15.10 用Cython包装C代码¶

问题¶

你想使用Cython来创建一个Python扩展模块，用来包装某个已存在的C函数库。

解决方案¶

使用Cython构建一个扩展模块看上去很手写扩展有些类似， 因为你需要创建很多包装函数。不过，跟前面不同的是，你不需要在C语言中做这些——代码看上去更像是Python。

作为准备，假设本章介绍部分的示例代码已经被编译到某个叫 `libsample` 的C函数库中了。 首先创建一个名叫 `csample.pxd` 的文件，如下所示：

```
# csample.pxd
#
# Declarations of "external" C functions and structures

cdef extern from "sample.h":
    int gcd(int, int)
    bint in_mandel(double, double, int)
    int divide(int, int, int *)
    double avg(double *, int) nogil

    ctypedef struct Point:
         double x
         double y

    double distance(Point *, Point *)
```

这个文件在Cython中的作用就跟C的头文件一样。 初始声明 `cdef extern from "sample.h"` 指定了所学的C头文件。 接下来的声明都是来自于那个头文件。文件名是 `csample.pxd` ，而不是 `sample.pxd` ——这点很重要。

下一步，创建一个名为 `sample.pyx` 的问题。 该文件会定义包装器，用来桥接Python解释器到 `csample.pxd` 中声明的C代码。

```
# sample.pyx

# Import the low-level C declarations
cimport csample

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *

from libc.stdlib cimport malloc, free

# Wrappers
def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x, y)

def in_mandel(x, y, unsigned int n):
    return csample.in_mandel(x, y, n)

def divide(x, y):
    cdef int rem
    quot = csample.divide(x, y, &rem)
    return quot, rem

def avg(double[:] a):
    cdef:
        int sz
        double result

    sz = a.size
    with nogil:
```

```
        result = csample.avg(<double *> &a[0], sz)
    return result

# Destructor for cleaning up Point objects
cdef del_Point(object obj):
    pt = <csample.Point *> PyCapsule_GetPointer(obj,"Point")
    free(<void *> pt)

# Create a Point object and return as a capsule
def Point(double x,double y):
    cdef csample.Point *p
    p = <csample.Point *> malloc(sizeof(csample.Point))
    if p == NULL:
        raise MemoryError("No memory to make a Point")
    p.x = x
    p.y = y
    return PyCapsule_New(<void *>p,"Point",<PyCapsule_Destructor>del_Point)

def distance(p1, p2):
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1,"Point")
    pt2 = <csample.Point *> PyCapsule_GetPointer(p2,"Point")
    return csample.distance(pt1,pt2)
```

该文件更多的细节部分会在讨论部分详细展开。 最后，为了构建扩展模块，像下面这样创建一个 `setup.py` 文件：

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',

              ['sample.pyx'],
              libraries=['sample'],
              library_dirs=['.'])]
setup(
  name = 'Sample extension module',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

要构建我们测试的目标模块，像下面这样做：

```
bash % python3 setup.py build_ext --inplace
running build_ext
cythoning sample.pyx to sample.c
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
 -I/usr/local/include/python3.3m -c sample.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample.o
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o
  -L. -lsample -o sample.so
bash %
```

如果一切顺利的话，你应该有了一个扩展模块 `sample.so` ，可在下面例子中使用：

```
>>> import sample
>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
```

```
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

讨论¶

本节包含了很多前面所讲的高级特性，包括数组操作、包装隐形指针和释放GIL。 每一部分都会逐个被讲述到，但是我们最好能复习一下前面几小节。 在顶层，使用Cython是基于C之上。.pxd文件仅仅只包含C定义（类似.h文件），.pyx文件包含了实现（类似.c文件）。 `cimport` 语句被Cython用来导入.pxd文件中的定义。 它跟使用普通的加载Python模块的导入语句是不同的。

尽管 *.pxd* 文件包含了定义，但它们并不是用来自动创建扩展代码的。 因此，你还是要写包装函数。例如，就算 `csample.pxd` 文件声明了 `int gcd(int, int)` 函数， 你仍然需要在 `sample.pyx` 中为它写一个包装函数。例如：

```
cimport csample

def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x,y)
```

对于简单的函数，你并不需要去做太多的时。 Cython会生成包装代码来正确的转换参数和返回值。 绑定到属性上的C数据类型是可选的。不过，如果你包含了它们，你可以另外做一些错误检查。 例如，如果有人使用负数来调用这个函数，会抛出一个异常：

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

如果你想对包装函数做另外的检查，只需要使用另外的包装代码。例如：

```
def gcd(unsigned int x, unsigned int y):
    if x <= 0:
        raise ValueError("x must be > 0")
    if y <= 0:
        raise ValueError("y must be > 0")
    return csample.gcd(x,y)
```

在csample.pxd文件中的``in_mandel()`` 声明有个很有趣但是比较难理解的定义。 在这个文件中，函数被声明为然后一个bint而不是一个int。 它会让函数创建一个正确的Boolean值而不是简单的整数。 因此，返回值0表示False而1表示True。

在Cython包装器中，你可以选择声明C数据类型，也可以使用所有的常见Python对象。 对于 `divide()` 的包装器展示了这样一个例子，同时还有如何去处理一个指针参数。

```
def divide(x,y):
    cdef int rem
    quot = csample.divide(x,y,&rem)
    return quot, rem
```

在这里，`rem` 变量被显示的声明为一个C整型变量。 当它被传入 `divide()` 函数的时候，`&rem` 创建一个跟C一样的指向它的指针。 `avg()` 函数的代码演示了Cython更高级的特性。 首先 `def avg(double[:] a)` 声明了 `avg()` 接受一个一维的双精度内存视图。 最惊奇的部分是返回的结果函数可以接受任何兼容的数组对象，包括被numpy创建的。例如：

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> import numpy
>>> b = numpy.array([1., 2., 3.])
>>> import sample
```

```
>>> sample.avg(a)
2.0
>>> sample.avg(b)
2.0
>>>
```

在此包装器中，`a.size0` 和 `&a[0]` 分别引用数组元素个数和底层指针。 语法 `<double *> &a[0]` 教你怎样将指针转换为不同的类型。 前提是C中的 `avg()` 接受一个正确类型的指针。 参考下一节关于Cython内存视图的更高级讲述。

除了处理通常的数组外，`avg()` 的这个例子还展示了如何处理全局解释器锁。 语句 `with nogil:` 声明了一个不需要GIL就能执行的代码块。 在这个块中，不能有任何的普通Python对象——只能使用被声明为 `cdef` 的对象和函数。 另外，外部函数必须现实的声明它们能不依赖GIL就能执行。 因此，在csample.pxd文件中，`avg()` 被声明为 `double avg(double *, int) nogil.`

对Point结构体的处理是一个挑战。本节使用胶囊对象将Point对象当做隐形指针来处理，这个在15.4小节介绍过。 要这样做的话，底层Cython代码稍微有点复杂。 首先，下面的导入被用来引入C函数库和Python C API中定义的函数：

```
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free
```

函数 `del_Point()` 和 `Point()` 使用这个功能来创建一个胶囊对象， 它会包装一个 `Point *` 指针。`cdef del_Point()` 将 `del_Point()` 声明为一个函数， 只能通过Cython访问，而不能从Python中访问。 因此，这个函数对外部是不可见的——它被用来当做一个回调函数来清理胶囊分配的内存。 函数调用比如 `PyCapsule_New()`、`PyCapsule_GetPointer()` 直接来自Python C API并且以同样的方式被使用。

`distance` 函数从 `Point()` 创建的胶囊对象中提取指针。 这里要注意的是你不需要担心异常处理。 如果一个错误的对象被传进来，`PyCapsule_GetPointer()` 会抛出一个异常， 但是Cython已经知道怎么查找到它，并将它从 `distance()` 传递出去。

处理Point结构体一个缺点是它的实现是不可见的。 你不能访问任何属性来查看它的内部。 这里有另外一种方法去包装它，就是定义一个扩展类型，如下所示：

```
# sample.pyx

cimport csample
from libc.stdlib cimport malloc, free
...

cdef class Point:
    cdef csample.Point *_c_point
    def __cinit__(self, double x, double y):
        self._c_point = <csample.Point *> malloc(sizeof(csample.Point))
        self._c_point.x = x
        self._c_point.y = y

    def __dealloc__(self):
        free(self._c_point)

    property x:
        def __get__(self):
            return self._c_point.x
        def __set__(self, value):
            self._c_point.x = value

    property y:
        def __get__(self):
            return self._c_point.y
        def __set__(self, value):
            self._c_point.y = value

def distance(Point p1, Point p2):
    return csample.distance(p1._c_point, p2._c_point)
```

在这里，cdif类 `Point` 将Point声明为一个扩展类型。 类属性 `cdef csample.Point *_c_point` 声明了一个实例变量， 拥有一个指向底层Point结构体的指针。 `__cinit__()` 和 `__dealloc__()` 方法通过 `malloc()` 和 `free()` 创建并销毁底层C结构体。 x和y属性的声明让你获取和设置底层结构体的属性值。 `distance()` 的包装器还可以被修改，使得它能接受

`Point` 扩展类型实例作为参数， 而传递底层指针给C函数。

做了这个改变后，你会发现操作Point对象就显得更加自然了：

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

本节已经演示了很多Cython的核心特性，你可以以此为基准来构建更多更高级的包装。 不过，你最好先去阅读下官方文档来了解更多信息。

接下来几节还会继续演示一些Cython的其他特性。