Capítulo 6: Módulos

Se você sair do interpretador do Python e entrar novamente, as definições (funções e variáveis) que você havia feito estarão perdidas. Portanto, se você quer escrever um programa um pouco mais longo, você se sairá melhor usando um editor de texto para criar e salvar o programa em um arquivo, usando depois esse arquivo como entrada para a execução do interpretador. Isso é conhecido como gerar um *script*. A medida que seus programas crescem, pode ser desejável dividi-los em vários arquivos para facilitar a manutenção. Você também pode querer reutilizar uma função sem copiar sua definição a cada novo programa.

Para permitir isso, Python tem uma maneira de colocar definições em um arquivo e e então usá-las em um script ou em uma execução interativa no interpretador. Tal arquivo é chamado de "módulo"; definições de um módulo podem ser *importadas* em outros módulos ou no módulo *principal* (a coleção de variáveis a que você tem acesso no nível mais externo de um script executado como um programa, ou no modo calculadora).

Um módulo é um arquivo Python contendo definições e instruções. O nome do arquivo é o módulo com o sufixo `.py` adicionado. Dentro de um módulo, o nome do módulo (como uma string) está disponível na variável global `__name__`. Por exemplo, use seu editor de texto favorito para criar um arquivo chamado `fibo.py` no diretório atual com o seguinte conteúdo:

```
# coding: utf-8
# Módulo números de Fibonacci

def fib(n):    # exibe a série de Fibonacci de 0 até n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # devolve a série de Fibonacci de 0 até n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

Agora, entre no interpretador Python e importe esse módulo com o seguinte comando:

```
>>> import fibo
```

Isso não coloca os nomes das funções definidas em `fibo` diretamente na tabela de símbolos atual; isso coloca somente o nome do módulo `fibo`. Usando o nome do módulo você pode acessar as funções.

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

Se pretende usar uma função frequentemente, pode associá-la a um nome local:

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

Mais sobre módulos

Um módulo pode conter tanto comandos quanto definições de funções e classes. Esses comandos servem para inicializar o módulo. Eles são executados somente na *primeira* vez que o módulo é importado em algum lugar. [1]

Cada módulo tem sua própria tabela de símbolos privada, que é usada como tabela de símbolos global para todas as funções definidas no módulo. Assim, o autor de um módulo pode usar variáveis globais no seu módulo sem se preocupar com conflitos acidentais com as variáveis globais do usuário. Por outro lado, se você precisar usar uma variável global de um módulo, poderá fazê-lo com a mesma notação usada para se referir às suas funções, `nome_do_modulo.nome_do_item`.

Módulos podem importar outros módulos. É costume, porém não obrigatório, colocar todos os comandos `import` no início do módulo (ou script). se preferir). As definições do módulo importado são colocadas na tabela de símbolos global do módulo que faz a importação.

Existe uma variante do comando `import` que importa definições de um módulo diretamente para a tabela de símbolos do módulo importador. Por exemplo:

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

Isso não coloca o nome do módulo de onde foram feitas as importações para a tabela de símbolos local (assim, no exemplo `fibo` não está definido), mas somente o nome das funções `fib` e `fib2`.

Existe ainda uma variante que importa todos os nomes definidos em um módulo:

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

Isso importa todos as declarações de nomes, exceto aqueles que iniciam com um sublinhado (`_`). Na maioria dos casos, programadores Python não usam esta facilidade porque ela introduz um conjunto desconhecido de nomes no ambiente, podendo esconder outros nomes previamente definidos.

Note que, em geral, a prática do `import *` de um módulo ou pacote é desaprovada, uma vez que muitas vezes dificulta a leitura do código. Contudo, é aceitável para diminuir a digitação em sessões interativas.

Note!

1 Na verdade, definições de funções também são 'comandos' que são 'executados'; a execução da definição de uma função coloca o nome da função na tabela de símbolos global do módulo.

Por razões de eficiência, cada módulo é importado somente uma vez por sessão do interpretador. Portanto, se você alterar seus módulos, você deve reiniciar o interpretador -- ou, se é somente um módulo que você quer testar interativamente, use `reload`, ex. `reload(nome_do_modulo)`.

Executando módulos como scripts

Quando você executa um módulo Python assim:

```
python fibo.py <argumentos>
```

o código no módulo será executado, da mesma forma como você estivesse apenas importado, mas com a variável global `__name__` com o valor `"__main__"`. Isso significa que você pode acrescentar este código no fim do seu módulo:

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

para permitir que o arquivo seja usado tanto como um script quanto como um módulo que pode ser importado, porque o código que lê o argumento da linha de comando só será acionado se o módulo foi executado como o arquivo "principal":

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

Se o módulo é importado, o bloco dentro do `if __name__`... não é executado:

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

Isso é frequentemente usado para fornecer interface de usuário conveniente para um módulo, ou para realizar testes (rodando o módulo como um script, uma suíte de testes é executada).

O caminho de busca dos módulos

Quando um módulo chamado `spam` é importado, o interpretador procura um módulo embutido com este nome. Se não existe, procura um arquivo chamado `spam.py` em uma lista de diretórios incluídos na variável `sys.path`, que é inicializada com estes locais:

- o diretório que contém o script importador (ou o diretório atual).
- a variável de ambiente `PYTHONPATH` (uma lista de nomes de diretórios, com a mesma sintaxe da variável de ambiente `PATH`).
- um caminho default que depende da instalação do Python.

Após a inicialização, programas Python podem modificar `sys.path`. O diretório que contém o script sendo executado é colocado no início da lista de caminhos, à frente do caminho da biblioteca padrão. Isto significa que módulos nesse diretório serão carregados no lugar de módulos com o mesmo nome na biblioteca padrão. Isso costuma ser um erro, a menos que seja intencional. Veja a seção *Módulos padrão [neste capítulo]* para mais informações.

Arquivos Python "compilados"

Para acelerar a inicialização de programas curtos que usam muitos módulos da biblioteca padrão, sempre que existe um arquivo chamado `spam.pyc` no mesmo diretório de `spam.py`, o interpretador assume que aquele arquivo contém uma versão "byte-compilada" de `spam`. O horário de modificação da versão de `spam.py` a partir da qual `spam.pyc` foi gerado é é armazenada no arquivo compilado, e o `.pyc` não é utilizado se o horário não confere.

Normalmente, não é preciso fazer nada para gerar o arquivo `spam.pyc`. Sempre que `spam.py` é compilado com sucesso, o interpretador tenta salvar a versão compilada em `spam.pyc`. Não há geração de um erro se essa tentativa falhar; se por alguma razão o arquivo compilado não for inteiramente gravado, o arquivo `spam.pyc` resultante será reconhecido como inválido e, portanto, ignorado. O conteúdo do arquivo `spam.pyc` é independente de plataforma, assim um diretório de módulos Python pode ser compartilhado por máquinas de diferentes arquiteturas.

Algumas dicas para os experts:

- Quando o interpretador Python é invocado com a opção `-O`, é gerado um código otimizado, armazenado em arquivos `.pyo`. O otimizador atual não faz muita coisa; ele apenas remove instruções `assert`. Quando `-O` é utilizada, *todo* **b** é otimizado; arquivos `.pyc` são ignorados e os arquivos `.py` são compilados para bytecode otimizado.
- Passar duas opções `-O` para o interpretador Python (`-OO`) fará com que o compilador realize otimizações mais arriscadas, que em alguns casos raros podem acarretar o mal funcionamento de programas. Atualmente apenas strings `__doc__` são removidas do bytecode, resultando em arquivos `.pyo` mais compactos. Uma vez que alguns programas podem contar com a existência dessas docstrings, use essa opção somente se você souber o que está fazendo.
- Um programa não executa mais rápido quando é lido de um arquivo `.pyc` ou `.pyo` em comparação a quando é lido de um arquivo `.py`. A única diferença é que nos dois primeiros casos o tempo de inicialização do programa é menor.
- Quando um script é executado diretamente a partir o seu nome da linha de comando, não são geradas as formas compiladas deste script em formato `.pyc` ou `.pyo`. Portanto, o tempo de carga de um script pode ser melhorado se transferirmos a maior parte de seu código para um módulo e utilizarmos o script menor apenas para inicialização. Também é possível fornecer um arquivo `.pyc` ou `.pyo` diretamente para execução do interpretador, passando seu nome na linha de comando.
- Na presença das formas compiladas (`spam.pyc` e `spam.pyo`) de um script, não há necessidade do código fonte (`spam.py`). Isto é útil na para se distribuir bibliotecas Python de uma forma que dificulta moderadamente a engenharia reversa.
- O módulo `compileall` pode criar arquivos `.pyc` (ou `.pyo` quando `-O` é usada) para todos os módulos em um dado diretório.

Módulos padrão

Python possui uma biblioteca padrão de módulos, descrita em um documento em separado, a Python Library Reference (doravante “Library Reference”). Alguns módulos estão embutidos no

interpretador; estes possibilitam acesso a operações que não são parte do núcleo da linguagem, mas estão no interpretador seja por eficiência ou para permitir o acesso a chamadas do sistema operacional. O conjunto destes módulos é uma opção de configuração que depende também da plataforma subjacente. Por exemplo, o módulo `winreg` só está disponível em sistemas Windows. Existe um módulo que requer especial atenção: `sys`, que é embutido em qualquer interpretador Python. As variáveis `sys.ps1` e `sys.ps2` definem as strings utilizadas como prompt primário e secundário:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Eca!'
Eca!
C>
```

Essas variáveis só estão definidas se o interpretador está em modo interativo.

A variável `sys.path` contém uma lista de strings que determina os caminhos de busca de módulos conhecidos pelo interpretador. Ela é inicializada para um caminho padrão determinado pela variável de ambiente `PYTHONPATH`, ou por um valor default interno se a variável não estiver definida. Você pode modificar `sys.path` com as operações típicas de lista, por exemplo:

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

A função `dir`

A função embutida `dir` é usada para se descobrir quais nomes são definidos por um módulo. Ela devolve uma lista ordenada de strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding',
'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Sem argumentos, `dir` lista os nomes atualmente definidos:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
```

```
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
'sys']
```

Observe que ela lista todo tipo de nomes: variáveis, módulos, funções, etc.

`dir` não lista nomes de funções ou variáveis embutidas. Se você quiser conhecê-las, estão definidos no módulo padrão `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UserWarning', 'ValueError', 'Warning', 'WindowsError',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview',
 'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range',
 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Pacotes

Pacotes são uma maneira de estruturar espaços de nomes para módulos Python utilizando a sintaxe de "nomes pontuados" (dotted names). Como exemplo, o nome `A.B` designa um submódulo chamado `B` em um pacote denominado `A`. O uso de pacotes permite que os autores de pacotes com muitos módulos, como NumPy ou PIL (Python Imaging Library) não se preocupem com colisão entre os nomes de seus módulos e os nomes de módulos de outros autores.

Suponha que você queira projetar uma coleção de módulos (um "pacote") para o gerenciamento uniforme de arquivos de som. Existem muitos formatos diferentes (normalmente identificados pela extensão do nome de arquivo, por exemplo. `.wav`, `.aiff`, `.au`), de forma que você pode precisar criar e manter uma crescente coleção de módulos de conversão entre formatos. Ainda podem existir muitas operações diferentes passíveis de aplicação sobre os arquivos de som (mixagem, eco, equalização, efeito stereo artificial). Logo, possivelmente você também estará escrevendo uma coleção sempre crescente de módulos para aplicar estas operações. Eis uma

possível estrutura para o seu pacote (expressa em termos de um sistema de arquivos hierárquico):

```
sound/                          Pacote principal
      __init__.py               Inicializar o pacote sound
      formats/                  Subpacote para conversão de formatos
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpacote para efeitos de som
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpacote para filtros
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Ao importar esse pacote, Python busca pelo subdiretório com mesmo nome nos diretórios listados em `sys.path`.

Os arquivos `__init__.py` são necessários para que Python trate os diretórios como pacotes; isso foi feito para evitar que diretórios com nomes comuns, como `string`, inadvertidamente ocultassem módulos válidos que ocorram depois no caminho de busca. No caso mais simples, `__init__.py` pode ser um arquivo vazio. Porém, ele pode conter código de inicialização para o pacote ou definir a variável `__all__`, que será descrita depois.

Usuários do pacote podem importar módulos individuais, por exemplo:

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

Isso carrega o submódulo `sound.effects.echo`. Ele deve ser referenciado com seu nome completo, como em:

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

Uma maneira alternativa para a importação desse módulo é:

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

Isso carrega o submódulo `echo` sem necessidade de mencionar o prefixo do pacote no momento da utilização, assim:

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

Também é possível importar diretamente uma única variável ou função:

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

Novamente, isso carrega o submódulo `echo`, mas a função `echofilter` está acessível diretamente sem prefixo:

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

Observe que ao utilizar `from package import item`, o item pode ser um subpacote, submódulo, classe, função ou variável. O comando `import` primeiro testa se o item está definido no pacote, senão assume que é um módulo e tenta carregá-lo. Se falhar em encontrar o módulo uma exceção `ImportError` é lançada.

Em oposição, em uma construção como `import item.subitem.subsubitem`, cada item, com exceção do último, deve ser um pacote. O último pode ser também um pacote ou módulo, mas nunca uma classe, função ou variável contida em um módulo.

*Importando * de um pacote*

Agora, o que acontece quando um usuário escreve `from sound.effects import *` ? Idealmente, poderia se esperar que este comando vasculhasse o sistema de arquivos, encontrasse todos submódulos presentes no pacote, e os importassem. Isso pode demorar muito e a importação de submódulos pode ocasionar efeitos colaterais que somente deveriam ocorrer quando o submódulo é explicitamente importado.

A única solução é o autor do pacote fornecer um índice explícito do pacote. O comando `import` usa a seguinte convenção: se o arquivo `__init__.py` do pacote define uma lista chamada `__all__`, então esta lista indica os nomes dos módulos a serem importados quando o comando `from pacote import *` é acionado. Fica a cargo do autor do pacote manter esta lista atualizada, inclusive fica a seu critério excluir inteiramente o suporte a importação direta de todo o pacote através de `from pacote import *`. Por exemplo, o arquivo `sounds/effects/__init__.py` poderia conter apenas:

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

Isso significaria que `from sound.effects import *` importaria apenas os três submódulos especificados no pacote `sound`.

Se `__all__` não estiver definido, o comando `from sound.effects import *` não importa todos os submódulos do pacote `sound.effects` no espaço de nomes atual. Há apenas garantia que o pacote `sound.effects` foi importado (possivelmente executando qualquer código de inicialização em `__init__.py`) juntamente com os nomes definidos no pacote. Isso inclui todo nome definido em `__init__.py` bem como em qualquer submódulo importado a partir deste. Também inclui quaisquer submódulos do pacote que tenham sido carregados explicitamente por comandos `import` anteriores. Considere o código abaixo:

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

Nesse exemplo, os nomes `echo` e `surround` são importados no espaço de nomes atual no momento em que o comando `from ... import` é executado, pois estão definidos no pacote `sound.effects`. (Isso também funciona quando `__all__` estiver definida.)

Apesar de que certos módulos são projetados para exportar apenas nomes conforme algum critério quando se faz `import *`, ainda assim essa sintaxe é considerada uma prática ruim em código de produção.

Lembre-se que não há nada de errado em utilizar `from pacote import submodulo_especifico`! De fato, essa é a notação recomendada a menos que o módulo efetuando a importação precise utilizar submódulos homônimos de diferentes pacotes.

Referências em um mesmo pacote

Os submódulos frequentemente precisam referenciar uns aos outros. Por exemplo, o módulo `surround` talvez precise utilizar o módulo `echo`. De fato, tais referências são tão comuns que o comando `import` primeiro busca módulos dentro do pacote antes de utilizar o caminho de busca padrão. Portanto, o módulo `surround` pode usar simplesmente `import echo` ou `from echo import echofilter`. Se o módulo importado não for encontrado no pacote atual (o pacote do qual o módulo atual é submódulo), então o comando `import` procura por um módulo de mesmo nome fora do pacote (nos locais definidos em `sys.path`).

Quando pacotes são estruturados em subpacotes (como no pacote `sound` do exemplo), pode ser usar a sintaxe de um `import` absoluto para se referir aos submódulos de pacotes irmãos (o que na prática é uma forma de fazer um import relativo, a partir da base do pacote). Por exemplo, se o módulo `sound.filters.vocoder` precisa usar o módulo `echo` do pacote `sound.effects`, é preciso importá-lo com `from sound.effects import echo`.

A partir do Python 2.5, em adição à importação relativa implícita descrita acima, você pode usar importação relativa explícita na forma `from import`. Essas importações relativas explícitas usam prefixos com pontos indicar os pacotes atuais e seus pais envolvidos na importação. A partir do módulo `surround` por exemplo, pode-se usar:

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

Note que tanto a importação relativa explícita quanto a implícita baseiam-se no nome do módulo atual. Uma vez que o nome do módulo principal é sempre `"__main__"`, módulos que serão o módulo principal de uma aplicação Python devem sempre usar importações absolutas.

Pacotes em múltiplos diretórios

Pacotes possuem mais um atributo especial, `__path__`. Ele é inicializado como uma lista contendo o nome do diretório onde está o arquivo `__init__.py` do pacote, antes do código naquele arquivo ser executado. Esta variável pode ser modificada; isso afeta a busca futura de módulos e subpacotes contidos no pacote.

Apesar de não ser muito usado, esse mecanismo permite estender o conjunto de módulos encontrados em um pacote.

Notas