Cython 官方文档中文版

moonlet

Published with GitBook

目錄

README 0
基础教程 1
C 函数的调用 2
C 库的使用 3

Cython 官方文档中文版

英文版地址：http://docs.cython.org/src/tutorial/

> 1. 本文档基于官方 0.23.2 翻译.
> 2. 有问题请联系 xiaorixin@gmail.com, bahuafeng@gmail.com

规范名词

英文名	中文译名
extension	拓展
module	模块

基础教程

Cython 的基础

Cython 的本质可以总结如下：Cython 是包含 C 数据类型的 Python。

Cython 是 Python：几乎所有 Python 代码都是合法的 Cython 代码。（存在一些限制，但是差不多也可以。） Cython 的编译器会转化 Python 代码为 C 代码，这些 C 代码均可以调用 Python/C 的 API。

Cython 可不仅仅包含这些，Cython 中的参数和变量还可以以 C 数据类型来声明。代码中的 Python 值和 C 的值可以自由地交叉混合（intermixed）使用, 所有的转化都是自动进行。Python 中的引用计数维护（Reference count maintenance）和错误检查（error checking）操作同样是自动进行的，并且全面支持 Python 的异常处理工具（facilities），包括 `try-except` 和 `try-finally`，即便在其中操作 C 数据都是可以的。

Cython 的 Hello World

由于 Cython 能接受几乎所有的合法 Python 源文件，开始使用 Cython 的最难的事情之一是怎么编译你的拓展（extension）。

那么，让我们从典型的（canonical）Python `hello world` 开始：

```
print "Hello World"
```

将代码保存在文件 `helloworld.pyx` 中。现在，我们需要创建 `setup.py`，它是一个类似 Python Makefile 的文件（更多信息请看源文件和编译过程）。你编写的 `setup.py` 应该看起来类似这样：

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

输入如下命令来构建你的 Cython 文件:

```
$ python setup.py build_ext --inplace
```

运行完上述命令会在你的当前目录生成一个新文件，如果你的系统是 Unix，文件名为 `helloworld.so` ，如果你的系统是 Windows，文件名为 `helloworld.pyd` 。现在我们用一用刚生成的文件：打开 Python 的解释器（interpreter），像 import 普通文件一样直接 `import` 你刚生成的文件：

```
>>> import helloworld
Hello World
```

恭喜！你已经学会了怎样构建 Cython 的拓展了。但是到现在为止，这个例子并没有给我们一个使用 Cython 的理由。所以，让我们创建一个更现实的例子。

pyximport：Cython 简单编译

如果你的模块不需要额外的 C 库活特殊的构件安装，那你可以在 `import` 时使用 Paul Prescod 和 Stefan Behnel 编写的 `pyximport` 模块来直接读取 `.pyx` 文件，而不需要编写 `setup.py` 文件。 它随同 Cython 一并发布和安装，你可以这样使用它：

```
>>> import pyximport; pyximport.install()
>>> import helloworld
Hello World
```

自 Cython 0.11 起， `pyximport` 模块同样实验性地支持普通 Python 模块的编译了。它允许你在所有 Python import 的 `.pyx` 和 `.py` 模块上自动运行 Cython，包括哪些标准库和第三方库。但是，任然有不少 Python 模块 Cython 无法编译，遇到这种情况 import 机制（mechanism）会退回去读取 Python 原模块。 `.py` 的 import 机制可按如下方式安装：

```
>>> pyximport.install(pyimport = True)
```

斐波那契（Fibonacci）函数

Python 的官方教程中斐波那契函数是这样定义的：

```
def fib(n):
    """Print the Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a + b
```

现在，我们模仿 Hello World 例子中的步骤，第一步将 Python 官方教程中斐波那契函数文件名改为 `.pyx`，例如 `fib.pyx`，然后我们创建 `setup.py` 文件。你只需要修改一下 Cython 文件的文件名和生成模块的名字就可以直接复用 Hello World 例子中的 `setup.py` 文件。这样，我们有了这么个文件：

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("fib.pyx"),
)
```

使用和 `helloworld.pyx` 一样的命令来构建该拓展：

```
$ python setup.py build_ext --inplace
```

使用新的拓展：

```
>>> import fib
>>> fib.fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

质数（Primes）

本段给出一个小例子来展示一些我们可以做的事。这个例子给出一个用来寻找质数的程序。你告诉它你需要多少个质数，程序以 Python `list` 的形式将这些质数返回给你。

`primes.pyx`：

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

可以从上面的代码中看出，除了参数 `kmax` 是以 `int` 类型声明的外，代码和普通 Python 函数定义一样。这意味着传入 `kmax` 的对象将会转化成 C 语言的整数变量。（如果无法转化为 `int` 型，将会抛出 `TypeError` 异常）。

第 2、3 行使用了 `cdef` 来定义 C 语言的局部变量。第 4 行创建了一个用来返回结果的 Python `list` 。注意，代码的编写和 Python 代码的编写一模一样。因为结果变量还没给定类型，它只是用来储存 Python 对象。

第 7-9 行配置了一个循环用来测试候选数字是否是质数，一直到找到了足够多的质数为止。第 11-12 行用候选数字除以已经找到的质数，这两行很有意思. 因为没有涉及 Python 对象，所以循环将会完全翻译为 C 语言代码，所以运行非常快！

当一个质数被找到，第 14-15 行会将其添加到队列 `p` 中，以便在循环中检测质数时用来快速检索，第 16 行将其添加到结果队列中。第 16 行看起来也非常类似 Python 代码，它也的确是 Python 代码，在 `twist` 的作用下 C 语言定义的变量 `n` 在 `append` 方法调用前会自动转化为 Python 对象。最后，在第 18 行通过普通的 Python `return` 命令返回结果队列。

使用 Cython 编译器编译 `primes.pyx` 文件来生成一个拓展模块，我们可以在互动解释器（interactive interpreter）中来试用：

```
>>> import primes
>>> primes.primes(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

正常运行！如果你好奇 Cython 为你节约了多少资源，可以看看这个模块生成的 C 代码。

语言细节

想要获取更多地 Cython 语言信息，请看 Language Basics。 想在数字计算中运用 Cython 请看 Cython for NumPy Users。

C 函数的调用

这个教程简要讲述了如何使用 Cython 调用 C 库函数。 如果想了解更多关于 C 库函数调用的内容，请参考C 函数的调用。

简单来说，我们先以一个 C 标准库中的函数为例。 你不需要向你的代码中引入 额外的依赖，Cython 都已经帮你定义好了这些函数。所以你可以将这些函数直接 cimport 进来并使用。

举个例子，比如说当你想用最简单的方法将 `char*` 类型的值转化为一个整型值时， 你可以使用 `atoi()` 函数，这个函数是在 `stdlib.h` 头文件中定义的。我们可以这样来写：

```
from libc.stdlib cimport atoi

cdef parse_charptr_to_py_int(char* s):
    assert s is not NULL, "byte string value is NULL"
    return atoi(s)   # note: atoi() has no error detection!
```

你可以在 Cython 的源代码包 `Cython/Includes/ <https://github.com/cython/cython/tree/master/Cython/Includes>` _. 中找到所有的标准 cimport 文件。这些文件保存在 `.pxd` 文件中，这是一种标准再模块间共享 Cython 函数声明的方法(见:ref: `sharing-declarations`)。

Cython 也有一整套的 Cython 的C-API 函数声明集。 例如，为了测试你的 Cython 代码的 C 编译时间，你可以这样做：

```
from cpython.version cimport PY_VERSION_HEX

# Python version >= 3.2 final ?
print PY_VERSION_HEX >= 0x030200F0
```

Cython 也提供了 C math 库的声明：

```
from libc.math cimport sin

cdef double f(double x):
    return sin(x*x)
```

动态链接（Dynamic linking）

在一些类 Unix 系统（例如 linux）中，默认不提供libc math 库。 所以除了 cimport函数声明外，你还必须配置你的编译器以链接共享库 `m` 。 对于 distutils来说，在 `Extension()` 安装变量 `libraries` 中将其添加进来就可以了。

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

ext_modules=[
    Extension("demo",
              sources=["demo.pyx"],
              libraries=["m"] # Unix-like specific
    )
]

setup(
  name = "Demos",
  ext_modules = cythonize(ext_modules)
)
```

外部声明（External declarations）

如果你想调用一个 Cython 中没有定义的函数声明，那么你必须自己进行声明。例如，上文中的 `sin()` 函数就是这样定义的：

```
cdef extern from "math.h":
    double sin(double x)
```

此处声明了 `sin()` 函数，这时我们便可在 Cython 代码中使用这个函数，并且让 Cython 生成一份包括 `math.h` 头文件的 C 代码。C 编译器在编译时能够在 `math.h` 中找到原始的函数声明。但是 Cython 不能解析 `math.h` 并需要一个单独的定义。

正如math 库中的 `sin()` 函数一样，只要 Cython 生成的模块正确的链接了共享库或静态库，我们就可以声明并调用任意的 C 库函数。

注意，只要简单地通过 `cpdef` 声明，你就能从 Cython 模块中导出一个外部 C 函数。而且生成了一个 Python 扩展，使得我们可以在 Python 代码中直接访问到 C 函数 `sin()` ：

```
>>> sin(0)
0.0
```

```
cdef extern from "math.h":
    cpdef double sin(double x)
```

在属于 Cython 模块的 `.pxd` 文件中（一般与模块名一致，见:ref: `sharing-declaration` ）声明函数时， 你也可以达到同样的效果。 这使得其他 Cython 模块可以复用某个 C 声明。然而还是会在 Cython 模块中产生一份自动生成的 python 扩展。

变量的命名（Naming parameters）

C 和 Cython 都支持没有参数明的signature declarations：

```
cdef extern from "string.h":
    char* strstr(const char*, const char*)
```

然而，这样的话 Cython 代码将不能通过关键字参数来调用这个函数（由Cython 0.19及以后的版本所支持）。所以，我们最好这样去声明一个函数：

```
cdef extern from "string.h":
    char* strstr(const char *haystack, const char *needle)
```

这会让清楚地知道你所调用了哪两个参数，从而能够避免二义性并增强你的代码的可读性:

```
cdef char* data = "hfvcakdfagbcffvschvxcdfgccbcfhvgcsnfxjh"

pos = strstr(needle='akd', haystack=data)
print pos != NULL
```

注意，正如 Python 代码一样，对已有参数名的修改是不向后兼容的。那么， 如果你为外部的 C 或 C++ 函数进行了自己的声明，通常花一点时间去 将参数名命名的更好是非常值得的。

Using C libraries

Apart from writing fast code, one of the main use cases of Cython is to call external C libraries from Python code. As Cython code compiles down to C code itself, it is actually trivial to call C functions directly in the code. The following gives a complete example for using (and wrapping) an external C library in Cython code, including appropriate error handling and considerations about designing a suitable API for Python and Cython code.

Imagine you need an efficient way to store integer values in a FIFO queue. Since memory really matters, and the values are actually coming from C code, you cannot afford to create and store Python `int` objects in a list or deque. So you look out for a queue implementation in C.

After some web search, you find the C-algorithms library [CAlg]_ and decide to use its double ended queue implementation. To make the handling easier, however, you decide to wrap it in a Python extension type that can encapsulate all memory management.

.. [CAlg] Simon Howard, C Algorithms library, http://c-algorithms.sourceforge.net/

Defining external declarations

The C API of the queue implementation, which is defined in the header file `libcalg/queue.h` , essentially looks like this::

```
/* file: queue.h */

typedef struct _Queue Queue;
typedef void *QueueValue;

Queue *queue_new(void);
void queue_free(Queue *queue);

int queue_push_head(Queue *queue, QueueValue data);
QueueValue queue_pop_head(Queue *queue);
QueueValue queue_peek_head(Queue *queue);

int queue_push_tail(Queue *queue, QueueValue data);
QueueValue queue_pop_tail(Queue *queue);
QueueValue queue_peek_tail(Queue *queue);

int queue_is_empty(Queue *queue);
```

To get started, the first step is to redefine the C API in a `.pxd` file, say, `cqueue.pxd`::

```
# file: cqueue.pxd

cdef extern from "libcalg/queue.h":
    ctypedef struct Queue:
        pass
    ctypedef void* QueueValue

    Queue* queue_new()
    void queue_free(Queue* queue)

    int queue_push_head(Queue* queue, QueueValue data)
    QueueValue  queue_pop_head(Queue* queue)
    QueueValue queue_peek_head(Queue* queue)

    int queue_push_tail(Queue* queue, QueueValue data)
    QueueValue queue_pop_tail(Queue* queue)
    QueueValue queue_peek_tail(Queue* queue)

    bint queue_is_empty(Queue* queue)
```

Note how these declarations are almost identical to the header file declarations, so you can often just copy them over. However, you do not need to provide *all* declarations as above, just those that you use in your code or in other declarations, so that Cython gets to see a sufficient and consistent subset of them. Then, consider adapting them somewhat to make them more comfortable to work with in Cython.

Specifically, you should take care of choosing good argument names for the C functions, as Cython allows you to pass them as keyword arguments. Changing them later on is a backwards incompatible API modification. Choosing good names right away will make these functions more pleasant to work with from Cython code.

One noteworthy difference to the header file that we use above is the declaration of the `Queue` struct in the first line. `Queue` is in this case used as an *opaque handle*; only the library that is called knows what is really inside. Since no Cython code needs to know the contents of the struct, we do not need to declare its contents, so we simply provide an empty definition (as we do not want to declare the `_Queue` type which is referenced in the C header) [#]_.

.. [#] There's a subtle difference between `cdef struct Queue: pass` and `ctypedef struct Queue: pass`. The former declares a type which is referenced in C code as `struct Queue`, while the latter is referenced in C as `Queue`. This is a C language quirk that Cython is not able to hide. Most modern C libraries use the `ctypedef` kind of struct.

Another exception is the last line. The integer return value of the `queue_is_empty()` function is actually a C boolean value, i.e. the only interesting thing about it is whether it is non-zero or zero, indicating if the queue is empty or not. This is best expressed by Cython's `bint` type, which is a normal `int` type when used in C but maps to Python's boolean values `True` and `False` when converted to a Python object. This way of tightening declarations in a `.pxd` file can often simplify the code that uses them.

It is good practice to define one `.pxd` file for each library that you use, and sometimes even for each header file (or functional group) if the API is large. That simplifies their reuse in other projects. Sometimes, you may need to use C functions from the standard C library, or want to call C-API functions from CPython directly. For common needs like this, Cython ships with a set of standard `.pxd` files that provide these declarations in a readily usable way that is adapted to their use in Cython. The main packages are `cpython`, `libc` and `libcpp`. The NumPy library also has a standard `.pxd` file `numpy`, as it is often used in Cython code. See Cython's `Cython/Includes/` source package for a complete list of provided `.pxd` files.

Writing a wrapper class

After declaring our C library's API, we can start to design the Queue class that should wrap the C queue. It will live in a file called `queue.pyx`. [#]_

.. [#] Note that the name of the `.pyx` file must be different from the `cqueue.pxd` file with declarations from the C library, as both do not describe the same code. A `.pxd` file next to a `.pyx` file with the same name defines exported declarations for code in the `.pyx` file. As the `cqueue.pxd` file contains declarations of a regular C library, there must not be a `.pyx` file with the same name that Cython associates with it.

Here is a first start for the Queue class::

```
# file: queue.pyx

cimport cqueue

cdef class Queue:
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
```

Note that it says `__cinit__` rather than `__init__`. While `__init__` is available as well, it is not guaranteed to be run (for instance, one could create a subclass and forget to call the ancestor's constructor). Because not initializing C pointers often leads to hard crashes of the

Python interpreter, Cython provides `__cinit__` which is *always* called immediately on construction, before CPython even considers calling `__init__`, and which therefore is the right place to initialise `cdef` fields of the new instance. However, as `__cinit__` is called during object construction, `self` is not fully constructed yet, and one must avoid doing anything with `self` but assigning to `cdef` fields.

Note also that the above method takes no parameters, although subtypes may want to accept some. A no-arguments `__cinit__()` method is a special case here that simply does not receive any parameters that were passed to a constructor, so it does not prevent subclasses from adding parameters. If parameters are used in the signature of `__cinit__()`, they must match those of any declared `__init__` method of classes in the class hierarchy that are used to instantiate the type.

Memory management

Before we continue implementing the other methods, it is important to understand that the above implementation is not safe. In case anything goes wrong in the call to `queue_new()`, this code will simply swallow the error, so we will likely run into a crash later on. According to the documentation of the `queue_new()` function, the only reason why the above can fail is due to insufficient memory. In that case, it will return `NULL`, whereas it would normally return a pointer to the new queue.

The Python way to get out of this is to raise a `MemoryError` [#]_. We can thus change the init function as follows::

```
cimport cqueue

cdef class Queue:
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
        if self._c_queue is NULL:
            raise MemoryError()
```

.. [#] In the specific case of a `MemoryError`, creating a new exception instance in order to raise it may actually fail because we are running out of memory. Luckily, CPython provides a C-API function `PyErr_NoMemory()` that safely raises the right exception for us. Since version 0.14.1, Cython automatically substitutes this C-API call whenever you write `raise MemoryError` or `raise MemoryError()`. If you use an older version, you have to cimport the C-API function from the standard package `cpython.exc` and call it directly.

The next thing to do is to clean up when the Queue instance is no longer used (i.e. all references to it have been deleted). To this end, CPython provides a callback that Cython makes available as a special method `__dealloc__()`. In our case, all we have to do is to free the C Queue, but only if we succeeded in initialising it in the init method::

```
    def __dealloc__(self):
        if self._c_queue is not NULL:
            cqueue.queue_free(self._c_queue)
```

Compiling and linking

At this point, we have a working Cython module that we can test. To compile it, we need to configure a `setup.py` script for distutils. Here is the most basic script for compiling a Cython module::

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

setup(
    ext_modules = cythonize([Extension("queue", ["queue.pyx"])])
)
```

To build against the external C library, we must extend this script to include the necessary setup. Assuming the library is installed in the usual places (e.g. under `/usr/lib` and `/usr/include` on a Unix-like system), we could simply change the extension setup from

::

```
ext_modules = cythonize([Extension("queue", ["queue.pyx"])])
```

to

::

```
ext_modules = cythonize([
    Extension("queue", ["queue.pyx"],
              libraries=["calg"])
    ])
```

If it is not installed in a 'normal' location, users can provide the required parameters externally by passing appropriate C compiler flags, such as::

```
CFLAGS="-I/usr/local/otherdir/calg/include"  \
LDFLAGS="-L/usr/local/otherdir/calg/lib"     \
    python setup.py build_ext -i
```

Once we have compiled the module for the first time, we can now import it and instantiate a new Queue::

```
$ export PYTHONPATH=.
$ python -c 'import queue.Queue as Q ; Q()'
```

However, this is all our Queue class can do so far, so let's make it more usable.

Mapping functionality

Before implementing the public interface of this class, it is good practice to look at what interfaces Python offers, e.g. in its `list` or `collections.deque` classes. Since we only need a FIFO queue, it's enough to provide the methods `append()`, `peek()` and `pop()`, and additionally an `extend()` method to add multiple values at once. Also, since we already know that all values will be coming from C, it's best to provide only `cdef` methods for now, and to give them a straight C interface.

In C, it is common for data structures to store data as a `void*` to whatever data item type. Since we only want to store `int` values, which usually fit into the size of a pointer type, we can avoid additional memory allocations through a trick: we cast our `int` values to `void*` and vice versa, and store the value directly as the pointer value.

Here is a simple implementation for the `append()` method::

```
    cdef append(self, int value):
        cqueue.queue_push_tail(self._c_queue, <void*>value)
```

Again, the same error handling considerations as for the `__cinit__()` method apply, so that we end up with this implementation instead::

```
    cdef append(self, int value):
        if not cqueue.queue_push_tail(self._c_queue,
                                      <void*>value):
            raise MemoryError()
```

Adding an `extend()` method should now be straight forward::

```
cdef extend(self, int* values, size_t count):
    """Append all ints to the queue.
    """
    cdef size_t i
    for i in range(count):
        if not cqueue.queue_push_tail(
                self._c_queue, <void*>values[i]):
            raise MemoryError()
```

This becomes handy when reading values from a NumPy array, for example.

So far, we can only add data to the queue. The next step is to write the two methods to get the first element: `peek()` and `pop()`, which provide read-only and destructive read access respectively::

```
cdef int peek(self):
    return <int>cqueue.queue_peek_head(self._c_queue)

cdef int pop(self):
    return <int>cqueue.queue_pop_head(self._c_queue)
```

Handling errors

Now, what happens when the queue is empty? According to the documentation, the functions return a `NULL` pointer, which is typically not a valid value. Since we are simply casting to and from ints, we cannot distinguish anymore if the return value was `NULL` because the queue was empty or because the value stored in the queue was `0`. However, in Cython code, we would expect the first case to raise an exception, whereas the second case should simply return `0`. To deal with this, we need to special case this value, and check if the queue really is empty or not::

```
cdef int peek(self) except? -1:
    value = <int>cqueue.queue_peek_head(self._c_queue)
    if value == 0:
        # this may mean that the queue is empty, or
        # that it happens to contain a 0 value
        if cqueue.queue_is_empty(self._c_queue):
            raise IndexError("Queue is empty")
    return value
```

Note how we have effectively created a fast path through the method in the hopefully common cases that the return value is not `0`. Only that specific case needs an additional check if the queue is empty.

The `except? -1` declaration in the method signature falls into the same category. If the function was a Python function returning a Python object value, CPython would simply return `NULL` internally instead of a Python object to indicate an exception, which would immediately be propagated by the surrounding code. The problem is that the return type is `int` and any `int` value is a valid queue item value, so there is no way to explicitly signal an error to the calling code. In fact, without such a declaration, there is no obvious way for Cython to know what to return on exceptions and for calling code to even know that this method *may* exit with an exception.

The only way calling code can deal with this situation is to call `PyErr_Occurred()` when returning from a function to check if an exception was raised, and if so, propagate the exception. This obviously has a performance penalty. Cython therefore allows you to declare which value it should implicitly return in the case of an exception, so that the surrounding code only needs to check for an exception when receiving this exact value.

We chose to use `-1` as the exception return value as we expect it to be an unlikely value to be put into the queue. The question mark in the `except? -1` declaration indicates that the return value is ambiguous (there *may* be a `-1` value in the queue, after all) and that an additional exception check using `PyErr_Occurred()` is needed in calling code. Without it, Cython code that calls this method and receives the exception return value would silently (and sometimes incorrectly) assume that an exception has been raised. In any case, all other return values will be passed through almost without a penalty, thus again creating a fast path for 'normal' values.

Now that the `peek()` method is implemented, the `pop()` method also needs adaptation. Since it removes a value from the queue, however, it is not enough to test if the queue is empty *after* the removal. Instead, we must test it on entry::

```
cdef int pop(self) except? -1:
    if cqueue.queue_is_empty(self._c_queue):
        raise IndexError("Queue is empty")
    return <int>cqueue.queue_pop_head(self._c_queue)
```

The return value for exception propagation is declared exactly as for `peek()` .

Lastly, we can provide the Queue with an emptiness indicator in the normal Python way by implementing the `__bool__()` special method (note that Python 2 calls this method `__nonzero__` , whereas Cython code can use either name)::

```
def __bool__(self):
    return not cqueue.queue_is_empty(self._c_queue)
```

Note that this method returns either `True` or `False` as we declared the return type of the `queue_is_empty()` function as `bint` in `cqueue.pxd` .

Testing the result

Now that the implementation is complete, you may want to write some tests for it to make sure it works correctly. Especially doctests are very nice for this purpose, as they provide some documentation at the same time. To enable doctests, however, you need a Python API that you can call. C methods are not visible from Python code, and thus not callable from doctests.

A quick way to provide a Python API for the class is to change the methods from `cdef` to `cpdef` . This will let Cython generate two entry points, one that is callable from normal Python code using the Python call semantics and Python objects as arguments, and one that is callable from C code with fast C semantics and without requiring intermediate argument conversion from or to Python types. Note that `cpdef` methods ensure that they can be appropriately overridden by Python methods even when they are called from Cython. This adds a tiny overhead compared to `cdef` methods.

The following listing shows the complete implementation that uses `cpdef` methods where possible::

```
cimport cqueue

cdef class Queue:
    """A queue class for C integer values.

    >>> q = Queue()
    >>> q.append(5)
    >>> q.peek()
    5
    >>> q.pop()
    5
    """
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
        if self._c_queue is NULL:
            raise MemoryError()

    def __dealloc__(self):
        if self._c_queue is not NULL:
            cqueue.queue_free(self._c_queue)

    cpdef append(self, int value):
        if not cqueue.queue_push_tail(self._c_queue,
                                      <void*>value):
            raise MemoryError()

    cdef extend(self, int* values, size_t count):
        cdef size_t i
        for i in xrange(count):
            if not cqueue.queue_push_tail(
                    self._c_queue, <void*>values[i]):
                raise MemoryError()

    cpdef int peek(self) except? -1:
        cdef int value = \
            <int>cqueue.queue_peek_head(self._c_queue)
        if value == 0:
            # this may mean that the queue is empty,
            # or that it happens to contain a 0 value
            if cqueue.queue_is_empty(self._c_queue):
                raise IndexError("Queue is empty")
        return value

    cpdef int pop(self) except? -1:
        if cqueue.queue_is_empty(self._c_queue):
            raise IndexError("Queue is empty")
        return <int>cqueue.queue_pop_head(self._c_queue)

    def __bool__(self):
        return not cqueue.queue_is_empty(self._c_queue)
```

The `cpdef` feature is obviously not available for the `extend()` method, as the method signature is incompatible with Python argument types. However, if wanted, we can rename the C-ish `extend()` method to e.g. `c_extend()`, and write a new `extend()` method instead that accepts an arbitrary Python iterable::

```
    cdef c_extend(self, int* values, size_t count):
        cdef size_t i
        for i in range(count):
            if not cqueue.queue_push_tail(
                    self._c_queue, <void*>values[i]):
                raise MemoryError()

    cpdef extend(self, values):
        for value in values:
            self.append(value)
```

As a quick test with 10000 numbers on the author's machine indicates, using this Queue from Cython code with C `int` values is about five times as fast as using it from Cython code with Python object values, almost eight times faster than using it from Python code in a Python loop, and still more than twice as fast as using Python's highly optimised `collections.deque` type from Cython code with Python integers.

Callbacks

Let's say you want to provide a way for users to pop values from the queue up to a certain user defined event occurs. To this end, you want to allow them to pass a predicate function that determines when to stop, e.g.::

```
def pop_until(self, predicate):
    while not predicate(self.peek()):
        self.pop()
```

Now, let us assume for the sake of argument that the C queue provides such a function that takes a C callback function as predicate. The API could look as follows::

```
/* C type of a predicate function that takes a queue value and returns
 * -1 for errors
 *  0 for reject
 *  1 for accept
 */
typedef int (*predicate_func)(void* user_context, QueueValue data);

/* Pop values as long as the predicate evaluates to true for them,
 * returns -1 if the predicate failed with an error and 0 otherwise.
 */
int queue_pop_head_until(Queue *queue, predicate_func predicate,
                         void* user_context);
```

It is normal for C callback functions to have a generic :c:type: `void*` argument that allows passing any kind of context or state through the C-API into the callback function. We will use this to pass our Python predicate function.

First, we have to define a callback function with the expected signature that we can pass into the C-API function::

```
cdef int evaluate_predicate(void* context, cqueue.QueueValue value):
    "Callback function that can be passed as predicate_func"
    try:
        # recover Python function object from void* argument
        func = <object>context
        # call function, convert result into 0/1 for True/False
        return bool(func(<int>value))
    except:
        # catch any Python errors and return error indicator
        return -1
```

The main idea is to pass a pointer (a.k.a. borrowed reference) to the function object as the user context argument. We will call the C-API function as follows::

```
def pop_until(self, python_predicate_function):
    result = cqueue.queue_pop_head_until(
        self._c_queue, evaluate_predicate,
        <void*>python_predicate_function)
    if result == -1:
        raise RuntimeError("an error occurred")
```

The usual pattern is to first cast the Python object reference into a :c:type: `void*` to pass it into the C-API function, and then cast it back into a Python object in the C predicate callback function. The cast to :c:type: `void*` creates a borrowed reference. On the cast to `<object>`, Cython increments the reference count of the object and thus converts the borrowed reference back into an owned reference. At the end of the predicate function, the owned reference goes out of scope again and Cython discards it.

The error handling in the code above is a bit simplistic. Specifically, any exceptions that the predicate function raises will essentially be discarded and only result in a plain `RuntimeError()` being raised after the fact. This can be improved by storing away the exception in an object passed through the context parameter and re-raising it after the C-API function has returned `-1` to indicate the error.