

## 6. 模块

退出 Python 解释器后，再次进入时，之前在 Python 解释器中定义的函数和变量就丢失了。因此，编写较长程序时，建议用文本编辑器代替解释器，执行文件中的输入内容，这就是编写 *脚本*。随着程序越来越长，为了方便维护，最好把脚本拆分成多个文件。编写脚本还有一个好处，不同程序调用同一个函数时，不用每次把函数复制到各个程序。

为实现这些需求，Python 把各种定义存入一个文件，在脚本或解释器的交互式实例中使用。这个文件就是 *模块*；模块中的定义可以 *导入* 到其他模块或 *主* 模块（在顶层和计算器模式下，执行脚本中可访问的变量集）。

模块是包含 Python 定义和语句的文件。其文件名是模块名加后缀名 `.py`。在模块内部，通过全局变量 `__name__` 可以获取模块名（即字符串）。例如，用文本编辑器在当前目录下创建 `fibonacci.py` 文件，输入以下内容：

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

现在，进入 Python 解释器，用以下命令导入该模块：

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current `namespace` (see [Python 作用域和命名空间](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果经常使用某个函数，可以把它赋值给局部变量：

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

### 6.1. 模块详解

模块包含可执行语句及函数定义。这些语句用于初始化模块，且仅在 `import` 语句第一次遇到模块名时执行。[\[1\]](#) (文件作为脚本运行时，也会执行这些语句。)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

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

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fib` is not defined).

还有一种变体可以导入模块内定义的所有名称：

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

这种方式会导入所有不以下划线（`_`）开头的名称。大多数情况下，不要用这个功能，这种方式向解释器导入了一批未知的名称，可能会覆盖已经定义的名称。

注意，一般情况下，不建议从模块或包内导入 `*`，因为，这项操作经常让代码变得难以理解。不过，为了在交互式编译器中少打几个字，这么用也没问题。

模块名后使用 `as` 时，直接把 `as` 后的名称与导入模块绑定。

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

与 `import fibo` 一样，这种方式也可以有效地导入模块，唯一的区别是，导入的名称是 `fib`。

`from` 中也可以使用这种方式，效果类似：

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

**注解：** 为了保证运行效率，每次解释器会话只导入一次模块。如果更改了模块内容，必须重启解释器；仅交互测试一个模块时，也可以使用 `importlib.reload()`，例如 `import importlib; importlib.reload(module_name)`。

### 6.1.1. 以脚本方式执行模块

可以用以下方式运行 Python 模块：

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

这项操作将执行模块里的代码，和导入模块一样，但会把 `__name__` 赋值为 `"__main__"`。也就是把下列代码添加到模块末尾：

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

既可以把这个文件当脚本使用，也可以用作导入的模块，因为，解析命令行的代码只有在模块以“main”文件执行时才会运行：

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

导入模块时，不运行这些代码：

```
>>> import fibo
>>>
```

这种操作常用于为模块提供便捷用户接口，或用于测试（把模块当作执行测试套件的脚本运行）。

### 6.1.2. 模块搜索路径

当一个名为 `spam` 的模块被导入时，解释器首先搜索具有该名称的内置模块。这些模块的名字被列在 `sys.builtin_module_names` 中。如果没有找到，它就在变量 `sys.path` 给出的目录列表中搜索一个名为 `spam.py` 的文件，`sys.path` 从这些位置初始化：

- 输入脚本的目录（或未指定文件时的当前目录）。
- `PYTHONPATH`（目录列表，与 `shell` 变量 `PATH` 的语法一样）。
- 依赖于安装的默认值（按照惯例包括一个 `site-packages` 目录，由 `site` 模块处理）。

**注解：** 在支持 `symlink` 的文件系统中，输入脚本目录是在追加 `symlink` 后计算出来的。换句话说，包含 `symlink` 的目录并没有添加至模块搜索路径。

初始化后，Python 程序可以更改 `sys.path`。运行脚本的目录在标准库路径之前，置于搜索路径的开头。即，加载的是该目录里的脚本，而不是标准库的同名模块。除非刻意替换，否则会报错。详见 [标准模块](#)。

### 6.1.3. “已编译的” Python 文件

为了快速加载模块，Python 把模块的编译版缓存在 `__pycache__` 目录中，文件名为 `module.version.pyc`，`version` 对编译文件格式进行编码，一般是 Python 的版本号。例如，CPython 的 3.3 发行版中，`spam.py` 的编译版本缓存在 `__pycache__/spam.cpython-33.pyc`。使用这种命名惯例，可以让不同 Python 发行版及不同版本的已编译模块共存。

Python 对比编译版本与源码的修改日期，查看它是否已过期，是否要重新编译，此过程完全自动化。此外，编译模块与平台无关，因此，可在不同架构系统之间共享相同的支持库。

Python 在两种情况下不检查缓存。其一，从命令行直接载入模块，只重新编译，不存储编译结果；其二，没有源模块，就不会检查缓存。为了支持无源文件（仅编译）发行版本，编译模块必须在源目录下，并且绝不能有源模块。

给专业人士的一些小建议：

- 在 Python 命令中使用 `-O` 或 `-OO` 开关，可以减小编译模块的大小。`-O` 去除断言语句，`-OO` 去除断言语句和 `__doc__` 字符串。有些程序可能依赖于这些内容，因此，没有十足的把握，不要使用这两个选项。“优化过的”模块带有 `opt-` 标签，并且文件通常会小一些。将来的发行版或许会改进优化的效果。
- 从 `.pyc` 文件读取的程序不比从 `.py` 读取的执行速度快，`.pyc` 文件只是加载速度更快。
- `compileall` 模块可以为一个目录下的所有模块创建 `.pyc` 文件。
- 本过程的细节及决策流程图，详见 [PEP 3147](#)。

## 6.2. 标准模块

Python 自带一个标准模块的库，它在 Python 库参考（此处以下称为“库参考”）里另外描述。一些模块是内嵌到编译器里面的，它们给一些虽并非语言核心但却内嵌的操作提供接口，要么是为了效率，要么是给操作系统基础操作例如系统调入提供接口。这些模块集是一个配置选项，并且还依赖于底层的操作系统。例如，`winreg` 模块只在 Windows 系统上提供。一个特别值得注意的模块 `sys`，它被内嵌到每一个 Python 编译器中。`sys.ps1` 和 `sys.ps2` 变量定义了一些字符，它们可以用作主提示符和辅助提示符：

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

只有解释器用于交互模式时，才定义这两个变量。

变量 `sys.path` 是字符串列表，用于确定解释器的模块搜索路径。该变量以环境变量 `PYTHONPATH` 提取的默认路径进行初始化，如未设置 `PYTHONPATH`，则使用内置的默认路径。可以用标准列表操作修改该变量：

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

### 6.3. `dir()` 函数

内置函数 `dir()` 用于查找模块定义的名称。返回结果是经过排序的字符串列表：

```
>>> 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_',
'_getfilesystemcodeerrors_', '_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_']
```

没有参数时，`dir()` 列出当前定义的名称：

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

注意，该函数列出所有类型的名称：变量、模块、函数等。

`dir()` 不会列出内置函数和变量的名称。这些内容的定义在标准模块 `builtins` 里：

```
>>> 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_']
```

## 6.4. 包

包是一种用“点式模块名”构造 Python 模块命名空间的方法。例如，模块名 `A.B` 表示包 `A` 中名为 `B` 的子模块。正如模块可以区分不同模块之间的全局变量名称一样，点式模块名可以区分 `NumPy` 或 `Pillow` 等不同多模块包之间的模块名称。

假设要为统一处理声音文件与声音数据设计一个模块集（“包”）。声音文件的格式很多（通常以扩展名来识别，例如：`.wav`，`.aiff`，`.au`），因此，为了不同文件格式之间的转换，需要创建和维护一个不断增长的模块集合。为了实现对声音数据的不同处理（例如，混声、

添加回声、均衡器功能、创造人工立体声效果），还要编写无穷无尽的模块流。下面这个分级文件树展示了这个包的架构：

```
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
    ...
```

导入包时，Python 搜索 `sys.path` 里的目录，查找包的子目录。

Python 只把含 `__init__.py` 文件的目录当成包。这样可以防止以 `string` 等通用名称命名的目录，无意中屏蔽出现在后方模块搜索路径中的有效模块。最简情况下，`__init__.py` 只是一个空文件，但该文件也可以执行包的初始化代码，或设置 `__all__` 变量，详见下文。

还可以从包中导入单个模块，例如：

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

这段代码加载子模块 `sound.effects.echo`，但引用时必须使用子模块的全名：

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

另一种导入子模块的方法是：

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

这段代码还可以加载子模块 `echo`，不加包前缀也可以使用。因此，可以按如下方式使用：

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

Import 语句的另一种变体是直接导入所需的函数或变量：

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

同样，这样也会加载子模块 `echo`，但可以直接使用函数 `echofilter()`：

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

注意，使用 `from package import item` 时，`item` 可以是包的子模块（或子包），也可以是包中定义的函数、类或变量等其他名称。`import` 语句首先测试包中是否定义了 `item`；如果未在包中定义，则假定 `item` 是模块，并尝试加载。如果找不到 `item`，则触发 `ImportError` 异常。

相反，使用 `import item.subitem.subsubitem` 句法时，除最后一项外，每个 `item` 都必须是包；最后一项可以是模块或包，但不能是上一项中定义的类、函数或变量。

#### 6.4.1. 从包中导入 \*

使用 `from sound.effects import *` 时会发生什么？理想情况下，该语句在文件系统查找并导入包的所有子模块。这项操作花费的时间较长，并且导入子模块可能会产生不必要的副作用，这种副作用只有在显式导入子模块时才会发生。

唯一的解决方案是提供包的显式索引。`import` 语句使用如下惯例：如果包的 `__init__.py` 代码定义了列表 `__all__`，运行 `from package import *` 时，它就是用于导入的模块名列表。发布包的新版本时，包的作者应更新此列表。如果包的作者认为没有必要在包中执行导入 `*` 操作，也可以不提供此列表。例如，`sound/effects/__init__.py` 文件包含以下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这将意味着将 `from sound.effects import *` 导入 `sound.effects` 包的三个命名的子模块。

如果没有定义 `__all__`，`from sound.effects import *` 语句 不会把包 `sound.effects` 中所有子模块都导入到当前命名空间；该语句只确保导入包 `sound.effects`（可能还会运行 `__init__.py` 中的初始化代码），然后，再导入包中定义的名称。这些名称包括 `__init__.py` 中定义的任何名称（以及显式加载的子模块），还包括之前 `import` 语句显式加载的包里的子模块。请看以下代码：

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

本例中，执行 `from...import` 语句时，将把 `echo` 和 `surround` 模块导入至当前命名空间，因为，它们是在 `sound.effects` 包里定义的。（该导入操作在定义了 `__all__` 时也有效。）

虽然，可以把模块设计为用 `import *` 时只导出遵循指定模式的名称，但仍不提倡在生产代码中使用这种做法。

记住，使用 `from package import specific_submodule` 没有任何问题！实际上，除了导入模块使用不同包的同名子模块之外，这种方式是推荐用法。

## 6.4.2. 子包参考

包中含有多个子包时（与示例中的 `sound` 包一样），可以使用绝对导入引用兄弟包中的子模块。例如，要在模块 `sound.filters.vocoder` 中使用 `sound.effects` 包的 `echo` 模块时，可以用 `from sound.effects import echo` 导入。

还可以用 `import` 语句的 `from module import name` 形式执行相对导入。这些导入语句使用前导句点表示相对导入中的当前包和父包。例如，相对于 `surround` 模块，可以使用：

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

注意，相对导入基于当前模块名。因为主模块名是 `"__main__"`，所以 Python 程序的主模块必须始终使用绝对导入。

## 6.4.3. 多目录中的包

包支持一个更特殊的属性 `__path__`。在包的 `:file: __init__.py` 文件中的代码被执行前，该属性被初始化为包含 `:file: __init__.py` 文件所在的目录名在内的列表。可以修改此变量；但这样做会影响在此包中搜索子模块和子包。

这个功能虽然不常用，但可用于扩展包中的模块集。

### 备注

- [1] In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function definition adds the function name to the module's global namespace.

© 版权所有 2001-2022, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

最后更新于 9月 15, 2022. [Found a bug?](#)

Created using [Sphinx 3.4.3](#).



3.10.7



转向