模块的使用

python解释器可以使用模块来定义重复使用的代码，这样可以让一些代码不需要通过复制或者重写的方式在不同的项目之间复用。python代码存储文件使用`.py`作为文件名后缀。

在python解释器（运行环境中），通过使用`import`来引入需要的模块。

引入一个模块时，每个模块所拥有的私有符号表。一个模块引入另一个模块时，导入的模块名称放在导入模块的全局符号表中。也可以使用`from fibo import fib, fib2`将名称直接导入模块的符号表中。这样不会将模块的名字引入到本地符号表中。例如上面的fibo名称将不存在。

通过`from fibo import *`可以导入模块定义的所有名称（但是不包括那些用"_"开始的名称）。但是我们不鼓励使用*的方式去引入模块，这样做会造成代码的可读性较差。不过在进行交互式尝试的时候，这样的方式可以帮助我们节省打字时间。

如果在引入模块名字后使用`as`，那么在之后的名字被绑定到导入的模块。在使用from语句引入时也可以用`as`来进行名称绑定。

执行和导入模块

在导入一个模块时，和使用下述语句执行一个模块是类似的

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

但是在执行过程中`__name__`被赋值为`__main__`。因此，我们在模块代码末尾通常会添加下面的形式，用来让这个文件在作为模块导入的时候不

被当作执行的脚本。

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

有了上面的判断后，对于fibo.py作为脚本执行时，如下效果

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

而导入的时候，代码不会被执行

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

模块的搜索路径

当一个module（模块）需要被引入时，python按照下面的顺序查找这个包的位置

1. 当前输入脚本的目录（或者在没有指定文件时，搜索当前目录）；
2. `PYTHONPATH`（类似环境变量PATH格式）给出的目录列表；
3. 依赖于安装的默认值

当初始化完成后，Python程序会把当前目录放在搜索目录的最开始，在标准库的路径之前，这意味着在这个目录中，如果有和库目录同名的脚本会优先引入。这容易造成一些错误。

“编译”Python文件

这个部分以后考虑补充

标准库

Python安装会覆盖有一个标准库。其中一些模块内置于解释器中；这些操作提供不属于语言核心但仍然内置的操作访问，这样可以提高效率或者提供对于操作系统的调用。

`sys.ps1`和`sys.ps2`提供了python解释器交互模式下的提示符。

`sys.path`初始化提供了环境中默认的PYTHONPATH，我们可以直接通过修改这个参数的值来进行变更。

`dir()`

内置的`dir()`用来显示模块已经定义了哪些名称。调用返回一个排序的字符串列表。如果不提供参数，列表返回当前已经定义的全部名称。

```
>>> import fibo, sys
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__',
>>>
```

dir()不会列出所有内建的函数和变量，如果需要显示这些内建的名称列表，可以查看标准库builtins

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException'
```

Package

Package包就是用来给Python提供一个结构化保存module模块名称空间的方法。类似与前面学习的Module中使用模块点名称的表示方式。比如：名称A.B表示包A下面有一个子模块B。使用这种命名空间的方式，可以让不同的模块作者可以不用担心自己的命名和其他模块有冲突。

如果我们想设计一个包由很多的模块组成。而且考虑到这个模块以后根据需要还会扩展支持新的内容，那么会考虑设计下面这样一个结构。

```
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
              ...
```

根目录下的`__init__.py`文件用来让Python知道这是一个包含有Package的目录，这个文件最简单的形式就是以空文件的方式存在即可。这里需要注意，我们不能给这个根目录命名为一个已经存在的公共名称，例如：`string`。

按照上面的目录结构，我们可以用下面的方式引入单个的独立模块：

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

使用的时候需要我们用完整的名称引用：

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

一个可以替代的方式如下：

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

这种方式下，子模块被作为echo引入，因此使用的时候就不需要再给出完整的名称引用，使用方式如下：

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

还有一个方式是直接引入需要的方法，那么可以直接使用

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

在使用`from package import item`时，item可以是子模块，也可以是在包中定义的其他名称，例如：一个方法，类或者是变量。在引入时，首先会检查这个名称是否存在，如果存在就尝试加载导入，如果失败，那么会报出一个`ImportError`的错误。

在使用`import item.subitem.subsubitem`这样的语法时，除了最后一个item，其他都必须是包Package，而最后一个item可以是模块或者包，但是不能是在前一个item包中定义的类或者方法。

import * from package

如果执行`from sound.effects import *`会发生什么？如果依赖于通过文件系统去搜索和检查包中所有的内容并进行导入，那么这个过程不仅会花费比较长的时间，可能会有我们不希望引入的一些内容或者情况发生。

因此，如果一个包的作者希望避免问题，那么可以在包的`__init__.py`文件中定义一个列表，名字为`__all__`，它用来保存所有通过`import *`需要导入的模块名称。

当`__all__`没有定义时，通过上述导入语句将不会去尝试从当前包中导入可能的子包内容，仅会将当前包中的模块导入。

包内引用

当在一个包内的子包需要相互引用时，仍然可以通过直接的绝对方式进行引用。如`sound.filters.vocoder`需要去引用在`sound.effects`包中的`echo`模块，那么可以直接通过`from sound.effects import echo`引入。

除了绝对引入，还可以考虑采用相对的引用方式，只需使用前导的点去表明当前或父包需要导入

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

相对引用因为是给予当前模块的位置，而主模块就只能使用绝对引用。