class CLanguage:
    def __init__(self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def __repr__(self):
        return "CLanguage[name="+ self.name +",add=" + self.add +"]"
clangs = CLanguage()
print(clangs)
```

程序运行结果为：

```
CLanguage[name=C 语言中文网,add=http://c.biancheng.net]
```

由此可见，__repr__() 方法是类的实例化对象用来做“自我介绍”的方法，默认情况下，它会返回当前对象的“类名+object at+内存地址”，而如果对该方法进行重写，可以为其制作自定义的自我描述信息。

9.3 Python __del__()方法：销毁对象

我们知道，Python 通过调用 __init__() 方法构造当前类的实例化对象，而本节要学的 __del__() 方法，功能正好和 __init__() 相反，其用来销毁实例化对象。

事实上在编写程序时，如果之前创建的类实例化对象后续不再使用，最好在适当位置手动将其销毁，释放其占用的内存空间（整个过程称为垃圾回收（简称 GC））。

大多数情况下，Python 开发者不需要手动进行垃圾回收，因为 Python 有自动的垃圾回收机制（下面会讲），能自动将不需要使用的实例对象进行销毁。
无论是手动销毁，还是 Python 自动帮我们销毁，都会调用 __del__() 方法。举个例子：

```
class CLanguage:
    def __init__(self):
        print("调用 __init__() 方法构造对象")
    def __del__(self):
        print("调用__del__() 销毁对象，释放其空间")
clangs = CLanguage()
del clangs
```

程序运行结果为：

```
调用 __init__() 方法构造对象
调用__del__() 销毁对象，释放其空间
```

但是，读者千万不要误认为，只要为该实例对象调用 __del__() 方法，该对象所占用的内存空间就会被释放。举个例子：

```
class CLanguage:
    def __init__(self):
        print("调用 __init__() 方法构造对象")
    def __del__(self):
        print("调用__del__() 销毁对象，释放其空间")
clangs = CLanguage()
#添加一个引用 clangs 对象的实例对象
cl = clangs
del clangs
print("**********")
```

程序运行结果为：

```
调用 __init__() 方法构造对象
**********
调用__del__() 销毁对象，释放其空间
```

注意，最后一行输出信息，是程序执行即将结束时调用 __del__() 方法输出的。
可以看到，当程序中有其它变量（比如这里的 cl）引用该实例对象时，即便手动调用 __del__() 方法，该方法也不会立即执行。这和 Python 的垃圾回收机制的实现有关。

Python 采用自动引用计数（简称 ARC）的方式实现垃圾回收机制。该方法的核心思想是：每个 Python 对象都会配置一个计数器，初始 Python 实例对象的计数器值都为 0，如果有变量引用该实例对象，其计数器的值会加 1，依次类推；反之，每当一个变量取消对该实例对象的引用，计数器会减 1。如果一个 Python 对象的的计数器值为 0，则表明没有变量引用该 Python 对象，即证明程序不再需要它，此时 Python 就会自动调用 __del__() 方法将其回收。

以上面程序中的 clangs 为例，实际上构建 clangs 实例对象的过程分为 2 步，先使用 CLanguage() 调用该类中的 __init__() 方法构造出一个该类的对象（将其称为 C，计数器为 0），并立即用 clangs 这个变量作为所建实例对象的引用（ C 的计数器值 + 1）。在此基础上，又有一个 clang 变量引用 clangs（其实相当于引用 CLanguage()，此时 C 的计数器再 +1 ），这时如果调用 `del clangs` 语句，只会导致 C 的计数器减 1（值变为 1），因为 C 的计数器值不为 0，因此 C 不会被销毁（不会执行 __del__() 方法）。

如果在上面程序结尾，添加如下语句：

```
del cl
print("-----------")
```

则程序的执行结果为：

```
调用 __init__() 方法构造对象
**********
调用__del__() 销毁对象，释放其空间
----------
```

可以看到，当执行 del cl 语句时，其应用的对象实例对象 C 的计数器继续 -1（变为 0），对于计数器为 0 的实例对象，Python 会自动将其视为垃圾进行回收。

需要额外说明的是，如果我们重写子类的 __del__() 方法（父类为非 object 的类），则必须显式调用父类的 __del__() 方法，这样才能保证在回收子类对象时，其占用的资源（可能包含继承自父类的部分资源）能被彻底释放。为了说明这一点，这里举一个反例：

```
class CLanguage:
    def __del__(self):
        print("调用父类 __del__() 方法")

class cl(CLanguage):
    def __del__(self):
        print("调用子类 __del__() 方法")
c = cl()
del c
```

程序运行结果为：

```
调用子类 __del__() 方法
```

9.4 Python __dir__()用法：列出对象的所有属性（方法）名

前面在介绍 Python 内置函数时，提到了 dir() 函数，通过此函数可以某个对象拥有的所有的属性名和方法名，该函数会返回一个包含有所有属性名和方法名的有序列表。

举个例子：

```
class CLanguage:
    def __init__ (self,):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say():
        pass
clangs = CLanguage()
print(dir(clangs))
```

程序运行结果为：

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'add', 'name', 'say']
```

注意，通过 dir() 函数，不仅仅输出本类中新添加的属性名和方法（最后 3 个），还会输出从父类（这里为 object 类）继承得到的属性名和方法名。
值得一提的是，dir() 函数的内部实现，其实是在调用参数对象 __dir__() 方法的基础上，对该方法返回的属性名和方法名做了排序。

所以，除了使用 dir() 函数，我们完全可以自行调用该对象具有的 __dir__() 方法：

```
class CLanguage:
    def __init__ (self,):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say():
        pass
clangs = CLanguage()
print(clangs.__dir__())
```

程序运行结果为：

```
['name', 'add', '__module__', '__init__', 'say', '__dict__', '__weakref__', '__doc__', '__repr__', '__hash__', '__str__', '__getattribute__', '__setattr__', '__delattr__', '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__new__', '__reduce_ex__', '__reduce__', '__subclasshook__', '__init_subclass__', '__format__', '__sizeof__', '__dir__', '__class__']
```

显然，使用 __dir__() 方法和 dir() 函数输出的数据是相同，仅仅顺序不同。

9.5 Python __dict__属性：查看对象内部所有属性名和属性值组成的字典

在 Python 类的内部，无论是类属性还是实例属性，都是以字典的形式进行存储的，其中属性名作为键，而值作为该键对应的值。

为了方便用户查看类中包含哪些属性，Python 类提供了 __dict__ 属性。需要注意的一点是，该属性可以用类名或者类的实例对象来调用，用类名直接调用 __dict__，会输出该由类中所有类属性组成的字典；而使用类的实例对象调用 __dict__，会输出由类中所有实例属性组成的字典。

举个例子：

```
class CLanguage:
    a = 1
    b = 2
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
#通过类名调用__dict__
print(CLanguage.__dict__)

#通过类实例对象调用 __dict__
clangs = CLanguage()
print(clangs.__dict__)
```

程序输出结果为：

```
{'__module__': '__main__', 'a': 1, 'b': 2, '__init__': <function CLanguage.__init__ at 0x0000022C69833E18>, '__dict__': <attribute '__dict__' of 'CLanguage' objects>, '__weakref__': <attribute '__weakref__' of 'CLanguage' objects>, '__doc__': None}
{'name': 'C 语言中文网', 'add': 'http://c.biancheng.net'}
```

不仅如此，对于具有继承关系的父类和子类来说，父类有自己的 __dict__，同样子类也有自己的 __dict__，它不会包含父类的 __dict__。例如：

```
class CLanguage:
    a = 1
    b = 2
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"

class CL(CLanguage):
```

```
    c = 1
    d = 2
    def __init__ (self):
        self.na = "Python 教程"
        self.ad = "http://c.biancheng.net/python"
#父类名调用__dict__
print(CLanguage.__dict__)
#子类名调用__dict__
print(CL.__dict__)

#父类实例对象调用 __dict__
clangs = CLanguage()
print(clangs.__dict__)
#子类实例对象调用 __dict__
cl = CL()
print(cl.__dict__)
```

运行结果为：

```
{'__module__': '__main__', 'a': 1, 'b': 2, '__init__': <function CLanguage.__init__ at 0x000001721A853E18>, '__dict__': <attribute '__dict__' of 'CLanguage' objects>, '__weakref__': <attribute '__weakref__' of 'CLanguage' objects>, '__doc__': None}
{'__module__': '__main__', 'c': 1, 'd': 2, '__init__': <function CL.__init__ at 0x000001721CD15510>, '__doc__': None}
{'name': 'C 语言中文网', 'add': 'http://c.biancheng.net'}
{'na': 'Python 教程', 'ad': 'http://c.biancheng.net/python'}
```

显然，通过子类直接调用的 __dict__ 中，并没有包含父类中的 a 和 b 类属性；同样，通过子类对象调用的 __dict__，也没有包含父类对象拥有的 name 和 add 实例属性。

除此之外，借助由类实例对象调用 __dict__ 属性获取的字典，可以使用字典的方式对其中实例属性的值进行修改，例如：

```
class CLanguage:
    a = "aaa"
    b = 2
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"

#通过类实例对象调用 __dict__
clangs = CLanguage()
print(clangs.__dict__)
```

```
clangs.__dict__['name'] = "Python 教程"
print(clangs.name)
```

程序运行结果为：

```
{'name': 'C 语言中文网', 'add': 'http://c.biancheng.net'}
Python 教程
```

注意，无法通过类似的方式修改类变量的值。

9.6 Python setattr()、getattr()、hasattr()函数用法详解

除了前面介绍的几个类中的特殊方法外，本节再介绍 3 个常用的函数，分别是 hasattr()、getattr() 以及 setattr。

Python hasattr()函数

hasattr() 函数用来判断某个类实例对象是否包含指定名称的属性或方法。该函数的语法格式如下：

```
hasattr(obj, name)
```

其中 obj 指的是某个类的实例对象，name 表示指定的属性名或方法名。同时，该函数会将判断的结果（True 或者 False）作为返回值反馈回来。

举个例子：

```
class CLanguage:
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say(self):
        print("我正在学 Python")

clangs = CLanguage()
print(hasattr(clangs,"name"))
print(hasattr(clangs,"add"))
print(hasattr(clangs,"say"))
```

程序输出结果为：
True
True
True

显然，无论是属性名还是方法名，都在 hasattr() 函数的匹配范围内。因此，我们只能通过该函数判断实例对象是否包含该名称的属性或方法，但不能精确判断，该名称代表的是属性还是方法。

Python getattr() 函数

getattr() 函数获取某个类实例对象中指定属性的值。没错，和 hasattr() 函数不同，该函数只会从类对象包含的所有属性中进行查找。

getattr() 函数的语法格式如下：

```
getattr(obj, name[, default])
```

其中，obj 表示指定的类实例对象，name 表示指定的属性名，而 default 是可选参数，用于设定该函数的默认返回值，即当函数查找失败时，如果不指定 default 参数，则程序将直接报 AttributeError 错误，反之该函数将返回 default 指定的值。

举个例子：

```
class CLanguage:
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say(self):
        print("我正在学 Python")

clangs = CLanguage()
print(getattr(clangs,"name"))
print(getattr(clangs,"add"))
print(getattr(clangs,"say"))
print(getattr(clangs,"display",'nodisplay'))
```

程序执行结果为：

```
C 语言中文网
http://c.biancheng.net
<bound method CLanguage.say of <__main__.CLanguage object at 0x000001FC2F2E3198>>
nodisplay
```

可以看到，对于类中已有的属性，getattr() 会返回它们的值，而如果该名称为方法名，则返回该方法的状态信息；反之，如果该明白不为类对象所有，要么返回默认的参数，要么程序报 AttributeError 错误。

Python setattr()函数

setattr() 函数的功能相对比较复杂，它最基础的功能是修改类实例对象中的属性值。其次，它还可以实现为实例对象动态添加属性或者方法。

setattr() 函数的语法格式如下：

```
setattr(obj, name, value)
```

首先，下面例子演示如何通过该函数修改某个类实例对象的属性值：

```
class CLanguage:
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say(self):
```

```
        print("我正在学 Python")
clangs = CLanguage()
print(clangs.name)
print(clangs.add)
setattr(clangs,"name","Python 教程")
setattr(clangs,"add","http://c.biancheng.net/python")
print(clangs.name)
print(clangs.add)
```

程序运行结果为：

```
C 语言中文网
http://c.biancheng.net
Python 教程
http://c.biancheng.net/python
```

甚至利用 setattr() 函数，还可以将类属性修改为一个类方法，同样也可以将类方法修改成一个类属性。例如：

```
def say(self):
    print("我正在学 Python")

class CLanguage:
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"

clangs = CLanguage()
print(clangs.name)
print(clangs.add)
setattr(clangs,"name",say)
clangs.name(clangs)
```

程序运行结果为：

```
C 语言中文网
http://c.biancheng.net
我正在学 Python
```

显然，通过修改 name 属性的值为 say（这是一个外部定义的函数），原来的 name 属性就变成了一个 name() 方法。

使用 setattr() 函数对实例对象中执行名称的属性或方法进行修改时，如果该名称查找失败，Python 解释器不会报错，而是会给该实例对象动态添加一个指定名称的属性或方法。例如：

```
def say(self):
    print("我正在学 Python")

class CLanguage:
    pass

clangs = CLanguage()
setattr(clangs,"name","C 语言中文网")
setattr(clangs,"say",say)
print(clangs.name)
clangs.say(clangs)
```

程序执行结果为：

```
C 语言中文网
我正在学 Python
```

可以看到，虽然 CLanguage 为空类，但通过 setattr() 函数，我们为 clangs 对象动态添加了一个 name 属性和一个 say() 方法。

9.7Python issubclass 和 isinstance 函数：检查类型

Python 提供了如下两个函数来检查类型：

- issubclass(cls, class_or_tuple)：检查 cls 是否为后一个类或元组包含的多个类中任意类的子类。
- isinstance(obj, class_or_tuple)：检查 obj 是否为后一个类或元组包含的多个类中任意类的对象。

通过使用上面两个函数，程序可以方便地先执行检查，然后才调用方法，这样可以保证程序不会出现意外情况。

如下程序示范了通过这两个函数来检查类型：

```
# 定义一个字符串
hello = "Hello";
# "Hello"是 str 类的实例，输出 True
print('"Hello"是否是 str 类的实例：', isinstance(hello, str))
# "Hello"是 object 类的子类的实例，输出 True
print('"Hello"是否是 object 类的实例：', isinstance(hello, object))
# str 是 object 类的子类，输出 True
print('str 是否是 object 类的子类：', issubclass(str, object))
# "Hello"不是 tuple 类及其子类的实例，输出 False
print('"Hello"是否是 tuple 类的实例：', isinstance(hello, tuple))
# str 不是 tuple 类的子类，输出 False
print('str 是否是 tuple 类的子类：', issubclass(str, tuple))
# 定义一个列表
my_list = [2, 4]
# [2, 4]是 list 类的实例，输出 True
print('[2, 4]是否是 list 类的实例：', isinstance(my_list, list))
# [2, 4]是 object 类的子类的实例，输出 True
print('[2, 4]是否是 object 类及其子类的实例：', isinstance(my_list, object))
# list 是 object 类的子类，输出 True
print('list 是否是 object 类的子类：', issubclass(list, object))
# [2, 4]不是 tuple 类及其子类的实例，输出 False
print('[2, 4]是否是 tuple 类及其子类的实例：', isinstance([2, 4], tuple))
# list 不是 tuple 类的子类，输出 False
print('list 是否是 tuple 类的子类：', issubclass(list, tuple))
```

通过上面程序可以看出，issubclass() 和 isinstance() 两个函数的用法差不多，区别只是 issubclass() 的第一个参数是类名，而 isinstance() 的第一个参数是变量，这也与两个函数的意义对应：issubclass 用于判断是否为子类，而 isinstance() 用于判断是否为该

类或子类的实例。

issubclass() 和 isinstance() 两个函数的第二个参数都可使用元组。例如如下代码：

```
data = (20, 'fkit')
print('data 是否为列表或元组：', isinstance(data, (list, tuple))) # True
# str 不是 list 或者 tuple 的子类，输出 False
print('str 是否为 list 或 tuple 的子类：', issubclass(str, (list, tuple)))
# str 是 list 或 tuple 或 object 的子类，输出 True
print('str 是否为 list 或 tuple 或 object 的子类 ', issubclass(str, (list, tuple, object)))
```

此外，Python 为所有类都提供了一个 __bases__ 属性，通过该属性可以查看该类的所有直接父类，该属性返回所有直接父类组成的元组。例如如下代码：

```
class A:
    pass
class B:
    pass
class C(A, B):
    pass
print('类 A 的所有父类:', A.__bases__)
print('类 B 的所有父类:', B.__bases__)
print('类 C 的所有父类:', C.__bases__)
```

运行上面程序，可以看到如下运行结果：

```
类 A 的所有父类: (<class 'object'>,)
类 B 的所有父类: (<class 'object'>,)
类 C 的所有父类: (<class '__main__.A'>, <class '__main__.B'>)
```

从上面的运行结果可以看出，如果在定义类时没有显式指定它的父类，则这些类默认的父类是 object 类。

Python 还为所有类都提供了一个 __subclasses__() 方法，通过该方法可以查看该类的所有直接子类，该方法返回该类的所有子类组成的列表。例如在上面程序中增加如下两行：

```
print('类 A 的所有子类:', A.__subclasses__())
print('类 B 的所有子类:', B.__subclasses__())
```

运行上面代码，可以看到如下输出结果：

```
类 A 的所有子类: [<class '__main__.C'>]
类 B 的所有子类: [<class '__main__.C'>]
```

9.8 Python __call__()方法（详解版）

本节再介绍 Python 类中一个非常特殊的实例方法，即 __call__()。该方法的功能类似于在类中重载 () 运算符，使得类实例对象可以像调用普通函数那样，以“对象名()”的形式使用。

举个例子：

```
class CLanguage:
    # 定义__call__方法
    def __call__(self,name,add):
        print("调用__call__()方法",name,add)

clangs = CLanguage()
clangs("C 语言中文网","http://c.biancheng.net")
```

程序执行结果为：

```
调用__call__()方法 C 语言中文网 http://c.biancheng.net
```

可以看到，通过在 CLanguage 类中实现 __call__() 方法，使的 clangs 实例对象变为了可调用对象。

Python 中，凡是可以将 () 直接应用到自身并执行，都称为可调用对象。可调用对象包括自定义的函数、Python 内置函数以及本节所讲的类实例对象。
对于可调用对象，实际上“名称()”可以理解为是“名称.__call__()”的简写。仍以上面程序中定义的 clangs 实例对象为例，其最后一行代码还可以改写为如下形式：

```
clangs.__call__("C 语言中文网","http://c.biancheng.net")
```

运行程序会发现，其运行结果和之前完全相同。

这里再举一个自定义函数的例子，例如：

```
def say():
    print("Python 教程：http://c.biancheng.net/python")
say()
say.__call__()
```

程序执行结果为：

```
Python 教程：http://c.biancheng.net/python
Python 教程：http://c.biancheng.net/python
```

不仅如此，类中的实例方法也有以上 2 种调用方式，这里不再举例，有兴趣的读者可自行编写代码尝试。

用 __call__() 弥补 hasattr() 函数的短板

前面章节介绍了 hasattr() 函数的用法，该函数的功能是查找类的实例对象中是否包含指定名称的属性或者方法，但该函数有一个缺陷，即它无法判断该指定的名称，到底是类属性还是类方法。

要解决这个问题，我们可以借助可调用对象的概念。要知道，类实例对象包含的方法，其实也属于可调用对象，但类属性却不是。举个例子：

```
class CLanguage:
    def __init__ (self):
        self.name = "C 语言中文网"
        self.add = "http://c.biancheng.net"
    def say(self):
        print("我正在学 Python")

clangs = CLanguage()
if hasattr(clangs,"name"):
    print(hasattr(clangs.name,"__call__"))
print("**********")
if hasattr(clangs,"say"):
    print(hasattr(clangs.say,"__call__"))
```

程序执行结果为：

```
False
**********
True
```

可以看到，由于 name 是类属性，它没有以 __call__ 为名的 __call__() 方法；而 say 是类方法，它是可调用对象，因此它有 __call__() 方法。

9.9 什么是运算符重载，Python 可重载运算符有哪些？

前面章节介绍了 Python 中的各个序列类型，每个类型都有其独特的操作方法，例如列表类型支持直接做加法操作实现添加元素的功能，字符串类型支持直接做加法实现字符串的拼接功能，也就是说，同样的运算符对于不同序列类型的意义是不一样的，这是怎么做到的呢？

其实在 Python 内部，每种序列类型都是 Python 的一个类，例如列表是 list 类，字典是 dict 类等，这些序列类的内部使用了一个叫作“重载运算符”的技术来实现不同运算符所对应的操作。

所谓重载运算符，指的是在类中定义并实现一个与运算符对应的处理方法，这样当类对象在进行运算符操作时，系统就会调用类中相应的方法来处理。
这里给大家举一个与重载运算符相关的实例：

```
class MyClass: #自定义一个类
    def __init__(self, name , age): #定义该类的初始化函数
        self.name = name #将传入的参数值赋值给成员交量
        self.age = age
    def __str__(self): #用于将值转化为字符串形式，等同于 str(obj)
        return "name:"+self.name+";age:"+str(self.age)

    __repr__ = __str__ #转化为供解释器读取的形式

    def __lt__(self, record): #重载 self<record 运算符
        if self.age < record.age:
            return True
        else:
            return False

    def __add__(self, record): #重载 + 号运算符
        return MyClass(self.name, self.age+record.age)

myc = MyClass("Anna", 42) #实例化一个对象 Anna，并为其初始化
mycl = MyClass("Gary", 23) #实例化一个对象 Gary，并为其初始化
print(repr(myc)) #格式化对象 myc，
print(myc) #解释器读取对象 myc，调用 repr
print (str (myc)) #格式化对象 myc ，输出"name:Anna;age:42"
print(myc < mycl) #比较 myc<mycl 的结果，输出 False
print (myc+mycl) #进行两个 MyClass 对象的相加运算，输出 "name:Anna;age:65"
```

输出结果为：

```
name:Anna;age:42
name:Anna;age:42
name:Anna;age:42
False
name:Anna;age:65
```

这个例子中，MyClass 类中重载了 repr、str、<、+ 运算符，并用 MyClass 实例化了两个对象 myc 和 mycl。

通过将 myc 进行 repr、str 运算，从输出结果中可以看到，程序调用了重载的操作符方法 __repr__ 和 __str__。而令 myc 和 mycl 进行 < 号的比较运算以及加法运算，从输出结果中可以看出，程序调用了重载 < 号的方法 __lt__ 和 __add__ 方法。

那么，Python 类支持对哪些方法进行重载呢？这个给大家提供一个表格（表 1），列出了 Python 中常用的可重载的运算符，以及各自的含义。

表 1 Python 常用重载运算符

重载运算符	含义
__new__	创建类，在 __init__ 之前创建对象
__init__	类的构造函数，其功能是创建类对象时做初始化工作。
__del__	析构函数，其功能是销毁对象时进行回收资源的操作
__add__	加法运算符 +，当类对象 X 做例如 X+Y 或者 X+=Y 等操作，内部会调用此方法。但如果类中对 __iadd__ 方法进行了重载，则类对象 X 在做 X+=Y 类似操作时，会优先选择调用 __iadd__ 方法。
__radd__	当类对象 X 做类似 Y+X 的运算时，会调用此方法。
__iadd__	重载 += 运算符，也就是说，当类对象 X 做类似 X+=Y 的操作时，会调用此方法。
__or__	“或”运算符 |，如果没有重载 __ior__，则在类似 X|Y、X|=Y 这样的语句中，“或”符号生效
__repr__，__str__	格式转换方法，分别对应函数 repr(X)、str(X)
__call__	函数调用，类似于 X(*args, **kwargs) 语句
__getattr__	点号运算，用来获取类属性
__setattr__	属性赋值语句，类似于 X.any=value
__delattr__	删除属性，类似于 del X.any
__getattribute__	获取属性，类似于 X.any
__getitem__	索引运算，类似于 X[key]，X[i:j]
__setitem__	索引赋值语句，类似于 X[key], X[i:j]=sequence
__delitem__	索引和分片删除
__get__, __set__, __delete__	描述符属性，类似于 X.attr，X.attr=value，del X.attr
__len__	计算长度，类似于 len(X)
__lt__，__gt__，__le__，__ge__，__eq__，__ne__	比较，分别对应于 <、>、<=、>=、=、!= 运算符。
__iter__，__next__	迭代环境下，生成迭代器与取下一条，类似于 I=iter(X) 和 next()
__contains__	成员关系测试，类似于 item in X
__index__	整数值，类似于 hex(X)，bin(X)，oct(X)
__enter__，__exit__	在对类对象执行类似 with obj as var 的操作之前，会先调用 __enter__ 方法，其结果会传给 var；在最终结束该操作之前，会调用 __exit__ 方法（常用于做一些清理、扫尾的工作）

9.10 Python 重载运算符实现自定义序列

除了前面章节介绍的几个类特殊方法（方法名以双下划线（__）开头和结尾），在 Python 类中，我们还可以通过重写几个特殊方法，实现自定义一个序列类。表 1 列出了和自定义序列类有关的几个特殊方法。

表 1 和序列相关的特殊方法

方法名	功能
__len__(self)	返回序列类中存储元素的个数。
__contains__(self, value)	判断当前序列中是否包含 value 这个指定元素。
__getitem__(self, key)	通过指定的 key（键），返回对应的 value（值）。
__setitem__(self, key)	修改指定 key（键）对应的 value（值）。
__delitem__(self, key)	删除指定键值对。

注意，在对表 1 中的这些特殊方法进行重写时，在实现其基础功能的基础上，还可以根据实际情况，对各个方法的具体实现进行适当调整。以 __setitem__() 方法为例，当在序列中未找到指定 key 的情况下，该方法可以报错，当然也可以将此键值对添加到当前序列中。

另外值得一提的是，在实现自定义序列类时，并不是必须重写表 1 中全部的特殊方法。如果该自定义序列是一个不可变序列（即序列中的元素不能做修改），则无需重写 __setitem__() 和 __delitem__() 方法；反之，如果该自定义序列是一个可变序列，可以重写以上 5 个特殊方法。

下面程序实现了一个比较简单的序列类，这是一个字典类，其特点是只能存储 int 类型的元素：

```
class IntDic:
    def __init__(self):
        # 用于存储数据的字典
        self.__date = {}

    def __len__(self):
        return len(list(self.__date.values()))

    def __getitem__(self, key):
        # 如果在 self.__changed 中找到已经修改后的数据
        if key in self.__date :
            return self.__date[key]
        return None

    def __setitem__(self, key, value):
        #判断 value 是否为整数
        if not isinstance(value, int):
            raise TypeError('必须是整数')
```

```
        #修改现有 key 对应的 value 值，或者直接添加
        self.__date[key] = value

    def __delitem__(self, key):
        if key in self.__date : del self.__date[key]
dic = IntDic()
#输出序列中元素的个数，调用 __len__() 方法
print(len(dic))
#向序列中添加元素，调用 __setitem__() 方法
dic['a'] = 1
dic['b'] = 2

print(len(dic))
dic['a'] = 3
dic['c'] = 4
print(dic['a'])
#删除指定元素，调用 __delitem__() 方法
del dic['a']
print(dic['a'])
print(len(dic))
```

程序执行结果为：

```
0
2
3
None
2
```

9.11 什么是迭代器，Python 迭代器及其用法

前面章节中，已经对列表（list）、元组（tuple）、字典（dict）、集合（set）这些序列式容器做了详细的介绍。值得一提的是，这些序列式容器有一个共同的特性，它们都支持使用 for 循环遍历存储的元素，都是可迭代的，因此它们又有一个别称，即迭代器。

从字面来理解，迭代器指的就是支持迭代的容器，更确切的说，是支持迭代的容器类对象，这里的容器可以是列表、元组等这些 Python 提供的基础容器，也可以是自定义的容器类对象，只要该容器支持迭代即可。

《Python 实现自定义序列》一节中，已经学会了如何自定义一个序列类，但该序列类对象并不支持迭代，因此还不能称之为迭代器。如果要自定义实现一个迭代器，则类中必须实现如下 2 个方法：

1. __next__(self)：返回容器的下一个元素。
2. __iter__(self)：该方法返回一个迭代器（iterator）。

例如，下面程序自定义了一个简易的列表容器迭代器，支持迭代：

```
class listDemo:
    def __init__(self):
        self.__date=[]
        self.__step = 0
    def __next__(self):
        if self.__step <= 0:
            raise StopIteration
        self.__step -= 1
        #返回下一个元素
        return self.__date[self.__step]
    def __iter__(self):
        #实例对象本身就是迭代器对象，因此直接返回 self 即可
        return self
    #添加元素
    def __setitem__(self,key,value):
        self.__date.insert(key,value)
        self.__step += 1
mylist = listDemo()
mylist[0]=1
mylist[1]=2
for i in mylist:
    print (i)
```

程序执行结果为：

```
2
1
```

除此之外，Python 内置的 iter() 函数也会返回一个迭代器，该函数的语法格式如下：

```
iter(obj[, sentinel])
```

其中，obj 必须是一个可迭代的容器对象，而 sentinel 作为可选参数，如果使用此参数，要求 obj 必须是一个可调用对象，具体功能后面会讲。

可调用对象，指的是该类的实例对象可以像函数那样，直接以“对象名()”的形式被使用。通过在类中添加 __call__() 方法，就可以将该类的实例对象编程可调用对象。有关 __call__() 方法，可阅读《Python __call__()》做详细了解。
我们常用的是仅有 1 个参数的 iter() 函数，通过传入一个可迭代的容器对象，我们可以获得一个迭代器，通过调用该迭代器中的 __next__() 方法即可实现迭代。例如；

```
# 将列表转换为迭代器
myIter = iter([1, 2, 3])
# 依次获取迭代器的下一个元素
print(myIter.__next__())
print(myIter.__next__())
print(myIter.__next__())
print(myIter.__next__())
```

运行结果为：

```
1
2
3
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print(myIter.__next__())
StopIteration
```

另外，也可以使用 next() 内置函数来迭代，即 next(myIter)，和 __next__() 方法是完全一样的。
从程序的执行结果可以看出，当迭代完存储的所有元素之后，如果继续迭代，则 __next__() 方法会抛出 StopIteration 异常。

这里介绍 iter() 函数第 2 个参数的作用，如果使用该参数，则要求第一个 obj 参数必须传入可调用对象（可以不支持迭代），这样当使用返回的迭代器调用 __next__() 方法时，它会通过执行 obj() 调用 __call__() 方法，如果该方法的返回值和第 2 个参数值相同，则输出 StopInteration 异常；反之，则输出 __call__() 方法的返回值。

例如，修改 listDemo 类如下所示：

```
class listDemo:
    def __init__(self):
        self.__date=[]
        self.__step = 0

```

```
    def __setitem__(self,key,value):
        self.__date.insert(key,value)
        self.__step += 1
    #是该类实例对象成为可调用对象
    def __call__(self):
        self.__step-=1
        return self.__date[self.__step]

mylist = listDemo()
mylist[0]=1
mylist[1]=2
#将 mylist 变为迭代器
a = iter(mylist,1)
print(a.__next__())
print(a.__next__())
```

程序执行结果为：

```
2
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 20, in <module>
    print(a.__next__())
StopIteration
```

输出结果中，之所以最终抛出 StopIteration 异常，是因为这里原本要输出的元素 1 和 iter() 函数的第 2 个参数相同。

迭代器本身是一个底层的特性和概念，在程序中并不常用，但它为生成器这一更有趣的特性提供了基础。有关生成器的相关知识，会在后续章节中介绍。

9.12 Python 项目实战之迭代器实现字符串的逆序输出

《Python 迭代器》一节已经对如何创建迭代器做了详细的介绍，本节将利用迭代器完成对字符串的逆序操作。项目要求是这样的，定义一个类，要求在实现迭代器功能的基础上，能够对用户输入的字符串做逆序输出操作。

实现思路是这样的，自定义一个类并重载其 __init__() 初始化方法，实现为自身私有成员赋值。同时重载 __iter__() 和 __next__() 方法，使其具有迭代器功能。在此基础上，如果想实现对用户输入的字符串进行逆序输出，就需要在 __next__() 方法中实现从后往前返回字符。

实现代码如下：

```
1.  class Reverse:
2.      def __init__(self, string):
3.          self.__string = string
4.          self.__index = len(string)
5.      def __iter__(self):
6.          return self
7.      def __next__(self):
8.          self.__index -= 1
9.          return self.__string[self.__index]
10. revstr = Reverse('Python')
11. for c in revstr:
12.     print(c,end=" ")
```

运行结果为：

```
n o h t y P n o h t y P Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 11, in <module>
    for c in revstr:
  File "C:\Users\mengma\Desktop\demo.py", line 9, in __next__
    return self.__string[self.__index]
IndexError: string index out of range
```

可以看到，上面程序在逆序输出两遍"python"的同时，Python 解释器报出 IndexError 错误，这是什么原因呢？

很简单，因为程序没有设置遍历的终止条件，换句话说，没有对 __index 私有变量的值对限制，这里 __index 的取值范围应为（-len(self.__index), len(self.__index)），这也是导致上面程序运行结果的根本原因。

编写迭代器最容易忽视的一个环节，就是在自定义类中加入对循环结束的判断，并抛出 StopIteration 异常，只有这么做了，for 循环才会接收到 StopIteration 异常，并当做终止信号来结束循环。

所以，我们需要对上面程序做适当的调整，如下所示：

```
1.  class Reverse:
2.      def __init__(self, string):
```

```
3.            self.__string = string
4.            self.__index = len(string)
5.        def __iter__(self):
6.            return self
7.        def __next__(self):
8.            if self.__index == 0:
9.                raise(StopIteration)
10.           self.__index -= 1
11.           return self.__string[self.__index]
12.   revstr = Reverse('Python')
13.   for c in revstr:
14.       print(c,end=" ")
```

运行结果为：

```
n o h t y P
```

9.13 Python 生成器详解

前面章节中，已经详细介绍了什么是迭代器。生成器本质上也是迭代器，不过它比较特殊。

以 list 容器为例，在使用该容器迭代一组数据时，必须事先将所有数据存储到容器中，才能开始迭代；而生成器却不同，它可以实现在迭代的同时生成元素。

也就是说，对于可以用某种算法推算得到的多个数据，生成器并不会一次性生成它们，而是什么时候需要，才什么时候生成。
不仅如此，生成器的创建方式也比迭代器简单很多，大体分为以下 2 步：

1. 定义一个以 yield 关键字标识返回值的函数；
2. 调用刚刚创建的函数，即可创建一个生成器。

举个例子：

```
def intNum():
    print("开始执行")
    for i in range(5):
        yield i
        print("继续执行")
num = intNum()
```

由此，我们就成功创建了一个 num 生成器对象。显然，和普通函数不同，intNum() 函数的返回值用的是 yield 关键字，而不是 return 关键字，此类函数又成为生成器函数。

和 return 相比，yield 除了可以返回相应的值，还有一个更重要的功能，即每当程序执行完该语句时，程序就会暂停执行。不仅如此，即便调用生成器函数，Python 解释器也不会执行函数中的代码，它只会返回一个生成器（对象）。

要想使生成器函数得以执行，或者想使执行完 yield 语句立即暂停的程序得以继续执行，有以下 2 种方式：

1. 通过生成器（上面程序中的 num ）调用 next() 内置函数或者 __next__() 方法；
2. 通过 for 循环遍历生成器。

例如，在上面程序的基础上，添加如下语句：

```
#调用 next() 内置函数
print(next(num))
#调用 __next__() 方法
print(num.__next__())
#通过 for 循环遍历生成器
for i in num:
    print(i)
```

程序执行结果为：

```
开始执行
0
继续执行
1
继续执行
2
继续执行
3
继续执行
4
继续执行
```

这里有必要给读者分析一个程序的执行流程：
1) 首先，在创建有 num 生成器的前提下，通过其调用 next() 内置函数，会使 Python 解释器开始执行 intNum() 生成器函数中的代码，因此会输出“开始执行”，程序会一直执行到 `yield i`，而此时的 i==0，因此 Python 解释器输出“0”。由于受到 yield 的影响，程序会在此处暂停。

2) 然后，我们使用 num 生成器调用 __next__() 方法，该方法的作用和 next() 函数完全相同（事实上，next() 函数的底层执行的也是 __next__() 方法），它会是程序继续执行，即输出“继续执行”，程序又会执行到 `yield i`，此时 i==1，因此输出“1”，然后程序暂停。

3) 最后，我们使用 for 循环遍历 num 生成器，之所以能这么做，是因为 for 循环底层会不断地调用 next() 函数，使暂停的程序继续执行，因此会输出后续的结果。

注意，在 Python 2.x 版本中不能使用 __next__() 方法，可以使用 next() 内置函数，另外生成器还有 next() 方法（即以 num.next() 的方式调用）。

除此之外，还可以使用 list() 函数和 tuple() 函数，直接将生成器能生成的所有值存储成列表或者元组的形式。例如：

```
num = intNum()
print(list(num))

num = intNum()
print(tuple(num))
```

程序执行结果为：

```
开始执行
继续执行
继续执行
继续执行
继续执行
继续执行
[0, 1, 2, 3, 4]
开始执行
继续执行
继续执行
继续执行
继续执行
继续执行
(0, 1, 2, 3, 4)
```

通过输出结果可以判断出，list() 和 tuple() 底层实现和 for 循环的遍历过程是类似的。

相比迭代器，生成器最明显的优势就是节省内存空间，即它不会一次性生成所有的数据，而是什么时候需要，什么时候生成。

9.14 Python 生成器（send，close，throw）方法详解

《Python 生成器》一节中，详细介绍了如何创建一个生成器，以及生成器的基础用法。本节将在其基础上，继续讲解和生成器有关的一些方法。

Python 生成器 send()方法

我们知道，通过调用 next() 或者 __next__() 方法，可以实现从外界控制生成器的执行。除此之外，通过 send() 方法，还可以向生成器中传值。

值得一提的是，send() 方法可带一个参数，也可以不带任何参数（用 None 表示）。其中，当使用不带参数的 send() 方法时，它和 next() 函数的功能完全相同。例如：

```
def intNum():
    print("开始执行")
    for i in range(5):
        yield i
        print("继续执行")
num = intNum()
print(num.send(None))
print(num.send(None))
```

程序执行结果为：

```
开始执行
0
继续执行
1
```

注意，虽然 send(None) 的功能是 next() 完全相同，但更推荐使用 next()，不推荐使用 send(None)。
这里重点讲解一些带参数的 send(value) 的用法，其具备 next() 函数的部分功能，即将暂停在 yield 语句出的程序继续执行，但与此同时，该函数还会将 value 值作为 yield 语句返回值赋值给接收者。

注意，带参数的 send(value) 无法启动执行生成器函数。也就是说，程序中第一次使用生成器调用 next() 或者 send() 函数时，不能使用带参数的 send() 函数。

举个例子：

```
def foo():
    bar_a = yield "hello"
    bar_b = yield bar_a
    yield bar_b

f = foo()
```

```
print(f.send(None))
print(f.send("C 语言中文网"))
print(f.send("http://c.biancheng.net"))
```

分析一下此程序的执行流程：
1) 首先，构建生成器函数，并利用器创建生成器（对象）f 。

2) 使用生成器 f 调用无参的 send() 函数，其功能和 next() 函数完全相同，因此开始执行生成器函数，即执行到第一个 yield "hello" 语句，该语句会返回 "hello" 字符串，然后程序停止到此处（注意，此时还未执行对 bar_a 的赋值操作）。

3) 下面开始使用生成器 f 调用有参的 send() 函数，首先它会将暂停的程序开启，同时还会将其参数“C 语言中文网”赋值给当前 yield 语句的接收者，也就是 bar_a 变量。程序一直执行完 yield bar_a 再次暂停，因此会输出“C 语言中文网”。

4）最后依旧是调用有参的 send() 函数，同样它会启动餐厅的程序，同时将参数“http://c.biancheng.net”传给 bar_b，然后执行完 yield bar_b 后（输出 http://c.biancheng.net），程序执行再次暂停。

因此，该程序的执行结果为：

```
hello
C 语言中文网
http://c.biancheng.net
```

Python 生成器 close()方法

当程序在生成器函数中遇到 yield 语句暂停运行时，此时如果调用 close() 方法，会阻止生成器函数继续执行，该函数会在程序停止运行的位置抛出 GeneratorExit 异常。

举个例子：

```
def foo():
    try:
        yield 1
    except GeneratorExit:
        print('捕获到 GeneratorExit')
f = foo()
print(next(f))
f.close()
```

程序执行结果为：

```
1
捕获到 GeneratorExit
```

注意，虽然通过捕获 GeneratorExit 异常，可以继续执行生成器函数中剩余的代码，带这部分代码中不能再包含 yield 语句，否则程序会抛出 RuntimeError 异常。例如：

```
def foo():
```

```
    try:
        yield 1
    except GeneratorExit:
        print('捕获到 GeneratorExit')
        yield 2 #抛出 RuntimeError 异常

f = foo()
print(next(f))
f.close()
```

程序执行结果为：

```
1
捕获到 GeneratorExit Traceback (most recent call last):
  File "D:\python3.6\1.py", line 10, in <module>
    f.close()
RuntimeError: generator ignored GeneratorExit
```

另外，生成器函数一旦使用 close() 函数停止运行，后续将无法再调用 next() 函数或者 __next__() 方法启动执行，否则会抛出 StopIteration 异常。例如：

```
def foo():
    yield "c.biancheng.net"
    print("生成器停止执行")

f = foo()
print(next(f)) #输出 "c.biancheng.net"
f.close()
next(f) #原本应输出"生成器停止执行"
```

程序执行结果为：

```
c.biancheng.net
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 8, in <module>
    next(f) #原本应输出"生成器停止执行"
StopIteration
```

Python 生成器 throw()方法

生成器 throw() 方法的功能是，在生成器函数执行暂停处，抛出一个指定的异常，之后程序会继续执行生成器函数中后续的代码，直到遇到下一个 yield 语句。需要注意的是，如果到剩余代码执行完毕没有遇到下一个 yield 语句，则程序会抛出 StopIteration 异常。

举个例子：

```
def foo():
    try:
        yield 1
    except ValueError:
        print('捕获到 ValueError')

f = foo()
print(next(f))
f.throw(ValueError)
```

程序执行结果为：

```
1
捕获到 ValueError
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 9, in <module>
    f.throw(ValueError)
StopIteration
```

显然，一开始生成器函数在 yield 1 处暂停执行，当执行 throw() 方法时，它会先抛出 ValueError 异常，然后继续执行后续代码找到下一个 yield 语句，该程序中由于后续不再有 yield 语句，因此程序执行到最后，会抛出一个 StopIteration 异常。

9.15 Python @函数装饰器及用法（超级详细）

前面章节中，我们已经讲解了 Python 内置的 3 种函数装饰器，分别是 @staticmethod、@classmethod 和 @property，其中 staticmethod()、classmethod() 和 property() 都是 Python 的内置函数。

那么，函数装饰器的工作原理是怎样的呢？假设用 funA() 函数装饰器去装饰 funB() 函数，如下所示：

```
#funA 作为装饰器函数
def funA(fn):
    #...
    fn() # 执行传入的 fn 参数
    #...
    return '...'

@funA
def funB():
    #...
```

实际上，上面程序完全等价于下面的程序：

```
def funA(fn):
    #...
    fn() # 执行传入的 fn 参数
    #...
    return '...'

def funB():
    #...

funB = funA(funB)
```

通过比对以上 2 段程序不难发现，使用函数装饰器 A() 去装饰另一个函数 B()，其底层执行了如下 2 步操作：

1. 将 B 作为参数传给 A() 函数；
2. 将 A() 函数执行完成的返回值反馈回 B。

举个实例：

```
#funA 作为装饰器函数
def funA(fn):
```

```
    print("C 语言中文网")
    fn() # 执行传入的 fn 参数
    print("http://c.biancheng.net")
    return "装饰器函数的返回值"

@funA
def funB():
    print("学习 Python")
```

程序执行流程为：

```
C 语言中文网
学习 Python
http://c.biancheng.net
```

在此基础上，如果在程序末尾添加如下语句：

```
print(funB)
```

其输出结果为：

```
装饰器函数的返回值
```

显然，被“@函数”修饰的函数不再是原来的函数，而是被替换成一个新的东西（取决于装饰器的返回值），即如果装饰器函数的返回值为普通变量，那么被修饰的函数名就变成了变量名；同样，如果装饰器返回的是一个函数的名称，那么被修饰的函数名依然表示一个函数。

实际上，所谓函数装饰器，就是通过装饰器函数，在不修改原函数的前提下，来对函数的功能进行合理的扩充。

带参数的函数装饰器

在分析 funA() 函数装饰器和 funB() 函数的关系时，细心的读者可能会发现一个问题，即当 funB() 函数无参数时，可以直接将 funB 作为 funA() 的参数传入。但是，如果被修饰的函数本身带有参数，那应该如何传值呢？

比较简单的解决方法就是在函数装饰器中嵌套一个函数，该函数带有的参数个数和被装饰器修饰的函数相同。例如：

```
def funA(fn):
    # 定义一个嵌套函数
    def say(arc):
        print("Python 教程:",arc)
    return say

@funA
def funB(arc):
```

```
9.      print("funB():", a)
10. funB("http://c.biancheng.net/python")
```

程序执行结果为：

```
Python 教程: http://c.biancheng.net/python
```

这里有必要给读者分析一下这个程序，其实，它和如下程序是等价的：

```
1.  def funA(fn):
2.      # 定义一个嵌套函数
3.      def say(arc):
4.          print("Python 教程:",arc)
5.      return say
6.
7.  def funB(arc):
8.      print("funB():", a)
9.
10. funB = funA(funB)
11. funB("http://c.biancheng.net/python")
```

如果运行此程序会发现，它的输出结果和上面程序相同。

显然，通过 funB() 函数被装饰器 funA() 修饰，funB 就被赋值为 say。这意味着，虽然我们在程序显式调用的是 funB() 函数，但其实执行的是装饰器嵌套的 say() 函数。

但还有一个问题需要解决，即如果当前程序中，有多个（≥ 2）函数被同一个装饰器函数修饰，这些函数带有的参数个数并不相等，怎么办呢？

最简单的解决方式是用 *args 和 **kwargs 作为装饰器内部嵌套函数的参数，*args 和 **kwargs 表示接受任意数量和类型的参数。举个例子：

```
1.  def funA(fn):
2.      # 定义一个嵌套函数
3.      def say(*args,**kwargs):
4.          fn(*args,**kwargs)
5.      return say
6.
7.  @funA
8.  def funB(arc):
9.      print("C 语言中文网：",arc)
10.
```

```
@funA
def other_funB(name,arc):
    print(name,arc)
funB("http://c.biancheng.net")
other_funB("Python 教程：","http://c.biancheng.net/python")
```

运行结果为：

```
C 语言中文网： http://c.biancheng.net
Python 教程： http://c.biancheng.net/python
```

函数装饰器可以嵌套

上面示例中，都是使用一个装饰器的情况，但实际上，Python 也支持多个装饰器，比如：

```
@funA
@funB
@funC
def fun():
    #...
```

上面程序的执行顺序是里到外，所以它等效于下面这行代码：

```
fun = funA( funB ( funC (fun) ) )
```

这里不再给出具体实例，有兴趣的读者可自行编写程序进行测试。

9.16 Python 装饰器的应用场景

前面章节已经讲解了装饰器的基本概念及用法，本节将结合实际工作中的几个例子，带读者加深对它的理解。

装饰器用于身份认证

首先是最常见的身份认证的应用。这个很容易理解，举个最常见的例子，大家登录微信，需要输入用户名密码，然后点击确认，这样服务器端便会查询你的用户名是否存在、是否和密码匹配等等。如果认证通过，就可以顺利登录；反之，则提示你登录失败。

再比如一些网站，你不登录也可以浏览内容，但如果你想要发布文章或留言，在点击发布时，服务器端便会查询你是否登录。如果没有登录，就不允许这项操作等等。

如下是一个实现身份认证的简单示例：

```
1.  import functools
2.
3.  def authenticate(func):
4.      @functools.wraps(func)
5.      def wrapper(*args, **kwargs):
6.          request = args[0]
7.          # 如果用户处于登录状态
8.          if check_user_logged_in(request):
9.              # 执行函数 post_comment()
10.             return func(*args, **kwargs)
11.         else:
12.             raise Exception('Authentication failed')
13.     return wrapper
14.
15. @authenticate
16. def post_comment(request, ...)
17.     ...
```

注意，对于函数来说，它也有自己的一些属性，例如 __name__ 属性，代码中 @functools.wraps(func) 也是一个装饰器，如果不使用它，则 post_comment.__name__ 的值为 wrapper。而使用它之后，则 post_comment.__name__ 的值依然为 post_comment。

上面这段代码中，定义了装饰器 authenticate，函数 post_comment() 则表示发表用户对某篇文章的评论，每次调用这个函数前，都会先检查用户是否处于登录状态，如果是登录状态，则允许这项操作；如果没有登录，则不允许。

装饰器用于日志记录

日志记录同样是很常见的一个案例。在实际工作中，如果你怀疑某些函数的耗时过长，导致整个系统的延迟增加，想在线上测试某些函数的执行时间，那么，装饰器就是一种很常用的手段。

我们通常用下面的方法来表示：

```
import time
import functools

def log_execution_time(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        res = func(*args, **kwargs)
        end = time.perf_counter()
        print('{} took {} ms'.format(func.__name__, (end - start) * 1000))
        return res
    return wrapper

@log_execution_time
def calculate_similarity(items):
    ...
```

这里，装饰器 log_execution_time 记录某个函数的运行时间，并返回其执行结果。如果你想计算任何函数的执行时间，在这个函数上方加上@log_execution_time 即可。

装饰器用于输入合理性检查

在大型公司的机器学习框架中，调用机器集群进行模型训练前，往往会用装饰器对其输入（往往是很长的 json 文件）进行合理性检查。这样就可以大大避免输入不正确对机器造成的巨大开销。

它的写法往往是下面的格式：

```
import functools

def validation_check(input):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        ... # 检查输入是否合法
```

```
7.
8.  @validation_check
9.  def neural_network_training(param1, param2, ...):
10.     ...
```

其实在工作中，很多情况下都会出现输入不合理的现象。因为我们调用的训练模型往往很复杂，输入的文件有成千上万行，很多时候确实也很难发现。

试想一下，如果没有输入的合理性检查，很容易出现“模型训练了好几个小时后，系统却报错说输入的一个参数不对，成果付之一炬”的现象。这样的“惨案”，大大减缓了开发效率，也对机器资源造成了巨大浪费。

缓存装饰器

关于缓存装饰器的用法，其实十分常见，这里以 Python 内置的 LRU cache 为例来说明。

LRU cache，在 Python 中的表示形式是 @lru_cache。@lru_cache 会缓存进程中的函数参数和结果，当缓存满了以后，会删除最近最久未使用的数据。

正确使用缓存装饰器，往往能极大地提高程序运行效率。举个例子，大型公司服务器端的代码中往往存在很多关于设备的检查，比如使用的设备是安卓还是 iPhone，版本号是多少。这其中的一个原因，就是一些新的功能，往往只在某些特定的手机系统或版本上才有（比如 Android v200+）。

这样一来，我们通常使用缓存装饰器来包裹这些检查函数，避免其被反复调用，进而提高程序运行效率，比如写成下面这样：


```
1.  @lru_cache
2.  def check(param1, param2, ...) # 检查用户设备类型，版本号等等
3.      ...
```

第 10 章：Python 异常处理机制

10.1 什么是异常处理，Python 常见异常类型（入门必读）

开发人员在编写程序时，难免会遇到错误，有的是编写人员疏忽造成的语法错误，有的是程序内部隐含逻辑问题造成的数据错误，还有的是程序运行时与系统的规则冲突造成的系统错误，等等。

总的来说，编写程序时遇到的错误可大致分为 2 类，分别为语法错误和运行时错误。

Python 语法错误

语法错误，也就是解析代码时出现的错误。当代码不符合 Python 语法规则时，Python 解释器在解析时就会报出 SyntaxError 语法错误，与此同时还会明确指出最早探测到错误的语句。例如：

```
print "Hello,World!"
```

我们知道，Python 3 已不再支持上面这种写法，所以在运行时，解释器会报如下错误：

```
SyntaxError: Missing parentheses in call to 'print'
```

语法错误多是开发者疏忽导致的，属于真正意义上的错误，是解释器无法容忍的，因此，只有将程序中的所有语法错误全部纠正，程序才能执行。

Python 运行时错误

运行时错误，即程序在语法上都是正确的，但在运行时发生了错误。例如：

```
a = 1/0
```

上面这句代码的意思是“用 1 除以 0，并赋值给 a 。因为 0 作除数是没有意义的，所以运行后会产生如下错误：

```
>>> a = 1/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    a = 1/0
ZeroDivisionError: division by zero
```

以上运行输出结果中，前两段指明了错误的位置，最后一句表示出错的类型。在 Python 中，把这种运行时产生错误的情况叫做**异常（Exceptions）**。这种异常情况还有很多，常见的几种异常情况如表 1 所示。

表 1 Python 常见异常类型

异常类型	含义	实例

AssertionError	当 assert 关键字后的条件为假时，程序运行会停止并抛出 AssertionError 异常	>>> demo_list = ['C 语言中文网']
		>>> assert len(demo_list) > 0
		>>> demo_list.pop()
		'C 语言中文网'
		>>> assert len(demo_list) > 0
		Traceback (most recent call last):
		File "<pyshell#6>", line 1, in <module>
		assert len(demo_list) > 0
		AssertionError

AttributeError	当试图访问的对象属性不存在时抛出的异常	>>> demo_list = ['C 语言中文网']
		>>> demo_list.len
		Traceback (most recent call last):
		File "<pyshell#10>", line 1, in <module>
		demo_list.len
		AttributeError: 'list' object has no attribute 'len'

IndexError	索引超出序列范围会引发此异常	>>> demo_list = ['C 语言中文网']
		>>> demo_list[3]
		Traceback (most recent call last):
		File "<pyshell#8>", line 1, in <module>
		demo_list[3]
		IndexError: list index out of range

KeyError	字典中查找一个不存在的关键字时引发此异常	>>> demo_dict={'C 语言中文网
		':"c.biancheng.net"}
		>>> demo_dict["C 语言"]
		Traceback (most recent call last):
		File "<pyshell#12>", line 1, in <module>
		demo_dict["C 语言"]
		KeyError: 'C 语言'

NameError	尝试访问一个未声明的变量时，引发此异常	>>> C 语言中文网 Traceback (most recent call last): File "<pyshell#15>", line 1, in <module> C 语言中文网 NameError: name 'C 语言中文网' is not defined
TypeError	不同类型数据之间的无效操作	>>> 1+'C 语言中文网' Traceback (most recent call last): File "<pyshell#17>", line 1, in <module> 1+'C 语言中文网' TypeError: unsupported operand type(s) for +: 'int' and 'str'
ZeroDivisionError	除法运算中除数为 0 引发此异常	>>> a = 1/0 Traceback (most recent call last): File "<pyshell#2>", line 1, in <module> a = 1/0 ZeroDivisionError: division by zero

提示：表中的异常类型不需要记住，只需简单了解即可。

当一个程序发生异常时，代表该程序在执行时出现了非正常的情况，无法再执行下去。默认情况下，程序是要终止的。如果要避免程序退出，可以使用捕获异常的方式获取这个异常的名称，再通过其他的逻辑代码让程序继续运行，这种根据异常做出的逻辑处理叫作异常处理。

开发者可以使用异常处理全面地控制自己的程序。异常处理不仅仅能够管理正常的流程运行，还能够在程序出错时对程序进行必的处理。大大提高了程序的健壮性和人机交互的友好性。

那么，应该如何捕获和处理异常呢？可以使用 try 语句来实现。有关 try 语句的语法和用法，会在后续章节继续详解。

10.2Python 异常处理机制到底有什么用？

异常处理是现代编程语言不可或缺的能力，它已经成为衡量一门编程语言是否成熟和健壮的标准之一，C++、Java、C#、Python 等高级语言都提供了异常处理机制。

无论你是多么优秀的程序员，你都不能保证自己的程序永远不会出错。就算你的程序没有错，用户也不一定按照你设定的规则来使用你的程序，总有一些小白或者极客会“玩弄”你的程序。

除此以外，你也不能保证程序的运行环境永远稳定，比如操作系统可能崩溃，网络可能无法连接，内存可能突然坏掉……

总之，你基本什么都保证不了。但是，作为一个负责任的程序员，我们要让自己的程序尽可能的健壮，尽可能保证在恶劣环境下还能正常运行，或者给用户提示错误，让用户决定是否退出。

例如有一个五子棋程序，当用户输入落子的坐标时，程序既要判断输入格式是否正确（横坐标和纵坐标之间由逗号分隔），还要判断坐标是否在合法的范围内。一般我们都会这样来处理：

```
if 坐标包含了除逗号之外的其它非数字字符：
    alert 坐标只能是数值
    goto retry
elif 坐标不包含逗号：
    alert 必须使用逗号分隔横坐标和纵坐标
    goto retry
elif 坐标落在了棋盘外：
    alert 坐标必须位于棋盘之内
    goto retry
elif 作为位置已有其它棋子：
    alert 只能在没有棋子的位置落子
    goto retry
else:
    #正常的业务代码
    ……
```

上面的代码并没有涉及所有出错情形，只是考虑了四种可能出错的情形，代码量就已经急剧增加了。

在实际开发中，不可预料的情况呈数量级增长，甚至不能穷举，按照上面的逻辑来处理各种错误简直让人抓狂。

如果每次在实现真正的业务逻辑之前，都需要不厌其烦地考虑各种可能出错的情况，针对各种错误情况给出补救措施，这是多么乏味的事情啊。程序员喜欢解决问题，喜欢开发带来的“创造”快感，但不喜欢像一个“堵漏”工人，去堵那些由外在条件造成的“漏洞”。

对于构造大型、健壮、可维护的应用而言，错误处理是整个应用需要考虑的重要方面，程序员不能仅仅只做“对”的事情，程序员开发程序的过程，是一个创造的过程，这个过程需要有全面的考虑，仅做“对”的事情是远远不够的。

对于上面的错误处理机制，主要有如下两个缺点：

- 无法穷举所有的异常情况。因为人类知识的限制，异常情况总比可以考虑到的情况多，总有“漏网之鱼”的异常情况，所以程序总是不够健壮。
- 错误处理代码和业务实现代码混杂。这种错误处理和业务实现混杂的代码严重影响程序的可读性，会增加程序维护的难度。

程序员希望有一种强大的机制来解决上面的问题，能够将上面程序改成如下的形式：

```
if 用户输入不合法:
    alert 输入不合法
    goto retry
else :
```

```
    #正常的业务代码
    ......
```

上面伪码提供了一个非常强大的“if 块”，即程序不管输入错误的原因是什么，只要用户输入不满足要求，程序就一次处理所有的错误。这种处理方法的好处是，使得错误处理代码变得更有条理，只需在一个地方处理错误。

现在的问题是，“用户输入不合法”这个条件怎么定义？当然，对于这个简单的要求，可以使用正则表达式对用户输入进行匹配，当用户输入与正则表达式不匹配时即可判断“用户输入不合法”。但对于更复杂的情形，就没有这么简单了。使用 Python 的异常处理机制就可以解决这个问题，例如：

```
try：
    if(用户输入不合理)：
        raise 异常
except Exception：
    alert 输入不合法
    goto retry
#正常的业务代码
```

此程序中，通过在 try 块中判断用户的输入数据是否合理，如果不合理，程序受 raise 的影响会进行到 except 代码块，对用户的错误输出进行处理，然后会继续执行正常的业务代码；反之，如果用户输入合理，那么程序将直接执行正常的业务代码。

try except 是 Python 实现异常处理机制的核心结构，其具体用法会在后续章节做详细介绍。

显然，使用 Python 异常处理机制，可以让程序中的异常处理代码和正常业务代码分离，使得程序代码更加优雅，并可以提高程序的健壮性。

10.3 Python try except 异常处理详解（入门必读）

Python 中，用 try except 语句块捕获并处理异常，其基本语法结构如下所示：

```
try:
    可能产生异常的代码块
except [ (Error1, Error2, ... ) [as e] ]:
    处理异常的代码块 1
except [ (Error3, Error4, ... ) [as e] ]:
    处理异常的代码块 2
except  [Exception]:
    处理其它异常
```

该格式中，[] 括起来的部分可以使用，也可以省略。其中各部分的含义如下：

- (Error1, Error2,...) 、(Error3, Error4,...)：其中，Error1、Error2、Error3 和 Error4 都是具体的异常类型。显然，一个 except 块可以同时处理多种异常。
- [as e]：作为可选参数，表示给异常类型起一个别名 e，这样做的好处是方便在 except 块中调用异常类型（后续会用到）。
- [Exception]：作为可选参数，可以代指程序可能发生的所有异常情况，其通常用在最后一个 except 块。

从 try except 的基本语法格式可以看出，try 块有且仅有一个，但 except 代码块可以有多个，且每个 except 块都可以同时处理多种异常。

当程序发生不同的意外情况时，会对应特定的异常类型，Python 解释器会根据该异常类型选择对应的 except 块来处理该异常。
try except 语句的执行流程如下：

1. 首先执行 try 中的代码块，如果执行过程中出现异常，系统会自动生成一个异常类型，并将该异常提交给 Python 解释器，此过程称为捕获异常。
2. 当 Python 解释器收到异常对象时，会寻找能处理该异常对象的 except 块，如果找到合适的 except 块，则把该异常对象交给该 except 块处理，这个过程被称为处理异常。如果 Python 解释器找不到处理异常的 except 块，则程序运行终止，Python 解释器也将退出。

事实上，不管程序代码块是否处于 try 块中，甚至包括 except 块中的代码，只要执行该代码块时出现了异常，系统都会自动生成对应类型的异常。但是，如果此段程序没有用 try 包裹，又或者没有为该异常配置处理它的 except 块，则 Python 解释器将无法处理，程序就会停止运行；反之，如果程序发生的异常经 try 捕获并由 except 处理完成，则程序可以继续执行。

举个例子：

```
try:
    a = int(input("输入被除数："))
    b = int(input("输入除数："))
    c = a / b
    print("您输入的两个数相除的结果是：", c )
except (ValueError, ArithmeticError):
    print("程序发生了数字格式异常、算术异常之一")
except :
    print("未知异常")
```

```
10.  print("程序继续运行")
```

程序运行结果为：

```
输入被除数：a
程序发生了数字格式异常、算术异常之一
程序继续运行
```

上面程序中，第 6 行代码使用了（ValueError, ArithmeticError）来指定所捕获的异常类型，这就表明该 except 块可以同时捕获这 2 种类型的异常；第 8 行代码只有 except 关键字，并未指定具体要捕获的异常类型，这种省略异常类的 except 语句也是合法的，它表示可捕获所有类型的异常，一般会作为异常捕获的最后一个 except 块。

除此之外，由于 try 块中引发了异常，并被 except 块成功捕获，因此程序才可以继续执行，才有了“程序继续运行”的输出结果。

获取特定异常的有关信息

通过前面的学习，我们已经可以捕获程序中可能发生的异常，并对其进行处理。但是，由于一个 except 可以同时处理多个异常，那么我们如何知道当前处理的到底是哪种异常呢？

其实，每种异常类型都提供了如下几个属性和方法，通过调用它们，就可以获取当前处理异常类型的相关信息：

- args：返回异常的错误编号和描述字符串；
- str(e)：返回异常信息，但不包括异常信息的类型；
- repr(e)：返回较全的异常信息，包括异常信息的类型。

举个例子：

```
1.  try:
2.      1/0
3.  except Exception as e:
4.      # 访问异常的错误编号和详细信息
5.      print(e.args)
6.      print(str(e))
7.      print(repr(e))
```

输出结果为：

```
('division by zero',)
division by zero
ZeroDivisionError('division by zero',)
```

除此之外，如果想要更加详细的异常信息，可以使用 traceback 模块。有兴趣的读者，可自行查阅资料学习。
从程序中可以看到，由于 except 可能接收多种异常，因此为了操作方便，可以直接给每一个进入到此 except 块的异常，起一个统一的别名 e。

在 Python 2.x 的早期版本中，除了使用 as e 这个格式，还可以将其中的 as 用逗号（ , ）代替。

10.4 Python 异常处理机制的底层实现

前面章节中，我们详细介绍了 try except 异常处理的用法，简单来说，当位于 try 块中的程序执行出现异常时，会将该种异常捕获，同时找到对应的 except 块处理该异常，那么这里就有一个问题，它是如何找到对应的 except 块的呢？

我们知道，一个 try 块也可以对应多个 except 块，一个 except 块可以同时处理多种异常。如果我们想使用一个 except 块处理所有异常，就可以这样写：

```
try:
    #...
except Exception:
    #...
```

这种情况下，对于 try 块中可能出现的任何异常，Python 解释器都会交给仅有的这个 except 块处理，因为它的参数是 Exception，表示可以接收任何类型的异常。

注意，对于可以接收任何异常的 except 来说，其后可以跟 Exception，也可以不跟任何参数，但表示的含义都是一样的。
这里就要详细介绍一下 Exception。要知道，为了表示程序中可能出现的各种异常，Python 提供了大量的异常类，这些异常类之间有严格的继承关系，图 1 显示了 Python 的常见异常类之间的继承关系。

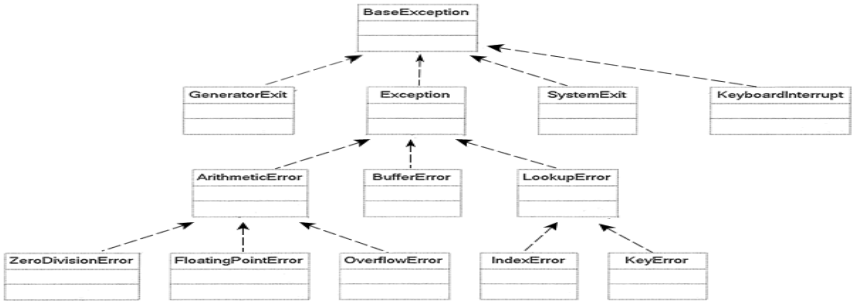


图 1 Python 的常见异常类之间的继承关系

从图 1 中可以看出，BaseException 是 Python 中所有异常类的基类，但对于我们来说，最主要的是 Exception 类，因为程序中可能出现的各种异常，都继承自 Exception。

因此，如果用户要实现自定义异常，不应该继承 BaseException ，而应该继承 Exception 类。关于如何自定义一个异常类，可阅读《Python 自定义异常类》一节。
当 try 块捕获到异常对象后，Python 解释器会拿这个异常类型依次和各个 except 块指定的异常类进行比较，如果捕获到的这个异常类，和某个 except 块后的异常类一样，又或者是该异常类的子类，那么 Python 解释器就会调用这个 except 块来处理异常；反之，Python 解释器会继续比较，直到和最后一个 except 比较完，如果没有比对成功，则证明该异常无法处理。

图 2 演示了位于 try 块中的程序发生异常时，从捕获异常到处理异常的整个流程。

图 2 Python 异常捕获流程示意图

下面看几个简单的异常捕获的例子：

```
try:
        a = int(input("输入 a: "))
        b = int(input("输入 b: "))
        print( a/b )
except ValueError:
        print("数值错误: 程序只能接收整数参数")
except ArithmeticError:
        print("算术错误")
except Exception:
        print("未知异常")
```

该程序中，根据用户输入 a 和 b 值的不同，可能会导致 ValueError、ArithmeticError 异常：

1. 如果用户输入的 a 或者 b 是其他字符，而不是数字，会发生 ValueError 异常，try 块会捕获到该类型异常，同时 Python 解释器会调用第一个 except 块处理异常；
2. 如果用户输入的 a 和 b 是数字，但 b 的值为 0，由于在进行除法运算时除数不能为 0，因此会发生 ArithmeticError 异常，try 块会捕获该异常，同时 Python 解释器会调用第二个 except 块处理异常；
3. 当然，程序运行过程中，还可能由于其他因素出现异常，try 块都可以捕获，同时 Python 会调用最后一个 except 块来处理。

当一个 try 块配有多个 except 块时，这些 except 块应遵循这样一个排序规则，即可处理全部异常的 except 块（参数为 Exception，也可以什么都不写）要放到所有 except 块的后面，且所有父类异常的 except 块要放到子类异常的 except 块的后面。

10.5 Python try except else 详解

在原本的 try except 结构的基础上，Python 异常处理机制还提供了一个 else 块，也就是原有 try except 语句的基础上再添加一个 else 块，即 try except else 结构。

使用 else 包裹的代码，只有当 try 块没有捕获到任何异常时，才会得到执行；反之，如果 try 块捕获到异常，即便调用对应的 except 处理完异常，else 块中的代码也不会得到执行。

举个例子：

```
try:
    result = 20 / int(input('请输入除数:'))
    print(result)
except ValueError:
    print('必须输入整数')
except ArithmeticError:
    print('算术错误，除数不能为 0')
else:
    print('没有出现异常')
print("继续执行")
```

可以看到，在原有 try except 的基础上，我们为其添加了 else 块。现在执行该程序：

```
请输入除数:4
5.0
没有出现异常
继续执行
```

如上所示，当我们输入正确的数据时，try 块中的程序正常执行，Python 解释器执行完 try 块中的程序之后，会继续执行 else 块中的程序，继而执行后续的程序。

读者可能会问，既然 Python 解释器按照顺序执行代码，那么 else 块有什么存在的必要呢？直接将 else 块中的代码编写在 try except 块的后面，不是一样吗？

当然不一样，现在再次执行上面的代码：

```
请输入除数:a
必须输入整数
继续执行
```

可以看到，当我们试图进行非法输入时，程序会发生异常并被 try 捕获，Python 解释器会调用相应的 except 块处理该异常。但是异常处理完毕之后，Python 解释器并没有接着执行 else 块中的代码，而是跳过 else，去执行后续的代码。

也就是说，else 的功能，只有当 try 块捕获到异常时才能显现出来。在这种情况下，else 块中的代码不会得到执行的机会。而如果我们直接把 else 块去掉，将其中的代码编写到 try except 的后面：

```
try:
    result = 20 / int(input('请输入除数:'))
    print(result)
```

```
except ValueError:
    print('必须输入整数')
except ArithmeticError:
    print('算术错误，除数不能为 0')
print('没有出现异常')
print("继续执行")
```

程序执行结果为：

```
请输入除数:a
必须输入整数
没有出现异常
继续执行
```

可以看到，如果不使用 else 块，try 块捕获到异常并通过 except 成功处理，后续所有程序都会依次被执行。

10.6 Python try except finally：资源回收

Python 异常处理机制还提供了一个 finally 语句，通常用来为 try 块中的程序做扫尾清理工作。

注意，和 else 语句不同，finally 只要求和 try 搭配使用，而至于该结构中是否包含 except 以及 else，对于 finally 不是必须的（else 必须和 try except 搭配使用）。
在整个异常处理机制中，finally 语句的功能是：无论 try 块是否发生异常，最终都要进入 finally 语句，并执行其中的代码块。

基于 finally 语句的这种特性，在某些情况下，当 try 块中的程序打开了一些物理资源（文件、数据库连接等）时，由于这些资源必须手动回收，而回收工作通常就放在 finally 块中。

Python 垃圾回收机制，只能帮我们回收变量、类对象占用的内存，而无法自动完成类似关闭文件、数据库连接等这些的工作。
读者可能会问，回收这些物理资源，必须使用 finally 块吗？当然不是，但使用 finally 块是比较好的选择。首先，try 块不适合做资源回收工作，因为一旦 try 块中的某行代码发生异常，则其后续的代码将不会得到执行；其次 except 和 else 也不适合，它们都可能不会得到执行。而 finally 块中的代码，无论 try 块是否发生异常，该块中的代码都会被执行。

举个例子：

```
try:
    a = int(input("请输入 a 的值:"))
    print(20/a)
except:
    print("发生异常！")
else:
    print("执行 else 块中的代码")
finally :
    print("执行 finally 块中的代码")
```

运行此程序：

```
请输入 a 的值:4
5.0
执行 else 块中的代码
执行 finally 块中的代码
```

可以看到，当 try 块中代码为发生异常时，except 块不会执行，else 块和 finally 块中的代码会被执行。

再次运行程序：

```
请输入 a 的值:a
发生异常！
执行 finally 块中的代码
```

可以看到，当 try 块中代码发生异常时，except 块得到执行，而 else 块中的代码将不执行，finally 块中的代码仍然会被执行。

finally 块的强大还远不止此，即便当 try 块发生异常，且没有合适和 except 处理异常时，finally 块中的代码也会得到执行。例如：

```
try:
    #发生异常
```

```
    print(20/0)

finally :
    print("执行 finally 块中的代码")
```

程序执行结果为：

```
执行 finally 块中的代码
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 3, in <module>
    print(20/0)
ZeroDivisionError: division by zero
```

可以看到，当 try 块中代码发生异常，导致程序崩溃时，在崩溃前 Python 解释器也会执行 finally 块中的代码。

10.7 Python 异常处理机制结构详解

到本节为止，读者已经学习了整个 Python 的异常处理机制的结构，接下来带领大家回顾一下，在此过程还会讲解一些新的知识。

首先，Python 完整的异常处理语法结构如下：

```
try:
    #业务实现代码
except Exception1 as e:
    #异常处理块 1
    ...
except Exception2 as e:
    #异常处理块 2
    ...
#可以有多个 except
...
else:
    #正常处理块
finally :
    #资源回收块
    ...
```

整个异常处理结构的执行过程，如图 1 所示。

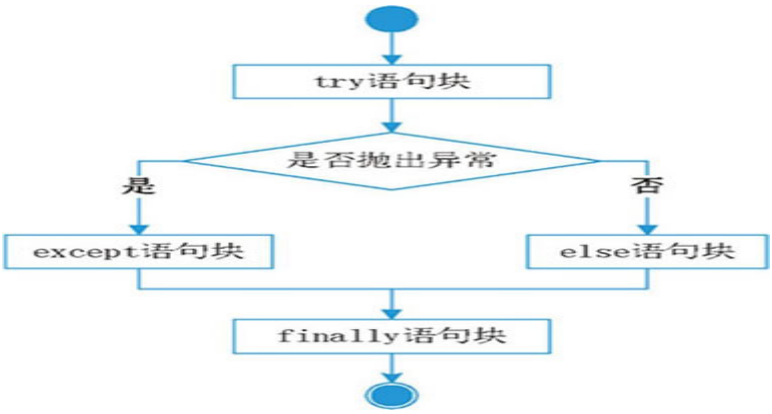


图 1 异常处理语句块的执行流程

注意，在整个异常处理结构中，只有 try 块是必需的，也就是说：

- 如果没有 try 块，则不能有后面的 except 块、else 块和 finally 块。但是也不能只使用 try 块，要么使用 try except 结构，要么使用 try finally 结构；
- except 块、else 块、finally 块都是可选的，当然也可以同时出现；
- 可以有多个 except 块，但捕获父类异常的 except 块应该位于捕获子类异常的 except 块的后面；
- 多个 except 块必须位于 try 块之后，finally 块必须位于所有的 except 块之后。
- 要使用 else 块，其前面必须包含 try 和 except。

其中，很多初学者分不清 finally 和 else 的区别，这里着重说一下。else 语句块只有在没有异常发生的情况下才会执行，而 finally 语句则不管异常是否发生都会执行。不仅如此，无论是正常退出、遇到异常退出，还是通过 break、continue、return 语句退出，finally 语句块都会执行。

注意，如果程序中运行了强制退出 Python 解释器的语句（如 os._exit(1)），则 finally 语句将无法得到执行。例如：

```
import os
try:
    os._exit(1)
finally:
    print("执行 finally 语句")
```

运行程序，没有任何输出。因此，除非在 try 块、except 块中调用了退出 Python 解释器的方法，否则不管在 try 块、except 块中执行怎样的代码，出现怎样的情况，异常处理的 finally 块总会被执行。

另外在通常情况下，不要在 finally 块中使用如 return 或 raise 等导致方法中止的语句（raise 语句将在后面介绍），一旦在 finally 块中使用了 return 或 raise 语句，将会导致 try 块、except 块中的 return、raise 语句失效。看如下程序：

```
def test():
    try:
        # 因为 finally 块中包含了 return 语句
        # 所以下面的 return 语句失去作用
        return True
    finally:
        return False
print(test())
```

上面程序在 finally 块中定义了一条 return False 语句，这将导致 try 块中的 return true 失去作用。运行上面程序，输出结果为：

```
False
```

同样，如果 Python 程序在执行 try 块、except 块包含有 return 或 raise 语句，则 Python 解释器执行到该语句时，会先去查找 finally 块，如果没有 finally 块，程序才会立即执行 return 或 raise 语句；反之，如果找到 finally 块，系统立即开始执行 finally 块，只有当 finally 块执行完成后，系统才会再次跳回来执行 try 块、except 块里的 return 或 raise 语句。

但是，如果在 finally 块里也使用了 return 或 raise 等导致方法中止的语句，finally 块已经中止了方法，系统将不会跳回去执行 try 块、except 块里的任何代码。

尽量避免在 finally 块里使用 return 或 raise 等导致方法中止的语句，否则可能出现一些很奇怪的情况。

10.8 Python raise 用法（超级详细，看了无师自通）

在前面章节的学习中，遗留过一个问题，即是否可以在程序的指定位置手动抛出一个异常？答案是肯定的，Python 允许我们在程序中手动设置异常，使用 raise 语句即可。

读者可能会感到疑惑，即我们从来都是想方设法地让程序正常运行，为什么还要手动设置异常呢？首先要分清楚程序发生异常和程序执行错误，它们完全是两码事，程序由于错误导致的运行异常，是需要程序员想办法解决的；但还有一些异常，是程序正常运行的结果，比如用 raise 手动引发的异常。

raise 语句的基本语法格式为：

```
raise [exceptionName [(reason)]]
```

其中，用 [] 括起来的为可选参数，其作用是指定抛出的异常名称，以及异常信息的相关描述。如果可选参数全部省略，则 raise 会把当前错误原样抛出；如果仅省略 (reason)，则在抛出异常时，将不附带任何的异常描述信息。

也就是说，raise 语句有如下三种常用的用法：

1. raise：单独一个 raise。该语句引发当前上下文中捕获的异常（比如在 except 块中），或默认引发 RuntimeError 异常。
2. raise 异常类名称：raise 后带一个异常类名称，表示引发执行类型的异常。
3. raise 异常类名称(描述信息)：在引发指定类型的异常的同时，附带异常的描述信息。

想了解一下常用的异常类名称，可以阅读《Python 常见异常类型》一节。
显然，每次执行 raise 语句，都只能引发一次执行的异常。首先，我们来测试一下以上 3 种 raise 的用法：

```
>>> raise
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise
RuntimeError: No active exception to reraise
>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError
>>> raise ZeroDivisionError("除数不能为零")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise ZeroDivisionError("除数不能为零")
ZeroDivisionError: 除数不能为零
```

当然，我们手动让程序引发异常，很多时候并不是为了让其崩溃。事实上，raise 语句引发的异常通常用 try except（else finally）异常处理结构来捕获并进行处理。例如：

```
try:
    a = input("输入一个数：")
    #判断用户输入的是否为数字
    if(not a.isdigit()):
        raise ValueError("a 必须是数字")
except ValueError as e:
```

```
7.      print("引发异常：",repr(e))
```

程序运行结果为：

```
输入一个数：a
引发异常： ValueError('a 必须是数字',)
```

可以看到，当用户输入的不是数字时，程序会进入 if 判断语句，并执行 raise 引发 ValueError 异常。但由于其位于 try 块中，因为 raise 抛出的异常会被 try 捕获，并由 except 块进行处理。

因此，虽然程序中使用了 raise 语句引发异常，但程序的执行是正常的，手动抛出的异常并不会导致程序崩溃。

raise 不需要参数

正如前面所看到的，在使用 raise 语句时可以不带参数，例如：

```
1.  try:
2.      a = input("输入一个数：")
3.      if(not a.isdigit()):
4.          raise ValueError("a 必须是数字")
5.  except ValueError as e:
6.      print("引发异常：",repr(e))
7.      raise
```

程序执行结果为：

```
输入一个数：a
引发异常： ValueError('a 必须是数字',)
Traceback (most recent call last):
  File "D:\python3.6\1.py", line 4, in <module>
    raise ValueError("a 必须是数字")
ValueError: a 必须是数字
```

这里重点关注位于 except 块中的 raise，由于在其之前我们已经手动引发了 ValueError 异常，因此这里当再使用 raise 语句时，它会再次引发一次。

当在没有引发过异常的程序使用无参的 raise 语句时，它默认引发的是 RuntimeError 异常。例如：

```
1.  try:
2.      a = input("输入一个数：")
3.      if(not a.isdigit()):
4.          raise
5.  except RuntimeError as e:
6.      print("引发异常：",repr(e))
```

程序执行结果为：

```
输入一个数：a
引发异常： RuntimeError('No active exception to reraise',)
```

10.9 Python sys.exc_info()方法：获取异常信息

在实际调试程序的过程中，有时只获得异常的类型是远远不够的，还需要借助更详细的异常信息才能解决问题。

捕获异常时，有 2 种方式可获得更多的异常信息，分别是：

1. 使用 sys 模块中的 exc_info 方法；
2. 使用 traceback 模块中的相关函数。

本节首先介绍如何使用 sys 模块中的 exc_info() 方法获得更多的异常信息。

> 有关 sys 模块更详细的介绍，可阅读《Python sys 模块》。

模块 sys 中，有两个方法可以返回异常的全部信息，分别是 exc_info() 和 last_traceback()，这两个函数有相同的功能和用法，本节仅以 exc_info() 方法为例。

exc_info() 方法会将当前的异常信息以元组的形式返回，该元组中包含 3 个元素，分别为 type、value 和 traceback，它们的含义分别是：

- type：异常类型的名称，它是 BaseException 的子类（有关 Python 异常类，可阅读《Python 常见异常类型》一节）
- value：捕获到的异常实例。
- traceback：是一个 traceback 对象。

举个例子：

```
#使用 sys 模块之前，需使用 import 引入
import sys
try:
    x = int(input("请输入一个被除数："))
    print("30 除以",x,"等于",30/x)
except:
    print(sys.exc_info())
    print("其他异常...")
```

当输入 0 时，程序运行结果为：

```
请输入一个被除数：0
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',), <traceback object at 0x000001FCF638DD48>)
其他异常...
```

输出结果中，第 2 行是抛出异常的全部信息，这是一个元组，有 3 个元素，第一个元素是一个 ZeroDivisionError 类；第 2 个元素是异常类型 ZeroDivisionError 类的一个实例；第 3 个元素为一个 traceback 对象。其中，通过前 2 个元素可以看出抛出的异常类型以及描述信息，对于第 3 个元素，是一个 traceback 对象，无法直接看出有关异常的信息，还需要对其做进一步处理。

要查看 traceback 对象包含的内容，需要先引进 traceback 模块，然后调用 traceback 模块中的 print_tb 方法，并将 sys.exc_info() 输出的 traceback 对象作为参数参入。例如：

```
#使用 sys 模块之前，需使用 import 引入
```

```
import sys
#引入 traceback 模块
import traceback
try:
    x = int(input("请输入一个被除数："))
    print("30 除以",x,"等于",30/x)
except:
    #print(sys.exc_info())
    traceback.print_tb(sys.exc_info()[2])
    print("其他异常...")
```

输入 0，程序运行结果为：

```
请输入一个被除数：0
  File "C:\Users\mengma\Desktop\demo.py", line 7, in <module>
    print("30 除以",x,"等于",30/x)
其他异常...
```

可以看到，输出信息中包含了更多的异常信息，包括文件名、抛出异常的代码所在的行数、抛出异常的具体代码。

print_tb 方法也仅是 traceback 模块众多方法中的一个，有关 traceback 模块如何获取更多异常信息，后续章节会做详细介绍。

10.10 Python traceback 模块：获取异常信息

除了使用 sys.exc_info() 方法获取更多的异常信息之外，还可以使用 traceback 模块，该模块可以用来查看异常的传播轨迹，追踪异常触发的源头。

下面示例显示了如何显示异常传播轨迹：

```
1.  class SelfException(Exception):
2.      pass
3.  def main():
4.      firstMethod()
5.  def firstMethod():
6.      secondMethod()
7.  def secondMethod():
8.      thirdMethod()
9.  def thirdMethod():
10.     raise SelfException("自定义异常信息")
11. main()
```

上面程序中 main() 函数调用 firstMethod()，firstMethod() 调用 secondMethod()，secondMethod() 调用 thirdMethod()，thirdMethod() 直接引发一个 SelfException 异常。运行上面程序，将会看到如下所示的结果：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 11, in <module>
    main()
  File "C:\Users\mengma\Desktop\1.py", line 4, in main              <--mian 函数
    firstMethod()
  File "C:\Users\mengma\Desktop\1.py", line 6, in firstMethod       <--第三个
    secondMethod()
  File "C:\Users\mengma\Desktop\1.py", line 8, in secondMethod     <--第二个
    thirdMethod()
  File "C:\Users\mengma\Desktop\1.py", line 10, in thirdMethod      <--异常源头
    raise SelfException("自定义异常信息")
SelfException: 自定义异常信息
```

从输出结果可以看出，异常从 thirdMethod() 函数开始触发，传到 secondMethod() 函数，再传到 firstMethod() 函数，最后传到 main() 函数，在 main() 函数止，这个过程就是整个异常的传播轨迹。

在实际应用程序的开发中，大多数复杂操作都会被分解成一系列函数或方法调用。这是因为，为了具有更好的可重用性，会将每个可重用的代码单元定义成函数或方法，将复杂任务逐渐分解为更易管理的小型子任务。由于一个大的业务功能需要由多个函数或方法来共同实现，在最终编程模型中，很多对象将通过一系列函数或方法调用来实现通信，执行任务。

所以，当应用程序运行时，经常会发生一系列函数或方法调用，从而形成“函数调用战”。异常的传播则相反，只要异常没有被完全捕获（包括异常没有被捕获，或者异常被处理后重新引发了新异常），异常就从发生异常的函数或方法逐渐向外传播，首先传给该函数或方法的调用者，该函数或方法的调用者再传给其调用者，直至最后传到 Python 解释器，此时 Python 解释器会中止该程序，并打印异常的传播轨迹信息。

很多初学者一看到输出结果所示的异常提示信息，就会惊慌失措，他们以为程序出现了很多严重的错误，其实只有一个错误，系统提示那么多行信息，只不过是显示异常依次触发的轨迹。

其实，上面程序的运算结果显示的异常传播轨迹信息非常清晰，它记录了应用程序中执行停止的各个点。最后一行信息详细显示了异常的类型和异常的详细消息。从这一行向上，逐个记录了异常发生源头、异常依次传播所经过的轨迹，并标明异常发生在哪个文件、哪一行、哪个函数处。

使用 traceback 模块查看异常传播轨迹，首先需要将 traceback 模块引入，该模块提供了如下两个常用方法：

- traceback.print_exc()：将异常传播轨迹信息输出到控制台或指定文件中。
- format_exc()：将异常传播轨迹信息转换成字符串。

可能有读者好奇，从上面方法看不出它们到底处理哪个异常的传播轨迹信息。实际上我们常用的 print_exc() 是 print_exc([limit[, file]]) 省略了 limit、file 两个参数的形式。而 print_exc([limit[, file]]) 的完整形式是 `print_exception(etype, value, tb[,limit[, file]])`，在完整形式中，前面三个参数用于分别指定异常的如下信息：

- etype：指定异常类型；
- value：指定异常值；
- tb：指定异常的 traceback 信息；

当程序处于 except 块中时，该 except 块所捕获的异常信息可通过 sys 对象来获取，其中 sys.exc_type、sys.exc_value、sys.exc_traceback 就代表当前 except 块内的异常类型、异常值和异常传播轨迹。

简单来说， print_exc([limit[, file]]) 相当于如下形式：

```
print_exception(sys.exc_etype, sys.exc_value, sys.exc_tb[, limit[, file]])
```

也就是说，使用 print_exc([limit[, file]]) 会自动处理当前 except 块所捕获的异常。该方法还涉及两个参数：

1. limit：用于限制显示异常传播的层数，比如函数 A 调用函数 B，函数 B 发生了异常，如果指定 limit=1，则只显示函数 A 里面发生的异常。如果不设置 limit 参数，则默认全部显示。
2. file：指定将异常传播轨迹信息输出到指定文件中。如果不指定该参数，则默认输出到控制台。

借助于 traceback 模块的帮助，我们可以使用 except 块捕获异常，并在其中打印异常传播信息，包括把它输出到文件中。例如如下程序：

```
# 导入 trackback 模块
import traceback
class SelfException(Exception): pass

def main():
    firstMethod()
def firstMethod():
    secondMethod()
def secondMethod():
    thirdMethod()
```

```
def thirdMethod():
    raise SelfException("自定义异常信息")
try:
    main()
except:
    # 捕捉异常，并将异常传播信息输出控制台
    traceback.print_exc()
    # 捕捉异常，并将异常传播信息输出指定文件中
    traceback.print_exc(file=open('log.txt', 'a'))
```

上面程序第一行先导入了 traceback 模块，接下来程序使用 except 捕获程序的异常，并使用 traceback 的 print_exc() 方法输出异常传播信息，分别将它输出到控制台和指定文件中。

运行上面程序，同样可以看到在控制台输出异常传播信息，而且在程序目录下生成了一个 log.txt 文件，该文件中同样记录了异常传播信息。

10.11 Python 自定义异常类及用法

前面的例子里充斥了很多 Python 内置的异常类型，读者也许会问，我可以创建自己的异常类型吗？

答案是肯定的，Python 允许用户自定义异常类型。实际开发中，有时候系统提供的异常类型不能满足开发的需求。这时就可以创建一个新的异常类来拥有自己的异常。

其实，在前面章节中，已经涉及到了异常类的创建，例如：

```
1.  class SelfExceptionError(Exception):
2.      pass
3.  try:
4.      raise SelfExceptionError()
5.  except SelfExceptionError as err:
6.      print("捕捉到自定义异常")
```

运行结果为：

```
捕捉到自定义异常
```

可以看到，此程序中就自定义了一个名为 SelfExceptionError 的异常类，只不过该类是一个空类。

由于大多数 Python 内置异常的名字都以 "Error" 结尾，所以实际命名时尽量跟标准的异常命名一样。
需要注意的是，自定义一个异常类，通常应继承自 Exception 类（直接继承），当然也可以继承自那些本身就是从 Exception 继承而来的类（间接继承 Exception）。

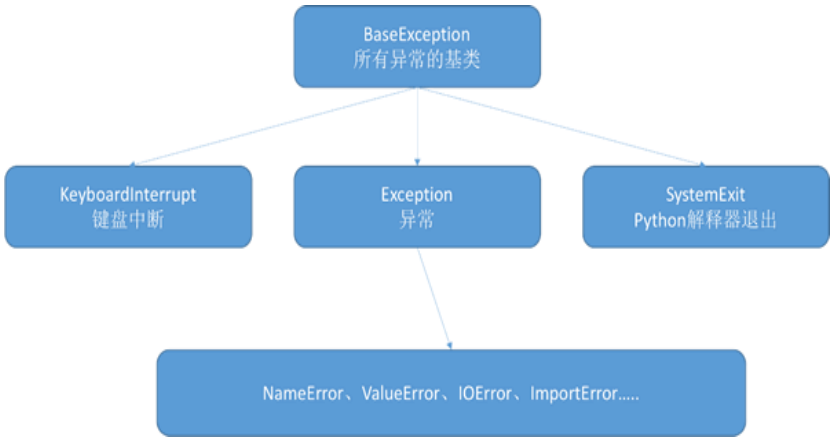


图 1 Python 异常类继承图

注意，虽然所有类同时继承自 BaseException，但它是为系统退出异常而保留的，假如直接继承 BaseException，可能会导致自定义异常不会被捕获，而是直接发送信号退出程序运行，脱离了我们自定义异常类的初衷。

另外，系统自带的异常只要触发会自动抛出（比如 NameError、ValueError 等），但用户自定义的异常需要用户自己决定什么时候抛出。也就是说，自定义的异常需要使用 raise 手动抛出。

下面也是自定义的异常类，和上面的异常类相比，其内部实现了 __init__() 方法和 __str__() 方法：

```
1.  class InputError(Exception):
2.      '''当输出有误时，抛出此异常'''
3.      #自定义异常类型的初始化
```

```
    def __init__(self, value):
        self.value = value
    # 返回异常类对象的说明信息
    def __str__(self):
        return ("{} is invalid input".format(repr(self.value)))

try:
    raise InputError(1) # 抛出 MyInputError 这个异常
except InputError as err:
    print('error: {}'.format(err))
```

运行结果为：

```
error: 1 is invalid input
```

注意，只要自定义的类继承自 Exception，则该类就是一个异常类，至于此类中包含的内容，并没有做任何规定。

10.12 Python 异常机制使用细则，正确使用 Python 异常处理机制（入门必读）

前面介绍了使用异常处理的优势、便捷之处，本节将进一步从程序性能优化、结构优化的角度给出异常处理的一般规则。

成功的异常处理应该实现如下 4 个目标：

1. 使程序代码混乱最小化。
2. 捕获并保留诊断信息。
3. 通知合适的人员。
4. 采用合适的方式结束异常活动。

下面介绍达到这些效果的基本准则。

不要过度使用异常

不可否认，Python 的异常机制确实方便，但滥用异常机制也会带来一些负面影响。过度使用异常主要表现在两个方面：

1. 把异常和普通错误混淆在一起，不再编写任何错误处理代码，而是以简单地引发异常来代苦所有的错误处理。
2. 使用异常处理来代替流程控制。

熟悉了异常使用方法后，程序员可能不再愿意编写烦琐的错误处理代码，而是简单地引发异常。实际上这样做是不对的，对于完全己知的错误和普通的错误，应该编写处理这种错误的代码，增加程序的健壮性。只有对于外部的、不能确定和预知的运行时错误才使用异常。

对比前面五子棋游戏中，处理用户输入坐标点己有棋子的两种方式。如果用户试图下棋的坐标点己有棋子：

```
#如果要下棋的点不为空
if board[int(y_str) - 1) [int(x_str) - 1] !="十" :
    inputStr = input ("您输入的坐标点己有棋子了，请重新输入\n")
    continue
```

上面这种处理方式检测到用户试图下棋的坐标点己经有棋子，立即打印一条提示语句，并重新开始下一次循环。这种处理方式简洁明了、逻辑清晰，程序的运行效率也很好程序进入 if 块后，即结束了本次循环。

如果将上面的处理机制改为如下方式：

```
#如果要下棋的点不为空
if board[int(y_str) - 1) [int(x_str) - 1) != "十":
    #引发默认的 RuntimeError 异常
    raise
```

上面这种处理方式没有提供有效的错误处理代码，当程序检测到用户试图下棋的坐标点己经有棋子时，并没有提供相应的处理，而是简单地引发一个异常。这种处理方式虽然简单，但 Python 解释器接收到这个异常后，还需要进入相应的 except 块来捕获该异常，所以运行效率要差一些。而且用户下棋重复这个错误完全是可预料的，所以程序完全可以针对该错误提供相应的处理，而不是引发异常。

> 必须指出，异常处理机制的初衷是将不可预期异常的处理代码和正常的业务逻辑处理代码分离，因此绝不要使用异常处理来代替正常的业务逻辑判断。

另外，异常机制的效率比正常的流程控制效率差，所以不要使用异常处理来代替正常的程序流程控制。例如，对于如下代码：

```
#定义一个字符串列表
my_list =["Hello", "Python", "Spring"]
#使用异常处理来遍历 arr 数组的每个元素
try:
    i = 0
    while True:
    print (my_list [i])
    i += 1
except:
    pass
```

运行上面程序确实可以实现遍历 my_list 列表的功能，但这种写法可读性较差，而且运行效率也不高。程序完全有能力避免产生 indexError 异常，程序“故意”制造这种异常，然后使用 except 块去捕获该异常，这是不应该的。将程序改为如下形式肯定要好得多：

```
i = 0
while i < len(my_list):
    print(my_list[i])
    i += 1
```

注意，异常只应该用于处理非正常的情况，不要使用异常处理来代替正常的流程控制。对于一些完全可预知，而且处理方式清楚的错误，程序应该提供相应的错误处理代码，而不是将其笼统地称为异常。

不要使用过于庞大的 try 块

很多初学异常机制的读者喜欢在 try 块里放置大量的代码，这看上去很“简单”，但这种“简单”只是一种假象，只是在编写程序时看上去比较简单。但因为 try 块里的代码过于庞大，业务过于复杂，就会造成 try 块中出现异常的可能性大大增加，从而导致分析异常原因的难度也大大增加。

而且当时块过于庞大时，就难免在 try 块后紧跟大量的 except 块才可以针对不同的异常提供不同的处理逻辑。在同一个 try 块后紧跟大量的 except 块则需要分析它们之间的逻辑关系，反而增加了编程复杂度。

正确的做法是，把大块的 try 块分割成多个可能出现异常的程序段落，并把它们放在单独的 try 块中，从而分别捕获并处理异常。

不要忽略捕获到的异常

不要忽略异常！既然已捕获到异常，那么 except 块理应做些有用的事情，及处理并修复异常。except 块整个为空，或者仅仅打印简单的异常信息都是不妥的！

except 块为空就是假装不知道甚至瞒天过海，这是最可怕的事情，程序出了错误，所有人都看不到任何异常，但整个应用可能已经彻底坏了。仅在 except 块里打印异常传播信息稍微好一点，但仅仅比空白多了几行异常信息。通常建议对异常采取适当措施，比如：

- 处理异常。对异常进行合适的修复，然后绕过异常发生的地方继续运行；或者用别的数据进行计算，以代替期望的方法返回值；或者提示用户重新操作……总之，程序应该尽量修复异常，使程序能恢复运行。
- 重新引发新异常。把在当前运行环境下能做的事情尽量做完，然后进行异常转译，把异常包装成当前层的异常，重新传给上层调用者。
- 在合适的层处理异常。如果当前层不清楚如何处理异常，就不要在当前层使用 except 语句来捕获该异常，让上层调用者来负责处理该异常。

10.13 Python logging 模块用法快速攻略

无论使用哪种编程语言，最常用的调试代码的方式是：使用输出语句（比如 C 语言中使用 printf，Python 中使用 print() 函数）输出程序运行过程中一些关键的变量的值，查看它们的值是否正确，从而找到出错的地方。这种调试方法最大的缺点是，当找到问题所在之后，需要再将用于调试的输出语句删掉。

在 Python 中，有一种比频繁使用 print() 调试程序更简便的方法，就是使用 logging 模块，该模块可以很容易地创建自定义的消息记录，这些日志消息将描述程序执行何时到达日志函数调用，并列出指定的任何变量当时的值。

启用 logging 模块很简单，直接将下面的代码复制到程序开头：

```
1.  import logging
2.  logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

读者不需要关心这两行代码的具体工作原理，但基本上，当 Python 记录一个事件的日志时，它会创建一个 LogRecord 对象，保存关于该事件的信息。

假如我们编写了如下一个函数，其设计的初衷是用来计算一个数的阶乘，但该函数有些问题，需要调试：

```
1.  import logging
2.  logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
3.  logging.debug('Start of program')
4.  def factorial(n):
5.      logging.debug('Start of factorial(%s%%)' % (n))
6.      total = 1
7.      for i in range(n + 1):
8.          total *= i
9.          logging.debug('i is ' + str(i) + ', total is ' + str(total))
10.     logging.debug('End of factorial(%s%%)' % (n))
11.     return total
12. print(factorial(5))
13. logging.debug('End of program')
```

运行结果为：

```
2019-09-11 14:14:56,928 - DEBUG - Start of program
2019-09-11 14:14:56,945 - DEBUG - Start of factorial(5%)
2019-09-11 14:14:56,959 - DEBUG - i is 0, total is 0
2019-09-11 14:14:56,967 - DEBUG - i is 1, total is 0
2019-09-11 14:14:56,979 - DEBUG - i is 2, total is 0
2019-09-11 14:14:56,991 - DEBUG - i is 3, total is 0
2019-09-11 14:14:57,000 - DEBUG - i is 4, total is 0
2019-09-11 14:14:57,013 - DEBUG - i is 5, total is 0
2019-09-11 14:14:57,024 - DEBUG - End of factorial(5%)
0
2019-09-11 14:14:57,042 - DEBUG - End of program
```

可以看到，通过 logging.debug() 函数可以打印日志信息，这个 debug() 函数将调用 basicConfig() 打印一行信息，这行信息的格式是在 basicConfig() 函数中指定的，并且包括传递给 debug() 的消息。

分析程序的运行结果，factorial(5) 返回 0 作为 5 的阶乘的结果，这显然是不对的。for 循环应该用从 1 到 5 的数，乘以 total 的值，但 logging.debug() 显示的日志信息表明，i 变量从 0 开始，而不是 1。因为 0 乘任何数都是 0，所以接下来的迭代中，total 的值都是错的。日志消息提供了可以追踪的痕迹，帮助我们弄清楚程序运行过程哪里不对。

将代码行 for i in range（n + 1）：改为 for i in range（1，n + 1）：，再次运行程序，输出结果为：

```
2019-09-11 14:21:18,047 - DEBUG - Start of program
2019-09-11 14:21:18,067 - DEBUG - Start of factorial(5%)
2019-09-11 14:21:18,072 - DEBUG - i is 1, total is 1
2019-09-11 14:21:18,082 - DEBUG - i is 2, total is 2
2019-09-11 14:21:18,087 - DEBUG - i is 3, total is 6
2019-09-11 14:21:18,093 - DEBUG - i is 4, total is 24
2019-09-11 14:21:18,101 - DEBUG - i is 5, total is 120
2019-09-11 14:21:18,106 - DEBUG - End of factorial(5%)
120
2019-09-11 14:21:18,123 - DEBUG - End of program
```

Python logging 日志级别

“日志级别”提供了一种方式，按重要性对日志消息进行分类。5 个日志级别如表 1 所示，从最不重要到最重要。利用不同的日志函数，消息可以按某个级别记入日志。

表 1 Python logging 日志级别

级别	对应的函数	描述
DEBUG	logging.debug()	最低级别，用于小细节，通常只有在诊断问题时，才会关心这些消息。
INFO	logging.info()	用于记录程序中一般事件的信息，或确认一切工作正常。
WARNING	logging.warning()	用于表示可能的问题，它不会阻止程序的工作，但将来可能会。
ERROR	logging.error()	用于记录错误，它导致程序做某事失败。
CRITICAL	logging.critical()	最高级别，用于表示致命的错误，它导致或将要导致程序完全停止工作。

日志消息将会作为一个字符串，传递给这些函数。另外，日志级别只是一种建议，归根到底还是由程序员自己来决定日志消息属于哪一种类型。

举个例子：

```
>>>import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2019-09-11 14:32:34,249 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2019-09-11 14:32:47,456 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2019-09-11 14:33:02,391 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2019-09-11 14:33:14,413 - ERROR - An error has occurred.
```

```
>>> logging.critical('The program is unable to recover!')
2019-09-11 14:33:24,071 - CRITICAL - The program is unable to recover!
```

日志级别的好处在于，我们可以改变想看到的日志消息的优先级。比如说，向 basicConfig() 函数传入 logging.DEBUG 作为 level 关键字参数，这将显示所有级别为 DEBUG 的日志消息。当开发了更多的程序后，我们可能只对错误感兴趣，在这种情况下，可以将 basicConfig() 的 level 参数设置为 logging.ERROR，这将只显示 ERROR 和 CRITICAL 消息，跳过 DEBUG、INFO 和 WARNING 消息。

Python logging 禁用日志

在调试完程序后，可能并不希望所有这些日志消息出现在屏幕上，这时就可以使用 logging.disable() 函数禁用这些日志消息，从而不必进入到程序中，手工删除所有的日志调用。

logging.disable() 函数的用法是，向其传入一个日志级别，它会禁止该级别以及更低级别的所有日志消息。因此，如果想要禁用所有日志，只要在程序中添加 logging.disable(logging.CRITICAL) 即可，例如：

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s - %(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2019-09-11 14:42:14,833 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

因为 logging.disable() 将禁用它之后的所有消息，所以可以将其添加到程序中更接近 import logging 的位置，这样更容易找到它，方便根据需要注释掉它，或取消注释，从而启用或禁用日志消息。

将日志消息输出到文件中

虽然日志消息很有用，但它们可能塞满屏幕，让你很难读到程序的输出。考虑到这种情况，可以将日志信息写入到文件，既能使屏幕保持干净，又能保存信息，一举两得。

将日志消息输出到文件中的实现方法很简单，只需要设置 logging.basicConfig() 函数中的 filename 关键字参数即可，例如：

```
>>> import logging
>>> logging.basicConfig(filename='demo.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
```

- 此程序中，将日志消息存储到了 demo.txt 文件中，该文件就位于运行的程序文件所在的目录。

10.14 Python IDLE 调试程序详解

在程序开发过程中，免不了会出现一些错误，既有语法方面的，也有逻辑方面的。语法方面的相对比较好检测，因为当程序中有语法错误时，程序运行会直接停止，同时 Python 解释器会给出错误提示。而对于逻辑错误，可能并不太容易发现，因为程序本身运行没有问题，只是运行结果是错误的。

当遇到程序有逻辑错误时，最好的解决方法就是对程序进行调试，即通过观察程序的运行过程，以及运行过程中变量（局部变量和全局变量）值的变化，可以快速找到引起运行结果异常的根本原因，从而解决逻辑错误。

掌握一定的程序调试方法，是每一名合适的程序员的必备技能。多数的集成开发工具都提供了程序调试功能，本教程中使用的 IDLE 也不例外。本节将给大家演示如何使用 IDLE 调试 Python 程序。

在保证程序没有语法错误的前提下，使用 IDLE 调试程序的基本步骤如下：

1. 打开 Python Shell，在主菜单上选择“Debug -> Debugger”选项，打开 Debug Control 对话框，同时 Python Shell 窗口中会显示“[DEBUG ON]”，表示已经处于调试状态，如图 1 所示：

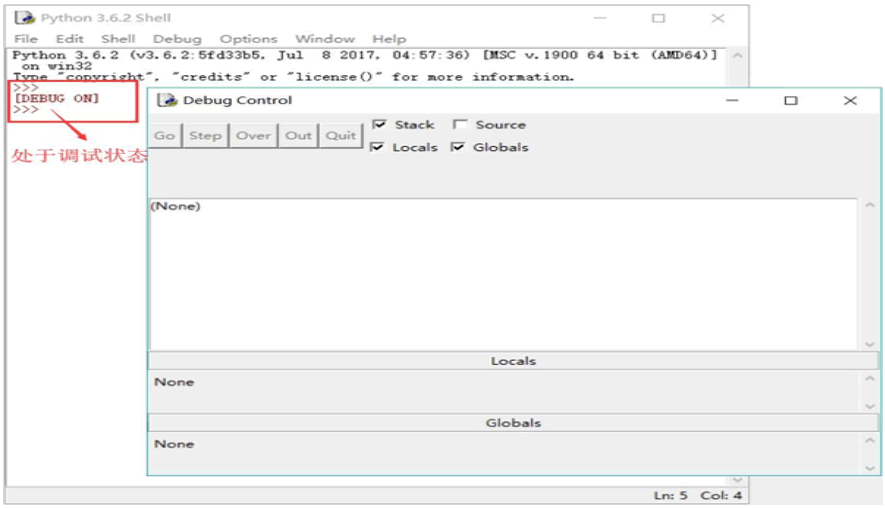


图 1 处于调试状态的 Python Shell

2. 在 Python Shell 窗口中，选择“File -> Open”菜单项，打开要调试的程序文件，并向程序中的代码添加断点，其作用是：当程序执行至断点位置时，会暂时中断执行。根据需要，程序还可以恢复执行。

> 向程序中添加断点，不能胡乱地添加，要有目的的添加。一般情况下，当想要查看某个变量运行至某处代码的值，就可以在该代码位置添加一个断点。

程序中添加断点的方法是：在想要添加断点的行上，点击鼠标右键，在弹出的快捷菜单中选择“Set BreakPoint”菜单项，添加断点的代码行，其背景会变成黄色，如图 2 所示。

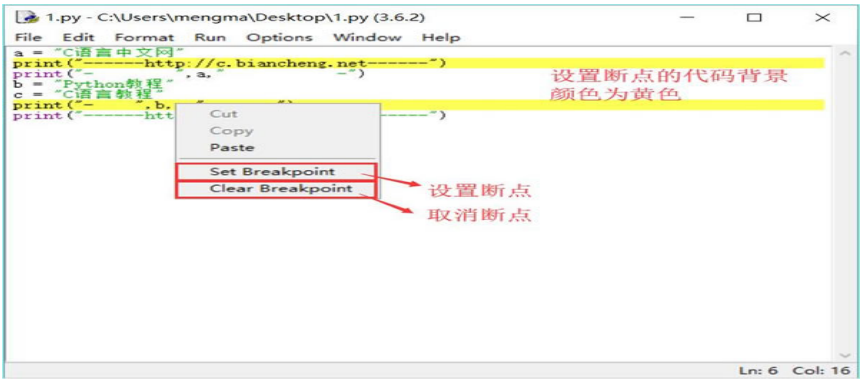


图 2 给代码添加断点

同样，如果想删除已添加的断点，可以选中已添加断点的行，然后点击鼠标右键，选择“Clear Breakpoint”。

3. 添加完断点之后，可以按 F5 快捷键，或者在打开的程序文件菜单栏中选择“Run -> Run Module”执行程序，这时 Debug Control 对话框中将显示程序的执行信息。如图 3 所示。

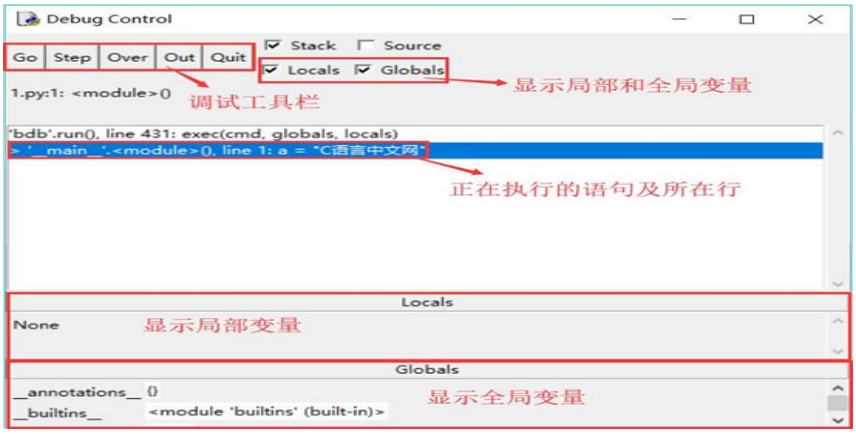


图 3 显示程序的执行信息

需要注意的是，勾选 Globals 复选框，将显示全局变量，Debug Control 默认只显示局部变量。

4. 图 3 中，调试工具栏中的 5 个按钮的作用分别是：

- Go 按钮：直接运行至下一个断点处；
- Step 按钮：用于进入要执行的函数；
- Over 按钮：表示单步执行；
- Out 按钮：表示跳出当前运行的函数；
- Quit 按钮：表示结束调试。

通过使用这 5 个按钮，可以查看程序执行过程中各个变量值的变化，直至程序运行结束。程序调试完毕后，可以关闭 Debug Control 窗口，此时在 Python Shell 窗口中将显示 “[DEBUG OFF]” ，表示已经结束调试。

10.15 Python assert 调试程序

前面章节介绍了如何使用 IDLE 自身的调试工具调试程序，除此之外，Python 还提供了 assert 语句，也可以用来调试程序。

《Python assert 断言》一节中，已经对 assert 的基本用法做了简单介绍，assert 语句的完整语法格式为：

```
assert 条件表达式 [,描述信息]
```

assert 语句的作用是：当条件表达式的值为真时，该语句什么也不做，程序正常运行；反之，若条件表达式的值为假，则 assert 会抛出 AssertionError 异常。其中，[,描述信息] 作为可选参数，用于对条件表达式可能产生的异常进行描述。

例如：

```
1.  s_age = input("请输入您的年龄:")
2.  age = int(s_age)
3.  assert 20 < age < 80 , "年龄不在 20-80 之间"
4.  print("您输入的年龄在 20 和 80 之间")
```

程序运行结果为：

```
请输入您的年龄:10
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 3, in <module>
    assert 20 < age < 80 , "年龄不在 20-80 之间"
AssertionError: 年龄不在 20-80 之间
```

通过运行结果可以看出，当 assert 中条件表达式的值为假时，程序将抛出异常，并附带异常的描述性信息，与此同时，程序立即停止执行。

通常情况下，assert 可以和 try except 异常处理语句配合使用，以前面代码为例：

```
1.  try:
2.      s_age = input("请输入您的年龄:")
3.      age = int(s_age)
4.      assert 20 < age < 80 , "年龄不在 20-80 之间"
5.      print("您输入的年龄在 20 和 80 之间")
6.  except AssertionError as e:
7.      print("输入年龄不正确",e)
```

程序运行结果为：

```
请输入您的年龄:10
输入年龄不正确 年龄不在 20-80 之间
```

通过在程序的适当位置，使用 assert 语句判断变量或表达式的值，可以起到调试代码的作用。

当在命令行模式运行 Python 程序时，传入 -O（注意是大写）参数，可以禁用程序中包含的 assert 语句。

第 11 章：Python 模块和包

11.1 什么是模块，Python 模块化编程（入门必读）

Python 提供了强大的模块支持，主要体现在，不仅 Python 标准库中包含了大量的模块（称为标准模块），还有大量的第三方模块，开发者自己也可以开发自定义模块。通过这些强大的模块可以极大地提高开发者的开发效率。

那么，模块到底指的是什么呢？模块，英文为 Modules，至于模块到底是什么，可以用一句话总结：模块就是 Python 程序。换句话说，任何 Python 程序都可以作为模块，包括在前面章节中写的所有 Python 程序，都可以作为模块。

模块可以比作一盒积木，通过它可以拼出多种主题的玩具，这与前面介绍的函数不同，一个函数仅相当于一块积木，而一个模块（.py 文件）中可以包含多个函数，也就是很多积木。模块和函数的关系如图 1 所示。

图 1 模块和函数的关系

经过前面的学习，读者已经能够将 Python 代码写到一个文件中，但随着程序功能的复杂，程序体积会不断变大，为了便于维护，通常会将其分为多个文件（模块），这样不仅可以提高代码的可维护性，还可以提高代码的可重用性。

> 代码的可重用性体现在，当编写好一个模块后，只要编程过程中需要用到该模块中的某个功能（由变量、函数、类实现），无需做重复性的编写工作，直接在程序中导入该模块即可使用该功能。

前面讲了封装，并且还介绍了很多具有封装特性的结构，比如说：

- 诸多容器，例如列表、元组、字符串、字典等，它们都是对数据的封装；
- 函数是对 Python 代码的封装；
- 类是对方法和属性的封装，也可以说是对函数和数据的封装。

本节所介绍的模块，可以理解为是对代码更高级的封装，即把能够实现某一特定功能的代码编写在同一个 .py 文件中，并将其作为一个独立的模块，这样既可以方便其它程序或脚本导入并使用，同时还能有效避免函数名和变量名发生冲突。

举个简单的例子，在某一目录下（桌面也可以）创建一个名为 hello.py 文件，其包含的代码如下：

```
1.  def say ():
2.      print("Hello,World!")
```

在同一目录下，再创建一个 say.py 文件，其包含的代码如下：

```
#通过 import 关键字，将 hello.py 模块引入此文件
import hello
hello.say()
```

运行 say.py 文件，其输出结果为：

```
Hello,World!
```

读者可能注意到，say.py 文件中使用了原本在 hello.py 文件中才有的 say() 函数，相对于 day.py 来说，hello.py 就是一个自定义的模块（有关自定义模块，后续章节会做详细讲解），我们只需要将 hellp.py 模块导入到 say.py 文件中，就可以直接在 say.py 文件中使用模块中的资源。

与此同时，当调用模块中的 say() 函数时，使用的语法格式为“模块名.函数”，这是因为，相对于 say.py 文件，hello.py 文件中的代码自成一个命名空间，因此在调用其他模块中的函数时，需要明确指明函数的出处，否则 Python 解释器将会报错。

11.2Python 导入模块，Python import 用法（超级详细）

使用 Python 进行编程时，有些功能没必须自己实现，可以借助 Python 现有的标准库或者其他人提供的第三方库。比如说，在前面章节中，我们使用了一些数学函数，例如余弦函数 cos()、绝对值函数 fabs() 等，它们位于 Python 标准库中的 math（或 cmath）模块中，只需要将此模块导入到当前程序，就可以直接拿来用。

前面章节中，已经看到使用 import 导入模块的语法，但实际上 import 还有更多详细的用法，主要有以下两种：

1. `import 模块名 1 [as 别名 1], 模块名 2 [as 别名 2]，…`：使用这种语法格式的 import 语句，会导入指定模块中的所有成员（包括变量、函数、类等）。不仅如此，当需要使用模块中的成员时，需用该模块名（或别名）作为前缀，否则 Python 解释器会报错。
2. `from 模块名 import 成员名 1 [as 别名 1]，成员名 2 [as 别名 2]，…`：使用这种语法格式的 import 语句，只会导入模块中指定的成员，而不是全部成员。同时，当程序中使用该成员时，无需附加任何前缀，直接使用成员名（或别名）即可。

注意，用 [] 括起来的部分，可以使用，也可以省略。

> 其中，第二种 import 语句也可以导入指定模块中的所有成员，即使用 form 模块名 import * ，但此方式不推荐使用，具体原因本节后续会做详细说明。

import 模块名 as 别名

下面程序使用导入整个模块的最简单语法来导入指定模块：

```
1.  # 导入 sys 整个模块
2.  import sys
3.  # 使用 sys 模块名作为前缀来访问模块中的成员
4.  print(sys.argv[0])
```

上面第 2 行代码使用最简单的方式导入了 sys 模块，因此在程序中使用 sys 模块内的成员时，必须添加模块名作为前缀。

运行上面程序，可以看到如下输出结果（sys 模块下的 argv 变量用于获取运行 Python 程序的命令行参数，其中 argv[0] 用于获取当前 Python 程序的存储路径）：

```
C:\Users\mengma\Desktop\hello.py
```

导入整个模块时，也可以为模块指定别名。例如如下程序：

```
1.  # 导入 sys 整个模块，并指定别名为 s
2.  import sys as s
3.  # 使用 s 模块别名作为前缀来访问模块中的成员
4.  print(s.argv[0])
```

第 2 行代码在导入 sys 模块时才指定了别名 s，因此在程序中使用 sys 模块内的成员时，必须添加模块别名 s 作为前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

也可以一次导入多个模块，多个模块之间用逗号隔开。例如如下程序：

```
# 导入 sys、os 两个模块
import sys,os
# 使用模块名作为前缀来访问模块中的成员
print(sys.argv[0])
# os 模块的 sep 变量代表平台上的路径分隔符
print(os.sep)
```

上面第 2 行代码一次导入了 sys 和 os 两个模块，因此程序要使用 sys、os 两个模块内的成员，只要分别使用 sys、os 模块名作为前缀即可。在 Windows 平台上运行该程序，可以看到如下输出结果（os 模块的 sep 变量代表平台上的路径分隔符）：

```
C:\Users\mengma\Desktop\hello.py
\
```

在导入多个模块的同时，也可以为模块指定别名，例如如下程序：

```
# 导入 sys、os 两个模块，并为 sys 指定别名 s，为 os 指定别名 o
import sys as s,os as o
# 使用模块别名作为前缀来访问模块中的成员
print(s.argv[0])
print(o.sep)
```

上面第 2 行代码一次导入了 sys 和 os 两个模块，并分别为它们指定别名为 s、o，因此程序可以通过 s、o 两个前缀来使用 sys、os 两个模块内的成员。在 Windows 平台上运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
\
```

from 模块名 import 成员名 as 别名

下面程序使用了 from...import 最简单的语法来导入指定成员：

```
# 导入 sys 模块的 argv 成员
from sys import argv
# 使用导入成员的语法，直接使用成员名访问
print(argv[0])
```

第 2 行代码导入了 sys 模块中的 argv 成员，这样即可在程序中直接使用 argv 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

导入模块成员时，也可以为成员指定别名，例如如下程序：

```
# 导入 sys 模块的 argv 成员，并为其指定别名 v
from sys import argv as v
# 使用导入成员（并指定别名）的语法，直接使用成员的别名访问
print(v[0])
```

第 2 行代码导入了 sys 模块中的 argv 成员，并为该成员指定别名 v，这样即可在程序中通过别名 v 使用 argv 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
```

form...import 导入模块成员时，支持一次导入多个成员，例如如下程序：

```
# 导入 sys 模块的 argv,winver 成员
from sys import argv, winver
# 使用导入成员的语法，直接使用成员名访问
print(argv[0])
print(winver)
```

上面第 2 行代码导入了 sys 模块中的 argv、 winver 成员，这样即可在程序中直接使用 argv、winver 两个成员，无须使用任何前缀。运行该程序，可以看到如下输出结果（sys 的 winver 成员记录了该 Python 的版本号）：

```
C:\Users\mengma\Desktop\hello.py
3.6
```

一次导入多个模块成员时，也可指定别名，同样使用 as 关键字为成员指定别名，例如如下程序：

```
# 导入 sys 模块的 argv,winver 成员，并为其指定别名 v、wv
from sys import argv as v, winver as wv
# 使用导入成员（并指定别名）的语法，直接使用成员的别名访问
print(v[0])
print(wv)
```

上面第 2 行代码导入了 sys 模块中的 argv、winver 成员，并分别为它们指定了别名 v、wv，这样即可在程序中通过 v 和 wv 两个别名使用 argv、winver 成员，无须使用任何前缀。运行该程序，可以看到如下输出结果：

```
C:\Users\mengma\Desktop\hello.py
3.6
```

不推荐使用 from import 导入模块所有成员

在使用 from...import 语法时，可以一次导入指定模块内的所有成员（此方式不推荐），例如如下程序：

```
#导入 sys 棋块内的所有成员
from sys import *
#使用导入成员的语法，直接使用成员的别名访问
print(argv[0])
print(winver)
```

上面代码一次导入了 sys 模块中的所有成员，这样程序即可通过成员名来使用该模块内的所有成员。该程序的输出结果和前面程序的输出结果完全相同。

需要说明的是，一般不推荐使用“from 模块 import”这种语法导入指定模块内的所有成员，因为它存在潜在的风险。比如同时导入 module1 和 module2 内的所有成员，假如这两个模块内都有一个 foo() 函数，那么当在程序中执行如下代码时：

```
foo()
```

上面调用的这个 foo() 函数到底是 module1 模块中的还是 module2 模块中的？因此，这种导入指定模块内所有成员的用法是有风险的。

但如果换成如下两种导入方式：

```
import module1
import module2 as m2
```

接下来要分别调用这两个模块中的 foo() 函数就非常清晰。程序可使用如下代码：

```
#使用模块 module1 的模块名作为前缀调用 foo()函数
module1.foo()
#使用 module2 的模块别名作为前缀调用 foo()函数
m2.foo()
```

或者使用 from...import 语句也是可以的：

```
#导入 module1 中的 foo 成员，并指定其别名为 foo1
from module1 import foo as foo1
#导入 module2 中的 foo 成员，并指定其别名为 foo2
from module2 import foo as foo2
```

此时通过别名将 module1 和 module2 两个模块中的 foo 函数很好地进行了区分，接下来分别调用两个模块中 foo() 函数就很清晰：

```
foo1() #调用 module1 中的 foo()函数
foo2() #调用 module2 中的 foo()函数
```

11.3 Python 自定义模块

到目前为止，读者已经掌握了导入 Python 标准库并使用其成员（主要是函数）的方法，接下来要解决的问题是，怎样自定义一个模块呢？

前面章节中讲过，Python 模块就是 Python 程序，换句话说，只要是 Python 程序，都可以作为模块导入。例如，下面定义了一个简单的模块（编写在 demo.py 文件中）：

```
name = "Python 教程"
add = "http://c.biancheng.net/python"
print(name,add)

def say():
    print("人生苦短，我学 Python！")

class CLanguage:
    def __init__(self,name,add):
        self.name = name
        self.add = add
    def say(self):
        print(self.name,self.add)
```

可以看到，我们在 demo.py 文件中放置了变量（name 和 add）、函数（ say() ）以及一个 Clanguage 类，该文件就可以作为一个模板。

但通常情况下，为了检验模板中代码的正确性，我们往往需要为其设计一段测试代码，例如：

```
say()
clangs = CLanguage("C 语言中文网","http://c.biancheng.net")
clangs.say()
```

运行 demo.py 文件，其执行结果为：

```
Python 教程 http://c.biancheng.net/python
人生苦短，我学 Python！
C 语言中文 http://c.biancheng.net
```

通过观察模板中程序的执行结果可以断定，模板文件中包含的函数以及类，是可以正常工作的。

在此基础上，我们可以新建一个 test.py 文件，并在该文件中使用 demo.py 模板文件，即使用 import 语句导入 demo.py：

```
import demo
```

注意，虽然 demo 模板文件的全称为 demo.py，但在使用 import 语句导入时，只需要使用该模板文件的名称即可。

此时，如果直接运行 test.py 文件，其执行结果为：

```
Python 教程 http://c.biancheng.net/python
人生苦短，我学 Python！
C 语言中文 http://c.biancheng.net
```

可以看到，当执行 test.py 文件时，它同样会执行 demo.py 中用来测试的程序，这显然不是我们想要的效果。正常的效果应该是，只有直接运行模板文件时，测试代码才会被执行；反之，如果是其它程序以引入的方式执行模板文件，则测试代码不应该被执行。

要实现这个效果，可以借助 Python 内置的 __name__ 变量。当直接运行一个模块时，name 变量的值为 __main__；而将模块被导入其他程序中并运行该程序时，处于模块中的 __name__ 变量的值就变成了模块名。因此，如果希望测试函数只有在直接运行模块文件时才执行，则可在调用测试函数时增加判断，即只有当 __name__ =='__main__' 时才调用测试函数。

因此，我们可以修改 demo.py 模板文件中的测试代码为：

```
if __name__ == '__main__':
    say()
    clangs = CLanguage("C 语言中文网","http://c.biancheng.net")
    clangs.say()
```

这样，当我们直接运行 demo.py 模板文件时，其执行结果不变；而运行 test.py 文件时，其执行结果为：

```
Python 教程 http://c.biancheng.net/python
```

显然，这里执行的仅是模板文件中的输出语句，测试代码并未执行。

自定义模块编写说明文档

我们知道，在定义函数或者类时，可以为其添加说明文档，以方便用户清楚的知道该函数或者类的功能。自定义模块也不例外。

为自定义模块添加说明文档，和函数或类的添加方法相同，即只需在模块开头的位置定义一个字符串即可。例如，为 demo.py 模板文件添加一个说明文档：

```
'''
demo 模块中包含以下内容：
name 字符串变量：初始值为“Python 教程”
add  字符串变量：初始值为“http://c.biancheng.net/python”
say() 函数
CLanguage 类：包含 name 和 add 属性和 say() 方法。
'''
```

在此基础上，我们可以通过模板的 __doc__ 属性，来访问模板的说明文档。例如，在 test.py 文件中添加如下代码：

```
import demo
print(demo.__doc__)
```

程序运行结果为：

```
Python 教程 http://c.biancheng.net/python

demo 模块中包含以下内容：
name 字符串变量：初始值为“Python 教程”
```

add　字符串变量：初始值为“http://c.biancheng.net/python”
say() 函数
CLanguage 类：包含 name 和 add 属性和 say() 方法。

11.4 Python __import__()函数引入模块名

前面讲过，其实模块就是一个代码文件，因此要求其文件名要符合操作系统的命名规则。

这可能会遇到一个问题，即操作系统中允许文件名中包含空格，也就是说，模块文件可以起名为类似“a b”的形式。但这和 Python 语法相矛盾，换句话说，Python 是以空格来隔离一行语句中的不同元素的，如果模块名中出现空格，就无法再使用 import 引入。

例如，我们自定义一个模块，并起名为 "demo text.py"，该模块中只包含如下输出语句：

```
1.  print("C 语言中文网")
```

如果在其他文件中，仍以 import 语句将其引入，Python 解释器会报 SyntaxError 错误：

```
>>>import demo text
SyntaxError: invalid syntax
```

不仅如此，如果模块名称以数字开头，也无法使用 import 语句正常导入。例如将 "demo text"模块文件名改为 "1demo"，并使用 import 尝试导入，也会报 SyntaxError 错误：

```
>>> import 1demo
SyntaxError: invalid syntax
```

针对以上这两种情况，如果模块中包含空格或者以数字开头，就需要使用 Python 提供的 __import__() 内置函数引入模块。例如，当模块名为"demo text" 时，引入方法如下：

```
1.  __import__("demo text")
```

运行结果为：

```
C 语言中文网
```

同样，如果模块名为 "1demo"，则引入方法如下：

```
1.  __import__("1demo")
```

运行结果为：

```
C 语言中文网
```

注意，使用 __import__() 函数引入模块名时，要以字符串的方式将模块名引入，否则会报 SyntaxError 错误。

11.5 Python __name__=='__main__'作用详解

前面章节已经对模块及其用法做了详解的介绍，相信有很多读者已经开始去尝试阅读别人的代码了（通常阅读比自己牛的人写的代码，会让自己的技术水平飞速提高）。不过，在阅读别人写的自定义模块时，经常会看到有如下这行判断语句：

```
if __name__ == '__main__':
```

这行代码的作用是什么呢？本节就详解讲解以下它的作用。

一般情况下，当我们写完自定义的模块之后，都会写一个测试代码，检验一些模块中各个功能是否能够成功运行。例如，创建一个 candf.py 文件，并编写如下代码：

```
'''
摄氏度和华氏度的相互转换模块
'''
def c2f(cel):
    fah = cel * 1.8 + 32
    return fah
def f2c(fah):
    cel = (fah - 32) / 1.8
    return cel
def test():
    print("测试数据：0 摄氏度 = %.2f 华氏度" % c2f(0))
    print("测试数据：0 华氏度 = %.2f 摄氏度" % f2c(0))
test()
```

单独运行此模块文件，可以看到如下运行结果：

```
测试数据：0 摄氏度 = 32.00 华氏度
测试数据：0 华氏度 = -17.78 摄氏度
```

在 candf.py 模块文件的基础上，在同目录下再创建一个 demo.py 文件，并编写如下代码：

```
import candf
print("32 摄氏度 = %.2f 华氏度" % candf.c2f(32))
print("99 华氏度 = %.2f 摄氏度" % candf.f2c(99))
```

运行 demo.py 文件，其运行结果如下所示：

```
测试数据：0 摄氏度 = 32.00 华氏度
测试数据：0 华氏度 = -17.78 摄氏度
32 摄氏度 = 89.60 华氏度
99 华氏度 = 37.22 摄氏度
```

可以看到，Python 解释器将模块（candf.py）中的测试代码也一块儿运行了，这并不是我们想要的结果。想要避免这种情况的关键在于，要让 Python 解释器知道，当前要运行的程度代码，是模块文件本身，还是导入模块的其它程序。

为了实现这一点，就需要使用 Python 内置的系统变量 __name__，它用于标识所在模块的模块名。例如，在 demo.py 程序文件中，添加如下代码：

```
print(__name__)
print(candf.__name__)
其运行结果为：
__main__
candf
```

可以看到，当前运行的程序，其 __name__ 的值为 __main__，而导入到当前程序中的模块，其 __name__ 值为自己的模块名。

因此，if __name__ == '__main__': 的作用是确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码；反之，如果只是作为模块导入到其他程序文件中，则此表达式将不成立，运行其它程序时，也就不会执行该判断语句中的测试代码。

11.6 Python 导入模块的 3 种方式（超级详细）

很多初学者经常遇到这样的问题，即自定义 Python 模板后，在其它文件中用 import（或 from...import）语句引入该文件时，Python 解释器同时如下错误：

ModuleNotFoundError: No module named '模块名'
意思是 Python 找不到这个模块名，这是什么原因导致的呢？要想解决这个问题，读者要先搞清楚 Python 解释器查找模块文件的过程。

通常情况下，当使用 import 语句导入模块后，Python 会按照以下顺序查找指定的模块文件：

- 在当前目录，即当前执行的程序文件所在目录下查找；
- 到 PYTHONPATH（环境变量）下的每个目录中查找；
- 到 Python 默认的安装目录下查找。

以上所有涉及到的目录，都保存在标准模块 sys 的 sys.path 变量中，通过此变量我们可以看到指定程序文件支持查找的所有目录。换句话说，如果要导入的模块没有存储在 sys.path 显示的目录中，那么导入该模块并运行程序时，Python 解释器就会抛出 ModuleNotFoundError（未找到模块）异常。

解决“Python 找不到指定模块”的方法有 3 种，分别是：

1. 向 sys.path 中临时添加模块文件存储位置的完整路径；
2. 将模块放在 sys.path 变量中已包含的模块加载路径中；
3. 设置 path 系统环境变量。

不过，在详细介绍这 3 种方式之前，为了能更方便地讲解，本节使用前面章节已建立好的 hello.py 自定义模块文件（D:\python_module\hello.py）和 say.py 程序文件（C:\Users\mengma\Desktop\say.py，位于桌面上），它们各自包含的代码如下：

```
#hello.py
def say ():
    print("Hello,World!")

#say.py
import hello
hello.say()
```

显然，hello.py 文件和 say.py 文件并不在同一目录，此时运行 say.py 文件，其运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\say.py", line 1, in <module>
    import hello
ModuleNotFoundError: No module named 'hello'
```

可以看到，Python 解释器抛出了 ModuleNotFoundError 异常。接下来，分别用以上 3 种方法解决这个问题。

导入模块方式一：临时添加模块完整路径

模块文件的存储位置，可以临时添加到 sys.path 变量中，即向 sys.path 中添加 D:\python_module（hello.py 所在目录），在 say.py 中的开头位置添加如下代码：

```
1.  import sys
2.  sys.path.append('D:\\python_module')
```

注意：在添加完整路径中，路径中的 '\' 需要使用 \ 进行转义，否则会导致语法错误。再次运行 say.py 文件，运行结果如下：

```
Hello,World!
```

可以看到，程序成功运行。在此基础上，我们在 say.py 文件中输出 sys.path 变量的值，会得到以下结果：

```
['C:\\Users\\mengma\\Desktop', 'D:\\python3.6\\Lib\\idlelib', 'D:\\python3.6\\python36.zip', 'D:\\python3.6\\DLLs',
'D:\\python3.6\\lib', 'D:\\python3.6', 'C:\\Users\\mengma\\AppData\\Roaming\\Python\\Python36\\site-packages',
'D:\\python3.6\\lib\\site-packages', 'D:\\python3.6\\lib\\site-packages\\win32', 'D:\\python3.6\\lib\\site-
packages\\win32\\lib', 'D:\\python3.6\\lib\\site-packages\\Pythonwin', 'D:\\python_module']
```

该输出信息中，红色部分就是临时添加进去的存储路径。需要注意的是，通过该方法添加的目录，只能在执行当前文件的窗口中有效，窗口关闭后即失效。

导入模块方式二：将模块保存到指定位置

如果要安装某些通用性模块，比如复数功能支持的模块、矩阵计算支持的模块、图形界面支持的模块等，这些都属于对 Python 本身进行扩展的模块，这种模块应该直接安装在 Python 内部，以便被所有程序共享，此时就可借助于 Python 默认的模块加载路径。

Python 程序默认的模块加载路径保存在 sys.path 变量中，因此，我们可以在 say.py 程序文件中先看看 sys.path 中保存的默认加载路径，向 say.py 文件中输出 sys.path 的值，如下所示：

```
['C:\\Users\\mengma\\Desktop', 'D:\\python3.6\\Lib\\idlelib', 'D:\\python3.6\\python36.zip', 'D:\\python3.6\\DLLs',
'D:\\python3.6\\lib', 'D:\\python3.6', 'C:\\Users\\mengma\\AppData\\Roaming\\Python\\Python36\\site-
packages', 'D:\\python3.6\\lib\\site-packages', 'D:\\python3.6\\lib\\site-packages\\win32', 'D:\\python3.6\\lib\\site-
packages\\win32\\lib', 'D:\\python3.6\\lib\\site-packages\\Pythonwin']
```

上面的运行结果中，列出的所有路径都是 Python 默认的模块加载路径，但通常来说，我们默认将 Python 的扩展模块添加在 `lib\site-packages` 路径下，它专门用于存放 Python 的扩展模块和包。

所以，我们可以直接将我们已编写好的 hello.py 文件添加到 `lib\site-packages` 路径下，就相当于为 Python 扩展了一个 hello 模块，这样任何 Python 程序都可使用该模块。

移动工作完成之后，再次运行 say.py 文件，可以看到成功运行的结果：

```
Hello,World!
```

导入模块方式三：设置环境变量

PYTHONPATH 环境变量（简称 path 变量）的值是很多路径组成的集合，Python 解释器会按照 path 包含的路径进行一次搜索，直到找到指定要加载的模块。当然，如果最终依旧没有找到，则 Python 就报 ModuleNotFoundError 异常。

由于不同平台，设置 path 环境变量的设置流程不尽相同，因此接下来就使用最多的 Windows、Linux、Mac OS X 这 3 个平台，给读者介绍如何设置 path 环境变量。

在 Windows 平台上设置环境变量

首先，找到桌面上的“计算机”（或者我的电脑），并点击鼠标右键，单击“属性”。此时会显示“控制面板\所有控制面板项\系统”窗口，单击该窗口左边栏中的“高级系统设置”菜单，出现“系统属性”对话框，如图 1 所示。

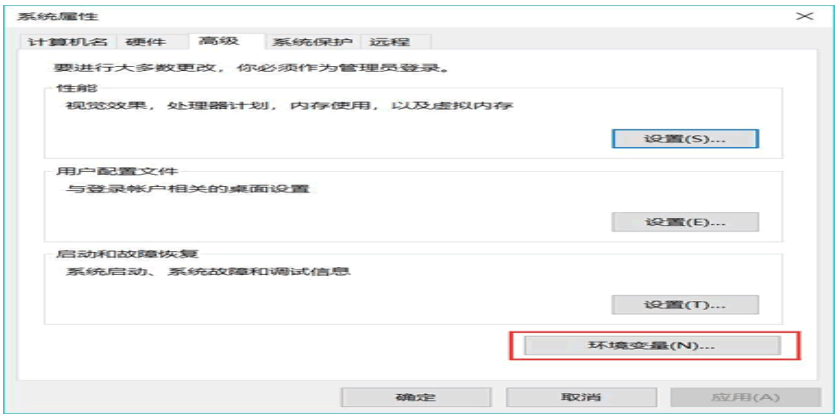


图 1 系统属性对话框

如图 1 所示，点击“环境变量”按钮，此时将弹出图 2 所示的对话框：

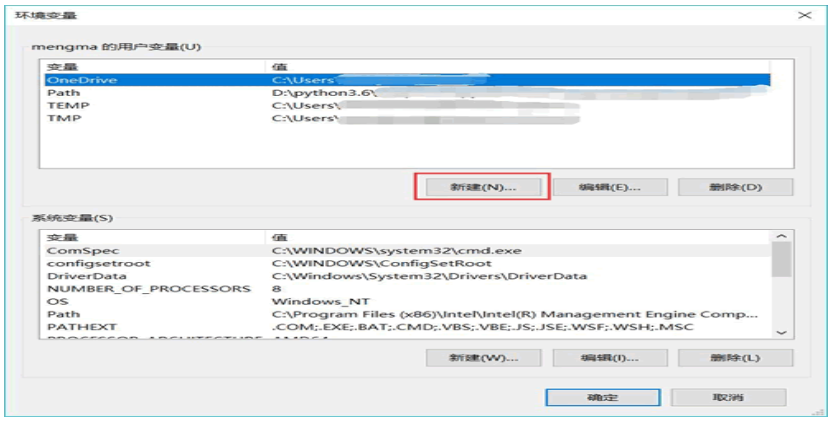


图 2 环境变量对话框

如图 2 所示，通过该对话框，就可以完成 path 环境变量的设置。需要注意的是，该对话框分为上下 2 部分，其中上面的“用户变量”部分用于设置当前用户的环境变量，下面的“系统变量”部分用于设置整个系统的环境变量。

通常情况下，建议大家设置设置用户的 path 变量即可，因为此设置仅对当前登陆系统的用户有效，而如果修改系统的 path 变量，则对所有用户有效。

对于普通用户来说，设置用户 path 变量和系统 path 变量的效果是相同的，但 Python 在使用 path 变量时，会先按照系统 path 变量的路径去查找，然后再按照用户 path 变量的路径去查找。
这里我们选择设置当前用户的 path 变量。单击用户变量中的“新建”按钮， 系统会弹出如图 3 所示的对话框。

图 3 新建 PYTHONPATH 环境变量

其中，在“变量名”文本框内输入 PYTHONPATH，表明将要建立名为 PYTHONPATH 的环境变量；在“变量值”文本框内输入 .;d:\python_ module。注意，这里其实包含了两条路径（以分号 ；作为分隔符）：

- 第一条路径为一个点（.），表示当前路径，当运行 Python 程序时，Python 将可以从当前路径加载模块；
- 第二条路径为 d:\python_ module，当运行 Python 程序时，Python 将可以从 d:\python_ module 中加载模块。

然后点击“确定” ，即成功设置 path 环境变量。此时，我们只需要将模块文件移动到和引入该模块的文件相同的目录，或者移动到 d:\python_ module 路径下，该模块就能被成功加载。

在 Linux 上设置环境变量

启动 Linux 的终端窗口，进入当前用户的 home 路径下，然后在 home 路径下输入如下命令：

```
ls - a
```

该命令将列出当前路径下所有的文件，包括隐藏文件。Linux 平台的环境变量是通过 .bash_profile 文件来设置的，使用无格式编辑器打开该文件，在该文件中添加 PYTHONPATH 环境变量。也就是为该文件增加如下一行：

```
#设置 PYTHON PATH 环境变量
PYTHONPATH=.:/home/mengma/python_module
```

Linux 与 Windows 平台不一样，多个路径之间以冒号（:）作为分隔符，因此上面一行同样设置了两条路径，点（.）代表当前路径，还有一条路径是 /home/mengma/python_module（mengma 是在 Linux 系统的登录名）。

在完成了 PYTHONPATH 变量值的设置后，在 .bash_profile 文件的最后添加导出 PYTHONPATH 变量的语句。

```
#导出 PYTHONPATH 环境变量
export PYTHONPATH
```

重新登录 Linux 平台，或者执行如下命令：

```
source.bash_profile
```

这两种方式都是为了运行该文件，使在文件中设置的 PYTHONPATH 变量值生效。

在成功设置了上面的环境变量之后，接下来只要把前面定义的模块（Python 程序）放在与当前所运行 Python 程序相同的路径中（或放在 `/home/mengma/python_module` 路径下），该模块就能被成功加载了。

在 Mac OS X 上设置环境变量

在 Mac OS X 上设置环境变量与 Linux 大致相同（因为 Mac OS X 本身也是类 UNIX 系统）。启动 Mac OS X 的终端窗口（命令行界面），进入当前用户的 home 路径下，然后在 home 路径下输入如下命令：

```
ls -a
```

该命令将列出当前路径下所有的文件，包括隐藏文件。Mac OS X 平台的环境变量也可通过，bash_profile 文件来设置，使用无格式编辑器打开该文件，在该文件中添加 PYTHONPATH 环境变量。也就是为该文件增加如下一行：

```
#设置 PYTHON PATH 环境变盘
PYTHONPATH=.:/Users/mengma/python_module
```

Mac OS X 的多个路径之间同样以冒号（:）作为分隔符，因此上面一行同样设置了两条路径：点（.）代表当前路径，还有一条路径是 `/Users/mengma/python_module`（memgma 是作者在 Mac OS X 系统的登录名）。

在完成了 PYTHONPATH 变量值的设置后，在 .bash_profile 文件的最后添加导出 PYTHONPATH 变量的语句。

```
#导出 PYTHON PATH 环境变量
export PYTHONPATH
```

重新登录 Mac OS X 系统，或者执行如下命令：

```
source.bash_profile
```

这两种方式都是为了运行该文件，使在文件中设置的 PYTHONPATH 变量值生效。

在成功设置了上面的环境变量之后，接下来只要把前面定义的模块（Python 程序）放在与当前所运行 Python 程序相同的路径中（或放在 `Users/mengma/python_module` 路径下），该模块就能被成功加载了。

11.7 Python 导入模块的本质

为了帮助大家更好地理解导入模块，下面定义一个新的模块，该模块比较简单，所以不再为之编写测试代码。该模块代码如下（编写在 fk_module.py 文件中）：

```
'一个简单的测试模块: fk_module'
print("this is fk_module")
name = 'fkit'
def hello():
    print("Hello, Python")
```

接下来，在相同的路径下定义如下程序来使用该模块：

```
import fk_module
print("===============")
# 打印 fk_module 的类型
print(type(fk_module))
print(fk_module)
```

由于前面在 PYTHONPATH 环境变量中已经添加了点（.），因此 Python 程序总可以加载相同路径下的模块。所以，上面程序可以成功导入 fk_module 模块。

运行上面程序，可以看到如下输出结果：

```
this is fk_module
=================
<class 'module'>
<module 'fk_module' from 'C:\\Users\\mengma\\Desktop\\fk_module.py'>
```

从输出结果来看，当程序导入 fk_module 时，该模块中的输出语句会在 import 时自动执行。该程序中还包含一个与模块同名的变量，该变量的类型是 module。

使用“import fk_module”导入模块的本质就是，将 fk_module.py 中的全部代码加载到内存并执行，然后将整个模块内容赋值给与模块同名的变量，该变量的类型是 module，而在该模块中定义的所有程序单元都相当于该 module 对象的成员。

下面再试试使用 `from...import` 语句来执行导入，例如使用如下程序来测试该模块：

```
from fk_module import name, hello
print("===============")
print(name)
print(hello)
# 打印 fk_module
print(fk_module)
```

运行上面程序，可以看到如下输出结果：

```
this is fk_module
==================
fkit
<function hello at 0x0000000001E7BAE8>
Traceback (most recent call last):
  File "fk_module_test2.py", line 22, in <module>
print(fk_module)
NameError: name 'fk_module' is not defined
```

从上面的输出结果可以看出，即便使用 `from...import` 只导入模块中部分成员，该模块中的输出语句也会在 import 时自动执行，这说明 Python 依然会加载并执行模块中的代码。

使用“from fk_module import name, hello”导入模块中成员的本质就是将 fk_module.py 中的全部代码加载到内存并执行，然后只导入指定变量、函数等成员单元，并不会将整个模块导入，因此上面程序在输出 fk_module 时将看到错误提示：`name 'fk module' is not defined`。

在导入模块后，可以在模块文件所在目录下看到一个名为“__pycache__”的文件夹，打开该文件夹，可以看到 Python 为每个模块都生成一个 *.cpython-36.pyc 文件，比如 Python 为 fk_module 模块生成一个 fk_ module.cpython-36.pyc 文件，该文件其实是 Python 为模块编译生成的字节码，用于提升该模块的运行效率。

导入同一个模块多次，Python 只执行一次

先看一个例子，将本节开头处的程序文件修改成如下内容：

```
1.  import fk_module
2.  import fk_module
3.  print("===============")
4.  # 打印 fk_module 的类型
5.  print(type(fk_module))
6.  print(fk_module)
```

运行结果为：

```
this is fk_module
==================
<class 'module'>
<module 'fk_module' from 'C:\\Users\\mengma\\Desktop\\fk_module.py'>
```

可以看到，修改后的程序中导入了 2 次 fk_module 模块，其实完全没必要，此处导入两次只是为了说明一点：Python 很智能。虽然上面程序两次导入了 fk_module 模块，但最后运行程序，我们看到输出语句只输出一条“this is fk_module”，这说明第二次导入的 fk_module 模块并没有起作用，这就是 Python 的“智能”之处。

当程序重复导入同一个模块时，Python 只会导入一次。道理很简单，因为这些变量、函数、类等程序单元都只需要定义一次即可，何必导入多次呢？相反，如果 Python 允许导入多次，反而可能会导致严重的后果。比如程序定义了 foo 和 bar 两个模块，假如 foo 模块导入了 bar 模块，而 bar 模块又导入了 foo 模块，这似乎形成了无限循环导入，但由于 Python 只会导入一次，所以这个无限循环导入的问题完全可以避免。

11.8 Python __all__变量用法

事实上，当我们向文件导入某个模块时，导入的是该模块中那些名称不以下划线（单下划线“_”或者双下划线“__”）开头的变量、函数和类。因此，如果我们不想模块文件中的某个成员被引入到其它文件中使用，可以在其名称前添加下划线。

以前面章节中创建的 demo.py 模块文件和 test.py 文件为例（它们位于同一目录），各自包含的内容如下所示：

```
#demo.py
def say():
    print("人生苦短，我学 Python！")

def CLanguage():
    print("C 语言中文网：http://c.biancheng.net")

def disPython():
    print("Python 教程：http://c.biancheng.net/python")

#test.py
from demo import *
say()
CLanguage()
disPython()
```

执行 test.py 文件，输出结果为：

```
人生苦短，我学 Python！
C 语言中文网：http://c.biancheng.net
Python 教程：http://c.biancheng.net/python
```

在此基础上，如果 demo.py 模块中的 disPython() 函数不想让其它文件引入，则只需将其名称改为 _disPython() 或者 __disPython()。修改之后，再次执行 test.py，其输出结果为：

```
人生苦短，我学 Python！
C 语言中文网：http://c.biancheng.net
Traceback (most recent call last):
  File "C:/Users/mengma/Desktop/2.py", line 4, in <module>
    disPython()
NameError: name 'disPython' is not defined
```

显然，test.py 文件中无法使用未引入的 disPython() 函数。

Python 模块__all__变量

除此之外，还可以借助模块提供的 __all__ 变量，该变量的值是一个列表，存储的是当前模块中一些成员（变量、函数或者类）的名称。通过在模块文件中设置 __all__ 变量，当其它文件以“from 模块名 import *”的形式导入该模块时，该文件中只能使用 __all__ 列表中

指定的成员。

也就是说，只有以“from 模块名 import *”形式导入的模块，当该模块设有 __all__ 变量时，只能导入该变量指定的成员，未指定的成员是无法导入的。

举个例子，修改 demo.py 模块文件中的代码：

```
def say():
    print("人生苦短，我学 Python！")

def CLanguage():
    print("C 语言中文网：http://c.biancheng.net")

def disPython():
    print("Python 教程：http://c.biancheng.net/python")
__all__ = ["say","CLanguage"]
```

可见，__all__ 变量只包含 say() 和 CLanguage() 的函数名，不包含 disPython() 函数的名称。此时直接执行 test.py 文件，其执行结果为：

```
人生苦短，我学 Python！
C 语言中文网：http://c.biancheng.net
Traceback (most recent call last):
  File "C:/Users/mengma/Desktop/2.py", line 4, in <module>
    disPython()
NameError: name 'disPython' is not defined
```

显然，对于 test.py 文件来说，demo.py 模块中的 disPython() 函数是未引入，这样调用是非法的。

再次声明，__all__ 变量仅限于在其它文件中以“from 模块名 import *”的方式引入。也就是说，如果使用以下 2 种方式引入模块，则 __all__ 变量的设置是无效的。

1) 以“import 模块名”的形式导入模块。通过该方式导入模块后，总可以通过模块名前缀（如果为模块指定了别名，则可以使用模快的别名作为前缀）来调用模块内的所有成员（除了以下划线开头命名的成员）。

仍以 demo.py 模块文件和 test.py 文件为例，修改它们的代码如下所示：

```
#demo.py
def say():
    print("人生苦短，我学 Python！")
def CLanguage():
    print("C 语言中文网：http://c.biancheng.net")
def disPython():
    print("Python 教程：http://c.biancheng.net/python")
__all__ = ["say"]

```

```
#test.py
import demo
demo.say()
demo.CLanguage()
demo.disPython()
```

运行 test.py 文件，其输出结果为：

```
人生苦短，我学 Python！
C 语言中文网：http://c.biancheng.net
Python 教程：http://c.biancheng.net/python
```

可以看到，虽然 demo.py 模块文件中设置有 __all__ 变量，但是当以“import demo”的方式引入后，__all__ 变量将不起作用。

2) 以“from 模块名 import 成员”的形式直接导入指定成员。使用此方式导入的模块，__all__ 变量即便设置，也形同虚设。

仍以 demo.py 和 test.py 为例，修改 test.py 文件中的代码，如下所示：

```
from demo import say
from demo import CLanguage
from demo import disPython
say()
CLanguage()
disPython()
```

运行 test.py，输出结果为：

```
人生苦短，我学 Python！
C 语言中文网：http://c.biancheng.net
Python 教程：http://c.biancheng.net/python
```

11.9 Python 包（存放多个模块的文件夹）

实际开发中，一个大型的项目往往需要使用成百上千的 Python 模块，如果将这些模块都堆放在一起，势必不好管理。而且，使用模块可以有效避免变量名或函数名重名引发的冲突，但是如果模块名重复怎么办呢？因此，Python 提出了包（Package）的概念。

什么是包呢？简单理解，包就是文件夹，只不过在该文件夹下必须存在一个名为“__init__.py” 的文件。

注意，这是 Python 2.x 的规定，而在 Python 3.x 中，__init__.py 对包来说，并不是必须的。
每个包的目录下都必须建立一个 __init__.py 的模块，可以是一个空模块，可以写一些初始化代码，其作用就是告诉 Python 要将该目录当成包来处理。

注意，__init__.py 不同于其他模块文件，此模块的模块名不是 __init__，而是它所在的包名。例如，在 settings 包中的 __init__.py 文件，其模块名就是 settings。

包是一个包含多个模块的文件夹，它的本质依然是模块，因此包中也可以包含包。例如，在前面章节中，我们安装了 numpy 模块之后可以在 Lib\site-packages 安装目录下找到名为 numpy 的文件夹，它就是安装的 numpy 模块（其实就是一个包），它所包含的内容如图 1 所示。

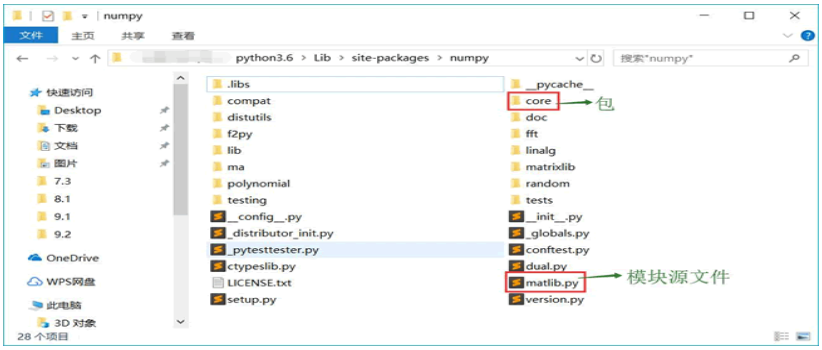


图 1 numpy 包（模块）

从图 1 可以看出，在 numpy 包（模块）中，有必须包含的 __init__.py 文件，还有 matlib.py 等模块源文件以及 core 等子包（也是模块）。这正印证了我们刚刚讲过的，包的本质依然是模块，包可以包含包。

Python 库：相比模块和包，库是一个更大的概念，例如在 Python 标准库中的每个库都有好多个包，而每个包中都有若干个模块。

11.10 Python 创建包，导入包（入门必读）

《Python 包》一节中已经提到，包其实就是文件夹，更确切的说，是一个包含“__init__.py”文件的文件夹。因此，如果我们想手动创建一个包，只需进行以下 2 步操作：

1. 新建一个文件夹，文件夹的名称就是新建包的包名；
2. 在该文件夹中，创建一个 __init__.py 文件（前后各有 2 个下划线‘_’），该文件中可以不编写任何代码。当然，也可以编写一些 Python 初始化代码，则当有其它程序文件导入包时，会自动执行该文件中的代码（本节后续会有实例）。

例如，现在我们创建一个非常简单的包，该包的名称为 my_package，可以仿照以上 2 步进行：

1. 创建一个文件夹，其名称设置为 my_package；
2. 在该文件夹中添加一个 __init__.py 文件，此文件中可以不编写任何代码。不过，这里向该文件编写如下代码：

```
'''
http://c.biancheng.net/
创建第一个 Python 包
'''
print('http://c.biancheng.net/python/')
```

可以看到，__init__.py 文件中，包含了 2 部分信息，分别是此包的说明信息和一条 print 输出语句。

由此，我们就成功创建好了一个 Python 包。

创建好包之后，我们就可以向包中添加模块（也可以添加包）。这里给 my_package 包添加 2 个模块，分别是 module1.py、module2.py，各自包含的代码分别如下所示（读者可直接复制下来）：

```
#module1.py 模块文件
def display(arc):
    print(arc)

#module2.py 模块文件
class CLanguage:
    def display(self):
        print("http://c.biancheng.net/python/")
```

现在，我们就创建好了一个具有如下文件结构的包：

```
my_package
    ├── __init__.py
    ├── module1.py
    └── module2.py
```

当然，包中还有容纳其它的包，不过这里不再演示，有兴趣的读者可以自行调整包的结构。

Python 包的导入

通过前面的学习我们知道，包其实本质上还是模块，因此导入模块的语法同样也适用于导入包。无论导入我们自定义的包，还是导入从他处下载的第三方包，导入方法可归结为以下 3 种：

1. import 包名[.模块名 [as 别名]]
2. from 包名 import 模块名 [as 别名]
3. from 包名.模块名 import 成员名 [as 别名]

用 [] 括起来的部分，是可选部分，即可以使用，也可以直接忽略。

注意，导入包的同时，会在包目录下生成一个含有 __init__.cpython-36.pyc 文件的 __pycache__ 文件夹。

1) import 包名[.模块名 [as 别名]]

以前面创建好的 my_package 包为例，导入 module1 模块并使用该模块中成员可以使用如下代码：

```
1.  import my_package.module1
2.  my_package.module1.display("http://c.biancheng.net/java/")
```

运行结果为：

```
http://c.biancheng.net/java/
```

可以看到，通过此语法格式导入包中的指定模块后，在使用该模块中的成员（变量、函数、类）时，需添加“包名.模块名”为前缀。当然，如果使用 as 给包名.模块名”起一个别名的话，就使用直接使用这个别名作为前缀使用该模块中的方法了，例如：

```
1.  import my_package.module1 as module
2.  module.display("http://c.biancheng.net/python/")
```

程序执行结果为：

```
http://c.biancheng.net/python/
```

另外，当直接导入指定包时，程序会自动执行该包所对应文件夹下的 __init__.py 文件中的代码。例如：

```
1.  import my_package
2.  my_package.module1.display("http://c.biancheng.net/linux_tutorial/")
```

直接导入包名，并不会将包中所有模块全部导入到程序中，它的作用仅仅是导入并执行包下的 __init__.py 文件，因此，运行该程序，在执行 __init__.py 文件中代码的同时，还会抛出 AttributeError 异常（访问的对象不存在）：

```
http://c.biancheng.net/python/
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 2, in <module>
    my_package.module1.display("http://c.biancheng.net/linux_tutorial/")
AttributeError: module 'my_package' has no attribute 'module1'
```

我们知道，包的本质就是模块，导入模块时，当前程序中会包含一个和模块名同名且类型为 module 的变量，导入包也是如此：

```
1.  import my_package
2.  print(my_package)
3.  print(my_package.__doc__)
4.  print(type(my_package))
```

运行结果为：

```
http://c.biancheng.net/python/
<module 'my_package' from 'C:\\Users\\mengma\\Desktop\\my_package\\__init__.py'>

http://c.biancheng.net/
创建第一个 Python 包

<class 'module'>
```

2) from 包名 import 模块名 [as 别名]

仍以导入 my_package 包中的 module1 模块为例，使用此语法格式的实现代码如下：

```
1.  from my_package import module1
2.  module1.display("http://c.biancheng.net/golang/")
```

运行结果为：

```
http://c.biancheng.net/python/
http://c.biancheng.net/golang/
```

可以看到，使用此语法格式导入包中模块后，在使用其成员时不需要带包名前缀，但需要带模块名前缀。

当然，我们也可以使用 as 为导入的指定模块定义别名，例如：

```
1.  from my_package import module1 as module
2.  module.display("http://c.biancheng.net/golang/")
```

此程序的输出结果和上面程序完全相同。

同样，既然包也是模块，那么这种语法格式自然也支持 `from 包名 import *` 这种写法，它和 import 包名 的作用一样，都只是将该包的 __init__.py 文件导入并执行。

3) from 包名.模块名 import 成员名 [as 别名]

此语法格式用于向程序中导入“包.模块”中的指定成员（变量、函数或类）。通过该方式导入的变量（函数、类），在使用时可以直接使用变量名（函数名、类名）调用，例如：

```
1.  from my_package.module1 import display
2.  display("http://c.biancheng.net/shell/")
```

运行结果为：

```
http://c.biancheng.net/python/
http://c.biancheng.net/shell/
```

当然，也可以使用 as 为导入的成员起一个别名，例如：

```
from my_package.module1 import display as dis
dis("http://c.biancheng.net/shell/")
```

该程序的运行结果和上面相同。

另外，在使用此种语法格式加载指定包的指定模块时，可以使用 * 代替成员名，表示加载该模块下的所有成员。例如：

```
from my_package.module1 import *
display("http://c.biancheng.net/python")
```

11.11 Python __init__.py 作用详解

前面章节中，已经对包的创建和导入进行了详细讲解，并提供了大量的实例，这些实例虽然可以正常运行，但存在一个通病，即为了调用包内模块的成员（变量、函数或者类），代码中包含了诸多的 import 导入语句，非常繁琐。

要解决这个问题，就需要搞明白包内 __init__.py 文件的作用和用法。

我们知道，导入包就等同于导入该包中的 __init__.py 文件，因此完全可以在 __init__.py 文件中直接编写实现模块功能的变量、函数和类，但实际上并推荐大家这样做，因为包的主要作用是包含多个模块。因此 __init__.py 文件的主要作用是导入该包内的其他模块。

也就是说，通过在 __init__.py 文件使用 import 语句将必要的模块导入，这样当向其他程序中导入此包时，就可以直接导入包名，也就是使用 `import 包名`（或 `from 包名 import *`）的形式即可。

上节中，我们已经创建好的 my_package 包，该包名包含 module1 模块、module2 模块和 __init__.py 文件。现在向 my_package 包的 __init__.py 文件中编写如下代码：

```
# 从当前包导入 module1 模块
from . import module1
#from .module1 import *
# 从当前包导入 module2 模块
#from . import module2
from .module2 import *
```

可以看到，在 __init__.py 文件中用点（.）来表示当前包的包名，除此之外，from import 语句的用法和在程序中导入包的用法完全相同。

有关 from...import 语句的用法，可阅读《Python 创建包，导入包》一节详细了解。
总的来说，__init__.py 文件是通过如下 2 种方式来导入包中模块的：

```
# 从当前包导入指定模块
from . import 模块名
# 从.模块名 导入所有成员到包中
from .模块名 import *
```

第 1 种方式用于导入当前包（模块）中的指定模块，这样即可在包中使用该模块。当在其他程序使用模块内的成员时，需要添加“包名.模块名”作为前缀，例如：

```
import my_package
my_package.module1.display("http://c.biancheng.net/python/")
```

运行结果为：

```
http://c.biancheng.net/python/
```

第 2 种方式表示从指定模块中导入所有成员，采用这种导入方式，在其他程序中使用该模块的成员时，只要使用包名作为前缀即可。例如如下程序：

```
import my_package
clangs = my_package.CLanguage()
clangs.display()
```

运行结果为：

```
http://c.biancheng.net/python/
```

11.12 Python 查看模块（变量、函数、类）方法

前面章节中，详细介绍了模块和包的创建和使用（严格来说，包本质上也是模块），有些读者可能有这样的疑问，即正确导入模块或者包之后，怎么知道该模块中具体包含哪些成员（变量、函数或者类）呢？

查看已导入模块（包）中包含的成员，本节给大家介绍 2 种方法。

查看模块成员：dir()函数

事实上，在前面章节的学习中，曾多次使用 dir() 函数。通过 dir() 函数，我们可以查看某指定模块包含的全部成员（包括变量、函数和类）。注意这里所指的全部成员，不仅包含可供我们调用的模块成员，还包含所有名称以双下划线 “__” 开头和结尾的成员，而这些 “特殊” 命名的成员，是为了在本模块中使用的，并不希望被其它文件调用。

这里以导入 string 模块为例，string 模块包含操作字符串相关的大量方法，下面通过 dir() 函数查看该模块中包含哪些成员：

```
1.  import string
2.  print(dir(string))
```

程序执行结果为：

```
['Formatter', 'Template', '_ChainMap', '_TemplateMetaclass', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_re', '_string', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords',
'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation', 'whitespace']
```

可以看到，通过 dir() 函数获取到的模块成员，不仅包含供外部文件使用的成员，还包含很多 “特殊” （名称以 2 个下划线开头和结束）的成员，列出这些成员，对我们并没有实际意义。

因此，这里给读者推荐一种可以忽略显示 dir() 函数输出的特殊成员的方法。仍以 string 模块为例：

```
1.  import string
2.  print([e for e in dir(string) if not e.startswith('_')])
```

程序执行结果为：

```
['Formatter', 'Template', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords', 'digits', 'hexdigits', 'octdigits',
'printable', 'punctuation', 'whitespace']
```

显然通过列表推导式，可在 dir() 函数输出结果的基础上，筛选出对我们有用的成员并显示出来。

查看模块成员：__all__变量

除了使用 dir() 函数之外，还可以使用 __all__ 变量，借助该变量也可以查看模块（包）内包含的所有成员。

仍以 string 模块为例，举个例子：

```
1.  import string
2.  print(string.__all__)
```

程序执行结果为：

```
['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords', 'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation',
'whitespace', 'Formatter', 'Template']
```

显然，和 dir() 函数相比，__all__ 变量在查看指定模块成员时，它不会显示模块中的特殊成员，同时还会根据成员的名称进行排序显示。

不过需要注意的是，并非所有的模块都支持使用 __all__ 变量，因此对于获取有些模块的成员，就只能使用 dir() 函数。

11.13 Python __doc__属性：查看文档

在使用 dir() 函数和 __all__ 变量的基础上，虽然我们能知晓指定模块（或包）中所有可用的成员（变量、函数和类），比如：

```
import string
print(string.__all__)
```

程序执行结果为：

```
['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'capwords', 'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation', 'whitespace', 'Formatter', 'Template']
```

但对于以上的输出结果，对于不熟悉 string 模块的用户，还是不清楚这些名称分别表示的是什么意思，更不清楚各个成员有什么功能。

针对这种情况，我们可以使用 help() 函数来获取指定成员（甚至是该模块）的帮助信息。以前面章节创建的 my_package 包为例，该包中包含 __init__.py 、module1.py 和 module2.py 这 3 个模块，它们各自包含的内容分别如下所示：

```
#***__init__.py 文件中的内容***
from my_package.module1 import *
from my_package.module2 import *

#***module1.py 中的内容***
#module1.py 模块文件
def display(arc):
    '''
    直接输出指定的参数
    '''
    print(arc)

#***module2.py 中的内容***
#module2.py 模块文件
class CLanguage:
    '''
    CLanguage 是一个类，其包含：
    display() 方法
    '''
    def display(self):
        print("http://c.biancheng.net/python/")
```

现在，我们先借助 dir() 函数，查看 my_package 包中有多少可供我们调用的成员：

```
import my_package
```

```
print([e for e in dir(my_package) if not e.startswith('_')])
```

程序输出结果为：

```
['CLanguage', 'display', 'module1', 'module2']
```

通过此输出结果可以得知，在 my_package 包中，有以上 4 个成员可供我们使用。接下来，我们使用 help() 函数来查看这些成员的具体含义（以 module1 为例）：

```
import my_package
help(my_package.module1)
```

输出结果为：

```
Help on module my_package.module1 in my_package:

NAME
    my_package.module1 - #module1.py 模块文件

FUNCTIONS
    display(arc)
        直接输出指定的参数

FILE
    c:\users\mengma\desktop\my_package\module1.py
```

通过输出结果可以得知，module1 实际上是一个模块文件，其包含 display() 函数，该函数的功能是直接输出指定的 arc 参数。同时，还显示出了该模块具体的存储位置。

当然，有兴趣的读者还可以尝试运行如下几段代码：

```
#输出 module2 成员的具体信息
help(my_package.module2)
#输出 display 成员的具体信息
help(my_package.module1.display)
#输出 CLanguage 成员的具体信息
help(my_package.module2.CLanguage)
```

值得一提的是，之所以我们可以使用 help() 函数查看具体成员的信息，是因为该成员本身就包含表示自身身份的说明文档（本质是字符串，位于该成员内部开头的位置）。前面讲过，无论是函数还是类，都可以使用 __doc__ 属性获取它们的说明文档，模块也不例外。

以 my_package 包 module1 模块中的 display() 函数为例，我们尝试用 __doc__ 变量获取其说明文档：

```
import my_package
print(my_package.module1.display.__doc__)
```

程序执行结果为：

直接输出指定的参数

其实，help() 函数底层也是借助 __doc__ 属性实现的。

那么，如果使用 help() 函数或者 __doc__ 属性，仍然无法满足我们的需求，还可以使用以下 2 种方法：

1. 调用 __file__ 属性，查看该模块或者包文件的具体存储位置，直接查看其源代码（后续章节或详细介绍）；
2. 对于非自定义的模块或者包，可以查阅 Python 库的参考文档 https://docs.python.org/3/library/index.html。

11.14 Python __file__属性：查看模块的源文件路径

前面章节提到，当指定模块（或包）没有说明文档时，仅通过 help() 函数或者 __doc__ 属性，无法有效帮助我们理解该模块（包）的具体功能。在这种情况下，我们可以通过 __file__ 属性查找该模块（或包）文件所在的具体存储位置，直接查看其源代码。

仍以前面章节创建的 my_package 包为例，下面代码尝试使用 __file__ 属性获取该包的存储路径：

```
1.  import my_package
2.  print(my_package.__file__)
```

程序输出结果为：

```
C:\Users\mengma\Desktop\my_package\__init__.py
```

注意，因为当引入 my_package 包时，其实际上执行的是 __init__.py 文件，因此这里查看 my_package 包的存储路径，输出的 __init__.py 文件的存储路径。

再以 string 模块为例：

```
1.  import string
2.  print(string.__file__)
```

程序输出结果为：

```
D:\python3.6\lib\string.py
```

由此，通过调用 __file__ 属性输出的绝对路径，我们可以很轻易地找到该模块（或包）的源文件。

注意，并不是所有模块都提供 __file__ 属性，因为并不是所有模块的实现都采用 Python 语言，有些模块采用的是其它编程语言（如 C 语言）。

11.15 Python 第三方库（模块）下载和安装（使用 pip 命令）

进行 Python 程序开发时，除了使用 Python 内置的标准模块以及我们自定义的模块之外，还有很多第三方模块可以使用，这些第三方模块可以借助 Python 官方提供的查找包页面（https://pypi.org/）找到。

使用第三方模块之前，需要先下载并安装该模块，然后就能像使用标准模块和自定义模块那样导入并使用了。因此，本节主要讲解如何下载并安装第三方模块。

下载和安装第三方模块，可以使用 Python 提供的 pip 命令实现。pip 命令的语法格式如下：

```
pip install|uninstall|list 模块名
```

其中，install、uninstall、list 是常用的命令参数，各自的含义为：

1. install：用于安装第三方模块，当 pip 使用 install 作为参数时，后面的模块名不能省略。
2. uninstall：用于卸载已经安装的第三方模块，选择 uninstall 作为参数时，后面的模块名也不能省略。
3. list：用于显示已经安装的第三方模块。

以安装 numpy 模块为例（该模块用于进行科学计算），可以在命令行窗口中输入以下代码：

```
pip install numpy
```

执行此代码，它会在线自动安装 numpy 模块。安装完成后，将显示图 1 所示的结果：

图 1 numpy 模块安装成功示意图

pip 命令会将下载完成的第三方模块，默认安装到 Python 安装目录中的 \Lib\site-packages 目录下。打开此目录，你就会发现 numpy 包，也就是 numpy 文件夹，如图 2 所示。

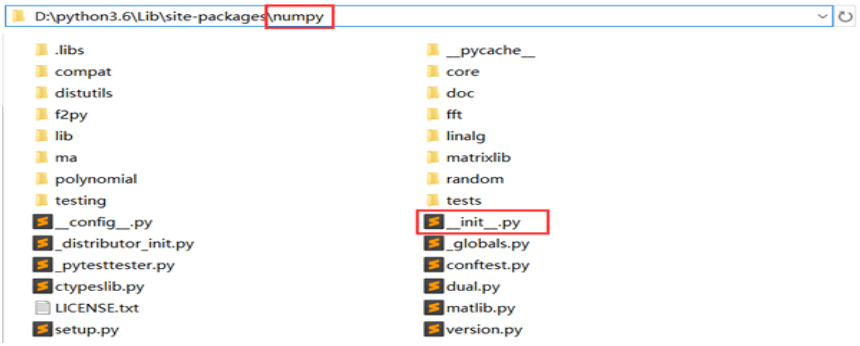


图 2 numpy 包内容示意图

前面讲过，对于向程序中导入模块，\Lib\site-packages 目录是 Python 肯定会搜索的目录，因此位于此目录的模块，可以直接使用 import 语句引入，例如：

```
#直接导入 numpy 模块即可
import numpy as nu
#用 numpy 模块中的开发函数
print(nu.sqrt(16))
```

运行结果为：

```
4.0
```

通过 pip 命令，我们可以下载并安装很多第三方模块，如果想要查看 Python 中目前有哪些模块（包括标准模块和第三方模块），可以在 IDLE 中输入以下命令：

```
help('modules')
```

在此基础上，如果只是想要查看已经安装的第三方模块，可以在使用如下命令：

```
pip list
```

提示：在大型程序中，往往需要导入很多模块，建议初学者在导入模块时，优先导入 Python 提供的标准模块，然后再导入第三方模块，最后导入自定义模块。

第 12 章：Python 文件操作（I/O）

12.1 什么是文件路径，Python 中如何书写文件路径？

当程序运行时，变量是保存数据的好方法，但变量、序列以及对象中存储的数据是暂时的，程序结束后就会丢失，如果希望程序结束后数据仍然保持，就需要将数据保存到文件中。Python 提供了内置的文件对象，以及对文件、目录进行操作的内置模块，通过这些技术可以很方便地将数据保存到文件（如文本文件等）中。

关于文件，它有两个关键属性，分别是“文件名”和“路径”。其中，文件名指的是为每个文件设定的名称，而路径则用来指明文件在计算机上的位置。例如，我的 Windows 7 笔记本上有一个文件名为 projects.docx（句点之后的部分称为文件的“扩展名”，它指出了文件的类型），它的路径在 D:\demo\exercise，也就是说，该文件位于 D 盘下 demo 文件夹中 exercise 子文件夹下。

通过文件名和路径可以分析出，project.docx 是一个 Word 文档，demo 和 exercise 都是指“文件夹”（也称为目录）。文件夹可以包含文件和其他文件夹，例如 project.docx 在 exercise 文件夹中，该文件夹又在 demo 文件夹中。

注意，路径中的 D:\ 指的是“根文件夹”，它包含了所有其他文件夹。在 Windows 中，根文件夹名为 D:\，也称为 D: 盘。在 OS X 和 Linux 中，根文件夹是 /。本教程使用的是 Windows 风格的根文件夹，如果你在 OS X 或 Linux 上输入交互式环境的例子，请用 / 代替。

另外，附加卷（诸如 DVD 驱动器或 USB 闪存驱动器），在不同的操作系统上显示也不同。在 Windows 上，它们表示为新的、带字符的根驱动器。诸如 D:\ 或 E:\。在 OS X 上，它们表示为新的文件夹，在 /Volumes 文件夹下。在 Linux 上，它们表示为新的文件夹，在 /mnt 文件夹下。同时也要注意，虽然文件夹名称和文件名在 Windows 和 OS X 上是不区分大小写的，但在 Linux 上是区分大小写的。

Windows 上的反斜杠以及 OS X 和 Linux 上的正斜杠

在 Windows 上，路径书写使用反斜杠 "\" 作为文件夹之间的分隔符。但在 OS X 和 Linux 上，使用正斜杠 "/" 作为它们的路径分隔符。如果想要程序运行在所有操作系统上，在编写 Python 脚本时，就必须处理这两种情况。

好在，用 os.path.join() 函数来做这件事很简单。如果将单个文件和路径上的文件夹名称的字符串传递给它，os.path.join() 就会返回一个文件路径的字符串，包含正确的路径分隔符。在交互式环境中输入以下代码：

```
>>> import os
>>> os.path.join('demo', 'exercise')
'demo\\exercise'
```

因为此程序是在 Windows 上运行的，所以 os.path.join('demo', 'exercise') 返回 'demo\\exercise'（请注意，反斜杠有两个，因为每个反斜杠需要由另一个反斜杠字符来转义）。如果在 OS X 或 Linux 上调用这个函数，该字符串就会是 'demo/exercise'。

不仅如此，如果需要创建带有文件名称的文件存储路径，os.path.join() 函数同样很有用。例如，下面的例子将一个文件名列表中的名称，添加到文件夹名称的末尾：

```
1.  import os
2.  myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
3.  for filename in myFiles:
4.      print(os.path.join('C:\\demo\\exercise', filename))
```

运行结果为：

```
C:\demo\exercise\accounts.txt
C:\demo\exercise\details.csv
C:\demo\exercise\invite.docx
```

12.2 Python 绝对路径和相对路径详解

在介绍绝对路径和相对路径之前，先要了解一下什么是当前工作目录。

什么是当前工作目录

每个运行在计算机上的程序，都有一个“当前工作目录”（或 cwd）。所有没有从根文件夹开始的文件名或路径，都假定在当前工作目录下。

注意，虽然文件夹是目录的更新的名称，但当前工作目录（或当前目录）是标准术语，没有当前工作文件夹这种说法。
在 Python 中，利用 os.getcwd() 函数可以取得当前工作路径的字符串，还可以利用 os.chdir() 改变它。例如，在交互式环境中输入以下代码：

```
>>> import os
>>> os.getcwd()
'C:\\Users\\mengma\\Desktop'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

可以看到，原本当前工作路径为 'C:\\Users\\mengma\\Desktop'（也就是桌面），通过 os.chdir() 函数，将其改成了 'C:\\Windows\\System32'。

需要注意的是，如果使用 os.chdir() 修改的工作目录不存在，Python 解释器会报错，例如：

```
>>> os.chdir('C:\\error')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    os.chdir('C:\\error')
FileNotFoundError: [WinError 2] 系统找不到指定的文件。: 'C:\\error'
```

了解了当前工作目录的具体含义之后，接下来介绍绝对路径和相对路径各自的含义和用法。

什么是绝对路径与相对路径

明确一个文件所在的路径，有 2 种表示方式，分别是：

- 绝对路径：总是从根文件夹开始，Window 系统中以盘符（C：、D：）作为根文件夹，而 OS X 或者 Linux 系统中以 / 作为根文件夹。
- 相对路径：指的是文件相对于当前工作目录所在的位置。例如，当前工作目录为 "C:\Windows\System32"，若文件 demo.txt 就位于这个 System32 文件夹下，则 demo.txt 的相对路径表示为 ".\demo.txt"（其中 .\ 就表示当前所在目录）。

在使用相对路径表示某文件所在的位置时，除了经常使用 .\ 表示当前所在目录之外，还会用到 ..\ 表示当前所在目录的父目录。

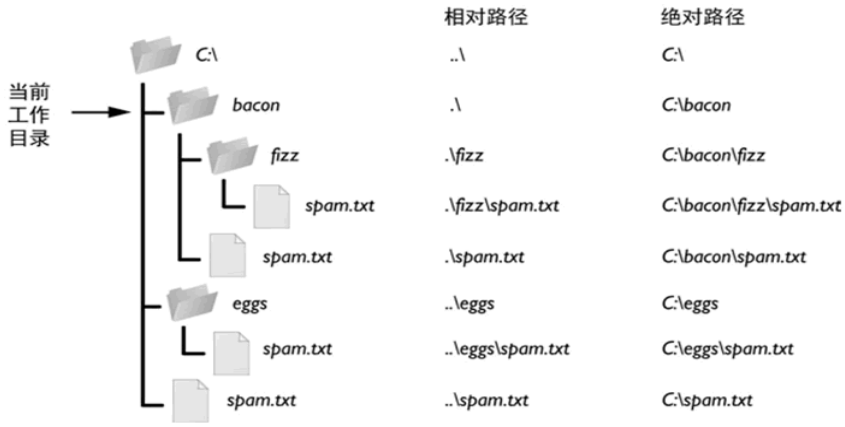


图 1 相对路径和绝对路径

以图 1 为例，如果当前工作目录设置为 C:\bacon，则这些文件夹和文件的相对路径和绝对路径，就对应为该图右侧所示的样子。

Python 处理绝对路径和相对路径

Python os.path 模块提供了一些函数，可以实现绝对路径和相对路径之间的转换，以及检查给定的路径是否为绝对路径，比如说：

- 调用 os.path.abspath(path) 将返回 path 参数的绝对路径的字符串，这是将相对路径转换为绝对路径的简便方法。
- 调用 os.path.isabs(path)，如果参数是一个绝对路径，就返回 True，如果参数是一个相对路径，就返回 False。
- 调用 os.path.relpath(path, start) 将返回从 start 路径到 path 的相对路径的字符串。如果没有提供 start，就使用当前工作目录作为开始路径。
- 调用 os.path.dirname(path) 将返回一个字符串，它包含 path 参数中最后一个斜杠之前的所有内容；调用 os.path.basename(path) 将返回一个字符串，它包含 path 参数中最后一个斜杠之后的所有内容。

在交互式环境中尝试上面提到的函数：

```
>>> os.getcwd()
'C:\\Windows\\System32'
>>> os.path.abspath('.')
'C:\\Windows\\System32'
>>> os.path.abspath('.\\Scripts')
'C:\\Windows\\System32\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

注意，由于读者的系统文件和文件夹可能与我的不同，所以读者不必完全遵照本节的例子，根据自己的系统环境对本节代码做适当调整即可。

除此之外，如果同时需要一个路径的目录名称和基本名称，就可以调用 os.path.split() 获得这两个字符串的元组，例如：

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(path)
('C:\\Windows\\System32', 'calc.exe')
```

注意，可以调用 os.path.dirname()和 os.path.basename()，将它们的返回值放在一个元组中，从而得到同样的元组。但使用 os.path.split() 无疑是很好的快捷方式。

同时，如果提供的路径不存在，许多 Python 函数就会崩溃并报错，但好在 os.path 模块提供了以下函数用于检测给定的路径是否存在，以及它是文件还是文件夹：

- 如果 path 参数所指的文件或文件夹存在，调用 os.path.exists(path) 将返回 True，否则返回 False。
- 如果 path 参数存在，并且是一个文件，调用 os.path.isfile(path) 将返回 True，否则返回 False。
- 如果 path 参数存在，并且是一个文件夹，调用 os.path.isdir(path) 将返回 True，否则返回 False。

下面是在交互式环境中尝试这些函数的结果：

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

12.3 Python 文件基本操作（入门必读）

Python 中，对文件的操作有很多种，常见的操作包括创建、删除、修改权限、读取、写入等，这些操作可大致分为以下 2 类：

1. 删除、修改权限：作用于文件本身，属于系统级操作。
2. 写入、读取：是文件最常用的操作，作用于文件的内容，属于应用级操作。

其中，对文件的系统级操作功能单一，比较容易实现，可以借助 Python 中的专用模块（os、sys 等），并调用模块中的指定函数来实现。例如，假设如下代码文件的同级目录中有一个文件“a.txt”，通过调用 os 模块中的 remove 函数，可以将该文件删除，具体实现代码如下：

```
1. import os
2. os.remove("a.txt")
```

有关使用 os 模块操作文件更详解的介绍，可阅读《Python os 模块》一节。
而对于文件的应用级操作，通常需要按照固定的步骤进行操作，且实现过程相对比较复杂，同时也是本章重点要讲解的部分。

文件的应用级操作可以分为以下 3 步，每一步都需要借助对应的函数实现：

1. 打开文件：使用 open() 函数，该函数会返回一个文件对象；
2. 对已打开文件做读/写操作：读取文件内容可使用 read()、readline() 以及 readlines() 函数；向文件中写入内容，可以使用 write() 函数。
3. 关闭文件：完成对文件的读/写操作之后，最后需要关闭文件，可以使用 close() 函数。

一个文件，必须在打开之后才能对其进行操作，并且在操作结束之后，还应该将其关闭，这 3 步的顺序不能打乱。
以上操作文件的各个函数，会各自作为一节进行详细介绍。

12.4 Python open()函数详解：打开指定文件

在 Python 中，如果想要操作文件，首先需要创建或者打开指定的文件，并创建一个文件对象，而这些工作可以通过内置的 open() 函数实现。

open() 函数用于创建或打开指定文件，该函数的常用语法格式如下：

```
file = open(file_name [, mode='r' [ , buffering=-1 [ , encoding = None ]]])
```

此格式中，用 [] 括起来的部分为可选参数，即可以使用也可以省略。其中，各个参数所代表的含义如下：

- file：表示要创建的文件对象。
- file_name：要创建或打开文件的文件名称，该名称要用引号（单引号或双引号都可以）括起来。需要注意的是，如果要打开的文件和当前执行的代码文件位于同一目录，则直接写文件名即可；否则，此参数需要指定打开文件所在的完整路径。
- mode：可选参数，用于指定文件的打开模式。可选的打开模式如表 1 所示。如果不写，则默认以只读（r）模式打开文件。
- buffering：可选参数，用于指定对文件做读写操作时，是否使用缓冲区（本节后续会详细介绍）。
- encoding：手动设定打开文件时所使用的编码格式，不同平台的 ecoding 参数值也不同，以 Windows 为例，其默认为 cp936（实际上就是 GBK 编码）。

open() 函数支持的文件打开模式如表 1 所示。

表 1 open 函数支持的文件打开模式

模式	意义	注意事项
r	只读模式打开文件，读文件内容的指针会放在文件的开头。	操作的文件必须存在。
rb	以二进制格式、采用只读模式打开文件，读文件内容的指针位于文件的开头，一般用于非文本文件，如图片文件、音频文件等。	
r+	打开文件后，既可以从头读取文件内容，也可以从开头向文件中写入新的内容，写入的新内容会覆盖文件中等长度的原有内容。	
rb+	以二进制格式、采用读写模式打开文件，读写文件的指针会放在文件的开头，通常针对非文本文件（如音频文件）。	
w	以只写模式打开文件，若该文件存在，打开时会清空文件中原有的内容。	若文件存在，会清空其原有内容（覆盖文件）；反之，则创建新文件。
wb	以二进制格式、只写模式打开文件，一般用于非文本文件（如音频文件）	
w+	打开文件后，会对原有内容进行清空，并对该文件有读写权限。	
wb+	以二进制格式、读写模式打开文件，一般用于非文本文件	
a	以追加模式打开一个文件，对文件只有写入权限，如果文件已经存在，文件指针将放在文件的末尾（即新写入内容会位于已有内容之后）；反之，则会创建新文件。	
ab	以二进制格式打开文件，并采用追加模式，对文件只有写权限。如果该文件已存在，文件指针位于文件末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
a+	以读写模式打开文件；如果文件存在，文件指针放在文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
ab+	以二进制模式打开文件，并采用追加模式，对文件具有读写权限，如果文件存在，则文件指针位于文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	

文件打开模式，直接决定了后续可以对文件做哪些操作。例如，使用 r 模式打开的文件，后续编写的代码只能读取文件，而无法修改文件内容。

图 2 中，将以上几个容易混淆的文件打开模式的功能做了很好的对比：

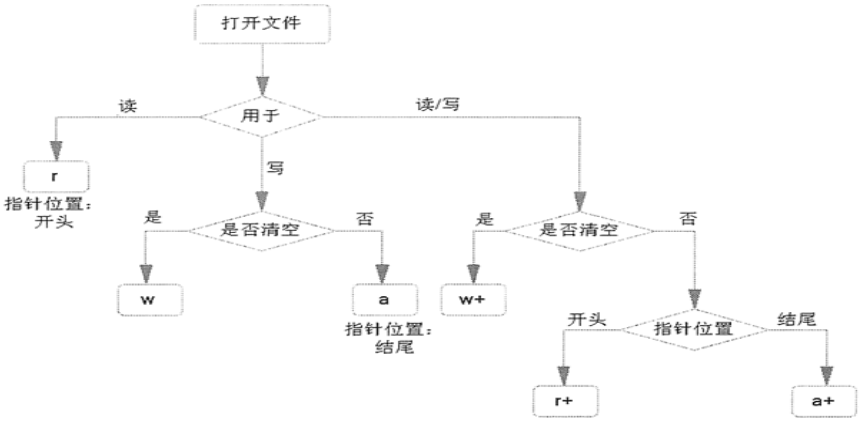


图 2 不同文件打开模式的功能

【例 1】默认打开 "a.txt" 文件。

```
#当前程序文件同目录下没有 a.txt 文件
file = open("a.txt")
print(file)
```

当以默认模式打开文件时，默认使用 r 权限，由于该权限要求打开的文件必须存在，因此运行此代码会报如下错误：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 1, in <module>
    file = open("a.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'a.txt'
```

现在，在程序文件同目录下，手动创建一个 a.txt 文件，并再次运行该程序，其运行结果为：

```
<_io.TextIOWrapper name='a.txt' mode='r' encoding='cp936'>
```

可以看到，当前输出结果中，输出了 file 文件对象的相关信息，包括打开文件的名称、打开模式、打开文件时所使用的编码格式。

使用 open() 打开文件时，默认采用 GBK 编码。但当要打开的文件不是 GBK 编码格式时，可以在使用 open() 函数时，手动指定打开文件的编码格式，例如：

```
file = open("a.txt",encoding="utf-8")
```

注意，手动修改 encoding 参数的值，仅限于文件以文本的形式打开，也就是说，以二进制格式打开时，不能对 encoding 参数的值做任何修改，否则程序会抛出 ValueError 异常，如下所示：

```
ValueError: binary mode doesn't take an encoding argument
```

open()是否需要缓冲区

通常情况下、建议大家在使用 open() 函数时打开缓冲区，即不需要修改 buffing 参数的值。

如果 buffing 参数的值为 0（或者 False），则表示在打开指定文件时不使用缓冲区；如果 buffing 参数值为大于 1 的整数，该整数用于指定缓冲区的大小（单位是字节）；如果 buffing 参数的值为负数，则代表使用默认的缓冲区大小。
为什么呢？原因很简单，目前为止计算机内存的 I/O 速度仍远远高于计算机外设（例如键盘、鼠标、硬盘等）的 I/O 速度，如果不使用缓冲区，则程序在执行 I/O 操作时，内存和外设就必须进行同步读写操作，也就是说，内存必须等待外设输入（输出）一个字节之后，

才能再次输出（输入）一个字节。这意味着，内存中的程序大部分时间都处于等待状态。

而如果使用缓冲区，则程序在执行输出操作时，会先将所有数据都输出到缓冲区中，然后继续执行其它操作，缓冲区中的数据会有外设自行读取处理；同样，当程序执行输入操作时，会先等外设将数据读入缓冲区中，无需同外设做同步读写操作。

open()文件对象常用的属性

成功打开文件之后，可以调用文件对象本身拥有的属性获取当前文件的部分信息，其常见的属性为：

- file.name：返回文件的名称；
- file.mode：返回打开文件时，采用的文件打开模式；
- file.encoding：返回打开文件时使用的编码格式；
- file.closed：判断文件是否已经关闭。

举个例子：

```
# 以默认方式打开文件
f = open('my_file.txt')

# 输出文件是否已经关闭
print(f.closed)

# 输出访问模式
print(f.mode)

#输出编码格式
print(f.encoding)

# 输出文件名
print(f.name)
```

程序执行结果为：

```
False
r
cp936
my_file.txt
```

注意，使用 open() 函数打开的文件对象，必须手动进行关闭（后续章节会详细讲解），Python 垃圾回收机制无法自动回收打开文件所占用的资源。

12.5 以文本格式和二进制格式打开文件，到底有什么区别？

我们知道，open() 函数第二个参数是一个字符串，用于指定文件的打开方式，如果该字符串中出现 b，则表示以二进制格式打开文件；反之，则以普通的文本格式打开文件。

那么，文本文件和二进制文件有什么区别呢？

根据我们以往的经验，文本文件通常用来保存肉眼可见的字符，比如 .txt 文件、.c 文件、.dat 文件等，用文本编辑器打开这些文件，我们能够顺利看懂文件的内容。而二进制文件通常用来保存视频、图片、音频等不可阅读的内容，当用文本编辑器打开这些文件，会看到一堆乱码，根本看不懂。

实际上，从数据存储的角度上分析，二进制文件和文本文件没有区别，它们的内容都是以二进制的形式保存在磁盘中的。

我们之所以能看懂文本文件的内容，是因为文本文件中采用的是 ASCII、UTF-8、GBK 等字符编码，文本编辑器可以识别出这些编码格式，并将编码值转换成字符展示出来。而对于二进制文件，文本编辑器无法识别这些文件的编码格式，只能按照字符编码格式胡乱解析，所以最终看到的是一堆乱码。

open()的文本格式和二进制格式

使用 open() 函数以文本格式打开文件和以二进制格式打开文件，唯一的区别是对文件中换行符的处理不同。

在 Windows 系统中，文件中用 "\r\n" 作为行末标识符（即换行符），当以文本格式读取文件时，会将 "\r\n" 转换成 "\n"；反之，以文本格式将数据写入文件时，会将 "\n" 转换成 "\r\n"。这种隐式转换换行符的行为，对用文本格式打开文本文件是没有问题的，但如果用文本格式打开二进制文件，就有可能改变文本中的数据（将 \r\n 隐式转换为 \n）。

而在 Unix/Linux 系统中，默认的文件换行符就是 \n，因此在 Unix/Linux 系统中文本格式和二进制格式并无本质的区别。

总的来说，为了保险起见，对于 Windows 平台最好用 b 打开二进制文件；对于 Unix/Linux 平台，打开二进制文件，可以用 b，也可以不用。

12.6 Python read()函数：按字节（字符）读取文件

《Python open()函数》一节中，介绍了如何通过 open() 函数打开一个文件。在其基础上，本节继续讲解如何读取已打开文件中的数据。

Python 提供了如下 3 种函数，它们都可以帮我们实现读取文件中数据的操作：

1. read() 函数：逐个字节或者字符读取文件中的内容；
2. readline() 函数：逐行读取文件中的内容；
3. readlines() 函数：一次性读取文件中多行内容。

本节先讲解 read() 函数的用法，readline() 和 readlines() 函数会放到后续章节中作详细介绍。

Python read()函数

对于借助 open() 函数，并以可读模式（包括 r、r+、rb、rb+）打开的文件，可以调用 read() 函数逐个字节（或者逐个字符）读取文件中的内容。

如果文件是以文本模式（非二进制模式）打开的，则 read() 函数会逐个字符进行读取；反之，如果文件以二进制模式打开，则 read() 函数会逐个字节进行读取。
read() 函数的基本语法格式如下：

```
file.read([size])
```

其中，file 表示已打开的文件对象；size 作为一个可选参数，用于指定一次最多可读取的字符（字节）个数，如果省略，则默认一次性读取所有内容。

举个例子，首先创建一个名为 my_file.txt 的文本文件，其内容为：

```
Python 教程
http://c.biancheng.net/python/
```

然后在和 my_file.txt 同目录下，创建一个 file.py 文件，并编写如下语句：

```
#以 utf-8 的编码格式打开指定文件
f = open("my_file.txt",encoding = "utf-8")
#输出读取到的数据
print(f.read())
#关闭文件
f.close()
```

程序执行结果为：

```
Python 教程
http://c.biancheng.net/python/
```

注意，当操作文件结束后，必须调用 close() 函数手动将打开的文件进行关闭，这样可以避免程序发生不必要的错误。

当然，我们也可以通过使用 size 参数，指定 read() 每次可读取的最大字符（或者字节）数，例如：

```
#以 utf-8 的编码格式打开指定文件
f = open("my_file.txt",encoding = "utf-8")
#输出读取到的数据
print(f.read(6))
#关闭文件
f.close()
```

程序执行结果为：

```
Python
```

显然，该程序中的 read() 函数只读取了 my_file 文件开头的 6 个字符。

再次强调，size 表示的是一次最多可读取的字符（或字节）数，因此，即便设置的 size 大于文件中存储的字符（字节）数，read() 函数也不会报错，它只会读取文件中所有的数据。

除此之外，对于以二进制格式打开的文件，read() 函数会逐个字节读取文件中的内容。例如：

```
#以二进制形式打开指定文件
f = open("my_file.txt",'rb+')
#输出读取到的数据
print(f.read())
#关闭文件
f.close()
```

程序执行结果为：

```
b'Python\xe6\x95\x99\xe7\xa8\x8b\r\nhttp://c.biancheng.net/python/'
```

可以看到，输出的数据为 bytes 字节串。我们可以调用 decode() 方法，将其转换成我们认识的字符串。

有关 bytes 字节串，读者可阅读《Python bytes 类型》一节做详细了解。

另外需要注意的一点是，想使用 read() 函数成功读取文件内容，除了严格遵守 read() 的语法外，其还要求 open() 函数必须以可读默认（包括 r、r+、rb、rb+）打开文件。举个例子，将上面程序中 open( ) 的打开模式改为 w，程序会抛出 io.UnsupportedOperation 异常，提示文件没有读取权限：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\file.py", line 3, in <module>
    print(f.read())
io.UnsupportedOperation: not readable
```

read()函数抛出 UnicodeDecodeError 异常的解决方法

在使用 read() 函数时，如果 Python 解释器提示 UnicodeDecodeError 异常，其原因在于，目标文件使用的编码格式和 open() 函数打开该文件时使用的编码格式不匹配。

举个例子，如果目标文件的编码格式为 GBK 编码，而我们在使用 open() 函数并以文本模式打开该文件时，手动指定 encoding 参数为 UTF-8。这种情况下，由于编码格式不匹配，当我们使用 read() 函数读取目标文件中的数据时，Python 解释器就会提示 `UnicodeDecodeError` 异常。

要解决这个问题，要么将 open() 函数中的 encoding 参数值修改为和目标文件相同的编码格式，要么重新生成目标文件（即将该文件的编码格式改为和 open() 函数中的 encoding 参数相同）。

除此之外，还有一种方法：先使用二进制模式读取文件，然后调用 bytes 的 decode() 方法，使用目标文件的编码格式，将读取到的字节串转换成认识的字符串。

举个例子：

```
#以二进制形式打开指定文件，该文件编码格式为 utf-8
f = open("my_file.txt",'rb+')
byt = f.read()
print(byt)
print("\n 转换后：")
print(byt.decode('utf-8'))
#关闭文件
f.close()
```

程序执行结果为：

```
b'Python\xe6\x95\x99\xe7\xa8\x8b\r\nhttp://c.biancheng.net/python/'

转换后：
Python 教程
http://c.biancheng.net/python/
```

12.7 Python readline()和 readlines()函数：按行读取文件

前面章节中讲到，如果想读取用 open() 函数打开的文件中的内容，除了可以使用 read() 函数，还可以使用 readline() 和 readlines() 函数。

和 read() 函数不同，这 2 个函数都以“行”作为读取单位，即每次都读取目标文件中的一行。对于读取以文本格式打开的文件，读取一行很好理解；对于读取以二进制格式打开的文件，它们会以“\n”作为读取一行的标志。

Python readline()函数

readline() 函数用于读取文件中的一行，包含最后的换行符“\n”。此函数的基本语法格式为：

```
file.readline([size])
```

其中，file 为打开的文件对象；size 为可选参数，用于指定读取每一行时，一次最多读取的字符（字节）数。

和 read() 函数一样，此函数成功读取文件数据的前提是，使用 open() 函数指定打开文件的模式必须为可读模式（包括 r、rb、r+、rb+ 4 种）。
仍以前面章节中创建的 my_file.txt 文件为例，该文件中有如下 2 行数据：

```
Python 教程
http://c.biancheng.net/python/
```

下面程序演示了 readline() 函数的具体用法：

```
f = open("my_file.txt")
读取一行数据
byt = f.readline()
print(byt)
```

程序执行结果为：

```
Python 教程

```

由于 readline() 函数在读取文件中一行的内容时，会读取最后的换行符“\n”，再加上 print() 函数输出内容时默认会换行，所以输出结果中会看到多出了一个空行。

不仅如此，在逐行读取时，还可以限制最多可以读取的字符（字节）数，例如：

```
#以二进制形式打开指定文件
f = open("my_file.txt",'rb')
byt = f.readline(6)
print(byt)
```

运行结果为：

```
b'Python'
```

和上一个例子的输出结果相比，由于这里没有完整读取一行的数据，因此不会读取到换行符。

Python readlines()函数

readlines() 函数用于读取文件中的所有行，它和调用不指定 size 参数的 read() 函数类似，只不过该函数返回是一个字符串列表，其中每个元素为文件中的一行内容。

和 readline() 函数一样，readlines() 函数在读取每一行时，会连同行尾的换行符一块读取。
readlines() 函数的基本语法格式如下：

```
file.readlines()
```

其中，file 为打开的文件对象。和 read()、readline() 函数一样，它要求打开文件的模式必须为可读模式（包括 r、rb、r+、rb+ 4 种）。

举个例子：

```
1.  f = open("my_file.txt",'rb')
2.  byt = f.readlines()
3.  print(byt)
```

运行结果为：

```
[b'Python\xbd\xcc\xb3\xcc\r\n', b'http://c.biancheng.net/python/']
```

12.8 Python write()和 writelines()：向文件中写入数据

前面章节中学习了如何使用 read()、readline() 和 readlines() 这 3 个函数读取文件，如果我们想把一些数据保存到文件中，又该如何实现呢？

Python 中的文件对象提供了 write() 函数，可以向文件中写入指定内容。该函数的语法格式如下：

```
file.write(string)
```

其中，file 表示已经打开的文件对象；string 表示要写入文件的字符串（或字节串，仅适用写入二进制文件中）。

注意，在使用 write() 向文件中写入数据，需保证使用 open() 函数是以 r+、w、w+、a 或 a+ 的模式打开文件，否则执行 write() 函数会抛出 io.UnsupportedOperation 错误。
例如，创建一个 a.txt 文件，该文件内容如下：

```
C 语言中文网
http://c.biancheng.net
```

然后，在和 a.txt 文件同级目录下，创建一个 Python 文件，编写如下代码：

```
1.  f = open("a.txt", 'w')
2.  f.write("写入一行新数据")
3.  f.close()
```

前面已经讲过，如果打开文件模式中包含 w（写入），那么向文件中写入内容时，会先清空原文件中的内容，然后再写入新的内容。因此运行上面程序，再次打开 a.txt 文件，只会看到新写入的内容：

```
写入一行新数据
```

而如果打开文件模式中包含 a（追加），则不会清空原有内容，而是将新写入的内容会添加到原内容后边。例如，还原 a.txt 文件中的内容，并修改上面代码为：

```
1.  f = open("a.txt", 'a')
2.  f.write("\n 写入一行新数据")
3.  f.close()
```

再次打开 a.txt，可以看到如下内容：

```
C 语言中文网
http://c.biancheng.net
写入一行新数据
```

因此，采用不同的文件打开模式，会直接影响 write() 函数向文件中写入数据的效果。

另外，在写入文件完成后，一定要调用 close() 函数将打开的文件关闭，否则写入的内容不会保存到文件中。例如，将上面程序中最后一行 f.close() 删掉，再次运行此程序并打开 a.txt，你会发现该文件是空的。这是因为，当我们在写入文件内容时，操作系统不会立刻把数据写入磁盘，而是先缓存起来，只有调用 close() 函数时，操作系统才会保证把没有写入的数据全部写入磁盘文件中。

除此之外，如果向文件写入数据后，不想马上关闭文件，也可以调用文件对象提供的 flush() 函数，它可以实现将缓冲区的数据写入文件中。例如：

```
f = open("a.txt", 'w')
f.write("写入一行新数据")
f.flush()
```

打开 a.txt 文件，可以看到写入的新内容：

```
写入一行新数据
```

有读者可能会想到，通过设置 open() 函数的 buffering 参数可以关闭缓冲区，这样数据不就可以直接写入文件中了？对于以二进制格式打开的文件，可以不使用缓冲区，写入的数据会直接进入磁盘文件；但对于以文本格式打开的文件，必须使用缓冲区，否则 Python 解释器会 ValueError 错误。例如：

```
f = open("a.txt", 'w',buffering = 0)
f.write("写入一行新数据")
```

运行结果为：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 1, in <module>
    f = open("a.txt", 'w',buffering = 0)
ValueError: can't have unbuffered text I/O
```

Python writelines()函数

Python 的文件对象中，不仅提供了 write() 函数，还提供了 writelines() 函数，可以实现将字符串列表写入文件中。

注意，写入函数只有 write() 和 writelines() 函数，而没有名为 writeline 的函数。
例如，还是以 a.txt 文件为例，通过使用 writelines() 函数，可以轻松实现将 a.txt 文件中的数据复制到其它文件中，实现代码如下：

```
f = open('a.txt', 'r')
n = open('b.txt','w+')
n.writelines(f.readlines())
n.close()
f.close()
```

执行此代码，在 a.txt 文件同级目录下会生成一个 b.txt 文件，且该文件中包含的数据和 a.txt 完全一样。

需要注意的是，使用 writelines() 函数向文件中写入多行数据时，不会自动给各行添加换行符。上面例子中，之所以 b.txt 文件中会逐行显示数据，是因为 readlines() 函数在读取各行数据时，读入了行尾的换行符。

12.9 Python close()函数：关闭文件

在前面章节中，对于使用 open() 函数打开的文件，我们一直都在用 close() 函数将其手动关闭。本节就来详细介绍一下 close() 函数。

close() 函数是专门用来关闭已打开文件的，其语法格式也很简单，如下所示：

```
file.close()
```

其中，file 表示已打开的文件对象。

读者可能一直存在这样的疑问，即使用 open() 函数打开的文件，在操作完成之后，一定要调用 close() 函数将其关闭吗？答案是肯定的。文件在打开并操作完成之后，就应该及时关闭，否则程序的运行可能出现问题。

举个例子，分析如下代码：

```
1.  import os
2.  f = open("my_file.txt",'w')
3.  #...
4.  os.remove("my_file.txt")
```

代码中，我们引入了 os 模块，调用了该模块中的 remove() 函数，该函数的功能是删除指定的文件。但是，如果运行此程序，Python 解释器会报如下错误：

```
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\demo.py", line 4, in <module>
    os.remove("my_file.txt")
PermissionError: [WinError 32] 另一个程序正在使用此文件，进程无法访问。: 'my_file.txt'
```

显然，由于我们使用了 open() 函数打开了 my_file.txt 文件，但没有及时关闭，直接导致后续的 remove() 函数运行出现错误。因此，正确的程序应该是这样的：

```
1.  import os
2.  f = open("my_file.txt",'w')
3.  f.close()
4.  #...
5.  os.remove("my_file.txt")
```

当确定 my_file.txt 文件可以被删除时，再次运行程序，可以发现该文件已经被成功删除了。

再举个例子，如果我们不调用 close() 函数关闭已打开的文件，确定不影响读取文件的操作，但会导致 write() 或者 writeline() 函数向文件中写数据时，写入失败。例如：

```
1.  f = open("my_file.txt", 'w')
2.  f.write("http://c.biancheng.net/shell/")
```

程序执行后，虽然 Python 解释器不报错，但打开 my_file.txt 文件会发现，根本没有写入成功。这是因为，在向以文本格式（而不是二进制格式）打开的文件中写入数据时，Python 出于效率的考虑，会先将数据临时存储到缓冲区中，只有使用 close() 函数关闭文件时，才会将缓冲区中的数据真正写入文件中。

因此，在上面程序的最后添加如下代码：

```
1.  f.close()
```

再次运行程序，就会看到 "http://c.biancheng.net/shell/" 成功写入到了 a.txt 文件。

当然在某些实际场景中，我们可能需要在将数据成功写入到文件中，但并不想关闭文件。这也是可以实现的，调用 flush() 函数即可，例如：

```
1.  f = open("my_file.txt", 'w')
2.  f.write("http://c.biancheng.net/shell/")
3.  f.flush()
```

打开 my_file.txt 文件，会发现已经向文件中成功写入了字符串“http://c.biancheng.net/shell/”。

12.10 Python seek()和 tell()函数详解

在讲解 seek() 函数和 tell() 函数之前，首先来了解一下什么是文件指针。

我们知道，使用 open() 函数打开文件并读取文件中的内容时，总是会从文件的第一个字符（字节）开始读起。那么，有没有办法可以自定指定读取的起始位置呢？答案是肯定，这就需要移动文件指针的位置。

文件指针用于标明文件读写的起始位置。假如把文件看成一个水流，文件中每个数据（以 b 模式打开，每个数据就是一个字节；以普通模式打开，每个数据就是一个字符）就相当于一个水滴，而文件指针就标明了文件将要从文件的哪个位置开始读起。图 1 简单示意了文件指针的概念。

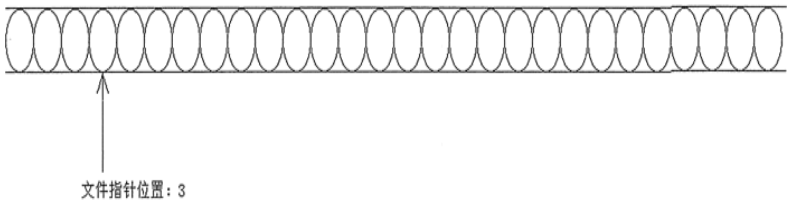


图 1 文件指针概念示意图

可以看到，通过移动文件指针的位置，再借助 read() 和 write() 函数，就可以轻松实现，读取文件中指定位置的数据（或者向文件中的指定位置写入数据）。

注意，当向文件中写入数据时，如果不是文件的尾部，写入位置的原有数据不会自行向后移动，新写入的数据会将文件中处于该位置的数据直接覆盖掉。

实现对文件指针的移动，文件对象提供了 tell() 函数和 seek() 函数。tell() 函数用于判断文件指针当前所处的位置，而 seek() 函数用于移动文件指针到文件的指定位置。

tell() 函数

tell() 函数的用法很简单，其基本语法格式如下：

```
file.tell()
```

其中，file 表示文件对象。

例如，在同一目录下，编写如下程序对 a.txt 文件做读取操作，a.txt 文件中内容为：

```
http://c.biancheng.net
```

读取 a.txt 的代码如下：

```
1.  f = open("a.txt",'r')
2.  print(f.tell())
3.  print(f.read(3))
4.  print(f.tell())
```

运行结果为：

```
0
htt
3
```

可以看到，当使用 open() 函数打开文件时，文件指针的起始位置为 0，表示位于文件的开头处，当使用 read() 函数从文件中读取 3 个字符之后，文件指针同时向后移动了 3 个字符的位置。这就表明，当程序使用文件对象读写数据时，文件指针会自动向后移动：读写了多少个数据，文件指针就自动向后移动多少个位置。

seek()函数

seek() 函数用于将文件指针移动至指定位置，该函数的语法格式如下：

```
file.seek(offset[, whence])
```

其中，各个参数的含义如下：

- file：表示文件对象；
- whence：作为可选参数，用于指定文件指针要放置的位置，该参数的参数值有 3 个选择：0 代表文件头（默认值）、1 代表当前位置、2 代表文件尾。
- offset：表示相对于 whence 位置文件指针的偏移量，正数表示向后偏移，负数表示向前偏移。例如，当 whence == 0 &&offset == 3（即 seek(3,0) ），表示文件指针移动至距离文件开头处 3 个字符的位置；当 whence == 1 &&offset == 5（即 seek(5,1) ），表示文件指针向后移动，移动至距离当前位置 5 个字符处。

注意，当 offset 值非 0 时，Python 要求文件必须要以二进制格式打开，否则会抛出 io.UnsupportedOperation 错误。
下面程序示范了文件指针操作：

```
f = open('a.txt', 'rb')
# 判断文件指针的位置
print(f.tell())
# 读取一个字节，文件指针自动后移 1 个数据
print(f.read(1))
print(f.tell())
# 将文件指针从文件开头，向后移动到 5 个字符的位置
f.seek(5)
print(f.tell())
print(f.read(1))
# 将文件指针从当前位置，向后移动到 5 个字符的位置
f.seek(5, 1)
print(f.tell())
print(f.read(1))
# 将文件指针从文件结尾，向前移动到距离 2 个字符的位置
f.seek(-1, 2)
print(f.tell())
print(f.read(1))
```

运行结果为：

```
0
b'h'
1
5
b'/'
11
b'a'
21
b't'
```

注意：由于程序中使用 seek() 时，使用了非 0 的偏移量，因此文件的打开方式中必须包含 b，否则就会报 io.UnsupportedOperation 错误，有兴趣的读者可自定尝试。

上面程序示范了使用 seek() 方法来移动文件指针，包括从文件开头、指针当前位置、文件结尾处开始计算。运行上面程序，结合程序输出结果可以体会文件指针移动的效果。

12.11 Python with as 用法详解

任何一门编程语言中，文件的输入输出、数据库的连接断开等，都是很常见的资源管理操作。但资源都是有限的，在写程序时，必须保证这些资源在使用过后得到释放，不然就容易造成资源泄露，轻者使得系统处理缓慢，严重时会使系统崩溃。

例如，前面在介绍文件操作时，一直强调打开的文件最后一定要关闭，否则会程序的运行造成意想不到的隐患。但是，即便使用 close() 做好了关闭文件的操作，如果在打开文件或文件操作过程中抛出了异常，还是无法及时关闭文件。

为了更好地避免此类问题，不同的编程语言都引入了不同的机制。在 Python 中，对应的解决方式是使用 with as 语句操作上下文管理器（context manager），它能够帮助我们自动分配并且释放资源。

简单的理解，同时包含 __enter__() 和 __exit__() 方法的对象就是上下文管理器。常见构建上下文管理器的方式有 2 种，分别是基于类实现和基于生成器实现，在《什么是上下文管理器，深入底层了解 with as 语句》一文有详细介绍。

例如，使用 with as 操作已经打开的文件对象（本身就是上下文管理器），无论期间是否抛出异常，都能保证 with as 语句执行完毕后自动关闭已经打开的文件。

首先学习如何使用 with as 语句。with as 语句的基本语法格式为：

```
with 表达式 [as target]：
    代码块
```

此格式中，用 [] 括起来的部分可以使用，也可以省略。其中，target 参数用于指定一个变量，该语句会将 expression 指定的结果保存到该变量中。with as 语句中的代码块如果不想执行任何语句，可以直接使用 pass 语句代替。

举个例子，假设有一个 a.txt 文件，其存储内容如下：

```
C 语言中文网
http://c.biancheng.net
```

在和 a.txt 同级目录下，创建一个 .py 文件，并编写如下代码：

```
1.  with open('a.txt', 'a') as f:
2.      f.write("\nPython 教程")
```

运行结果为：

```
C 语言中文网
http://c.biancheng.net
Python 教程
```

可以看到，通过使用 with as 语句，即便最终没有关闭文件，修改文件内容的操作也能成功。

with as 语句实现的底层原理到底是怎样的呢？可以阅读《什么是上下文管理器，深入底层了解 with as 语句》一文做详细了解。

12.12 什么是上下文管理器，Python with as 底层原理详解

在介绍 with as 语句时讲到，该语句操作的对象必须是上下文管理器。那么，到底什么是上下文管理器呢？

简单的理解，同时包含 __enter__() 和 __exit__() 方法的对象就是上下文管理器。也就是说，上下文管理器必须实现如下两个方法：

1. __enter__(self)：进入上下文管理器自动调用的方法，该方法会在 with as 代码块执行之前执行。如果 with 语句有 as 子句，那么该方法的返回值会被赋值给 as 子句后的变量；该方法可以返回多个值，因此在 as 子句后面也可以指定多个变量（多个变量必须由“()”括起来组成元组）。
2. __exit__（self, exc_type, exc_value, exc_traceback）：退出上下文管理器自动调用的方法。该方法会在 with as 代码块执行之后执行。如果 with as 代码块成功执行结束，程序自动调用该方法，调用该方法的三个参数都为 None；如果 with as 代码块因为异常而中止，程序也自动调用该方法，使用 sys.exc_info 得到的异常信息将作为调用该方法的参数。

当 with as 操作上下文管理器时，就会在执行语句体之前，先执行上下文管理器的 __enter__() 方法，然后再执行语句体，最后执行 __exit__() 方法。

构建上下文管理器，常见的有 2 种方式：基于类实现和基于生成器实现。

基于类的上下文管理器

通过上面的介绍不难发现，只要一个类实现了 __enter__() 和 __exit__() 这 2 个方法，程序就可以使用 with as 语句来管理它，通过 __exit__() 方法的参数，即可判断出 with 代码块执行时是否遇到了异常。其实，上面程序中的文件对象也实现了这两个方法，因此可以接受 with as 语句的管理。

下面我们自定义一个实现上下文管理协议的类，并尝试用 with as 语句来管理它：

```
class FkResource:
    def __init__(self, tag):
        self.tag = tag
        print('构造器,初始化资源: %s' % tag)
    # 定义__enter__方法，with 体之前的执行的方法
    def __enter__(self):
        print('[__enter__ %s]: ' % self.tag)
        # 该返回值将作为 as 子句中变量的值
        return 'fkit'  # 可以返回任意类型的值
    # 定义__exit__方法，with 体之后的执行的方法
    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('[__exit__ %s]: ' % self.tag)
        # exc_traceback 为 None，代表没有异常
        if exc_traceback is None:
            print('没有异常时关闭资源')
        else:
            print('遇到异常时关闭资源')
```

```
            return False   # 可以省略，默认返回 None 也被看做是 False
with FkResource('孙悟空') as dr:
    print(dr)
    print('[with 代码块] 没有异常')
print('------------------------------')
with FkResource('白骨精'):
    print('[with 代码块] 异常之前的代码')
    raise Exception
    print('[with 代码块] ^^^^^^^^异常之后的代码')
```

运行上面的程序，可以看到如下输出结果：

```
构造器,初始化资源: 孙悟空
[__enter__ 孙悟空]:
fkit
[with 代码块] 没有异常
[__exit__ 孙悟空]:
没有异常时关闭资源
------------------------------
构造器,初始化资源: 白骨精
[__enter__ 白骨精]:
[with 代码块] 异常之前的代码
[__exit__ 白骨精]:
遇到异常时关闭资源
Traceback (most recent call last):
  File "C:\Users\mengma\Desktop\1.py", line 26, in <module>
    raise Exception
Exception
```

上面程序定义了一个 FkResource 类，并包含了 __enter__() 和 __exit__() 两个方法，因此该类的对象可以被 with as 语句管理。

此外，程序中两次使用 with as 语句管理 FkResource 对象。第一次代码块没有出现异常，第二次代码块出现了异常。从上面的输出结果来看，使用 with as 语句管理资源，无论代码块是否有异常，程序总可以自动执行 __exit__() 方法。

> 注意，当出现异常时，如果 __exit__ 返回 False（默认不写返回值时，即为 False），则会重新抛出异常，让 with as 之外的语句逻辑来处理异常；反之，如果返回 True，则忽略异常，不再对异常进行处理。

基于生成器的上下文管理器

除了基于类的上下文管理器，它还可以基于生成器实现。接下来先看一个例子。比如，我们可以使用装饰器 contextlib.contextmanager，来定义自己所需的基于生成器的上下文管理器，用以支持 with as 语句：

```
from contextlib import contextmanager

@contextmanager
def file_manager(name, mode):
    try:
```

```
6.          f = open(name, mode)
7.          yield f
8.      finally:
9.          f.close()
10.
11. with file_manager('a.txt', 'w') as f:
12.     f.write('hello world')
```

这段代码中，函数 file_manager() 就是一个生成器，当我们执行 with as 语句时，便会打开文件，并返回文件对象 f；当 with 语句执行完后，finally 中的关闭文件操作便会执行。另外可以看到，使用基于生成器的上下文管理器时，不再用定义 __enter__() 和 __exit__() 方法，但需要加上装饰器 @contextmanager，这一点新手很容易疏忽。

需要强调的是，基于类的上下文管理器和基于生成器的上下文管理器，这两者在功能上是一致的。只不过，基于类的上下文管理器更加灵活，适用于大型的系统开发，而基于生成器的上下文管理器更加方便、简洁，适用于中小型程序。但是，无论使用哪一种，不用忘记在方法“__exit__()”或者是 finally 块中释放资源，这一点尤其重要。

12.13 Python pickle 模块：实现 Python 对象的持久化存储

Python 中有个序列化过程叫作 pickle，它能够实现任意对象与文本之间的相互转化，也可以实现任意对象与二进制之间的相互转化。也就是说，pickle 可以实现 Python 对象的存储及恢复。

值得一提的是，pickle 是 python 语言的一个标准模块，安装 python 的同时就已经安装了 pickle 库，因此它不需要再单独安装，使用 import 将其导入到程序中，就可以直接使用。

pickle 模块提供了以下 4 个函数供我们使用：

1. dumps()：将 Python 中的对象序列化成二进制对象，并返回；
2. loads()：读取给定的二进制对象数据，并将其转换为 Python 对象；
3. dump()：将 Python 中的对象序列化成二进制对象，并写入文件；
4. load()：读取指定的序列化数据文件，并返回对象。

以上这 4 个函数可以分成两类，其中 dumps 和 loads 实现基于内存的 Python 对象与二进制互转；dump 和 load 实现基于文件的 Python 对象与二进制互转。

pickle.dumps()函数

此函数用于将 Python 对象转为二进制对象，其语法格式如下：

```
dumps(obj, protocol=None, *, fix_imports=True)
```

此格式中各个参数的含义为：

- obj：要转换的 Python 对象；
- protocol：pickle 的转码协议，取值为 0、1、2、3、4，其中 0、1、2 对应 Python 早期的版本，3 和 4 则对应 Python 3.x 版本及之后的版本。未指定情况下，默认为 3。
- 其它参数：为了兼容 Python 2.x 版本而保留的参数，Python 3.x 中可以忽略。

【例 1】

```
1.  import pickle
2.  tup1 = ('I love Python', {1,2,3}, None)
3.  #使用 dumps() 函数将 tup1 转成 p1
4.  p1 = pickle.dumps(tup1)
5.  print(p1)
```

输出结果为：

```
b'\x80\x03X\r\x00\x00\x00I love Pythonq\x00cbuiltins\nset\nq\x01]q\x02(K\x01K\x02K\x03e\x85q\x03Rq\x04N\x87q\x05.'
```

pickle.loads()函数

此函数用于将二进制对象转换成 Python 对象，其基本格式如下：

```
loads(data, *, fix_imports=True, encoding='ASCII', errors='strict')
```

其中，data 参数表示要转换的二进制对象，其它参数只是为了兼容 Python 2.x 版本而保留的，可以忽略。

【例 2】在例 1 的基础上，将 p1 对象反序列化为 Python 对象。

```
1.  import pickle
2.  tup1 = ('I love Python', {1,2,3}, None)
3.  p1 = pickle.dumps(tup1)
4.  #使用 loads() 函数将 p1 转成 Python 对象
5.  t2 = pickle.loads(p1)
6.  print(t2)
```

运行结果为：

```
('I love Python', {1, 2, 3}, None)
```

注意，在使用 loads() 函数将二进制对象反序列化成 Python 对象时，会自动识别转码协议，所以不需要将转码协议当作参数传入。并且，当待转换的二进制对象的字节数超过 pickle 的 Python 对象时，多余的字节将被忽略。

pickle.dump()函数

此函数用于将 Python 对象转换成二进制文件，其基本语法格式为：

```
dump (obj, file,protocol=None, *, fix mports=True)
```

其中各个参数的具体含义如下：

- obj：要转换的 Python 对象。
- file：转换到指定的二进制文件中，要求该文件必须是以"wb"的打开方式进行操作。
- protocol：和 dumps() 函数中 protocol 参数的含义完全相同，因此这里不再重复描述。
- 其他参数：为了兼容以前 Python 2.x 版本而保留的参数，可以忽略。

【例 3】将 tup1 元组转换成二进制对象文件。

```
1.  import pickle
2.  tup1 = ('I love Python', {1,2,3}, None)
3.  #使用 dumps() 函数将 tup1 转成 p1
4.  with open ("a.txt", 'wb') as f: #打开文件
5.      pickle.dump(tup1, f) #用 dump 函数将 Python 对象转成二进制对象文件
```

运行完此程序后，会在该程序文件同级目录中，生成 a.txt 文件，但由于其内容为二进制数据，因此直接打开会看到乱码。

pickle.load()函数

此函数和 dump() 函数相对应，用于将二进制对象文件转换成 Python 对象。该函数的基本语法格式为：

```
load(file, *, fix_imports=True, encoding='ASCII', errors='strict')
```

其中，file 参数表示要转换的二进制对象文件（必须以 "rb" 的打开方式操作文件），其它参数只是为了兼容 Python 2.x 版本而保留的参数，可以忽略。

【例 4】将例 3 转换的 a.txt 二进制文件对象转换为 Python 对象。

```
import pickle
tup1 = ('I love Python', {1,2,3}, None)
#使用 dumps() 函数将 tup1 转成 p1
with open ("a.txt", 'wb') as f: #打开文件
    pickle.dump(tup1, f) #用 dump 函数将 Python 对象转成二进制对象文件
with open ("a.txt", 'rb') as f: #打开文件
    t3 = pickle.load(f) #将二进制文件对象转换成 Python 对象
    print(t3)
```

运行结果为：

```
('I love Python', {1, 2, 3}, None)
```

总结

看似强大的 pickle 模块，其实也有它的短板，即 pickle 不支持并发地访问持久性对象，在复杂的系统环境下，尤其是读取海量数据时，使用 pickle 会使整个系统的 I/O 读取性能成为瓶颈。这种情况下，可以使用 ZODB。

ZODB 是一个健壮的、多用户的和面向对象的数据库系统，专门用于存储 Python 语言中的对象数据，它能够存储和管理任意复杂的 Python 对象，并支持事务操作和并发控制。并且，ZODB 也是在 Python 的序列化操作基础之上实现的，因此要想有效地使用 ZODB，必须先学好 pickle。

有关 ZODB 的详细介绍，读者可自行搜索相关文档，本节不再具体讲解。

12.14 Python fileinput 模块：逐行读取多个文件

前面章节中，我们学会了使用 open() 和 read()（或者 readline()、readlines() ）组合，来读取单个文件中的数据。但在某些场景中，可能需要读取多个文件的数据，这种情况下，再使用这个组合，显然就不合适了。

庆幸的是，Python 提供了 fileinput 模块，通过该模块中的 input() 函数，我们能同时打开指定的多个文件，还可以逐个读取这些文件中的内容。

fileinput 模块中 input() 该函数的语法格式如下：

```
fileinput.input ( files="filename1, filename2, ...", inplace=False, backup='', bufsize=0, mode='r', openhook=None )
```

此函数会返回一个 FileInput 对象，它可以理解为是将多个指定文件合并之后的文件对象。其中，各个参数的含义如下：

- files：多个文件的路径列表；
- inplace：用于指定是否将标准输出的结果写回到文件，此参数默认值为 False；
- backup：用于指定备份文件的扩展名；
- bufsize：指定缓冲区的大小，默认为 0；
- mode：打开文件的格式，默认为 r（只读格式）；
- openhook：控制文件的打开方式，例如编码格式等。

注意，和 open() 函数不同，input() 函数不能指定打开文件的编码格式，这意味着使用该函数读取的所有文件，除非以二进制方式进行读取，否则该文件编码格式都必须和当前操作系统默认的编码格式相同，不然 Python 解释器可能会提示 UnicodeDecodeError 错误。和 open() 函数返回单个的文件对象不同，fileinput 对象无需调用类似 read()、readline()、readlines() 这样的函数，直接通过 for 循环即可按次序读取多个文件中的数据。

值得一提的是，fileinput 模块还提供了很多使用的函数（如表 1 所示），通过调用这些函数，可以帮我们更快地实现想要的功能。

表 1 fileinput 模块常用函数

函数名	功能描述
fileinput.filename()	返回当前正在读取的文件名称。
fileinput.fileno()	返回当前正在读取文件的文件描述符。
fileinput.lineno()	返回当前读取了多少行。
fileinput.filelineno()	返回当前正在读取的内容位于当前文件中的行号。
fileinput.isfirstline()	判断当前读取的内容在当前文件中是否位于第 1 行。
fileinput.nextfile()	关闭当前正在读取的文件，并开始读取下一个文件。
fileinput.close()	关闭 FileInput 对象。

文件描述符是一个文件的代号，其值为一个整数。后续章节将会介绍关于文件描述符的操作。

讲了这么多，接下来举个例子。假设使用 input() 读取 2 个文件，分别为 my_file.txt 和 file.txt，它们位于同一目录，且各自包含的内容如下所示：

```
#file.txt
Python 教程
http://c.biancheng.net/python/

#my_file.txt
```

Linux 教程
http://c.biancheng.net/linux_tutorial/

下面程序演示了如何使用 input() 函数依次读取这 2 个文件：

```
import fileinput
#使用 for 循环遍历 fileinput 对象
for line in fileinput.input(files=('my_file.txt', 'file.txt')):
    # 输出读取到的内容
    print(line)
# 关闭文件流
fileinput.close()
```

在使用 fileinput 模块中的 input() 函数之前，一定要先引入 fileinput 模块。
程序执行结果为：

```
Linux 教程

http://c.biancheng.net/linux_tutorial/
Python 教程

http://c.biancheng.net/python/
```

显然，读取文件内容的次序，取决于 input() 函数中文件名的先后次序。

12.15 Python linecache 模块用法：随机读取文件指定行

除了可以借助 fileinput 模块实现读取文件外，Python 还提供了 linecache 模块。和前者不同，linecache 模块擅长读取指定文件中的指定行。换句话说，如果我们想读取某个文件中指定行包含的数据，就可以使用 linecache 模块。

值得一提的是，linecache 模块常用来读取 Python 源文件中的代码，它使用的是 UTF-8 编码格式来读取文件内容。这意味着，使用该模块读取的文件，其编码格式也必须为 UTF-8，否则要么读取出来的数据是乱码，要么直接读取失败（Python 解释器会报 SyntaxError 异常）。

要使用 linecache 模块，就必须知道其包含了哪些函数。linecache 模块中常用的函数及其功能如表 1 所示。

表 1 linecache 模块常用函数及功能

函数基本格式	功能
linecache.getline(filename, lineno, module_globals=None)	读取指定模块中指定文件的指定行（仅读取指定文件时，无需指定模块）。其中，filename 参数用来指定文件名，lineno 用来指定行号，module_globals 参数用来指定要读取的具体模块名。注意，当指定文件以相对路径的方式传给 filename 参数时，该函数以按照 sys.path 规定的路径查找该文件。
linecache.clearcache()	如果程序某处，不再需要之前使用 getline() 函数读取的数据，则可以使用该函数清空缓存。
linecache.checkcache(filename=None)	检查缓存的有效性，即如果使用 getline() 函数读取的数据，其实在本地已经被修改，而我们需要的是新的数据，此时就可以使用该函数检查缓存的是否为新的数据。注意，如果省略文件名，该函数将检车所有缓存数据的有效性。

举个例子：

```
1.  import linecache
2.  import string
3.  #读取 string 模块中第 3 行的数据
4.  print(linecache.getline(string.__file__, 3))
5.
6.  # 读取普通文件的第 2 行
7.  print(linecache.getline('my_file.txt', 2))
```

在执行该程序之前，需保证 my_file.txt 文件是以 UTF-8 编码格式保存的（Python 提供的模块，通常编码格式为 UTF-8）。在此基础上，执行该程序，其输出结果为：

```
Public module variables:

http://c.biancheng.net/linux_tutorial/
```

12.16 Python pathlib 模块用法详解

和前面章节中引入的模板不同，pathlib 模块中包含的是一些类，它们的继承关系如图 1 所示。

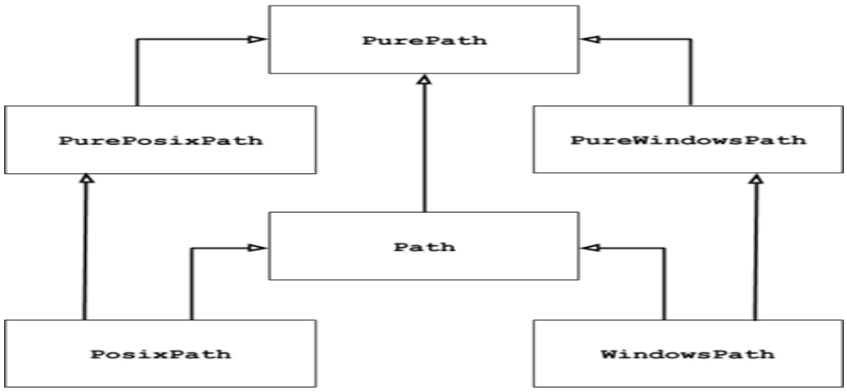


图 1 pathlib 模块中类的组织结构

图 1 中，箭头连接的是有继承关系的两个类，以 PurePosixPath 和 PurePath 类为例，PurePosizPath 继承自 PurePath，即前者是后者的子类。

pathlib 模块的操作对象是各种操作系统中使用的路径（例如指定文件位置的路径，包括绝对路径和相对路径）。这里简单介绍一下图 1 中包含的几个类的具体功能：

- PurePath 类会将路径看做是一个普通的字符串，它可以实现将多个指定的字符串拼接成适用于当前操作系统的路径格式，同时还可以判断任意两个路径是否相等。注意，使用 PurePath 操作的路径，它并不会关心该路径是否真实有效。
- PurePosixPath 和 PureWindowsPath 是 PurePath 的子类，前者用于操作 UNIX（包括 Mac OS X）风格的路径，后者用于操作 Windows 风格的路径。
- Path 类和以上 3 个类不同，它操作的路径一定是真实有效的。Path 类提供了判断路径是否真实存在的方法。
- PosixPath 和 WindowPath 是 Path 的子类，分别用于操作 Unix（Mac OS X）风格的路径和 Windows 风格的路径。

注意，UNIX 操作系统和 Windows 操作系统上，路径的格式是完全不同的，主要区别在于根路径和路径分隔符，UNIX 系统的根路径是斜杠（/），而 Windows 系统的根路径是盘符（C:）；UNIX 系统路径使用的分隔符是斜杠（/），而 Windows 使用的是反斜杠（\）。

PurePath 类的用法

PurePath 类（以及 PurePosixPath 类和 PureWindowsPath 类）都提供了大量的构造方法、实例方法以及类实例属性，供我们使用。

PurePath 类构造方法

需要注意的是，在使用 PurePath 类时，考虑到操作系统的不同，如果在 UNIX 或 Mac OS X 系统上使用 PurePath 创建对象，该类的构造方法实际返回的是 PurePosixPath 对象；反之，如果在 Windows 系统上使用 PurePath 创建对象，该类的构造方法返回的是 PureWindowsPath 对象。

当然，我们完全可以直接使用 PurePosixPath 类或者 PureWindowsPath 类创建指定操作系统使用的类对象。

例如，在 Windows 系统上执行如下语句：

```
1.  from pathlib import *
2.  # 创建 PurePath，实际上使用 PureWindowsPath
3.  path = PurePath('my_file.txt')
4.  print(type(path))
```

程序执行结果为：

```
<class 'pathlib.PureWindowsPath'>
```

显然，在 Windows 操作系统上，使用 PurePath 类构造函数创建的是 PureWindowsPath 类对象。

读者可自行尝试在 UNIX 或者 Mac OS X 系统上执行该程序。

除此之外，PurePath 在创建对象时，也支持传入多个路径字符串，它们会被拼接成一个路径格式的字符串。例如：

```
1.  from pathlib import *
2.  # 创建 PurePath，实际上使用 PureWindowsPath
3.  path = PurePath('http:','c.biancheng.net','python')
4.  print(path)
```

程序执行结果为：

```
http:\c.biancheng.net\python
```

可以看到，由于本机为 Windows 系统，因此这里输出的是适用于 Windows 平台的路径。如果想在 Windows 系统上输出 UNIX 风格的路径字符串，就需要使用 PurePosixPath 类。例如：

```
1.  from pathlib import *
2.  path = PurePosixPath('http:','c.biancheng.net','python')
3.  print(path)
```

程序执行结果为：

```
http:/c.biancheng.net/python
```

值的一提的是，如果在使用 PurePath 类构造方法时，不传入任何参数，则等同于传入点 '.'（表示当前路径）作为参数。例如：

```
1.  from pathlib import *
2.  path = PurePath()
3.  print(path)
4.
5.  path = PurePath('.')
6.  print(path)
```

程序执行结果为：

```
.
.
```

另外，如果传入 PurePath 构造方法中的多个参数中，包含多个根路径，则只会有最后一个根路径及后面的子路径生效。例如：

```
from pathlib import *
path = PurePath('C://','D://','my_file.txt')
print(path)
```

程序执行结果为：

```
D:\my_file.txt
```

注意，对于 Windows 风格的路径，只有盘符（如 C、D 等）才能算根路径。

需要注意的是，如果传给 PurePath 构造方法的参数中包含有多余的斜杠或者点（ . ，表示当前路径），会直接被忽略（ .. 不会被忽略）。举个例子：

```
from pathlib import *
path = PurePath('C://./my_file.txt')
print(path)
```

程序执行结果为：

```
C:\my_file.txt
```

PurePath 类还重载各种比较运算符，多余同种风格的路径字符串来说，可以判断是否相等，也可以比较大小（实际上就是比较字符串的大小）；对于不同种风格的路径字符串之间，只能判断是否相等（显然，不可能相等），但不能比较大小。

举个例子：

```
from pathlib import *
# Unix 风格的路径区分大小写
print(PurePosixPath('C://my_file.txt') == PurePosixPath('c://my_file.txt'))

# Windows 风格的路径不区分大小写
print(PureWindowsPath('C://my_file.txt') == PureWindowsPath('c://my_file.txt'))
```

程序执行结果为：

```
False
True
```

比较特殊的是，PurePath 类对象支持直接使用斜杠（/）作为多个字符串之间的连接符，例如：

```
from pathlib import *
path = PurePosixPath('C://')
print(path / 'my_file.txt')
```

程序执行结果为：

```
C:/my_file.txt
```

通过以上方式构建的路径，其本质上就是字符串，因此我们完全可以使用 str() 将 PurePath 对象转换成字符串。例如：

```
from pathlib import *
# Unix 风格的路径区分大小写
path = PurePosixPath('C://','my_file.txt')
print(str(path))
```

程序执行结果为：

```
C:/my_file.txt
```

PurePath 类实例属性和实例方法

表 1 中罗列出了常用的以下 PurePath 类实例方法和属性。由于从本质上讲，PurePath 的操作对象是字符串，因此表 1 中的这些实例属性和实例方法，实质也是对字符串进行操作。

表 1 PurePath 类属性和方法

类实例属性和实例方法名	功能描述
PurePath.parts	返回路径字符串中所包含的各部分。
PurePath.drive	返回路径字符串中的驱动器盘符。
PurePath.root	返回路径字符串中的根路径。
PurePath.anchor	返回路径字符串中的盘符和根路径。
PurePath.parents	返回当前路径的全部父路径。
PurPath.parent	返回当前路径的上一级路径，相当于 parents[0] 的返回值。
PurePath.name	返回当前路径中的文件名。
PurePath.suffixes	返回当前路径中的文件所有后缀名。
PurePath.suffix	返回当前路径中的文件后缀名。相当于 suffixes 属性返回的列表的最后一个元素。
PurePath.stem	返回当前路径中的主文件名。
PurePath.as_posix()	将当前路径转换成 UNIX 风格的路径。
PurePath.as_uri()	将当前路径转换成 URL。只有绝对路径才能转换，否则将会引发 ValueError。
PurePath.is_absolute()	判断当前路径是否为绝对路径。
PurePath.joinpath(*other)	将多个路径连接在一起，作用类似于前面介绍的斜杠（/）连接符。
PurePath.match(pattern)	判断当前路径是否匹配指定通配符。
PurePath.relative_to(*other)	获取当前路径中去除基准路径之后的结果。
PurePath.with_name(name)	将当前路径中的文件名替换成新文件名。如果当前路径中没有文件名，则会引发 ValueError。
PurePath.with_suffix(suffix)	将当前路径中的文件后缀名替换成新的后缀名。如果当前路径中没有后缀名，则会添加新的后缀名。

对于表 1 中的这些实例属性和实例方法的用法，这里不再举例演示，有兴趣的读者可自行尝试它们的功能。

Path 类的功能和用法

和 PurPath 类相比，Path 类的最大不同，就是支持对路径的真实性进行判断。

从图 1 可以轻易看出，Path 是 PurePath 的子类，因此 Path 类除了支持 PurePath 提供的各种构造函数、实例属性以及实例方法之外，还提供甄别路径字符串有效性的方法，甚至还可以判断该路径对应的是文件还是文件夹，如果是文件，还支持对文件进行读写等操作。

和 PurePath 一样，Path 同样有 2 个子类，分别为 PosixPath（表示 UNIX 风格的路径）和 WindowsPath（表示 Windows 风格的路径）。

由于文章篇幅有限，Path 类属性和方法众多，因此这里不再一一进行讲解，后续章节用到时会进行详细的介绍。当然，感兴趣的读者可通过官方手册 https://docs.python.org/3/library/pathlib.html 进行查阅。

12.17 Python os.path 模块常见函数用法（实例+详细注释）

相比 pathlib 模块，os.path 模块不仅提供了一些操作路径字符串的方法，还包含一些或者指定文件属性的一些方法，如表 1 所示。

表 1 os.path 模块常用的属性和方法

方法	说明
os.path.abspath(path)	返回 path 的绝对路径。
os.path.basename(path)	获取 path 路径的基本名称，即 path 末尾到最后一个斜杠的位置之间的字符串。
os.path.commonprefix(list)	返回 list（多个路径）中，所有 path 共有的最长的路径。
os.path.dirname(path)	返回 path 路径中的目录部分。
os.path.exists(path)	判断 path 对应的文件是否存在，如果存在，返回 True；反之，返回 False。和 lexists() 的区别在于，exists()会自动判断失效的文件链接（类似 Windows 系统中文件的快捷方式），而 lexists() 却不会。
os.path.lexists(path)	判断路径是否存在，如果存在，则返回 True；反之，返回 False。
os.path.expanduser(path)	把 path 中包含的 "~" 和 "~user" 转换成用户目录。
os.path.expandvars(path)	根据环境变量的值替换 path 中包含的 "$name" 和 "${name}"。
os.path.getatime(path)	返回 path 所指文件的最近访问时间（浮点型秒数）。
os.path.getmtime(path)	返回文件的最近修改时间（单位为秒）。
os.path.getctime(path)	返回文件的创建时间（单位为秒，自 1970 年 1 月 1 日起（又称 Unix 时间））。
os.path.getsize(path)	返回文件大小，如果文件不存在就返回错误。
os.path.isabs(path)	判断是否为绝对路径。
os.path.isfile(path)	判断路径是否为文件。
os.path.isdir(path)	判断路径是否为目录。
os.path.islink(path)	判断路径是否为链接文件（类似 Windows 系统中的快捷方式）。
os.path.ismount(path)	判断路径是否为挂载点。
os.path.join(path1[, path2[, ...]])	把目录和文件名合成一个路径。
os.path.normcase(path)	转换 path 的大小写和斜杠。
os.path.normpath(path)	规范 path 字符串形式。
os.path.realpath(path)	返回 path 的真实路径。
os.path.relpath(path[, start])	从 start 开始计算相对路径。
os.path.samefile(path1, path2)	判断目录或文件是否相同。
os.path.sameopenfile(fp1, fp2)	判断 fp1 和 fp2 是否指向同一文件。
os.path.samestat(stat1, stat2)	判断 stat1 和 stat2 是否指向同一个文件。
os.path.split(path)	把路径分割成 dirname 和 basename，返回一个元组。
os.path.splitdrive(path)	一般用在 windows 下，返回驱动器名和路径组成的元组。
os.path.splitext(path)	分割路径，返回路径名和文件扩展名的元组。
os.path.splitunc(path)	把路径分割为加载点与文件。
os.path.walk(path, visit, arg)	遍历 path，进入每个目录都调用 visit 函数，visit 函数必须有 3 个参数(arg, dirname, names)，dirname 表示当前目录的目录名，names 代表当前目录下的所有文件名，args 则

	为 walk 的第三个参数。
os.path.supports_unicode_filenames	设置是否可以将任意 Unicode 字符串用作文件名。

下面程序演示了表 1 中部分函数的功能和用法：

```
from os import path
# 获取绝对路径
print(path.abspath("my_file.txt"))
# 获取共同前缀
print(path.commonprefix(['C://my_file.txt', 'C://a.txt']))
# 获取共同路径
print(path.commonpath(['http://c.biancheng.net/python/', 'http://c.biancheng.net/shell/']))
# 获取目录
print(path.dirname('C://my_file.txt'))
# 判断指定目录是否存在
print(path.exists('my_file.txt'))
```

程序执行结果为：

```
C:\Users\mengma\Desktop\my_file.txt
C://
http:\c.biancheng.net
C://
True
```

12.18 Python fnmatch 模块：用于文件名的匹配

fnmatch 模块主要用于文件名称的匹配，其能力比简单的字符串匹配更强大，但比使用正则表达式相比稍弱。。如果在数据处理操作中，只需要使用简单的通配符就能完成文件名的匹配，则使用 fnmatch 模块是不错的选择。

fnmatch 模块中，常用的函数及其功能如表 1 所示。

Python fnmatch 模块常用函数及功能

函数名	功能
fnmatch.filter(names, pattern)	对 names 列表进行过滤，返回 names 列表中匹配 pattern 的文件名组成的子集合。
fnmatch.fnmatch(filename, pattern)	判断 filename 文件名，是否和指定 pattern 字符串匹配
fnmatch.fnmatchcase(filename, pattern)	和 fnmatch() 函数功能大致相同，只是该函数区分大小写。
fnmatch.translate(pattern)	将一个 UNIX shell 风格的 pattern 字符串，转换为正则表达式

fnmatch 模块匹配文件名的模式使用的就是 UNIX shell 风格，其支持使用如下几个通配符：

- *：可匹配任意个任意字符。
- ？：可匹配一个任意字符。
- [字符序列]：可匹配中括号里字符序列中的任意字符。该字符序列也支持中画线表示法。比如 [a-c] 可代表 a、b 和 c 字符中任意一个。
- [!字符序列]：可匹配不在中括号里字符序列中的任意字符。

例如，下面程序演示表 1 中一些函数的用法及功能：

```
1.  import fnmatch
2.
3.  #filter()
4.  print(fnmatch.filter(['dlsf', 'ewro.txt', 'te.py', 'youe.py'], '*.txt'))
5.
6.  #fnmatch()
7.  for file in ['word.doc','index.py','my_file.txt']:
8.      if fnmatch.fnmatch(file,'*.txt'):
9.          print(file)
10.
11. #fnmatchcase()
12. print([addr for addr in ['word.doc','index.py','my_file.txt','a.TXT'] if fnmatch.fnmatchcase(addr, '*.txt')])
13.
14. #translate()
15. print(fnmatch.translate('a*b.txt'))
```

程序执行结果为：

```
['ewro.txt']
my_file.txt
['my_file.txt']
(?s:a.*b\.txt)\Z
```

12.19 Python os 模块详解

除前面章节介绍的各种函数之外，os 模块还提供了大量操作文件和目录的函数，本节将介绍 os 模块下常用的函数。

> 如果读者需要查阅有关这些函数的说明，则可访问 https://docs.python.org/3/library/os.html 页面。

os 模块与目录相关的函数

与目录相关的函数如下：

- os.getcwd()：获取当前目录。
- os.chdir(path)：改变当前目录。
- os.fchdir(fd)：通过文件描述利改变当前目录。该函数与上一个函数的功能基本相似，只是该函数以文件描述符作为参数来代表目录。

下面程序测试了与目录相关的函数的用法：

```
import os

# 获取当前目录
print(os.getcwd())  # G:\publish\codes\12.7
# 改变当前目录
os.chdir('../12.6')
# 再次获取当前目录
print(os.getcwd())  # G:\publish\codes\12.6
```

上面程序示范了使用 getcwd() 来获取当前目录，也示范了使用 chdir() 来改变当前目录。

- os.chroot(path)：改变当前进程的根目录。
- os.listdir(path)：返回 path 对应目录下的所有文件和子目录。
- os.mkdir(path[, mode])：创建 path 对应的目录，其中 mode 用于指定该目录的权限。该 mode 参数代表一个 UNIX 风格的权限，比如 0o777 代表所有者可读/可写/可执行、组用户可读/可写/可执行、其他用户可读/可写/可执行。
- os.makedirs(path[, mode])：其作用类似于 mkdir()，但该函数的功能更加强大，它可以边归创建目录。比如要创建 abc/xyz/wawa 目录，如果在当前目录下没有 abc 目录，那么使用 mkdir() 函数就会报错，而使用 makedirs() 函数则会先创建 abc，然后在其中创建 xyz 子目录，最后在 xyz 子目录下创建 wawa 子目录。

如下程序示范了如何创建目录：

```
import os
path = 'my_dir'
# 直接在当前目录下创建目录
os.mkdir(path, 0o755)
path = "abc/xyz/wawa"
# 递归创建目录
os.makedirs(path, 0o755)
```

正如从上面代码所看到的，直接在当前目录下创建 mydir 子目录，因此可以使用 mkdir() 函数创建；需要程序递归创建 abc/xyz/wawa 目录，因此使用 makedirs() 函数。

- os.rmdir(path)：删除 path 对应的空目录。如果目录非空，则抛出一个 OSError 异常。程序可以先用 os.remove() 函数删除文件。
- os.removedirs(path)：边归删除目录。其功能类似于 rmdir()，但该函数可以递归删除 abc/xyz/wawa 目录，它会从 wawa 子目录开始删除，然后删除 xyz 子目录，最后删除 abc 目录。

如下程序示范了如何删除目录：

```
1.  import os
2.
3.  path = 'my_dir'
4.  # 直接删除当前目录下的子目录
5.  os.rmdir(path)
6.  path = "abc/xyz/wawa"
7.  # 递归删除子目录
8.  os.removedirs(path)
```

上面程序中第 5 行代码使用 rmdir() 函数删除当前目录下的 my_dir 子目录，该函数不会执行递归删除；第 8 行代码使用 removedirs() 函数删除 abc/xyz/wawa 目录，该函数会执行递归删除，它会先删除 wawa 子目录，然后删除 xyz 子目录，最后才删除 abc 目录。

- os.rename(src, dst)：重命名文件或目录，将 src 重名为 dst。
- os.renames(old, new)：对文件或目录进行递归重命名。其功能类似于 rename()，但该函数可以递归重命名 abc/xyz/wawa 目录，它会从 wawa 子目录开始重命名，然后重命名 xyz 子目录，最后重命名 abc 目录。

如下程序示范了如何重命名目录：

```
1.  import os
2.
3.  path = 'my_dir'
4.  # 直接重命名当前目录下的子目录
5.  os.rename(path, 'your_dir')
6.  path = "abc/xyz/wawa"
7.  # 递归重命名子目录
8.  os.renames(path, 'foo/bar/haha')
```

上面程序中第 5 行代码直接重命名当前目录下的 my_dir 子目录，程序会将该子目录重命名为 your_dir；第 8 行代码则执行递归重命名，程序会将 wawa 重命名为 haba，将 xyz 重命名为 bar，将 abc 重命名为 foo。

os 模块与权限相关的函数

与权限相关的函数如下：

- os.access(path, mode)：检查 path 对应的文件或目录是否具有指定权限。该函数的第二个参数可能是以下四个状态值的一个或多个值：
- os.F_OK：判断是否存在。
- os.R_OK：判断是否可读。
- os.W_OK：判断是否可写。

- os.X_OK：判断是否可执行。

例如如下程序：

```
import os

# 判断当前目录的权限
ret = os.access('.', os.F_OK|os.R_OK|os.W_OK|os.X_OK)
print("os.F_OK|os.R_OK|os.W_OK|os.X_OK - 返回值:", ret)
# 判断 os.access_test.py 文件的权限
ret = os.access('os.access_test.py', os.F_OK|os.R_OK|os.W_OK)
print("os.F_OK|os.R_OK|os.W_OK - 返回值:", ret)
```

上面程序判断当前目录的权限和当前文件的权限，这里特意将此文件设为只读的。运行该程序，可以看到如下输出结果：

```
os.F_OK|os.R_OK|os.W_OK|os.X_OK - 返回值：True
os.F_OK|os.R_OK|os.W_OK - 返回值：False
```

os.chrnod(path, mode)：更改权限。其中 mode 参数代表要改变的权限，该参数支持的值可以是以下一个或多个值的组合：

- stat.S_IXOTH：其他用户有执行权限。
- stat.S_IWOTH：其他用户有写权限。
- stat.S_TROTH：其他用户有读权限。
- stat.S_IRWXO：其他用户有全部权限。
- stat.S_IXGRP：组用户有执行权限。
- stat.S_IWGRP：组用户有写权限。
- stat.S_IRGRP：组用户有读权限。
- stat.S_IRWXG：组用户有全部权限。
- stat.S_IXUSR：所有者有执行权限。
- stat.S_IWUSR：所有者有写权限。
- stat.S_IRUSR：所有者有读权限。
- stat.S_IRWXU：所有者有全部权限。
- stat.S_IREAD：Windows 将该文件设为只读的。
- stat.S_IWRITE：Windows 将该文件设为可写的。

前面的那些权限都是 UNIX 文件系统下有效的概念，UNIX 文件系统下的文件有一个所有者，跟所有者处于同一组的其他用户被称为组用户。因此在 UNIX 文件系统下允许为不同用户分配不同的权限。

例如如下程序：

```
import os, stat

# 将 os.chmod_test.py 文件改为只读
os.chmod('os.chmod_test.py', stat.S_IREAD)
```

```
# 判断是否可写
ret = os.access('os.chmod_test.py', os.W_OK)
print("os.W_OK - 返回值:", ret)
```

运行上面程序后，os.chmod_test.py 变成只读文件。

os.chown(path, uid, gid)：更改文件的所有者。其中 uid 代表用户 id，gid 代表组 id。该命令主要在 UNIX 文件系统下有效。

os.fchmod(fd, mode)：改变一个文件的访问权限，该文件由文件描述符 fd 指定。该函数的功能与 os.chmod() 函数的功能相似，只是该函数使用 fd 代表文件。

os.fchown(fd, uid, gid)：改变文件的所有者，该文件由文件描述符 fd 指定。该函数的功能与 os.chown() 函数的功能相似，只是该函数使用 fd 代表文件。

os 模块与文件访问相关的函数

与文件访问相关的函数如下：

- os.open(file, flags[, mode])：打开一个文件，并且设置打开选项，mode 参数是可选的。该函数返回文件描述符。其中 flags 代表打开文件的旗标，它支持如下一个或多个选项：
- os.O_RDONLY：以只读的方式打开。
- os.O_WRONLY：以只写的方式打开。
- os.O_RDWR：以读写的方式打开。
- os.O_NONBLOCK：打开时不阻塞。
- os.O_APPEND：以追加的方式打开。
- os.O_CREAT：创建并打开一个新文件。
- os.O_TRUNC：打开一个文件并截断它的长度为 0（必须有写权限）。
- os.O_EXCL：在创建文件时，如果指定的文件存在，则返回错误。
- os.O_SHLOCK：自动获取共享锁。
- os.O_EXLOCK：自动获取独立锁。
- os.O_DIRECT：消除或减少缓存效果。
- os.O_FSYNC：同步写入。
- os.O_NOFOLLOW：不追踪软链接。
- os.read(fd, n)：从文件描述符 fd 中读取最多 n 个字节，返回读到的字符串。如果文件描述符副对应的文件已到达结尾，则返回一个空字节串。
- os.write(fd, str)：将字节串写入文件描述符 fd 中，返回实际写入的字节串长度。
- os.close(fd)：关闭文件描述符 fd。
- os.lseek(fd, pos, how)：该函数同样用于移动文件指针。其中 how 参数指定从哪里开始移动，如果将 how 设为 0 或 SEEK_SET，则表明从文件开头开始移动；如果将 how 设为 1 或 SEEK_CUR，则表明从文件指针当前位置开始移动；如果将 how 设为 2 或 SEEK_END，则表明从文件结束处开始移动。上面几个函数同样可用于执行文件的读写，程序通常会先通过 os.open() 打开文件，然后调用 os.read()、os.write() 来读写文件，当操作完成后通过 os.close() 关闭文件。

如下程序示范了使用上面的函数来读写文件：

```
import os

# 以读写、创建方式打开文件
f = os.open('abc.txt', os.O_RDWR|os.O_CREAT)
# 写入文件内容
```

```
6.  len1 = os.write(f, '水晶潭底银鱼跃，\n'.encode('utf-8'))
7.  len2 = os.write(f, '清徐风中碧竿横。\n'.encode('utf-8'))
8.  # 将文件指针移动到开始处
9.  os.lseek(f, 0, os.SEEK_SET)
10. # 读取文件内容
11. data = os.read(f, len1 + len2)
12. # 打印读取到字节串
13. print(data)
14. # 将字节串恢复成字符串
15. print(data.decode('utf-8'))
16. os.close(f)
```

上面程序中，第 6、7 行代码用于向所打开的文件中写入数据；第 11 行代码用于读取文件内容。

os.fdopen(fd[, mode[, bufsize]])：通过文件描述符 fd 打开文件，并返回对应的文件对象。
os.closerange(fd_low, fd_high)：关闭从 fd_low（包含）到 fd_high（不包含）范围的所有文件描述符。
os.dup(fd)：复制文件描述符。
os.dup2(fd,fd2)：将一个文件描述符 fd 复制到另一个文件描述符 fd2 中。
os.ftruncate(fd, length)：将 fd 对应的文件截断到 length 长度，因此此处传入的 length 参数不应该超过文件大小。
os.remove(path)：删除 path 对应的文件。如果 path 是一个文件夹，则抛出 OSError 错误。如果要删除目录，则使用 os.rmdir()。
os.link(src, dst)：创建从 src 到 dst 的硬链接。硬链接是 UNIX 系统的概念，如果在 Windows 系统中就是复制目标文件。
os.symlink(src, dst)：创建从 src 到 dst 的符号链接，对应于 Windows 的快捷方式。

> 由于 Windows 权限的缘故，因此必须以管理员身份执行 os.symlink() 函数来创建快捷方式。

下面程序示范了在 Windows 系统中使用 os.symlink(src, dst) 函数来创建快捷方式：

```
1.  import os
2.
3.  # 为 os.link_test.py 文件创建快捷方式
4.  os.symlink('os.link_test.py', 'tt')
5.  # 为 os.link_test.py 文件创建硬连接（Windows 上就是复制文件）
6.  os.link('os.link_test.py', 'dst')
```

上面程序使用 symlink() 函数为指定文件创建符号链接，在 Windows 系统中就是创建快捷方式；使用 link() 函数创建硬链接，在 Windows 系统中就是复制文件。

运行上面程序，将会看到程序在当前目录下创建了一个名为“tt”的快捷方式，并将 os.link_test.py 文件复制为 dst 文件。

12.20 Python tempfile 模块：生成临时文件和临时目录

tempfile 模块专门用于创建临时文件和临时目录，它既可以在 UNIX 平台上运行良好，也可以在 Windows 平台上运行良好。

tempfile 模块中常用的函数，如表 1 所示。

表 1 tempfile 模块常用函数及功能

tempfile 模块函数	功能描述
tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	创建临时文件。该函数返回一个类文件对象，也就是支持文件 I/O。
tempfile.NamedTemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True)	创建临时文件。该函数的功能与上一个函数的功能大致相同，只是它生成的临时文件在文件系统中有文件名。
tempfile.SpooledTemporaryFile(max_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	创建临时文件。与 TemporaryFile 函数相比，当程序向该临时文件输出数据时，会先输出到内存中，直到超过 max_size 才会真正输出到物理磁盘中。
tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None)	生成临时目录。
tempfile.gettempdir()	获取系统的临时目录。
tempfile.gettempdirb()	与 gettempdir() 相同，只是该函数返回字节串。
tempfile.gettempprefix()	返回用于生成临时文件的前缀名。
tempfile.gettempprefixb()	与 gettempprefix() 相同，只是该函数返回字节串。

提示：表中有些函数包含很多参数，但这些参数都具有自己的默认值，因此如果没有特殊要求，可以不对其传参。

tempfile 模块还提供了 tempfile.mkstemp() 和 tempfile.mkdtemp() 两个低级别的函数。上面介绍的 4 个用于创建临时文件和临时目录的函数都是高级别的函数，高级别的函数支持自动清理，而且可以与 with 语句一起使用，而这两个低级别的函数则不支持，因此一般推荐使用高级别的函数来创建临时文件和临时目录。

此外，tempfile 模块还提供了 tempfile.tempdir 属性，通过对该属性赋值可以改变系统的临时目录。

下面程序示范了如何使用临时文件和临时目录：

```
import tempfile

# 创建临时文件
fp = tempfile.TemporaryFile()
print(fp.name)
fp.write('两情若是久长时，'.encode('utf-8'))
fp.write('又岂在朝朝暮暮。'.encode('utf-8'))
# 将文件指针移到开始处，准备读取文件
fp.seek(0)
print(fp.read().decode('utf-8')) # 输出刚才写入的内容
# 关闭文件，该文件将会被自动删除
```

```
fp.close()

# 通过 with 语句创建临时文件，with 会自动关闭临时文件
with tempfile.TemporaryFile() as fp:
    # 写入内容
    fp.write(b'I Love Python!')
    # 将文件指针移到开始处，准备读取文件
    fp.seek(0)
    # 读取文件内容
    print(fp.read()) # b'I Love Python!'

# 通过 with 语句创建临时目录
with tempfile.TemporaryDirectory() as tmpdirname:
    print('创建临时目录', tmpdirname)
```

上面程序以两种方式来创建临时文件：

1. 第一种方式是手动创建临时文件，读写临时文件后需要主动关闭它，当程序关闭该临时文件时，该文件会被自动删除。
2. 第二种方式则是使用 with 语句创建临时文件，这样 with 语句会自动关闭临时文件。

上面程序最后还创建了临时目录。由于程序使用 with 语句来管理临时目录，因此程序也会自动删除该临时目录。

运行上面程序，可以看到如下输出结果：

```
C:\Users\admin\AppData\Local\Temp\tmphvehw9z1
两情若是久长时，又岂在朝朝暮暮。
b'I Love Python!'
创建临时目录 C:\Users\admin\AppData\Local\Temp\tmp3sjbnwob
```

上面第一行输出结果就是程序生成的临时文件的文件名，最后一行输出结果就是程序生成的临时目录的目录名。需要注意的是，不要去找临时文件或临时文件夹，因为程序退出时该临时文件和临时文件夹都会被删除。