模块

杨资璋（翻译）

2021 年 10 月 11 日

如果你退出 Python 解释器并再一次进入，你做的定义（函数和变量）将会丢失。然而，如果你想写一个更长的程序，你最好使用一个文本编辑器来准备给解释器的输入并使用那个文件作为输入来运行。这被称为创建一个*脚本*。随着你的程序变得越来越长，你可能会想将它分为多个文件以方便维护。你也可能想使用一个你在多个程序中写过的一个方便的函数——在不将它的定义复制到每一个程序的情况下。

为了支持这个，Python 有一种在一个文件中定义并在脚本中或可交互的解释器实例中使用它们的方式。这样的文件被叫做*模块*；模块中的定义可以被*导入*到其他模块或者到*主*模块中。

一个模块就是一个包含 Python 定义和声明的文件。文件名就是模块名加上.py 后缀。在一个模块中，模块的名字（作为一个字符串）可以通过全局变量`__name__`得到。

1　模块的更多内容

模块既可以包含可执行语句也可以包含函数定义。这些语句意在初始化这个模块。它们仅在模块名第一次碰到导入语句时执行。（如果文件作为一个脚本执行它们也会运行。）

每个模块有自己的私有符号表——它被在这个模块中定义的所有函数用作全局符号表。因此，模块的作者在模块中使用全局变量时，不必担心与用户全局变量的意外冲突。另一方面，如果你知道你在干什么，你可以通过与指向它的函数相同的记号来使用模块的全局变量，`modname.itemname`。

模块可以导入其他模块。将所有导入语句放在模块（或者脚本）的开头是习惯性但不是必须的做法。被导入模块的名字被放在导入模块的全局符

号表中。

还有一种导入语句的变种——它将模块中的名字直接导入到导入模块的符号表中，例如：

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这不会将导入发生的模块名引入局部符号表（所以在这个例子中，fibo 没有被定义）。

甚至还有一个导入模块定义的所有名字的变种：

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这将导入除以下划线（_）开头的名字外的所有名字。在大多数情况下，由于它为解释器引入了一个未知的名字集合，所以可能会隐藏一些你已经定义的东西，因此 Python 程序员不会使用这种便利。

注意一般来说由于这经常会导致代码可读性降低，所以使用 * 从模块导入并不是一个被喜爱的操作。然而，在可交互时期使用它来节省输入是可以的。

注意：由于效率原因，在每个解释器时间每个模块只被导入一次。因此，如果你改变了你的模块，你必须重启解释器——或者，如果你想交互式地测试一个模块，使用 importlib.reload()，例如 import importlib; importlib.reload(modulename)。

1.1 像脚本一样执行模块

当你通过如下语句运行一个 Python 模块

```
python fibo.py <arguments>
```

模块中的代码将会被执行，正如同你导入了它，但是`__name__`将会被设置为`"__main__"`。这意味着通过在模块末尾加入如下语句

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

可以使得这个文件即可作为脚本也可作为可导入的模块，因为解析命令行的代码仅会在模块被当做“main”文件时执行。

这通常被用于为用户提供一个方便的接口或者用于测试意图（在测试样例上将模块作为脚本运行）。

1.2 模块搜索路径

当一个名为 spam 的模块被导入时，解释器首先在自建模块中搜索这个名字。如果没有找到，它接下来会在变量 sys.path 提供的目录列表中搜索名为`spam.py`的文件。sys.path 通过下面这些位置初始化：

- 包含输入脚本的目录（或者当没有指定文件时则为当前目录）
- PYTHONPATH（一个目录名的列表，与 shell 变量 PATH 有相同的语法）
- 与安装相关的默认（习惯上包含一个`site-packages`目录，被 site 模块处理）

注意：在支持 symlinks 的系统，包含输入脚本的目录指的是 symlinks 之后的目录。换句话说，包含 symlink 的目录**不会**包含在模块搜索路径中。

初始化后，Python 程序可以修改 sys.path。包含正在运行脚本的目录被放置在搜索路径的最开始，在标准库路径之前。这意味着在这个目录中的模块而不是库目录中的同名模块将会被载入。除非替换是有意为之，否则这将会是一个错误。更多信息见标准模块章节。

1.3 “编译好的” Python 文件

为了加快载入模块速度，Python 将会以 module.version.pyc 为名字，缓存`__pycache__`目录中每个模块的已编译版本，其中 version 编码了已编译文件的格式；它基本上包含 Python 版本号。例如，在 Cpython 发布的 3.3 已编译版本中的 spam.py 将会被缓存为`__pycache__/spam.cpython-33.pyc`。这种命名习俗允许不同发布和 Python 不同版本的已编译模块共存。

Python 会检查源的修改数据和已编译版本来查看它是不是已过时以及是否需要被重新编译。这是一个完全自动的过程。同时，已编译模块是与平台无关的，所以同一个库可以在不同架构的系统间共享。

Python 不会检查两个情况的缓存。首先，它总是为从命令行直接载入的模块重编译并不会存储结果。其次，如果没有源模块，它不会检查缓存。

为了支持无源（仅有已编译）分发，已编译模块必须在源目录下，同时这里也不能有源模块。

2 标准模块

Python 有一个有一些标准模块的库，它在一个单独的文档中描述，Python 库参考。一些模块被构造入解释器中；这提供了使用不是语言核心然而自建操作的途径，或者为了效率或者提供接触操作系统原型例如系统调用的途径。这样的模块集合是一个设置选项，取决于底层运行的平台。例如，winreg模块仅在视窗系统中提供。一个特殊的模块值得一些关注：sys，它自带在每一个 Python 解释器中。变量sys.ps1和sys.ps2定义了作为提示的首要和次要字符串:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C>'
C>print('Yuck!')
C>
```

这两个变量尽在交互模式下的解释器中被定义。

变量sys.path是决定解释器模块搜索路径的字符串列表。它首先通过从环境变量PYTHONPATH中得到的默认路径初始化，如果PYTHONPATH没有被设置，则通过自建的默认路径。你可以通过使用标准的列表操作修改它

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

3 dir() 函数

自建的 dir() 函数被用来查看模块定义的名字。它返回一个有序的字符串列表:

```
    >>> import fibo, sys
    >>> dir(fibo)
    ['__name__', 'fib', 'fib2']
    >>> dir(sys)
    ['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
```

```
    '__interactivehook__', '__loader__', '__name__', '__package__', '
    __spec__',
    '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
    '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework
    ',
    '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
    'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
    'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
    'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', '
    exc_info',
    'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info'
    ,
    'float_repr_style', 'get_asyncgen_hooks', '
    get_coroutine_origin_tracking_depth',
    'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
    'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
    'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
    'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
    'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
    'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
    'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', '
    pycache_prefix',
    'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', '
    setdlopenflags',
    'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', '
    stderr',
    'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', '
    version_info',
    'warnoptions']
```

如果不提供参数，dir() 将会展示你现在已经定义的名字：

```
    >>> a = [1, 2, 3, 4, 5]
    >>> import fibo
    >>> fib = fibo.fib
    >>> dir()
    ['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

注意它展示了所有类型的名字：变量、模块、函数等等。

dir() 不会展示自带函数和变量的名字。如果你想要这样的列表，它们定义自标准模块 buildins 中：

```
    >>> import builtins
    >>> dir(builtins)
    ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
    'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
    'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
```

```
    'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
    'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
    'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
    'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
    'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
    'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
    'MemoryError', 'NameError', 'None', 'NotADirectoryError', '
    NotImplemented',
    'NotImplementedError', 'OSError', 'OverflowError',
    'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
    'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
    'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
    'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
    'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
    'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning'
    ,
    'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
    '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
    'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
    'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
    'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
    'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr
    ',
    'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
    'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
    'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property
    ',
    'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
    'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars'
    ,
    'zip']
```

4 包

包是一种通过使用“带点模块名”来构建 Python 的模块命名空间的方法。例如，名为 A.B 的模块指定 A 包中的 B 模块。正如模块的使用使得不同模块的作者可以不必担心各自的全局变量名，带点模块名的使用使得例如 NumPy 或 Pillow 等多模块包的作者不必担心各自的模块名。

假设你想设计一个统一处理声音文件和声音数据的模块集合（一个“包”）。由于存在很多不同的声音文件格式（通常通过它们的扩展名辨别，例如，.wav,.aiff,.au），所以为了多种文件格式的转换，你可能需要创建并维护一个越来越大的模块集合。还存在许多你想在声音数据上进行的不

同的操作（例如混合，添加回音，应用均衡器函数，创建人工立体声），所以额外的，你需要写源源不断的模块来进行这些操作。这里是你的包的一个可能结构（通过层级化文件系统表达）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

当导入这个包时，Python 沿着`sys.path`中的目录寻找包子目录。

为了使得 Python 将包含文件的目录视为包，你需要`__init__.py`文件。这避免了有共同名字的目录，例如 string，无意间隐藏了之后将会在模块搜索路径中出现的有用模块。在最简单的情况下，`__init__.py`文件可以使一个空文件，但是它也可以为包执行初始化代码或者设置`__all__`变量。

包的用户可以从包导入单独的模块，例如：

```
import sound.effects.echo
```

这会加载子模块`sound.effects.echo`。它必须通过它的全名引用。

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

另外一种导入这个子模块的方法是：

```
from sound.effects import echo
```

这也会载入子模块`echo`，并且可以在没有包前缀的情况下使用它，所以它可以如此使用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

然而另外一种变种是直接导入想用的函数或变量：

```
from sound.effects.echo import echofilter
```

注意当使用`from package import item`时，item 既可以是包的子模块（或者子包），也可以是在包中定义的一些名字，例如函数，类或变量。`import`语句首先检查 item 是否在包中定义；如果没有，它假设它是一个模块，并尝试导入它。如果没有找到它，将会产生一个 ImportError 异常。

4.1 从一个包中导入 *

当用户写下`from sound.effects import *`时，会发生什么？理想情况下，人们会希望这会走出文件系统，找到这个包中的子模块，之后导入所有。这将花费很长时间并且导入子模块可能会有人们并不希望的仅应该在子模块被显式导入时才发生的副作用。

唯一的解决办法就是包的作者提供一个包的显式索引。import 语句使用如下传统：如果包的`__init__.py`代码定义了一个名为`__all__`的列表，那么发生`from package import *`时，它将会作为应该被导入的模块名字的列表。当这个包的新版本发布时，是否将这个列表更新取决于作者。如果包的作者没有发现`import *`用法，他们也可以选择不去支持它。例如，文件`sound/effects/__init__.py`可以包含如下代码：

```
__all_ = ['echo', 'surround', 'reverse']
```

这将意味着`from sound.effect import *`将会导入 sound 包的三个子模块。

如果没有定义`__all__`，`from sound.effects import *`语句**不会**从 sound.effects 包中导入所有的子模块；可以肯定的仅仅是包`sound.effects`是被导入了的（可能运行`__init__.py`中的任意初始化代码）同时导入了在包中定义的名字。这包括通过`__init__.py`定义的任何名字（以及显式导入的子模块）。它也会包括通过之前的导入语句显式载入的包的子模块。考虑以下例子：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

在这个例子中，由于`echo`和`surround`模块在`sound.effects`包中被定义，所以当`from ...import`执行后，它们会被导入到当前的命名空间。（当`__all__`被定义时，这也工作。）

尽管当你使用`import *`时，有些模块被设计来仅导出遵循特定模式的名字，但是在产品代码中使用它依旧是个不好的选择。

请记住，使用`from package import specific_submodule`没有任何问题！实际上，这才是推荐的记号除非导入的模块需要使用来自不同包的同名子模块。

4.2 包内引用

当包被构建为子包时（正如例子中的`sound`包），你可以使用绝对引用来指向兄弟包的子模块。例如，如果`sound.filters.vocoder`模块需要使用`sound.effects`包中的`echo`模块，它可以使用`from sound.effects import echo`。

你也可以通过`from module import name`形式的导入语句来使用相对引用。这些导入使用前导的点（.）来表示相对导入设计的当前和父包。在`surround`包的例子中，你可以使用：

```
from . import echo
from .. import formats
from ..filters import equalizer
```

注意相对引用是基于当前模块的名字。由于主模块的名字一直是`"__main__"`，所以 Python 应用中意欲用作主模块的模块必须一直使用绝对导入。

4.3 多个目录中的包

包支持一个额外的特殊属性，`__path__`。在文件中的代码执行前，它被初始化为包含包的`__init__.py`的目录的列表。这个变量可以被修改；这样做会影响将来对包含在这个包中的模块和子包的搜索。

虽然这个特性并不经常被需要，它可以被用来扩展一个包中找到的模块集合。