# Cython tutorial

Stefan Behnel (stefan_ml@behnel.de) – *Senacor Technologies AG*, Germany
Robert W. Bradshaw (robertwb@math.washington.edu) – *University of Washington*[a], USA
Dag Sverre Seljebotn (dagss@student.matnat.uio.no) – *University of Oslo*[b c d], Norway

[a]Department of Mathematics, University of Washington, Seattle, WA, USA
[b]Institute of Theoretical Astrophysics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway
[c]Department of Mathematics, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway
[d]Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway

**Cython is a programming language based on Python with extra syntax to provide static type declarations. This takes advantage of the benefits of Python while allowing one to achieve the speed of C. In this paper we describe the Cython language and show how it can be used both to write optimized code and to interface with external C libraries.**

## Cython - an overview

[Cython] is a programming language based on Python, with extra syntax allowing for optional static type declarations. It aims to become a superset of the [Python] language which gives it high-level, object-oriented, functional, and dynamic programming. The source code gets translated into optimized C/C++ code and compiled as Python extension modules. This allows for both very fast program execution and tight integration with external C libraries, while keeping up the high programmer productivity for which the Python language is well known.

The primary Python execution environment is commonly referred to as CPython, as it is written in C. Other major implementations use Java (Jython [Jython]), C# (IronPython [IronPython]) and Python itself (PyPy [PyPy]). Written in C, CPython has been conducive to wrapping many external libraries that interface through the C language. It has, however, remained non trivial to write the necessary glue code in C, especially for programmers who are more fluent in a high-level language like Python than in a do-it-yourself language like C.

Originally based on the well-known Pyrex [Pyrex], the Cython project has approached this problem by means of a source code compiler that translates Python code to equivalent C code. This code is executed within the CPython runtime environment, but at the speed of compiled C and with the ability to call directly into C libraries. At the same time, it keeps the original interface of the Python source code, which makes it directly usable from Python code. These two-fold characteristics enable Cython's two major use cases: extending the CPython interpreter with fast binary modules, and interfacing Python code with external C libraries.

While Cython can compile (most) regular Python code, the generated C code usually gains major (and sometime impressive) speed improvements from optional static type declarations for both Python and C types. These allow Cython to assign C semantics to parts of the code, and to translate them into very efficient C code. Type declarations can therefore be used for two purposes: for moving code sections from dynamic Python semantics into static-and-fast C semantics, but also for directly manipulating types defined in external libraries. Cython thus merges the two worlds into a very broadly applicable programming language.

## Installing Cython

Many scientific Python distributions, such as the Enthought Python Distribution [EPD], Python(x,y) [Pythonxy], and Sage [Sage], bundle Cython and no setup is needed. Note however that if your distribution ships a version of Cython which is too old you can still use the instructions below to update Cython. Everything in this tutorial should work with Cython 0.11.2 and newer, unless a footnote says otherwise.

Unlike most Python software, Cython requires a C compiler to be present on the system. The details of getting a C compiler varies according to the system used:

- **Linux** The GNU C Compiler (gcc) is usually present, or easily available through the package system. On Ubuntu or Debian, for instance, the command `sudo apt-get install build-essential` will fetch everything you need.
- **Mac OS X** To retrieve gcc, one option is to install Apple's XCode, which can be retrieved from the Mac OS X's install DVDs or from http://developer.apple.com.
- **Windows** A popular option is to use the open source MinGW (a Windows distribution of gcc). See the appendix for instructions for setting up MinGW manually. EPD and Python(x,y) bundle MinGW, but some of the configuration steps in the appendix might still be necessary. Another option is to use Microsoft's Visual C. One must then use the same version which the installed Python was compiled with.

The newest Cython release can always be downloaded from http://cython.org. Unpack the tarball or zip file, enter the directory, and then run:

```
python setup.py install
```

If you have Python setuptools set up on your system, you should be able to fetch Cython from PyPI and install it using:

```
easy_install cython
```

For Windows there is also an executable installer available for download.

## Building Cython code

Cython code must, unlike Python, be compiled. This happens in two stages:

- A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module
- The `.c` file is compiled by a C compiler to a `.so` file (or `.pyd` on Windows) which can be `import`-ed directly into a Python session.

There are several ways to build Cython code:

- Write a distutils `setup.py`.
- Use `pyximport`, importing Cython `.pyx` files as if they were `.py` files (using distutils to compile and build the background).
- Run the `cython` command-line utility manually to produce the `.c` file from the `.pyx` file, then manually compiling the `.c` file into a shared object library or `.dll` suitable for import from Python. (This is mostly for debugging and experimentation.)
- Use the [Sage] notebook which allows Cython code inline and makes it easy to experiment with Cython code without worrying about compilation details (see figure 1 below).

Currently, distutils is the most common way Cython files are built and distributed.

### Building a Cython module using distutils

Imagine a simple "hello world" script in a file `hello.pyx`:

```
def say_hello_to(name):
    print(Hello %s!" % name)
```

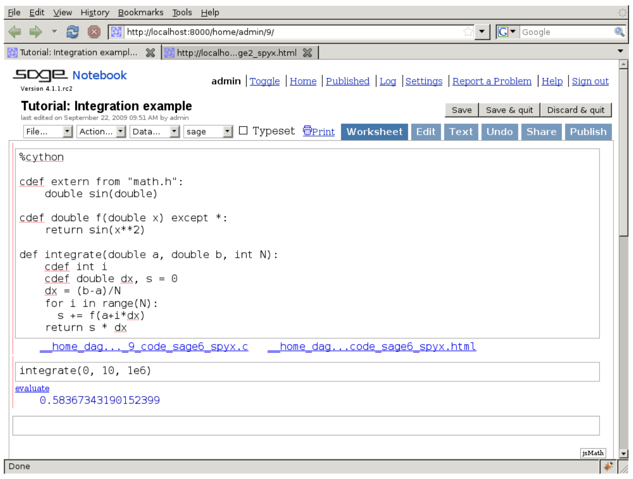
The following could be a corresponding `setup.py` script:

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

ext_modules = [Extension("hello", ["hello.pyx"])]

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

To build, run `python setup.py build_ext --inplace`. Then simply start a Python session and do `from hello import say_hello_to` and use the imported function as you see fit.


File Edit View History Bookmarks Tools Help
http://localhost:8000/home/admin/9/
Google
Tutorial: Integration exampl...
http://localho...ge2_spyx.html
sage Notebook
Version 4.1.1.rc2
admin | Toggle | Home | Published | Log | Settings | Report a Problem | Help | Sign out
Tutorial: Integration example
last edited on September 22, 2009 09:51 AM by admin
Save | Save & quit | Discard & quit
File... | Action... | Data... | sage | Typeset | Print
Worksheet | Edit | Text | Undo | Share | Publish
evaluate
jsMath
Done

```
%cython

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

cdef double f(double x) except *:
    return sin(x**2)

def integrate(double a, double b, int N):
    cdef int i
    cdef double dx, s = 0
    dx = (b-a)/N
    for i in range(N):
      s += f(a+i*dx)
    return s * dx
```

__home_dag..._9_code_sage6_spyx.c    __home_dag...code_sage6_spyx.html

```
integrate(0, 10, 1e6)
```

```
0.58367343190152399
```

***Figure 1*** *The Sage notebook allows transparently editing and compiling Cython code simply by typing `%cython` at the top of a cell and evaluate it. Variables and functions defined in a Cython cell imported into the running session.*

## Data types in Cython

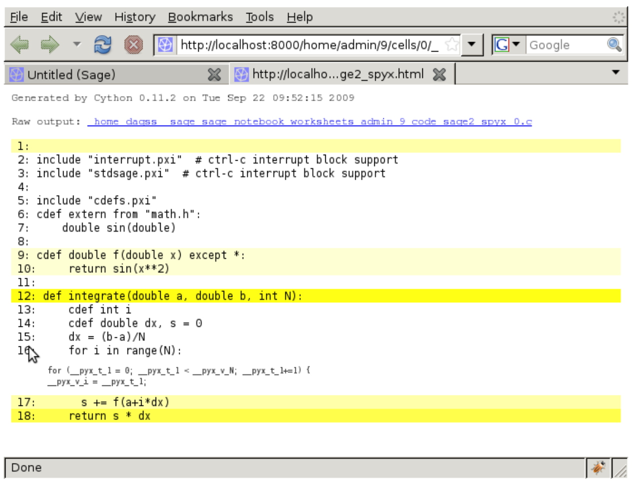
Cython is a Python compiler. This means that it can compile normal Python code without changes (with a few obvious exceptions of some as-yet unsupported language features). However, for performance-critical code, it is often helpful to add static type declarations, as they will allow Cython to step out of the dynamic nature of the Python code and generate simpler and faster C code - sometimes faster by orders of magnitude.

It must be noted, however, that type declarations can make the source code more verbose and thus less readable. It is therefore discouraged to use them without good reason, such as where benchmarks prove that they really make the code substantially faster in a performance critical section. Typically a few types in the right spots go a long way. Cython can produce annotated output (see figure 2 below) that can be very useful in determining where to add types.

All C types are available for type declarations: integer and floating point types, complex numbers, structs, unions and pointer types. Cython can automatically and correctly convert between the types on assignment. This also includes Python's arbitrary size integer types, where value overflows on conversion to a C type will raise a Python `OverflowError` at runtime. The generated C code will handle the platform dependent sizes of C types correctly and safely in this case.

## Faster code by adding types

Consider the following pure Python code:


File Edit View History Bookmarks Tools Help
http://localhost:8000/home/admin/9/cells/0/_
Google
Untitled (Sage)
http://localho...ge2_spyx.html
Done

```
Generated by Cython 0.11.2 on Tue Sep 22 09:52:15 2009

Raw output: _home_dagss__sage_sage_notebook_worksheets_admin_9_code_sage2_spyx_0.c


include "interrupt.pxi"  # ctrl-c interrupt block support
include "stdsage.pxi"  # ctrl-c interrupt block support

include "cdefs.pxi"
cdef extern from "math.h":
    double sin(double)

cdef double f(double x) except *:
    return sin(x**2)

def integrate(double a, double b, int N):
    cdef int i
    cdef double dx, s = 0
    dx = (b-a)/N
    for i in range(N):
      for (__pyx_t_1 = 0; __pyx_t_1 < __pyx_v_N; __pyx_t_1+=1) {
      __pyx_v_i = __pyx_t_1;
        s += f(a+i*dx)
    return s * dx
```

***Figure 2*** *Using the* `-a` *switch to the* `cython` *command line program (or following a link from the Sage notebook) results in an HTML report of Cython code interleaved with the generated C code. Lines are colored according to the level of "typedness" - white lines translates to pure C without any Python API calls. This report is invaluable when optimizing a function for speed.*

```
from math import sin

def f(x):
    return sin(x**2)

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Simply compiling this in Cython merely gives a 5% speedup. This is better than nothing, but adding some static types can make a much larger difference.

With additional type declarations, this might look like:

```
from math import sin

def f(double x):
    return sin(x**2)

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Since the iterator variable `i` is typed with C semantics, the for-loop will be compiled to pure C code. Typing `a`, `s` and `dx` is important as they are involved in arithmetic withing the for-loop; typing `b` and `N` makes less of a difference, but in this case it is not much extra work to be consistent and type the entire function.

This results in a 24 times speedup over the pure Python version.

## cdef functions

Python function calls can be expensive, and this is especially true in Cython because one might need to convert to and from Python objects to do the call. In our example above, the argument is assumed to be a C double both inside f() and in the call to it, yet a Python `float` object must be constructed around the argument in order to pass it.

Therefore Cython provides a syntax for declaring a C-style function, the cdef keyword:

```
cdef double f(double) except *:
    return sin(x**2)
```

Some form of except-modifier should usually be added, otherwise Cython will not be able to propagate exceptions raised in the function (or a function it calls). Above `except *` is used which is always safe. An except clause can be left out if the function returns a Python object or if it is guaranteed that an exception will not be raised within the function call.

A side-effect of cdef is that the function is no longer available from Python-space, as Python wouldn't know how to call it. Using the `cpdef` keyword instead of cdef, a Python wrapper is also created, so that the function is available both from Cython (fast, passing typed values directly) and from Python (wrapping values in Python objects).

Note also that it is no longer possible to change `f` at runtime.

Speedup: 45 times over pure Python.

## Calling external C functions

It is perfectly OK to do `from math import sin` to use Python's `sin()` function. However, calling C's own `sin()` function is substantially faster, especially in tight loops. It can be declared and used in Cython as follows:

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

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

At this point there are no longer any Python wrapper objects around our values inside of the main for loop, and so we get an impressive speedup to 219 times the speed of Python.

Note that the above code re-declares the function from `math.h` to make it available to Cython code. The C compiler will see the original declaration in `math.h` at compile time, but Cython does not parse "math.h" and requires a separate definition.

When calling C functions, one must take care to link in the appropriate libraries. This can be platform-specific; the below example works on Linux and Mac OS X:

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

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

setup(
  name = "Demos",
  cmdclass = {"build_ext": build_ext},
  ext_modules = ext_modules
)
```

If one uses the Sage notebook to compile Cython code, one can use a special comment to tell Sage to link in libraries:

```
#clib: m
```

Just like the `sin()` function from the math library, it is possible to declare and call into any C library as long as the module that Cython generates is properly linked against the shared or static library. A more extensive example of wrapping a C library is given in the section Using C libraries.

## Extension types (aka. cdef classes)

To support object-oriented programming, Cython supports writing normal Python classes exactly as in Python:

```
class MathFunction(object):
    def __init__(self, name, operator):
        self.name = name
        self.operator = operator

    def __call__(self, *operands):
        return self.operator(*operands)
```

Based on what Python calls a "built-in type", however, Cython supports a second kind of class: *extension types*, sometimes referred to as "cdef classes" due to the keywords used for their declaration. They are somewhat restricted compared to Python classes, but are generally more memory efficient and faster than generic Python classes. The main difference is that they use a C struct to store their fields and methods instead of a Python dict. This allows them to store arbitrary C types in their fields without requiring a Python wrapper for them, and to access fields and methods directly at the C level without passing through a Python dictionary lookup.
Normal Python classes can inherit from cdef classes, but not the other way around. Cython requires to know the complete inheritance hierarchy in order to lay out their C structs, and restricts it to single inheritance. Normal Python classes, on the other hand, can inherit from any number of Python classes and extension types, both in Cython code and pure Python code.
So far our integration example has not been very useful as it only integrates a single hard-coded function. In order to remedy this, without sacrificing speed, we will use a cdef class to represent a function on floating point numbers:

```
cdef class Function:
    cpdef double evaluate(self, double x) except *:
        return 0
```

Like before, cpdef makes two versions of the method available; one fast for use from Cython and one slower for use from Python. Then:

```
cdef class SinOfSquareFunction(Function):
    cpdef double evaluate(self, double x) except *:
        return sin(x**2)
```

Using this, we can now change our integration example:

```
def integrate(Function f, double a, double b, int N):
  cdef int i
  cdef double s, dx
  if f is None:
      raise ValueError("f cannot be None")
  s = 0
  dx = (b-a)/N
  for i in range(N):
      s += f.evaluate(a+i*dx)
  return s * dx

print(integrate(SinOfSquareFunction(), 0, 1, 10000))
```

This is almost as fast as the previous code, however it is much more flexible as the function to integrate can be changed. It is even possible to pass in a new function defined in Python-space. Assuming the above code is in the module *integrate.pyx*, we can do:

```
>>> import integrate
>>> class MyPolynomial(integrate.Function):
...     def evaluate(self, x):
...         return 2*x*x + 3*x - 10
...
>>> integrate.integrate(MyPolynomial(), 0, 1, 10000)
-7.8335833300000077
```

This is about 20 times slower than *SinOfSquareFunction*, but still about 10 times faster than the original Python-only integration code. This shows how large the speed-ups can easily be when whole loops are moved from Python code into a Cython module.
Some notes on our new implementation of `evaluate`:

- The fast method dispatch here only works because `evaluate` was declared in `Function`. Had `evaluate` been introduced in `SinOfSquareFunction`, the code would still work, but Cython would have used the slower Python method dispatch mechanism instead.
- In the same way, had the argument `f` not been typed, but only been passed as a Python object, the slower Python dispatch would be used.
- Since the argument is typed, we need to check whether it is `None`. In Python, this would have resulted in an `AttributeError` when the `evaluate` method was looked up, but Cython would instead try to access the (incompatible) internal structure of `None` as if it were a `Function`, leading to a crash or data corruption.

There is a *compiler directive* `nonecheck` which turns on checks for this, at the cost of decreased speed. Here's how compiler directives are used to dynamically switch on or off `nonecheck`:

```
#cython: nonecheck=True
#        ^^^ Turns on nonecheck globally

import cython

# Turn off nonecheck locally for the function
@cython.nonecheck(False)
def func():
    cdef MyClass obj = None
    try:
        # Turn nonecheck on again for a block
        with cython.nonecheck(True):
            print obj.myfunc() # Raises exception
    except AttributeError:
        pass
    print obj.myfunc() # Hope for a crash!
```

Attributes in cdef classes behave differently from attributes in regular classes:

- All attributes must be pre-declared at compile-time
- Attributes are by default only accessible from Cython (typed access)
- Properties can be declared to expose dynamic attributes to Python-space

```
cdef class WaveFunction(Function):
    # Not available in Python-space:
    cdef double offset
    # Available in Python-space:
    cdef public double freq
    # Available in Python-space:
    property period:
        def __get__(self):
            return 1.0 / self. freq
        def __set__(self, value):
            self. freq = 1.0 / value
    <...>
```

## pxd files

In addition to the `.pyx` source files, Cython uses `.pxd` files which work like C header files - they contain Cython declarations (and sometimes code sections) which are only meant for sharing C-level declarations with other Cython modules. A `pxd` file is imported into a `pyx` module by using the `cimport` keyword.

`pxd` files have many use-cases:

1. They can be used for sharing external C declarations.
2. They can contain functions which are well suited for inlining by the C compiler. Such functions should be marked `inline`, example:

   ```
   cdef inline int int_min(int a, int b):
       return b if b < a else a
   ```

3. When accompanying an equally named `pyx` file, they provide a Cython interface to the Cython module so that other Cython modules can communicate with it using a more efficient protocol than the Python one.

In our integration example, we might break it up into `pxd` files like this:

1. Add a `cmath.pxd` function which defines the C functions available from the C `math.h` header file, like `sin`. Then one would simply do `from cmath import sin` in `integrate.pyx`.
2. Add a `integrate.pxd` so that other modules written in Cython can define fast custom functions to integrate.

   ```
   cdef class Function:
       cpdef evaluate(self, double x)
   cpdef integrate(Function f, double a,
                   double b, int N)
   ```

   Note that if you have a cdef class with attributes, the attributes must be declared in the class declaration `pxd` file (if you use one), not the `pyx` file. The compiler will tell you about this.

## Using Cython with NumPy

Cython has support for fast access to NumPy arrays. To optimize code using such arrays one must `cimport` the NumPy pxd file (which ships with Cython), and declare any arrays as having the `ndarray` type. The data type and number of dimensions should be fixed at compile-time and passed. For instance:

```
import numpy as np
cimport numpy as np
def myfunc(np.ndarray[np.float64_t, ndim=2] A):
    <...>
```

`myfunc` can now only be passed two-dimensional arrays containing double precision floats, but array indexing operation is much, much faster, making it suitable for numerical loops. Expect speed increases well over 100 times over a pure Python loop; in some cases the speed increase can be as high as 700 times or more. [Seljebotn09] contains detailed examples and benchmarks.

Fast array declarations can currently only be used with function local variables and arguments to `def`-style functions (not with arguments to `cpdef` or `cdef`, and neither with fields in cdef classes or as global variables). These limitations are considered known defects and we hope to remove them eventually. In most circumstances it is possible to work around these limitations rather easily and without a significant speed penalty, as all NumPy arrays can also be passed as untyped objects.

Array indexing is only optimized if exactly as many indices are provided as the number of array dimensions. Furthermore, all indices must have a native integer type. Slices and NumPy "fancy indexing" is not optimized. Examples:

```
def myfunc(np.ndarray[np.float64_t, ndim=1] A):
    cdef Py_ssize_t i, j
    for i in range(A.shape[0]):
        print A[i, 0] # fast
        j = 2*i
        print A[i, j] # fast
        k = 2*i
        print A[i, k] # slow, k is not typed
        print A[i][j] # slow
        print A[i,:]  # slow
```

`Py_ssize_t` is a signed integer type provided by Python which covers the same range of values as is supported as NumPy array indices. It is the preferred type to use for loops over arrays.

Any Cython primitive type (float, complex float and integer types) can be passed as the array data type. For each valid dtype in the `numpy` module (such as `np.uint8`, `np.complex128`) there is a corresponding Cython compile-time definition in the cimport-ed NumPy pxd file with a `_t` suffix[1]. Cython structs are also allowed and corresponds to NumPy record arrays. Examples:

```
cdef packed struct Point:
    np.float64_t x, y

def f():
    cdef np.ndarray[np.complex128_t, ndim=3] a = \
        np.zeros((3,3,3), dtype=np.complex128)
    cdef np.ndarray[Point] b = np.zeros(10,
        dtype=np.dtype([('x', np.float64),
                        ('y', np.float64)]))
    <...>
```

Note that `ndim` defaults to 1. Also note that NumPy record arrays are by default unaligned, meaning data is packed as tightly as possible without considering the alignment preferences of the CPU. Such unaligned record arrays corresponds to a Cython `packed` struct. If one uses an aligned dtype, by passing `align=True` to the `dtype` constructor, one must drop the `packed` keyword on the struct definition.

Some data types are not yet supported, like boolean arrays and string arrays. Also data types describing data which is not in the native endian will likely never be supported. It is however possible to access such arrays on a lower level by casting the arrays:

```
cdef np.ndarray[np.uint8, cast=True] boolarr = (x < y)
cdef np.ndarray[np.uint32, cast=True] values = \
    np.arange(10, dtype='>i4')
```

Assuming one is on a little-endian system, the `values` array can still access the raw bit content of the array (which must then be reinterpreted to yield valid results on a little-endian system).

Finally, note that typed NumPy array variables in some respects behave a little differently from untyped arrays. `arr.shape` is no longer a tuple. `arr.shape[0]` is valid but to e.g. print the shape one must do `print (<object>arr).shape` in order to “untype” the variable first. The same is true for `arr.data` (which in typed mode is a C data pointer).

There are many more options for optimizations to consider for Cython and NumPy arrays. We again refer the interested reader to [Seljebotn09].

[1] In Cython 0.11.2, `np.complex64_t` and `np.complex128_t` does not work and one must write `complex` or `double complex` instead. This is fixed in 0.11.3. Cython 0.11.1 and earlier does not support complex numbers.

## 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 seen for the C string decoding functions above, it is actually trivial to call C functions directly in the code. The following describes what needs to be done to use an external C library in 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.

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

```
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`:

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

    Queue* new_queue()
    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. One exception is the last line. The return value of the `queue_is_empty` method is actually a C boolean value, i.e. it is either zero or non-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. Another difference is the first line. `Queue` is in this case used as an *opaque handle*; only the library that is called know what is actually 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)[2].

Next, we need to design the Queue class that should wrap the C queue. Here is a first start for the Queue class:

```
cimport cqueue
cimport python_exc

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

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 constructor). Because not initializing C pointers often leads to crashing the Python interpreter without leaving as much as a stack trace, Cython provides `__cinit__` which is *always* called on construction. 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. Although it is guaranteed to get called, the no-arguments `__cinit__()` method is a special case here as it does not prevent subclasses from adding parameters as they see fit. If parameters are added they must match those of any declared `__init__` method.

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 `new_queue()`, this code will simply swallow the error, so we will likely run into a crash later on. According to the documentation of the `new_queue()` 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 normal way to get out of this is to raise an exception, but allocating a new exception instance may actually fail when we are running out of memory. Luckily, CPython provides a function `PyErr_NoMemory()` that raises the right exception for us. We can thus change the init function as follows:

```
def __cinit__(self):
    self._c_queue = cqueue.new_queue()
    if self._c_queue is NULL:
        python_exc.PyErr_NoMemory()
```

The next thing to do is to clean up when the Queue is no longer used. 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 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)
```

At this point, we have a compilable Cython module that we can test. To compile it, we need to configure a `setup.py` script for distutils. Based on the example presented earlier on, we can extend the script to include the necessary setup for building against the external C library. Assuming it's installed in the normal places (e.g. under `/usr/lib` and `/usr/include` on a Unix-like system), we could simply change the extension setup from

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

to

```
ext_modules = [
    Extension("hello", ["hello.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 try to import it:

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

However, our class doesn't do much yet so far, so let's make it more usable.

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 better 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.

[2]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.

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:

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

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

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

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)
```

Simple enough. 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? 0:
    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
```

The except? 0 declaration is worth explaining. If the function was a Python function returning a Python object value, CPython would simply return NULL instead of a Python object to indicate a raised exception, which would immediately be propagated by the surrounding code. The problem is that any int value is a valid queue item value, so there is no way to explicitly indicate an error to the calling code.

The only way CPython (and Cython) 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 indicate which value is explicitly returned in the case of an exception, so that the surrounding code only needs to check for an exception when receiving this special value. All other values will be accepted almost without a penalty.

Now that the peek() method is implemented, the pop() method is almost identical. It only calls a different C function:

```
cdef int pop(self) except? 0:
    cdef int value = \
        <int>cqueue.queue_pop_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
```

Lastly, we can provide the Queue with an emptiness indicator in the normal Python way:

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

Note that this method returns either True or False as the return value of the queue_is_empty function is declared as a bint.

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.

The following listing shows the complete implementation that uses cpdef methods where possible. This feature is obviously not available for the extend() method, as the method signature is incompatible with Python argument types.

```
cimport cqueue
cimport python_exc

cdef class Queue:
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
        if self._c_queue is NULL:
            python_exc.PyErr_NoMemory()

    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):
            python_exc.PyErr_NoMemory()

    cdef extend(self, int* values, Py_ssize_t count):
        cdef Py_ssize_t i
        for i in range(count):
            if not cqueue.queue_push_tail(
                    self._c_queue, <void*>values[i]):
                python_exc.PyErr_NoMemory()

    cpdef int peek(self) except? 0:
        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? 0:
        cdef int value = \
            <int>cqueue.queue_pop_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

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

As a quick test with numbers from 0 to 9999 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 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.

## Unicode and passing strings

Similar to the string semantics in Python 3, Cython also strictly separates byte strings and unicode strings. Above all, this means that there is no automatic conversion between byte strings and unicode strings (except for what Python 2 does in string operations). All encoding and decoding must pass through an explicit encoding/decoding step.
It is, however, very easy to pass byte strings between C code and Python. When receiving a byte string from a C library, you can let Cython convert it into a Python byte string by simply assigning it to a Python variable:

```
cdef char* c_string = c_call_returning_a_c_string()
py_string = c_string
```

This creates a Python byte string object that holds a copy of the original C string. It can be safely passed around in Python code, and will be garbage collected when the last reference to it goes out of scope.
To convert the byte string back into a C `char*`, use the opposite assignment:

```
cdef char* other_c_string = py_string
```

This is a very fast operation after which `other_c_string` points to the byte string buffer of the Python string itself. It is tied to the life time of the Python string. When the Python string is garbage collected, the pointer becomes invalid. It is therefore important to keep a reference to the Python string as long as the `char*` is in use. Often enough, this only spans the call to a C function that receives the pointer as parameter. Special care must be taken, however, when the C function stores the pointer for later use. Apart from keeping a Python reference to the string, no manual memory management is required.
The above way of passing and receiving C strings is as simple that, as long as we only deal with binary data in the strings. When we deal with encoded text, however, it is best practice to decode the C byte strings to Python Unicode strings on reception, and to encode Python Unicode strings to C byte strings on the way out.
With a Python byte string object, you would normally just call the `.decode()` method to decode it into a Unicode string:

```
ustring = byte_string.decode('UTF-8')
```

You can do the same in Cython for a C string, but the generated code is rather inefficient for small strings. While Cython could potentially call the Python C-API function for decoding a C string from UTF-8 to Unicode (`PyUnicode_DecodeUTF8()`), the problem is that this requires passing the length of the C string, which Cython cannot know at compile time nor runtime. So it would have to call `strlen()` first, although the user code will already know the length of the string in almost all cases. Also, the encoded byte string might actually contain null bytes, so this isn't even a safe solution. It is therefore currently recommended to call the API functions directly:

```
# .pxd file that comes with Cython
cimport python_unicode

cdef char* c_string = NULL
cdef Py_ssize_t length = 0

# get pointer and length from a C function
get_a_c_string(&c_string, &length)

# decode the string to Unicode
ustring = python_unicode.PyUnicode_DecodeUTF8(
    c_string, length, 'strict')
```

It is common practice to wrap this in a dedicated function, as this needs to be done whenever receiving text from C. This could look as follows:

```
cimport python_unicode
cimport stdlib
cdef extern from "string.h":
    size_t strlen(char *s)

cdef unicode tounicode(char* s):
    return python_unicode.PyUnicode_DecodeUTF8(
        s, strlen(s), 'strict')

cdef unicode tounicode_with_length(
        char* s, size_t length):
    return python_unicode.PyUnicode_DecodeUTF8(
        s, length, 'strict')

cdef unicode tounicode_with_length_and_free(
        char* s, size_t length):
    try:
        return python_unicode.PyUnicode_DecodeUTF8(
            s, length, 'strict')
    finally:
        stdlib.free(s)
```

Most likely, you will prefer shorter function names in your code based on the kind of string being handled. Different types of content often imply different ways of handling them on reception. To make the code more readable and to anticipate future changes, it is good practice to use separate conversion functions for different types of strings.

The reverse way, converting a Python unicode string to a C `char*`, is pretty efficient by itself, assuming that what you actually want is a memory managed byte string:

```
py_byte_string = py_unicode_string.encode('UTF-8')
cdef char* c_string = py_byte_string
```

As noted above, this takes the pointer to the byte buffer of the Python byte string. Trying to do the same without keeping a reference to the intermediate byte string will fail with a compile error:

```
# this will not compile !
cdef char* c_string = py_unicode_string.encode('UTF-8')
```

Here, the Cython compiler notices that the code takes a pointer to a temporary string result that will be garbage collected after the assignment. Later access to the invalidated pointer will most likely result in a crash. Cython will therefore refuse to compile this code.

## Caveats

Since Cython mixes C and Python semantics, some things may be a bit surprising or unintuitive. Work always goes on to make Cython more natural for Python users, so this list may change in the future.

- `10**-2 == 0`, instead of `0.01` like in Python.
- Given two typed `int` variables `a` and `b`, `a % b` has the same sign as the first argument (following C semantics) rather then having the same sign as the second (as in Python). This will change in Cython 0.12.
- Care is needed with unsigned types. `cdef unsigned n = 10; print(range(-n, n))` will print an empty list, since `-n` wraps around to a large positive integer prior to being passed to the `range` function.
- Python's `float` type actually wraps C `double` values, and Python's `int` type wraps C `long` values.

## Further reading

The main documentation is located at http://docs.cython.org/. Some recent features might not have documentation written yet, in such cases some notes can usually be found in the form of a Cython Enhancement Proposal (CEP) on http://wiki.cython.org/enhancements.

[Seljebotn09] contains more information about Cython and NumPy arrays. If you intend to use Cython code in a multi-threaded setting, it is essential to read up on Cython's features for managing the Global Interpreter Lock (the GIL). The same paper contains an explanation of the GIL, and the main documentation explains the Cython features for managing it.

Finally, don't hesitate to ask questions (or post reports on successes!) on the Cython users mailing list [UserList]. The Cython developer mailing list, [DevList], is also open to everybody. Feel free to use it to report a bug, ask for guidance, if you have time to spare to develop Cython, or if you have suggestions for future development.

## Related work

Pyrex [Pyrex] is the compiler project that Cython was originally based on. Many features and the major design decisions of the Cython language were developed by Greg Ewing as part of that project. Today, Cython supersedes the capabilities of Pyrex by providing a higher compatibility with Python code and Python semantics, as well as superior optimisations and better integration with scientific Python extensions like NumPy.

ctypes [ctypes] is a foreign function interface (FFI) for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python code. Compared to Cython, it has the major advantage of being in the standard library and being usable directly from Python code, without any additional dependencies. The major drawback is its performance, which suffers from the Python call overhead as all operations must pass through Python code first. Cython, being a compiled language, can avoid much of this overhead by moving more functionality and long-running loops into fast C code.

SWIG [SWIG] is a wrapper code generator. It makes it very easy to parse large API definitions in C/C++ header files, and to generate straight forward wrapper code for a large set of programming languages. As opposed to Cython, however, it is not a programming language itself. Thin wrappers are easy to generate, but the more functionality a wrapper needs to provide, the harder it gets to implement it with SWIG. Cython, on the other hand, makes it very easy to write very elaborate wrapper code specifically for the Python language. Also, there exists third party code for parsing C header files and using it to generate Cython definitions and module skeletons.

ShedSkin [ShedSkin] is an experimental Python-to-C++ compiler. It uses profiling information and very powerful type inference engine to generate a C++ program from (restricted) Python source code. The main drawback is has no support for calling the Python/C API for operations it does not support natively, and supports very few of the standard Python modules.

## Appendix: Installing MinGW on Windows

1. Download the MinGW installer from http://mingw.org. (As of this writing, the download link is a bit difficult to find; it's under "About" in the menu on the left-hand side). You want the file entitled "Automated MinGW Installer" (currently version 5.1.4).
2. Run it and install MinGW. Only the basic package is strictly needed for Cython, although you might want to grab at least the C++ compiler as well.
3. You need to set up Windows' "PATH" environment variable so that includes e.g. "c:\mingw\bin" (if you installed MinGW to "c:\mingw"). The following web-page describes the procedure in Windows XP (the Vista procedure is similar): http://support.microsoft.com/kb/310519
4. Finally, tell Python to use MinGW as the default compiler (otherwise it will try for Visual C). If Python is installed to "c:\Python26", create a file named "c:\Python26\Lib\distutils\distutils.cfg" containing:

```
[build]
compiler = mingw32
```

The [WinInst] wiki page contains updated information about this procedure. Any contributions towards making the Windows install process smoother is welcomed; it is an unfortunate fact that none of the regular Cython developers have convenient access to Windows.

## References

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn et al., The Cython compiler, http://cython.org.

[Python] G. van Rossum et al., The Python programming language, http://python.org.

[Sage] W. Stein et al., Sage Mathematics Software, http://sagemath.org

[EPD] Enthought, Inc., *The Enthought Python Distribution* http://www.enthought.com/products/epd.php

[Pythonxy] P. Raybault, http://www.pythonxy.com/

[Jython] J. Huginin, B. Warsaw, F. Bock, et al., Jython: Python for the Java platform, http://www.jython.org/

[Seljebotn09] D. S. Seljebotn, Fast numerical computations with Cython, Proceedings of the 8th Python in Science Conference, 2009.

[NumPy] T. Oliphant et al., NumPy, http://numpy.scipy.org/

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

[Pyrex] G. Ewing, Pyrex: C-Extensions for Python, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[ShedSkin] M. Dufour, J. Coughlan, ShedSkin, http://code.google.com/p/shedskin/

[PyPy] The PyPy Group, PyPy: a Python implementation written in Python, http://codespeak.net/pypy.

[IronPython] J. Hugunin et al., http://www.codeplex.com/IronPython.

[SWIG] D.M. Beazley et al., SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++, http://www.swig.org.

[WinInst] http://wiki.cython.org/InstallingOnWindows

[ctypes] T. Heller et al., http://docs.python.org/library/ctypes.html.

[UserList] Cython users mailing list: http://groups.google.com/group/cython-users

[DevList] Cython developer mailing list: http://codespeak.net/mailman/listinfo/cython-dev.