15.1 使用ctypes访问C代码¶

问题¶

你有一些C函数已经被编译到共享库或DLL中。你希望可以使用纯Python代码调用这些函数， 而不用编写额外的C代码或使用第三方扩展工具。

解决方案¶

对于需要调用C代码的一些小的问题，通常使用Python标准库中的 `ctypes` 模块就足够了。 要使用 `ctypes` ，你首先要确保你要访问的C代码已经被编译到和Python解释器兼容 （同样的架构、字大小、编译器等）的某个共享库中了。 为了进行本节的演示，假设你有一个共享库名字叫 `libsample.so` ，里面的内容就是15章介绍部分那样。 另外还假设这个 `libsample.so` 文件被放置到位于 `sample.py` 文件相同的目录中了。

要访问这个函数库，你要先构建一个包装它的Python模块，如下这样：

```
# sample.py
import ctypes
import os

# Try to locate the .so file in the same directory as this file
_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)

    return quot,rem.value

# void avg(double *, int n)
# Define a special type for the 'double *' argument
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

    # Cast from array.array objects
    def from_array(self, param):
        if param.typecode != 'd':
            raise TypeError('must be an array of doubles')
        ptr, _ = param.buffer_info()
        return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
```

```
    # Cast from lists/tuples
    def from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return val

    from_tuple = from_list

    # Cast from a numpy array
    def from_ndarray(self, param):
        return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double
```

如果一切正常，你就可以加载并使用里面定义的C函数了。例如：

```
>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

讨论¶

本小节有很多值得我们详细讨论的地方。 首先是对于C和Python代码一起打包的问题，如果你在使用 `ctypes` 来访问编译后的C代码， 那么需要确保这个共享库放在 `sample.py` 模块同一个地方。 一种可能是将生成的 `.so` 文件放置在要使用它的Python代码同一个目录下。 我们在 `recipe-sample.py` 中使用 `__file__` 变量来查看它被安装的位置， 然后构造一个指向同一个目录中的 `libsample.so` 文件的路径。

如果C函数库被安装到其他地方，那么你就要修改相应的路径。 如果C函数库在你机器上被安装为一个标准库了， 那么可以使用 `ctypes.util.find_library()` 函数来查找：

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

一旦你知道了C函数库的位置，那么就可以像下面这样使用 `ctypes.cdll.LoadLibrary()` 来加载它， 其中 `_path` 是标

准库的全路径：

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

函数库被加载后，你需要编写几个语句来提取特定的符号并指定它们的类型。 就像下面这个代码片段一样：

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

在这段代码中，`.argtypes` 属性是一个元组，包含了某个函数的输入按时， 而 `.restype` 就是相应的返回类型。 `ctypes` 定义了大量的类型对象（比如c_double, c_int, c_short, c_float等）， 代表了对应的C数据类型。如果你想让Python能够传递正确的参数类型并且正确的转换数据的话， 那么这些类型签名的绑定是很重要的一步。如果你没有这么做，不但代码不能正常运行， 还可能会导致整个解释器进程挂掉。 使用ctypes有一个麻烦点的地方是原生的C代码使用的术语可能跟Python不能明确的对应上来。 `divide()` 函数是一个很好的例子，它通过一个参数除以另一个参数返回一个结果值。 尽管这是一个很常见的C技术，但是在Python中却不知道怎样清晰的表达出来。 例如，你不能像下面这样简单的做：

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```

就算这个能正确的工作，它会违反Python对于整数的不可更改原则，并且可能会导致整个解释器陷入一个黑洞中。 对于涉及到指针的参数，你通常需要先构建一个相应的ctypes对象并像下面这样传进去：

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

在这里，一个 `ctypes.c_int` 实例被创建并作为一个指针被传进去。 跟普通Python整形不同的是，一个 `c_int` 对象是可以被修改的。 `.value` 属性可被用来获取或更改这个值。

对于那些不像Python的C调用，通常可以写一个小的包装函数。 这里，我们让 `divide()` 函数通过元组来返回两个结果：

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

`avg()` 函数又是一个新的挑战。C代码期望接受到一个指针和一个数组的长度值。 但是，在Python中，我们必须考虑这个问题：数组是啥？它是一个列表？一个元组？ 还是 `array` 模块中的一个数组？还是一个 `numpy` 数组？还是说所有都是？ 实际上，一个Python“数组”有多种形式，你可能想要支持多种可能性。

`DoubleArrayType` 演示了怎样处理这种情况。 在这个类中定义了一个单个方法 `from_param()` 。 这个方法的角色是接受一个单个参数然后将其向下转换为一个合适的ctypes对象 （本例中是一个 `ctypes.c_double` 的指针）。 在 `from_param()` 中，你可以做任何你想做的事。 参数的类型名被提取出来并被用于分发到一个更具体的方法中去。 例如，如果一个列表被传递过来，那么 `typename` 就是 `list` ， 然后 `from_list` 方法被调用。

对于列表和元组，`from_list` 方法将其转换为一个 `ctypes` 的数组对象。 这个看上去有点奇怪，下面我们使用一个交互

式例子来将一个列表转换为一个 ctypes 数组：

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

对于数组对象，from_array() 提取底层的内存指针并将其转换为一个 ctypes 指针对象。例如：

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>
```

from_ndarray() 演示了对于 numpy 数组的转换操作。 通过定义 DoubleArrayType 类并在 avg() 类型签名中使用它，那么这个函数就能接受多个不同的类数组输入了：

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>
```

本节最后一部分向你演示了怎样处理一个简单的C结构。 对于结构体，你只需要像下面这样简单的定义一个类，包含相应的字段和类型即可：

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

一旦类被定义后，你就可以在类型签名中或者是需要实例化结构体的代码中使用它。例如：

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

最后一些小的提示：如果你想在Python中访问一些小的C函数，那么 ctypes 是一个很有用的函数库。 尽管如此，如果你想要去访问一个很大的库，那么可能就需要其他的方法了，比如 Swig (15.9节会讲到) 或 Cython（15.10节）。

对于大型库的访问有个主要问题，由于ctypes并不是完全自动化， 那么你就必须花费大量时间来编写所有的类型签名，就像例子中那样。 如果函数库够复杂，你还得去编写很多小的包装函数和支持类。 另外，除非你已经完全精通了所有底层的C接口细节，包括内存分配和错误处理机制， 通常一个很小的代码缺陷、访问越界或其他类似错误就能让

Python程序奔溃。

作为 `ctypes` 的一个替代，你还可以考虑下CFFI。CFFI提供了很多类似的功能， 但是使用C语法并支持更多高级的C代码类型。 到写这本书为止，CFFI还是一个相对较新的工程， 但是它的流行度正在快速上升。 甚至还有在讨论在Python将来的版本中将它包含进去。因此，这个真的值得一看。

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。

15.11 用Cython写高性能的数组操作¶

问题¶

你要写高性能的操作来自NumPy之类的数组计算函数。 你已经知道了Cython这样的工具会让它变得简单，但是并不确定该怎样去做。

解决方案¶

作为一个例子，下面的代码演示了一个Cython函数，用来修整一个简单的一维双精度浮点数数组中元素的值。

```
# sample.pyx (Cython)

cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    '''
    Clip the values in a to be between min and max. Result in out
    '''
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        if a[i] < min:
            out[i] = min
        elif a[i] > max:
            out[i] = max
        else:
            out[i] = a[i]
```

要编译和构建这个扩展，你需要一个像下面这样的 `setup.py` 文件 （使用 `python3 setup.py build_ext --inplace` 来构建它）：

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

ext_modules = [
    Extension('sample',
              ['sample.pyx'])
]

setup(
  name = 'Sample app',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

你会发现结果函数确实对数组进行的修正，并且可以适用于多种类型的数组对象。例如：

```
>>> # array module example
>>> import sample
>>> import array
>>> a = array.array('d',[1,-3,4,7,2,0])
>>> a

array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0])
>>> sample.clip(a,1,4,a)
>>> a
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])

>>> # numpy example
>>> import numpy
```

```
>>> b = numpy.random.uniform(-10,10,size=1000000)
>>> b
array([-9.55546017,  7.45599334,  0.69248932, ...,  0.69583148,
       -3.86290931,  2.37266888])
>>> c = numpy.zeros_like(b)
>>> c
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
>>> sample.clip(b,-5,5,c)
>>> c
array([-5.        ,  5.        ,  0.69248932, ...,  0.69583148,
       -3.86290931,  2.37266888])
>>> min(c)
-5.0
>>> max(c)
5.0
>>>
```

你还会发现运行生成结果非常的快。 下面我们将本例和numpy中的已存在的 `clip()` 函数做一个性能对比：

```
>>> timeit('numpy.clip(b,-5,5,c)','from __main__ import b,c,numpy',number=1000)
8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)','from __main__ import b,c,sample',
...         number=1000)
3.760528204000366
>>>
```

正如你看到的，它要快很多——这是一个很有趣的结果，因为NumPy版本的核心代码还是用C语言写的。

讨论¶

本节利用了Cython类型的内存视图，极大的简化了数组的操作。 `cpdef clip()` 声明了 `clip()` 同时为C级别函数以及Python级别函数。 在Cython中，这个是很重要的，因为它表示此函数调用要比其他Cython函数更加高效 （比如你想在另外一个不同的Cython函数中调用clip()）。

类型参数 `double[:] a` 和 `double[:] out` 声明这些参数为一维的双精度数组。 作为输入，它们会访问任何实现了内存视图接口的数组对象，这个在PEP 3118有详细定义。 包括了NumPy中的数组和内置的array库。

当你编写生成结果为数组的代码时，你应该遵循上面示例那样设置一个输出参数。 它会将创建输出数组的责任给调用者，不需要知道你操作的数组的具体细节 （它仅仅假设数组已经准备好了，只需要做一些小的检查比如确保数组大小是正确的）。 在像NumPy之类的库中，使用 `numpy.zeros()` 或 `numpy.zeros_like()` 创建输出数组相对而言比较容易。另外，要创建未初始化数组， 你可以使用 `numpy.empty()` 或 `numpy.empty_like()` . 如果你想覆盖数组内容作为结果的话选择这两个会比较快点。

在你的函数实现中，你只需要简单的通过下标运算和数组查找（比如a[i],out[i]等）来编写代码操作数组。 Cython会负责为你生成高效的代码。

`clip()` 定义之前的两个装饰器可以优化下性能。 `@cython.boundscheck(False)` 省去了所有的数组越界检查， 当你知道下标访问不会越界的时候可以使用它。 `@cython.wraparound(False)` 消除了相对数组尾部的负数下标的处理（类似Python列表）。 引入这两个装饰器可以极大的提升性能（测试这个例子的时候大概快了2.5倍）。

任何时候处理数组时，研究并改善底层算法同样可以极大的提示性能。 例如，考虑对 `clip()` 函数的如下修正，使用条件表达式：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

实际测试结果是，这个版本的代码运行速度要快50%以上（2.44秒对比之前使用 `timeit()` 测试的3.76秒）。

到这里为止，你可能想知道这种代码怎么能跟手写C语言PK呢？ 例如，你可能写了如下的C函数并使用前面几节的技术来手写扩展：

```
void clip(double *a, int n, double min, double max, double *out) {
  double x;
  for (; n >= 0; n--, a++, out++) {
    x = *a;

    *out = x > max ? max : (x < min ? min : x);
  }
}
```

我们没有展示这个的扩展代码，但是试验之后，我们发现一个手写C扩展要比使用Cython版本的慢了大概10%。 最底下的一行比你想象的运行的快很多。

你可以对实例代码构建多个扩展。 对于某些数组操作，最好要释放GIL，这样多个线程能并行运行。 要这样做的话，需要修改代码，使用 `with nogil:` 语句：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    with nogil:
        for i in range(a.shape[0]):
            out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

如果你想写一个操作二维数组的版本，下面是可以参考下：

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip2d(double[:,:] a, double min, double max, double[:,:] out):
    if min > max:
        raise ValueError("min must be <= max")
    for n in range(a.ndim):
        if a.shape[n] != out.shape[n]:
            raise TypeError("a and out have different shapes")
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if a[i,j] < min:
                out[i,j] = min
            elif a[i,j] > max:
                out[i,j] = max
            else:
                out[i,j] = a[i,j]
```

希望读者不要忘了本节所有代码都不会绑定到某个特定数组库（比如NumPy）上面。 这样代码就更有灵活性。 不过，要注意的是如果处理数组要涉及到多维数组、切片、偏移和其他因素的时候情况会变得复杂起来。 这些内容已经超出本节范围，更多信息请参考 PEP 3118 ， 同时 Cython文档中关于“类型内存视图” 篇也值得一读。

15.12 将函数指针转换为可调用对象¶

问题¶

你已经获得了一个被编译函数的内存地址，想将它转换成一个Python可调用对象， 这样的话你就可以将它作为一个扩展函数使用了。

解决方案¶

`ctypes` 模块可被用来创建包装任意内存地址的Python可调用对象。 下面的例子演示了怎样获取C函数的原始、底层地址，以及如何将其转换为一个可调用对象：

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Get the address of sin() from the C math library
>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760

>>> # Turn the address into a callable function
>>> functype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
>>> func = functype(addr)
>>> func
<CFunctionType object at 0x1006816d0>

>>> # Call the resulting function
>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

讨论¶

要构建一个可调用对象，你首先需要创建一个 `CFUNCTYPE` 实例。 `CFUNCTYPE()` 的第一个参数是返回类型。 接下来的参数是参数类型。一旦你定义了函数类型，你就能将它包装在一个整型内存地址上来创建一个可调用对象了。 生成的对象被当做普通的可通过 `ctypes` 访问的函数来使用。

本节看上去可能有点神秘，偏底层一点。 但是，但是它被广泛使用于各种高级代码生成技术比如即时编译，在LLVM函数库中可以看到。

例如，下面是一个使用 `llvmpy` 扩展的简单例子，用来构建一个小的聚集函数，获取它的函数指针， 并将其转换为一个Python可调用对象。

```
>>> from llvm.core import Module, Function, Type, Builder
>>> mod = Module.new('example')
>>> f = Function.new(mod,Type.function(Type.double(), \
                     [Type.double(), Type.double()], False), 'foo')
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0],f.args[0])
>>> y2 = builder.fmul(f.args[1],f.args[1])
>>> r = builder.fadd(x2,y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import ExecutionEngine
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, ctypes.c_double)(ptr)

>>> # Call the resulting function
```

```
>>> foo(2,3)
13.0
>>> foo(4,5)
41.0
>>> foo(1,2)
5.0
>>>
```

并不是说在这个层面犯了任何错误就会导致Python解释器挂掉。 要记得的是你是在直接跟机器级别的内存地址和本地机器码打交道，而不是Python函数。

15.13 传递NULL结尾的字符串给C函数库¶

问题¶

你要写一个扩展模块，需要传递一个NULL结尾的字符串给C函数库。 不过，你不是很确定怎样使用Python的Unicode字符串去实现它。

解决方案¶

许多C函数库包含一些操作NULL结尾的字符串，被声明类型为 `char *`. 考虑如下的C函数，我们用来做演示和测试用的：

```
void print_chars(char *s) {
    while (*s) {
        printf("%2x ", (unsigned char) *s);

        s++;
    }
    printf("\n");
}
```

此函数会打印被传进来字符串的每个字符的十六进制表示，这样的话可以很容易的进行调试了。例如：

```
print_chars("Hello");   // Outputs: 48 65 6c 6c 6f
```

对于在Python中调用这样的C函数，你有几种选择。 首先，你可以通过调用 `PyArg_ParseTuple()` 并指定”y“转换码来限制它只能操作字节，如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;

  if (!PyArg_ParseTuple(args, "y", &s)) {
    return NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

结果函数的使用方法如下。仔细观察嵌入了NULL字节的字符串以及Unicode支持是怎样被拒绝的：

```
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>>
```

如果你想传递Unicode字符串，在 `PyArg_ParseTuple()` 中使用”s“格式码，如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;

  if (!PyArg_ParseTuple(args, "s", &s)) {
    return NULL;
  }
  print_chars(s);
  Py_RETURN_NONE;
}
```

当被使用的时候，它会自动将所有字符串转换为以NULL结尾的UTF-8编码。例如：

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1o')  # Note: UTF-8 encoding
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

如果因为某些原因，你要直接使用 `PyObject *` 而不能使用 `PyArg_ParseTuple()`， 下面的例子向你展示了怎样从字节和字符串对象中检查和提取一个合适的 `char *` 引用：

```
/* Some Python Object (obtained somehow) */
PyObject *obj;

/* Conversion from bytes */
{
   char *s;
   s = PyBytes_AsString(o);
   if (!s) {
      return NULL;   /* TypeError already raised */
   }
   print_chars(s);
}

/* Conversion to UTF-8 bytes from a string */
{
   PyObject *bytes;
   char *s;
   if (!PyUnicode_Check(obj)) {
       PyErr_SetString(PyExc_TypeError, "Expected string");
       return NULL;
   }
   bytes = PyUnicode_AsUTF8String(obj);
   s = PyBytes_AsString(bytes);
   print_chars(s);
   Py_DECREF(bytes);
}
```

前面两种转换都可以确保是NULL结尾的数据， 但是它们并不检查字符串中间是否嵌入了NULL字节。 因此，如果这个很重要的话，那你需要自己去做检查了。

讨论¶

如果可能的话，你应该避免去写一些依赖于NULL结尾的字符串，因为Python并没有这个需要。 最好结合使用一个指针和长度值来处理字符串。 不过，有时候你必须去处理C语言遗留代码时就没得选择了。

尽管很容易使用，但是很容易忽视的一个问题是在 `PyArg_ParseTuple()` 中使用“s”格式化码会有内存损耗。 但你需要使用这种转换的时候，一个UTF-8字符串被创建并永久附加在原始字符串对象上面。 如果原始字符串包含非ASCII字符的话，就会导致字符串的尺寸增到一直到被垃圾回收。例如：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)     # Passing string
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)   # Notice increased size
103
>>>
```

如果你在乎这个内存的损耗，你最好重写你的C扩展代码，让它使用 `PyUnicode_AsUTF8String()` 函数。如下：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *o, *bytes;
  char *s;

  if (!PyArg_ParseTuple(args, "U", &o)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(o);
  s = PyBytes_AsString(bytes);
  print_chars(s);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

通过这个修改，一个UTF-8编码的字符串根据需要被创建，然后在使用过后被丢弃。下面是修订后的效果：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>
```

如果你试着传递NULL结尾字符串给ctypes包装过的函数， 要注意的是ctypes只能允许传递字节，并且它不会检查中间嵌入的NULL字节。例如：

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsample.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>>
```

如果你想传递字符串而不是字节，你需要先执行手动的UTF-8编码。例如：

```
>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>
```

对于其他扩展工具（比如Swig、Cython）， 在你使用它们传递字符串给C代码时要先好好学习相应的东西了。

15.14 传递Unicode字符串给C函数库¶

问题¶

你要写一个扩展模块，需要将一个Python字符串传递给C的某个库函数，但是这个函数不知道该怎么处理Unicode。

解决方案¶

这里我们需要考虑很多的问题，但是最主要的问题是现存的C函数库并不理解Python的原生Unicode表示。 因此，你的挑战是将Python字符串转换为一个能被C理解的形式。

为了演示的目的，下面有两个C函数，用来操作字符串数据并输出它来调试和测试。 一个使用形式为 `char *, int` 形式的字节， 而另一个使用形式为 `wchar_t *, int` 的宽字符形式：

```
void print_chars(char *s, int len) {
  int n = 0;

  while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}

void print_wchars(wchar_t *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%x ", s[n]);
    n++;
  }
  printf("\n");
}
```

对于面向字节的函数 `print_chars()` ，你需要将Python字符串转换为一个合适的编码比如UTF-8. 下面是一个这样的扩展函数例子：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

对于那些需要处理机器本地 `wchar_t` 类型的库函数，你可以像下面这样编写扩展代码：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  wchar_t *s;
  Py_ssize_t  len;

  if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
    return NULL;
  }
  print_wchars(s,len);
  Py_RETURN_NONE;
}
```

下面是一个交互会话来演示这个函数是如何工作的：

```
>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
```

```
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>
```

仔细观察这个面向字节的函数 `print_chars()` 是怎样接受UTF-8编码数据的， 以及 `print_wchars()` 是怎样接受Unicode编码值的

讨论¶

在继续本节之前，你应该首先学习你访问的C函数库的特征。 对于很多C函数库，通常传递字节而不是字符串会比较好些。要这样做，请使用如下的转换代码：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s;
  Py_ssize_t  len;

  /* accepts bytes, bytearray, or other byte-like object */
  if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  Py_RETURN_NONE;
}
```

如果你仍然还是想要传递字符串， 你需要知道Python 3可使用一个合适的字符串表示， 它并不直接映射到使用标准类型 `char *` 或 `wchar_t *` （更多细节参考PEP 393）的C函数库。 因此，要在C中表示这个字符串数据，一些转换还是必须要的。 在 `PyArg_ParseTuple()` 中使用”s#” 和”u#”格式化码可以安全的执行这样的转换。

不过这种转换有个缺点就是它可能会导致原始字符串对象的尺寸增大。 一旦转换过后，会有一个转换数据的复制附加到原始字符串对象上面，之后可以被重用。 你可以观察下这种效果：

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

对于少量的字符串对象，可能没什么影响， 但是如果你需要在扩展中处理大量的文本，你可能想避免这个损耗了。 下面是一个修订版本可以避免这种内存损耗：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s;
  Py_ssize_t   len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  bytes = PyUnicode_AsUTF8String(obj);
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

而对 `wchar_t` 的处理时想要避免内存损耗就更加难办了。 在内部，Python使用最高效的表示来存储字符串。 例如，只包含ASCII的字符串被存储为字节数组， 而包含范围从U+0000到U+FFFF的字符的字符串使用双字节表示。 由于对于数据的表示形式不是单一的，你不能将内部数组转换为 `wchar_t *` 然后期望它能正确的工作。 你应该创建一个

wchar_t 数组并向其中复制文本。 PyArg_ParseTuple() 的"u#"格式码可以帮助你高效的完成它（它将复制结果附加到字符串对象上）。

如果你想避免长时间内存损耗，你唯一的选择就是复制Unicode数据懂啊一个临时的数组， 将它传递给C函数，然后回收这个数组的内存。下面是一个可能的实现：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  wchar_t *s;
  Py_ssize_t len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
    return NULL;
  }
  print_wchars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

在这个实现中，PyUnicode_AsWideCharString() 创建一个临时的wchar_t缓冲并复制数据进去。 这个缓冲被传递给C然后被释放掉。 但是我写这本书的时候，这里可能有个bug，后面的Python问题页有介绍。

如果你知道C函数库需要的字节编码并不是UTF-8， 你可以强制Python使用扩展码来执行正确的转换，就像下面这样：

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  char *s = 0;
  int   len;
  if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {
    return NULL;
  }
  print_chars(s, len);
  PyMem_Free(s);
  Py_RETURN_NONE;
}
```

最后，如果你想直接处理Unicode字符串，下面的是例子，演示了底层操作访问：

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
  PyObject *obj;
  int n, len;
  int kind;
  void *data;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }
  if (PyUnicode_READY(obj) < 0) {
    return NULL;
  }

  len = PyUnicode_GET_LENGTH(obj);
  kind = PyUnicode_KIND(obj);
  data = PyUnicode_DATA(obj);

  for (n = 0; n < len; n++) {
    Py_UCS4 ch = PyUnicode_READ(kind, data, n);
    printf("%x ", ch);
  }
  printf("\n");
  Py_RETURN_NONE;
}
```

在这个代码中，PyUnicode_KIND() 和 PyUnicode_DATA() 这两个宏和Unicode的可变宽度存储有关，这个在PEP 393中有描述。 kind 变量编码底层存储（8位、16位或32位）以及指向缓存的数据指针相关的信息。 在实际情况中，你并不需要知道任何跟这些值有关的东西， 只需要在提取字符的时候将它们传给 PyUnicode_READ() 宏。

还有最后几句：当从Python传递Unicode字符串给C的时候，你应该尽量简单点。 如果有UTF-8和宽字符两种选择，请选择UTF-8. 对UTF-8的支持更加普遍一些，也不容易犯错，解释器也能支持的更好些。 最后，确保你仔细阅读了 关于处理Unicode的相关文档

15.15 C字符串转换为Python字符串¶

问题¶

怎样将C中的字符串转换为Python字节或一个字符串对象？

解决方案¶

C字符串使用一对 `char *` 和 `int` 来表示， 你需要决定字符串到底是用一个原始字节字符串还是一个Unicode字符串来表示。 字节对象可以像下面这样使用 `Py_BuildValue()` 来构建：

```
char *s;     /* Pointer to C string data */
int   len;   /* Length of data */

/* Make a bytes object */
PyObject *obj = Py_BuildValue("y#", s, len);
```

如果你要创建一个Unicode字符串，并且你知道 `s` 指向了UTF-8编码的数据，可以使用下面的方式：

```
PyObject *obj = Py_BuildValue("s#", s, len);
```

如果 `s` 使用其他编码方式，那么可以像下面使用 `PyUnicode_Decode()` 来构建一个字符串：

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");

/* Examples /*
obj = PyUnicode_Decode(s, len, "latin-1", "strict");
obj = PyUnicode_Decode(s, len, "ascii", "ignore");
```

如果你恰好有一个用 `wchar_t *, len` 对表示的宽字符串， 有几种选择性。首先你可以使用 `Py_BuildValue()` ：

```
wchar_t *w;    /* Wide character string */
int len;       /* Length */

PyObject *obj = Py_BuildValue("u#", w, len);
```

另外，你还可以使用 `PyUnicode_FromWideChar()` ：

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

对于宽字符串，并没有对字符数据进行解析——它被假定是原始Unicode编码指针，可以被直接转换成Python。

讨论¶

将C中的字符串转换为Python字符串遵循和I/O同样的原则。 也就是说，来自C中的数据必须根据一些解码器被显式的解码为一个字符串。 通常编码格式包括ASCII、Latin-1和UTF-8. 如果你并不确定编码方式或者数据是二进制的，你最好将字符串编码成字节。 当构造一个对象的时候，Python通常会复制你提供的字符串数据。 如果有必要的话，你需要在后面去释放C字符串。 同时，为了让程序更加健壮，你应该同时使用一个指针和一个大小值， 而不是依赖NULL结尾数据来创建字符串。

15.16 不确定编码格式的C字符串¶

问题¶

你要在C和Python直接来回转换字符串，但是C中的编码格式并不确定。 例如，可能C中的数据期望是UTF-8，但是并没有强制它必须是。 你想编写代码来以一种优雅的方式处理这些不合格数据，这样就不会让Python奔溃或者破坏进程中的字符串数据。

解决方案¶

下面是一些C的数据和一个函数来演示这个问题：

```
/* Some dubious string data (malformed UTF-8) */
const char *sdata = "Spicy Jalape\xc3\xb1o\xae";
int slen = 16;

/* Output character data */
void print_chars(char *s, int len) {
  int n = 0;
  while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
  }
  printf("\n");
}
```

在这个代码中，字符串 `sdata` 包含了UTF-8和不合格数据。 不过，如果用户在C中调用 `print_chars(sdata, slen)` ，它缺能正常工作。 现在假设你想将 `sdata` 的内容转换为一个Python字符串。 进一步假设你在后面还想通过一个扩展将那个字符串传个 `print_chars()` 函数。 下面是一种用来保护原始数据的方法，就算它编码有问题。

```
/* Return the C string back to Python */
static PyObject *py_retstr(PyObject *self, PyObject *args) {
  if (!PyArg_ParseTuple(args, "")) {
    return NULL;
  }
  return PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");
}

/* Wrapper for the print_chars() function */
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
  PyObject *obj, *bytes;
  char *s = 0;
  Py_ssize_t   len;

  if (!PyArg_ParseTuple(args, "U", &obj)) {
    return NULL;
  }

  if ((bytes = PyUnicode_AsEncodedString(obj,"utf-8","surrogateescape"))
        == NULL) {
    return NULL;
  }
  PyBytes_AsStringAndSize(bytes, &s, &len);
  print_chars(s, len);
  Py_DECREF(bytes);
  Py_RETURN_NONE;
}
```

如果你在Python中尝试这些函数，下面是运行效果：

```
>>> s = retstr()
>>> s
'Spicy Jalapeño\udcae'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
```

```
>>>
```

仔细观察结果你会发现，不合格字符串被编码到一个Python字符串中，并且并没有产生错误， 并且当它被回传给C的时候，被转换为和之前原始C字符串一样的字节。

讨论¶

本节展示了在扩展模块中处理字符串时会配到的一个棘手又很恼火的问题。 也就是说，在扩展中的C字符串可能不会严格遵循Python所期望的Unicode编码/解码规则。 因此，很可能一些不合格C数据传递到Python中去。 一个很好的例子就是涉及到底层系统调用比如文件名这样的字符串。 例如，如果一个系统调用返回给解释器一个损坏的字符串，不能被正确解码的时候会怎样呢？

一般来讲，可以通过制定一些错误策略比如严格、忽略、替代或其他类似的来处理Unicode错误。 不过，这些策略的一个缺点是它们永久性破坏了原始字符串的内容。 例如，如果例子中的不合格数据使用这些策略之一解码，你会得到下面这样的结果：

```
>>> raw = b'Spicy Jalape\xc3\xb1o\xae'
>>> raw.decode('utf-8','ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8','replace')
'Spicy Jalapeño?'
>>>
```

`surrogateescape` 错误处理策略会将所有不可解码字节转化为一个代理对的低位字节（udcXX中XX是原始字节值）。例如：

```
>>> raw.decode('utf-8','surrogateescape')
'Spicy Jalapeño\udcae'
>>>
```

单独的低位代理字符比如 `\udcae` 在Unicode中是非法的。 因此，这个字符串就是一个非法表示。 实际上，如果你将它传个一个执行输出的函数，你会得到一个错误：

```
>>> s = raw.decode('utf-8', 'surrogateescape')
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcae'
in position 14: surrogates not allowed
>>>
```

然而，允许代理转换的关键点在于从C传给Python又回传给C的不合格字符串不会有任何数据丢失。 当这个字符串再次使用 `surrogateescape` 编码时，代理字符会转换回原始字节。例如：

```
>>> s
'Spicy Jalapeño\udcae'
>>> s.encode('utf-8','surrogateescape')
b'Spicy Jalape\xc3\xb1o\xae'
>>>
```

作为一般准则，最好避免代理编码——如果你正确的使用了编码，那么你的代码就值得信赖。 不过，有时候确实会出现你并不能控制数据编码并且你又不能忽略或替换坏数据，因为其他函数可能会用到它。 那么就可以使用本节的技术了。

最后一点要注意的是，Python中许多面向系统的函数，特别是和文件名、环境变量和命令行参数相关的 都会使用代理编码。例如，如果你使用像 `os.listdir()` 这样的函数， 传入一个包含了不可解码文件名的目录的话，它会返回一个代理转换后的字符串。 参考5.15的相关章节。

PEP 383 中有更多关于本机提到的以及和surrogateescape错误处理相关的信息。

15.17 传递文件名给C扩展¶

问题¶

你需要向C库函数传递文件名，但是需要确保文件名根据系统期望的文件名编码方式编码过。

解决方案¶

写一个接受一个文件名为参数的扩展函数，如下这样：

```
static PyObject *py_get_filename(PyObject *self, PyObject *args) {
  PyObject *bytes;
  char *filename;
  Py_ssize_t len;
  if (!PyArg_ParseTuple(args,"O&", PyUnicode_FSConverter, &bytes)) {
    return NULL;
  }
  PyBytes_AsStringAndSize(bytes, &filename, &len);
  /* Use filename */
  ...

  /* Cleanup and return */
  Py_DECREF(bytes)
  Py_RETURN_NONE;
}
```

如果你已经有了一个 `PyObject *` ，希望将其转换成一个文件名，可以像下面这样做：

```
PyObject *obj;    /* Object with the filename */
PyObject *bytes;
char *filename;
Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj);
PyBytes_AsStringAndSize(bytes, &filename, &len);
/* Use filename */
...

/* Cleanup */
Py_DECREF(bytes);

If you need to return a filename back to Python, use the following code:

/* Turn a filename into a Python object */

char *filename;       /* Already set */
int   filename_len;   /* Already set */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filename_len);
```

讨论¶

以可移植方式来处理文件名是一个很棘手的问题，最后交由Python来处理。 如果你在扩展代码中使用本节的技术，文件名的处理方式和和Python中是一致的。 包括编码/界面字节，处理坏字符，代理转换和其他复杂情况。

15.18 传递已打开的文件给C扩展¶

问题¶

你在Python中有一个打开的文件对象，但是需要将它传给要使用这个文件的C扩展。

解决方案¶

要将一个文件转换为一个整型的文件描述符，使用 `PyFile_FromFd()` ，如下：

```
PyObject *fobj;     /* File object (already obtained somehow) */
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0) {
   return NULL;
}
```

结果文件描述符是通过调用 `fobj` 中的 `fileno()` 方法获得的。 因此，任何以这种方式暴露给一个描述器的对象都适用（比如文件、套接字等）。 一旦你有了这个描述器，它就能被传递给多个低级的可处理文件的C函数。

如果你需要转换一个整型文件描述符为一个Python对象，适用下面的 `PyFile_FromFd()` ：

```
int fd;     /* Existing file descriptor (already open) */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

`PyFile_FromFd()` 的参数对应内置的 `open()` 函数。 NULL表示编码、错误和换行参数使用默认值。

讨论¶

如果将Python中的文件对象传给C，有一些注意事项。 首先，Python通过 `io` 模块执行自己的I/O缓冲。 在传递任何类型的文件描述符给C之前，你都要首先在相应文件对象上刷新I/O缓冲。 不然的话，你会打乱文件系统上面的数据。

其次，你需要特别注意文件的归属者以及关闭文件的职责。 如果一个文件描述符被传给C，但是在Python中还在被使用着，你需要确保C没有意外的关闭它。 类似的，如果一个文件描述符被转换为一个Python文件对象，你需要清楚谁应该去关闭它。 `PyFile_FromFd()` 的最后一个参数被设置成1，用来指出Python应该关闭这个文件。

如果你需要从C标准I/O库中使用如 `fdopen()` 函数来创建不同类型的文件对象比如 `FILE *` 对象， 你需要特别小心了。这样做会在I/O堆栈中产生两个完全不同的I/O缓冲层 （一个是来自Python的 `io` 模块，另一个来自C的 `stdio` ）。像C中的 `fclose()` 会关闭Python要使用的文件。 如果让你选的话，你应该会选择去构建一个扩展代码来处理底层的整型文件描述符， 而不是使用来自<stdio.h>的高层抽象功能。

15.19 从C语言中读取类文件对象¶

问题¶

你要写C扩展来读取来自任何Python类文件对象中的数据（比如普通文件、StringIO对象等）。

解决方案¶

要读取一个类文件对象的数据，你需要重复调用 `read()` 方法，然后正确的解码获得的数据。

下面是一个C扩展函数例子，仅仅只是读取一个类文件对象中的所有数据并将其输出到标准输出：

```
#define CHUNK_SIZE 8192

/* Consume a "file-like" object and write bytes to stdout */
static PyObject *py_consume_file(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *read_meth;
  PyObject *result = NULL;
  PyObject *read_args;

  if (!PyArg_ParseTuple(args,"O", &obj)) {
    return NULL;
  }

  /* Get the read method of the passed object */
  if ((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
    return NULL;
  }

  /* Build the argument list to read() */
  read_args = Py_BuildValue("(i)", CHUNK_SIZE);
  while (1) {
    PyObject *data;
    PyObject *enc_data;
    char *buf;
    Py_ssize_t len;

    /* Call read() */
    if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
      goto final;
    }

    /* Check for EOF */
    if (PySequence_Length(data) == 0) {
      Py_DECREF(data);
      break;
    }

    /* Encode Unicode as Bytes for C */
    if ((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))==NULL) {
      Py_DECREF(data);
      goto final;
    }

    /* Extract underlying buffer data */
    PyBytes_AsStringAndSize(enc_data, &buf, &len);

    /* Write to stdout (replace with something more useful) */
    write(1, buf, len);

    /* Cleanup */
    Py_DECREF(enc_data);
    Py_DECREF(data);
  }
  result = Py_BuildValue("");
```

```
 final:
  /* Cleanup */
  Py_DECREF(read_meth);
  Py_DECREF(read_args);
  return result;
}
```

要测试这个代码，先构造一个类文件对象比如一个StringIO实例，然后传递进来：

```
>>> import io
>>> f = io.StringIO('Hello\nWorld\n')
>>> import sample
>>> sample.consume_file(f)
Hello
World
>>>
```

讨论¶

和普通系统文件不同的是，一个类文件对象并不需要使用低级文件描述符来构建。 因此，你不能使用普通的C库函数来访问它。 你需要使用Python的C API来像普通文件类似的那样操作类文件对象。

在我们的解决方案中，`read()` 方法从被传递的对象中提取出来。 一个参数列表被构建然后不断的被传给 `PyObject_Call()` 来调用这个方法。 要检查文件末尾（EOF），使用了 `PySequence_Length()` 来查看是否返回对象长度为0.

对于所有的I/O操作，你需要关注底层的编码格式，还有字节和Unicode之前的区别。 本节演示了如何以文本模式读取一个文件并将结果文本解码为一个字节编码，这样在C中就可以使用它了。 如果你想以二进制模式读取文件，只需要修改一点点即可，例如：

```
...
/* Call read() */
if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
  goto final;
}

/* Check for EOF */
if (PySequence_Length(data) == 0) {
  Py_DECREF(data);
  break;
}
if (!PyBytes_Check(data)) {
  Py_DECREF(data);
  PyErr_SetString(PyExc_IOError, "File must be in binary mode");
  goto final;
}

/* Extract underlying buffer data */
PyBytes_AsStringAndSize(data, &buf, &len);
...
```

本节最难的地方在于如何进行正确的内存管理。 当处理 `PyObject *` 变量的时候，需要注意管理引用计数以及在不需要的变量的时候清理它们的值。 对 `Py_DECREF()` 的调用就是来做这个的。

本节代码以一种通用方式编写，因此他也能适用于其他的文件操作，比如写文件。 例如，要写数据，只需要获取类文件对象的 `write()` 方法，将数据转换为合适的Python对象 （字节或Unicode），然后调用该方法将输入写入到文件。

最后，尽管类文件对象通常还提供其他方法（比如readline(), read_info()）， 我们最好只使用基本的 `read()` 和 `write()` 方法。 在写C扩展的时候，能简单就尽量简单。

15.2 简单的C扩展模块¶

问题¶

你想不依靠其他工具，直接使用Python的扩展API来编写一些简单的C扩展模块。

解决方案¶

对于简单的C代码，构建一个自定义扩展模块是很容易的。 作为第一步，你需要确保你的C代码有一个正确的头文件。例如：

```
/* sample.h */

#include <math.h>

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

通常来讲，这个头文件要对应一个已经被单独编译过的库。 有了这些，下面我们演示下编写扩展函数的一个简单例子：

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */
static PyObject *py_gcd(PyObject *self, PyObject *args) {
  int x, y, result;

  if (!PyArg_ParseTuple(args,"ii", &x, &y)) {
    return NULL;
  }
  result = gcd(x,y);
  return Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */
static PyObject *py_in_mandel(PyObject *self, PyObject *args) {
  double x0, y0;
  int n;
  int result;

  if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
    return NULL;
  }
  result = in_mandel(x0,y0,n);
  return Py_BuildValue("i", result);
}

/* int divide(int, int, int *) */
static PyObject *py_divide(PyObject *self, PyObject *args) {
  int a, b, quotient, remainder;
  if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
    return NULL;
  }
  quotient = divide(a,b, &remainder);
  return Py_BuildValue("(ii)", quotient, remainder);
}
```

```
/* Module method table */
static PyMethodDef SampleMethods[] = {
  {"gcd",  py_gcd, METH_VARARGS, "Greatest common divisor"},
  {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},
  {"divide", py_divide, METH_VARARGS, "Integer division"},
  { NULL, NULL, 0, NULL}
};

/* Module structure */
static struct PyModuleDef samplemodule = {
  PyModuleDef_HEAD_INIT,

  "sample",           /* name of module */
  "A sample module",  /* Doc string (may be NULL) */
  -1,                 /* Size of per-interpreter state or -1 */
  SampleMethods       /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  return PyModule_Create(&samplemodule);
}
```

要绑定这个扩展模块，像下面这样创建一个 `setup.py` 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      ext_modules=[
        Extension('sample',
                  ['pysample.c'],
                  include_dirs = ['/some/dir'],
                  define_macros = [('FOO','1')],
                  undef_macros = ['BAR'],
                  library_dirs = ['/usr/local/lib'],
                  libraries = ['sample']
                  )
        ]
)
```

为了构建最终的函数库，只需简单的使用 `python3 buildlib.py build_ext --inplace` 命令即可：

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

如上所示，它会创建一个名字叫 `sample.so` 的共享库。当被编译后，你就能将它作为一个模块导入进来了：

```
>>> import sample
>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)

0
>>> sample.divide(42, 8)
(5, 2)
>>>
```

如果你是在Windows机器上面尝试这些步骤，可能会遇到各种环境和编译问题，你需要花更多点时间去配置。 Python的

二进制分发通常使用了Microsoft Visual Studio来构建。 为了让这些扩展能正常工作，你需要使用同样或兼容的工具来编译它。 参考相应的 Python文档

讨论¶

在尝试任何手写扩展之前，最好能先参考下Python文档中的 扩展和嵌入Python解释器 . Python的C扩展API很大，在这里整个去讲述它没什么实际意义。 不过对于最核心的部分还是可以讨论下的。

首先，在扩展模块中，你写的函数都是像下面这样的一个普通原型：

```
static PyObject *py_func(PyObject *self, PyObject *args) {
  ...
}
```

`PyObject` 是一个能表示任何Python对象的C数据类型。 在一个高级层面，一个扩展函数就是一个接受一个Python对象（在 PyObject *args中）元组并返回一个新Python对象的C函数。 函数的 `self` 参数对于简单的扩展函数没有被使用到，不过如果你想定义新的类或者是C中的对象类型的话就能派上用场了。比如如果扩展函数是一个类的一个方法， 那么 `self` 就能引用那个实例了。

`PyArg_ParseTuple()` 函数被用来将Python中的值转换成C中对应表示。 它接受一个指定输入格式的格式化字符串作为输入，比如“i”代表整数，“d”代表双精度浮点数， 同样还有存放转换后结果的C变量的地址。 如果输入的值不匹配这个格式化字符串，就会抛出一个异常并返回一个NULL值。 通过检查并返回NULL，一个合适的异常会在调用代码中被抛出。

`Py_BuildValue()` 函数被用来根据C数据类型创建Python对象。 它同样接受一个格式化字符串来指定期望类型。 在扩展函数中，它被用来返回结果给Python。 `Py_BuildValue()` 的一个特性是它能构建更加复杂的对象类型，比如元组和字典。 在 `py_divide()` 代码中，一个例子演示了怎样返回一个元组。不过，下面还有一些实例：

```
return Py_BuildValue("i", 34);      // Return an integer
return Py_BuildValue("d", 3.4);     // Return a double
return Py_BuildValue("s", "Hello"); // Null-terminated UTF-8 string
return Py_BuildValue("(ii)", 3, 4); // Tuple (3, 4)
```

在扩展模块底部，你会发现一个函数表，比如本节中的 `SampleMethods` 表。 这个表可以列出C函数、Python中使用的名字、文档字符串。 所有模块都需要指定这个表，因为它在模块初始化时要被使用到。

最后的函数 `PyInit_sample()` 是模块初始化函数，但该模块第一次被导入时执行。 这个函数的主要工作是在解释器中注册模块对象。

最后一个要点需要提出来，使用C函数来扩展Python要考虑的事情还有很多，本节只是一小部分。 （实际上，C API包含了超过500个函数）。你应该将本节当做是一个入门篇。 更多高级内容，可以看看 `PyArg_ParseTuple()` 和 `Py_BuildValue()` 函数的文档， 然后进一步扩展开。

15.20 处理C语言中的可迭代对象¶

问题¶

你想写C扩展代码处理来自任何可迭代对象如列表、元组、文件或生成器中的元素。

解决方案¶

下面是一个C扩展函数例子，演示了怎样处理可迭代对象中的元素：

```
static PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
  PyObject *obj;
  PyObject *iter;
  PyObject *item;

  if (!PyArg_ParseTuple(args, "O", &obj)) {
    return NULL;
  }
  if ((iter = PyObject_GetIter(obj)) == NULL) {
    return NULL;
  }
  while ((item = PyIter_Next(iter)) != NULL) {
    /* Use item */
    ...
    Py_DECREF(item);
  }

  Py_DECREF(iter);
  return Py_BuildValue("");
}
```

讨论¶

本节中的代码和Python中对应代码类似。 `PyObject_GetIter()` 的调用和调用 `iter()` 一样可获得一个迭代器。 `PyIter_Next()` 函数调用 `next` 方法返回下一个元素或NULL(如果没有元素了)。 要注意正确的内存管理——`Py_DECREF()` 需要同时在产生的元素和迭代器对象本身上同时被调用， 以避免出现内存泄露。

15.21 诊断分段错误¶

问题¶

解释器因为某个分段错误、总线错误、访问越界或其他致命错误而突然间奔溃。 你想获得Python堆栈信息，从而找出在发生错误的时候你的程序运行点。

解决方案¶

`faulthandler` 模块能被用来帮你解决这个问题。 在你的程序中引入下列代码：

```
import faulthandler
faulthandler.enable()
```

另外还可以像下面这样使用 `-Xfaulthandler` 来运行Python：

```
bash % python3 -Xfaulthandler program.py
```

最后，你可以设置 `PYTHONFAULTHANDLER` 环境变量。 开启faulthandler后，在C扩展中的致命错误会导致一个Python错误堆栈被打印出来。例如：

```
Fatal Python error: Segmentation fault

Current thread 0x00007fff71106cc0:
  File "example.py", line 6 in foo
  File "example.py", line 10 in bar
  File "example.py", line 14 in spam
  File "example.py", line 19 in <module>
Segmentation fault
```

尽管这个并不能告诉你C代码中哪里出错了，但是至少能告诉你Python里面哪里有错。

讨论¶

faulthandler会在Python代码执行出错的时候向你展示跟踪信息。 至少，它会告诉你出错时被调用的最顶级扩展函数是哪个。 在pdb和其他Python调试器的帮助下，你就能追根溯源找到错误所在的位置了。

faulthandler不会告诉你任何C语言中的错误信息。 因此，你需要使用传统的C调试器，比如gdb。 不过，在faulthandler追踪信息可以让你去判断从哪里着手。 还要注意的是在C中某些类型的错误可能不太容易恢复。 例如，如果一个C扩展丢弃了程序堆栈信息，它会让faulthandler不可用， 那么你也得不到任何输出（除了程序奔溃外）。

15.3 编写扩展函数操作数组¶

问题¶

你想编写一个C扩展函数来操作数组，可能是被array模块或类似Numpy库所创建。 不过，你想让你的函数更加通用，而不是针对某个特定的库所生成的数组。

解决方案¶

为了能让接受和处理数组具有可移植性，你需要使用到 *Buffer Protocol* . 下面是一个手写的C扩展函数例子， 用来接受数组数据并调用本章开篇部分的 `avg(double *buf, int len)` 函数：

```
/* Call double avg(double *, int) */
static PyObject *py_avg(PyObject *self, PyObject *args) {
  PyObject *bufobj;
  Py_buffer view;
  double result;
  /* Get the passed Python object */
  if (!PyArg_ParseTuple(args, "O", &bufobj)) {
    return NULL;
  }

  /* Attempt to extract buffer information from it */

  if (PyObject_GetBuffer(bufobj, &view,
      PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return NULL;
  }

  if (view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Check the type of items in the array */
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    PyBuffer_Release(&view);
    return NULL;
  }

  /* Pass the raw buffer and size to the C function */
  result = avg(view.buf, view.shape[0]);

  /* Indicate we're done working with the buffer */
  PyBuffer_Release(&view);
  return Py_BuildValue("d", result);
}
```

下面我们演示下这个扩展函数是如何工作的：

```
>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
```

```
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])

2.0
>>>
```

讨论¶

将一个数组对象传给C函数可能是一个扩展函数做的最常见的事。 很多Python应用程序，从图像处理到科学计算，都是基于高性能的数组处理。 通过编写能接受并操作数组的代码，你可以编写很好的兼容这些应用程序的自定义代码，而不是只能兼容你自己的代码。

代码的关键点在于 `PyBuffer_GetBuffer()` 函数。 给定一个任意的Python对象，它会试着去获取底层内存信息，它简单的抛出一个异常并返回-1. 传给 `PyBuffer_GetBuffer()` 的特殊标志给出了所需的内存缓冲类型。 例如，`PyBUF_ANY_CONTIGUOUS` 表示是一个连续的内存区域。

对于数组、字节字符串和其他类似对象而言，一个 `Py_buffer` 结构体包含了所有底层内存的信息。 它包含一个指向内存地址、大小、元素大小、格式和其他细节的指针。下面是这个结构体的定义：

```
typedef struct bufferinfo {
    void *buf;              /* Pointer to buffer memory */
    PyObject *obj;          /* Python object that is the owner */
    Py_ssize_t len;         /* Total size in bytes */
    Py_ssize_t itemsize;    /* Size in bytes of a single item */
    int readonly;           /* Read-only access flag */
    int ndim;               /* Number of dimensions */
    char *format;           /* struct code of a single item */
    Py_ssize_t *shape;      /* Array containing dimensions */
    Py_ssize_t *strides;    /* Array containing strides */
    Py_ssize_t *suboffsets; /* Array containing suboffsets */
} Py_buffer;
```

本节中，我们只关注接受一个双精度浮点数数组作为参数。 要检查元素是否是一个双精度浮点数，只需验证 `format` 属性是不是字符串”d”. 这个也是 `struct` 模块用来编码二进制数据的。 通常来讲，`format` 可以是任何兼容 `struct` 模块的格式化字符串， 并且如果数组包含了C结构的话它可以包含多个值。 一旦我们已经确定了底层的缓存区信息，那只需要简单的将它传给C函数，然后会被当做是一个普通的C数组了。 实际上，我们不必担心是怎样的数组类型或者它是被什么库创建出来的。 这也是为什么这个函数能兼容 `array` 模块也能兼容 `numpy` 模块中的数组了。

在返回最终结果之前，底层的缓冲区视图必须使用 `PyBuffer_Release()` 释放掉。 之所以要这一步是为了能正确的管理对象的引用计数。

同样，本节也仅仅只是演示了接受数组的一个小的代码片段。 如果你真的要处理数组，你可能会碰到多维数据、大数据、不同的数据类型等等问题， 那么就得去学更高级的东西了。你需要参考官方文档来获取更多详细的细节。

如果你需要编写涉及到数组处理的多个扩展，那么通过Cython来实现会更容易下。参考15.11节。

15.4 在C扩展模块中操作隐形指针¶

问题¶

你有一个扩展模块需要处理C结构体中的指针， 但是你又不想暴露结构体中任何内部细节给Python。

解决方案¶

隐形结构体可以很容易的通过将它们包装在胶囊对象中来处理。 考虑我们例子代码中的下列C代码片段：

```
typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

下面是一个使用胶囊包装Point结构体和 `distance()` 函数的扩展代码实例：

```
/* Destructor function for points */
static void del_Point(PyObject *obj) {
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
  return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

/* Create a new Point object */
static PyObject *py_Point(PyObject *self, PyObject *args) {

  Point *p;
  double x,y;
  if (!PyArg_ParseTuple(args,"dd",&x,&y)) {
    return NULL;
  }
  p = (Point *) malloc(sizeof(Point));
  p->x = x;
  p->y = y;
  return PyPoint_FromPoint(p, 1);
}

static PyObject *py_distance(PyObject *self, PyObject *args) {
  Point *p1, *p2;
  PyObject *py_p1, *py_p2;
  double result;

  if (!PyArg_ParseTuple(args,"OO",&py_p1, &py_p2)) {
    return NULL;
  }
  if (!(p1 = PyPoint_AsPoint(py_p1))) {
    return NULL;
  }
  if (!(p2 = PyPoint_AsPoint(py_p2))) {
    return NULL;
  }
  result = distance(p1,p2);
  return Py_BuildValue("d", result);
}
```

在Python中可以像下面这样来使用这些函数：

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1004ea330>
>>> p2
<capsule object "Point" at 0x1005d1db0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

讨论¶

胶囊和C指针类似。在内部，它们获取一个通用指针和一个名称，可以使用 `PyCapsule_New()` 函数很容易的被创建。另外，一个可选的析构函数能被绑定到胶囊上，用来在胶囊对象被垃圾回收时释放底层的内存。

要提取胶囊中的指针，可使用 `PyCapsule_GetPointer()` 函数并指定名称。 如果提供的名称和胶囊不匹配或其他错误出现，那么就会抛出异常并返回NULL。

本节中，一对工具函数—— `PyPoint_FromPoint()` 和 `PyPoint_AsPoint()` 被用来创建和从胶囊对象中提取Point实例。在任何扩展函数中，我们会使用这些函数而不是直接使用胶囊对象。 这种设计使得我们可以很容易的应对将来对Point底下的包装的更改。 例如，如果你决定使用另外一个胶囊了，那么只需要更改这两个函数即可。

对于胶囊对象一个难点在于垃圾回收和内存管理。 `PyPoint_FromPoint()` 函数接受一个 `must_free` 参数， 用来指定当胶囊被销毁时底层Point * 结构体是否应该被回收。 在某些C代码中，归属问题通常很难被处理（比如一个Point结构体被嵌入到一个被单独管理的大结构体中）。 程序员可以使用 `extra` 参数来控制，而不是单方面的决定垃圾回收。 要注意的是和现有胶囊有关的析构器能使用 `PyCapsule_SetDestructor()` 函数来更改。

对于涉及到结构体的C代码而言，使用胶囊是一个比较合理的解决方案。 例如，有时候你并不关心暴露结构体的内部信息或者将其转换成一个完整的扩展类型。 通过使用胶囊，你可以在它上面放一个轻量级的包装器，然后将它传给其他的扩展函数。

15.5 从扩展模块中定义和导出C的API¶

问题¶

你有一个C扩展模块，在内部定义了很多有用的函数，你想将它们导出为一个公共的C API供其他地方使用。 你想在其他扩展模块中使用这些函数，但是不知道怎样将它们链接起来， 并且通过C编译器/链接器来做看上去特别复杂（或者不可能做到）。

解决方案¶

本节主要问题是如何处理15.4小节中提到的Point对象。仔细回一下，在C代码中包含了如下这些工具函数：

```
/* Destructor function for points */
static void del_Point(PyObject *obj) {

  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
  return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}
```

现在的问题是怎样将 `PyPoint_AsPoint()` 和 `Point_FromPoint()` 函数作为API导出， 这样其他扩展模块能使用并链接它们，比如如果你有其他扩展也想使用包装的Point对象。

要解决这个问题，首先要为 `sample` 扩展写个新的头文件名叫 `pysample.h` ，如下：

```
/* pysample.h */
#include "Python.h"
#include "sample.h"
#ifdef __cplusplus
extern "C" {
#endif

/* Public API Table */
typedef struct {
  Point *(*aspoint)(PyObject *);
  PyObject *(*frompoint)(Point *, int);
} _PointAPIMethods;

#ifndef PYSAMPLE_MODULE
/* Method table in external module */
static _PointAPIMethods *_point_api = 0;

/* Import the API table from sample */
static int import_sample(void) {
  _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._point_api",0);
  return (_point_api != NULL) ? 1 : 0;
}

/* Macros to implement the programming interface */
#define PyPoint_AsPoint(obj) (_point_api->aspoint)(obj)
#define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus
}
#endif
```

这里最重要的部分是函数指针表 `_PointAPIMethods` . 它会在导出模块时被初始化，然后导入模块时被查找到。 修改原

始的扩展模块来填充表格并将它像下面这样导出：

```
/* pysample.c */

#include "Python.h"
#define PYSAMPLE_MODULE
#include "pysample.h"

...
/* Destructor function for points */
static void del_Point(PyObject *obj) {
  printf("Deleting point\n");
  free(PyCapsule_GetPointer(obj,"Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
  return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int free) {
  return PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

static _PointAPIMethods _point_api = {
  PyPoint_AsPoint,
  PyPoint_FromPoint
};
...

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  PyObject *m;
  PyObject *py_point_api;

  m = PyModule_Create(&samplemodule);
  if (m == NULL)
    return NULL;

  /* Add the Point C API functions */
  py_point_api = PyCapsule_New((void *) &_point_api, "sample._point_api", NULL);
  if (py_point_api) {
    PyModule_AddObject(m, "_point_api", py_point_api);
  }
  return m;
}
```

最后，下面是一个新的扩展模块例子，用来加载并使用这些API函数：

```
/* ptexample.c */

/* Include the header associated with the other module */
#include "pysample.h"

/* An extension function that uses the exported API */
static PyObject *print_point(PyObject *self, PyObject *args) {
  PyObject *obj;
  Point *p;
  if (!PyArg_ParseTuple(args,"O", &obj)) {
    return NULL;
  }

  /* Note: This is defined in a different module */
  p = PyPoint_AsPoint(obj);
  if (!p) {
    return NULL;
  }
  printf("%f %f\n", p->x, p->y);
  return Py_BuildValue("");
```

```
}

static PyMethodDef PtExampleMethods[] = {
  {"print_point", print_point, METH_VARARGS, "output a point"},
  { NULL, NULL, 0, NULL}
};

static struct PyModuleDef ptexamplemodule = {
  PyModuleDef_HEAD_INIT,
  "ptexample",           /* name of module */
  "A module that imports an API",  /* Doc string (may be NULL) */
  -1,                 /* Size of per-interpreter state or -1 */
  PtExampleMethods       /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_ptexample(void) {
  PyObject *m;

  m = PyModule_Create(&ptexamplemodule);
  if (m == NULL)
    return NULL;

  /* Import sample, loading its API functions */
  if (!import_sample()) {
    return NULL;
  }

  return m;
}
```

编译这个新模块时，你甚至不需要去考虑怎样将函数库或代码跟其他模块链接起来。 例如，你可以像下面这样创建一个简单的 `setup.py` 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='ptexample',
      ext_modules=[
        Extension('ptexample',
                  ['ptexample.c'],
                  include_dirs = [],  # May need pysample.h directory
                  )
        ]
)
```

如果一切正常，你会发现你的新扩展函数能和定义在其他模块中的C API函数一起运行的很好。

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample
>>> ptexample.print_point(p1)
2.000000 3.000000
>>>
```

讨论¶

本节基于一个前提就是，胶囊对象能获取任何你想要的对象的指针。 这样的话，定义模块会填充一个函数指针的结构体，创建一个指向它的胶囊，并在一个模块级属性中保存这个胶囊， 例如 `sample._point_api` .

其他模块能够在导入时获取到这个属性并提取底层的指针。 事实上，Python提供了 `PyCapsule_Import()` 工具函数，为了完成所有的步骤。 你只需提供属性的名字即可（比如sample._point_api），然后他就会一次性找到胶囊对象并提取出指针来。

在将被导出函数变为其他模块中普通函数时，有一些C编程陷阱需要指出来。 在 `pysample.h` 文件中，一个

`_point_api` 指针被用来指向在导出模块中被初始化的方法表。 一个相关的函数 `import_sample()` 被用来指向胶囊导入并初始化这个指针。 这个函数必须在任何函数被使用之前被调用。通常来讲，它会在模块初始化时被调用到。 最后，C的预处理宏被定义，被用来通过方法表去分发这些API函数。 用户只需要使用这些原始函数名称即可，不需要通过宏去了解其他信息。

最后，还有一个重要的原因让你去使用这个技术来链接模块——它非常简单并且可以使得各个模块很清晰的解耦。 如果你不想使用本机的技术，那你就必须使用共享库的高级特性和动态加载器来链接模块。 例如，将一个普通的API函数放入一个共享库并确保所有扩展模块链接到那个共享库。 这种方法确实可行，但是它相对繁琐，特别是在大型系统中。 本节演示了如何通过Python的普通导入机制和仅仅几个胶囊调用来将多个模块链接起来的魔法。 对于模块的编译，你只需要定义头文件，而不需要考虑函数库的内部细节。

更多关于利用C API来构造扩展模块的信息可以参考 Python的文档

15.6 从C语言中调用Python代码¶

问题¶

你想在C中安全的执行某个Python调用并返回结果给C。 例如，你想在C语言中使用某个Python函数作为一个回调。

解决方案¶

在C语言中调用Python非常简单，不过涉及到一些小窍门。 下面的C代码告诉你怎样安全的调用：

:: c

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter. The
    arguments and return result of the function must be Python floats */
double call_func(PyObject *func, double x, double y) {

    PyObject *args; PyObject *kwargs; PyObject *result = 0; double retval;

    /* Make sure we own the GIL */ PyGILState_STATE state = PyGILState_Ensure();

    /* Verify that func is a proper callable */ if (!PyCallable_Check(func)) {

        fprintf(stderr,"call_func: expected a callablen"); goto fail;

    } /* Build arguments */ args = Py_BuildValue("(dd)", x, y); kwargs = NULL;

    /* Call the function */ result = PyObject_Call(func, args, kwargs); Py_DECREF(args); Py_XDECREF(kwargs);

    /* Check for Python exceptions (if any) */ if (PyErr_Occurred()) {

        PyErr_Print(); goto fail;

    }

    /* Verify the result is a float object */ if (!PyFloat_Check(result)) {

        fprintf(stderr,"call_func: callable didn't return a floatn"); goto fail;

    }

    /* Create the return value */ retval = PyFloat_AsDouble(result); Py_DECREF(result);

    /* Restore previous GIL state and return */ PyGILState_Release(state); return retval;

fail:
    Py_XDECREF(result); PyGILState_Release(state); abort(); // Change to something more appropriate

}
```

要使用这个函数，你需要获取传递过来的某个已存在Python调用的引用。 有很多种方法可以让你这样做， 比如将一个可调用对象传给一个扩展模块或直接写C代码从已存在模块中提取出来。

下面是一个简单例子用来展示从一个嵌入的Python解释器中调用一个函数：

```
#include <Python.h>

/* Definition of call_func() same as above */
...
```

```
/* Load a symbol from a module */
PyObject *import_name(const char *modname, const char *symbol) {
  PyObject *u_name, *module;
  u_name = PyUnicode_FromString(modname);
  module = PyImport_Import(u_name);
  Py_DECREF(u_name);
  return PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */
int main() {
  PyObject *pow_func;
  double x;

  Py_Initialize();
  /* Get a reference to the math.pow function */
  pow_func = import_name("math","pow");

  /* Call it using our call_func() code */
  for (x = 0.0; x < 10.0; x += 0.1) {
    printf("%0.2f %0.2f\n", x, call_func(pow_func,x,2.0));
  }
  /* Done */
  Py_DECREF(pow_func);
  Py_Finalize();
  return 0;
}
```

要构建例子代码，你需要编译C并将它链接到Python解释器。 下面的Makefile可以教你怎样做（不过在你机器上面需要一些配置）。

```
all::
        cc -g embed.c -I/usr/local/include/python3.3m \
          -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m
```

编译并运行会产生类似下面的输出：

```
0.00 0.00
0.10 0.01
0.20 0.04
0.30 0.09
0.40 0.16
...
```

下面是一个稍微不同的例子，展示了一个扩展函数， 它接受一个可调用对象和其他参数，并将它们传递给 `call_func()` 来做测试：

```
/* Extension function for testing the C-Python callback */
PyObject *py_call_func(PyObject *self, PyObject *args) {
  PyObject *func;

  double x, y, result;
  if (!PyArg_ParseTuple(args,"Odd", &func,&x,&y)) {
    return NULL;
  }
  result = call_func(func, x, y);
  return Py_BuildValue("d", result);
}
```

使用这个扩展函数，你要像下面这样测试它：

```
>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>
```

讨论¶

如果你在C语言中调用Python，要记住最重要的是C语言会是主体。 也就是说，C语言负责构造参数、调用Python函数、检查异常、检查类型、提取返回值等。

作为第一步，你必须先有一个表示你将要调用的Python可调用对象。 这可以是一个函数、类、方法、内置方法或其他任意实现了 `__call__()` 操作的东西。 为了确保是可调用的，可以像下面的代码这样利用 `PyCallable_Check()` 做检查：

```
double call_func(PyObject *func, double x, double y) {
  ...
  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  ...
```

在C代码里处理错误你需要格外的小心。一般来讲，你不能仅仅抛出一个Python异常。 错误应该使用C代码方式来被处理。在这里，我们打算将对错误的控制传给一个叫 `abort()` 的错误处理器。 它会结束掉整个程序，在真实环境下面你应该要处理的更加优雅些（返回一个状态码）。 你要记住的是在这里C是主角，因此并没有跟抛出异常相对应的操作。 错误处理是你在编程时必须要考虑的事情。

调用一个函数相对来讲很简单——只需要使用 `PyObject_Call()`， 传一个可调用对象给它、一个参数元组和一个可选的关键字字典。 要构建参数元组或字典，你可以使用 `Py_BuildValue()` ,如下：

```
double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;

  ...
  /* Build arguments */
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;

  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);
  ...
```

如果没有关键字参数，你可以传递NULL。当你要调用函数时， 需要确保使用了 `Py_DECREF()` 或者 `Py_XDECREF()` 清理参数。 第二个函数相对安全点，因为它允许传递NULL指针（直接忽略它）， 这也是为什么我们使用它来清理可选的关键字参数。

调用完Python函数之后，你必须检查是否有异常发生。 `PyErr_Occurred()` 函数可被用来做这件事。 对于异常的处理就有点麻烦了，由于是用C语言写的，没有Python那样的异常机制。 因此，你必须要设置一个异常状态码，打印异常信息或其他相应处理。 在这里，我们选择了简单的 `abort()` 来处理。另外，传统C程序员可能会直接让程序奔溃。

```
...
/* Check for Python exceptions (if any) */
if (PyErr_Occurred()) {
  PyErr_Print();
  goto fail;
}
...
fail:
  PyGILState_Release(state);
  abort();
```

从调用Python函数的返回值中提取信息通常要进行类型检查和提取值。 要这样做的话，你必须使用Python对象层中的函数。 在这里我们使用了 `PyFloat_Check()` 和 `PyFloat_AsDouble()` 来检查和提取Python浮点数。

最后一个问题是对于Python全局锁的管理。 在C语言中访问Python的时候，你需要确保GIL被正确的获取和释放了。 不然的话，可能会导致解释器返回错误数据或者直接奔溃。 调用 `PyGILState_Ensure()` 和 `PyGILState_Release()` 可以

确保一切都能正常。

```
double call_func(PyObject *func, double x, double y) {
  ...
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();
  ...
  /* Code that uses Python C API functions */
  ...
  /* Restore previous GIL state and return */
  PyGILState_Release(state);
  return retval;

fail:
  PyGILState_Release(state);
  abort();
}
```

一旦返回，`PyGILState_Ensure()` 可以确保调用线程独占Python解释器。 就算C代码运行于另外一个解释器不知道的线程也没事。 这时候，C代码可以自由的使用任何它想要的Python C-API 函数。 调用成功后，PyGILState_Release()被用来将解释器恢复到原始状态。

要注意的是每一个 `PyGILState_Ensure()` 调用必须跟着一个匹配的 `PyGILState_Release()` 调用——即便有错误发生。在这里，我们使用一个 `goto` 语句看上去是个可怕的设计， 但是实际上我们使用它来将控制权转移给一个普通的exit块来执行相应的操作。 在 `fail:` 标签后面的代码和Python的 `final:` 块的用途是一样的。

如果你使用所有这些约定来编写C代码，包括对GIL的管理、异常检查和错误检查， 你会发现从C语言中调用Python解释器是可靠的——就算再复杂的程序，用到了高级编程技巧比如多线程都没问题。

15.7 从C扩展中释放全局锁¶

问题¶

你想让C扩展代码和Python解释器中的其他进程一起正确的执行， 那么你就需要去释放并重新获取全局解释器锁（GIL）。

解决方案¶

在C扩展代码中，GIL可以通过在代码中插入下面这样的宏来释放和重新获取：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code.  Must not use Python API functions
   ...
   Py_END_ALLOW_THREADS
   ...
   return result;
}
```

讨论¶

只有当你确保没有Python C API函数在C中执行的时候你才能安全的释放GIL。 GIL需要被释放的常见的场景是在计算密集型代码中需要在C数组上执行计算（比如在numpy中） 或者是要执行阻塞的I/O操作时（比如在一个文件描述符上读取或写入时）。

当GIL被释放后，其他Python线程才被允许在解释器中执行。 `Py_END_ALLOW_THREADS` 宏会阻塞执行直到调用线程重新获取了GIL。

15.8 C和Python中的线程混用¶

问题¶

你有一个程序需要混合使用C、Python和线程， 有些线程是在C中创建的，超出了Python解释器的控制范围。 并且一些线程还使用了Python C API中的函数。

解决方案¶

如果你想将C、Python和线程混合在一起，你需要确保正确的初始化和管理Python的全局解释器锁（GIL）。 要想这样做，可以将下列代码放到你的C代码中并确保它在任何线程被创建之前被调用。

```
#include <Python.h>
  ...
  if (!PyEval_ThreadsInitialized()) {
    PyEval_InitThreads();
  }
  ...
```

对于任何调用Python对象或Python C API的C代码，确保你首先已经正确地获取和释放了GIL。 这可以用 `PyGILState_Ensure()` 和 `PyGILState_Release()` 来做到，如下所示：

```
...
/* Make sure we own the GIL */
PyGILState_STATE state = PyGILState_Ensure();

/* Use functions in the interpreter */
...
/* Restore previous GIL state and return */
PyGILState_Release(state);
...
```

每次调用 `PyGILState_Ensure()` 都要相应的调用 `PyGILState_Release()` .

讨论¶

在涉及到C和Python的高级程序中，很多事情一起做是很常见的—— 可能是对C、Python、C线程、Python线程的混合使用。 只要你确保解释器被正确的初始化，并且涉及到解释器的C代码执行了正确的GIL管理，应该没什么问题。

要注意的是调用 `PyGILState_Ensure()` 并不会立刻抢占或中断解释器。 如果有其他代码正在执行，这个函数被中断知道那个执行代码释放掉GIL。 在内部，解释器会执行周期性的线程切换，因此如果其他线程在执行， 调用者最终还是可以运行的（尽管可能要先等一会）。

15.9 用SWIG包装C代码¶

问题¶

你想让你写的C代码作为一个C扩展模块来访问，想通过使用 Swig包装生成器 来完成。

解决方案¶

Swig通过解析C头文件并自动创建扩展代码来操作。 要使用它，你先要有一个C头文件。例如，我们示例的头文件如下：

```
/* sample.h */

#include <math.h>
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

一旦你有了这个头文件，下一步就是编写一个Swig”接口”文件。 按照约定，这些文件以”.i”后缀并且类似下面这样：

```
// sample.i - Swig interface
%module sample
%{
#include "sample.h"
%}

/* Customizations */
%extend Point {
    /* Constructor for Point objects */
    Point(double x, double y) {
        Point *p = (Point *) malloc(sizeof(Point));
        p->x = x;
        p->y = y;
        return p;
   };
};

/* Map int *remainder as an output argument */
%include typemaps.i
%apply int *OUTPUT { int * remainder };

/* Map the argument pattern (double *a, int n) to arrays */
%typemap(in) (double *a, int n)(Py_buffer view) {
  view.obj = NULL;
  if (PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    SWIG_fail;
  }
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    SWIG_fail;
  }
  $1 = (double *) view.buf;
  $2 = view.len / sizeof(double);
}

%typemap(freearg) (double *a, int n) {
  if (view$argnum.obj) {
    PyBuffer_Release(&view$argnum);
  }
```

```
}

/* C declarations to be included in the extension module */

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

一旦你写好了接口文件，就可以在命令行工具中调用Swig了：

```
bash % swig -python -py3 sample.i
bash %
```

swig的输出就是两个文件，sample_wrap.c和sample.py。 后面的文件就是用户需要导入的。 而sample_wrap.c文件是需要被编译到名叫 `_sample` 的支持模块的C代码。 这个可以通过跟普通扩展模块一样的技术来完成。 例如，你创建了一个如下所示的 `setup.py` 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      py_modules=['sample.py'],
      ext_modules=[
        Extension('_sample',
                  ['sample_wrap.c'],
                  include_dirs = [],
                  define_macros = [],

                  undef_macros = [],
                  library_dirs = [],
                  libraries = ['sample']
                  )
        ]
)
```

要编译和测试，在setup.py上执行python3，如下：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building '_sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c sample_wrap.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
sample_wrap.c: In function ‘SWIG_InitializeModule’:
sample_wrap.c:3589: warning: statement with no effect
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o
 build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample
bash %
```

如果一切正常的话，你会发现你就可以很方便的使用生成的C扩展模块了。例如：

```
>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
```

```
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>
```

讨论¶

Swig是Python历史中构建扩展模块的最古老的工具之一。 Swig能自动化很多包装生成器的处理。

所有Swig接口都以类似下面这样的为开头：

```
%module sample
%{
#include "sample.h"
%}
```

这个仅仅只是声明了扩展模块的名称并指定了C头文件， 为了能让编译通过必须要包含这些头文件（位于 %{ 和 %} 的代码）， 将它们之间复制粘贴到输出代码中，这也是你要放置所有包含文件和其他编译需要的定义的地方。

Swig接口的底下部分是一个C声明列表，你需要在扩展中包含它。 这通常从头文件中被复制。在我们的例子中，我们仅仅像下面这样直接粘贴在头文件中：

```
%module sample
%{
#include "sample.h"
%}
...
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

有一点需要强调的是这些声明会告诉Swig你想要在Python模块中包含哪些东西。 通常你需要编辑这个声明列表或相应的修改下它。 例如，如果你不想某些声明被包含进来，你要将它从声明列表中移除掉。

使用Swig最复杂的地方是它能给C代码提供大量的自定义操作。 这个主题太大，这里无法展开，但是我们在本节还剩展示了一些自定义的东西。

第一个自定义是 `%extend` 指令允许方法被附加到已存在的结构体和类定义上。 我例子中，这个被用来添加一个Point结构体的构造器方法。 它可以让你像下面这样使用这个结构体：

```
>>> p1 = sample.Point(2,3)
>>>
```

如果略过的话，Point对象就必须以更加复杂的方式来被创建：

```
>>> # Usage if %extend Point is omitted
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

第二个自定义涉及到对 `typemaps.i` 库的引入和 `%apply` 指令， 它会指示Swig参数签名 `int *remainder` 要被当做是输出值。 这个实际上是一个模式匹配规则。 在接下来的所有声明中，任何时候只要碰上 `int *remainder`，他就会被作为输出。 这个自定义方法可以让 `divide()` 函数返回两个值。

```
>>> sample.divide(42,8)
[5, 2]
```

```
>>>
```

最后一个涉及到 `%typemap` 指令的自定义可能是这里展示的最高级的特性了。 一个typemap就是一个在输入中特定参数模式的规则。 在本节中，一个typemap被定义为匹配参数模式 `(double *a, int n)`. 在typemap内部是一个C代码片段，它告诉Swig怎样将一个Python对象转换为相应的C参数。 本节代码使用了Python的缓存协议去匹配任何看上去类似双精度数组的输入参数 （比如NumPy数组、array模块创建的数组等），更多请参考15.3小节。

在typemap代码内部，$1和$2这样的变量替换会获取typemap模式的C参数值 （比如$1映射为 `double *a` ）。$input指向一个作为输入的 `PyObject *` 参数， 而 `$argnum` 就代表参数的个数。

编写和理解typemaps是使用Swig最基本的前提。 不仅是说代码更神秘，而且你需要理解Python C API和Swig和它交互的方式。 Swig文档有更多这方面的细节，可以参考下。

不过，如果你有大量的C代码需要被暴露为扩展模块。 Swig是一个非常强大的工具。关键点在于Swig是一个处理C声明的编译器， 通过强大的模式匹配和自定义组件，可以让你更改声明指定和类型处理方式。 更多信息请去查阅 Swig网站， 还有 特定于Python的相关文档

第十五章：C语言扩展¶

本章着眼于从Python访问C代码的问题。许多Python内置库是用C写的， 访问C是让Python的对现有库进行交互一个重要的组成部分。 这也是一个当你面临从Python 2 到 Python 3扩展代码的问题。 虽然Python提供了一个广泛的编程API，实际上有很多方法来处理C的代码。 相比试图给出对于每一个可能的工具或技术的详细参考， 我们采用的是是集中在一个小片段的C++代码，以及一些有代表性的例子来展示如何与代码交互。 这个目标是提供一系列的编程模板，有经验的程序员可以扩展自己的使用。

这里是我们将在大部分秘籍中工作的代码：

```
/* sample.c */_method
#include <math.h>

/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

/* Test if (x0,y0) is in the Mandelbrot set or not */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = xtemp;
        n -= 1;
        if (x*x + y*y > 4) return 0;
    }
    return 1;
}

/* Divide two numbers */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}

/* Average values in an array */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}

/* A C data structure */
typedef struct Point {
    double x,y;
} Point;

/* Function involving a C data structure */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}
```

这段代码包含了多种不同的C语言编程特性。 首先，这里有很多函数比如 `gcd()` 和 `is_mandel()` 。 `divide()` 函数是一个返回多个值的C函数例子，其中有一个是通过指针参数的方式。 `avg()` 函数通过一个C数组执行数据聚集操作。`Point` 和 `distance()` 函数涉及到了C结构体。

对于接下来的所有小节，先假定上面的代码已经被写入了一个名叫“sample.c”的文件中， 然后它们的定义被写入一个名叫“sample.h”的头文件中， 并且被编译为一个库叫“libsample”，能被链接到其他C语言代码中。 编译和链接的细节依据系统的不同而不同，但是这个不是我们关注的。 如果你要处理C代码，我们假定这些基础的东西你都掌握了。

Contents: