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，最好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。编写脚本还一个好处，不同程序调用同一个函数时，不用把函数定义复制到各个程序。

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 `fibo.py` 文件，输入以下内容：

```
# 斐波那契数列模块

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
```

此操作不会直接把 `fibo` 中定义的函数名称添加到当前 namespace 中（请参阅 Python 作用域和命名空间 了解详情）； 它只是将模块名称 `fibo` 添加到那里。 使用该模块名称你可以访问其中的函数:

```
>>> 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。)

每个模块都有它自己的私有符号表，该表被定义在该模块里的所有函数当作全局符号表使用。 因此，一个模块的作者可以在模块内放心使用全局变量，而不必担心它们会和模块使用者的全局变量发生意外冲突。 另一方面，如果您知道自己在做什么，您可以使用与引用模块函数相同的语法去访问一个模块的全局变量，即 `modname.itemname`。

模块可以导入其他模块。 根据惯例可以将所有 `import` 语句都放在模块（或者也可以说是脚本）的开头但这并非强制要求。 如果被放置于一个模块的最高层级，则被导入的模块名称会被添加到该模块的全局命名空间。

还有一种 `import` 语句的变化形式可以将来自某个模块的名称直接导入到导入方模块的命名空间中。 例如:

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

这条语句不会将所导入的模块的名称引入到局部命名空间中（因此在本示例中，`fibo` 将是未定义的名称）。

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，为了在交互式会话中少打几个字，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
```

备注： 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(modulename)`。

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]))
```

这个文件既能被用作脚本，又能被用作一个可供导入的模块，因为解析命令行参数的那两行代码只有在模块作为“main”文件执行时才会运行：

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

当这个模块被导入到其它模块时，那两行代码不运行：

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

这常用于为模块提供一个便捷的用户接口，或用于测试（把模块作为执行测试套件的脚本运行）。

6.1.2. 模块搜索路径

当导入一个名为 `spam` 的模块时，解释器首先会搜索具有该名称的内置模块。 这些模块的名称在 `sys.builtin_module_names` 中列出。 如果未找到，它将在变量 `sys.path` 所给出的目录列表中搜索名为 `spam.py` 的文件。 `sys.path` 是从这些位置初始化的:

- 被命令行直接运行的脚本所在的目录（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）。

更多细节请参阅 sys.path 模块搜索路径的初始化。

备注： 在支持符号链接的文件系统中，“被命令行直接运行的脚本所在的目录”是符号链接最终指向的目录。换句话说，符号链接所在的目录并 **没有** 被添加至模块搜索路径。

初始化后，Python 程序可以更改 `sys.path`。脚本所在的目录先于标准库所在的路径被搜索。这意味着，脚本所在的目录如果有和标准库同名的文件，那么加载的是该目录里的，而不是标准库的。这一般是一个错误，除非这样的替换是你有意为之。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 把模块的编译版本缓存在 `__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 的编译版本缓存为 `__pycache__/spam.cpython-33.pyc`。这种命名惯例让不同 Python 版本编译的模块可以共存。

Python 对比编译版与源码的修改日期，查看编译版是否已过期，是否要重新编译。此进程完全是自动的。此外，编译模块与平台无关，因此，可在不同架构的系统之间共享相同的库。

Python 在两种情况下不检查缓存。一，从命令行直接载入的模块，每次都会重新编译，且不储存编译结果；二，没有源模块，就不会检查缓存。为了让一个库能以隐藏源代码的形式分发（通过将所有源代码变为编译后的版本），编译后的模块必须放在源目录而非缓存目录中，并且源目录绝不能包含同名的未编译的源模块。

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，并且文件通常会一小些。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。 一些模块是内嵌到解释器里面的， 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 解释器中。`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',
 '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']
```

没有参数时，dir() 列出当前已定义的名称：

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

注意它列出所有类型的名称：变量，模块，函数，……。

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 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']
```

6.4. 包

包是通过使用“带点号模块名”来构造 Python 模块命名空间的一种方式。 例如，模块名 `A.B` 表示名为 `A` 的包中名为 `B` 的子模块。 就像使用模块可以让不同模块的作者不必担心彼此的全局变量名一样，使用带点号模块名也可以让 NumPy 或 Pillow 等多模块包的作者也不必担心彼此的模块名冲突。

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/                          最高层级的包
      __init__.py               初始化 sound 包
      formats/                  用于文件格式转换的子包
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  用于音效的子包
              __init__.py
              echo.py
              surround.py
              reverse.py
```

```
              ...
      filters/                用于过滤器的子包
              __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。

需要有 `__init__.py` 文件才能让 Python 将包含该文件的目录当作包来处理（除非使用 namespace package，这是一个相对高级的特性）。 这可以防止重名的目录如 `string` 在无意中屏蔽后继出现在模块搜索路径中的有效模块。 在最简单的情况下，`__init__.py` 可以只是一个空文件，但它也可以执行包的初始化代码或设置 `__all__` 变量，这将在稍后详细描述。

还可以从包中导入单个模块，例如：

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

这将加载子模块 `sound.effects.echo`。 它必须通过其全名来引用。

```
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
```

这也会加载子模块 `echo`，并使其不必加包前缀，因此可按如下方式使用:

```
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，这将加载子模块 `echo`，但这使其函数 `echofilter()` 直接可用:

```
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？你可能希望它会查找并导入包的所有子模块，但事实并非如此。因为这将花费很长的时间，并且可能会产生你不想要的副作用，如果这种副作用被你设计为只有在导入某个特定的子模块时才应该发生。

唯一的解决办法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，它就是被导入的模块名列表。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` 文件可以包含以下代码：

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

这意味着 `from sound.effects import *` 将导入 `sound.effects` 包的三个命名子模块。

请注意子模块可能会受到本地定义名称的影响。 例如，如果你在 `sound/effects/__init__.py` 文件中添加了一个 `reverse` 函数，`from sound.effects import *` 将只导入 `echo` 和 `surround` 这两个子模块，但 **不会** 导入 `reverse` 子模块，因为它被本地定义的 `reverse` 函数所遮挡:

```
__all__ = [
    "echo",      # 指向 'echo.py' 文件
    "surround",  # 指向 'surround.py' 文件
    "reverse",   # !!! 现在指向 'reverse' 函数 !!!
]

def reverse(msg: str):  # <-- 此名称将覆盖 'reverse.py' 子模块
    return msg[::-1]    #     针对 'from sound.effects import *' 的情况
```

如果没有定义 `__all__`，`from sound.effects import *` 语句 *不会* 把包 `sound.effects` 中的所有子模块都导入到当前命名空间；它只是确保包 `sound.effects` 已被导入（可能还会运行 `__init__.py` 中的任何初始化代码），然后再导入包中定义的任何名称。 这包括由 `__init__.py` 定义的任何名称（以及显式加载的子模块）。 它还包括先前 `import` 语句显式加载的包里的任何子模块。 请看以下代码:

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

在本例中，`echo` 和 `surround` 模块被导入到当前命名空间，因为在执行 `from...import` 语句时它们已在 `sound.effects` 包中定义了。 （当定义了 `__all__` 时也是如此）。

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. 相对导入

当包由多个子包构成（如示例中的 `sound` 包）时，可以使用绝对导入来引用同级包的子模块。 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`。

你还可以编写相对导入代码，即使用 `from module import name` 形式的 import 语句。 这些导入使用前导点号来表示相对导入所涉及的当前包和上级包。 例如对于 `surround` 模块，可以使用:

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

需要注意的是，相对导入是基于当前模块所属包的名称进行的。由于主模块（即直接运行的脚本）没有所属包，因此那些打算作为 Python 应用程序主模块使用的模块，必须始终使用绝对导入。

6.4.3. 多目录中的包

包还支持一个特殊的属性， `__path__` 。 在执行该文件中的代码之前，它被初始化为字符串的 sequence，其中包含包的 `__init__.py` 的目录名称。这个变量可以修改；修改后会影响今后对模块和包中包含的子包的搜索。

这个功能虽然不常用，但可用于扩展包中的模块集。

备注

[1] 实际上函数定义也是被执行的语句；模块级函数定义的执行会将函数名称添加到模块的全局命名空间。