第7章. Python的运行时服务

在计算机系统中，直接与硬件打交道的代码集合被称为“操作系统”。用编译语言编写的程序需要使用操作系统提供的API，也就是所谓的“系统调用（system calls）”。然而不同操作系统提供的系统调用存在很大差异，所以用编译语言开发一款跨平台应用的工作量是很大的。解释语言的一个优势是用解释器屏蔽了不同操作系统之间的差异（解释器自身一般用编译语言开发），使得开发跨平台应用变得相当容易。

作为一门解释语言，Python同样具有这一优势，只要你不使用与特定操作系统绑定的模块（例如posix和winreg），那么编写出的Python脚本将能够完美跨平台。（但有些通用模块依然具有一些仅在特定操作系统上才有效的标识符，使用时需特别处理。）Python脚本在运行时仅使用由Python解释器提供的服务。在计算机科学的术语中，这些服务被统称为“运行时服务（runtime services）”。本章将讨论Python解释器提供的运行时服务，它们构成了Python脚本的运行环境。

7-1. sys模块和builtins模块

（语言参考手册：9.1）
（标准库：sys、builtins）

第6章已经提到，Python解释器启动时会自动为一部分内置模块创建模块对象。这些内置模块中最重要的是sys模块和builtins模块。可以这样理解：sys模块是Python解释器的核心部分，实现了Python的语法并保证了运行逻辑；而builtins模块则是Python解释器中对应内置函数、内置常量、内置类型和内置异常的部分，实现了内置名字空间。

虽然对应这两个模块的模块对象总是存在的，但必须通过import语句导入sys模块和builtins模块才能获得引用它们的标识符，进而访问这些模块的属性。因此，sys和builtins也被列入了标准库，但本质上是访问内置模块的接口。（事实上，所有内置模块在标准库中都有对应接口。）

sys模块提供了大量属性，是Python运行时服务的核心模块。第6章已经介绍了与导入模块相关的sys属性，而对于其他sys属性的介绍将分布在本书剩余章节中。

本节介绍与描述和定制Python解释器行为相关的sys属性。

表7-1. Python解释器的实现信息相关sys属性

属性	说明
sys.implementation	一个包含解释器实现信息的对象，该对象必须具有如下属性： name：解释器的名称。 cache_tag：导入机制使用的标记。 version：与sys.version_info相同。 hexversion：与sys.hexversion相同。 该对象还可以额外具有一些以“_”开头的属性。

属性	说明
sys.version_info	一个具名元组，用于给出版本信息，包含如下五个元素： major：主版本号。 minor：次版本号。 micro：微版本号。 releaselevel：发行级别。 serial：序列号。
sys.hexversion	一个代表内部版本号的十六进制数字。
sys.api_version	一个数字，为该解释器提供的C API的版本号。
sys.version	一个字符串，为交互式启动解释器时显示的版本信息。
sys.copyright	一个字符串，为解释器的版权声明。

表7-1中列出的sys属性使Python脚本可以知道自己在被什么样的Python解释器执行。注意sys.version_info引用的“具名元组（named tuples）”是一种从元组派生出的类型，与元组的区别在于其内的元素除了可以通过索引指定外还可以通过名字指定，但本书不详细讨论。请通过如下命令行和语句验证上述sys属性：

```
      $ python3

      >>> import sys
      >>> sys.implementation
      namespace(name='cpython', cache_tag='cpython-311',
version=sys.version_info(major=3, minor=11, micro=0, releaselevel='beta',
serial=3), hexversion=51052723, _multiarch='darwin')
      >>> sys.version_info
      sys.version_info(major=3, minor=11, micro=0, releaselevel='beta', serial=3)
      >>> sys.hexversion
      51052723
      >>> sys.api_version
      1013
      >>> sys.version
      '3.11.0b3 (v3.11.0b3:eb0004c271, Jun  1 2022, 10:03:01) [Clang 13.0.0
(clang-1300.0.29.30)]'
      >>> sys.copyright
      'Copyright (c) 2001-2022 Python Software Foundation.\nAll Rights Reserved.
\n\nCopyright (c) 2000 BeOpen.com.\nAll Rights Reserved.\n\nCopyright (c)
1995-2001 Corporation for National Research Initiatives.\nAll Rights Reserved.
\n\nCopyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.\nAll
Rights Reserved.'
      >>>
```

表7-2. 平台信息相关sys属性

属性	说明
sys.platform	一个字符串，代表平台的操作系统类型。
sys.byteorder	“big”或“little”，分别代表大端法和小端法，对应平台的字节序。
sys.maxsize	一个整数，为索引的最大取值。通常为平台的字长减去1。

属性	说明
sys.maxunicode	一个整数，为平台支持的Unicode字符的最大编码。
sys.thread_info	一个具名元组，用于描述该平台的线程实现方式，包含如下三个元素： name：线程类型。 lock：锁的实现机制。 version：线程库的名称和版本。
sys.base_prefix	一个字符串，为一个绝对路径，指向的目录在安装Python时被用于储存所有平台无关文件。
sys.prefix	与sys.base_prefix在逻辑上相同，但能够用于描述虚拟环境。
sys.base_exec_prefix	一个字符串，为一个绝对路径，指向的目录在安装Python时被用于储存所有平台相关文件。
sys.exec_prefix	与sys.base_exec_prefix在逻辑上相同，但能够用于描述虚拟环境。
sys.platlibdir	一个字符串，为一个相对路径，与sys.prefix和sys.exec_prefix合在一起构成绝对路径，指向的目录被用于储存标准库和第三方分发包。
sys.executable	一个字符串，为一个绝对路径，指向Python解释器对应的可执行文件。

表7-2中列出的sys属性使Python脚本可以知道自己在什么样的平台上被执行。sys.platform说明了该平台的操作系统类型，但通常只引用一个单词，例如“aix”、“freebsd”、“linux”、“win32”、“darwin”和“cygwin”。sys.byteorder说明该平台的字节序。sys.maxsize限制了容器索引的范围。sys.maxunicode给出了该平台能够表示的Unicode字符的范围。这些都很容易理解，不需要额外解释。

sys.thread_info引用的具名元组中，用于说明线程类型的name元素可能引用如下字符串：

- “pthread”：POSIX线程。
- “nt”：Windows线程。
- “solaris”：Solaris线程。

而每种线程类型都可以支持多种线程锁的实现机制，因此需要通过lock元素说明，它可能引用如下字符串：

- “semaphore”：用信号量实现锁。
- “mutex+cond”：用互斥量和条件实现锁。

如果遇到其他实现机制，或者该信息未知，则lock元素引用None。如果线程库的名称和版本信息未知，则version元素引用None；否则它会引用给出代表该信息的字符串。

剩下的6个sys属性共同给出了Python在该平台上的安装位置，其中sys.base_prefix和sys.base_exec_prefix引用的绝对路径是Python的真实安装目录，在很多平台上都是同一个目录，且分别是sys.prefix和sys.exec_prefix的默认值。仅当创建了虚拟环境时（在第17章讨

论），sys.prefix和sys.exec_prefix引用的绝对路径才可能指向其他目录。sys.platlibdir引用的相对路径被附加在上述4个sys属性引用的绝对路径之后，给出了标准库和第三方分发包的安装位置，这些位置将影响sys.path的默认值。而sys.executable引用的绝对路径应指向sys.base_prefix指定目录中的某个可执行文件，启动Python解释器等价于执行该文件。如果该信息无法获得，那么sys.executable将引用空串或None。

请通过如下语句验证上述sys属性：

```
>>> import sys
>>> sys.platform
'darwin'
>>> sys.byteorder
'little'
>>> sys.maxsize
9223372036854775807
>>> sys.maxunicode
1114111
>>> sys.thread_info
sys.thread_info(name='pthread', lock='mutex+cond', version=None)
>>> sys.base_prefix
'/Library/Frameworks/Python.framework/Versions/3.11'
>>> sys.prefix
'/Library/Frameworks/Python.framework/Versions/3.11'
>>> sys.base_exec_prefix
'/Library/Frameworks/Python.framework/Versions/3.11'
>>> sys.exec_prefix
'/Library/Frameworks/Python.framework/Versions/3.11'
>>> sys.platlibdir
'lib'
>>> sys.executable
'/Library/Frameworks/Python.framework/Versions/3.11/bin/python3'
>>>
```

表7-3. 针对特定平台的补充信息相关sys属性

属性	说明
sys.abiflags	一个字符串，为通过源代码编译生成Python解释器时使用的ABI标签。仅适用于POSIX系统（Unix和类Unix操作系统，下同）。
sys.getwindowsversion()	返回一个具名元组，用于提供Windows的详细版本信息。仅适用于Windows。
sys.dllhandle	一个整数，为指向Python的Windows句柄。仅适用于Windows。
sys.winver	一个字符串，为Python在注册表中的键。仅适用于Windows。
sys.getandroidapilevel()	一个整数，为Android的API的版本。仅适用于Android。

表7-3中列出的sys属性是对表7-2中列出的sys属性的补充，仅能在特定平台上使用。

表7-4. 字符编码转换相关sys属性

属性	说明
sys.getdefaultencoding()	返回一个字符串，为解释器当前使用的Unicode编码方式。
sys.getfilesystemencoding()	返回一个字符串，为文件系统的字符编码方式。
sys.getfilesystemencodeerrors()	返回一个字符串，为文件系统编码错误处理方式。
sys._enablelegacywindowsfsencoding()	将文件系统的字符编码方式修改为“mbcs”，编码错误处理方式修改为“replace”。仅适用于Windows。

表7-4列出的sys属性与Python解释器和平台的文件系统之间传输数据时的字符编码转换有关。Python解释器总是使用Unicode字符集，但Unicode字符集有五种编码方式：utf-8、utf-16le、utf-16be、utf-32le和utf-32be。可以通过sys.getdefaultencoding()取得解释器当前使用的Unicode编码方式。而平台的文件系统则不一定采用Unicode字符集，更不用说编码方式，故需通过sys.getfilesystemencoding()来了解。

当遇到无法用当前编码方式编码的字符时，有多种处理方式，例如将它们替换为转义序列，或将它们替换为代表替换标记的特殊字符。sys.getfilesystemencodeerrors()使Python脚本可以知道平台的文件系统具体使用了哪种处理方式。

一般而言Python解释器会自动收集平台的文件系统的字符编码方式和编码错误处理方式相关信息。从Python 3.6开始，我们可以通过调用sys._enablelegacywindowsfsencoding()强制Python解释器认为文件系统的字符编码方式为“mbcs”，错误处理方式是“replace”。但这一技巧只能在Windows上使用。

请用如下语句验证上述前3个sys属性：

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

表7-5. 解释器行为控制相关sys属性

属性	说明
sys.dont_write_bytecode	一个布尔值，表示是否不生成.pyc文件。
sys.pycache_prefix	None或一个字符串，当引用字符串时将在该字符串对应相对路径指定的目录下生成.pyc文件。
sys.getrecursionlimit()	返回一个整数，代表当前递归限制。
sys.setrecursionlimit()	设置递归限制。

属性	说明
sys.getswitchinterval()	返回一个浮点数，代表当前线程切换间隔。
sys.setswitchinterval()	设置线程切换间隔。
sys.getdlopenflags()	返回一个整数，代表调用dlopen()时的标志位设置。仅适用于POSIX系统。
sys.setdlopenflags()	设置调用dlopen()时的标志位。仅适用于POSIX系统。

表7-5列出的sys属性用于控制Python解释器的行为细节。注意它们对于解释器的影响都是临时性的，在解释器重启后就会失效。

首先考虑.pyc文件。第6章已经说明，在默认情况下Python解释器在第一次加载一个模块时，会在该模块对应的Python脚本所在目录下的“__pycache__”子目录内创建相应的.pyc文件（如果__pycache__目录不存在则自动创建）。以后再次加载该模块时，会比较Python脚本与对应的.pyc文件的创建日期，如果后者晚于前者则直接使用.pyc文件；否则重新编译Python脚本并更新.pyc文件。这一机制可以优化Python解释器的运行速度，此外.pyc文件改名后还可以作为该模块的无源发行版。

可用于改变上述默认行为的sys属性有两个，即sys.dont_write_bytecode和sys.pycache_prefix。sys.dont_write_bytecode引用一个布尔值，根据启动Python解释器时是否添加了-B选项确定：如果没有添加-B，则引用False，意味着会创建.pyc文件；否则引用True，意味着不会创建.pyc文件。请通过下面的命令行和语句验证：

```
$ python3

>>> import sys
>>> sys.dont_write_bytecode
False
>>> ^D

$ python3 -B

>>> import sys
>>> sys.dont_write_bytecode
True
>>>
```

而sys.dont_write_bytecode是可写的，因此我们可以动态地改变其引用的布尔值。请将第6章讨论包时作为例子创建的dir1目录及其子目录下的所有__pycache__子目录删除，然后通过下面的命令行和语句验证当sys.dont_write_bytecode引用True时，导入模块不会创建.pyc文件：

```
$ python3

>>> import sys
>>> sys.dont_write_bytecode = True
>>> import dir1.dir4
>>>
```

注意在执行完了“import dir1.dir4”之后，dir1目录及其子目录内没有__pycache__目录，也没有.pyc文件。

sys.pycache_prefix则默认引用None，但在启动Python解释器时可以通过-X选项将其设置为一个代表相对路径的字符串，即“`-X pycache_prefix=`*`path`*”。请通过如下命令行和语句验证：

```
$ python3

>>> import sys
>>> sys.pycache_prefix
>>> ^D

$ python3 -X pycache_prefix=caches

>>> import sys
>>> sys.pycache_prefix
'caches'
>>>
```

此外，sys.pycache_prefix同样是可写的，因此我们能动态改变它引用的对象。

当sys.pycache_prefix引用None时，解释器会如上述那样自动创建__pycache__目录并在其下创建.pyc文件。但当sys.pycache_prefix引用一个代表相对路径的字符串，行为就变成了在工作目录下寻找该相对路径指定的目录，如果没找到则自动创建该目录，然后在该目录下创建一棵子目录树，该子目录树为包含所有相关Python脚本的子目录树的拷贝，但只拷贝目录不拷贝文件，此后.pyc文件将在这棵拷贝子目录树内创建，且与相应Python脚本的位置一一对应。

请通过如下命令行和语句验证：

```
$ python3

>>> import sys
>>> sys.pycache_prefix = 'caches'
>>> import dir1.dir4
>>>
```

请检查工作目录下新创建的caches目录，以及其内的拷贝子目录树和所有.pyc文件。

除了sys.dont_write_bytecode和sys.pycache_prefix之外，表7-5中列出的剩余6个sys属性都较容易理解。它们都是成对出现的。sys.getrecursionlimit()返回的整数其实是函数栈的最大深度，因此理论上也是递归的最大层数。（但当递归调用在外层函数调用之内时，由于函数栈已经不为空，所以递归的层数上限会减小。）sys.setrecursionlimit()则用于设置函数栈的最大深度，其语法为：

```
sys.setrecursionlimit(limit)
```

其limit参数需被传入一个整数。

sys.getswitchinterval()返回的浮点数是Python解释器规定的线程切换的最小时间间隔，其单位是秒。需要强调的是，线程切换其实是由操作系统负责调度的，因此Python解释器只能保证两次线程切换的时间间隔不小于该值，但却不能保证恰好是该值。
sys.setswitchinterval()则用于设置该最小时间间隔，其语法为：

```
sys.setswitchinterval(interval)
```

其interval参数需被传入一个浮点数。

sys.getdlopenflags()返回的整数为通过C库函数dlopen()打开动态链接库文件时的标志位设置。这些标志位会影响dlopen()的行为，可通过标准库中的os模块定义的常量来分析，但这超出了本书的范围。sys.setdlopenflags()则用于改变dlopen()标志位设置，其语法为：

```
sys.setdlopenflags(n)
```

其n参数需被传入一个整数。由于只有Unix和类Unix操作系统中才有dlopen()，Windows中访问动态链接库的对应API是LoadLibrary，因此这两个sys属性只能在Unix和类Unix操作系统上使用。

请通过如下命令行和语句验证（省略 sys.getdlopenflags()和 sys.setdlopenflags()）：

```
$ python3

>>> import sys
>>> sys.getdlopenflags()
2
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(500)
>>> sys.getrecursionlimit()
500
>>> sys.getswitchinterval()
0.005
>>> sys.setswitchinterval(0.01)
>>> sys.getswitchinterval()
0.01
>>>
```

从逻辑上讲，builtins模块也提供了大量的属性，然而几乎没有情况需要访问它们，因为这些属性其实就是内置函数、内置常量、内置类型和内置异常，可以直接使用。举例来说，

builtins.type引用的就是内置函数type()，在绝大多数情况下不需要使用builtins.type来访问它。下面的例子有一定程度生造的意味：

```
$ python3

>>> import builtins
>>> def type(obj):
...     print("wrapper of built-in type()")
...     return builtins.type(obj)
...
>>> type(1)
wrapper of built-in type()
<class 'int'>
>>>
```

值得一提的是，CPython创建模块对象时，几乎总是会自动为其添加__builtins__属性，该属性引用的正是builtins模块。这使得在这些模块内可以访问内置函数、内置常量、内置类型和内置异常。换句话说，__builtins__属性是连接到内置名字空间的桥梁。不过该属性是CPython的实现细节，并不通用，也不保证将来不会被其他更好的机制取代。

7-2. 主模块和顶层环境

（语言参考手册：5.8、9）
（标准库：__main__）

第6章已经提到了，Python解释器启动后必然会创建主模块，其__name__特殊属性会引用“__main__”。与sys模块和builtins模块不同，主模块不属于内置模块。尽管如此，由于主模块是自动创建的，所以必须导入标准库中的__main__模块才能获得引用主模块的标识符“__main__”。但主模块的一切属性都是全局变量，可以直接访问，所以没有任何情况必须导入__main__模块。

第2章列出了CPython的各种启动方式，不同启动方式执行的Python代码来源不同，而这同样也决定了主模块基于什么创建，下面重新总结：

- 交互模式启动：创建一个空的主模块，然后每获得一条语句就执行一条。
- 非交互模式启动，给出了指向某个Python脚本的路径：将该Python脚本对应的模块创建为主模块。
- 非交互模式启动，通过-m指定了某个模块：将该模块创建为主模块。
- 非交互模式启动，通过-c指定了一段代码：创建一个空的主模块，将指定的代码填入，然后执行。

除了上面4种标准启动方式外还存在一些变体，例如2、3和4可以通过结合-i选项在执行完指定代码后进入交互模式；通过添加Shebang转化为shell脚本的Python脚本被直接执行时等价于2。但从逻辑上只需要考虑这4种标准启动方式。

上述4种标准启动方式中，仅3（即使用-m选项）会使被创建的主模块的__spec__实例属性引用一个模块规格说明，其他方式都会导致主模块的__spec__引用None。请创建一个空的test.py文件，然后通过如下命令行和语句验证（该例子有意通过“__main__”标识符来访问全局变量__spec__）：

```
$ python3 -i test.py

>>> import __main__
>>> type(__main__.__spec__)
<class 'NoneType'>
>>> ^D

$ python3 -i -m test

>>> import __main__
>>> type(__main__.__spec__)
<class '_frozen_importlib.ModuleSpec'>
>>>
```

第6章已经在讨论导入模块的细节时指出，模块对象的创建和初始化是分开的，分别由加载器的create_module属性和exec_module属性实现。这使得不论通过哪种方式启动Python解释器，都会先创建一个空的主模块，只不过2、3和4会在初始化过程中将代码一次性填入主模块，而1则在初始化完成后才将代码持续性地填入主模块。创建主模块的目的是提供所谓的“顶层环境（top-level environment）”。

顶层环境的特殊性在于其内的代码可以通过表7-6列出的sys属性来处理Python程序的启动、交互和终止。

表7-6. 启动、交互和终止相关sys属性

属性	说明
sys.argv	一个列表，为解析后的命令行参数。
sys.orig_argv	一个列表，为解析后的命令行。
sys.flags	一个具名元组，为解释器的状态标志。
sys.__interactivehook__	一个无参数的可调用对象，当解释器以交互模式启动时自动调用。
sys.ps1	一个字符串，作为解释器交互模式的主提示符。
sys.ps2	一个字符串，作为解释器交互模式的子提示符。
sys.is_finalizing()	返回一个布尔值，表明解释器是否正在关闭。
sys.exit()	使解释器退出，可通过arg参数设置退出状态。

sys.argv用于获得启动Python解释器时给出的命令行参数。事实上“argv”是“参数向量（argument vector）”的意思，源自C中main()函数的argv参数，因此sys.argv[0]也按照惯例为被执行Python脚本的相关信息。但由于Python解释器有4种标准启动方式，所以sys.argv[0]有4种情况（分别对应上述4种标准启动方式）：

- 空串。
- Python脚本的文件名或路径（取决于操作系统）。
- 实现被执行模块的文件的路径。
- 字符串“-c”。

而从sys.argv[1]开始依次为命令行中给出的参数。请编写argv.py脚本，使其内容为：

```
import sys

print(sys.argv)
```

然后通过如下命令行和语句验证第一种情况：

```
$ python3

>>> import sys
>>> print(sys.argv)
['']
>>>
```

通过如下命令行验证第二种情况：

```
$ python3 argv.py
['argv.py']

$ python3 argv.py a b c
['argv.py', 'a', 'b', 'c']

$ python3 argv.py 0 1
['argv.py', '0', '1']
```

通过如下命令行验证第三种情况：

```
$ python3 -m argv
['/Users/wwy/argv.py']

$ python3 -m argv a b c
['/Users/wwy/argv.py', 'a', 'b', 'c']

$ python3 -m argv 0 1
['/Users/wwy/argv.py', '0', '1']
```

通过如下命令行验证第四种情况：

```
$ python3 -c '''
> import sys
> print(sys.argv)
> '''
['-c']

$ python3 -c '''
import sys
print(sys.argv)
''' a b c
['-c', 'a', 'b', 'c']

$ python3 -c '''
import sys
print(sys.argv)
''' 0 1
['-c', '0', '1']
```

sys.orig_argv用于解析启动Python解释器的整个命令行，其中sys.orig_argv[0]为用于启动Python解释器的可执行文件的路径，后续各元素依次为命令行中选项名、选项参数和命令行参数。请看下面的例子：

```
$ python3 -c '''
import sys
print(sys.orig_argv)
''' 0 1
['/Library/Frameworks/Python.framework/Versions/3.10/Resources/Python.app/
Contents/MacOS/Python', '-c', '\nimport sys\nprint(sys.orig_argv)\n', '0', '1']
```

可以这样认为：sys.org_argv将命令行基于空白符拆分成了一个令牌列表，然后将第一个令牌替换为相应可执行文件的路径，其余令牌则是选项名、选项参数和命令行参数。

sys.flags引用的具名元组反映了Python解释器的当前状态，其内的元素又被称为“状态标志（status flags）”。这些状态标志是通过命令行中的选项来控制的，相当于一些开关，只能引用0（关）或1（开）。表7-7列出了所有状态标志及相关选项（safe_path是Python 3.11引入的）。

表7-7. Python状态标志

标志	引用1的含义	相关选项
debug	开启解释器调试输出。（限专家使用。）	-d
inspect	脚本触发异常时，检查全局变量或堆栈回溯。	-i
interactive	进入交互模式。	-i
isolated	在隔离模式下运行。	-I
optimize	编译时进行优化。	-O、-OO
dont_write_bytecode	不产生.pyc文件。	-B
no_user_site	导入site模块时不进行用户相关站点设置。	-s

标志	引用1的含义	相关选项
no_site	导入site模块时不进行站点设置。	-S
ignore_environment	忽略所有PYTHON*环境变量。	-E
verbose	导入模块和清理模块时输出说明信息。	-v
bytes_warning	比较字符串与二进制序列，以及数值和二进制序列时发出警告。	-b
quiet	以交互模式启动时不显示版本信息。	-q
hash_randomization	启用哈希随机化。	-R
dev_mode	启用Python开发模式。	-X dev
utf8_mode	启用Python的UTF-8模式。	-X utf8
safe_path	启用sys.path保护。	-P

请编写flags.py脚本，使其内容为：

```
import sys

print(sys.flags)
```

然后通过如下命令行验证状态标志：

```
$ python3 flags.py
sys.flags(debug=0, inspect=0, interactive=0, optimize=0,
dont_write_bytecode=0, no_user_site=0, no_site=0, ignore_environment=0,
verbose=0, bytes_warning=0, quiet=0, hash_randomization=1, isolated=0,
dev_mode=False, utf8_mode=0, warn_default_encoding=0, safe_path=False)

$ python3 -i flags.py
sys.flags(debug=0, inspect=1, interactive=1, optimize=0,
dont_write_bytecode=0, no_user_site=0, no_site=0, ignore_environment=0,
verbose=0, bytes_warning=0, quiet=0, hash_randomization=1, isolated=0,
dev_mode=False, utf8_mode=0, warn_default_encoding=0, safe_path=False)

$ python3 -B flags.py
sys.flags(debug=0, inspect=0, interactive=0, optimize=0,
dont_write_bytecode=1, no_user_site=0, no_site=0, ignore_environment=0,
verbose=0, bytes_warning=0, quiet=0, hash_randomization=1, isolated=0,
dev_mode=False, utf8_mode=0, warn_default_encoding=0, safe_path=False)
```

sys.__interactivehook__引用一个可调用对象，当Python解释器以交互模式启动时会在执行完主模块之后自动调用它。如果该sys属性引用None则会跳过这一步。下面的例子给默认行为增加了输出“Interactive Mode!”的步骤：

```
import sys

_old__interactivehook__ = sys.__interactivehook__

def echo():
    print("Interactive Mode!")
    if _old__interactivehook__:
        _old__interactivehook__()

sys.__interactivehook__ = echo

del echo
```

请将其保存为interactivehook.py，然后通过如下命令行验证：

```
$ python3 interactivehook.py
$

$ python3 -i interactivehook.py
Interactive Mode!
>>>
```

sys.ps1和sys.ps2分别储存着Python解释器在交互模式下的主提示符和子提示符，第2章已经说明它们默认分别为“>>>”和“...”。这两个sys属性都是可写的，因此我们可以通过它们来自定义主提示符和子提示符。请通过下面的命令行和语句验证：

```
$ python3

>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps1 = '@ '
@ sys.ps2
'... '
@ sys.ps2 = '? '
@ '''
? yeah
? '''
'\nyeah\n'
@
```

sys.is_finalizing()的功能是返回一个布尔值，以说明当前解释器是否正在关闭。可以在对象的__del__属性中使用它，以使该对象在因解释器关闭而被销毁时做出与因其他原因被销毁时不同的行为。请看下面的例子：

```
import sys

class Prophet:
    def __del__(self):
        #该类的实例因解释器关闭而被销毁。
        if sys.is_finalizing():
            print("I am dying with the world.")
        #该类的实例因其他原因被销毁。
        else:
            print("I am dying but the world goes on.")
```

请将上述代码保存为is_finalizing.py，然后通过如下命令行和语句验证：

```
$ python3 -i is_finalizing.py

>>> p1 = Prophet()
>>> del p1
I am dying but the world goes on.
>>> p2 = Prophet()
>>> ^D
I am dying with the world.
```

最后，在顶层环境中可以通过调用sys.exit()使Python解释器退出，其语法为：

```
sys.exit([arg])
```

事实上，即便在Python脚本中不显式调用sys.exit()，Python解释器在执行完该脚本后也会自动执行语句“sys.exit(None)”，故Python解释器的退出总是通过调用sys.exit()实现的。而sys.exit()的核心功能是抛出SystemExit异常，进而启动解释器的退出流程。

就像其他应用程序一样，Python解释器退出时需要返回一个整数来代表“退出状态（exit status）”。大多数操作系统都要求退出状态取值范围是0～127，其中0代表“正常终止（normal termination）”，非0值代表“异常终止（abnormal termination）”。在调用sys.exit()时，可以通过arg参数指定退出状态，但要注意若传入的不是整数，则按照如下规则处理：

- 传入None等同于传入0。
- 传入其他对象，首先将该对象输出到标准出错，然后以1作为退出状态。

请通过如下命令行和语句验证sys.exit()的用法：

```
$python3 -q
>>> import sys
>>> sys.exit(0)
$

$python3 -q
>>> import sys
>>> sys.exit(127)
$

$python3 -q
>>> import sys
>>> sys.exit('abc')
abc
$

$python3 -q
>>> import sys
>>> sys.exit(0.5)
0.5
$

$python3 -q
>>> import sys
>>> sys.exit(None)
$
```

上述sys属性仅能在顶层环境中使用，然而一个模块既可以被创建为主模块，也可以被其他模块导入。因此Python脚本必须能够判断自身是否在顶层环境中被执行，并将“顶层代码（top-level codes）”与其他代码隔离开。第6章已经说明主模块总是以“__main__”为模块名，Python脚本可以利用这点判断它是否被作为主模块运行。但第6章给出的例子非常简单，if语句中的每个分支都只包含了一次对print()的调用。这并不规范，存在安全隐患。下面给出Python脚本最规范的格式：

```
#Shebang goes here.
#Module-level comments go here.

"""Module-level docstring goes here."""

import sys

#Definition of __all__ goes here.

#... other codes go here ...


def main():
    #... top-level codes go here...
    return 0


#... other codes go here ...


if __name__ == '__main__':
    sys.exit(main())
```

也就是说定义一个函数main()来容纳所有顶层代码，仅当模块名为“__main__”时才通过sys.exit()调用main()，以main()的返回值作为退出状态。显然，main()可以具有多种返回值，但至少要在某种情况下返回0（或None）以表示正常终止。

按照上述规范来编写Python脚本的目的有两个：

➤ 在main()中定义的标识符是本地变量而非全局变量，因此不会被脚本的其他部分访问，更加安全。
➤ 当通过给Python脚本添加Shebang使其变成shell脚本（可直接执行）时，能保证退出状态符合shell脚本的规范。

事实上，main()将成为整个Python程序的“入口点（entry point）”，就像C中的main()函数一样。

下面给出一个符合上述规范的Python脚本的例子demo.py：

```
#!/usr/bin/env python3
#
#这是一个用于展示规范格式的Python脚本。

"""This module defines a factorial sequence generator.

    The function factorial_sequence() generate the sequence and return it
as a list.

    When runs as the main module, it asks the user to enter an integer,
then print the according factorial sequence.
"""

import sys
import re

__all__ = ['factorial_sequence']


#该函数根据参数n生成序列[1!, 2!, ..., n!]。
def factorial_sequence(n):
    """This function is the factorial sequence generator."""

    l = []
    i = 1
    factorial = 1

    while i <= n:
        factorial = factorial * i
        i = i + 1
        l.append(factorial)

    return l


#入口点函数。
def main():
```

```
        """This function is called when this script runs in the top-level
environment."""

        print("这是一个生成阶乘序列的工具。输入“quit”退出。\n")

        while True:
           s = input("请输入一个正整数: ")
           if s == "quit":
               break
           if re.match(r"^[1-9][0-9]*$", s):
               n = int(s)
               print(factorial_sequence(n))
           else:
               print("您输入的并非正整数。")

        return 0


    #判断脚本是否在顶层环境中被执行。
    if __name__ == '__main__':
        sys.exit(main())
```

首先通过如下命令行验证demo.py在顶层环境下的执行效果：

```
    $ python3 demo.py
    这是一个生成阶乘序列的工具。输入“quit”退出。

    请输入一个正整数: 3
    [1, 2, 6]
    请输入一个正整数: 4
    [1, 2, 6, 24]
    请输入一个正整数: -1
    您输入的并非正整数。
    请输入一个正整数: quit
```

如果你在使用Unix或类Unix操作系统，那么请通过如下命令行将demo.py改造成shell脚本：

```
    $ chmod 755 demo.py
```

然后就可以通过如下命令行直接执行demo.py，效果与通过python3命令执行它相同：

```
    $ ./demo.py
```

最后，我们已经知道导入一个常规包等价于执行其内的__init__.py。此外，通过-m选项执行一个常规包时，会在执行完__init__.py之后自动执行其内的__main__.py，这使得常规包也可以包含顶层代码。但需要强调，每个文件通过import语句导入的标识符的作用域都不可能超出该文件，因此一个标识符被__init__.py导入后并不能直接在__main__.py中使用，而需要被__main__.py重新导入。

下面将demo.py改造成demo包。请在工作目录下创建子目录demo，然后在子目录内创建四个Python脚本：factorial_sequence.py、main.py、__init__.py和__main__.py。

factorial_sequence.py的内容为：

```
    #该函数根据参数n生成序列[1!, 2!, ..., n!]。
    def factorial_sequence(n):
        """This function is the factorial sequence generator."""

        l = []
        i = 1
        factorial = 1

        while i <= n:
            factorial = factorial * i
            i = i + 1
            l.append(factorial)

        return l
```

main.py的内容为（注意由于在第二条import语句中使用了相对模块名，main.py不再能够被直接执行，而只能通过包——也就是__main__.py——来执行）：

```
    import re
    from .factorial_sequence import *

    #入口点函数。
    def main():
        """This function is called when this script runs in the top-level
environment."""

        print("这是一个生成阶乘序列的工具。输入“quit”退出。\n")

        while True:
           s = input("请输入一个正整数: ")
           if s == "quit":
               break
           if re.match(r"^[1-9][0-9]*$", s):
               n = int(s)
               print(factorial_sequence(n))
           else:
               print("您输入的并非正整数。")

        return 0
```

__init__.py的内容为：

```
    #这是一个用于展示规范格式的包。

    """This packet contains a factorial sequence generator.

        The function factorial_sequence() generate the sequence and return it
as a list.
```

```
        When runs as the main module, it asks the user to enter an integer,
then print the according factorial sequence.
    """

    __all__ = ['factorial_sequence']

    from .factorial_sequence import *
```

而__main__.py的内容为：

```
    import sys
    from .main import main

    #判断包是否在顶层环境中运行。
    if __name__ == '__main__':
        sys.exit(main())
```

请通过如下命令行验证demo包在顶层环境下执行时效果与demo.py相同：

```
    $ python3 -m demo
```

7-3. Python的I/O模型

（语言参考手册：3.2）
（标准库：io）

一个程序通过系统调用访问外围设备以获取数据或给出计算结果的过程被称为“输入/输出”，简记为“I/O”。用户必须通过I/O与程序交互，脱离了I/O的程序依然可以运行但毫无用处，就好像一个人失去了感觉器官和四肢后依然可以思考，但这些思考既与外界无关也无法影响外界。本书前面的例子中已经多次用到了内置函数input()和print()，它们其实都属于I/O操作。此外，在Python解释器的交互模式下对一个表达式求值，返回的对象会被显示，这也是一种I/O操作。然而这些都只是最简单的I/O操作，仅涉及标准I/O（将在本章后面详细讨论）。在这一节，我们将系统性地讨论Python的I/O模型。

关于I/O，我们必须知道这样一些常识：

➤ 与CPU中的运算以及内存访问比较，I/O操作的速度要慢几个数量级，通常是限制程序运行速度的瓶颈，因此编写程序时应尽可能减少I/O操作的次数。
➤ 外围设备多种多样，不同外围设备的信息处理方式迥异。好在计算机系统为每个外围设备安装了其独有的驱动程序，而操作系统包装了这些驱动程序，提供了相对统一的接口，即I/O系统调用。
➤ 不同操作系统提供的I/O系统调用是存在差异的。Unix和类Unix操作系统将一切外围设备都抽象为文件，而打开文件将得到“文件描述符（file descriptors）”。Windows则依然区分文件和其他外围设备，且通过“句柄（handles）”来指向它们。

综上所述，尽管操作系统已经屏蔽了外围设备的绝大部分差异性，但提供的I/O系统调用依然不是完全统一的。Python解释器为了做到跨平台，通过标准库中的os模块包装了所有操作系统通用的操作（同一操作在不同操作系统上需通过不同系统调用实现），而将某些操作系统独有的操作放在其他标准库模块中（例如posix和winreg）。易知，I/O系统调用也是通过这些模块中的属性包装的，然而本书不详细讨论这些模块。标准库中的io模块则建立在上述模块之上，通过其属性提供了更统一的I/O操作，实现了Python的I/O模型。下面将详细讨论io模块中定义的类和函数。

Python将所有外围设备都抽象为“流（streams）”，并通过所谓的“文件对象（file objects）”来表示。（也有些人将此类对象称为“类文件对象（file-like objects）”。）在抽象程度最低的视角，流被视为简单的字节序列，称为“原始二进制流（raw binary streams）”。抽象程度再高一点的视角会引入对缓冲区的支持，相应的流被称为“缓冲二进制流（buffered binary streams）”。而“文本流（text streams）”属于最高抽象程度的视角，不仅支持缓冲区，还需要将字节解读为字符的编码，进而将字节序列转化为字符序列。

需要强调的是，一个外围设备并非产生上述三种流中的某一种，而是产生的流从不同视角被视为不同的流。举例来说，我们可以用上述三种流中的任意一种来访问一个文本文件。不论以哪种视角，一个流都以“文件结束符（EOF）”作为结束标记，它是一个特殊的字节，但在不同操作系统中具体编码不同。

根据可对一个流执行的操作，我们又把流分为：“只读”、“只写”和“读写”三类。根据是否支持设置游标，流又被分为：“随机访问”和“顺序访问”两类。这两种分类方法与上面介绍的分类方法分别属于三个维度，可以自由组合，例如“顺序只读文本流”和“随机读写缓冲二进制流”等等。

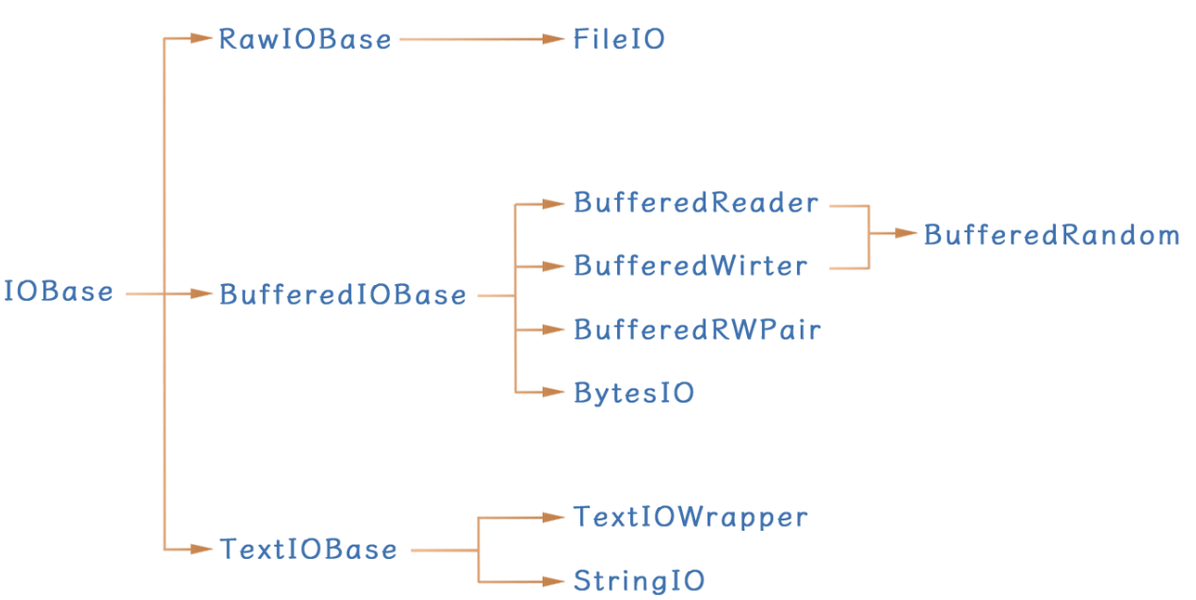

图7-1. 文件对象的类型

由于流存在这么多种类型，所以文件对象并非指某个特定类的实例，而可以属于从抽象基类IOBase派生出的任何一个类。（“抽象基类”指的是自身不能被实例化，只能被其他类继

承的类，将在第15章讨论。）图7-1显示了io模块定义的所有能实例化出文件对象的类以及它们的基类，而我们还能以它们为基类定义自己的I/O类型。下面将分别讨论这些类。

IOBase提供对（几乎）所有类型的流都适用的通用操作。IOBase自身实现的属性被总结在表7-8中，任何文件对象都具有它们。表7-9则列出了IOBase的“抽象属性（abstract attributes）”（即自己没有实现，但它的子类需要根据情况实现的属性）。IOBase同时是一个上下文管理器，即实现了__enter__和__exit__魔术属性（会在第8章详细讨论）。IOBase还支持迭代器协议，即实现了__iter__和__next__魔术属性（会在第12章详细讨论）。除此之外，IOBase还实现了__del__特殊属性。

表7-8. IOBase实现的属性

属性	说明
`isatty()`	返回一个布尔值，表明该流是否被关联到一个终端。
`seekable()`	返回一个布尔值，表明该流是否支持随机访问。
`readable()`	返回一个布尔值，表明该流是否可读。
`writable()`	返回一个布尔值，表明该流是否可写。
`closed`	一个布尔值，表明该流是否已关闭。
`tell()`	返回一个非负整数，代表该流的游标的当前位置。
`readlines()`	从流中读取并返回多行。可限制读取的行数。
`readline()`	从流中读取一行，并返回其中指定个字节。
`writelines()`	向流中写入多行。
`flush()`	刷新流的写入缓冲区。
`close()`	刷新流的写入缓冲区，然后关闭流。

表7-9. IOBase的抽象属性

属性	说明
`seek()`	设置流的游标的位置。
`truncate()`	将流调整到指定大小。
`fileno()`	返回一个整数，为该流的文件描述符。仅适用于Unix和类Unix操作系统。

下面详细讨论IOBase所提供的的I/O操作。首先，isatty()、seekable()、readable()、writable()和closed提供了流的一些类型和状态信息，但类型信息并不完整，例如我们无法基于这些属性区分原始二进制流、缓冲二进制流和文本流。

isatty()判断该流是否关联到一个终端，如果是则说明该流被用于和用户交互。在20世纪60年代和70年代，终端是一种独立的外围设备，专门用于和用户交互，同时具备输入功能和输出功能。目前终端已经成为了一个抽象概念，代表与用户直接交互的外围设备组合（例如显示器、键盘和鼠标）。

seekable()判断该流是否支持随机访问。目前大部分外围设备都支持随机访问，只有一些老式外围设备（例如磁带机）仅支持顺序访问。如果seekable()返回True，则相应文件对象的tell()和seek()可用；否则，调用tell()或seek()会抛出UnsupportedOperation异常。

tell()返回一个非负整数，解读为相对于流的开头的偏移量，亦即游标的当前值。

seek()的语法为：

```
io.IOBase.seek(offset, whence=SEEK_SET)
```

它通过如下参数修改游标的值：
offset：相对于whence指定位置的偏移量，取值范围根据whence被传入的对象确定，单位是字节。
whence：可以被传入如下对象（注意SEEK_*是os模块定义的常量）：

- SEEK_SET：等价于0，代表流的开头。此时offset必须为非负整数。
- SEEK_CUR：等价于1，代表游标的当前位置。此时offset可以是任意整数。
- SEEK_END：等价于2，代表流的末尾。此时offset必须为非正整数。

不论给定什么参数组合，seek()都会返回一个非负整数，代表相对于流的开头的偏移量，以给出游标被设置后的值。

readable()判断该流是否支持读操作，如果返回True，则相应文件对象的readlines()、readline()和read()可用，否则调用它们会抛出UnsupportedOperation异常。writable()判断该流是否支持写操作，如果返回True，则该文件对象的writelines()和write()可用，否则调用它们会抛出UnsupportedOperation异常。需要注意，read()和write()并不是BaseIO提供的属性，而是由其直接子类（RawIOBase、BufferedIOBase和TextIOBase）提供的属性。

readlines()、readline()和writelines()提供了所有流都支持的读写方式，即按行读写，其语法为：

```
io.IOBase.readlines(hint=-1)
io.IOBase.readline(size=-1)
io.IOBase.writelines(lines)
```

为了把数据分行，必须定义“行结束符（EOL）”。对于二进制流（包括原始二进制流和缓冲二进制流，下同）来说，EOL总是取值00001010的字节（这也是ASCII码中\n的编码）；而对于文本流来说，EOL可以是\n、\r和\r\n之一，默认情况下为\n，并将\r和\r\n自动替换为\n，但能够在创建流时指定。此外，EOF总被视为单独一行。

readlines()的行为是这样的：假设从游标指向的字节所在行开始到EOF为止（不包括EOF）的行数为m，hint参数被传入n（n≤0或为None时当成+∞处理），则从游标指向的字节开始读取min{m, n}行数据，整理成一个字节串形成的列表返回，同时将游标移动到min{m, n}+1行的行首。如果游标初始位置指向某行的第一个字节，那么readlines()返回的列表中的第一行是完整的；否则，返回列表中的第一行是不完整的（缺少前面若干字节）。此外，readlines()有个怪癖：处理连续的EOL时每两个EOL视为一个空行。

readline()的行为是这样的：假设从游标指向的字节开始到下一个EOL为止（包括EOL）的字节数为m，size参数被传入n（n=-1时当成+∞处理），则从游标指向的字节开始读取min{m, n}个字节，整理成一个字节串返回，同时将游标向前移动min{m, n}个字节。与readlines()不同，处理连续的EOL时realine()将每个EOL视为一个空行。

writelines()的行为是这样的：lines参数必须被传入一个“类字节对象（bytes-like objects）”（典型例子是二进制序列，会在第10章讨论）形成的列表（或其他可迭代对象）。列表中的类字节对象会被依次拼接起来，然后从游标指向的字节开始写入，并覆盖原有的字节。如果覆盖到了EOF，则会增大流的长度以容纳后续字节，并在完成写入后追加EOF。不论是哪种情况，游标都会被移动到指向最后一个被写入的字节（不包括追加的EOF）的后面。writelines()的返回值是None。

当seekable()和writable()同时返回True时，相应文件对象必须实现truncate()属性，其语法为：

```
io.IOBase.truncate(size=None)
```

它的行为是这样的：假设size参数被传入了非负整数n，则将流的大小调整为n个字节（需要计入EOL但不计入EOF）。当n小于流的当前大小时，保留前n个字节；当n大于流的当前大小时，在末尾用所有位都为0的字节（这也是ASCII码中\0的编码）填充；当n等于流的当前大小时，不改变流的大小。特别的，如果size参数被传入None则截断该流，即保留从流的开头开始直到游标指向字节为止的部分。不论是哪种情况，游标都会被设置为指向EOF。truncate()的返回值与tell()的返回值含义相同，而由于游标总是指向EOF，因此该返回值总是等于流最后的大小，亦即size参数被传入的整数。

不论一个流是否使用了缓冲区，flush()总是可用的。如果存在缓冲区，则调用flush()会将写入缓冲区里的数据全部同步到外围设备，然后清空缓冲区；否则，flush()什么也不做。

close()的功能是关闭一个流，使其不再能被执行读写操作。close()会在内部调用flush()，以确保使用了缓冲区的流不会丢失缓冲区中的数据。调用一个文件对象的close()属性后，它的closed属性就将引用True。处于这种状态的文件对象已经没有什么作用了，所以通常接下来会让其引用数变为0，以被垃圾回收机制销毁。IOBase实现的__del__等价于如下代码：

```
    def __del__(self):
        if not self.closed:
            self.close()
```

这是为了最大程度保证在文件对象被销毁前已经清空了缓冲区，但由于__del__是不可靠的，所以强烈建议在不再需要一个文件对象时主动调用它的close()属性。此外，根据第5章对__del__的讨论，由于IOBase实现了__del__，所以它的一切子类都必须以如下方式实现__del__：

```
    def __del__(self):
        # ... extra codes go here ...
        super().__del__()
```

最后，仅当Python解释器运行在Unix或类Unix操作系统之上时，文件对象才会实现fileno()属性，可通过它取得相应的文件描述符。该信息仅当需要使用包装了系统调用的模块（例如os）时才有用。

7-4. 原始二进制流

（标准库：io）

接下来讨论原始二进制流。RawIOBase是IOBase的直接子类，但仍然是抽象基类，提供所有原始二进制流通用的操作，这些操作本质上是对系统调用的包装。RawIOBase实现的属性被总结在表7-10中，而它的抽象属性被总结在表7-11中。可以通过内置函数isinstance()判断一个文件对象是否是RawIOBase的实例，进而确定该文件对象是否代表一个原始二进制流。

表7-10. RawIOBase实现的属性

属性	说明
read()	读取并返回至多指定个字节。支持阻塞式I/O。
readall()	一次性读取所有字节。支持阻塞式I/O。

表7-11. RawIOBase的抽象属性

属性	说明
write()	写入指定类字节对象，并返回成功写入的字节数。支持阻塞式I/O。
readinto()	将读取的字节写入指定类字节对象，并返回成功读取的字节数。支持阻塞式I/O。

RawIOBase提供的核心属性是read()和write()，与readlines()和writelines()相比，它们不是按行读写的，且支持阻塞式I/O。所谓的“阻塞式I/O”指的是当一个读操作读取不到任何数据，或者一个写操作无法写入任何数据时，并不会立刻返回失败结果，而是让执行该I/O操

作的线程进入睡眠，直到被访问的外围设备状态发生变化，使该I/O操作成功完成。（关于线程和阻塞式I/O的知识可在讨论操作系统的书籍中找到。）

我们已经知道，仅当readable()返回True时才能使用read()，其语法为：

```
io.RawIOBase.read(size=-1)
```

size参数指定read()读取多少字节，但这只是期望值，真正读取到的字节数可能小于该值（例如直到遇到EOF都未能读取到足够的字节）。若size参数被传入-1，则从游标指向的字节读取到EOF之前的字节。read()读取了n字节，就会将游标向前移动n字节，以避免读取重复的数据。read()的返回值总是一个字节串，如果该字节串是空的，则说明游标已经指向了EOF。

另外，虽然read()支持阻塞式I/O，但如果该流对应的外围设备本身并不支持阻塞式I/O，那么read()依然只能以非阻塞式I/O的方式被执行，有可能读取失败，这种情况下返回None。不论哪种情况，read()都只会执行一次I/O系统调用。

同样，仅当writable()返回True时才能使用write()，其语法为：

```
io.RawIOBase.write(b)
```

调用write()时需给b参数传入一个类字节对象，其内容会从游标指向的字节开始写入原始二进制流。对于支持阻塞式I/O的外围设备来说，write()能够保证将所有数据都写入后才返回；而对于不支持阻塞式I/O的外围设备来说，write()有可能只写入数据的前若干字节。write()会返回成功写入的字节数，可据此判断类字节对象中的哪些数据被成功写入。与read()一样，write()只会执行一次I/O系统调用。

readall()会读取从游标指向直接开始到EOF之前为止的所有数据，并使游标指向EOF标志。readall()与read(-1)的区别在于前者允许执行多次I/O系统调用，以确保读取到EOF之前的所有字节。readall()总是可用的，但当readable()返回False时调用它会抛出OSError异常。

对大多数Python解释器来说，readinto()其实是read()的内部实现，即read()在内部调用readinto()，其语法为：

```
io.RawIOBase.readinto(b)
```

直接调用readinto()时必须给b参数传入一个可写的类字节对象（例如字节数组），读取到的数据会被写入该对象直到超出该对象的容量，而返回值则是成功写入的字节数。换句话说，readinto()通过类字节对象的容量来控制一次读取多少数据的，不需要size参数。当read()不可用时，readinto()也不可用。

FileIO是RawIOBase的直接子类，且不再是抽象基类，可以被实例化，相应语法为：

```
class io.FileIO(name, mode='r', closefd=True, opener=None)
```

其name参数可以是如下两种对象：

- 一个代表文件路径的字符串或二进制序列，指向一个即将被打开的文件或外围设备。
- 一个文件描述符，对应一个已经被打开的文件。（仅适用于Unix和类Unix操作系统。）

不论是哪种情况，新创建的原始二进制流都是访问相应外围设备的一个接口，而mode参数则用于指定该接口允许什么形式的访问，可以是由表7-11列出的字符之一，或者这四个字符之一结合“+”（表示同时允许读取和写入）构成的字符串。

表7-11. FileIO的模式

模式	说明
r	只允许读取，如果文件不存在则抛出FileNotFoundError异常。
w	只允许写入，并清空文件，如果文件不存在则创建它。
a	只允许写入，且强制将数据追加到该文件已有内容之后，如果文件不存在则创建它。
x	只允许写入，且强制新建该文件，如果该文件已经存在则抛出FileExistsError异常。

FileIO的closedfd参数用于控制当该FileIO对象被销毁时是否同时关闭文件（会导致相应文件描述符消失）。当name参数是路径时，closedfd参数必须是True，以确保关闭该文件；否则，closedfd参数可以是False。

FileIO的opener参数用于支持自定义文件打开器，即一个“opener(name, flags)”形式的可调用对象，调用它最后会得到一个文件描述符。编写文件打开器需要用到系统调用，且必须考虑操作系统的差异性，因此非常复杂，超出了本书的范围。在绝大部分情况下，使用默认的文件打开器即可。

FileIO对象具有表7-12列出的实例属性。但要注意，mode属性引用的字符串与FileIO的mode参数有细微的区别，而是符合内置函数open()的mode参数的格式（例如若FileIO的mode参数是“r”，则其实例的mode属性引用“rb”），这会在后面讨论。

表7-12. FileIO对象的实例属性

属性	说明
name	创建该FileIO对象时指定的路径或文件描述符。
mode	创建该FileIO对象时指定的模式。

至此，可以通过一些例子来验证上面介绍的类和属性了。请创建test1.txt（注意中间有两个空行）：

```
Python is fun!


Life is short, you need Python.
```

test2.txt：

```
你好
こんにちは
```

和test3.txt（“1111”后面有个空格，末尾不要换行）：

```
0000 0001 0010 0011 1100 1101 1110 1111
```

请将它们都保存为采用UTF-8编码的文本文件。

下面的例子以“r”模式创建一个FileIO对象，然后通过readlines()和readline()读取test1.txt中的内容：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test1.txt')
>>> raw_stream.isatty()
False
>>> raw_stream.seekable()
True
>>> raw_stream.readable()
True
>>> raw_stream.writable()
False
>>> raw_stream.closed
False
>>> raw_stream.tell()
0
>>> raw_stream.readlines()
[b'Python is fun!\n', b'\n', b'\n', b'Life is short, you need Python.\n']
>>> raw_stream.tell()
49
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines(1)
[b'Python is fun!\n']
>>> raw_stream.tell()
```

```
15
>>> raw_stream.readlines(2)
[b'\n', b'\n', b'Life is short, you need Python.\n']
>>> raw_stream.tell()
49
>>> raw_stream.seek(4)
4
>>> raw_stream.readlines()
[b'on is fun!\n', b'\n', b'\n', b'Life is short, you need Python.\n']
>>> raw_stream.seek(0)
0
>>> raw_stream.readline()
b'Python is fun!\n'
>>> raw_stream.readline()
b'\n'
>>> raw_stream.readline()
b'\n'
>>> raw_stream.readline(3)
b'Lif'
>>> raw_stream.readline(4)
b'e is'
>>> raw_stream.readline(5)
b' shor'
>>> raw_stream.readline()
b't, you need Python.\n'
>>> raw_stream.close()
>>> raw_stream.closed
True
>>> raw_stream.readlines()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>> del raw_stream
>>>
```

注意字节串被显示为添加了前缀“b”的字符串（这会在第10章解释）。这个例子说明了如何查看流的类型和状态，以及按行读取流中的数据。

test1.txt中仅包含ASCII字符，而test2.txt则包含采用多字节编码的非ASCII字符。下面的例子以“r”模式创建一个FileIO对象，然后通过readlines()读取test2.txt的内容：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test2.txt')
>>> raw_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n']
>>> raw_stream.close()
>>>
```

注意此时字节串同样被显示为添加了前缀“b”的字符串，但字符串中包含的是转义序列，每个转义序列对应一个字节的编码。这更能反映出用二进制流进行I/O操作时是无法识别字符的（前面例子中仅包含ASCII字符的字符串属于特殊情况）。

下面的例子以“w+”模式创建一个FileIO对象，然后通过writelines()修改test1.txt的内容：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test1.txt', 'w+')
>>> raw_stream.readable()
True
>>> raw_stream.writable()
True
>>> raw_stream.tell()
0
>>> raw_stream.readlines()
[]
>>> raw_stream.writelines([b'Python is fun!\n', b'\n'])
>>> raw_stream.tell()
16
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'Python is fun!\n', b'\n']
>>> raw_stream.writelines([b'\n', b'Life is short,\n'])
>>> raw_stream.tell()
32
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'Python is fun!\n', b'\n', b'\n', b'Life is short,\n']
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines(1)
[b'Python is fun!\n']
>>> raw_stream.readlines(1)
[b'\n', b'\n']
>>> raw_stream.writelines([b'Life is short, you need Python.\n'])
>>> raw_stream.tell()
49
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'Python is fun!\n', b'\n', b'\n', b'Life is short, you need Python.\n']
>>> raw_stream.close()
>>>
```

从上面的例子可以看出，当以“w”或“w+”模式创建FileIO对象时，会自动将已有文件的内容清空，这在有些情况下是危险的。“x”或“x+”模式要安全一些，因为保证不会清空已经存在的文件。这可以通过下面的例子验证：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test1.txt', 'x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'test1.txt'
>>>
```

如果想写入一个文件但同时保留它的已有内容，则应以“a”或“a+”模式创建FileIO对象。但需要强调的是，这种模式只允许追加已有内容，不允许覆盖已有内容，即便改变了游标的位置也没有用。下面的例子说明了这一技巧：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test2.txt', 'a+')
>>> raw_stream.tell()
24
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n']
>>> raw_stream.writelines([b'\n', b'Bonjour\n'])
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n', b'\n',
b'Bonjour\n']
>>> raw_stream.seek(-9, io.SEEK_END)
24
>>> raw_stream.writelines([b'\n', b'GutenTag\n'])
>>> raw_stream.seek(0)
0
>>> raw_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n', b'\n',
b'Bonjour\n', b'\n', b'GutenTag\n']
>>> raw_stream.close()
>>>
```

现在让我们思考这样一个问题：想要保持已有文件的内容不变，同时向其任意位置插入新内容，该怎么实现呢？“w”和“w+”模式会清空文件，“x”和“x+”模式在文件已经存在的情况下不可用，“a”和“a+”模式只能追加内容，“r”模式不允许写入，所以只能使用“r+”模式。下面的例子向test1.txt已有的两行中插入了一行：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test1.txt', 'r+')
>>> raw_stream.readable()
True
>>> raw_stream.writable()
True
>>> raw_stream.tell()
0
>>> content = raw_stream.readlines()
>>> content
[b'Python is fun!\n', b'\n', b'\n', b'Life is short, you need Python.\n']
>>> raw_stream.seek(0)
0
```

```
      >>> raw_stream.writelines([content[0], content[1], b'Insert a line is not
easy...\n', content[2], content[3]])
      >>> raw_stream.seek(0)
      0
      >>> raw_stream.readlines()
      [b'Python is fun!\n', b'\n', b'Insert a line is not easy...\n', b'\n',
b'Life is short, you need Python.\n']
      >>> raw_stream.close()
      >>>
```

这个例子给出的方法仅能用于处理较小的文件，以保证其内容可以完全存放在内存空间中。对于大文件，只能用一个“r”模式的流读取它的内容，同时用另一个“w”模式的流将这些内容连同插入的新内容一起写入另一个文件，再将新文件的内容写回旧文件，或通过os模块提供的文件系统操作属性删除原文件，再将新文件改名。而使用缓冲二进制流中的BufferedRWPair流完成上述操作更容易。

上面例子中的所有读写操作都是通过readlines()、readline()和writelines()实现的。然而按行读写依赖于EOL，而有些文件格式不存在EOL。注意test3.txt中储存的是用空格分隔的4位0-1序列，直到末尾都没有换行。读写这类文件用read()和write()更合适。下面的例子说明了如何用read()读取test3.txt中的0-1序列，以及用write()追加新的0-1序列：

```
      $ python3

      >>> import io
      >>> raw_stream = io.FileIO('test3.txt', 'a+')
      >>> raw_stream.seek(0)
      0
      >>> raw_stream.read(5)
      b'0000 '
      >>> raw_stream.read(5)
      b'0001 '
      >>> raw_stream.read(10)
      b'0010 0011 '
      >>> raw_stream.read()
      b'1100 1101 1110 1111 '
      >>> raw_stream.write(b'1011 1010 ')
      10
      >>> raw_stream.seek(0)
      0
      >>> raw_stream.readall()
      b'0000 0001 0010 0011 1100 1101 1110 1111 1011 1010 '
      >>> raw_stream.close()
      >>>
```

7-5. 缓冲二进制流

（标准库：io）

缓冲二进制流是对原始二进制流的包装，核心区别在于引入缓冲区，此外还引入了一些更便于使用的属性。这里需要强调，此处的缓冲区是用户级的，在Python解释器的内存空间中实现。事实上I/O系统调用也会使用缓冲区，但这些缓冲区是在操作系统内存空间中实现的，对用户不可见。用户级缓冲区又分为两部分，针对读操作的部分被称为“读取缓冲区”，针

对写操作的部分被称为“写入缓冲区”。引入用户级缓冲区的好处是可以进一步减少读写外围设备的次数，以提高程序的执行速度，但代价是因故障导致数据丢失的风险变高。

BufferedIOBase也是IOBase的直接子类，且同样是抽象基类。表7-13和表7-14分别列出了BufferedIOBase实现的属性和抽象属性。可以通过内置函数isinstance()判断一个文件对象是否是BufferedIOBase的实例，进而确定该文件对象是否代表一个缓冲二进制流。

表7-13. BufferedIOBase实现的属性

属性	说明
`readinto()`	将读取的字节写入指定类字节对象，并返回成功读取的字节数。
`readinto1()`	与readinto()的区别是仅调用一次原始二进制流的readinto()。

表7-14. BufferedIOBase的抽象属性

属性	说明
`read()`	读取并返回至多指定个字节。
`read1()`	与read()的区别是仅调用一次原始二进制流的read()。
`write()`	写入指定类字节对象，并返回成功写入的字节数。
`detach()`	将缓冲区与原始二进制流分离，并返回原始二进制流。

BufferedIOBase的read()、readinto()和write()分别是通过在内部调用其包装的原始二进制流的read()、readinto()和write()实现的，语法为：

```
io.BufferedIOBase.read(size=-1)
io.BufferedIOBase.readinto(b)
io.BufferedIOBase.write(b)
```

其区别有两点：

➤ 为了尽可能读写指定的字节数，可能会多次调用原始二进制流的同名属性。
➤ 当原始二进制流对应的外围设备不支持阻塞式I/O，而读写操作又失败时，不会正常返回，而是抛出BlockingIOError异常。

而read1()和readinto1()则分别是read()和readinto()的变体，区别仅在于只调用原始二进制流的read()和readinto()一次，其语法为：

```
io.BufferedIOBase.read1(size=-1)
io.BufferedIOBase.readinto1(b)
```

在创建缓冲二进制流时，需要给出已经存在的原始二进制流，并将其与指定的用户级缓冲区绑定。调用detach()则会解除这一绑定，返回原始二进制流，而相关缓冲区则会被销毁。换言之，detach()的作用是将一个缓冲二进制流降级成一个原始二进制流。但并非所有缓冲二进制流都实现了detach()，后面遇到这种情况时会特别说明。此外，调用缓冲二进制流的close()会导致其包装的原始二进制流也自动调用close()。

BufferedReader和BufferedWriter都是BufferedIOBase的直接子类，且可以被实例化，语法分别为：

```
class io.BufferedReader(raw, buffer_size=DEFAULT_BUFFER_SIZE)
class io.BufferedWriter(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```

其中raw参数是一个已经存在的原始二进制流，而buffer_size参数用于指定缓冲区的大小。

BufferedReader用于包装“r”模式的原始二进制流，额外提供了peek()属性（见表7-15），其语法为：

```
io.BufferedReader.peek([size])
```

该属性类似于read1()，但会保持游标不变。

表7-15. BufferedReader实现的属性

属性	说明
peek()	读取并返回至多指定个字节，且保持游标位置不变。仅执行一次原始二进制流的read()。

BufferedWriter用于包装“w”、“x”和“a”模式的原始二进制流，重写了flush()以使它在非阻塞式I/O失败时会抛出BlockingIOError异常。

BufferedRandom同时继承了BufferedReader和BufferedWriter，用于包装“r+”、“w+”、“x+”和“a+”模式的原始二进制流，其实例化语法为：

```
class io.BufferedRandom(raw, buffer_size=DEFAULT_BUFFER_SIZE)
```

此外，BufferedRandom确保缓冲二进制流支持随机访问，即便原始二进制流本身不支持随机访问。（这是利用缓冲区实现的。）

BufferedRWPair也是BufferedIOBase的直接子类，用于将一个缓冲区同时与两个原始二进制流绑定，第一个必须可读，第二个必须可写。该流的读操作会作用到第一个原始二进制流上，而该流的写操作会作用到第二个原始二进制流上。它的实例化语法为：

```
    class io.BufferedRWPair(reader, writer, buffer_size=
DEFAULT_BUFFER_SIZE)
```

其中reader参数是第一个原始二进制流，writer参数是第二个原始二进制流。易知，使用BufferedRWPair给文件在任意位置插入数据更加方便。由于同时涉及两个原始二进制流，所以BufferedRWPair没有实现detach()。此外，BufferedRWPair不支持随机访问。

对于上述四种缓冲二进制流来说，每次读操作都会填满读取缓冲区，不论期望的字节数是多少（基于局部性原理这样做能减少I/O操作的次数）；而写入缓冲区中的内容仅在如下情况下被同步到外围设备：

- 写入缓冲区剩余空间已经容纳不下新的数据。
- flush()被调用。
- BufferedRandom对象调用了seek()。

最后，BytesIO同样是BufferedIOBase的直接子类，用于创建一种特殊的缓冲二进制流——它没有包装已有的原始二进制流，而是代表缓冲区本身，其实例化语法为：

```
    class io.BytesIO([initial_bytes])
```

其中initial_bytes参数是一个用于初始化缓冲区的类字节对象，如果省略则初始化一个空的缓冲区。BytesIO的一切操作都是针对缓冲区进行的，不涉及I/O操作，调用close()后缓冲区中的数据会丢失，所以它本质上是一种可任意扩展且不限制内容格式的特殊二进制类型。BytesIO重写了read1()和readinto1()，使它们分别与read()和readinto()完全相同。表7-16列出了BytesIO实现的其他属性。显然，BytesIO不支持detach()。

7-16. BytesIO实现的属性

属性	说明
`getbuffer()`	返回绑定到该缓冲区的可读可写内存视图。
`getvalue()`	以字节串的形式返回缓冲区的内容。

下面通过一些例子来验证缓冲二进制流的使用。首先通过一个BufferedRandom对象向test2.txt的中间位置插入行：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test2.txt', 'r+')
>>> buf_stream = io.BufferedRandom(raw_stream)
>>> buf_stream.tell()
0
>>> buf_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n', b'\n',
b'Bonjour\n', b'\n', b'GutenTag\n']
>>> buf_stream.seek(-18, io.SEEK_END)
25
>>> tail = buf_stream.peek()
>>> tail
b'Bonjour\n\nGutenTag\n'
>>> buf_stream.tell()
25
>>> buf_stream.write(b'Hello\n\n' + tail)
25
>>> buf_stream.seek(0)
0
>>> buf_stream.readlines()
[b'\xe4\xbd\xa0\xe5\xa5\xbd\n', b'\n',
b'\xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf\n', b'\n',
b'Hello\n', b'\n', b'Bonjour\n', b'\n', b'GutenTag\n']
>>> det_stream = buf_stream.detach()
>>> det_stream == raw_stream
True
>>> buf_stream.closed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: raw stream has been detached
>>> raw_stream.closed
False
>>> raw_stream.close()
>>>
```

该例子还说明了peek()和detach()的作用。当缓冲二进制流调用了detach()后自身就不再是一个流，因此不需要调用close()。

接下来通过BufferedRWPair将test3.py的内容拷贝到test4.py，并将“1000 1001 ”插入到“1010 ”之前：

```
$ python3

>>> import io
>>> raw_reader = io.FileIO('test3.txt')
>>> raw_writer = io.FileIO('test4.txt', 'w')
>>> buf_stream = io.BufferedRWPair(raw_reader, raw_writer)
>>> buf_stream.seekable()
False
>>> head = buf_stream.read(40)
>>> head
```

```
b'0000 0001 0010 0011 1100 1101 1110 1111 '
>>> buf_stream.write(head)
40
>>> buf_stream.write(b'1000 1001 ')
10
>>> tail = buf_stream.read()
>>> tail
b'1010 1011 '
>>> buf_stream.write(tail)
10
>>> buf_stream.close()
>>> raw_reader.closed
True
>>> raw_writer.closed
True
>>>
```

注意这会导致test4.txt被创建，其内容为：

```
0000 0001 0010 0011 1100 1101 1110 1111 1000 1001 1011 1010
```

该例子证实了当缓冲二进制流关闭时，其包装的原始二进制流也会自动关闭。

最后，下面的例子说明了BytesIO的用法：

```
$ python3

>>> import io
>>> bytes_stream = io.BytesIO(b'veni,vidi,vici')
>>> bytes_stream.readable()
True
>>> bytes_stream.writable()
True
>>> bytes_stream.seekable()
True
>>> type(bytes_stream.getbuffer())
<class 'memoryview'>
>>> bytes_stream.getvalue()
b'veni,vidi,vici'
>>> bytes_stream.tell()
0
>>> bytes_stream.read(4)
b'veni'
>>> bytes_stream.write(b' vidi vici')
10
>>> bytes_stream.getvalue()
b'veni vidi vici'
>>> bytes_stream.close()
>>>
```

该例子只验证了BytesIO对象的getbuffer()属性会返回一个memoryview对象。第10章会给出通过内存视图操作BytesIO对象的方法。

7-6. 文本流

（标准库：io）

文本流是对缓冲二进制流的包装，核心区别在于引入将字节序列解读为字符序列的机制，以使我们能够以字符为单位进行读写操作。此外，文本流还引入了自己的额外缓冲区，即“文本缓冲区（text buffer）”。

抽象基类TextIOBase也是IOBase的直接子类。表7-17和表7-18分别列出了TextIOBase实现的属性和抽象属性。此外，TextIOBase重写了readlines()、readline()和writelines()，使它们直接处理字符串而非类字节对象。可以通过内置函数isinstance()判断一个文件对象是否是TextIOBase的实例，进而确定该文件对象是否代表一个文本流。

表7-17. TextIOBase实现的属性

属性	说明
`encoding`	一个字符串或None，指定字符编码方式。
`errors`	一个字符串或None，指定遇到当前编码方式不能编码的字符时如何处理。
`newlines`	一个字符串或None，指定将什么字符组合视为EOL。

表7-18. TextIOBase的抽象属性

属性	说明
`read()`	读取并返回至多指定个字符。
`write()`	写入指定字符串，并返回成功写入的字符数。
`detach()`	使文本缓冲区与缓冲二进制流分离，并返回缓冲二进制流。

encoding、errors和newlines合在一起限定了文本流将字节序列转换为字符序列的方式。encoding指定它采用哪种字符编码方式，如果引用None则表示使用操作系统的本地字符编码方式（等价于locale.getpreferredencoding(False)的返回值）。

这里有必要说明的是Python解释器的UTF-8模式。前面已经说明，UTF-8模式被启用后状态标志utf8_mode将取值1。而UTF-8模式的启用和禁用有如下几种控制手段：

- LC_CTYPE环境变量取值C或POSIX将启用，取其他值将禁用。
- PYTHONUTF8环境变量取值1将启用，取值0将禁用。
- 添加选项“-X utf8”将启用，添加选项“-X utf8=0”将禁用。

上述三种手段的优先级依次增加，也就是说如果有PYTHONUTF8环境变量，则忽略LC_CTYPE环境变量；如果有“-X utf8”选项，则忽略PYTHONUTF8环境变量。

当启用了UTF-8模式后，不论本地字符编码方式是什么，locale.getpreferredencoding()都会返回“UTF-8”，因此当文本流的encoding属性引用None时将采用UTF-8编码。此外，在这种情况下sys.getfilesystemencoding()也会返回“UTF-8”。

errors指定遇到无法用当前编码方式表示的字符时如何处理，可引用的对象包括：

- “strict”或None：抛出ValueError异常。
- “ignore”：忽略，不做任何替换。（这会导致数据丢失。）
- “replace”：替换为某个代表替换标记的字符，例如“?”。（这会导致数据丢失。）
- “surrogateescape”：替换为Unicode替代码位（U+DC80~U+DCFF）。
- “backslashreplace”：替换为Python转义序列（\xhh、\uhhhh或\Uhhhhhhhh）。
- “namereplace”：替换为包含字符名称的转义序列（\N{name}）。（仅适用于写入。）
- “xmlcharrefreplace”：替换为XML字符引用（&#nnn;）。（仅适用于写入。）

需要强调的是，如果采用utf-8编码则不会遇到无法表示的字符。

newlines可以引用“\n”、“\r”、“\r\n”、空串和None之一。当newlines取默认值None时行为是这样的：将\n、\r和\r\n都视为EOL（这被称为“通用换行”），执行读取操作时将它们统一转换为\n，执行写入操作时将\n转换为操作系统的默认EOL。newlines引用空串时，依然采用通用换行，但读写操作都不执行任何转换。newlines引用“\n”、“\r”或“\r\n”时，仅将该字符串视为EOL，且读写操作都不执行任何转换。

TextIOBase的read()和write()的语法为：

```
io.TextIOBase.read(size=-1)
io.TextIOBase.write(s)
```

它们会在内部调用BufferedIOBase的read()和write()，但因为增加了字节序列和字符序列之间的转换，所以速度要慢得多。此外，执行这种转换时可以使用文本流自己的文本缓冲区来存放字符序列，此时缓冲二进制流的缓冲区成为了“下层缓冲区（lower buffer）”。调用文本流的detach()只会使文本缓冲区与缓冲二进制流分离，而不会使下层缓冲区与原始二进制流分离。调用文本流的close()会导致其包装的缓冲二进制流也自动调用close()。

TextIOWrapper是TextIOBase的直接子类，且可被实例化，相应语法为：

```
    class io.TextIOWrapper(buffer, encoding=None, errors=None,
newline=None, line_buffering=False, write_through=False)
```

其中buffer参数是一个已经存在的缓冲二进制流；参数encoding、errors和newline分别用于指定字符编码方式、不能编码的字符处理方式和EOL；参数line_buffering用于指定是否启用“行缓冲区（line buffer）”，如果启用则在写入时每遇到一个EOL或在交互模式按下

Enter键都会自动调用flush()；参数write_through用于指定是否禁用文本缓冲区，如果禁用那么写入的字符序列会被实时转换为字节序列并写入下层缓冲区。

表7-19列出了TextIOWrapper实现的属性。值得特别说明的是reconfigure()，其语法为：

```
io.TextIOWrapper.reconfigure(*[, encoding][, errors][, newline[, line_buffering][, write_through])
```

它的所有形式参数都是仅关键字形式参数，分别用于重设文本流的字符编码方式、不能编码的字符处理方式、EOL、行缓冲区和文本缓冲区。省略的形式参数意味着保留当前设置，但errors是个例外，不显式指定将默认使用“strict”。此外，如果已经执行了读取操作，则无法改变字符编码方式和EOL的设置。

表7-19. TextIOWrapper实现的属性

属性	说明
`line_buffering`	布尔值，代表是否启用行缓冲区。
`write_through`	布尔值，代表是否禁用文本缓冲区。
`reconfigure()`	改变文本流的基本参数设置。

最后，与BytesIO类似，StringIO同样是TextIOBase的直接子类，但用于创建一种特殊的文本流——不包装已有的缓冲二进制流，而是代表文本缓冲区本身，其实例化语法为：

```
class io.StringIO(initial_value='', newline='\n')
```

其中initial_value参数用于设置文本缓冲区的初始值，默认为空的文本缓冲区；而newline用于指定EOL。注意newline参数的影响其实是这样的：当从文本缓冲区读取数据时，不做任何转换，但识别newline指定的EOL；当向文本缓冲区写入数据时，将\n、\r和\n\r都转换为newline指定的EOL。易知，newline参数只能是“\n”、“\r”或“\n\r”。

由于不涉及I/O操作，所以StringIO对象的encoding、errors和newlines属性没有意义，也不存在行缓冲区和下层缓冲区。StringIO实现了getvalue()属性，被总结在表7-20中。此外，StringIO也不支持detach()。

表7-20. StringIO实现的属性

属性	说明
`getvalue()`	以字符串的形式返回文本缓冲区的内容。

下面用一些例子来验证文本流的使用。首先通过TextIOWrapper来访问test2.txt：

```
$ python3

>>> import io
>>> raw_stream = io.FileIO('test2.txt', 'r+')
>>> buf_stream = io.BufferedRandom(raw_stream)
>>> txt_stream = io.TextIOWrapper(buf_stream, 'utf-8')
>>> txt_stream.encoding
'utf-8'
>>> txt_stream.errors
'strict'
>>> txt_stream.newlines
>>> txt_stream.line_buffering
False
>>> txt_stream.write_through
False
>>> txt_stream.reconfigure(line_buffering=True)
>>> txt_stream.line_buffering
True
>>> txt_stream.tell()
0
>>> txt_stream.readlines()
['你好\n', '\n', 'こんにちは\n', '\n', 'Hello\n', '\n', 'Bonjour\n', '\n',
'GutenTag\n']
>>> txt_stream.writelines(['\n', 'привет\n'])
>>> txt_stream.seek(0)
0
>>> txt_stream.readline()
'你好\n'
>>> txt_stream.tell()
7
>>> txt_stream.read()
'\nこんにちは\n\nHello\n\nBonjour\n\nGutenTag\n\nпривет\n'
>>> txt_stream.write('\nCiao\n')
6
>>> txt_stream.seek(0)
0
>>> txt_stream.read()
'你好\n\nこんにちは\n\nHello\n\nBonjour\n\nGutenTag\n\nпривет\n\nCiao\n'
>>> det_stream = txt_stream.detach()
>>> det_stream == buf_stream
True
>>> txt_stream.closed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>> buf_stream.closed
False
>>> buf_stream.close()
>>>
```

下面的例子说明了StringIO的用法：

```
$ python3

>>> import io
>>> str_stream = io.StringIO('veni,vidi,vici', '\r')
>>> str_stream.readable()
```

```
True
>>> str_stream.writable()
True
>>> str_stream.seekable()
True
>>> str_stream.getvalue()
'veni,vidi,vici'
>>> str_stream.tell()
0
>>> str_stream.read(4)
'veni'
>>> str_stream.write('\nvidi\rvici')
10
>>> str_stream.getvalue()
'veni\rvidi\rvici'
>>> str_stream.seek(0)
0
>>> str_stream.readlines()
['veni\r', 'vidi\r', 'vici']
>>> str_stream.close()
>>>
```

7-7. 创建流的便捷方式

（教程：7.2）
（标准库：内置函数、io）

尽管我们可以通过上面例子中的方法指定创建哪种流，但除了创建FileIO对象、BytesIO对象和StringIO对象之外都需要多条语句，相对比较麻烦。为解决这一问题，io模块提供了open()函数，而内置函数open()则是对io.open()的包装，语法为：

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None closefd=True, opener=None)
```

该函数通过mode和buffering参数共同控制创建哪种流。

open()的mode参数在io.FileIO()的mode参数的基础上添加了字符“b”和“t”，分别表示二进制流和文本流。显然，“b”和“t”是相互矛盾的，而它们都被省略的情况下相当于默认添加了“t”。而buffering参数可以被传入下列对象：

- -1：采用默认缓冲策略，即对交互模式下的文本流启用行缓冲区，在其他情况下则开辟io.DEFAULT_BUFFER_SIZE大小的（下层）缓冲区。（这等于操作系统执行I/O操作时的一个数据块的大小，通常为4kB或8kB。）
- 0：不使用任何缓冲区。
- 1：启用行缓冲区。
- 2以上的正整数：开辟buffering字节大小的（下层）缓冲区。

而这两个参数的不同组合决定了open()返回的文件对象属于哪个类，被总结在表7-21中。

表7-21. open()返回的文件对象类型

类型	mode	buffering
FileIO	'rb', 'r+b', 'wb', 'w+b', 'xb', 'x+b', 'ab', 'a+b'	0
BufferedReader	'rb'	-1, 2以上的正整数。
BufferedWriter	'wb', 'xb', 'ab'	-1, 2以上的正整数。
BufferedRandom	'r+b', 'w+b', 'x+b', 'a+b'	-1, 2以上的正整数。
TextIOWrapper	'r', 'r+', 'w', 'w+', 'x', 'x+', 'a', 'a+', 'rt', 'r+t', 'wt', 'w+t', 'xt', 'x+t', 'at', 'a+t'	-1, 1, 2以上的正整数。

不论返回哪种类型的文件对象，open()在执行过程中都需要在内部调用io.FileIO()，因此参数file（对应io.FileIO()的name参数）、closefd和opener都是为io.FileIO()提供的。而参数encoding、errors和newline则仅在创建文本流时才需要提供。

在开发多线程的Python程序时，必须考虑多个线程访问同一文件或外围设备时产生的冲突。由于FileIO仅使用了系统调用，而操作系统已经处理好了I/O冲突，所以它是线程安全的，而其实例是可重入的。为解决多个线程访问同一缓冲区带来的冲突，BufferedReader、BufferedWriter、BufferedRandom和BufferedRWPair在内部使用了锁，因此它们也是线程安全的。但上述类的实例是不可重入的，如果某个线程尝试重入已经和原始二进制流绑定的缓冲区会引发RuntimeError异常。TextIOWrapper不是线程安全的，其实例也不是可重入的。

BytesIO和StringIO不涉及I/O操作，每次都会创建独立的缓冲区，因此它们的实例是可重入的。它们在内部也使用了锁，因此也是线程安全的。

关于多线程、线程安全和可重入这三个概念，请在讨论操作系统的书籍中了解更多。

7-8. 标准I/O

（标准库：内置函数、sys）

上面讨论的I/O模型适用于所有I/O操作，然而如果所有Python脚本都需要通过open()创建流之后才能进行I/O，将非常的麻烦。另一方面，当以交互式启动Python解释器时，如果没有已经存在的流用来与用户交互，那么用户就无法输入包括调用open()在内的任何语句，进而陷入瘫痪状态。“标准I/O（standard I/O）”是作为现代操作系统鼻祖的Unix为解决这一问题而引入的，后来的操作系统都沿袭了这一传统。

标准I/O指的是任何应用程序启动后都会自动创建三个文本流：“标准输入（standard input）”、“标准输出（standard output）”和“标准出错（standard error）”。最初这三个流都默认关联到终端，但在终端成为虚拟概念后，它们可以关联不同的外围设备，例如标准输入通常关联到键盘，标准输出和标准出错则通常关联到显示器或打印机。不论怎

样，标准I/O的默认设置是为了便于与用户交互。我们可以使用“重定向（redirection）”技术将标准I/O的三个文本流关联到其他文件或外围设备，但这并不影响标准I/O的内在逻辑。

任何模块都可以访问标准I/O，不论它是否在顶层环境中被执行。表7-22列出了与标准I/O相关的sys属性。

表7-22. 标准I/O相关sys属性

属性	说明
sys.stdin	引用标准输入。
sys.stdout	引用标准输出。
sys.stderr	引用标准出错。
sys.displayhook(*value*)	如果value不是None，则将repr(value)写入标准输出。
sys.__stdin__	引用sys.stdin的初始值。
sys.__stdout__	引用sys.stdout的初始值。
sys.__stderr__	引用sys.stderr的初始值。
sys.__displayhook__()	引用sys.displayhook的初始值。

请通过下面的命令行和语句验证sys.stdin、sys.stdout和sys.stderr：

```
$ python3

>>> sys.stdin
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
>>> sys.stdin.encoding
'utf-8'
>>> sys.stdin.errors
'strict'
>>> sys.stdin.newlines
>>> sys.stdin.line_buffering
True
>>> sys.stdin.write_through
False
>>>
>>> sys.stdout
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
>>> sys.stdout.encoding
'utf-8'
>>> sys.stdout.errors
'strict'
>>> sys.stdout.newlines
>>> sys.stdout.line_buffering
True
>>> sys.stdout.write_through
False
>>>
>>> sys.stderr
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>
>>> sys.stderr.encoding
```

```
'utf-8'
>>> sys.stderr.errors
'backslashreplace'
>>> sys.stderr.newlines
>>> sys.stderr.line_buffering
True
>>> sys.stderr.write_through
False
>>>
```

从上面的例子可以看出，标准输入是'r'模式的TextIOWrapper对象，标准输出和标准出错都是'w'模式的TextIOWrapper对象。它们的默认字符编码方式都是UTF-8，不论本地字符编码方式是什么。遇到当前字符编码方式不能表示的字符时，标准输入和标准输出默认抛出ValueError异常，而标准出错则默认会将其替换为Python转义序列。我们可以通过环境变量PYTHONIOENCODING改变上述默认设置，它的取值是“encodingname:errorhandler”格式的字符串，冒号前面部分指定字符编码方式，后面部分指定不能编码字符处理方式。但要注意，标准出错永远会将不能编码字符替换为Python转义序列，这一行为不能通过环境变量PYTHONIOENCODING改变。此外，标准输入、标准输出和标准出错都采用通用换行，并会自动进行EOL转换。

标准输入、标准输出和标准出错总是会启用行缓冲区，因为它们是用来与用户交互的。与其他文件对象不同，标准输入、标准输出和标准出错之间有密切的关联：

➤ 标准输入每读取到一个字符，就会将该字符写入标准输出的行缓冲区，以使得用户能确定自己刚输入的字符是什么。该机制被称为“回显（echo）”。
➤ 用户按下Enter键（或者通过其他方式提供了EOL）后，行缓冲区中的内容就会被正式提交给Python解释器，而Python解释器会将其当成一条语句（或一条语句的一部分）来执行。在这之前，行缓冲区中的内容是可以修改的。
➤ 执行语句的结果会被传入value参数来调用sys.displayhook()，这会导致执行该语句所得到的对象被写入标准输出来显示。而返回None的语句则会跳过该操作。
➤ 如果执行语句抛出了异常或警告，则会写入标准出错。此外，一些特殊信息（例如Python解释器启动时显示的版本信息）也会写入标准出错。

下面通过一些例子来验证上述论断。首先让标准输入通过调用read()读取10个字符：

```
$ python3

>>> import sys
>>> sys.stdin.read(10)
hello
world
'hello\nworl'
>>>
```

在调用read()之后，提示符会消失，表示等待用户提供标准输入可读取的字符串。此时通常会通过键盘操作行缓冲区，每输入一个字符都会回显一个字符，直到按下Enter键才会将行缓

冲区中的内容提交。在上面的例子中，第一次提交只有6个字符（包括EOL），而read()期望读取10个字符，因此会继续等待，直到第二次又提交了6个字符，read()才返回读取到的10个字符，而该字符串又被传入value参数以调用sys.displayhook()，导致它被写入标准输出。

接下来通过调用write()向标准输出写入一个字符串：

```
$ python3

>>> import sys
>>> sys.stdout.write('To be or not to be.\n')
To be or not to be.
20
>>>
```

注意write()会返回成功写入标准输出的字符数，而该数值同样会被传入value参数以调用sys.displayhook()，导致它也被写入标准输出。

最后通过调用write()向标准出错写入一个字符串：

```
$ python3

>>> import sys
>>> sys.stderr.write('Report an error.\n')
Report an error.
17
>>>
```

由于标准出错默认也关联到显示器，所以写入标准出错的数据也可以直接看到。而write()返回的成功写入字符数则依然被传入value参数以调用sys.displayhook()，导致被写入标准输出。

从上面的例子可以看出，sys.displayhook()对于标准输出至关重要，其语法为：

sys.displayhook(*value*)

其默认行为是：如果value参数被传入None，则什么也不做，立刻返回；否则将repr(value)写入标准输出，然后再向标准输出写入一个EOL，最后让保留标识符“_”引用value。注意标准出错并没有类似的勾子函数相关联，这是因为错误信息是自动生成的且总是一个字符串，也没有哪个保留标识符会指向最近一次错误信息。

我们可以通过自定义sys.displayhook()来控制写入标准输出的信息，而sys.__displayhook__永远引用具有上述默认行为的函数。请看下面的例子：

```
$ python3

>>> import sys
>>> def f(value):
```

```
...     sys.stderr.write(repr(value.__class__) + '\n')
...     return sys.__displayhook__(value)
...
>>> sys.displayhook = f
>>> a = 0
>>> a
<class 'int'>
0
>>>
```

该例子使得将一个对象通过标准输出显示时，不仅会显示该对象本身，还会显示它所属类。

sys.displayhook()在内部调用了内置函数repr()，其语法为：

```
repr(object)
```

该内置函数的功能是调用object.__repr__，得到一个字符串。第3章已经说明，object实现了__repr__，因此repr()总是能返回一个字符串。我们也可以通过自定义__repr__来控制一个对象被标准输出显示时的结果，例如：

```
$ python3

>>> class Coordinate:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __repr__(self):
...         return '(' + str(self.x) + ',' + str(self.y) + ')'
...
>>> p1 = Coordinate(-100, 200)
>>> p1
(-100,200)
>>>
```

接下来讨论重定向技术对标准I/O的影响。在启动Python解释器时，可以通过shell命令行的语法分别用“<”、“>”和“2>”重定向Python解释器的标准输入、标准输出和标准出错。（请查阅关于Unix或DOS的书籍以获得更详细的说明。）需要强调的是，当Python解释器启动后，标准I/O的初始设置不仅会对sys.stdin、sys.stdout和sys.stderr造成影响，还会被sys.__stdin__、sys.__stdout__和sys.__stderr__记录下来。然而后者并不是标准I/O的组成部分，只是一些状态记录器。在Python脚本的运行过程中，可以动态改变前者以实现重定向，但应保持后者不变以反映标准I/O的初始设置。

请看下面的例子：

```
$ python3 2> error.log1

>>> import sys
>>> a
>>> sys.__stderr__ == sys.stderr
True
>>> sys.stderr = open('error.log2', 'w')
>>> a
>>> sys.__stderr__ == sys.stderr
False
>>> sys.stderr = sys.stdout
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> sys.__stderr__ == sys.stderr
False
>>>
```

在该例子中，启动Python解释器时标准出错被重定向到文件error.log1，然后对未定义变量a的访问会导致将如下出错信息写入error.log1：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

而此时sys.__stderr__和sys.stderr引用的是同一个TextIOWrapper对象。随后标准出错被动态地重定向到文件error.log2，再次访问a会将同样的出错信息写入error.log2，而现在sys.__stderr__和sys.stderr引用的是不同的TextIOWrapper对象。最后标准出错被动态地重定向到标准输出，使得访问a会将出错信息写入标准输出。同样，此时sys.__stderr__和sys.stderr引用的是不同的TextIOWrapper对象。在整个过程中，sys.__stderr__引用的TextIOWrapper对象都指向文件error.log1。

除了将执行语句得到的对象通过标准输出显示外，使用标准I/O的更规范方法是通过内置函数input()从标准输入读取数据，通过内置函数print()向标准输出写入数据。在本书前面的例子中我们已经多次用到了这两个内置函数，下面将详细讨论它们的细节。

input()的语法为：

```
input([prompt])
```

其中可选的prompt参数能被传入一个字符串，如果该参数没有被省略，则会被写入标准输出，且不会自动添加一个EOL。不论是否省略prompt参数，input()都会等待用户通过标准输

入提交一行文本，然后它将去掉该行文本的EOL，将剩下的部分转换为一个字符串并返回。用户提交的文本中不能含有EOF，否则input()不会返回，而会抛出EOFError异常。

print()的语法则为：

```
    print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

其中objects参数可被传入任意多个对象，这些对象会被依次输出，输出时相当于以该对象为参数调用了str()内置函数（在第10章讨论）以得到一个字符串。这些字符串之间用sep参数指定的分隔符分隔，默认分隔符为一个空格。输出完所有字符串后，会追加一个通过end参数指定的字符串，默认为\n。file参数用于指定将上述内容写入哪个文本流，默认写入标准输出。flush参数则表明是否在完成写入后强制调用流的flush()，默认由流自己决定。

input()是Python脚本获得用户输入的最基本方式，而print()是Python脚本向用户报告运算结果的最基本方式。本书之前的例子中已经多次使用input()和print()，下面再给出一个例子：

```
    $ python3

    >>> s = input('请输入一个不少于3个字符的字符串: ')
    请输入一个不少于3个字符的字符串: abc
    >>> print(s[0], s[1], s[2])
    a b c
    >>> print(s[0], s[1], s[2], sep='@')
    a@b@c
    >>> print(s[0], s[1], s[2], sep=',', end='End')
    a,b,cEnd>>>
```

7-9. site模块

（教程：16.1.4）
（标准库：内置常量、site）

除了sys模块和builtins模块，Python解释器启动后还会自动加载site模块。site模块负责完成解释器的“站点设置（site-specific configuration）”。同样必须通过import语句导入site模块才能获得引用该模块的标识符。

site模块定义了main()函数，其功能就是完成解释器的站点设置。如果在启动Python解释器时没有添加-S选项，则site.main()会被自动调用；否则，可以在解释器启动之后通过手工调用site.main()动态完成站点设置。站点设置的核心任务是将指向存放第三方分发包（将在第17章解释）的目录的路径添加到sys.path。

第6章已经说明，sys.prefix和sys.exec_prefix与sys.platlibdir结合形成两个路径，它们指向的两个目录下有一个专门存放第三方分发包的子目录，在Unix和类Unix操作系统上是python3.11/site-packages，在Windows上则是site-packages。这两个目录对应的路径会

被site.main()添加到sys.path中，它们对于所有用户都有效。除此之外，下列基路径与sys.platlibdir形成的路径指向的目录被用于存放只有登录用户可以使用的模块，用于完成用户站点设置：

- ~/.local/：适用于大部分Unix和类Unix操作系统。
- ~/Library/Python/3.11/：适用于macOS X。
- %APPDATA%/Python/：适用于Windows。

而该些目录下的python3.11/site-packages、python/site-packages或site-packages子目录被用于存放仅登录用户可以使用的第三方分发包。不过安装Python时默认不会创建上述与用户站点设置有关的目录，除非我们后来手工创建，或者安装第三方分发包时指定仅我们自己可以使用。

site.PREFIXES常量引用一个列表，其内包含了sys.prefix和sys.exec_prefix当前引用的对象。site.USER_BASE常量引用一个字符串，为指向用户站点设置相关目录的路径。site.USER_SITE常量也引用一个字符串，为指向存放仅登录用户可以使用的第三方分发包的目录的路径。请通过下面的命令行和语句验证：

```
      $ python3

      >>> import site
      >>> site.PREFIXES
      ['/Library/Frameworks/Python.framework/Versions/3.11', '/Library/
Frameworks/Python.framework/Versions/3.11']
      >>> site.USER_BASE
      '/Users/wwy/Library/Python/3.11'
      >>> site.USER_SITE
      '/Users/wwy/Library/Python/3.11/lib/python/site-packages'
      >>>
```

当site.main()被调用，默认会将所有存放第三方分发包的目录都添加到sys.path中。如果在启动Python解释器时添加了-s选项，则site.USER_SITE指定的目录不会被添加到sys.path中。常量site.ENABLE_USER_SITE引用一个布尔值，表明是否已经将site.USER_SITE指定的目录添加到sys.path中。请通过如下命令行和语句验证（“mkdir -p”命令行仅适用于Unix和类Unix操作系统，在Windows上需替换为相应DOS命令，或通过GUI创建所需目录）：

```
      $ mkdir -p /Users/wwy/Library/Python/3.11/lib/python/site-packages
      $ python3

      >>> import sys, site
      >>> site.ENABLE_USER_SITE
      True
      >>> sys.path
      ['', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python311.zip', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
lib-dynload', '/Users/wwy/Library/Python/3.11/lib/python/site-packages', '/
Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages']
```

```
      >>>

      $ python3 -s

      >>> import sys, site
      >>> site.ENABLE_USER_SITE
      False
      >>> sys.path
      ['', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python311.zip', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
site-packages']
      >>>
```

在将上述目录添加到sys.path的过程中，site.main()还会在这些目录下寻找后缀名为.pth的文本文件。如果找到则将其内的每行（除去空行和“#”开头的注释行）视为一个路径：以“/”开头的行被视为绝对路径，其他行视为相对路径。如果这些路径中某些指向的目录存在则将其添加到sys.path中。

请通过下面的命令行和语句验证.pth文件的作用（这些命令行仅适用于Unix和类Unix操作系统，在Windows上需替换为相应DOS命令，或通过GUI完成所需目录和文件的创建）：

```
      $ cd /Users/wwy/Library/Python/3.11/lib/python/site-packages
      $ mkdir foo bar
      $ cat > a.pth
      # Adding folders foo and bletch
      foo
      bletch
      ^D
      $ cat > b.pth
      /Users/wwy/Library/Python/3.11/lib/python/site-packages/bar
      ^D

      $ python3

      >>> import sys
      >>> sys.path
      ['', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python311.zip', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
lib-dynload', '/Users/wwy/Library/Python/3.11/lib/python/site-packages', '/Users/
wwy/Library/Python/3.11/lib/python/site-packages/foo', '/Users/wwy/Library/
Python/3.11/lib/python/site-packages/bar', '/Library/Frameworks/Python.framework/
Versions/3.11/lib/python3.11/site-packages']
      >>>
```

注意“#”开头的注释行被忽略，而由于bletch目录不存在所以被跳过。

在向sys.path添加路径时，site.main()会检查该路径是否已经被添加，如果是则跳过，故sys.path中不会有重复的路径（除非后来sys.path又被修改）。site.main()不会检查一个路径指向的是一个目录还是一个文件（因为ZIP归档文件可以被当成目录使用），所以我们必须自己确保这些路径有效。此外，sys.path中的路径越多，加载模块时的查找速度就越慢，所以应避免sys.path包含太多路径。

在完成了上述操作后，site.main()会自动尝试导入sitecustomize模块，以完成全局个性化设置。一般而言，sitecustomize.py是由系统管理员创建的，并被放在sys.prefix或sys.exec_prefix指定的基目录内存放第三方分发包的子目录下。如果找不到sitecustomize模块，则会抛出ImportError异常，该异常的name属性被设置为“sitecustomize”，而这导致该异常进入默认异常处理流程后会被自动忽略。

在这之后，如果启动Python解释器时没有添加-s选项，则site.main()还会尝试导入usercustomize模块，以完成用户个性化设置。usercustomize.py是由用户自己创建的，并被放在site.USER_SITE记录的目录下。请通过如下命令行和语句验证：

```
$ cd /Users/wwy/Library/Python/3.11/lib/python/site-packages
$ cat > usercustomize.py
print('Hi')
^D

$ python3
Hi

>>> ^D

$ python3 -s

>>> ^D


$ rm -R *
```

注意最后一条shell命令“rm -R *”清空了/Users/wwy/Library/Python/3.11/lib/python/site-packages目录，但该目录本身被保留。

最后，在Unix或类Unix操作系统上，如果shell支持GNU readline，则sys.main()还会自动导入readline模块并配置rlcompleter模块，它们都属于标准库。

以上就是通过调用site.main()完成的站点设置。而site.main()还会在内部自动调用一些site模块定义的函数。这些函数以及site.main()都被总结在表7-23中。

表7-23. site模块定义的函数

属性	说明
site.main()	完成站点设置。
site.addsitedir()	将指定的目录及其下的.pth文件指定的目录添加到sys.path中。
site.getsitepackages()	设置site.PREFIXES常量，并将其返回。
site.getuserbase()	设置site.USER_BASE常量，并将其返回。
site.getusersitepackages()	设置site.USER_SITE常量，并将其返回。

我们也可以直接调用表7-23列出的函数，例如通过调用site.addsitedir()临时将额外的目录添加到sys.path中，其语法为：

site.addsitedir(*sitedir*, *known_paths*=None)

我们甚至还可以通过-m选项直接执行site模块，当没有给出任何参数时，会将sys.path、site.USER_BASE、site.USER_SITE和site.ENABLE_USER_SITE的当前值写入标准输出，并报告相关目录是否存在。如果在模块名后追加--user-base，则仅会写入site.USER_BASE相关信息。如果在模块名后追加--user-site，则仅会写入site.USER_SITE相关信息。请通过如下命令行验证：

```
      $ python3 -m site
      sys.path = [
          '/Users/wwy',
          '/Library/Frameworks/Python.framework/Versions/3.11/lib/python311.zip',
          '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11',
          '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/lib-
dynload',
          '/Users/wwy/Library/Python/3.11/lib/python/site-packages',
          '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
site-packages',
      ]
      USER_BASE: '/Users/wwy/Library/Python/3.11' (exists)
      USER_SITE: '/Users/wwy/Library/Python/3.11/lib/python/site-
packages' (exists)
      ENABLE_USER_SITE: True

      $ python3 -m site --user-base
      /Users/wwy/Library/Python/3.11

      $python3 -m site --user-site
      /Users/wwy/Library/Python/3.11/lib/python/site-packages
```

注意该结果中sys.path参数包含“/Users/wwy”，这是因为这是当前工作目录。如果以其他目录为工作目录执行上述命令行，则“/Users/wwy”会被替换成该目录。而以交互式启动Python解释器时，当前工作目录会被记录为空串。

最后，site模块还提供了一些内置常量，被总结在表7-24中。

表7-24. site模块实现的内置常量

属性	说明
quit(*code*=None)	打印此常量时，会将关于如何退出Python解释器的提示信息写入标准输出。当调用此常量时，会以code为参数调用sys.exit()。
exit(*code*=None)	
copyright	打印此常量时，会将版权信息写入标准输出。
credits	打印此常量时，会将贡献者信息写入标准输出。
license()	打印此常量时，会显示查看许可证的方法。调用此常量时，会以分页形式显示完整的许可证。

由于是内置常量，所以不需要导入site模块就可以使用。请通过如下命令行和语句验证：

```
$ python3

>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> quit()


$ python3

>>> exit('Ouch')
Ouch

$ python3

>>> copyright
Copyright (c) 2001-2022 Python Software Foundation.
All Rights Reserved

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>> credits
    Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a cast of
thousands
    for supporting Python development.  See www.python.org for more
information.
>>> license
Type license() to see the full license text
>>> license()
A. HISTORY OF THE SOFTWARE
==========================

Python was created in the early 1990s by Guido van Rossum at Stichting
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands
as a successor of a language called ABC.  Guido remains Python's
principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)
in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to
BeOpen.com to form the BeOpen PythonLabs team.  In October of the same
year, the PythonLabs team moved to Digital Creations, which became
Zope Corporation.  In 2001, the Python Software Foundation (PSF, see
https://www.python.org/psf/) was formed, a non-profit organization
created specifically to own Python-related Intellectual Property.
Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for
the Open Source Definition).  Historically, most, but not all, Python
Hit Return for more, or q (and Return) to quit: q
>>>
```