Cython 0.19.1 documentation »

next

Welcome to Cython's Documentation

- Getting Started
 - Cython an overview
 - Installing Cython
 - Building Cython code
 - Faster code via static typing
- Tutorials
 - Calling C functions
 - Using C libraries
 - Extension types (aka. cdef classes)
 - pxd files
 - Caveats
 - Profiling
 - Unicode and passing strings
 - Pure Python Mode
 - Working with NumPy
 - Further reading
 - Related work
 - Appendix: Installing MinGW on Windows
- Users Guide
 - Overview
 - Basic Tutorial
 - Language Basics
 - Extension Types
 - Special Methods of Extension Types
 - Sharing Declarations Between Cython Modules
 - Interfacing with External C Code
 - Source Files and Compilation
 - Early Binding for Speed
 - Using C++ in Cython

- Fused Types (Templates)
- Porting Cython code to PyPy
- Limitations
- Differences between Cython and Pyrex
- Typed Memoryviews
- Using Parallelism
- Debugging your Cython program
- Indices and tables
- Reference Guide
 - Compilation
 - Language Basics
 - Extension Types
 - Interfacing with Other Code
 - Special Mention
 - Limitations
 - Compiler Directives
 - Indices and tables

Cython 0.19.1 documentation »

Cython 0.19.1 documentation » Getting Started »

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 close-to-themetal 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 twofold 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.

G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn et [Cython] al., The Cython compiler, http://cython.org. Jim Hugunin et al., [IronPython] http://www.codeplex.com/IronPython. J. Huginin, B. Warsaw, F. Bock, et al., Jython: Python for [Jython] the Java platform, http://www.jython.org. The PyPy Group, PyPy: a Python implementation written in [PyPy] Python, http://pypy.org. G. Ewing, Pyrex: C-Extensions for Python, [Pyrex] http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/ G. van Rossum et al., The Python programming [Python] language, http://python.org.

Cython 0.19.1 documentation » Getting Started »

Cython 0.19.1 documentation » Getting Started »

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:

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.

[EPD] http://www.enthought.com/products/epd.php
 [Pythonxy] http://www.pythonxy.com/
 W. Stein et al., Sage Mathematics Software,
 [Sage] http://sagemath.org

Cython 0.19.1 documentation » Getting Started »

Cython 0.19.1 documentation » Getting Started »

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.

Currently, distutils is the most common way Cython files are built and distributed. The other methods are described in more detail in the *Source Files and Compilation* section of the reference manual.

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.

<u>F</u> ile <u>E</u> dit <u>V</u> iew Hi <u>s</u> tory <u>B</u> ookmarks <u>T</u> ools <u>H</u> elp			4 ⁴ 9 849
🖕 🛶 🔻 🔁 🛞 🔯 http://localhost:8000/home/admin/9/	Ę	G V Google	0
🚱 Tutorial: Integration exampl 🔉 🔯 http://localhoge2_spyx.htm	1 💥		•
Version 4.1.1.rc2	Iome Published Log Settings Rep	oort a Problem Help Sign out	
Tutorial: Integration example last edited on September 22, 2009 09:51 AM by admin	Sav	e Save & quit Discard & quit	
File 📕 Action 🕶 Data 🗾 sage 💽 🗆 Typeset	Print Worksheet Edit Tex	t Undo Share Publish	
%cython			
<u>cdef</u> extern from "math.h": double sin(double)			
<pre>cdef double f(double x) except *: return sin(x**2)</pre>			
<pre>def integrate(double a, double b, int N): <u>cdef</u> int i <u>cdef</u> double <u>dx</u>, s = 0 <u>dx</u> = (b-a)/N for i in range(N): s += f(a+i*<u>dx</u>) return s * <u>dx</u></pre>			
home_dag9_code_sage6_spyx.chom	ne_dagcode_sage6_spyx.htm	11	
integrate(0, 10, 1e6)			
evaluate 0.58367343190152399			
		jsMath	
Done		4	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.

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

Cython 0.19.1 documentation » Getting Started »

Cython 0.19.1 documentation » Getting Started »

Faster code via static typing

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.

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. (It does not, however, check for overflow when doing arithmetic.) The generated C code will handle the platform dependent sizes of C types correctly and safely in this case.

Types are declared via the cdef keyword.

Typing Variables

Consider the following pure Python code:

```
def f(x):
    return x**2-x

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 35% 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:

```
def f(double x):
    return x**2-x

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 4 times speedup over the pure Python version.

Typing Functions

Python function calls can be expensive – in Cython doubly so 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 x) except? -2:
    return x**2-x
```

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). The except? -2 means that an error will be checked for if -2 is returned (though the ? indicates that -2 may also be used as a valid return value). Alternatively, the slower except * 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. It is also no longer possible to change ``f` at runtime.

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). In fact, cpdef does not just provide a Python wrapper, it also installs logic to allow the method to be overridden by python methods, even when called from within cython. This does

add a tiny overhead compared to cdef methods.

Speedup: 150 times over pure Python.

Determining where to add types

Because static typing is often the key to large speed gains, beginners often have a tendency to type everything in sight. This cuts down on both readability and flexibility, and can even slow things down (e.g. by adding unnecessary type checks, conversions, or slow buffer unpacking). On the other hand, it is easy to kill performance by forgetting to type a critical loop variable. Two essential tools to help with this task are profiling and annotation. Profiling should be the first step of any optimization effort, and can tell you where you are spending your time. Cython's annotation can then tell you why your code is taking time.

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.

<u>F</u> ile <u>E</u> dit <u>V</u> iew Hi <u>s</u> tory <u>B</u> ookmarks <u>T</u> ools <u>H</u> elp	2 ⁴ 4 0 10
🛶 🛶 🔻 🔁 🚳 🔯 http://localhost:8000/home/admin/9/cells/0/_ 🔂 💌 💽 🗸 Google	0
🚱 Untitled (Sage) 🛛 💥 http://localhoge2_spyx.html 💥	•
Generated by Cython 0.11.2 on Tue Sep 22 09:52:15 2009	
Raw output: <u>home dagss sage sage notebook worksheets admin 9 code sage2 spyx 0.c</u>	
<pre>1: 2: include "interrupt.pxi" # ctrl-c interrupt block support 3: include "stdsage.pxi" # ctrl-c interrupt block support 4: 5: include "cdefs.pxi" 6: cdef extern from "math.h": 7: double sin(double) 8: 9: cdef double f(double x) except *: 10: return sin(x**2) 11: 12: def integrate(double a, double b, int N): 13: cdef int i 14: cdef double dx, s = 0 15: dx = (b-a)/N 16: for i in range(N):</pre>	
17: s += f(a+i*dx) 18: return s * dx	

Done

Cython 0.19.1 documentation » Getting Started »

previous | next

* //

Cython 0.19.1 documentation » Tutorials »

Calling C functions

This tutorial describes shortly what you need to know in order to call C library functions from Cython code. For a longer and more comprehensive tutorial about using external C libraries, wrapping them and handling errors, see *Using C libraries*.

For simplicity, let's start with a function from the standard C library. This does not add any dependencies to your code, and it has the additional advantage that Cython already defines many such functions for you. So you can just cimport and use them.

For example, let's say you need a low-level way to parse a number from a char* value. You could use the atoi() function, as defined by the stdlib.h header file. This can be done as follows:

```
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!
```

You can find a complete list of these standard cimport files in Cython's source package Cython/Includes/. It also has a complete set of declarations for CPython's C-API. For example, to test at C compilation time which CPython version your code is being compiled with, you can do this:

```
from cpython.version cimport PY_VERSION_HEX
print PY_VERSION_HEX >= 0x030200F0 # Python version >= 3.2 fina.
```

Cython also provides declarations for the C math library:

```
from libc.math cimport sin
```

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

Dynamic linking

The libc math library is special in that it is not linked by default on some Unix-like systems, such as Linux. In addition to cimporting the declarations, you must configure your build system to link against the shared library m. For distutils, it is enough to add it to the libraries parameter of the Extension() setup:

```
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
)
```

External declarations

If you want to access C code for which Cython does not provide a ready to use declaration, you must declare them yourself. For example, the above sin() function is defined as follows:

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

This declares the sin() function in a way that makes it available to Cython code and instructs Cython to generate C code that includes the math.h header file. 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.

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.

Naming parameters

Both C and Cython support signature declarations without parameter names like this:

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

However, this prevents Cython code from calling it with keyword arguments (supported since Cython 0.19). It is therefore preferable to write the declaration like this instead:

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

You can now make it clear which of the two arguments does what in your call, thus avoiding any ambiguities and often making your code more readable:

```
cdef char* data = "hfvcakdfagbcffvschvxcdfgccbcfhvgcsnfxjh"
pos = strstr(needle='akd', haystack=data)
print pos != NULL
```

Note that changing existing parameter names later is a backwards incompatible API modification, just as for Python code. Thus, if you provide your own declarations for external C or C++ functions, it is usually worth the additional bit of effort to choose the names of their arguments well.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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.

[CAIg] Simon Howard, C Algorithms library, http://calgorithms.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) [1].

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. [2]

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
[2] 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_new()
```

Note that it says <u>_______</u> rather than <u>______</u>. While <u>_______</u> 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 <u>______</u> which is *always* called immediately on construction, before CPython even considers calling <u>_______</u> init___, and which therefore is the right place to initialise cdef fields of the new instance. However, as <u>_______</u> 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 <u>__cinit__()</u> 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 <u>__cinit__()</u>, they must match those of any declared <u>__init__</u> 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 [3]. 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

[3] 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 <u>dealloc</u>(). 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.Distutils import build_ext
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [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 = [Extension("queue", ["queue.pyx"])]

to

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 <u>_____()</u> method apply, so that we end up with this implementation instead:

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 <u>_boo1_()</u> special method (note that Python 2 calls this method <u>_nonzero_</u>, 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_is_empty(self._c_queue):
            raise IndexError("Queue is empty")
    return value
cpdef int pop(self) except? -1:
    if cqueue_is_empty(self._c_queue):
        raise IndexError("Queue is empty")
    return <<u>int</u>>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
 * -1 for errors
 * 0 for reject
 * 1 for accept
 */
typedef int (*predicate_func)(void* user_context, QueueValue da
 /* Pop values as long as the predicate evaluates to true for th
 * returns -1 if the predicate failed with an error and 0 other
 */
int queue_pop_head_until(Queue *queue, predicate_func predicate
 void* user_context);
```

It is normal for C callback functions to have a generic **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 va
```

```
"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 **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 **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.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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, with hardly 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
```

The directive 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)
```

This does slightly more than providing a python wrapper for a cdef method: unlike a cdef method, a cpdef method is fully overrideable by subclasses and instance attributes. This adds a little calling overhead compared to a cdef method.

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. We can even pass in a new function defined in Python-space:

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

This is about 20 times slower, 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
    <...>
```

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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 inclusion by 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</pre>
```

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 $p \times d$ files like this:

- 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 cimport sin in integrate.pyx.
- 2. Add a integrate.pxd so that other modules written in Cython can define fast custom functions to integrate.

Note that if you have a cdef class with attributes, the attributes must be declared in the class declaration $p \times d$ file (if you use one), not the $p \times x$ file. The compiler will tell you about this.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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 second argument (following Python semantics) rather than having the same sign as the first (as in C). The C behavior can be obtained, at some speed gain, by enabling the cdivision directive (versions prior to Cython 0.12 always followed C semantics).
- 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.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

Profiling

This part describes the profiling abilities of Cython. If you are familiar with profiling pure Python code, you can only read the first section (*Cython Profiling Basics*). If you are not familiar with Python profiling you should also read the tutorial (*Profiling Tutorial*) which takes you through a complete example step by step.

Cython Profiling Basics

Profiling in Cython is controlled by a compiler directive. It can be set either for an entire file or on a per function basis via a Cython decorator.

Enabling profiling for a complete source file

Profiling is enabled for a complete source file via a global directive to the Cython compiler at the top of a file:

cython: profile=True

Note that profiling gives a slight overhead to each function call therefore making your program a little slower (or a lot, if you call some small functions very often).

Once enabled, your Cython code will behave just like Python code when called from the cProfile module. This means you can just profile your Cython code together with your Python code using the same tools as for Python code alone.

Disabling profiling function wise

If your profiling is messed up because of the call overhead to some small functions that you rather do not want to see in your profile either because you plan to inline them anyway or because you are sure that you can't make them any faster - you can use a special decorator to disable profiling for one function only:

```
cimport cython
```

```
@cython.profile(False)
def my_often_called_function():
    pass
```

Profiling Tutorial

This will be a complete tutorial, start to finish, of profiling Python code, turning it into Cython code and keep profiling until it is fast enough.

As a toy example, we would like to evaluate the summation of the reciprocals of squares up to a certain integer n for evaluating \pi. The relation we want to use has been proven by Euler in 1735 and is known as the Basel problem.

```
\label{eq:lim_k_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_limit_
```

A simple Python code for evaluating the truncated sum looks like this:

```
#!/usr/bin/env python
# encoding: utf-8
# filename: calc_pi.py

def recip_square(i):
   return 1./i**2

def approx_pi(n=10000000):
   val = 0.
   for k in range(1,n+1):
       val += recip_square(k)
   return (6 * val)**.5
```

On my box, this needs approximately 4 seconds to run the function with the default n. The higher we choose n, the better will be the approximation for \pi. An experienced Python programmer will already see plenty of places to optimize this code. But remember the golden rule of optimization: Never optimize without having profiled. Let me repeat this: **Never** optimize without having profiled your code.

Your thoughts about which part of your code takes too much time are wrong. At least, mine are always wrong. So let's write a short script to profile our code:

```
#!/usr/bin/env python
# encoding: utf-8
# filename: profile.py
import pstats, cProfile
import calc_pi
cProfile.runctx("calc_pi.approx_pi()", globals(), locals(), "Pr
s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

Running this on my box gives the following output:

```
TODO: how to display this not as code but verbatimly?
Sat Nov 7 17:40:54 2009
                            Profile.prof
         10000004 function calls in 6.211 CPU seconds
   Ordered by: internal time
   ncalls
          tottime percall cumtime percall filename:lineno(f
                      3.243
                                       6.211 calc_pi.py:7(appr
             3.243
                              6.211
        1
 10000000
             2.526
                     0.000
                              2.526
                                       0.000 calc_pi.py:4(reci
            0.442
                     0.442
                              0.442
                                       0.442 {range}
        1
                                       6.211 <string>:1(<modul</pre>
             0.000
                     0.000
                              6.211
        1
                                        0.000 {method 'disable'
        1
             0.000
                      0.000
                              0.000
4
```

This contains the information that the code runs in 6.2 CPU seconds. Note that the code got slower by 2 seconds because it ran inside the cProfile module. The table contains the real valuable information. You might want to check the Python profiling documentation for the nitty gritty details. The most important columns here are totime (total time spent in this function **not** counting functions that were called by this function) and cumtime (total time spent in this function **also** counting the functions called by this function). Looking at the tottime column, we see that approximately half the time is spent in approx_pi and the other half is spent in recip_square. Also half a second is spent in range ... of course we should have used xrange for such a big iteration. And in fact, just changing range to xrange makes the code run in 5.8 seconds.

We could optimize a lot in the pure Python version, but since we are interested in Cython, let's move forward and bring this module to Cython. We would do this anyway at some time to get the loop run faster. Here is our first Cython version:

```
# encoding: utf-8
# cython: profile=True
# filename: calc_pi.pyx
def recip_square(int i):
    return 1./i**2
def approx_pi(int n=10000000):
    cdef double val = 0.
    cdef int k
    for k in xrange(1,n+1):
        val += recip_square(k)
    return (6 * val)**.5
```

Note the second line: We have to tell Cython that profiling should be enabled. This makes the Cython code slightly slower, but without this we would not get meaningful output from the cProfile module. The rest of the code is mostly unchanged, I only typed some variables which will likely speed things up a bit.

We also need to modify our profiling script to import the Cython module directly. Here is the complete version adding the import of the pyximport module:

```
#!/usr/bin/env python
# encoding: utf-8
```

```
# filename: profile.py
import pstats, cProfile
import pyximport
pyximport.install()
import calc_pi
cProfile.runctx("calc_pi.approx_pi()", globals(), locals(), "Pr
s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

We only added two lines, the rest stays completely the same. Alternatively, we could also manually compile our code into an extension; we wouldn't need to change the profile script then at all. The script now outputs the following:

```
Sat Nov 7 18:02:33 2009
                           Profile.prof
         10000004 function calls in 4.406 CPU seconds
   Ordered by: internal time
   ncalls
          tottime
                   percall cumtime percall filename:lineno(f
                                       4.406 calc_pi.pyx:7(app
             3.305
                     3.305
        1
                              4.406
                                       0.000 calc_pi.pyx:4(rec
 10000000
            1.101
                     0.000
                              1.101
        1
             0.000
                     0.000
                              4.406
                                       4.406 {calc_pi.approx_p
                                       4.406 <string>:1(<modul
        1
            0.000
                     0.000
                              4.406
                                       0.000 {method 'disable'
        1
             0.000
                     0.000
                              0.000
4
```

We gained 1.8 seconds. Not too shabby. Comparing the output to the previous, we see that recip_square function got faster while the approx_pi function has not changed a lot. Let's concentrate on the recip_square function a bit more. First note, that this function is not to be called from code outside of our module; so it would be wise to turn it into a cdef to reduce call overhead. We should also get rid of the power operator: it is turned into a pow(i,2) function call by Cython, but we could instead just write i*i which could be faster. The

whole function is also a good candidate for inlining. Let's look at the necessary changes for these ideas:

```
# encoding: utf-8
# cython: profile=True
# filename: calc_pi.pyx
cdef inline double recip_square(int i):
   return 1./(i*i)
def approx_pi(int n=100000000):
   cdef double val = 0.
   cdef int k
   for k in xrange(1,n+1):
      val += recip_square(k)
   return (6 * val)**.5
```

Now running the profile script yields:

```
Sat Nov 7 18:10:11 2009
                                             Profile.prof
               10000004 function calls in 2.622 CPU seconds
     Ordered by: internal time
     ncalls tottime percall cumtime percall filename:lineno(f
                                                2.622
                                 1.782
                    1.782
                                                                2.622 calc pi.pyx:7(app
             1

      10000000
      0.840
      0.000

      1
      0.000
      0.000

      1
      0.000
      0.000

      1
      0.000
      0.000

      1
      0.000
      0.000

                                                 0.840 0.000 calc_pl.pyx.4(100
2.622 2.622 {calc_pi.approx_p
                                                2.622
                                                                2.622 <string>:1(<modul</pre>
                                                  0.000
                                                                0.000 {method 'disable'
•
```

That bought us another 1.8 seconds. Not the dramatic change we could have expected. And why is recip_square still in this table; it is supposed to be inlined, isn't it? The reason for this is that Cython still generates profiling code even if the function call is eliminated. Let's tell it to not profile recip_square any more; we couldn't get the function to be much faster anyway:

```
# encoding: utf-8
```

```
# cython: profile=True
```

```
# filename: calc_pi.pyx
cimport cython
@cython.profile(False)
cdef inline double recip_square(int i):
    return 1./(i*i)
def approx_pi(int n=10000000):
    cdef double val = 0.
    cdef int k
    for k in xrange(1,n+1):
        val += recip_square(k)
    return (6 * val)**.5
```

Running this shows an interesting result:

```
Profile.prof
Sat Nov 7 18:15:02 2009
        4 function calls in 0.089 CPU seconds
  Ordered by: internal time
  ncalls
         tottime percall cumtime
                                    percall filename:lineno(f
                                      0.089 calc_pi.pyx:10(ap
            0.089
                    0.089
                             0.089
       1
       1
            0.000
                             0.089
                                      0.089 {calc_pi.approx_p
                     0.000
                                      0.089 <string>:1(<modul
            0.000
                             0.089
       1
                     0.000
                                      0.000 {method 'disable'
       1
            0.000
                     0.000
                             0.000
```

First note the tremendous speed gain: this version only takes 1/50 of the time of our first Cython version. Also note that recip_square has vanished from the table like we wanted. But the most peculiar and import change is that approx_pi also got much faster. This is a problem with all profiling: calling a function in a profile run adds a certain overhead to the function call. This overhead is **not** added to the time spent in the called function, but to the time spent in the **calling** function. In this example, approx_pi didn't need 2.622 seconds in the last run; but it called recip_square 10000000 times, each time taking a little to set up profiling for it. This adds up to the massive time loss of around 2.6 seconds. Having disabled profiling

for the often called function now reveals realistic timings for approx_pi; we could continue optimizing it now if needed.

This concludes this profiling tutorial. There is still some room for improvement in this code. We could try to replace the power operator in approx_pi with a call to sqrt from the C stdlib; but this is not necessarily faster than calling pow(x,0.5).

Even so, the result we achieved here is quite satisfactory: we came up with a solution that is much faster then our original Python version while retaining functionality and readability.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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 by default 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. For simple cases, the module-level c_string_type and c_string_encoding directives can be used to implicitly insert these encoding/decoding steps to ease conversion between Python and C strings.
General notes about C strings

In many use cases, C strings (a.k.a. character pointers) are slow and cumbersome. For one, they usually require manual memory management in one way or another, which makes it more likely to introduce bugs into your code.

Then, Python string objects cache their length, so requesting it (e.g. to validate the bounds of index access or when concatenating two strings into one) is an efficient constant time operation. In contrast, calling strlen() to get this information from a C string takes linear time, which makes many operations on C strings rather costly.

Regarding text processing, Python has built-in support for Unicode, which C lacks completely. If you are dealing with Unicode text, you are usually better off using Python Unicode string objects than trying to work with encoded data in C strings. Cython makes this quite easy and efficient.

Generally speaking: unless you know what you are doing, avoid using C strings where possible and use Python string objects instead. The obvious exception to this is when passing them back and forth from and to external C code. Also, C++ strings remember their length as well, so they can provide a suitable alternative to Python bytes objects in some cases.

Passing byte strings

It is 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()
cdef bytes py_string = c_string
```

A type cast to object or bytes will do the same thing:

py_string = <bytes> 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. It is important to remember that null bytes in the string act as terminator character, as generally known from C. The above will therefore only work correctly for C strings that do not contain null bytes.

Besides not working for null bytes, the above is also very inefficient for long strings, since Cython has to call strlen() on the C string first to find out the length by counting the bytes up to the terminating null byte. In many cases, the user code will know the length already, e.g. because a C function returned it. In this case, it is much more efficient to tell Cython the exact number of bytes by slicing the C string:

```
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)
```

```
py_bytes_string = c_string[:length]
```

Here, no additional byte counting is required and length bytes from the c_string will be copied into the Python bytes object, including any null bytes. Keep in mind that the slice indices are assumed to be accurate in this case and no bounds checking is done, so incorrect slice indices will lead to data corruption and crashes.

Note that the creation of the Python bytes string can fail with an exception, e.g. due to insufficient memory. If you need to free() the string after the conversion, you should wrap the assignment in a try-finally construct:

```
cimport stdlib
cdef bytes py_string
cdef char* c_string = c_call_creating_a_new_c_string()
try:
    py_string = c_string
finally:
    stdlib.free(c_string)
```

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 object, no manual memory management is

required.

Dealing with "const"

Many C libraries use the const modifier in their API to declare that they will not modify a string, or to require that users must not modify a string they return, for example:

```
typedef const char specialChar;
int process_string(const char* s);
const unsigned char* look_up_cached_string(const unsigned char*
.
```

Since version 0.18, Cython has support for the const modifier in the language, so you can declare the above functions straight away as follows:



Previous versions required users to make the necessary declarations at a textual level. If you need to support older Cython versions, you can use the following approach.

In general, for arguments of external C functions, the const modifier does not matter and can be left out in the Cython declaration (e.g. in a .pxd file). The C compiler will still do the right thing, even if you declare this to Cython:

```
cdef extern from "someheader.h":
    int process_string(char* s) # note: looses API informatio
]
```

However, in most other situations, such as for return values and variables that use specifically typedef-ed API types, it does matter

and the C compiler will emit at least a warning if used incorrectly. To help with this, you can use the type definitions in the <code>libc.string</code> module, e.g.:

```
from libc.string cimport const_char, const_uchar

cdef extern from "someheader.h":
    ctypedef const_char specialChar
    int process_string(const_char* s)
    const_uchar* look_up_cached_string(const_uchar* key)
```

Note: even if the API only uses const for function arguments, it is still preferable to properly declare them using these provided const_char types in order to simplify adaptations. In Cython 0.18, these standard declarations have been changed to use the correct const modifier, so your code will automatically benefit from the new const support if it uses them.

Decoding bytes to text

The initially presented way of passing and receiving C strings is sufficient if your code only deals 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')
```

Cython allows you to do the same for a C string, as long as it contains no null bytes:

```
cdef char* some_c_string = c_call_returning_a_c_string()
ustring = some_c_string.decode('UTF-8')
```

And, more efficiently, for strings where the length is known:

```
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)
ustring = c_string[:length].decode('UTF-8')
```

The same should be used when the string contains null bytes, e.g. when it uses an encoding like UCS-4, where each character is encoded in four bytes most of which tend to be 0.

Again, no bounds checking is done if slice indices are provided, so incorrect indices lead to data corruption and crashes. However,

using negative indices is possible since Cython 0.17 and will inject a call to strlen() in order to determine the string length. Obviously, this only works for 0-terminated strings without internal null bytes. Text encoded in UTF-8 or one of the ISO-8859 encodings is usually a good candidate. If in doubt, it's better to pass indices that are 'obviously' correct than to rely on the data to be as expected.

It is common practice to wrap string conversions (and non-trivial type conversions in general) in dedicated functions, as this needs to be done in exactly the same way whenever receiving text from C. This could look as follows:

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.

Encoding text to bytes

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 before, this takes the pointer to the byte buffer of the Python byte string. Trying to do the same without keeping a reference to the Python 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 read invalid memory and likely result in a segfault. Cython will therefore refuse to compile this code.

C++ strings

When wrapping a C++ library, strings will usually come in the form of the **std::string** class. As with C strings, Python byte strings automatically coerce from and to C++ strings:

```
# distutils: language = c++
from libcpp.string cimport string
cdef string s = py_bytes_object
try:
    s.append('abc')
    py_bytes_object = s
finally:
    del s
```

The memory management situation is different than in C because the creation of a C++ string makes an independent copy of the string buffer which the string object then owns. It is therefore possible to convert temporarily created Python objects directly into C++ strings. A common way to make use of this is when encoding a Python unicode string into a C++ string:

cdef string cpp_string = py_unicode_string.encode('UTF-8')

Note that this involves a bit of overhead because it first encodes the Unicode string into a temporarily created Python bytes object and then copies its buffer into a new C++ string.

For the other direction, efficient decoding support is available in Cython 0.17 and later:

```
cdef string s = string('abcdefg')
ustring1 = s.decode('UTF-8')
ustring2 = s[2:-2].decode('UTF-8')
```

For C++ strings, decoding slices will always take the proper length of the string into account and apply Python slicing semantics (e.g. return empty strings for out-of-bounds indices).

Auto encoding and decoding

Cython 0.19 comes with two new directives: c_string_type and c_string_encoding. They can be used to change the Python string types that C/C++ strings coerce from and to. By default, they only coerce from and to the bytes type, and encoding or decoding must be done explicitly, as described above.

There are two use cases where this is inconvenient. First, if all C strings that are being processed (or the large majority) contain text, automatic encoding and decoding from and to Python unicode objects can reduce the code overhead a little. In this case, you can set the c_string_type directive in your module to unicode and the c_string_encoding to the encoding that your C code uses, for example:

```
# cython: c_string_type=unicode, c_string_encoding=utf8
cdef char* c_string = 'abcdefg'
# implicit decoding:
cdef object py_unicode_object = c_string
# explicit conversion to Python bytes:
py_bytes_object = <bytes>c_string
```

The second use case is when all C strings that are being processed only contain ASCII encodable characters (e.g. numbers) and you want your code to use the native legacy string type in Python 2 for them, instead of always using Unicode. In this case, you can set the string type to str:

```
# cython: c_string_type=str, c_string_encoding=ascii
cdef char* c_string = 'abcdefg'
```

```
# implicit decoding in Py3, bytes conversion in Py2:
cdef object py_str_object = c_string
# explicit conversion to Python bytes:
py_bytes_object = <bytes>c_string
# explicit conversion to Python unicode:
py_bytes_object = <unicode>c_string
```

The other direction, i.e. automatic encoding to C strings, is only supported for the ASCII codec (and the "default encoding", which is runtime specific and may or may not be ASCII). This is because CPython handles the memory management in this case by keeping an encoded copy of the string alive together with the original unicode string. Otherwise, there would be no way to limit the lifetime of the encoded string in any sensible way, thus rendering any attempt to extract a C string pointer from it a dangerous endeavour. As long as you stick to the ASCII encoding for the c_string_encoding directive, though, the following will work:

```
# cython: c_string_type=unicode, c_string_encoding=ascii
def func():
    ustring = u'abc'
    cdef char* s = ustring
    return s[0] # returns u'a'
```

(This example uses a function context in order to safely control the lifetime of the Unicode string. Global Python variables can be modified from the outside, which makes it dangerous to rely on the lifetime of their values.)

Source code encoding

When string literals appear in the code, the source code encoding is important. It determines the byte sequence that Cython will store in the C code for bytes literals, and the Unicode code points that Cython builds for unicode literals when parsing the byte encoded source file. Following PEP 263, Cython supports the explicit declaration of source file encodings. For example, putting the following comment at the top of an ISO-8859-15 (Latin-9) encoded source file (into the first or second line) is required to enable ISO-8859-15 decoding in the parser:

-*- coding: ISO-8859-15 -*-

When no explicit encoding declaration is provided, the source code is parsed as UTF-8 encoded text, as specified by PEP 3120. UTF-8 is a very common encoding that can represent the entire Unicode set of characters and is compatible with plain ASCII encoded text that it encodes efficiently. This makes it a very good choice for source code files which usually consist mostly of ASCII characters.

As an example, putting the following line into a UTF-8 encoded source file will print 5, as UTF-8 encodes the letter 'ö' in the two byte sequence '\xc3\xb6':

```
print( len(b'abcö') )
```

whereas the following 150-8859-15 encoded source file will print 4, as the encoding uses only 1 byte for this letter:

```
# -*- coding: ISO-8859-15 -*-
print( len(b'abcö') )
```

Note that the unicode literal u'abcö' is a correctly decoded four character Unicode string in both cases, whereas the unprefixed Python str literal 'abcö' will become a byte string in Python 2 (thus having length 4 or 5 in the examples above), and a 4 character Unicode string in Python 3. If you are not familiar with encodings, this may not appear obvious at first read. See CEP 108 for details.

As a rule of thumb, it is best to avoid unprefixed non-ASCII str literals and to use unicode string literals for all text. Cython also supports the <u>__future__</u> import <u>unicode_literals</u> that instructs the parser to read all unprefixed <u>str</u> literals in a source file as unicode string literals, just like Python 3.

Single bytes and characters

The Python C-API uses the normal C char type to represent a byte value, but it has two special integer types for a Unicode code point value, i.e. a single Unicode character: Py_UNICODE and Py_UCS4. Since version 0.13, Cython supports the first natively, support for Py_UCS4 is new in Cython 0.15. Py_UNICODE is either defined as an unsigned 2-byte or 4-byte integer, or as wchar_t, depending on the platform. The exact type is a compile time option in the build of the CPython interpreter and extension modules inherit this definition at C compile time. The advantage of Py_UCS4 is that it is guaranteed to be large enough for any Unicode code point value, regardless of the platform. It is defined as a 32bit unsigned int or long.

In Cython, the **char** type behaves differently from the **Py_UNICODE** and **Py_UCS4** types when coercing to Python objects. Similar to the behaviour of the bytes type in Python 3, the **char** type coerces to a Python integer value by default, so that the following prints 65 and not A:

```
# -*- coding: ASCII -*-
cdef char char_val = 'A'
assert char_val == 65  # ASCII encoded byte value of 'A'
print( char_val )
```

If you want a Python bytes string instead, you have to request it explicitly, and the following will print A (or b'A' in Python 3):

```
print( <bytes>char_val )
```

The explicit coercion works for any C integer type. Values outside of the range of a char or unsigned char will raise an OverflowError at

runtime. Coercion will also happen automatically when assigning to a typed variable, e.g.:

```
cdef bytes py_byte_string
py_byte_string = char_val
```

On the other hand, the $Py_UNICODE$ and Py_UCS4 types are rarely used outside of the context of a Python unicode string, so their default behaviour is to coerce to a Python unicode object. The following will therefore print the character A, as would the same code with the $Py_UNICODE$ type:

```
cdef Py_UCS4 uchar_val = u'A'
assert uchar_val == 65 # character point value of u'A'
print( uchar_val )
```

Again, explicit casting will allow users to override this behaviour. The following will print 65:

```
cdef Py_UCS4 uchar_val = u'A'
print( <long>uchar_val )
```

Note that casting to a C long (or unsigned long) will work just fine, as the maximum code point value that a Unicode character can have is 1114111 (0x10FFFF). On platforms with 32bit or more, int is just as good.

Narrow Unicode builds

In narrow Unicode builds of CPython before version 3.3, i.e. builds where sys.maxunicode is 65535 (such as all Windows builds, as opposed to 1114111 in wide builds), it is still possible to use Unicode character code points that do not fit into the 16 bit wide $Py_UNICODE$ type. For example, such a CPython build will accept the unicode literal u'\u00012345'. However, the underlying system level encoding leaks into Python space in this case, so that the length of this literal becomes 2 instead of 1. This also shows when iterating over it or when indexing into it. The visible substrings are u'\u00032345' and u'\u009455' in this example. They form a so-called surrogate pair that represents the above character.

For more information on this topic, it is worth reading the Wikipedia article about the UTF-16 encoding.

The same properties apply to Cython code that gets compiled for a narrow CPython runtime environment. In most cases, e.g. when searching for a substring, this difference can be ignored as both the text and the substring will contain the surrogates. So most Unicode processing code will work correctly also on narrow builds. Encoding, decoding and printing will work as expected, so that the above literal turns into exactly the same byte sequence on both narrow and wide Unicode platforms.

However, programmers should be aware that a single $Py_UNICODE$ value (or single 'character' unicode string in CPython) may not be enough to represent a complete Unicode character on narrow platforms. For example, if an independent search for u'\uD808' and u'\uDF45' in a unicode string succeeds, this does not necessarily mean that the character u'\u00012345 is part of that string. It may

well be that two different characters are in the string that just happen to share a code unit with the surrogate pair of the character in question. Looking for substrings works correctly because the two code units in the surrogate pair use distinct value ranges, so the pair is always identifiable in a sequence of code points.

As of version 0.15, Cython has extended support for surrogate pairs so that you can safely use an in test to search character values from the full Py_ucs4 range even on narrow platforms:

```
cdef Py_UCS4 uchar = 0x12345
print( uchar in some_unicode_string )
```

Similarly, it can coerce a one character string with a high Unicode code point value to a Py_UCS4 value on both narrow and wide Unicode platforms:

cdef Py_UCS4 uchar = u'\U00012345'
assert uchar == 0x12345

In CPython 3.3 and later, the Py_UNICODE type is an alias for the system specific wchar_t type and is no longer tied to the internal representation of the Unicode string. Instead, any Unicode character can be represented on all platforms without resorting to surrogate pairs. This implies that narrow builds no longer exist from that version on, regardless of the size of Py_UNICODE. See PEP 393 for details.

Cython 0.16 and later handles this change internally and does the right thing also for single character values as long as either type inference is applied to untyped variables or the portable Py_ucs4 type is explicitly used in the source code instead of the platform specific Py_UNICODE type. Optimisations that Cython applies to the Python unicode type will automatically adapt to PEP 393 at C compile time, as usual.

Iteration

Cython 0.13 supports efficient iteration over **char***, bytes and unicode strings, as long as the loop variable is appropriately typed. So the following will generate the expected C code:

```
cdef char* c_string = ...
cdef char c
for c in c_string[:100]:
    if c == 'A': ...
```

The same applies to bytes objects:

```
cdef bytes bytes_string = ...
```

```
cdef char c
for c in bytes_string:
    if c == 'A': ...
```

For unicode objects, Cython will automatically infer the type of the loop variable as Py_UCS4:

```
cdef unicode ustring = ...
# NOTE: no typing required for 'uchar' !
for uchar in ustring:
    if uchar == u'A': ...
```

The automatic type inference usually leads to much more efficient code here. However, note that some unicode operations still require the value to be a Python object, so Cython may end up generating redundant conversion code for the loop variable value inside of the loop. If this leads to a performance degradation for a specific piece of code, you can either type the loop variable as a Python object explicitly, or assign its value to a Python typed variable somewhere inside of the loop to enforce one-time coercion before running Python operations on it.

There are also optimisations for in tests, so that the following code will run in plain C code, (actually using a switch statement):

```
cdef Py_UCS4 uchar_val = get_a_unicode_character()
if uchar_val in u'abcABCxY':
....
```

Combined with the looping optimisation above, this can result in very efficient character switching code, e.g. in unicode parsers.

Windows and wide character APIs

Windows system APIs natively support Unicode in the form of zeroterminated UTF-16 encoded wchar_t* strings, so called "wide strings".

By default, Windows builds of CPython define Py_UNICODE as a synonym for wchar_t. This makes internal unicode representation compatible with UTF-16 and allows for efficient zero-copy conversions. This also means that Windows builds are always Narrow Unicode builds with all the caveats.

To aid interoperation with Windows APIs, Cython 0.19 supports wide strings (in the form of **Py_UNICODE***) and implicitly converts them to and from unicode string objects. These conversions behave the same way as they do for **char*** and bytes as described in Passing byte strings.

In addition to automatic conversion, unicode literals that appear in C context become C-level wide string literals and len() built-in function is specialized to compute the length of zero-terminated Py_UNICODE* string or array.

Here is an example of how one would call a Unicode API on Windows:

```
cdef extern from "Windows.h":
    ctypedef Py_UNICODE WCHAR
    ctypedef const WCHAR* LPCWSTR
    ctypedef void* HWND
    int MessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaptio
    title = u"Windows Interop Demo - Python %d.%d.%d" % sys.version
    MessageBoxW(NULL, u"Hello Cython \u263a", title, 0)
```

[▲]

Warning: The use of **Py_UNICODE*** strings outside of Windows is strongly discouraged. **Py_UNICODE** is inherently not portable between different platforms and Python versions.

CPython 3.3 has moved to a flexible internal representation of unicode strings (**PEP 393**), making all **Py_UNICODE** related APIs deprecated and inefficient.

One consequence of CPython 3.3 changes is that **len()** of **unicode** strings is always measured in *code points* ("characters"), while Windows API expect the number of UTF-16 *code units* (where each surrogate is counted individually). To always get the number of code units, call **PyUnicode_GetSize()** directly.

Cython 0.19.1 documentation » Tutorials »

previous | next

Cython 0.19.1 documentation » Tutorials »

previous | next

Pure Python Mode

Cython provides language constructs to let the same file be either interpreted or compiled. This is accomplished by the same "magic" module cython that directives use and which must be imported. This is available for both .py and .pyx files.

This is accomplished via special functions and decorators and an (optional) augmenting .pxd file.

Magic Attributes

The currently supported attributes of the cython module are:

 declare declares a typed variable in the current scope, which can be used in place of the cdef type var [= value] construct. This has two forms, the first as an assignment (useful as it creates a declaration in interpreted mode as well):

```
x = cython.declare(cython.int)  # cdef int x
y = cython.declare(cython.double, 0.57721) # cdef double y
```

and the second mode as a simple function call:

```
cython.declare(x=cython.int, y=cython.double) # cdef int x;
```

 locals is a decorator that is used to specify the types of local variables in the function body (including any or all of the argument types):

```
@cython.locals(a=cython.double, b=cython.double, n=cython.p
def foo(a, b, x, y):
    ...
```

address is used in place of the & operator:

```
cython.declare(x=cython.int, x_ptr=cython.p_int)
x_ptr = cython.address(x)
```

 sizeof emulates the sizeof operator. It can take both types and expressions.:

```
cython.declare(n=cython.longlong)
print cython.sizeof(cython.longlong), cython.sizeof(n)
```

struct can be used to create struct types.:

```
MyStruct = cython.struct(x=cython.int, y=cython.int, data=c
a = cython.declare(MyStruct)
```

is equivalent to the code:

```
cdef struct MyStruct:
    int x
    int y
    double data
cdef MyStruct a
```

- union creates union types with exactly the same syntax as struct
- typedef creates a new type:

```
T = cython.typedef(cython.p_int) # ctypedef int* T
```

• compiled is a special variable which is set to True when the compiler runs, and False in the interpreter. Thus the code:

```
if cython.compiled:
    print "Yep, I'm compiled."
else:
    print "Just a lowly interpreted script."
```

will behave differently depending on whether or not the code is loaded as a compiled .so file or a plain .py file.

Augmenting .pxd

If a .pxd file is found with the same name as a .py file, it will be searched for cdef classes and cdef/cpdef functions and methods. It will then convert the corresponding classes/functions/methods in the .py file to be of the correct type. Thus if one had a.pxd:

```
cdef class A:
    cpdef foo(self, int i)
```

the file a.py:

```
class A:
    def foo(self, i):
        print "Big" if i > 1000 else "Small"
```

would be interpreted as:

```
cdef class A:
    cpdef foo(self, int i):
        print "Big" if i > 1000 else "Small"
```

The special Cython module can also be imported and used within the augmenting .pxd file. This makes it possible to add types to a pure Python file without changing the file itself. For example, the following Python file dostuff.py:

```
def dostuff(n):
    t = 0
    for i in range(n):
        t += i
    return t
```

could be augmented with the following .pxd file dostuff.pxd:

import cython

```
@cython.locals(t = cython.int, i = cython.int)
cpdef int dostuff(int n)
```

Besides the cython.locals decorator, the cython.declare() function can also be used to add types to global variables in the augmenting .pxd file.

Note that normal Python (def) functions cannot be declared in .pxd files, so it is currently impossible to override the types of Python functions in .pxd files if they use *args or **kwargs in their signature, for instance.

Types

There are numerous types built in to the Cython module. One has all the standard C types, namely char, short, int, long, longlong as well as their unsigned versions uchar, ushort, uint, ulong, ulonglong. One also has bint and Py_ssize_t. For each type, there are pointer types p_int, pp_int, . . ., up to three levels deep in interpreted mode, and infinitely deep in compiled mode. The Python types int, long and bool are interpreted as C int, long and bint respectively. Also, the Python types list, dict, tuple, . . . may be used, as well as any user defined types.

Pointer types may be constructed with cython.pointer(cython.int), and arrays as cython.int[10]. A limited attempt is made to emulate these more complex types, but only so much can be done from the Python language.

Extension types and cdef functions

Use the @cython.cclass decorator to create a cdef class.

Use the @cython.cfunc decorator for cdef functions and the @cython.ccall decorators for cpdef functions respectively. To declare the argument types, use the @cython.locals() decorator. For the return type, use @cython.returns(a_type).

Here is an example of a **cdef** function:

```
@cython.cfunc
@cython.returns(cython.bint)
@cython.locals(a=cython.int, b=cython.int)
def c_compare(a,b):
    return a == b
```

Cython 0.19.1 documentation » Tutorials »

previous | next

Cython 0.19.1 documentation » Tutorials »

previous | next

Working with NumPy

You can use NumPy from Cython exactly the same as in regular Python, but by doing so you are loosing potentially high speedups because Cython has support for fast access to NumPy arrays. Let's see how this works with a simple example.

The code below does 2D discrete convolution of an image with a filter (and I'm sure you can do better!, let it serve for demonstration purposes). It is both valid Python and valid Cython code. I'll refer to it as both convolve_py.py for the Python version and convolve1.pyx for the Cython version – Cython uses ".pyx" as its file suffix.

```
from __future__ import division
import numpy as np
def naive_convolve(f, g):
   # f is an image and is indexed by (v, w)
   # g is a filter kernel and is indexed by (s, t),
   # it needs odd dimensions
   # h is the output image and is indexed by (x, y),
   # it is not cropped
   if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter support
   # smid and tmid are number of pixels between the center pix
   # and the edge, ie for a 5x5 filter they will be 2.
   #
   # The output size is calculated by adding smid, tmid to eac
   # side of the dimensions of the input image.
   vmax = f.shape[0]
   wmax = f.shape[1]
   smax = g.shape[0]
   tmax = g.shape[1]
   smid = smax // 2
   tmid = tmax // 2
   xmax = vmax + 2*smid
   ymax = wmax + 2*tmid
   # Allocate result image.
   h = np.zeros([xmax, ymax], dtype=f.dtype)
   # Do convolution
   for x in range(xmax):
       for y in range(ymax):
```

```
# Calculate pixel value for h at (x,y). Sum one com
            # for each pixel (s, t) of the filter g.
            s_from = max(smid - x, -smid)
            s_{to} = min((xmax - x) - smid, smid + 1)
            t_from = max(tmid - y, -tmid)
            t_t = min((ymax - y) - tmid, tmid + 1)
            value = 0
            for s in range(s_from, s_to):
                for t in range(t_from, t_to):
                    v = x - smid + s
                    w = y - tmid + t
                    value += g[smid - s, tmid - t] * f[v, w]
            h[x, y] = value
    return h
[▲]
                                                                F
```

This should be compiled to produce yourmod.so (for Linux systems). We run a Python session to test both the Python version (imported from .py-file) and the compiled Cython module.

```
In [1]: import numpy as np
In [2]: import convolve py
In [3]: convolve_py.naive_convolve(np.array([[1, 1, 1]], dtype=
        np.array([[1],[2],[1]], dtype=np.int))
. . .
Out [3]:
array([[1, 1, 1],
    [2, 2, 2],
    [1, 1, 1]])
In [4]: import convolve1
In [4]: convolve1.naive_convolve(np.array([[1, 1, 1]], dtype=np
        np.array([[1],[2],[1]], dtype=np.int))
. . .
Out [4]:
array([[1, 1, 1],
    [2, 2, 2],
    [1, 1, 1]])
In [11]: N = 100
In [12]: f = np.arange(N*N, dtype=np.int).reshape((N,N))
In [13]: q = np.arange(81, dtype=np.int).reshape((9, 9))
In [19]: %timeit -n2 -r3 convolve py.naive convolve(f, g)
2 loops, best of 3: 1.86 s per loop
In [20]: %timeit -n2 -r3 convolve1.naive_convolve(f, g)
2 loops, best of 3: 1.41 s per loop
4
```

There's not such a huge difference yet; because the C code still does exactly what the Python interpreter does (meaning, for instance, that a new object is allocated for each number used). Look at the generated html file and see what is needed for even the simplest statements you get the point quickly. We need to give Cython more information; we need to add types.

Adding types

To add types we use custom Cython syntax, so we are now breaking Python source compatibility. Consider this code (*read the comments!*)

```
from __future__ import division
import numpy as np
# "cimport" is used to import special compile-time information
# about the numpy module (this is stored in a file numpy.pxd wh
# currently part of the Cython distribution).
cimport numpy as np
# We now need to fix a datatype for our arrays. I've used the v
# DTYPE for this, which is assigned to the usual NumPy runtime
# type info object.
DTYPE = np.int
# "ctypedef" assigns a corresponding compile-time type to DTYPE
# every type in the numpy module there's a corresponding compil
# type with a _t-suffix.
ctypedef np.int t DTYPE t
# "def" can type its arguments but not have a return type. The
# arguments for a "def" function is checked at run-time when en
# function.
#
# The arrays f, g and h is typed as "np.ndarray" instances. The
# this has is to a) insert checks that the function arguments r
# NumPy arrays, and b) make some attribute access like f.shape[
# more efficient. (In this example this doesn't matter though.)
def naive_convolve(np.ndarray f, np.ndarray g):
    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter support
    assert f.dtype == DTYPE and g.dtype == DTYPE
    # The "cdef" keyword is also used within functions to type
    # can only be used at the top indendation level (there are
    # problems with allowing them in other places, though we'd
    # good and thought out proposals for it).
   #
   # For the indices, the "int" type is used. This corresponds
    # other C types (like "unsigned int") could have been used
    # Purists could use "Py_ssize_t" which is the proper Python
    # array indices.
    cdef int vmax = f.shape[0]
    cdef int wmax = f.shape[1]
```
```
cdef int smax = g.shape[0]
    cdef int tmax = g.shape[1]
    cdef int smid = smax // 2
    cdef int tmid = tmax // 2
    cdef int xmax = vmax + 2*smid
    cdef int ymax = wmax + 2*tmid
    cdef np.ndarray h = np.zeros([xmax, ymax], dtype=DTYPE)
    cdef int x, y, s, t, v, w
    # It is very important to type ALL your variables. You do n
    # warnings if not, only much slower code (they are implicit.
    # Python objects).
    cdef int s_from, s_to, t_from, t_to
    # For the value variable, we want to use the same data type
    # stored in the array, so we use "DTYPE_t" as defined above
    # NB! An important side-effect of this is that if "value" o
    # datatype size, it will simply wrap around like in C, rath
    # an error like in Python.
    cdef DTYPE t value
    for x in range(xmax):
        for y in range(ymax):
            s_from = max(smid - x, -smid)
            s_{to} = min((xmax - x) - smid, smid + 1)
            t_from = max(tmid - y, -tmid)
            t_t = min((ymax - y) - tmid, tmid + 1)
            value = 0
            for s in range(s_from, s_to):
                for t in range(t_from, t_to):
                    v = x - smid + s
                    w = y - tmid + t
                    value += g[smid - s, tmid - t] * f[v, w]
            h[x, y] = value
    return h
4
                                                               •
```

After building this and continuing my (very informal) benchmarks, I get:

```
In [21]: import convolve2
In [22]: %timeit -n2 -r3 convolve2.naive_convolve(f, g)
2 loops, best of 3: 828 ms per loop
```

Efficient indexing

There's still a bottleneck killing performance, and that is the array lookups and assignments. The []-operator still uses full Python operations – what we would like to do instead is to access the data buffer directly at C speed.

What we need to do then is to type the contents of the ndarray objects. We do this with a special "buffer" syntax which must be told the datatype (first argument) and number of dimensions ("ndim" keyword-only argument, if not provided then one-dimensional is assumed).

These are the needed changes:

```
def naive_convolve(np.ndarray[DTYPE_t, ndim=2] f, np.ndarray[DT
cdef np.ndarray[DTYPE_t, ndim=2] h = ...
```

Usage:

```
In [18]: import convolve3
In [19]: %timeit -n3 -r100 convolve3.naive_convolve(f, g)
3 loops, best of 100: 11.6 ms per loop
```

Note the importance of this change.

Gotcha: This efficient indexing only affects certain index operations, namely those with exactly ndim number of typed integer indices. So if v for instance isn't typed, then the lookup f[v, w] isn't optimized. On the other hand this means that you can continue using Python objects for sophisticated dynamic slicing etc. just as when the array is not typed.

Tuning indexing further

The array lookups are still slowed down by two factors:

- 1. Bounds checking is performed.
- 2. Negative indices are checked for and handled correctly. The code above is explicitly coded so that it doesn't use negative indices, and it (hopefully) always access within bounds. We can add a decorator to disable bounds checking:



Now bounds checking is not performed (and, as a side-effect, if you "do" happen to access out of bounds you will in the best case crash your program and in the worst case corrupt data). It is possible to switch bounds-checking mode in many ways, see [:reference/directives:compiler directives] for more information.

Negative indices are dealt with by ensuring Cython that the indices will be positive, by casting the variables to unsigned integer types (if you do have negative values, then this casting will create a very large positive value instead and you will attempt to access out-of-bounds values). Casting is done with a special <>-syntax. The code below is changed to use either unsigned ints or casting as appropriate:

```
...
cdef int s, t
cdef unsigned int x, y, v, w
cdef int s_from, s_to, t_from, t_to
```

```
cdef DTYPE_t value
for x in range(xmax):
    for y in range(ymax):
         s_from = max(smid - x, -smid)
         s_{to} = min((xmax - x) - smid, smid + 1)
         t_from = max(tmid - y, -tmid)
         t_to = min((ymax - y) - tmid, tmid + 1)
         value = 0
         for s in range(s_from, s_to):
             for t in range(t_from, t_to):
                 v = \langle unsigned int \rangle (x - smid + s)
                 w = \langle unsigned int \rangle (y - tmid + t)
                 value += g[<unsigned int>(smid - s), <unsigned</pre>
         h[x, y] = value
                                                                     .
```

The function call overhead now starts to play a role, so we compare the latter two examples with larger N:

```
In [11]: %timeit -n3 -r100 convolve4.naive_convolve(f, g)
3 loops, best of 100: 5.97 ms per loop
In [12]: N = 1000
In [13]: f = np.arange(N*N, dtype=np.int).reshape((N,N))
In [14]: g = np.arange(81, dtype=np.int).reshape((9, 9))
In [17]: %timeit -n1 -r10 convolve3.naive_convolve(f, g)
1 loops, best of 10: 1.16 s per loop
In [18]: %timeit -n1 -r10 convolve4.naive_convolve(f, g)
1 loops, best of 10: 597 ms per loop
```

(Also this is a mixed benchmark as the result array is allocated within the function call.)

Warning: Speed comes with some cost. Especially it can be dangerous to set typed objects (like f, g and h in our sample code) to None. Setting such objects to None is entirely legal, but all you can do with them is check whether they are None. All other use (attribute lookup or indexing) can potentially segfault or corrupt data (rather than raising exceptions as they would in Python).

The actual rules are a bit more complicated but the main message

is clear: Do not use typed objects without knowing that they are not set to None.

More generic code

It would be possible to do:

```
def naive_convolve(object[DTYPE_t, ndim=2] f, ...):
```

i.e. use **object** rather than **np.ndarray**. Under Python 3.0 this can allow your algorithm to work with any libraries supporting the buffer interface; and support for e.g. the Python Imaging Library may easily be added if someone is interested also under Python 2.x.

There is some speed penalty to this though (as one makes more assumptions compile-time if the type is set to np.ndarray, specifically it is assumed that the data is stored in pure strided mode and not in indirect mode).

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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, but focusses on core development issues. Feel free to use it to report a clear bug, to ask for guidance if you have time to spare to develop Cython, or if you have suggestions for future development.

 [DevList] Cython developer mailing list: http://mail.python.org/mailman/listinfo/cython-devel
 D. S. Seljebotn, Fast numerical computations with
 [Seljebotn09] Cython, Proceedings of the 8th Python in Science Conference, 2009.
 [UserList] Cython users mailing list: http://groups.google.com/group/cython-users

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

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 substantially 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, and to make it as thin or thick as needed at any given place. 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 a very powerful whole-module type inference engine to generate a C++ program from (restricted) Python source code. The main drawback is that it 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.

[ctypes]http://docs.python.org/library/ctypes.html.[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/[SwIG]David M. Beazley et al., SWIG: An Easy to Use Tool for
Integrating Scripting Languages with C and C++,
http://www.swig.org.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Tutorials »

Appendix: Installing MinGW on Windows

- 1. Download the MinGW installer from http://www.mingw.org/wiki/HOWTO_Install_the_MinGW_GCC_C (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
- 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.

Cython 0.19.1 documentation » Tutorials »

Cython 0.19.1 documentation » Users Guide »

Overview

About Cython

Cython is a language that makes writing C extensions for the Python language as easy as Python itself. Cython is based on the well-known Pyrex language by Greg Ewing, but supports more cutting edge functionality and optimizations [1]. The Cython language is very close to the Python language, but Cython additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code.

This makes Cython the ideal language for wrapping external C libraries, and for fast C modules that speed up the execution of Python code.

Future Plans

Cython is not finished. Substantial tasks remaining. See Limitations for a current list.

Footnotes

[1]

For differences with Pyrex see Differences between Cython and Pyrex.

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »



The Basics of Cython

The fundamental nature of Cython can be summed up as follows: Cython is Python with C data types.

Cython is Python: Almost any piece of Python code is also valid Cython code. (There are a few *Limitations*, but this approximation will serve for now.) The Cython compiler will convert it into C code which makes equivalent calls to the Python/C API.

But Cython is much more than that, because parameters and variables can be declared to have C data types. Code which manipulates Python values and C values can be freely intermixed, with conversions occurring automatically wherever possible. Reference count maintenance and error checking of Python operations is also automatic, and the full power of Python's exception handling facilities, including the try-except and try-finally statements, is available to you – even in the midst of manipulating C data.

Cython Hello World

As Cython can accept almost any valid python source file, one of the hardest things in getting started is just figuring out how to compile your extension.

So lets start with the canonical python hello world:

print "Hello World"

So the first thing to do is rename the file to helloworld.pyx. Now we need to make the setup.py, which is like a python Makefile (for more information see *Source Files and Compilation*). Your setup.py should look like:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("helloworld", ["helloworld.pyx"])]
)
```

To use this to build your Cython file use the commandline options:

\$ python setup.py build_ext --inplace

Which will leave a file in your local directory called helloworld.so in unix or helloworld.dll in Windows. Now to use this file: start the python interpreter and simply import it as if it was a regular python module:

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

Congratulations! You now know how to build a Cython extension. But so far this example doesn't really give a feeling why one would ever want to use Cython, so lets create a more realistic example.

pyximport: Cython Compilation the Easy Way

If your module doesn't require any extra C libraries or a special build setup, then you can use the pyximport module by Paul Prescod and Stefan Behnel to load .pyx files directly on import, without having to write a setup.py file. It is shipped and installed with Cython and can be used like this:

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

Since Cython 0.11, the **pyximport** module also has experimental compilation support for normal Python modules. This allows you to automatically run Cython on every .pyx and .py module that Python imports, including the standard library and installed packages. Cython will still fail to compile a lot of Python modules, in which case the import mechanism will fall back to loading the Python source modules instead. The .py import mechanism is installed like this:

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

Fibonacci Fun

From the official Python tutorial a simple fibonacci function is defined as:

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

Now following the steps for the Hello World example we first rename the file to have a *.pyx* extension, lets say fib.pyx, then we create the setup.py file. Using the file created for the Hello World example, all that you need to change is the name of the Cython filename, and the resulting module name, doing this we have:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("fib", ["fib.pyx"])]
)
```

Build the extension with the same command used for the helloworld.pyx:

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

And use the new extension with:

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

Primes

Here's a small example showing some of what can be done. It's a routine for finding prime numbers. You tell it how many primes you want, and it returns them as a Python list.

primes.pyx:

1	<pre>def primes(int kmax):</pre>
2	cdef int n, k, i
3	<pre>cdef int p[1000]</pre>
4	result = []
5	if kmax > 1000:
6	kmax = 1000
7	k = 0
8	n = 2
9	<pre>while k < kmax:</pre>
LO	i = 0
L1	<pre>while i < k and n % p[i] != 0:</pre>
L2	i = i + 1
L3	if i == k:
L4	p[k] = n
L5	k = k + 1
L6	result.append(n)
L7	n = n + 1
18	return result

You'll see that it starts out just like a normal Python function definition, except that the parameter kmax is declared to be of type int . This means that the object passed will be converted to a C integer (or a TypeError. will be raised if it can't be).

Lines 2 and 3 use the cdef statement to define some local C variables. Line 4 creates a Python list which will be used to return the result. You'll notice that this is done exactly the same way it would be in Python. Because the variable result hasn't been given a

type, it is assumed to hold a Python object.

Lines 7-9 set up for a loop which will test candidate numbers for primeness until the required number of primes has been found. Lines 11-12, which try dividing a candidate by all the primes found so far, are of particular interest. Because no Python objects are referred to, the loop is translated entirely into C code, and thus runs very fast.

When a prime is found, lines 14-15 add it to the p array for fast access by the testing loop, and line 16 adds it to the result list. Again, you'll notice that line 16 looks very much like a Python statement, and in fact it is, with the twist that the C parameter **n** is automatically converted to a Python object before being passed to the append method. Finally, at line 18, a normal Python return statement returns the result list.

Compiling primes.pyx with the Cython compiler produces an extension module which we can try out in the interactive interpreter as follows:

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

See, it works! And if you're curious about how much work Cython has saved you, take a look at the C code generated for this module.

Language Details

For more about the Cython language, see *Language Basics*. To dive right in to using Cython in a numerical computation context, see *Cython for NumPy users*.

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »

Language Basics

C variable and type definitions

The **cdef** statement is used to declare C variables, either local or module-level:

cdef int i, j, k
cdef float f, g[42], *h

and C struct, union Or enum types:

```
cdef struct Grail:
    int age
    float volume
cdef union Food:
    char *spam
    float *eggs
cdef enum CheeseType:
    cheddar, edam,
    camembert
cdef enum CheeseState:
    hard = 1
    soft = 2
    runny = 3
```

See also Styles of struct, union and enum declaration

There is currently no special syntax for defining a constant, but you can use an anonymous enum declaration for this purpose, for example,:

```
cdef enum:
    tons_of_spam = 3
```

Note: the words struct, union and enum are used only when defining a type, not when referring to it. For example, to declare a

variable pointing to a Grail you would write:

cdef Grail *gp

and not:

cdef struct Grail *gp # WRONG

There is also a ctypedef statement for giving names to types, e.g.:

ctypedef unsigned long ULong

ctypedef int* IntPtr

Grouping multiple C declarations

If you have a series of declarations that all begin with **cdef**, you can group them into a **cdef** block like this:

```
cdef:
    struct Spam:
        int tons
    int i
    float f
    Spam *p
    void f(Spam *s):
        print s.tons, "Tons of spam"
```

Python functions vs. C functions

There are two kinds of function definition in Cython:

Python functions are defined using the def statement, as in Python. They take Python objects as parameters and return Python objects.

C functions are defined using the new **cdef** statement. They take either Python objects or C values as parameters, and can return either Python objects or C values.

Within a Cython module, Python functions and C functions can call each other freely, but only Python functions can be called from outside the module by interpreted Python code. So, any functions that you want to "export" from your Cython module must be declared as Python functions using def. There is also a hybrid function, called cpdef. A cpdef can be called from anywhere, but uses the faster C calling conventions when being called from other Cython code. A cpdef can also be overridden by a Python method on a subclass or an instance attribute, even when called from Cython. If this happens, most performance gains are of course lost and even if it does not, there is a tiny overhead in calling a cpdef method from Cython

Parameters of either type of function can be declared to have C data types, using normal C declaration syntax. For example,:

```
def spam(int i, char *s):
    ...
cdef int eggs(unsigned long l, float f):
    ...
```

When a parameter of a Python function is declared to have a C data type, it is passed in as a Python object and automatically converted

to a C value, if possible. Automatic conversion is currently only possible for numeric types, string types and structs (composed recusively of any of these types); attempting to use any other type for the parameter of a Python function will result in a compile-time error. Care must be taken with strings to ensure a reference if the pointer is to be used after the call. Structs can be obtained from Python mappings, and again care must be taken with string attributes if they are to be used after the function returns.

C functions, on the other hand, can have parameters of any type, since they're passed in directly using a normal C function call.

A more complete comparison of the pros and cons of these different method types can be found at *Early Binding for Speed*.

Python objects as parameters and return values

If no type is specified for a parameter or return value, it is assumed to be a Python object. (Note that this is different from the C convention, where it would default to int.) For example, the following defines a C function that takes two Python objects as parameters and returns a Python object:

```
cdef spamobjs(x, y):
    ...
```

Reference counting for these objects is performed automatically according to the standard Python/C API rules (i.e. borrowed references are taken as parameters and a new reference is returned).

The name object can also be used to explicitly declare something as a Python object. This can be useful if the name being declared would otherwise be taken as the name of a type, for example,:

```
cdef ftang(object int):
```

. . .

declares a parameter called int which is a Python object. You can also use object as the explicit return type of a function, e.g.:

In the interests of clarity, it is probably a good idea to always be explicit about object parameters in C functions.

Error return values

If you don't do anything special, a function declared with **cdef** that does not return a Python object has no way of reporting Python exceptions to its caller. If an exception is detected in such a function, a warning message is printed and the exception is ignored.

If you want a C function that does not return a Python object to be able to propagate exceptions to its caller, you need to declare an exception value for it. Here is an example:

```
cdef int spam() except -1:
    ...
```

With this declaration, whenever an exception occurs inside spam, it will immediately return with the value **-1**. Furthermore, whenever a call to spam returns **-1**, an exception will be assumed to have occurred and will be propagated.

When you declare an exception value for a function, you should never explicitly return that value. If all possible return values are legal and you can't reserve one entirely for signalling errors, you can use an alternative form of exception value declaration:

```
cdef int spam() except? -1:
```

. . .

The "?" indicates that the value -1 only indicates a possible error. In this case, Cython generates a call to PyErr_Occurred() if the exception value is returned, to make sure it really is an error.

There is also a third form of exception value declaration:

```
cdef int spam() except *:
...
```

This form causes Cython to generate a call to PyErr_Occurred() after every call to spam, regardless of what value it returns. If you have a function returning void that needs to propagate errors, you will have to use this form, since there isn't any return value to test. Otherwise there is little use for this form.

An external C++ function that may raise an exception can be declared with:

cdef int spam() except +

See Using C++ in Cython for more details.

Some things to note:

- Exception values can only declared for functions returning an integer, enum, float or pointer type, and the value must be a constant expression. Void functions can only use the except * form.
- The exception value specification is part of the signature of the function. If you're passing a pointer to a function as a parameter or assigning it to a variable, the declared type of the parameter or variable must have the same exception value specification (or lack thereof). Here is an example of a pointer-to-function
declaration with an exception value:

```
int (*grail)(int, char*) except -1
```

 You don't need to (and shouldn't) declare exception values for functions which return Python objects. Remember that a function with no declared return type implicitly returns a Python object. (Exceptions on such functions are implicitly propagated by returning NULL.)

Checking return values of non-Cython functions

It's important to understand that the except clause does not cause an error to be raised when the specified value is returned. For example, you can't write something like:

```
cdef extern FILE *fopen(char *filename, char *mode) except NULL
```

and expect an exception to be automatically raised if a call to **fopen()** returns NULL. The except clause doesn't work that way; its only purpose is for propagating Python exceptions that have already been raised, either by a Cython function or a C function that calls Python/C API routines. To get an exception from a non-Python-aware function such as **fopen()**, you will have to check the return value and raise it yourself, for example,:

```
cdef FILE* p
p = fopen("spam.txt", "r")
if p == NULL:
    raise SpamError("Couldn't open the spam file")
```

Automatic type conversions

In most situations, automatic conversions will be performed for the basic numeric and string types when a Python object is used in a context requiring a C value, or vice versa. The following table summarises the conversion possibilities.

C types	From Python types	To Python types
[unsigned] char [unsigned] short int, long	int, long	int
unsigned int unsigned long [unsigned] long long	int, long	long
float, double, long double	int, long, float	float
char*	str/bytes	str/bytes [1]
struct		dict
[1] The conversion is to/from Python 3.x.	n str for Python 2.x, a	ind bytes for

Caveats when using a Python string in a C context

You need to be careful when using a Python string in a context expecting a char*. In this situation, a pointer to the contents of the Python string is used, which is only valid as long as the Python string exists. So you need to make sure that a reference to the original Python string is held for as long as the C string is needed. If you can't guarantee that the Python string will live long enough, you will need to copy the C string.

Cython detects and prevents some mistakes of this kind. For instance, if you attempt something like:

cdef char *s
s = pystring1 + pystring2

then Cython will produce the error message Obtaining char* from temporary Python value. The reason is that concatenating the two Python strings produces a new Python string object that is referenced only by a temporary internal variable that Cython generates. As soon as the statement has finished, the temporary variable will be decrefed and the Python string deallocated, leaving s dangling. Since this code could not possibly work, Cython refuses to compile it.

The solution is to assign the result of the concatenation to a Python variable, and then obtain the char* from that, i.e.:

cdef char *s
p = pystring1 + pystring2
s = p

It is then your responsibility to hold the reference p for as long as necessary.

Keep in mind that the rules used to detect such errors are only heuristics. Sometimes Cython will complain unnecessarily, and sometimes it will fail to detect a problem that exists. Ultimately, you need to understand the issue and be careful what you do.

Statements and expressions

Control structures and expressions follow Python syntax for the most part. When applied to Python objects, they have the same semantics as in Python (unless otherwise noted). Most of the Python operators can also be applied to C values, with the obvious semantics.

If Python objects and C values are mixed in an expression, conversions are performed automatically between Python objects and C numeric or string types.

Reference counts are maintained automatically for all Python objects, and all Python operations are automatically checked for errors, with appropriate action taken.

Differences between C and Cython expressions

There are some differences in syntax and semantics between C expressions and Cython expressions, particularly in the area of C constructs which have no direct equivalent in Python.

- There is no -> operator in Cython. Instead of p->x, use p.x
- There is no unary * operator in Cython. Instead of *p, use p[0]
- There is an & operator, with the same semantics as in C.
- The null C pointer is called NULL, not o (and NULL is a reserved

word).

• Type casts are written <type>value , for example,:

```
cdef char* p, float* q
p = <char*>q
```

Scope rules

Cython determines whether a variable belongs to a local scope, the module scope, or the built-in scope completely statically. As with Python, assigning to a variable which is not otherwise declared implicitly declares it to be a Python variable residing in the scope where it is assigned.

Note: A consequence of these rules is that the module-level scope behaves the same way as a Python local scope if you refer to a variable before assigning to it. In particular, tricks such as the following will not work in Cython:

```
try:
    x = True
except NameError:
    True = 1
```

because, due to the assignment, the True will always be looked up in the module-level scope. You would have to do something like this instead:

```
import __builtin__
try:
    True = __builtin__.True
except AttributeError:
    True = 1
```

Built-in Functions

Cython compiles calls to most built-in functions into direct calls to the corresponding Python/C API routines, making them particularly fast.

Only direct function calls using these names are optimised. If you do something else with one of these names that assumes it's a Python object, such as assign it to a Python variable, and later call it, the call will be made as a Python function call.

Function and arguments	Return type	Python/C API Equivalent
abs(obj)	object, double,	PyNumber_Absolute, fabs, fabsf,
callable(obj)	bint	PyObject_Callable
delattr(obj, name)	None	PyObject_DelAttr
exec(code, [glob, [loc]])	object	•
dir(obj)	list	PyObject_Dir
divmod(a, b)	tuple	PyNumber_Divmod
getattr(obj, name, [default]) (Note 1)	object	PyObject_GetAttr
hasattr(obj, name)	bint	PyObject_HasAttr
hash(obj)	int / long	PyObject_Hash
intern(obj)	object	Py*_InternFromString
isinstance(obj, type)	bint	PyObject_IsInstance
issubclass(obj, type)	bint	PyObject_IsSubclass
iter(obj, [sentinel])	object	PyObject_GetIter
len(obj)	Py_ssize_t	PyObject_Length
pow(x, y, [z])	object	PyNumber_Power
reload(obj)	object	PyImport_ReloadModule
repr(obj)	object	PyObject_Repr
setattr(obj, name)	void	PyObject_SetAttr

Note 1: Pyrex originally provided a function getattr3(obj, name, default)() corresponding to the three-argument form of the Python builtin getattr(). Cython still supports this function, but the usage is

deprecated in favour of the normal builtin, which Cython can optimise in both forms.

Operator Precedence

Keep in mind that there are some differences in operator precedence between Python and C, and that Cython uses the Python precedences, not the C ones.

Integer for-loops

Cython recognises the usual Python for-in-range integer loop pattern:

```
for i in range(n):
    ...
```

If **i** is declared as a **cdef** integer type, it will optimise this into a pure C loop. This restriction is required as otherwise the generated code wouldn't be correct due to potential integer overflows on the target architecture. If you are worried that the loop is not being converted correctly, use the annotate feature of the cython commandline (-a) to easily see the generated C code. See *Automatic range conversion*

For backwards compatibility to Pyrex, Cython also supports another form of for-loop:

for i **from** 0 <= i < n:

or:

. . .

. . .

for i from 0 <= i < n by s:</pre>

where s is some integer step size.

Some things to note about the for-from loop:

- The target expression must be a variable name.
- The name between the lower and upper bounds must be the same as the target name.
- The direction of iteration is determined by the relations. If they are both from the set {<, <=} then it is upwards; if they are both from the set {>, >=} then it is downwards. (Any other combination is disallowed.)

Like other Python looping statements, break and continue may be used in the body, and the loop may have an else clause.

The include statement

Warning: Historically the include statement was used for sharing declarations. Use *Sharing Declarations Between Cython Modules* instead.

A Cython source file can include material from other files using the include statement, for example,:

include "spamstuff.pxi"

The contents of the named file are textually included at that point. The included file can contain any complete statements or declarations that are valid in the context where the include statement appears, including other include statements. The contents of the included file should begin at an indentation level of zero, and will be treated as though they were indented to the level of the include statement that is including the file.

Note: There are other mechanisms available for splitting Cython code into separate parts that may be more appropriate in many cases. See *Sharing Declarations Between Cython Modules*.

Conditional Compilation

Some features are available for conditional compilation and compiletime constants within a Cython source file.

Compile-Time Definitions

A compile-time constant can be defined using the DEF statement:

```
DEF FavouriteFood = "spam"
DEF ArraySize = 42
DEF OtherArraySize = 2 * ArraySize + 17
```

The right-hand side of the DEF must be a valid compile-time expression. Such expressions are made up of literal values and names defined using DEF statements, combined using any of the Python expression syntax.

The following compile-time names are predefined, corresponding to the values returned by os.uname().

UNAME_SYSNAME, UNAME_NODENAME, UNAME_RELEASE, UNAME_VERSION, UNAME_MACHINE

The following selection of builtin constants and functions are also available:

None, True, False, abs, bool, chr, cmp, complex, dict, divmod, enumerate, float, hash, hex, int, len, list, long, map, max, min, oct, ord, pow, range, reduce, repr, round, slice, str, sum, tuple, xrange, zip

A name defined using DEF can be used anywhere an identifier can appear, and it is replaced with its compile-time value as though it were written into the source at that point as a literal. For this to work,

the compile-time expression must evaluate to a Python value of type int, long, float Or str.:

```
cdef int a1[ArraySize]
cdef int a2[OtherArraySize]
print "I like", FavouriteFood
```

Conditional Statements

The IF statement can be used to conditionally include or exclude sections of code at compile time. It works in a similar way to the #if preprocessor directive in C.:

```
IF UNAME_SYSNAME == "Windows":
    include "icky_definitions.pxi"
ELIF UNAME_SYSNAME == "Darwin":
    include "nice_definitions.pxi"
ELIF UNAME_SYSNAME == "Linux":
    include "penguin_definitions.pxi"
ELSE:
    include "other_definitions.pxi"
```

The ELIF and ELSE clauses are optional. An IF statement can appear anywhere that a normal statement or declaration can appear, and it can contain any statements or declarations that would be valid in that context, including DEF statements and other IF statements.

The expressions in the IF and ELIF clauses must be valid compiletime expressions as for the DEF statement, although they can evaluate to any Python value, and the truth of the result is determined in the usual Python way. Cython 0.19.1 documentation » Users Guide »

previous | next



Introduction

As well as creating normal user-defined classes with the Python class statement, Cython also lets you create new built-in Python types, known as extension types. You define an extension type using the cdef class statement. Here's an example:

```
cdef class Shrubbery:
    cdef int width, height
    def __init__(self, w, h):
        self.width = w
        self.height = h
    def describe(self):
        print "This shrubbery is", self.width, \
            "by", self.height, "cubits."
```

As you can see, a Cython extension type definition looks a lot like a Python class definition. Within it, you use the def statement to define methods that can be called from Python code. You can even define many of the special methods such as __init_() as you would in Python.

The main difference is that you can use the **cdef** statement to define attributes. The attributes may be Python objects (either generic or of a particular extension type), or they may be of any C data type. So you can use extension types to wrap arbitrary C data structures and provide a Python-like interface to them.

Attributes

Attributes of an extension type are stored directly in the object's C struct. The set of attributes is fixed at compile time; you can't add attributes to an extension type instance at run time simply by assigning to them, as you could with a Python class instance. (You can subclass the extension type in Python and add attributes to instances of the subclass, however.)

There are two ways that attributes of an extension type can be accessed: by Python attribute lookup, or by direct access to the C struct from Cython code. Python code is only able to access attributes of an extension type by the first method, but Cython code can use either method.

By default, extension type attributes are only accessible by direct access, not Python access, which means that they are not accessible from Python code. To make them accessible from Python code, you need to declare them as **public** or **readonly**. For example:

```
cdef class Shrubbery:
    cdef public int width, height
    cdef readonly float depth
```

makes the width and height attributes readable and writable from Python code, and the depth attribute readable but not writable.

Note: You can only expose simple C types, such as ints, floats, and strings, for Python access. You can also expose Python-valued attributes.

Note: Also the **public** and **readonly** options apply only to Python access, not direct access. All the attributes of an extension type are always readable and writable by C-level access.

Type declarations

Before you can directly access the attributes of an extension type, the Cython compiler must know that you have an instance of that type, and not just a generic Python object. It knows this already in the case of the self parameter of the methods of that type, but in other cases you will have to use a type declaration.

For example, in the following function:

```
cdef widen_shrubbery(sh, extra_width): # BAD
sh.width = sh.width + extra_width
```

because the sh parameter hasn't been given a type, the width attribute will be accessed by a Python attribute lookup. If the attribute has been declared **public** or **readonly** then this will work, but it will be very inefficient. If the attribute is private, it will not work at all – the code will compile, but an attribute error will be raised at run time.

The solution is to declare sh as being of type shrubbery, as follows:

```
cdef widen_shrubbery(Shrubbery sh, extra_width):
    sh.width = sh.width + extra_width
```

Now the Cython compiler knows that sh has a C attribute called width and will generate code to access it directly and efficiently. The same consideration applies to local variables, for example,:

```
cdef Shrubbery another_shrubbery(Shrubbery sh1):
    cdef Shrubbery sh2
    sh2 = Shrubbery()
    sh2.width = sh1.width
    sh2.height = sh1.height
    return sh2
```

Type Testing and Casting

Suppose I have a method quest() which returns an object of type **shrubbery**. To access it's width I could write:

```
cdef Shrubbery sh = quest()
print sh.width
```

which requires the use of a local variable and performs a type test on assignment. If you *know* the return value of **quest()** will be of type **Shrubbery** you can use a cast to write:

print (<Shrubbery>quest()).width

This may be dangerous if quest() is not actually a shrubbery, as it will try to access width as a C struct member which may not exist. At the C level, rather than raising an AttributeError, either an nonsensical result will be returned (interpreting whatever data is at at that address as an int) or a segfault may result from trying to access invalid memory. Instead, one can write:

print (<Shrubbery?>quest()).width

which performs a type check (possibly raising a TypeError) before making the cast and allowing the code to proceed.

To explicitly test the type of an object, use the isinstance() method. By default, in Python, the isinstance() method checks the __class__ attribute of the first argument to determine if it is of the required type. However, this is potentially unsafe as the __class__ attribute can be spoofed or changed, but the C structure of an extension type must be correct to access its cdef attributes and call its cdef methods. Cython detects if the second argument is a known extension type and does a type check instead, analogous to Pyrex's typecheck(). The old behavior is always available by passing a tuple as the second parameter:

print isinstance(sh, Shrubbery) # Check the type of sh
print isinstance(sh, (Shrubbery,)) # Check sh.__class__

Extension types and None

When you declare a parameter or C variable as being of an extension type, Cython will allow it to take on the value None as well as values of its declared type. This is analogous to the way a C pointer can take on the value NULL, and you need to exercise the same caution because of it. There is no problem as long as you are performing Python operations on it, because full dynamic type checking will be applied. However, when you access C attributes of an extension type (as in the widen_shrubbery function above), it's up to you to make sure the reference you're using is not None – in the interests of efficiency, Cython does not check this.

You need to be particularly careful when exposing Python functions which take extension types as arguments. If we wanted to make **widen_shrubbery()** a Python function, for example, if we simply wrote:

```
def widen_shrubbery(Shrubbery sh, extra_width): # This is
    sh.width = sh.width + extra_width # dangerous!
```

then users of our module could crash it by passing None for the sh parameter.

One way to fix this would be:

```
def widen_shrubbery(Shrubbery sh, extra_width):
    if sh is None:
        raise TypeError
        sh.width = sh.width + extra_width
```

but since this is anticipated to be such a frequent requirement, Cython provides a more convenient way. Parameters of a Python function declared as an extension type can have a not None clause:

```
def widen_shrubbery(Shrubbery sh not None, extra_width):
    sh.width = sh.width + extra_width
```

Now the function will automatically check that sh is not None along with checking that it has the right type.

Note: not None clause can only be used in Python functions (defined with def) and not C functions (defined with cdef). If you need to check whether a parameter to a C function is None, you will need to do it yourself.

Note: Some more things:

- The self parameter of a method of an extension type is guaranteed never to be None.
- When comparing a value with None, keep in mind that, if x is a Python object, x is None and x is not None are very efficient because they translate directly to C pointer comparisons, whereas x == None and x != None, or simply using x as a boolean value (as in if x: ...) will invoke Python operations and therefore be much slower.

Special methods

Although the principles are similar, there are substantial differences between many of the __xxx_() special methods of extension types and their Python counterparts. There is a *separate page* devoted to this subject, and you should read it carefully before attempting to use any special methods in your extension types.

Properties

There is a special syntax for defining properties in an extension class:

```
cdef class Spam:
    property cheese:
        "A doc string can go here."
        def __get__(self):
            # This is called when the property is read.
            ...
        def __set__(self, value):
            # This is called when the property is written.
            ...
        def __del__(self):
            # This is called when the property is deleted.
```

The <u>__get__()</u>, <u>__set__()</u> and <u>__de1__()</u> methods are all optional; if they are omitted, an exception will be raised when the corresponding operation is attempted.

Here's a complete example. It defines a property which adds to a list each time it is written to, returns the list when it is read, and empties the list when it is deleted.:

```
# cheesy.pyx
cdef class CheeseShop:
    cdef object cheeses
    def __cinit__(self):
        self.cheeses = []
    property cheese:
```

```
def __get__(self):
            return "We don't have: %s" % self.cheeses
        def __set__(self, value):
            self.cheeses.append(value)
        def __del__(self):
            del self.cheeses[:]
# Test input
from cheesy import CheeseShop
shop = CheeseShop()
print shop.cheese
shop.cheese = "camembert"
print shop.cheese
shop.cheese = "cheddar"
print shop.cheese
del shop.cheese
print shop.cheese
# Test output
```

```
We don't have: []
We don't have: ['camembert']
We don't have: ['camembert', 'cheddar']
We don't have: []
```

Subclassing

An extension type may inherit from a built-in type or another extension type:

```
cdef class Parrot:
    ...
cdef class Norwegian(Parrot):
    ...
```

A complete definition of the base type must be available to Cython, so if the base type is a built-in type, it must have been previously declared as an extern extension type. If the base type is defined in another Cython module, it must either be declared as an extern extension type or imported using the **cimport** statement.

An extension type can only have one base class (no multiple inheritance).

Cython extension types can also be subclassed in Python. A Python class can inherit from multiple extension types provided that the usual Python rules for multiple inheritance are followed (i.e. the C layouts of all the base classes must be compatible).

Since Cython 0.13.1, there is a way to prevent extension types from being subtyped in Python. This is done via the final directive, usually set on an extension type using a decorator:

```
cimport cython
```

```
@cython.final
cdef class Parrot:
    def done(self): pass
```

Trying to create a Python subclass from this type will raise a

TypeError at runtime. Cython will also prevent subtyping a final type inside of the same module, i.e. creating an extension type that uses a final type as its base type will fail at compile time. Note, however, that this restriction does not currently propagate to other extension modules, so even final extension types can still be subtyped at the C level by foreign code.

C methods

Extension types can have C methods as well as Python methods. Like C functions, C methods are declared using cdef or cpdef instead of def. C methods are "virtual", and may be overridden in derived extension types. In addition, cpdef methods can even be overridden by python methods when called as C method. This adds a little to their calling overhead compared to a cdef methd:

```
# pets.pyx
cdef class Parrot:
    cdef void describe(self):
        print "This parrot is resting."
cdef class Norwegian(Parrot):
    cdef void describe(self):
        Parrot.describe(self)
        print "Lovely plumage!"
cdef Parrot p1, p2
p1 = Parrot()
p2 = Norwegian()
print "p1:"
p1.describe()
print "p2:"
p2.describe()
```

```
# Output
p1:
This parrot is resting.
p2:
This parrot is resting.
Lovely plumage!
```

The above example also illustrates that a C method can call an inherited C method using the usual Python technique, i.e.:

Parrot.describe(self)

Forward-declaring extension types

Extension types can be forward-declared, like **struct** and **union** types. This will be necessary if you have two extension types that need to refer to each other, e.g.:

```
cdef class Shrubbery # forward declaration
cdef class Shrubber:
    cdef Shrubbery work_in_progress
cdef class Shrubbery:
    cdef Shrubber creator
```

If you are forward-declaring an extension type that has a base class, you must specify the base class in both the forward declaration and its subsequent definition, for example,:

```
cdef class A(B)
....
cdef class A(B):
    # attributes and methods
```

Fast instantiation

Cython provides two ways to speed up the instantiation of extension types. The first one is a direct call to the <u>__new__()</u> special static method, as known from Python. For an extension type <u>Penguin</u>, you could use the following code:

```
cdef class Penguin:
    cdef object food
    def __cinit__(self, food):
        self.food = food
    def __init__(self, food):
        print("eating!")
normal_penguin = Penguin('fish')
fast_penguin = Penguin.__new__(Penguin, 'wheat') # note: not c
```

Note that the path through <u>__new__()</u> will *not* call the type's <u>__init__()</u> method (again, as known from Python). Thus, in the example above, the first instantiation will print <u>eating</u>!, but the second will not. This is only one of the reasons why the <u>__cinit__()</u> method is safer and preferable over the normal <u>__init__()</u> method for extension types.

The second performance improvement applies to types that are often created and deleted in a row, so that they can benefit from a freelist. Cython provides the decorator @cython.freelist(N) for this, which creates a statically sized freelist of N instances for a given type. Example:

```
cimport cython
```

```
@cython.freelist(8)
```

Making extension types weakreferenceable

By default, extension types do not support having weak references made to them. You can enable weak referencing by declaring a C attribute of type object called <u>__weakref__</u>. For example,:

```
cdef class ExplodingAnimal:
    """This animal will self-destruct when it is
    no longer strongly referenced."""
    cdef object __weakref__
```

Public and external extension types

Extension types can be declared extern or public. An extern extension type declaration makes an extension type defined in external C code available to a Cython module. A public extension type declaration makes an extension type defined in a Cython module available to external C code.

External extension types

An extern extension type allows you to gain access to the internals of Python objects defined in the Python core or in a non-Cython extension module.

Note: In previous versions of Pyrex, extern extension types were also used to reference extension types defined in another Pyrex module. While you can still do that, Cython provides a better mechanism for this. See *Sharing Declarations Between Cython Modules*.

Here is an example which will let you get at the C-level members of the built-in complex object.:

```
cdef extern from "complexobject.h":
    struct Py_complex:
        double real
        double imag
    ctypedef class __builtin__.complex [object PyComplexObject]
        cdef Py_complex cval
# A function which uses the above type
def spam(complex c):
    print "Real:", c.cval.real
    print "Imag:", c.cval.imag
*
```

. .

Note: Some important things:

1. In this example, **ctypedef** class has been used. This is because, in the Python header files, the PyComplexObject struct is declared with:

```
typedef struct {
    ...
} PyComplexObject;
```

- 2. As well as the name of the extension type, the module in which its type object can be found is also specified. See the implicit importing section below.
- 3. When declaring an external extension type, you don't declare any methods. Declaration of methods is not required in order to call them, because the calls are Python method calls. Also, as with struct and union, if your extension class declaration is inside a cdef extern from block, you only need to declare those C members which you wish to access.

Name specification clause

The part of the class declaration in square brackets is a special feature only available for extern or public extension types. The full form of this clause is:

```
[object object_struct_name, type type_object_name ]
```

where <code>object_struct_name</code> is the name to assume for the type's C struct, and type_object_name is the name to assume for the type's statically declared type object. (The object and type clauses can be written in either order.)

If the extension type declaration is inside a cdef extern from block,

the object clause is required, because Cython must be able to generate code that is compatible with the declarations in the header file. Otherwise, for extern extension types, the object clause is optional.

For public extension types, the object and type clauses are both required, because Cython must be able to generate code that is compatible with external C code.

Implicit importing

Cython requires you to include a module name in an extern extension class declaration, for example,:

```
cdef extern class MyModule.Spam:
...
```

The type object will be implicitly imported from the specified module and bound to the corresponding name in this module. In other words, in this example an implicit:

```
from MyModule import Spam
```

statement will be executed at module load time.

The module name can be a dotted name to refer to a module inside a package hierarchy, for example,:

```
cdef extern class My.Nested.Package.Spam:
...
```

You can also specify an alternative name under which to import the type using an as clause, for example,:

```
cdef extern class My.Nested.Package.Spam as Yummy:
    ...
```

which corresponds to the implicit import statement:

from My.Nested.Package import Spam as Yummy

Type names vs. constructor names

Inside a Cython module, the name of an extension type serves two distinct purposes. When used in an expression, it refers to a modulelevel global variable holding the type's constructor (i.e. its typeobject). However, it can also be used as a C type name to declare variables, arguments and return values of that type.

When you declare:

```
cdef extern class MyModule.Spam:
    ...
```

the name Spam serves both these roles. There may be other names by which you can refer to the constructor, but only Spam can be used as a type name. For example, if you were to explicitly import MyModule, you could use MyModule.Spam() to create a Spam instance, but you wouldn't be able to use MyModule.Spam as a type name.

When an as clause is used, the name specified in the as clause also takes over both roles. So if you declare:

```
cdef extern class MyModule.Spam as Yummy:
....
```

then Yummy becomes both the type name and a name for the constructor. Again, there are other ways that you could get hold of the constructor, but only Yummy is usable as a type name.
Public extension types

An extension type can be declared public, in which case a .h file is generated containing declarations for its object struct and type object. By including the .h file in external C code that you write, that code can access the attributes of the extension type.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Special Methods of Extension Types

This page describes the special methods currently supported by Cython extension types. A complete list of all the special methods appears in the table at the bottom. Some of these methods behave differently from their Python counterparts or have no direct Python counterparts, and require special mention.

Declaration

Special methods of extension types must be declared with def, not cdef. This does not impact their performance–Python uses different calling conventions to invoke these special methods.

Docstrings

Currently, docstrings are not fully supported in some special methods of extension types. You can place a docstring in the source to serve as a comment, but it won't show up in the corresponding __doc__ attribute at run time. (This seems to be is a Python limitation – there's nowhere in the *PyTypeObject* data structure to put such docstrings.)

Initialisation methods: __cinit__() and __init__()

There are two methods concerned with initialising the object.

The <u>__cinit__()</u> method is where you should perform basic C-level initialisation of the object, including allocation of any C data structures that your object will own. You need to be careful what you do in the <u>__cinit__()</u> method, because the object may not yet be fully valid Python object when it is called. Therefore, you should be careful invoking any Python operations which might touch the object; in particular, its methods.

By the time your <u>__cinit__()</u> method is called, memory has been allocated for the object and any C attributes it has have been initialised to 0 or null. (Any Python attributes have also been initialised to None, but you probably shouldn't rely on that.) Your <u>__cinit__()</u> method is guaranteed to be called exactly once.

If your extension type has a base type, the <u>__cinit__()</u> method of the base type is automatically called before your <u>__cinit__()</u> method is called; you cannot explicitly call the inherited <u>__cinit__()</u> method. If you need to pass a modified argument list to the base type, you will have to do the relevant part of the initialisation in the <u>__init__()</u> method instead (where the normal rules for calling inherited methods apply).

Any initialisation which cannot safely be done in the <u>__cinit__()</u> method should be done in the <u>__init__()</u> method. By the time <u>__init__()</u> is called, the object is a fully valid Python object and all operations are safe. Under some circumstances it is possible for <u>__init__()</u> to be called more than once or not to be called at all, so your other methods should be designed to be robust in such situations.

Any arguments passed to the constructor will be passed to both the __cinit__() method and the __init__() method. If you anticipate subclassing your extension type in Python, you may find it useful to give the __cinit__() method * and ** arguments so that it can accept and ignore extra arguments. Otherwise, any Python subclass which has an __init__() with a different signature will have to override __new__() [1] as well as __init__(), which the writer of a Python class wouldn't expect to have to do. Alternatively, as a convenience, if you declare your __cinit__`() method to take no arguments (other than self) it will simply ignore any extra arguments passed to the constructor without complaining about the signature mismatch.

[1] http://docs.python.org/reference/datamodel.html#object.__new___

Finalization method: __dealloc__()

The counterpart to the <u>__cinit__()</u> method is the <u>__dealloc__()</u> method, which should perform the inverse of the <u>__cinit__()</u> method. Any C data that you explicitly allocated (e.g. via malloc) in your <u>__cinit__()</u> method should be freed in your <u>__dealloc__()</u> method.

You need to be careful what you do in a <u>___dealloc__()</u> method. By the time your <u>___dealloc__()</u> method is called, the object may already have been partially destroyed and may not be in a valid state as far as Python is concerned, so you should avoid invoking any Python operations which might touch the object. In particular, don't call any other methods of the object or do anything which might cause the object to be resurrected. It's best if you stick to just deallocating C data.

You don't need to worry about deallocating Python attributes of your object, because that will be done for you by Cython after your ___dealloc__() method returns.

When subclassing extension types, be aware that the <u>__dealloc__()</u> method of the superclass will always be called, even if it is overridden. This is in contrast to typical Python behavior where superclass methods will not be executed unless they are explicitly called by the subclass.

Note: There is no __del__() method for extension types.

Arithmetic methods

Arithmetic operator methods, such as <u>_add_()</u>, behave differently from their Python counterparts. There are no separate "reversed" versions of these methods (<u>_radd_()</u>, etc.) Instead, if the first operand cannot perform the operation, the same method of the second operand is called, with the operands in the same order.

This means that you can't rely on the first parameter of these methods being "self" or being the right type, and you should test the types of both operands before deciding what to do. If you can't handle the combination of types you've been given, you should return *NotImplemented*.

This also applies to the in-place arithmetic method <u>_ipow_()</u>. It doesn't apply to any of the other in-place methods (<u>_iadd_()</u>, etc.) which always take *self* as the first argument.

Rich comparisons

There are no separate methods for the individual rich comparison operations (__eq__(), __le__(), etc.) Instead there is a single method __richcmp__() which takes an integer indicating which operation is to be performed, as follows:

<	0
==	2
>	4
<=	1
!=	3
>=	5

The __next_() method

Extension types wishing to implement the iterator interface should define a method called __next_(), not next. The Python system will automatically supply a next method which calls your __next_(). Do *NOT* explicitly give your type a next() method, or bad things could happen.

Special Method Table

This table lists all of the special methods together with their parameter and return types. In the table below, a parameter name of self is used to indicate that the parameter has the type that the method belongs to. Other parameters with no type specified in the table are generic Python objects.

You don't have to declare your method as taking these parameter types. If you declare different types, conversions will be performed as necessary.

General

Name	Parameters	Return type	Description
cinit	self,		Basic initialisation (no direct Python equivalent)
init	self,		Further initialisation
dealloc	self		Basic deallocation (no direct Python equivalent)
cmp	х, у	int	3-way comparison
richcmp	x, y, int op	object	Rich comparison (no direct Python equivalent)
str	self	object	str(self)
repr	self	object	repr(self)
hash	self	int	Hash function
call	self,	object	self()
iter	self	object	Return iterator for sequence
getattr	self, name	object	Get attribute

getattribute	self, name	object	Get attribute, unconditionally
setattr	self, name, val		Set attribute
delattr	self, name		Delete attribute

Arithmetic operators

Name	Parameters	Return type	Description
add	х, у	object	binary + operator
sub	х, у	object	binary - operator
mul	х, у	object	* operator
div	х, у	object	/ operator for old-style division
floordiv	х, у	object	// operator
truediv	х, у	object	/ operator for new-style division
mod	х, у	object	% operator
divmod	х, у	object	combined div and mod
pow	x, y, z	object	** operator or pow(x, y, z)
neg	self	object	unary - operator
pos	self	object	unary + operator
abs	self	object	absolute value
nonzero	self	int	convert to boolean
invert	self	object	~ operator
lshift	х, у	object	<< operator
rshift	х, у	object	>> operator
and	х, у	object	& operator
or	х, у	object	operator
xor	х, у	object	^ operator

Numeric conversions

Name Parameters Return type Description	
--	--

int	self	object	Convert to integer
long	self	object	Convert to long integer
float	self	object	Convert to float
oct	self	object	Convert to octal
hex	self	object	Convert to hexadecimal
index (2.5+ only)	self	object	Convert to sequence index

In-place arithmetic operators

Name	Parameters	Return type	Description
iadd	self, x	object	+= operator
isub	self, x	object	-= operator
imul	self, x	object	*= operator
idiv	self, x	object	/= operator for old-style division
ifloordiv	self, x	object	//= operator
itruediv	self, x	object	/= operator for new-style division
imod	self, x	object	%= operator
ipow	x, y, z	object	**= operator
ilshift	self, x	object	<<= operator
irshift	self, x	object	>>= operator
iand	self, x	object	&= operator
ior	self, x	object	= operator
ixor	self, x	object	^= operator

Sequences and mappings

Name	Parameters	Return type	Description
len	self int		len(self)
getitem	self, x	object	self[x]
setitem	self, x, y		self[x] = y

delitem	self, x		del self[x]
getslice	self, Py_ssize_t i, Py_ssize_t j	object	self[i:j]
setslice	self, Py_ssize_t i, Py_ssize_t j, x		self[i:j] = x
delslice	self, Py_ssize_t i, Py_ssize_t j		del self[i:j]
contains	self, x	int	x in self

Iterators

Name	Parameters	Return type	Description
next	self	object	Get next item (called next in Python)

Buffer interface [PEP 3118] (no Python equivalents - see note 1)

Name	Parameters	Return type	Description
getbuffer	self, Py_buffer * <i>view</i> , int flags		
releasebuffer	self, Py_buffer * <i>view</i>		

Buffer interface [legacy] (no Python equivalents - see note 1)

Name	Parameters	Return type	Description
getreadbuffer	self, Py_ssize_t i, void **p		
getwritebuffer	self, Py_ssize_t i, void **p		
	self, Py_ssize_t		

getsegcount	*р
getcharbuffer	self, Py_ssize_t i, char **p

Descriptor objects (see note 2)

Name	Parameters	Return type	Description
get	self, instance, class	object	Get value of attribute
set	self, instance, value		Set value of attribute
delete	self, instance		Delete attribute

Note: (1) The buffer interface was intended for use by C code and is not directly accessible from Python. It is described in the Python/C API Reference Manual of Python 2.x under sections 6.6 and 10.6. It was superseded by the new PEP 3118 buffer protocol in Python 2.6 and is no longer available in Python 3.

Note: (2) Descriptor objects are part of the support mechanism for new-style Python classes. See the discussion of descriptors in the Python documentation. See also PEP 252, "Making Types Look More Like Classes", and PEP 253, "Subtyping Built-In Types".

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Sharing Declarations Between Cython Modules

This section describes a new set of facilities for making C declarations, functions and extension types in one Cython module available for use in another Cython module. These facilities are closely modeled on the Python import mechanism, and can be thought of as a compile-time version of it.

Definition and Implementation files

A Cython module can be split into two parts: a definition file with a .pxd suffix, containing C declarations that are to be available to other Cython modules, and an implementation file with a .pyx suffix, containing everything else. When a module wants to use something declared in another module's definition file, it imports it using the cimport statement.

A .pxd file that consists solely of extern declarations does not need to correspond to an actual .pyx file or Python module. This can make it a convenient place to put common declarations, for example declarations of functions from an *external library* that one wants to use in several modules.

What a Definition File contains

A definition file can contain:

- Any kind of C type declaration.
- extern C function or variable declarations.
- Declarations of C functions defined in the module.
- The definition part of an extension type (see below).

It cannot contain the implementations of any C or Python functions, or any Python class definitions, or any executable statements. It is needed when one wants to access **cdef** attributes and methods, or to inherit from **cdef** classes defined in this module.

Note: You don't need to (and shouldn't) declare anything in a declaration file public in order to make it available to other Cython modules; its mere presence in a definition file does that. You only need a public declaration if you want to make something available to external C code.

What an Implementation File contains

An implementation file can contain any kind of Cython statement, although there are some restrictions on the implementation part of an extension type if the corresponding definition file also defines that type (see below). If one doesn't need to **cimport** anything from this module, then this is the only file one needs.

The cimport statement

The **cimport** statement is used in a definition or implementation file to gain access to names declared in another definition file. Its syntax exactly parallels that of the normal Python import statement:

```
cimport module [, module...]
from module cimport name [as name] [, name [as name] ...]
```

Here is an example. dishes.pxd is a definition file which exports a C data type. restaurant.pxd an implementation file which imports and uses it.

dishes.pxd:

```
cdef enum otherstuff:
    sausage, eggs, lettuce
cdef struct spamdish:
    int oz_of_spam
    otherstuff filler
```

restaurant.pyx:

It is important to understand that the **cimport** statement can only be used to import C data types, C functions and variables, and extension types. It cannot be used to import any Python objects, and (with one exception) it doesn't imply any Python import at run time. If you want to refer to any Python names from a module that you have cimported, you will have to include a regular import statement for it as well.

The exception is that when you use **cimport** to import an extension type, its type object is imported at run time and made available by the name under which you imported it. Using **cimport** to import extension types is covered in more detail below.

If a .pxd file changes, any modules that **cimport** from it may need to be recompiled. The Cython.Build.cythonize utility can take care of this for you.

Search paths for definition files

When you **cimport** a module called modulename, the Cython compiler searches for a file called modulename.pxd along the search path for include files, as specified by -I command line options.

Also, whenever you compile a file modulename.pyx, the corresponding definition file modulename.pxd is first searched for along the same path, and if found, it is processed before processing the .pyx file.

Using cimport to resolve naming conflicts

The **cimport** mechanism provides a clean and simple way to solve the problem of wrapping external C functions with Python functions of the same name. All you need to do is put the extern C declarations into a .pxd file for an imaginary module, and **cimport** that module. You can then refer to the C functions by qualifying them with the name of the module. Here's an example:

c_lunch.pxd:

```
cdef extern from "lunch.h":
    void eject_tomato(float)
```

lunch.pyx:

```
cimport c_lunch
def eject_tomato(float speed):
    c_lunch.eject_tomato(speed)
```

You don't need any c_lunch.pyx file, because the only things defined in c_lunch.pxd are extern C entities. There won't be any actual c_lunch module at run time, but that doesn't matter; the c_lunch.pxd file has done its job of providing an additional namespace at compile time.

Sharing C Functions

C functions defined at the top level of a module can be made available via **cimport** by putting headers for them in the .pxd file, for example,:

:file:`volume.pxd`::

cdef float cube(float)

volume.pyx:

cdef float cube(float x):
 return x * x * x

spammery.pyx:

```
from volume cimport cube

def menu(description, size):
    print description, ":", cube(size), \
        "cubic metres of spam"

menu("Entree", 1)
menu("Main course", 3)
menu("Dessert", 2)
```

Note: When a module exports a C function in this way, an object appears in the module dictionary under the function's name. However, you can't make use of this object from Python, nor can you use it from Cython using a normal import statement; you have to use cimport.

Sharing Extension Types

An extension type can be made available via **cimport** by splitting its definition into two parts, one in a definition file and the other in the corresponding implementation file.

The definition part of the extension type can only declare C attributes and C methods, not Python methods, and it must declare all of that type's C attributes and C methods.

The implementation part must implement all of the C methods declared in the definition part, and may not add any further C attributes. It may also define Python methods.

Here is an example of a module which defines and exports an extension type, and another module which uses it:

Shrubbing.pxd:

```
cdef class Shrubbery:
    cdef int width
    cdef int length
```

Shrubbing.pyx:

```
cdef class Shrubbery:
    def __cinit__(self, int w, int l):
        self.width = w
        self.length = l
def standard_shrubbery():
    return Shrubbery(3, 7)
```

Landscaping.pyx:

cimport Shrubbing
import Shrubbing

```
cdef Shrubbing.Shrubbery sh
sh = Shrubbing.standard_shrubbery()
print "Shrubbery size is %d x %d" % (sh.width, sh.length)
```

One would then need to compile both of these modules, e.g. using

setup.py:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules = cythonize(["Landscaping.pyx", "Shrubbing.py
]
```

Some things to note about this example:

- There is a cdef class Shrubbery declaration in both Shrubbing.pxd and Shrubbing.pyx. When the Shrubbing module is compiled, these two declarations are combined into one.
- In Landscaping.pyx, the cimport Shrubbing declaration allows us to refer to the Shrubbery type as shrubbing.Shrubbery. But it doesn't bind the name Shrubbing in Landscaping's module namespace at run time, so to access Shrubbing.standard_shrubbery() we also need to import Shrubbing.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Interfacing with External C Code

One of the main uses of Cython is wrapping existing libraries of C code. This is achieved by using external declarations to declare the C functions and variables from the library that you want to use.

You can also use public declarations to make C functions and variables defined in a Cython module available to external C code. The need for this is expected to be less frequent, but you might want to do it, for example, if you are embedding Python in another application as a scripting language. Just as a Cython module can be used as a bridge to allow Python code to call C code, it can also be used to allow C code to call Python code.

External declarations

By default, C functions and variables declared at the module level are local to the module (i.e. they have the C static storage class). They can also be declared extern to specify that they are defined elsewhere, for example,:

```
cdef extern int spam_counter
```

```
cdef extern void order_spam(int tons)
```

Referencing C header files

When you use an extern definition on its own as in the examples above, Cython includes a declaration for it in the generated C file. This can cause problems if the declaration doesn't exactly match the declaration that will be seen by other C code. If you're wrapping an existing C library, for example, it's important that the generated C code is compiled with exactly the same declarations as the rest of the library.

To achieve this, you can tell Cython that the declarations are to be found in a C header file, like this:

```
cdef extern from "spam.h":
    int spam_counter
    void order_spam(int tons)
```

The cdef extern from clause does three things:

- 1. It directs Cython to place a *#include* statement for the named header file in the generated C code.
- 2. It prevents Cython from generating any C code for the

declarations found in the associated block.

3. It treats all declarations within the block as though they started with cdef extern.

It's important to understand that Cython does not itself read the C header file, so you still need to provide Cython versions of any declarations from it that you use. However, the Cython declarations don't always have to exactly match the C ones, and in some cases they shouldn't or can't. In particular:

- 1. Leave out any platform-specific extensions to C declarations such as __declspec().
- 2. If the header file declares a big struct and you only want to use a few members, you only need to declare the members you're interested in. Leaving the rest out doesn't do any harm, because the C compiler will use the full definition from the header file.

In some cases, you might not need any of the struct's members, in which case you can just put pass in the body of the struct declaration, e.g.:

```
cdef extern from "foo.h":
    struct spam:
    pass
```

Note: you can only do this inside a cdef extern from block; struct declarations anywhere else must be non-empty.

3. If the header file uses typedef names such as word to refer to platform-dependent flavours of numeric types, you will need a corresponding ctypedef statement, but you don't need to match the type exactly, just use something of the right general kind (int, float, etc). For example,: will work okay whatever the actual size of a **word** is (provided the header file defines it correctly). Conversion to and from Python types, if any, will also be used for this new type.

- 4. If the header file uses macros to define constants, translate them into a normal external variable declaration. You can also declare them as an enum if they contain normal int values. Note that Cython considers enum to be equivalent to int, so do not do this for non-int values.
- 5. If the header file defines a function using a macro, declare it as though it were an ordinary function, with appropriate argument and result types.
- 6. For archaic reasons C uses the keyword void to declare a function taking no parameters. In Cython as in Python, simply declare such functions as foo().

A few more tricks and tips:

• If you want to include a C header because it's needed by another header, but don't want to use any declarations from it, put pass in the extern-from block:

```
cdef extern from "spam.h":
    pass
```

 If you want to include some external declarations, but don't want to specify a header file (because it's included by some other header that you've already included) you can put * in place of the header file name:

```
cdef extern from *:
```

Styles of struct, union and enum declaration

There are two main ways that structs, unions and enums can be declared in C header files: using a tag name, or using a typedef. There are also some variations based on various combinations of these.

It's important to make the Cython declarations match the style used in the header file, so that Cython can emit the right sort of references to the type in the code it generates. To make this possible, Cython provides two different syntaxes for declaring a struct, union or enum type. The style introduced above corresponds to the use of a tag name. To get the other style, you prefix the declaration with ctypedef, as illustrated below.

The following table shows the various possible styles that can be found in a header file, and the corresponding Cython declaration that you should put in the cdef extern from block. Struct declarations are used as an example; the same applies equally to union and enum declarations.

C code	Possibilities for corresponding Cython Code	Comments
		Cython will refer to the
<pre>struct Foo { };</pre>	cdef struct Foo:	as struct Foo in the generated C code.
		Cython will refer to the type simply
<pre>typedef struct { } Foo;</pre>	ctypedef struct Foo:	as Foo in the generated

C code.

<pre>typedef struct foo { } Eoo:</pre>	<pre>cdef struct foo: ctypedef foo Foo #optional Or:</pre>	If the C header uses both a tag and a typedef with <i>different</i> names, you can use either form of declaration
5 1	ctypedef struct Foo: 	in Cython (although if you need to forward reference the type, you'll have to use the first form).
<pre>typedef struct Foo { } Foo;</pre>	cdef struct Foo:	If the header uses the same name for the tag and typedef, you won't be able to include a ctypedef for it – but then, it's not necessary.

Note that in all the cases below, you refer to the type in Cython code

simply as **Foo**, not struct Foo.

Accessing Python/C API routines

One particular use of the cdef extern from statement is for gaining access to routines in the Python/C API. For example,:

```
cdef extern from "Python.h":
```

```
object PyString_FromStringAndSize(char *s, Py_ssize_t len)
```

will allow you to create Python strings containing null bytes.

Special Types

Cython predefines the name Py_ssize_t for use with Python/C API routines. To make your extensions compatible with 64-bit systems, you should always use this type where it is specified in the documentation of Python/C API routines.

Windows Calling Conventions

The <u>__stdcall</u> and <u>__cdecl</u> calling convention specifiers can be used in Cython, with the same syntax as used by C compilers on Windows, for example,:

```
cdef extern int __stdcall FrobnicateWindow(long handle)
```

```
cdef void (__stdcall *callback)(void *)
```

If <u>__stdcall</u> is used, the function is only considered compatible with other <u>__stdcall</u> functions of the same signature.

Resolving naming conflicts - C name specifications

Each Cython module has a single module-level namespace for both Python and C names. This can be inconvenient if you want to wrap some external C functions and provide the Python user with Python functions of the same names.

Cython provides a couple of different ways of solving this problem. The best way, especially if you have many C functions to wrap, is probably to put the extern C function declarations into a different namespace using the facilities described in the section on sharing declarations between Cython modules.

The other way is to use a C name specification to give different Cython and C names to the C function. Suppose, for example, that you want to wrap an external function called <code>eject_tomato()</code>. If you declare it as:

```
cdef extern void c_eject_tomato "eject_tomato" (float speed)
```

then its name inside the Cython module will be c_eject_tomato, whereas its name in C will be eject_tomato. You can then wrap it with:

```
def eject_tomato(speed):
    c_eject_tomato(speed)
```

so that users of your module can refer to it as <code>eject_tomato</code>.

Another use for this feature is referring to external names that happen to be Cython keywords. For example, if you want to call an external function called print, you can rename it to something else in your Cython module.

As well as functions, C names can be specified for variables, structs, unions, enums, struct and union members, and enum values. For example,:
```
cdef extern int one "ein", two "zwei"
cdef extern float three "drei"
cdef struct spam "SPAM":
   int i "eye"
cdef enum surprise "inquisition":
   first "alpha"
   second "beta" = 3
```

Using Cython Declarations from C

Cython provides two methods for making C declarations from a Cython module available for use by external C code—public declarations and C API declarations.

Note: You do not need to use either of these to make declarations from one Cython module available to another Cython module – you should use the **cimport** statement for that. Sharing Declarations Between Cython Modules.

Public Declarations

You can make C types, variables and functions defined in a Cython module accessible to C code that is linked with the module, by declaring them with the public keyword:

```
cdef public struct Bunny: # public type declaration
    int vorpalness
cdef public int spam # public variable declaration
cdef public void grail(Bunny *): # public function declaration
    print "Ready the holy hand grenade"
```

If there are any public declarations in a Cython module, a header file called modulename.h file is generated containing equivalent C declarations for inclusion in other C code.

Users who are embedding Python in C with Cython need to make sure to call Py_Initialize() and Py_Finalize(). For example, in the following snippet that includes modulename.h:

```
#include <Python.h>
#include "modulename.h"
```

```
void grail() {
    Py_Initialize();
    initmodulename();
    Bunny b;
    grail(b);
    Py_Finalize();
}
```

Any C code wanting to make use of these declarations will need to be linked, either statically or dynamically, with the extension module.

If the Cython module resides within a package, then the name of the .h file consists of the full dotted name of the module, e.g. a module called **foo.spam** would have a header file called **foo.spam.h**.

C API Declarations

The other way of making declarations available to C code is to declare them with the api keyword. You can use this keyword with C functions and extension types. A header file called modulename_api.h is produced containing declarations of the functions and extension types, and a function called import_modulename().

C code wanting to use these functions or extension types needs to include the header and call the import_modulename() function. The other functions can then be called and the extension types used as usual.

Any public C type or extension type declarations in the Cython module are also made available when you include modulename_api.h.:

```
# delorean.pyx
cdef public struct Vehicle:
    int speed
    float power
```

```
cdef api void activate(Vehicle *v):
    if v.speed >= 88 and v.power >= 1.21:
        print "Time travel achieved"
```

```
# marty.c
#include "delorean_api.h"
Vehicle car;
int main(int argc, char *argv[]) {
    import_delorean();
    car.speed = atoi(argv[1]);
    car.power = atof(argv[2]);
    activate(&car);
}
```

Note: Any types defined in the Cython module that are used as argument or return types of the exported functions will need to be declared public, otherwise they won't be included in the generated header file, and you will get errors when you try to compile a C file that uses the header.

Using the **api** method does not require the C code using the declarations to be linked with the extension module in any way, as the Python import machinery is used to make the connection dynamically. However, only functions can be accessed this way, not variables.

You can use both **public** and **api** on the same function to make it available by both methods, e.g.:

```
cdef public api void belt_and_braces():
    ...
```

However, note that you should include either modulename.h or modulename_api.h in a given C file, not both, otherwise you may get conflicting dual definitions.

If the Cython module resides within a package, then:

- The name of the header file contains of the full dotted name of the module.
- The name of the importing function contains the full name with dots replaced by double underscores.

E.g. a module called **foo.spam** would have an API header file called **foo.spam_api.h** and an importing function called **import_foo_spam()**.

Multiple public and API declarations

You can declare a whole group of items as **public** and/or **api** all at once by enclosing them in a **cdef** block, for example,:

```
cdef public api:
    void order_spam(int tons)
    char *get_lunch(float tomato_size)
```

This can be a useful thing to do in a .pxd file (see *Sharing Declarations Between Cython Modules*) to make the module's public interface available by all three methods.

Acquiring and Releasing the GIL

Cython provides facilities for acquiring and releasing the Global Interpreter Lock (GIL). This may be useful when calling into (external C) code that may block, or when wanting to use Python from a C callback.

Releasing the GIL

You can release the GIL around a section of code using the with nogil statement:

Code in the body of the statement must not manipulate Python objects in any way, and must not call anything that manipulates Python objects without first re-acquiring the GIL. Cython currently does not check this.

Acquiring the GIL

A C function that is to be used as a callback from C code that is executed without the GIL needs to acquire the GIL before it can manipulate Python objects. This can be done by specifying with gil in the function header:

```
cdef void my_callback(void *data) with gil:
    ...
```

If the callback may be called from another non-Python thread, care must be taken to initialize the GIL first, through a call to PyEval_InitThreads(). If you're already using *cython.parallel* in your module, this will already have been taken care of.

The GIL may also be acquired through the with gil statement:

```
with gil:
        <execute this block with the GIL acquired>
```

Declaring a function as callable without the GIL

You can specify **nogil** in a C function header or function type to declare that it is safe to call without the GIL.:

```
cdef void my_gil_free_func(int spam) nogil:
```

. . .

If you are implementing such a function in Cython, it cannot have any Python arguments, Python local variables, or Python return type, and cannot manipulate Python objects in any way or call any function that does so without acquiring the GIL first. Some of these restrictions are currently checked by Cython, but not all. It is possible that more stringent checking will be performed in the future.

Note: This declaration declares that it is safe to call the function without the GIL, it does not in itself release the GIL.

Declaring a function with gil also implicitly makes its signature nogil.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Source Files and Compilation

Note: See *Compilation* reference section for more details

Cython source file names consist of the name of the module followed by a .pyx extension, for example a module called primes would have a source file named primes.pyx.

Once you have written your <u>.pyx</u> file, there are a couple of ways of turning it into an extension module. One way is to compile it manually with the Cython compiler, e.g.:

\$ cython primes.pyx

This will produce a file called primes.c, which then needs to be compiled with the C compiler using whatever options are appropriate on your platform for generating an extension module. For these options look at the official Python documentation.

The other, and probably better, way is to use the **distutils** extension provided with Cython. The benifit of this method is that it will give the platform specific compilation options, acting like a stripped down autotools.

Basic setup.py

The distutils extension provided with Cython allows you to pass .pyx files directly to the Extension constructor in your setup file.

If you have a single Cython file that you want to turn into a compiled extension, say with filename example.pyx the associated setup.py would be:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("example", ["example.pyx"])]
)
```

To understand the setup.py more fully look at the official distutils documentation. To compile the extension for use in the current directory use:

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

Cython Files Depending on C Files

When you have come C files that have been wrapped with cython and you want to compile them into your extension the basic setup.py file to do this would be:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
sourcefiles = ['example.pyx', 'helper.c', 'another_helper.c']
setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("example", sourcefiles)]
)
```

Notice that the files have been given a name, this is not necessary, but it makes the file easier to format if the list gets long.

The Extension class takes many options, and a fuller explanation can be found in the distutils documentation. Some useful options to know about are include_dirs, libraries, and library_dirs which specify where to find the .h and library files when linking to external libraries.

Multiple Cython Files in a Package

TODO

Distributing Cython modules

It is strongly recommended that you distribute the generated .c files as well as your Cython sources, so that users can install your module without needing to have Cython available.

It is also recommended that Cython compilation not be enabled by default in the version you distribute. Even if the user has Cython installed, he probably doesn't want to use it just to install your module. Also, the version he has may not be the same one you used, and may not compile your sources correctly.

This simply means that the setup.py file that you ship with will just be a normal distutils file on the generated *.c* files, for the basic example we would have instead:

```
from distutils.core import setup
from distutils.extension import Extension
setup(
    ext_modules = [Extension("example", ["example.c"])]
)
```

Pyximport

Cython is a compiler. Therefore it is natural that people tend to go through an edit/compile/test cycle with Cython modules. But my personal opinion is that one of the deep insights in Python's implementation is that a language can be compiled (Python modules are compiled to .pyc) files and hide that compilation process from the end-user so that they do not have to worry about it. Pyximport does this for Cython modules. For instance if you write a Cython module called foo.pyx, with Pyximport you can import it in a regular Python module like this:

import pyximport; pyximport.install()
import foo

Doing so will result in the compilation of foo.pyx (with appropriate exceptions if it has an error in it).

If you would always like to import Cython files without building them specially, you can also the first line above to your sitecustomize.py. That will install the hook every time you run Python. Then you can use Cython modules just with simple import statements. I like to test my Cython modules like this:

```
$ python -c "import foo"
```

Dependency Handling

In Pyximport 1.1 it is possible to declare that your module depends on multiple files, (likely .h and .pxd files). If your Cython module is named foo and thus has the filename foo.pyx then you should make another file in the same directory called foo.pyxdep. The modname.pyxdep file can be a list of filenames or "globs" (like *.pxd or include/*.h). Each filename or glob must be on a separate line. Pyximport will check the file date for each of those files before deciding whether to rebuild the module. In order to keep track of the fact that the dependency has been handled, Pyximport updates the modification time of your ".pyx" source file. Future versions may do something more sophisticated like informing distutils of the dependencies directly.

Limitations

Pyximport does not give you any control over how your Cython file is compiled. Usually the defaults are fine. You might run into problems if you wanted to write your program in half-C, half-Cython and build them into a single library. Pyximport 1.2 will probably do this.

Pyximport does not hide the Distutils/GCC warnings and errors generated by the import process. Arguably this will give you better feedback if something went wrong and why. And if nothing went wrong it will give you the warm fuzzy that pyximport really did rebuild your module as it was supposed to.

For further thought and discussion

I don't think that Python's reload() will do anything for changed .so's on some (all?) platforms. It would require some (easy) experimentation that I haven't gotten around to. But reload is rarely used in applications outside of the Python interactive interpreter and certainly not used much for C extension modules. Info about Windows http://mail.python.org/pipermail/python-list/2001-July/053798.html

setup.py install does not modify sitecustomize.py for you. Should it? Modifying Python's "standard interpreter" behaviour may be more than most people expect of a package they install.. Pyximport puts your .c file beside your .pyx file (analogous to .pyc beside .py). But it puts the platform-specific binary in a build directory as per normal for Distutils. If I could wave a magic wand and get Cython or distutils or whoever to put the build directory I might do it but not necessarily: having it at the top level is *VERY HELPFUL* for debugging Cython problems.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Early Binding for Speed

As a dynamic language, Python encourages a programming style of considering classes and objects in terms of their methods and attributes, more than where they fit into the class hierarchy.

This can make Python a very relaxed and comfortable language for rapid development, but with a price - the 'red tape' of managing data types is dumped onto the interpreter. At run time, the interpreter does a lot of work searching namespaces, fetching attributes and parsing argument and keyword tuples. This run-time 'late binding' is a major cause of Python's relative slowness compared to 'early binding' languages such as C++.

However with Cython it is possible to gain significant speed-ups through the use of 'early binding' programming techniques.

For example, consider the following (silly) code example:

```
cdef class Rectangle:
    cdef int x0, y0
    cdef int x1, y1
    def __init__(self, int x0, int y0, int x1, int y1):
        self.x0 = x0; self.y0 = y0; self.x1 = x1; self.y1 = y1
    def area(self):
        area = (self.x1 - self.x0) * (self.y1 - self.y0)
        if area < 0:
            area = -area
        return area
def rectArea(x0, y0, x1, y1):
    rect = Rectangle(x0, y0, x1, y1)
    return rect.area()
```

In the rectArea() method, the call to rect.area() and the area() method contain a lot of Python overhead.

However, in Cython, it is possible to eliminate a lot of this overhead

in cases where calls occur within Cython code. For example:

```
cdef class Rectangle:
    cdef int x0, y0
    cdef int x1, y1
    def __init__(self, int x0, int y0, int x1, int y1):
        self.x0 = x0; self.y0 = y0; self.x1 = x1; self.y1 = y1
    cdef int _area(self):
        cdef int area
        area = (self.x1 - self.x0) * (self.y1 - self.y0)
        if area < 0:
            area = -area
        return area
    def area(self):
        return self._area()
def rectArea(x0, y0, x1, y1):
    cdef Rectangle rect
    rect = Rectangle(x0, y0, x1, y1)
    return rect._area()
```

Here, in the Rectangle extension class, we have defined two different area calculation methods, the efficient _area() C method, and the Python-callable area() method which serves as a thin wrapper around _area(). Note also in the function rectArea() how we 'early bind' by declaring the local variable rect which is explicitly given the type Rectangle. By using this declaration, instead of just dynamically assigning to rect, we gain the ability to access the much more efficient C-callable _rect() method.

But Cython offers us more simplicity again, by allowing us to declare dual-access methods - methods that can be efficiently called at C level, but can also be accessed from pure Python code at the cost of the Python access overheads. Consider this code:

```
cdef class Rectangle:
    cdef int x0, y0
    cdef int x1, y1
    def __init__(self, int x0, int y0, int x1, int y1):
        self.x0 = x0; self.y0 = y0; self.x1 = x1; self.y1 = y1
```

```
cpdef int area(self):
    cdef int area
    area = (self.x1 - self.x0) * (self.y1 - self.y0)
    if area < 0:
        area = -area
    return area
def rectArea(x0, y0, x1, y1):
    cdef Rectangle rect
    rect = Rectangle(x0, y0, x1, y1)
    return rect.area()
```

Note: in earlier versions of Cython, the **cpdef** keyword is rdef - but has the same effect).

Here, we just have a single area method, declared as **cpdef** to make it efficiently callable as a C function, but still accessible from pure Python (or late-binding Cython) code.

If within Cython code, we have a variable already 'early-bound' (ie, declared explicitly as type Rectangle, (or cast to type Rectangle), then invoking its area method will use the efficient C code path and skip the Python overhead. But if in Pyrex or regular Python code we have a regular object variable storing a Rectangle object, then invoking the area method will require:

- an attribute lookup for the area method
- packing a tuple for arguments and a dict for keywords (both empty in this case)
- using the Python API to call the method

and within the area method itself:

- parsing the tuple and keywords
- executing the calculation code
- converting the result to a python object and returning it

So within Cython, it is possible to achieve massive optimisations by

using strong typing in declaration and casting of variables. For tight loops which use method calls, and where these methods are pure C, the difference can be huge.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Using C++ in Cython

Overview

Cython v0.13 introduces native support for most of the C++ language. This means that the previous tricks that were used to wrap C++ classes (as described in http://wiki.cython.org/WrappingCPlusPlus_ForCython012AndLower) are no longer needed.

Wrapping C++ classes with Cython is now much more straightforward. This document describe in details the new way of wrapping C++ code.

What's new in Cython v0.13 about C++

For users of previous Cython versions, here is a brief overview of the main new features of Cython v0.13 regarding C++ support:

- C++ objects can now be dynamically allocated with new and del keywords.
- C++ objects can be stack-allocated.
- C++ classes can be declared with the new keyword cppclass.
- Templated classes are supported.
- Overloaded functions are supported.
- Overloading of C++ operators (such as operator+, operator[],...) is supported.

Procedure Overview

The general procedure for wrapping a C++ file can now be described as follows:

• Specify C++ language in setup.py script or locally in a source file.

- Create one or more .pxd files with cdef extern from blocks and (if existing) the C++ namespace name. In these blocks,
 - declare classes as cdef cppclass blocks
 - declare public names (variables, methods and constructors)
- Write an extension modules, cimport from the .pxd file and use the declarations.

A simple Tutorial

An example C++ API

Here is a tiny C++ API which we will use as an example throughout this document. Let's assume it will be in a header file called Rectangle.h:

```
namespace shapes {
    class Rectangle {
        public:
            int x0, y0, x1, y1;
            Rectangle(int x0, int y0, int x1, int y1);
            ~Rectangle();
            int getLength();
            int getHeight();
            int getArea();
            void move(int dx, int dy);
        };
    };
}
```

and the implementation in the file called Rectangle.cpp:

```
{
    return (x1 - x0);
}
int Rectangle::getHeight()
{
    return (y1 - y0);
}
int Rectangle::getArea()
{
    return (x1 - x0) * (y1 - y0);
}
void Rectangle::move(int dx, int dy)
{
    x0 += dx;
    y0 += dy;
    x1 += dx;
    y_1 += dy;
}
```

This is pretty dumb, but should suffice to demonstrate the steps involved.

Specify C++ language in setup.py

The best way to build Cython code from setup.py scripts is the cythonize() function. To make Cython generate and compile C++ code with distutils, you just need to pass the option language="c++":

Cython will generate and compile the rect.cpp file (from the

rect.pyx), then it will compile Rectangle.cpp (implementation of the Rectangle class) and link both objects files together into rect.so, which you can then import in Python using import rect (if you forget to link the Rectangle.o, you will get missing symbols while importing the library in Python).

The options can also be passed directly from the source file, which is often preferable. Starting with version 0.17, Cython also allows to pass external source files into the cythonize() command this way. Here is a simplified setup.py file:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    name = "rectangleapp",
    ext_modules = cythonize('*.pyx'),
)
```

And in the .pyx source file, write this into the first comment block, before any source code, to compile it in C++ mode and link it statically against the Rectange.cpp code file:

```
# distutils: language = c++
# distutils: sources = Rectangle.cpp
```

To compile manually (e.g. using make), the cython command-line utility can be used to generate a C++ .cpp file, and then compile it into a python extension. C++ mode for the cython command is turned on with the --cplus option.

Declaring a C++ class interface

The procedure for wrapping a C++ class is quite similar to that for wrapping normal C structs, with a couple of additions. Let's start

here by creating the basic cdef extern from block:

cdef extern from "Rectangle.h" namespace "shapes":

This will make the C++ class def for Rectangle available. Note the namespace declaration. Namespaces are simply used to make the fully qualified name of the object, and can be nested (e.g. "outer::inner") or even refer to classes (e.g. "namespace::MyClass to declare static members on MyClass).

Declare class with cdef cppclass

Now, let's add the Rectangle class to this extern from block - just copy the class name from Rectangle.h and adjust for Cython syntax, so now it becomes:

```
cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Rectangle:
```

Add public attributes

We now need to declare the attributes and methods for use on Cython:

```
cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Rectangle:
        Rectangle(int, int, int, int) except +
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int, int)
```

Note that the constructor is declared as "except +". If the C++ code or the initial memory allocation raises an exception due to a failure, this will let Cython safely raise an appropriate Python exception instead (see below). Without this declaration, C++ exceptions originating from the constructor will not be handled by Cython.

Declare a var with the wrapped C++ class

Now, we use cdef to declare a var of the class with the C++ new statement:

```
cdef Rectangle *rec = new Rectangle(1, 2, 3, 4)
try:
    recLength = rec.getLength()
finally:
    del rec  # delete heap allocated object
```

It's also possible to declare a stack allocated object, as long as it has a "default" constructor:

```
cdef extern from "Foo.h":
    cdef cppclass Foo:
        Foo()
def func():
    cdef Foo foo
    ...
```

Note that, like C++, if the class has only one constructor and it is a default one, it's not necessary to declare it.

Create Cython wrapper class

At this point, we have exposed into our pyx file's namespace the interface of the C++ Rectangle type. Now, we need to make this accessible from external Python code (which is our whole point).

Common programming practice is to create a Cython extension type which holds a C++ instance pointer as an attribute thisptr, and create a bunch of forwarding methods. So we can implement the Python extension type as:

```
cdef class PyRectangle:
    cdef Rectangle *thisptr  # hold a C++ instance which we
    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.thisptr = new Rectangle(x0, y0, x1, y1)
    def __dealloc__(self):
        del self.thisptr
    def getLength(self):
        return self.thisptr.getLength()
    def getHeight(self):
        return self.thisptr.getHeight()
    def getArea(self):
        return self.thisptr.getArea()
    def move(self, dx, dy):
        self.thisptr.move(dx, dy)
```

And there we have it. From a Python perspective, this extension type will look and feel just like a natively defined Rectangle class. If you want to give attribute access, you could just implement some properties:

```
property x0:
    def __get__(self): return self.thisptr.x0
    def __set__(self, x0): self.thisptr.x0 = x0
....
```

Advanced C++ features

We describe here all the C++ features that were not discussed in the above tutorial.

Overloading

Overloading is very simple. Just declare the method with different parameters and use any of them:

```
cdef extern from "Foo.h":
    cdef cppclass Foo:
        Foo(int)
        Foo(bool)
        Foo(int, bool)
        Foo(int, int)
```

Overloading operators

Cython uses C++ for overloading operators:

```
cdef extern from "foo.h":
    cdef cppclass Foo:
        Foo()
        Foo* operator+(Foo*)
        Foo* operator-(Foo)
        int operator*(Foo*)
        int operator/(int)
cdef Foo* foo = new Foo()
cdef int x
cdef Foo* foo2 = foo[0] + foo
foo2 = foo[0] - foo[0]
x = foo[0] * foo2
x = foo[0] / 1
cdef Foo f
```

```
foo = f + &f
foo2 = f - f
del foo, foo2
```

Nested class declarations

C++ allows nested class declaration. Class declarations can also be nested in Cython:

```
cdef extern from "<vector>" namespace "std":
    cdef cppclass vector[T]:
        cppclass iterator:
        T operator*()
        iterator operator++()
        bint operator==(iterator)
        bint operator!=(iterator)
        vector()
        void push_back(T&)
        T& operator[](int)
        T& at(int)
        iterator begin()
        iterator end()
cdef vector[int].iterator iter #iter is declared as being of t
```

Note that the nested class is declared with a cppclass but without a cdef.

C++ operators not compatible with Python syntax

Cython try to keep a syntax as close as possible to standard Python. Because of this, certain C++ operators, like the preincrement ++fooor the dereferencing operator *foo cannot be used with the same syntax as C++. Cython provides functions replacing these operators in a special module cython.operator. The functions provided are:

• cython.operator.dereference for dereferencing.

dereference(foo) will produce the C++ code *(foo)

- cython.operator.preincrement for pre-incrementation. preincrement(foo) will produce the C++ code ++(foo)
- ...

These functions need to be cimported. Of course, one can use a from ... cimport ... as to have shorter and more readable functions. For example: from cython.operator cimport dereference as deref.

Templates

Cython uses a bracket syntax for templating. A simple example for wrapping C++ vector:

```
# import dereference and increment operators
from cython.operator cimport dereference as deref, preincrement
cdef extern from "<vector>" namespace "std":
    cdef cppclass vector[T]:
        cppclass iterator:
            T operator*()
            iterator operator++()
            bint operator==(iterator)
            bint operator!=(iterator)
        vector()
        void push back(T&)
        T& operator[](int)
        T& at(int)
        iterator begin()
        iterator end()
cdef vector[int] *v = new vector[int]()
cdef int i
for i in range(10):
    v.push_back(i)
cdef vector[int].iterator it = v.begin()
while it != v.end():
    print deref(it)
    inc(it)
```

del v

Multiple template parameters can be defined as a list, such as [T, U, V] or [int, bool, char].

Standard library

Most of the containers of the C++ Standard Library have been declared in pxd files located in /Cython/Includes/libcpp. These containers are: deque, list, map, pair, queue, set, stack, vector.

For example:

```
from libcpp.vector cimport vector
cdef vector[int] vect
cdef int i
for i in range(10):
    vect.push_back(i)
for i in range(10):
    print vect[i]
```

The pxd files in /cython/Includes/libcpp also work as good examples on how to declare C++ classes.

Since Cython 0.17, the STL containers coerce from and to the corresponding Python builtin types. The conversion is triggered either by an assignment to a typed variable (including typed function arguments) or by an explicit cast, e.g.:

```
from libcpp.string cimport string
from libcpp.vector cimport vector

cdef string s = py_bytes_object
print(s)
cpp_string = <string> py_unicode_object.encode('utf-8')

cdef vector[int] vect = xrange(1, 10, 2)
```

print(vect) # [1, 3, 5, 7, 9]
cdef vector[string] cpp_strings = b'ab cd ef gh'.split()
print(cpp_strings.get(1)) # b'cd'

The following coercions are available:

Python type =>	C++ type	=> Python type
bytes	std::string	bytes
iterable	std::vector	list
iterable	std::list	list
iterable	std::set	set
iterable (len 2)	std::pair	tuple (len 2)

All conversions create a new container and copy the data into it. The items in the containers are converted to a corresponding type automatically, which includes recursively converting containers inside of containers, e.g. a C++ vector of maps of strings.

Exceptions

Cython cannot throw C++ exceptions, or catch them with a tryexcept statement, but it is possible to declare a function as potentially raising an C++ exception and converting it into a Python exception. For example,

```
cdef extern from "some_file.h":
    cdef int foo() except +
```

This will translate try and the C++ error into an appropriate Python exception. The translation is performed according to the following table (the std:: prefix is omitted from the C++ identifiers):

C++	Python
bad_alloc	MemoryError
bad_cast	TypeError
domain_error	ValueError
invalid_argument	ValueError
-------------------	-----------------
ios_base::failure	I0Error
out_of_range	IndexError
overflow_error	OverflowError
range_error	ArithmeticError
underflow_error	ArithmeticError
(all others)	RuntimeError

The what() message, if any, is preserved. Note that a C++ ios_base_failure can denote EOF, but does not carry enough information for Cython to discern that, so watch out with exception masks on IO streams.

cdef int bar() except +MemoryError

This will catch any C++ error and raise a Python MemoryError in its place. (Any Python exception is valid here.)

```
cdef int raise_py_error()
cdef int something_dangerous() except +raise_py_error
```

If something_dangerous raises a C++ exception then raise_py_error will be called, which allows one to do custom C++ to Python error "translations." If raise_py_error does not actually raise an exception a RuntimeError will be raised.

Static member method

If the Rectangle class has a static member:

```
namespace shapes {
    class Rectangle {
        ...
        public:
            static void do_something();
        };
    };
}
```

you can declare it as a function living in the class namespace, i.e.:

cdef extern from "Rectangle.h" namespace "shapes::Rectangle":
 void do_something()

Caveats and Limitations

Access to C-only functions

Whenever generating C++ code, Cython generates declarations of and calls to functions assuming these functions are C++ (ie, not declared as <u>extern "C" {...}</u>. This is ok if the C functions have C++ entry points, but if they're C only, you will hit a roadblock. If you have a C++ Cython module needing to make calls to pure-C functions, you will need to write a small C++ shim module which:

- includes the needed C headers in an extern "C" block
- contains minimal forwarding functions in C++, each of which calls the respective pure-C function

Declaring/Using References

Question: How do you declare and call a function that takes a reference as an argument?

C++ left-values

C++ allows functions returning a reference to be left-values. This is currently not supported in Cython. cython.operator.dereference(foo) is also not considered a leftvalue.

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »

Fused Types (Templates)

Fused types allow you to have one type definition that can refer to multiple types. This allows you to write a single static-typed cython algorithm that can operate on values of multiple types. Thus fused types allow generic programming and are akin to templates in C++ or generics in languages like Java / C#.

Note: Support is experimental and new in this release, there may be bugs!

Quickstart

cimport cython

```
ctypedef fused char_or_float:
    cython.char
    cython.float
cpdef char_or_float plus_one(char_or_float var):
    return var + 1
def show_me():
    cdef:
        cython.char a = 127
        cython.float b = 127
    print 'char', plus_one(a)
    print 'float', plus_one(b)
```

This gives:

```
>>> show_me()
char -128
float 128.0
```

plus_one(a) "specializes" the fused type char_or_float as a char, whereas plus_one(b) specializes char_or_float as a float.

Declaring Fused Types

Fused types may be declared as follows:

```
cimport cython
```

```
ctypedef fused my_fused_type:
    cython.int
    cython.double
```

This declares a new type called my_fused_type which can be *either* an int *or* a double. Alternatively, the declaration may be written as:

my_fused_type = cython.fused_type(cython.int, cython.float)

Only names may be used for the constituent types, but they may be any (non-fused) type, including a typedef. i.e. one may write:

```
ctypedef double my_double
my_fused_type = cython.fused_type(cython.int, my_double)
```

Using Fused Types

Fused types can be used to declare parameters of functions or methods:

```
cdef cfunc(my_fused_type arg):
    return arg + 1
```

If the you use the same fused type more than once in an argument list, then each specialization of the fused type must be the same:

```
cdef cfunc(my_fused_type arg1, my_fused_type arg2):
    return cython.typeof(arg1) == cython.typeof(arg2)
```

In this case, the type of both parameters is either an int, or a double (according to the previous examples). However, because these arguments are the same fused type of my_fused_type, both arg1 and arg2 must be specialized to the same type. Therefore this function returns True for every possible valid invocation. You are allowed to mix fused types however:

```
def func(A x, B y):
```

where A and B are different fused types. This will result in specialized code paths for all combinations of types contained in A and B.

Fused types and arrays

Note that specializations of only numeric types may not be very useful, as one can usually rely on promotion of types. This is not true for arrays, pointers and typed views of memory however. Indeed, one may write:

```
def myfunc(A[:, :] x):
    ...
# and
cdef otherfunc(A *x):
    ...
```

Selecting Specializations

You can select a specialization (an instance of the function with specific or specialized (i.e., non-fused) argument types) in two ways: either by indexing or by calling.

Indexing

You can index functions with types to get certain specializations, i.e.:

```
cfunc[cython.p_double](p1, p2)
# From Cython space
func[float, double](myfloat, mydouble)
# From Python space
func[cython.float, cython.double](myfloat, mydouble)
```

If a fused type is used as a base type, this will mean that the base type is the fused type, so the base type is what needs to be specialized:

Calling

A fused function can also be called with arguments, where the dispatch is figured out automatically:

```
cfunc(p1, p2)
func(myfloat, mydouble)
```

For a cdef or cpdef function called from Cython this means that the specialization is figured out at compile time. For def functions the arguments are typechecked at runtime, and a best-effort approach is performed to figure out which specialization is needed. This means that this may result in a runtime TypeError if no specialization was found. A cpdef function is treated the same way as a def function if the type of the function is unknown (e.g. if it is external and there is no cimport for it).

The automatic dispatching rules are typically as follows, in order of preference:

- try to find an exact match
- choose the biggest corresponding numerical type (biggest float, biggest complex, biggest int)

Built-in Fused Types

There are some built-in fused types available for convenience, these are:

```
cython.integral # short, int, long
cython.floating # float, double
cython.numeric # short, int, long, float, double, float comple
```

Casting Fused Functions

Fused cdef and cpdef functions may be cast or assigned to C function pointers as follows:

Type Checking Specializations

Decisions can be made based on the specializations of the fused parameters. False conditions are pruned to avoid invalid code. One may check with is, is not and == and != to see if a fused type is equal to a certain other non-fused type (to check the specialization), or use in and not in to figure out whether a specialization is part of another set of types (specified as a fused type). In example:

```
ctypedef fused bunch_of_types:
    ...
ctypedef fused string_t:
    cython.p_char
    bytes
    unicode
cdef cython.integral myfunc(cython.integral i, bunch_of_types s
    cdef int *int_pointer
    cdef long *long_pointer
    # Only one of these branches will be compiled for each spec
    if cython.integral is int:
        int_pointer = &i
    else:
        long_pointer = &i
    if bunch_of_types in string_t:
        print "s is a string!"
```

signatures_

Finally, function objects from def or cpdef functions have an attribute ______signatures____, which maps the signature strings to the actual specialized functions. This may be useful for inspection. Listed signature strings may also be used as indices to the fused function, but the index format may change between Cython versions:

```
specialized_function = fused_function["MyExtensionClass|int|flo
]
```

It would usually be preferred to index like this, however:

```
specialized_function = fused_function[MyExtensionClass, int, fl
]
```

Although the latter will select the biggest types for int and float from Python space, as they are not type identifiers but builtin types there. Passing cython.int and cython.float would resolve that, however.

For memoryview indexing from python space we can do the following:

```
ctypedef fused my_fused_type:
    int[:, ::1]
    float[:, ::1]
def func(my_fused_type array):
    ...
my_fused_type[cython.int[:, ::1]](myarray)
```

The same goes for when using e.g. cython.numeric[:, :].

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »

Porting Cython code to PyPy

Since version 0.17, Cython has basic support for cpyext, the layer in PyPy that emulates CPython's C-API. This is achieved by making the generated C code adapt at C compile time, so the generated code will compile in both CPython and PyPy unchanged.

However, beyond what Cython can cover and adapt internally, the cpyext C-API emulation involves some differences to the real C-API in CPython that have a visible impact on user code. This page lists major differences and ways to deal with them in order to write Cython code that works in both CPython and PyPy.

Reference counts

A general design difference in PyPy is that the runtime does not use reference counting internally but always a garbage collector. Reference counting is only emulated at the cpyext layer by counting references being held in C space. This implies that the reference count in PyPy is generally different from that in CPython because it does not count any references held in Python space.

Object lifetime

As a direct consequence of the different garbage collection characteristics, objects may see the end of their lifetime at other points than in CPython. Special care therefore has to be taken when objects are expected to have died in CPython but may not in PyPy. Specifically, a deallocator method of an extension type (___dealloc__()) may get called at a much later point than in CPython, triggered rather by memory getting tighter than by objects dying.

If the point in the code is known when an object is supposed to die (e.g. when it is tied to another object or to the execution time of a function), it is worth considering if it can be invalidated and cleaned up manually at that point, rather than relying on a deallocator.

As a side effect, this can sometimes even lead to a better code design, e.g. when context managers can be used together with the with statement.

Borrowed references and data pointers

The memory management in PyPy is allowed to move objects around in memory. The C-API layer is only an indirect view on PyPy objects and often replicates data or state into C space that is then tied to the lifetime of a C-API object rather then the underlying PyPy object. It is important to understand that these two objects are separate things in cpyext.

The effect can be that when data pointers or borrowed references are used, and the owning object is no longer directly referenced from C space, the reference or data pointer may become invalid at some point, even if the object itself is still alive. As opposed to CPython, it is not enough to keep the reference to the object alive in a list (or other Python container), because the contents of those is only managed in Python space and thus only references the PyPy object. A reference in a Python container will not keep the C-API view on it alive. Entries in a Python class dict will obviously not work either.

One of the more visible places where this may happen is when accessing the **char*** buffer of a byte string. In PyPy, this will only work as long as the Cython code holds a direct reference to the byte string object itself.

Another point is when CPython C-API functions are used directly that return borrowed references, e.g. **PyTuple_GET_ITEM()** and similar functions, but also some functions that return borrowed references to built-in modules or low-level objects of the runtime environment. The GIL in PyPy only guarantees that the borrowed reference stays valid up to the next call into PyPy (or its C-API), but not necessarily longer.

When accessing the internals of Python objects or using borrowed references longer than up to the next call into PyPy, including

reference counting or anything that frees the GIL, it is therefore required to additionally keep direct owned references to these objects alive in C space, e.g. in local variables in a function or in the attributes of an extension type.

When in doubt, avoid using C-API functions that return borrowed references, or surround the usage of a borrowed reference explicitly by a pair of calls to Py_INCREF() when getting the reference and Py_DECREF() when done with it to convert it into an owned reference.

Builtin types, slots and fields

The following builtin types are not currently available in cpyext in form of their C level representation: PyComplexObject, PyFloatObject and PyBoolObject.

Many of the type slot functions of builtin types are not initialised in cpyext and can therefore not be used directly.

Similarly, almost none of the (implementation) specific struct fields of builtin types is exposed at the C level, such as the ob_digit field of **PyLongObject** or the allocated field of the **PyListObject** struct etc. Although the ob_size field of containers (used by the **Py_SIZE()** macro) is available, it is not guaranteed to be accurate.

It is best not to access any of these struct fields and slots and to use the normal Python types instead as well as the normal Python protocols for object operations. Cython will map them to an appropriate usage of the C-API in both CPython and cpyext.

Efficiency

Simple functions and especially macros that are used for speed in CPython may exhibit substantially different performance characteristics in cpyext.

Functions returning borrowed references were already mentioned as requiring special care, but they also induce substantially more runtime overhead because they often create weak references in PyPy where they only return a plain pointer in CPython. A visible example is PyTuple_GET_ITEM().

Some more high-level functions may also show entirely different performance characteristics, e.g. **PyDict_Next()** for dict iteration. While being the fastest way to iterate over a dict in CPython, having linear time complexity and a low overhead, it currently has quadratic runtime in PyPy because it maps to normal dict iteration, which cannot keep track of the current position between two calls and thus needs to restart the iteration on each call.

The general advice applies here even more than in CPython, that it is always best to rely on Cython generating appropriately adapted C-API handling code for you than to use the C-API directly - unless you really know what you are doing. And if you find a better way of doing something in PyPy and cpyext than Cython currently does, it's best to fix Cython for everyone's benefit.

Known problems

- As of PyPy 1.9, subtyping builtin types can result in infinite recursion on method calls in some rare cases.
- Docstrings of special methods are not propagated to Python space.

Bugs and crashes

The cpyext implementation in PyPy is much younger and substantially less mature than the well tested C-API and its underlying native implementation in CPython. This should be remembered when running into crashes, as the problem may not always be in your code or in Cython. Also, PyPy and its cpyext implementation are less easy to debug at the C level than CPython and Cython, simply because they were not designed for it.

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »

Limitations

This page used to list bugs in Cython that made the semantics of compiled code differ from that in Python. Most of the missing features have been fixed in Cython 0.15. The bug tracker has an up-to-date list of remaining compatibility issues. Note that a future version 1.0 of Cython is planned to provide full Python language compatibility.

Cython 0.19.1 documentation » Users Guide »

Cython 0.19.1 documentation » Users Guide »

Differences between Cython and Pyrex

Warning: Both Cython and Pyrex are moving targets. It has come to the point that an explicit list of all the differences between the two projects would be laborious to list and track, but hopefully this high-level list gives an idea of the differences that are present. It should be noted that both projects make an effort at mutual compatibility, but Cython's goal is to be as close to and complete as Python as reasonable.

Python 3 Support

Cython creates .c files that can be built and used with both Python 2.x and Python 3.x. In fact, compiling your module with Cython may very well be the easiest way to port code to Python 3. We are also working to make the compiler run in both Python 2.x and 3.x.

Many Python 3 constructs are already supported by Cython.

List/Set/Dict Comprehensions

Cython supports the different comprehensions defined by Python 3 for lists, sets and dicts:

```
[expr(x) for x in A] # list
{expr(x) for x in A} # set
{key(x) : value(x) for x in A} # dict
```

Looping is optimized if A is a list, tuple or dict. You can use the for ... from syntax, too, but it is generally preferred to use the usual for ... in range(...) syntax with a C run variable (e.g. cdef int i).

Note: see Automatic range conversion

Note that Cython also supports set literals starting from Python 2.3.

Keyword-only arguments

Python functions can have keyword-only arguments listed after the * parameter and before the ** parameter if any, e.g.:

def f(a, b, *args, c, d = 42, e, **kwds):
 ...

Here c, d and e cannot be passed as position arguments and must be passed as keyword arguments. Furthermore, c and e are required keyword arguments, since they do not have a default value.

If the parameter name after the ***** is omitted, the function will not accept any extra positional arguments, e.g.:

def g(a, b, *, c, d):

takes exactly two positional parameters and has two required keyword parameters.

Conditional expressions "x if b else y" (Python 2.5)

Conditional expressions as described in http://www.python.org/dev/peps/pep-0308/:

X if C else Y

Only one of x and Y is evaluated (depending on the value of C).

cdef inline

Module level functions can now be declared inline, with the inline keyword passed on to the C compiler. These can be as fast as macros.:

```
cdef inline int something_fast(int a, int b):
    return a*a + b
```

Note that class-level **cdef** functions are handled via a virtual function table, so the compiler won't be able to inline them in almost all cases.

Assignment on declaration (e.g. "cdef int spam = 5")

In Pyrex, one must write:

cdef int i, j, k
i = 2
j = 5
k = 7

Now, with cython, one can write:

cdef int i = 2, j = 5, k = 7

The expression on the right hand side can be arbitrarily complicated, e.g.:

cdef int n = python_call(foo(x,y), a + b + c) - 32

'by' expression in for loop (e.g. "for i from 0 <= i < 10 by 2")

```
for i from 0 <= i < 10 by 2:
    print i</pre>
```

yields:

0 2 4 6 8

Note: Usage of this syntax is discouraged as it is redundant with the normal Python **for** loop. See *Automatic range conversion*.
Boolean int type (e.g. it acts like a c int, but coerces to/from python as a boolean)

In C, ints are used for truth values. In python, any object can be used as a truth value (using the <u>__nonzero__()</u> method), but the canonical choices are the two boolean objects <u>True</u> and <u>False</u>. The <u>bint</u> (for "boolean int") type is compiled to a C int, but coerces to and from Python as booleans. The return type of comparisons and several builtins is a <u>bint</u> as well. This reduces the need for wrapping things in <u>bool()</u>. For example, one can write:

def is_equal(x):
 return x == y

which would return 1 or 0 in Pyrex, but returns True or False in Cython. One can declare variables and return values for functions to be of the **bint** type. For example:

cdef int i = x
cdef bint b = x

The first conversion would happen via $x._int_()$ whereas the second would happen via $x._nonzero_()$. (Actually, if x is the python object True or False then no method call is made.)

Executable class bodies

Including a working classmethod():

```
cdef class Blah:
    def some_method(self):
        print self
        some_method = classmethod(some_method)
        a = 2*3
        print "hi", a
```

cpdef functions

Cython adds a third function type on top of the usual def and cdef. If a function is declared cpdef it can be called from and overridden by both extension and normal python subclasses. You can essentially think of a cpdef method as a cdef method + some extras. (That's how it's implemented at least.) First, it creates a def method that does nothing but call the underlying cdef method (and does argument unpacking/coercion if needed). At the top of the cdef method a little bit of code is added to check to see if it's overridden. Specifically, in pseudocode:

```
if type(self) has a __dict__:
    foo = self.getattr('foo')
    if foo is not wrapper_foo:
        return foo(args)
[cdef method body]
```

To detect whether or not a type has a dictionary, it just checks the tp_dictoffset slot, which is NULL (by default) for extension types, but non- null for instance classes. If the dictionary exists, it does a single attribute lookup and can tell (by comparing pointers) whether or not the returned result is actually a new function. If, and only if, it is a new function, then the arguments packed into a tuple and the method called. This is all very fast. A flag is set so this lookup does not occur if one calls the method on the class directly, e.g.:

```
cdef class A:
    cpdef foo(self):
        pass
x = A()
x.foo() # will check to see if overridden
A.foo(x) # will call A's implementation whether overridden or n
```

See *Early Binding for Speed* for explanation and usage tips.

Automatic range conversion

This will convert statements of the form for i in range(...) to for i from ... when i is any cdef'd integer type, and the direction (i.e. sign of step) can be determined.

Warning: This may change the semantics if the range causes assignment to i to overflow. Specifically, if this option is set, an error will be raised before the loop is entered, whereas without this option the loop will execute until a overflowing value is encountered. If this effects you change Cython/Compiler/Options.py (eventually there will be a better way to set this).

More friendly type casting

In Pyrex, if one types <int>x where x is a Python object, one will get the memory address of x. Likewise, if one types <object>i where i is a C int, one will get an "object" at location i in memory. This leads to confusing results and segfaults.

In Cython <type>x will try and do a coercion (as would happen on assignment of x to a variable of type type) if exactly one of the types is a python object. It does not stop one from casting where there is no conversion (though it will emit a warning). If one really wants the address, cast to a void * first.

As in Pyrex <MyExtensionType>x will cast x to type MyExtensionType without any type checking. Cython supports the syntax <MyExtensionType?> to do the cast with type checking (i.e. it will throw an error if x is not a (subclass of) MyExtensionType.

Optional arguments in cdef/cpdef functions

Cython now supports optional arguments for cdef and cpdef functions.

The syntax in the .pyx file remains as in Python, but one declares such functions in the .pxd file by writing cdef foo(x=*). The number of arguments may increase on subclassing, but the argument types and order must remain the same. There is a slight performance penalty in some cases when a cdef/cpdef function without any optional is overridden with one that does have default argument values.

For example, one can have the .pxd file:

```
cdef class A:
    cdef foo(self)
cdef class B(A)
    cdef foo(self, x=*)
cdef class C(B):
    cpdef foo(self, x=*, int k=*)
```

with corresponding .pyx file:

```
cdef class A:
    cdef foo(self):
        print "A"
cdef class B(A)
    cdef foo(self, x=None)
        print "B", x
cdef class C(B):
    cpdef foo(self, x=True, int k=3)
        print "C", x, k
```

Note: this also demonstrates how **cpdef** functions can override **cdef** functions.

Function pointers in structs

Functions declared in **struct** are automatically converted to function pointers for convenience.

C++ Exception handling

cdef functions can now be declared as:

```
cdef int foo(...) except +
cdef int foo(...) except +TypeError
cdef int foo(...) except +python_error_raising_function
```

in which case a Python exception will be raised when a C++ error is caught. See Using C++ in Cython for more details.

Synonyms

cdef import from means the same thing as cdef extern from

Source code encoding

Cython supports PEP 3120 and PEP 263, i.e. you can start your Cython source file with an encoding comment and generally write your source code in UTF-8. This impacts the encoding of byte strings and the conversion of unicode string literals like <code>u'abcd'</code> to unicode objects.

Automatic typecheck

Rather than introducing a new keyword typecheck as explained in the Pyrex docs, Cython emits a (non-spoofable and faster) typecheck whenever isinstance() is used with an extension type as the second parameter.

From ____future___ directives

Cython supports several from __future__ directives, namely unicode_literals and division.

With statements are always enabled.

Pure Python mode

Cython has support for compiling .py files, and accepting type annotations using decorators and other valid Python syntax. This allows the same source to be interpreted as straight Python, or compiled for optimized results. See http://wiki.cython.org/pure for more details.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Typed Memoryviews

Typed memoryviews allow efficient access to memory buffers, such as those underlying NumPy arrays, without incurring any Python overhead. Memoryviews are similar to the current NumPy array buffer support (np.ndarray[np.float64_t, ndim=2]), but they have more features and cleaner syntax.

Memoryviews are more general than the old NumPy aray buffer support, because they can handle a wider variety of sources of array data. For example, they can handle C arrays and the Cython array type (*Cython arrays*).

A memoryview can be used in any context (function parameters, module-level, cdef class attribute, etc) and can be obtained from nearly any object that exposes writable buffer through the PEP 3118 buffer interface.

Quickstart

```
from cython.view cimport array as cvarray
import numpy as np
# Memoryview on a NumPy array
narr = np.arange(27, dtype=np.dtype("i")).reshape((3, 3, 3))
cdef int [:, :, :] narr_view = narr
# Memoryview on a C array
cdef int carr[3][3][3]
cdef int [:, :, :] carr_view = carr
# Memoryview on a Cython array
cyarr = cvarray(shape=(3, 3, 3), itemsize=sizeof(int), format="
cdef int [:, :, :] cyarr_view = cyarr
# Show the sum of all the arrays before altering it
print "NumPy sum of the NumPy array before assignments:", narr.
# We can copy the values from one memoryview into another using
# statement, by either indexing with ... or (NumPy-style) with
carr_view[...] = narr_view
cyarr_view[:] = narr_view
# NumPy-style syntax for assigning a single value to all elemen
narr_view[:, :, :] = 3
# Just to distinguish the arrays
carr_view[0, 0, 0] = 100
cyarr_view[0, 0, 0] = 1000
# Assigning into the memoryview on the NumPy array alters the 1
print "NumPy sum of NumPy array after assignments:", narr.sum()
# A function using a memoryview does not usually need the GIL
cpdef int sum3d(int[:, :, :] arr) nogil:
   cdef int total = 0
   I = arr.shape[0]
    J = arr.shape[1]
    K = arr.shape[2]
    for i in range(I):
        for j in range(J):
            for k in range(K):
                total += arr[i, j, k]
```

return total

A function accepting a memoryview knows how to use a NumPy ar # a C array, a Cython array... print "Memoryview sum of NumPy array is", sum3d(narr) print "Memoryview sum of C array is", sum3d(carr) print "Memoryview sum of Cython array is", sum3d(cyarr) # ... and of course, a memoryview. print "Memoryview sum of C memoryview is", sum3d(carr_view)

This code should give the following output:

```
NumPy sum of the NumPy array before assignments: 351
NumPy sum of NumPy array after assignments: 81
Memoryview sum of NumPy array is 81
Memoryview sum of C array is 451
Memoryview sum of Cython array is 1351
Memoryview sum of C memoryview is 451
```

Using memoryviews

Indexing and Slicing

Indexing and slicing can be done with or without the GIL. It basically works like NumPy. If indices are specified for every dimension you will get an element of the base type (e.g. *int*), otherwise you will get a new view. An Ellipsis means you get consecutive slices for every unspecified dimension:

```
cdef int[:, :, :] my_view = ...
# These are all equivalent
my_view[10]
my_view[10, :, :]
my_view[10, ...]
```

Copying

Memoryviews can be copied inplace:

```
cdef int[:, :, :] to_view, from_view
...
# copy the elements in from_view to to_view
to_view[...] = from_view
# or
to_view[:] = from_view
# or
to_view[:, :, :] = from_view
```

They can also be copied with the copy() and copy_fortran() methods; see *C* and Fortran contiguous copies.

Transposing

In most cases (see below), the memoryview can be transposed in the same way that NumPy slices can be transposed:

```
cdef int[:, ::1] c_contig = ...
cdef int[::1, :] f_contig = c_contig.T
```

This gives a new, transposed, view on the data.

Transposing requires that all dimensions of the memoryview have a direct access memory layout (i.e., there are no indirections through pointers). See *Specifying more general memory layouts* for details.

Newaxis

As for NumPy, new axes can be introduced by indexing an array with None

```
cdef double[:] myslice = np.linspace(0, 10, num=50)
# 2D array with shape (1, 50)
myslice[None] # or
myslice[None, :]
# 2D array with shape (50, 1)
myslice[:, None]
```

One may mix new axis indexing with all other forms of indexing and slicing. See also an example.

Comparison to the old buffer support

You will probably prefer memoryviews to the older syntax because:

- The syntax is cleaner
- Memoryviews do not usually need the GIL (see *Memoryviews* and the GIL)
- Memoryviews are considerably faster

For example, this is the old syntax equivalent of the sum3d function above:

```
cpdef int old_sum3d(object[int, ndim=3, mode='strided'] arr):
    cdef int I, J, K, total = 0
    I = arr.shape[0]
    J = arr.shape[1]
    K = arr.shape[2]
    for i in range(I):
        for j in range(J):
            for k in range(K):
                total += arr[i, j, k]
    return total
```

Note that we can't use nogil for the buffer version of the function as we could for the memoryview version of sum3d above, because buffer objects are Python objects. However, even if we don't use nogil with the memoryview, it is significantly faster. This is a output from an IPython session after importing both versions:

```
In [2]: import numpy as np
In [3]: arr = np.zeros((40, 40, 40), dtype=int)
In [4]: timeit -r15 old_sum3d(arr)
1000 loops, best of 15: 298 us per loop
In [5]: timeit -r15 sum3d(arr)
1000 loops, best of 15: 219 us per loop
```

Python buffer support

Cython memoryviews support nearly all objects exporting the interface of Python new style buffers. This is the buffer interface described in PEP 3118. NumPy arrays support this interface, as do *Cython arrays*. The "nearly all" is because the Python buffer interface allows the *elements* in the data array to themselves be pointers; Cython memoryviews do not yet support this.

Memory layout

The buffer interface allows objects to identify the underlying memory in a variety of ways. With the exception of pointers for data elements, Cython memoryviews support all Python new-type buffer layouts. It can be useful to know or specify memory layout if the memory has to be in a particular format for an external routine, or for code optimization.

Background

The concepts are as follows: there is data access and data packing. Data access means either direct (no pointer) or indirect (pointer). Data packing means your data may be contiguous or not contiguous in memory, and may use *strides* to identify the jumps in memory consecutive indices need to take for each dimension.

NumPy arrays provide a good model of strided direct data access, so we'll use them for a refresher on the concepts of C and Fortran contiguous arrays, and data strides.

Brief recap on C, Fortran and strided memory layouts

The simplest data layout might be a C contiguous array. This is the default layout in NumPy and Cython arrays. C contiguous means that the array data is continuous in memory (see below) and that neighboring elements in the first dimension of the array are furthest apart in memory, whereas neighboring elements in the last dimension are closest together. For example, in NumPy:

```
In [2]: arr = np.array([['0', '1', '2'], ['3', '4', '5']], dtyp
.▲]______
```

Here, arr[0, 0] and arr[0, 1] are one byte apart in memory, whereas arr[0, 0] and arr[1, 0] are 3 bytes apart. This leads us to the idea of *strides*. Each axis of the array has a stride length, which is the number of bytes needed to go from one element on this axis to the next element. In the case above, the strides for axes 0 and 1 will obviously be:

```
In [3]: arr.strides
Out[4]: (3, 1)
```

For a 3D C contiguous array:

```
In [5]: c_contig = np.arange(24, dtype=np.int8).reshape((2,3,4)
In [6] c_contig.strides
Out[6]: (12, 4, 1)
```

A Fortran contiguous array has the opposite memory ordering, with the elements on the first axis closest togther in memory:

```
In [7]: f_contig = np.array(c_contig, order='F')
In [8]: np.all(f_contig == c_contig)
Out[8]: True
In [9]: f_contig.strides
Out[9]: (1, 2, 6)
```

A contiguous array is one for which a single continuous block of memory contains all the data for the elements of the array, and therefore the memory block length is the product of number of elements in the array and the size of the elements in bytes. In the example above, the memory block is 2 * 3 * 4 * 1 bytes long, where 1 is the length of an int8.

An array can be contiguous without being C or Fortran order:

```
In [10]: c_contig.transpose((1, 0, 2)).strides
Out[10]: (4, 12, 1)
```

Slicing an NumPy array can easily make it not contiguous:

```
In [11]: sliced = c_contig[:,1,:]
In [12]: sliced.strides
Out[12]: (12, 1)
In [13]: sliced.flags
Out[13]:
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

Default behavior for memoryview layouts

As you'll see in *Specifying more general memory layouts*, you can specify memory layout for any dimension of an memoryview. For any dimension for which you don't specify a layout, then the data access is assumed to be direct, and the data packing assumed to be strided. For example, that will be the assumption for memoryviews like:

int [:, :, :] my_memoryview = obj

C and Fortran contiguous memoryviews

You can specify C and Fortran contiguous layouts for the memoryview by using the ::1 step syntax at definition. For example, if you know for sure your memoryview will be on top of a 3D C contiguous layout, you could write:

```
cdef int[:, :, ::1] c_contiguous = c_contig
```

where c_contig could be a C contiguous NumPy array. The ::1 at the 3rd position means that the elements in this 3rd dimension will be one element apart in memory. If you know you will have a 3D Fortran contiguous array: cdef int[::1, :, :] f_contiguous = f_contig

If you try to do this kind of thing:

```
# This array is C contiguous
c_contig = np.arange(24).reshape((2,3,4))
cdef int[:, :, ::1] c_contiguous = c_contig
# But this isn't
c_contiguous = np.array(c_contig, order='F')
```

you will get a valueError like this at runtime:

```
/Users/mb312/dev_trees/minimal-cython/mincy.pyx in init mincy (
    69
    70 # But this isn't
---> 71 c_contiguous = np.array(c_contig, order='F')
    72
    73 # Show the sum of all the arrays before altering it
/Users/mb312/dev_trees/minimal-cython/stringsource in View.Memo
/Users/mb312/dev_trees/minimal-cython/stringsource in View.Memo
ValueError: ndarray is not C-contiguous
```

Thus the ::1 in the slice type specification indicates in which dimension the data is contiguous. It can only be used to specify full C or Fortran contiguity.

C and Fortran contiguous copies

Copies can be made C or Fortran contiguous using the .copy() and .copy_fortran() methods:

```
# This view is C contiguous
cdef int[:, :, ::1] c_contiguous = myview.copy()
# This view is Fortran contiguous
```

Specifying more general memory layouts

Data layout can be specified using the previously seen ::1 slice syntax, or by using any of the constants in cython.view. If no specifier is given in any dimension, then the data access is assumed to be direct, and the data packing assumed to be strided. If you don't know whether a dimension will be direct or indirect (because you're getting an object with a buffer interface from some library perhaps), then you can specify the *generic* flag, in which case it will be determined at runtime.

The flags are as follows:

- generic strided and direct or indirect
- strided strided and direct (this is the default)
- indirect strided and indirect
- contiguous contiguous and direct
- indirect_contiguous the list of pointers is contiguous

and they can be used like this:

```
from cython cimport view
# direct access in both dimensions, strided in the first dimens.
cdef int[:, ::view.contiguous] a
# contiguous list of pointers to contiguous lists of ints
cdef int[::view.indirect_contiguous, ::1] b
# direct or indirect in the first dimension, direct in the seco
# strided in both dimensions
cdef int[::view.generic, :] c
```

Only the first, last or the dimension following an indirect dimension may be specified contiguous:

```
# INVALID
cdef int[::view.contiguous, ::view.indirect, :] a
cdef int[::1, ::view.indirect, :] b
# VALID
cdef int[::view.indirect, ::1, :] a
cdef int[::view.indirect, :, ::1] b
cdef int[::view.indirect_contiguous, ::1, :]
```

The difference between the *contiguous* flag and the *::1* specifier is that the former specifies contiguity for only one dimension, whereas the latter specifies contiguity for all following (Fortran) or preceding (C) dimensions:

```
cdef int[:, ::1] c_contig = ...
# VALID
cdef int[:, ::view.contiguous] myslice = c_contig[::2]
# INVALID
cdef int[:, ::1] myslice = c_contig[::2]
```

The former case is valid because the last dimension remains contiguous, but the first dimension does not "follow" the last one anymore (meaning, it was strided already, but it is not C or Fortran contiguous any longer), since it was sliced.

Memoryviews and the GIL

As you will see from the *Quickstart* section, memoryviews often do not need the GIL:

```
cpdef int sum3d(int[:, :, :] arr) nogil:
    ...
```

In particular, you do not need the GIL for memoryview indexing, slicing or transposing. Memoryviews require the GIL for the copy methods (*C and Fortran contiguous copies*), or when the dtype is object and an object element is read or written.

Memoryview Objects and Cython Arrays

These typed memoryviews can be converted to Python memoryview objects (*cython.view.memoryview*). These Python objects are indexable, slicable and transposable in the same way that the original memoryviews are. They can also be converted back to Cython-space memoryviews at any time.

They have the following attributes:

- shape
- strides
- suboffsets
- ndim
- size
- itemsize
- nbytes
- base

And of course the aforementioned **T** attribute (*Transposing*). These attributes have the same semantics as in NumPy. For instance, to retrieve the original object:

```
import numpy
cimport numpy as cnp
cdef cnp.int32_t[:] a = numpy.arange(10, dtype=numpy.int32)
a = a[::2]
print a, numpy.asarray(a), a.base
# this prints: <MemoryView of 'ndarray' object> [0 2 4 6 8] [0
]
```

Note that this example returns the original object from which the view was obtained, and that the view was resliced in the meantime.

Cython arrays

Whenever a Cython memoryview is copied (using any of the *copy* or *copy_fortran* methods), you get a new memoryview slice of a newly created cython.view.array object. This array can also be used manually, and will automatically allocate a block of data. It can later be assigned to a C or Fortran contiguous slice (or a strided slice). It can be used like:

```
from cython cimport view
my_array = view.array(shape=(10, 2), itemsize=sizeof(int), form
cdef int[:, :] my_slice = my_array
```

It also takes an optional argument *mode* ('c' or 'fortran') and a boolean *allocate_buffer*, that indicates whether a buffer should be allocated and freed when it goes out of scope:

```
cdef view.array my_array = view.array(..., mode="fortran", allo
my_array.data = <char *> my_data_pointer
# define a function that can deallocate the data (if needed)
my_array.callback_free_data = free
```

You can also cast pointers to array, or C arrays to arrays:

```
cdef view.array my_array = <int[:10, :2]> my_data_pointer
cdef view.array my_array = <int[:, :]> my_c_array
```

Of course, you can also immediately assign a cython.view.array to a typed memoryview slice. A C array may be assigned directly to a memoryview slice:

cdef int[:, ::1] myslice = my_2d_c_array

The arrays are indexable and slicable from Python space just like memoryview objects, and have the same attributes as memoryview objects.

CPython array module

An alternative to cython.view.array is the array module in the Python standard library. In Python 3, the array.array type supports the buffer interface natively, so memoryviews work on top of it without additional setup.

Starting with Cython 0.17, however, it is possible to use these arrays as buffer providers also in Python 2. This is done through explicitly cimporting the cpython.array module as follows:

```
cimport cpython.array
def sum_array(int[:] view):
    """
    >>> from array import array
    >>> sum_array( array('i', [1,2,3]) )
    6
    """
    cdef int total
    for i in range(view.shape[0]):
        total += view[i]
    return total
```

Note that the cimport also enables the old buffer syntax for the array type. Therefore, the following also works:

```
from cpython cimport array
def sum_array(array.array[int] arr): # using old buffer syntax
....
```

Coercion to NumPy

Memoryview (and array) objects can be coerced to a NumPy ndarray, without having to copy the data. You can e.g. do:

```
cimport numpy as np
import numpy as np
numpy_array = np.asarray(<np.int32_t[:10, :10]> my_pointer)
```

Of course, you are not restricted to using NumPy's type (such as np.int32_t here), you can use any usable type.

None Slices

Although memoryview slices are not objects they can be set to None and they can be be checked for being None as well:

```
def func(double[:] myarray = None):
    print myarray is None
```

If the function requires real memory views as input, it is therefore best to reject None input straight away in the signature, which is supported in Cython 0.17 and later as follows:

```
def func(double[:] myarray not None):
    ...
```

Unlike object attributes of extension classes, memoryview slices are not initialized to None.

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next
Using Parallelism

Cython supports native parallelism through the cython.parallel module. To use this kind of parallelism, the GIL must be released (see *Releasing the GIL*). It currently supports OpenMP, but later on more backends might be supported.

Note: Functionality in this module may only be used from the main thread or parallel regions due to OpenMP restrictions.

cython.parallel.prange([start,] stop[, step][, nogil=False][, schedule=None[, chunksize=None]][, num_threads=None])

This function can be used for parallel loops. OpenMP automatically starts a thread pool and distributes the work according to the schedule used. step must not be 0. This function can only be used with the GIL released. If nogil is true, the loop will be wrapped in a nogil section.

Thread-locality and reductions are automatically inferred for variables.

If you assign to a variable in a prange block, it becomes lastprivate, meaning that the variable will contain the value from the last iteration. If you use an inplace operator on a variable, it becomes a reduction, meaning that the values from the threadlocal copies of the variable will be reduced with the operator and assigned to the original variable after the loop. The index variable is always lastprivate. Variables assigned to in a parallel with block will be private and unusable after the block, as there is no concept of a sequentially last value.

The schedule is passed to OpenMP and can be one of the following:

static:

If a chunksize is provided, iterations are distributed to all threads ahead of time in blocks of the given chunksize. If no chunksize is given, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is assigned to each thread in advance.

This is most appropriate when the scheduling overhead matters and the problem can be cut down into equally sized chunks that are known to have approximately the same runtime.

dynamic:

The iterations are distributed to threads as they request them, with a default chunk size of 1.

This is suitable when the runtime of each chunk differs and is not known in advance and therefore a larger number of smaller chunks is used in order to keep all threads busy.

guided:

As with dynamic scheduling, the iterations are distributed to threads as they request them, but with decreasing chunk size. The size of each chunk is proportional to the number of unassigned iterations divided by the number of participating threads, decreasing to 1 (or the chunksize if provided).

This has an advantage over pure dynamic scheduling when it turns out that the last chunks take more time than expected or are otherwise being badly scheduled, so that most threads start running idle while the last chunks are being worked on by only a smaller number of threads.

runtime:

The schedule and chunk size are taken from the runtime

scheduling variable, which can be set through the openmp.omp_set_schedule() function call, or the OMP_SCHEDULE environment variable. Note that this essentially disables any static compile time optimisations of the scheduling code itself and may therefore show a slightly worse performance than when the same scheduling policy is statically configured at compile time.

```
cython.parallel.parallel(num_threads=None)
```

This directive can be used as part of a with statement to execute code sequences in parallel. This is currently useful to setup thread-local buffers used by a prange. A contained prange will be a worksharing loop that is not parallel, so any variable assigned to in the parallel section is also private to the prange. Variables that are private in the parallel block are unavailable after the parallel block.

Example with thread-local buffers:

```
from cython.parallel import parallel, prange
from libc.stdlib cimport abort, malloc, free
cdef Py_ssize_t idx, i, n = 100
cdef int * local buf
cdef size t size = 10
with nogil, parallel():
    local_buf = <int *> malloc(sizeof(int) * size)
    if local_buf == NULL:
        abort()
    # populate our local buffer in a sequential loop
    for i in xrange(size):
        local_buf[i] = i * 2
    # share the work using the thread-local buffer(s)
    for i in prange(n, schedule='guided'):
        func(local_buf)
    free(local_buf)
```

Later on sections might be supported in parallel blocks, to distribute code sections of work among threads.

cython.parallel.threadid()

Returns the id of the thread. For n threads, the ids will range from 0 to n-1.

Compiling

To actually use the OpenMP support, you need to tell the C or C++ compiler to enable OpenMP. For gcc this can be done as follows in a setup.py:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_module = Extension(
    "hello",
    ["hello.pyx"],
    extra_compile_args=['-fopenmp'],
    extra_link_args=['-fopenmp'],
)
setup(
    name = 'Hello world app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = [ext_module],
)
```

Breaking out of loops

The parallel with and prange blocks support the statements break, continue and return in nogil mode. Additionally, it is valid to use a with gil block inside these blocks, and have exceptions propagate from them. However, because the blocks use OpenMP, they can not just be left, so the exiting procedure is best-effort. For prange() this means that the loop body is skipped after the first break, return or exception for any subsequent iteration in any thread. It is undefined which value shall be returned if multiple different values may be returned, as the iterations are in no particular order:

```
from cython.parallel import prange
cdef int func(Py_ssize_t n):
    cdef Py_ssize_t i

    for i in prange(n, nogil=True):
        if i == 8:
            with gil:
                raise Exception()
    elif i == 4:
            break
    elif i == 2:
            return i
```

In the example above it is undefined whether an exception shall be raised, whether it will simply break or whether it will return 2.

Using OpenMP Functions

OpenMP functions can be used by cimporting openmp:



References

[1] http://www.openmp.org/mp-documents/spec30.pdf

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Users Guide »

previous | next

Debugging your Cython program

Cython comes with an extension for the GNU Debugger that helps users debug Cython code. To use this functionality, you will need to install gdb 7.2 or higher, built with Python support (linked to Python 2.5 or higher). The debugger supports debuggees with versions 2.6 and higher. For Python 3, code should be built with Python 3 and the debugger should be run with Python 2 (or at least it should be able to find the Python 2 Cython installation).

The debugger will need debug information that the Cython compiler can export. This can be achieved from within the setup script by passing pyrex_gdb=True to your Cython Extenion class:

```
from Cython.Distutils import extension
ext = extension.Extension('source', ['source.pyx'], pyrex_gdb=T
setup(..., ext_modules=[ext])
```

With this approach debug information can be enabled on a permodule basis. Another (easier) way is to simply pass the --pyrexgdb flag as a command line argument:

python setup.py build_ext --pyrex-gdb

For development it's often easy to use the --inplace flag also, which makes distutils build your project "in place", i.e., not in a separate *build* directory.

When invoking Cython from the command line directly you can have it write debug information using the --gdb flag:

```
cython --gdb myfile.pyx
```

Running the Debugger

To run the Cython debugger and have it import the debug information exported by Cython, run cygdb in the build directory:

```
$ python setup.py build_ext --pyrex-gdb --inplace
$ cygdb
GNU gdb (GDB) 7.2
...
(gdb)
```

When using the Cython debugger, it's preferable that you build and run your code with an interpreter that is compiled with debugging symbols (i.e. configured with --with-pydebug or compiled with the -g CFLAG). If your Python is installed and managed by your package manager you probably need to install debug support separately, e.g. for ubuntu:

```
$ sudo apt-get install python-dbg
$ python-dbg setup.py build_ext --pyrex-gdb --inplace
```

Then you need to run your script with python-dbg also.

You can also pass additional arguments to gdb:

\$ cygdb /path/to/build/directory/ GDBARGS

i.e.:

\$ cygdb . --args python-dbg mainscript.py

To tell cygdb not to import any debug information, supply -- as the first argument:

```
$ cygdb --
```

Using the Debugger

The Cython debugger comes with a set of commands that support breakpoints, stack inspection, source code listing, stepping, stepping over, etc. Most of these commands are analogous to their respective gdb command.

cy break breakpoints...

Break in a Python, Cython or C function. First it will look for a Cython function with that name, if cygdb doesn't know about a function (or method) with that name, it will set a (pending) C breakpoint. The -p option can be used to specify a Python breakpoint.

Breakpoints can be set for either the function or method name, or they can be fully "qualified", which means that the entire "path" to a function is given:

```
(gdb) cy break cython_function_or_method
(gdb) cy break packagename.cython_module.cython_function
(gdb) cy break packagename.cython_module.ClassName.cython_me
(gdb) cy break c_function
```

You can also break on Cython line numbers:

```
(gdb) cy break :14
(gdb) cy break cython_module:14
(gdb) cy break packagename.cython_module:14
```

Python breakpoints currently support names of the module (not the entire package path) and the function or method:

```
(gdb) cy break -p python_module.python_function_or_method
(gdb) cy break -p python_function_or_method
```

Note: Python breakpoints only work in Python builds where the Python frame information can be read from the debugger. To ensure this, use a Python debug build or a non-stripped build compiled with debug support.

cy step

Step through Python, Cython or C code. Python, Cython and C functions called directly from Cython code are considered relevant and will be stepped into.

cy next

Step over Python, Cython or C code.

cy run

Run the program. The default interpreter is the interpreter that was used to build your extensions with, or the interpreter cygdb is run with in case the "don't import debug information" option was in effect. The interpreter can be overridden using gdb's file command.

cy cont

Continue the program.

cy up

cy down

Go up and down the stack to what is considered a relevant frame.

cy finish

Execute until an upward relevant frame is met or something halts execution.

cy bt

cy backtrace

Print a traceback of all frames considered relevant. The -a option makes it print the full traceback (all C frames).

cy select

Select a stack frame by number as listed by cy backtrace. This command is introduced because cy backtrace prints a reversed stack trace, so frame numbers differ from gdb's bt.

cy print varname

Print a local or global Cython, Python or C variable (depending on the context). Variables may also be dereferenced:

```
(gdb) cy print x
x = 1
(gdb) cy print *x
*x = (PyObject) {
    _ob_next = 0x93efd8,
    _ob_prev = 0x93ef88,
    ob_refcnt = 65,
    ob_type = 0x83a3e0
}
```

cy set cython_variable = value

Set a Cython variable on the Cython stack to value.

cy list

List the source code surrounding the current line.

cy locals

cy globals

Print all the local and global variables and their values.

cy import FILE...

Import debug information from files given as arguments. The easiest way to import debug information is to use the cygdb command line tool.

cy exec code

Execute code in the current Python or Cython frame. This works like Python's interactive interpreter.

For Python frames it uses the globals and locals from the Python frame, for Cython frames it uses the dict of globals used on the Cython module and a new dict filled with the local Cython variables.

Note: cy exec modifies state and executes code in the debuggee and is therefore potentially dangerous.

Example:

```
(gdb) cy exec x + 1
2
(gdb) cy exec import sys; print sys.version_info
(2, 6, 5, 'final', 0)
(gdb) cy exec
>global foo
>
>foo = 'something'
>end
```

Convenience functions

The following functions are gdb functions, which means they can be used in a gdb expression.

cy_cname(varname)

Returns the C variable name of a Cython variable. For global variables this may not be actually valid.

cy_cvalue(varname)

Returns the value of a Cython variable.

cy_eval(expression)

Evaluates Python code in the nearest Python or Cython frame and returns the result of the expression as a gdb value. This gives a new reference if successful, NULL on error.

cy_lineno()

Returns the current line number in the selected Cython frame.

Example:

```
(gdb) print $cy_cname("x")
$1 = "__pyx_v_x"
(gdb) watch $cy_cvalue("x")
Hardware watchpoint 13: $cy_cvalue("x")
(gdb) cy set my_cython_variable = $cy_eval("{'spam': 'ham'}")
(gdb) print $cy_lineno()
$2 = 12
```

Configuring the Debugger

A few aspects of the debugger are configurable with gdb parameters. For instance, colors can be disabled, the terminal background color and breakpoint autocompletion can be configured.

cy_complete_unqualified

Tells the Cython debugger whether cy break should also complete plain function names, i.e. not prefixed by their module name. E.g. if you have a function named spam, in module M, it tells whether to only complete M.spam or also just spam.

The default is true.

cy_colorize_code

Tells the debugger whether to colorize source code. The default is true.

cy_terminal_background_color

Tells the debugger about the terminal background color, which affects source code coloring. The default is "dark", another valid option is "light".

This is how these parameters can be used:

```
(gdb) set cy_complete_unqualified off
(gdb) set cy_terminal_background_color light
(gdb) show cy_colorize_code
```

Cython 0.19.1 documentation » Users Guide »

previous | next

Cython 0.19.1 documentation » Reference Guide »

previous | next

Compilation

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

- A .pyx file is compiles by Cython to a .c file.
- The .c file is compiled by a C compiler to a .so file (or a .pyd file on Windows)

The following sub-sections describe several ways to build your extension modules, and how to pass directives to the Cython compiler.

Compiling from the command line

Run the Cython compiler command with your options and list of .pyx files to generate. For example:

```
$ cython -a yourmod.pyx
```

This creates a yourmod.c file, and the -a switch produces a generated html file. Pass the -h flag for a complete list of supported flags.

Compiling your .c files will vary depending on your operating system. Python documentation for writing extension modules should have some details for your system. Here we give an example on a Linux system:

```
$ gcc -shared -pthread -fPIC -fwrapv -02 -Wall -fno-strict-alia
-I/usr/include/python2.5 -o yourmod.so yourmod.c
```

[gcc will need to have paths to your included header files and paths to libraries you need to link with]

A yourmod.so file is now in the same directory and your module, yourmod, is available for you to import as you normally would.

Compiling with distutils

First, make sure that distutils package is installed in your system. It normally comes as part of the standard library. The following assumes a Cython file to be compiled called *hello.pyx*. Now, create a setup.py script:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    name = "My hello app",
    ext_modules = cythonize('hello.pyx'), # accepts a glob patt
)
```

Run the command python setup.py build_ext --inplace in your system's command shell and you are done. Import your new extension module into your python shell or script as normal.

The cythonize command also allows for multi-threaded compilation and dependency resolution. Recompilation will be skipped if the target file is up to date with its main source file and dependencies.

Compiling with pyximport

For generating Cython code right in your pure python module just type:

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

This allows you to automatically run Cython on every .pyx that Python is trying to import. You should use this for simple Cython builds only where no extra C libraries and no special building setup is needed.

In the case that Cython fails to compile a Python module, *pyximport* will fall back to loading the source modules instead.

It is also possible to compile new .py modules that are being imported (including the standard library and installed packages). For using this feature, just tell that to pyximport:

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

Compiling with cython.inline

One can also compile Cython in a fashion similar to SciPy's weave.inline. For example:

```
>>> import cython
>>> def f(a):
... ret = cython.inline("return a+b", b=3)
...
```

Unbound variables are automatically pulled from the surrounding local and global scopes, and the result of the compilation is cached for efficient re-use.

Compiling with Sage

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 are imported into the running session. Please check Sage documentation for details.

You can tailor the behavior of the Cython compiler by specifying the directives below.

Compiler directives

Compiler directives are instructions which affect the behavior of Cython code. Here is the list of currently supported directives:

boundscheck (True / False)

If set to False, Cython is free to assume that indexing operations ([]-operator) in the code will not cause any IndexErrors to be raised. Lists, tuples, and strings are affected only if the index can be determined to be non-negative (or if wraparound is False). Conditions which would normally trigger an IndexError may instead cause segfaults or data corruption if this is set to False. Default is True.

wraparound (True / False)

In Python arrays can be indexed relative to the end. For example A[-1] indexes the last value of a list. In C negative indexing is not supported. If set to False, Cython will neither check for nor correctly handle negative indices, possibly causing segfaults or data corruption. Default is True.

nonecheck (True / False)

If set to False, Cython is free to assume that native field accesses on variables typed as an extension type, or buffer accesses on a buffer variable, never occurs when the variable is set to None. Otherwise a check is inserted and the appropriate exception is raised. This is off by default for performance reasons. Default is False.

overflowcheck (True / False)

If set to True, raise errors on overflowing C integer arithmetic operations. Incurs a modest runtime penalty, but is much faster than using Python ints. Default is False.

```
overflowcheck.fold (True / False)
```

If set to True, and overflowcheck is True, check the overflow bit for nested, side-effect-free arithmetic expressions once rather than at every step. Depending on the compiler, architecture, and optimization settings, this may help or hurt performance. A simple suite of benchmarks can be found in Demos/overflow_perf.pyx. Default is True.

embedsignature (True / False)

If set to True, Cython will embed a textual copy of the call signature in the docstring of all Python visible functions and classes. Tools like IPython and epydoc can thus display the signature, which cannot otherwise be retrieved after compilation. Default is False.

cdivision (True / False)

If set to False, Cython will adjust the remainder and quotient operators C types to match those of Python ints (which differ when the operands have opposite signs) and raise a ZeroDivisionError when the right operand is 0. This has up to a 35% speed penalty. If set to True, no checks are performed. See CEP 516. Default is False.

cdivision_warnings (True / False)

If set to True, Cython will emit a runtime warning whenever division is performed with negative operands. See CEP 516. Default is False.

always_allow_keywords (True / False)

Avoid the METH_NOARGS and METH_O when constructing functions/methods which take zero or one arguments. Has no effect on special methods and functions with more than one argument. The METH_NOARGS and METH_O signatures provide faster calling conventions but disallow the use of keywords.

profile (True / False)

Add hooks for Python profilers into the compiled C code. Default is False.

linetrace (True / False)

Add line tracing hooks for Python profilers into the compiled C code. This also enables profiling. Default is False. Note that the generated module will not actually use line tracing, unless you additionally pass the C macro definition CYTHON_TRACE=1 to the C compiler (e.g. using the distutils option define_macros).

Note that this feature is currently EXPERIMENTAL. It will slow down your code, may not work at all for what you want to do with it, and may even crash arbitrarily.

```
infer_types (True / False)
```

Infer types of untyped variables in function bodies. Default is None, indicating that on safe (semantically-unchanging) inferences are allowed.

```
language_level (2/3)
```

Globally set the Python language level to be used for module compilation. Default is compatibility with Python 2. To enable Python 3 source code semantics, set this to 3 at the start of a module or pass the "-3" command line option to the compiler. Note that cimported and included source files inherit this setting from the module being compiled, unless they explicitly set their own language level.

c_string_type (bytes / str / unicode)

Globally set the type of an implicit coercion from char* or std::string.

c_string_encoding (ascii, default, utf-8, etc.)

Globally set the encoding to use when implicitly coercing char* or std:string to a unicode object. Coercion from a unicode object to C type is only allowed when set to ascii or default, the latter being utf-8 in Python 3 and nearly-always ascii in Python 2.

type_version_tag (True / False)

Enables the attribute cache for extension types in CPython by

setting the type flag $Py_TPFLAGS_HAVE_VERSION_TAG$. Default is True, meaning that the cache is enabled for Cython implemented types. To disable it explicitly in the rare cases where a type needs to juggle with its tp_dict internally without paying attention to cache consistency, this option can be set to False.

How to set directives

Globally

One can set compiler directives through a special header comment at the top of the file, like this:

#!python
#cython: boundscheck=False

The comment must appear before any code (but can appear after other comments or whitespace).

One can also pass a directive on the command line by using the -X switch:

\$ cython -X boundscheck=True ...

Directives passed on the command line will override directives set in header comments.

Locally

For local blocks, you need to cimport the special builtin cython module:

```
#!python
cimport cython
```

Then you can use the directives either as decorators or in a with statement, like this:



Warning: These two methods of setting directives are **not** affected by overriding the directive on the command-line using the -X option.

Cython 0.19.1 documentation » Reference Guide »

previous | next

Cython 0.19.1 documentation » Reference Guide »

previous | next

Language Basics

Cython File Types

There are three file types in Cython:

- Implementation files carry a .pyx suffix
- Definition files carry a .pxd suffix
- Include files which carry a .pxi suffix

Implementation File

What can it contain?

• Basically anything Cythonic, but see below.

What can't it contain?

• There are some restrictions when it comes to **extension types**, if the extension type is already defined else where... **more on this later**

Definition File

What can it contain?

- Any kind of C type declaration.
- extern C function or variable declarations.
- Declarations for module implementations.
- The definition parts of extension types.
- All declarations of functions, etc., for an external library

What can't it contain?

• Any non-extern C variable declaration.

- Implementations of C or Python functions.
- Python class definitions
- Python executable statements.
- Any declaration that is defined as **public** to make it accessible to other Cython modules.
 - This is not necessary, as it is automatic.
 - a **public** declaration is only needed to make it accessible to **external C code**.

What else?

cimport

- Use the **cimport** statement, as you would Python's import statement, to access these files from other definition or implementation files.
- **cimport** does not need to be called in .pyx file for for .pxd file that has the same name, as they are already in the same namespace.
- For cimport to find the stated definition file, the path to the file must be appended to the -I option of the **Cython compile command**.

compilation order

• When a .pyx file is to be compiled, Cython first checks to see if a corresponding .pxd file exits and processes it first.

Include File

What can it contain?

• Any Cythonic code really, because the entire file is textually

embedded at the location you prescribe.

How do I use it?

- Include the .pxi file with an include statement like: include "spamstuff.pxi
- The include statement can appear anywhere in your Cython file and at any indentation level
- The code in the .pxi file needs to be rooted at the "zero" indentation level.
- The included code can itself contain other include statements.

Declaring Data Types

As a dynamic language, Python encourages a programming style of considering classes and objects in terms of their methods and attributes, more than where they fit into the class hierarchy.

This can make Python a very relaxed and comfortable language for rapid development, but with a price - the 'red tape' of managing data types is dumped onto the interpreter. At run time, the interpreter does a lot of work searching namespaces, fetching attributes and parsing argument and keyword tuples. This run-time 'late binding' is a major cause of Python's relative slowness compared to 'early binding' languages such as C++.

However with Cython it is possible to gain significant speed-ups through the use of 'early binding' programming techniques.

Note: Typing is not a necessity

Providing static typing to parameters and variables is convenience to speed up your code, but it is not a necessity. Optimize where and when needed.

The cdef Statement

The cdef statement is used to make C level declarations for:

Variables:

```
cdef int i, j, k
cdef float f, g[42], *h
```

Structs:

```
cdef struct Grail:
    int age
    float volume
```

Unions:

cdef union Food: char *spam float *eggs

Enums:

```
cdef enum CheeseType:
    cheddar, edam,
    camembert
cdef enum CheeseState:
    hard = 1
    soft = 2
    runny = 3
```

Functions:

```
cdef int eggs(unsigned long l, float f):
...
```

Extension Types:

```
cdef class Spam:
```

Note: Constants

Constants can be defined by using an anonymous enum:

cdef enum: tons_of_spam = 3

Grouping cdef Declarations

A series of declarations can grouped into a cdef block:

```
cdef:

struct Spam:

    int tons

int i

    float f

    Spam *p

void f(Spam *s):

    print s.tons, "Tons of spam"
```

Note: ctypedef statement

The ctypedef statement is provided for naming types:

ctypedef unsigned long ULong

ctypedef int *IntPtr

Parameters

- Both C and Python **function** types can be declared to have parameters C data types.
- Use normal C declaration syntax:

```
def spam(int i, char *s):
    ...
    cdef int eggs(unsigned long l, float f):
    ...
```

- As these parameters are passed into a Python declared function, they are magically **converted** to the specified C type value.
 - This holds true for only numeric and string types
- If no type is specified for a parameter or a return value, it is assumed to be a Python object
 - The following takes two Python objects as parameters and returns a Python object:

```
cdef spamobjs(x, y):
```

Note: -

This is different then C language behavior, where it is an int by default.

- Python object types have reference counting performed according to the standard Python C-API rules:
 - Borrowed references are taken as parameters
 - New references are returned

Todo: link or label here the one ref count caveat for NumPy.

- The name object can be used to explicitly declare something as a Python Object.
 - For sake of code clarity, it recommended to always use object explicitly in your code.
 - This is also useful for cases where the name being declared would otherwise be taken for a type:

```
cdef foo(object int):
    ...
```

• As a return type:

```
cdef object foo(object int):
    ...
```

Todo: Do a see also here ..??

Optional Arguments

- Are supported for cdef and cpdef functions
- There differences though whether you declare them in a .pyx file or a .pxd file
 - When in a .pyx file, the signature is the same as it is in Python itself:

```
cdef class A:
    cdef foo(self):
        print "A"
cdef class B(A)
    cdef foo(self, x=None)
        print "B", x
cdef class C(B):
    cpdef foo(self, x=True, int k=3)
        print "C", x, k
```

 When in a .pxd file, the signature is different like this example: cdef foo(x=*):

```
cdef class A:
    cdef foo(self)
cdef class B(A)
    cdef foo(self, x=*)
cdef class C(B):
    cpdef foo(self, x=*, int k=*)
```

• The number of arguments may increase when subclassing, but the arg types and order must be the same.

• There may be a slight performance penalty when the optional arg is overridden with one that does not have default values.

Keyword-only Arguments

 As in Python 3, def functions can have keyword-only arguments listed after a "*" parameter and before a "**" parameter if any:

```
def f(a, b, *args, c, d = 42, e, **kwds):
    ...
```

- Shown above, the c, d and e arguments can not be passed as positional arguments and must be passed as keyword arguments.
- Furthermore, c and e are required keyword arguments since they do not have a default value.
- If the parameter name after the "*" is omitted, the function will not accept any extra positional arguments:

```
def g(a, b, *, c, d):
```

• Shown above, the signature takes exactly two positional parameters and has two required keyword parameters

Automatic Type Conversion

- For basic numeric and string types, in most situations, when a Python object is used in the context of a C value and vice versa.
- The following table summarizes the conversion possibilities, assuming sizeof(int) == sizeof(long):

C types	From Python types	To Python types
[unsigned] char	_	
[unsigned] short	int, long	int
int, long	_	
unsigned int		
unsigned long	int, long	long
[unsigned] long long	-	
float, double, long double	int, long, float	float
char *	str/bytes	str/bytes [1]
struct		dict

Note: Python String in a C Context

- A Python string, passed to C context expecting a char*, is only valid as long as the Python string exists.
- A reference to the Python string must be kept around for as long as the C string is needed.
- If this can't be guaranteed, then make a copy of the C string.
- Cython may produce an error message: Obtaining char* from a temporary Python value and will not resume compiling in situations like this:

```
cdef char *s
s = pystring1 + pystring2
```

- The reason is that concatenating to strings in Python produces a temporary variable.
 - The variable is decrefed, and the Python string deallocated as soon as the statement has finished,
 - Therefore the lvalue ``s`` is left dangling.
- The solution is to assign the result of the concatenation to a

Python variable, and then obtain the char* from that:

```
cdef char *s
p = pystring1 + pystring2
s = p
```

Note: It is up to you to be aware of this, and not to depend on Cython's error message, as it is not guaranteed to be generated for every situation.

Type Casting

• The syntax used in type casting are "<" and ">"

Note: The syntax is different from C convention

```
cdef char *p, float *q
p = <char*>q
```

 If one of the types is a python object for <type>x, Cython will try and do a coercion.

Note: Cython will not stop a casting where there is no conversion, but it will emit a warning.

• If the address is what is wanted, cast to a void* first.

Type Checking

- A cast like <MyExtensionType>x will cast x to type MyExtensionType without type checking at all.
- To have a cast type checked, use the syntax like: <MyExtensionType?>x.

- In this case, Cython will throw an error if "x" is not a (subclass) of MyExtensionType
- Automatic type checking for extension types can be obtained whenever isinstance() is used as the second parameter

Python Objects

Statements and Expressions

- For the most part, control structures and expressions follow Python syntax.
- When applied to Python objects, the semantics are the same unless otherwise noted.
- Most Python operators can be applied to C values with the obvious semantics.
- An expression with mixed Python and C values will have **conversions** performed automatically.
- Python operations are automatically checked for errors, with the appropriate action taken.

Differences Between Cython and C

- Most notable are C constructs which have no direct equivalent in Python.
 - An integer literal is treated as a C constant
 - It will be truncated to whatever size your C compiler thinks appropriate.
 - Cast to a Python object like this:

- The "L", "LL" and the "u" suffixes have the same meaning as in C
- There is no -> operator in Cython.. instead of p->x, use p.x.
- There is no * operator in Cython.. instead of *p, use p[0].
- & is permissible and has the same semantics as in C.
- NULL is the null C pointer.

- Do NOT use 0.
- NULL is a reserved word in Cython
- Syntax for Type casts are <type>value.

Scope Rules

- All determination of scoping (local, module, built-in) in Cython is determined statically.
- As with Python, a variable assignment which is not declared explicitly is implicitly declared to be a Python variable residing in the scope where it was assigned.

Note:

- Module-level scope behaves the same way as a Python local scope if you refer to the variable before assigning to it.
 - Tricks, like the following will NOT work in Cython:

• The above example will not work because True will always be looked up in the module-level scope. Do the following instead:

```
import __builtin__
try:
    True = __builtin__.True
except AttributeError:
    True = 1
```

Built-in Constants

Predefined Python built-in constants:

- None
- True
- False

Operator Precedence

• Cython uses Python precedence order, not C

For-loops

 range() is C optimized when the index value has been declared by cdef:

```
cdef i
for i in range(n):
    ...
```

• Iteration over C arrays is also permitted, e.g.:

```
cdef double x
cdef double* data
for x in data[:10]:
....
```

- Iterating over many builtin types such as lists and tuples is optimized.
- There is also a more C-style for-from syntax
 - The target expression must be a variable name.
 - The name between the lower and upper bounds must be the same as the target name.

```
for i from 0 <= i < n:
...
```

• Or when using a step size:

```
for i from 0 <= i < n by s:
    ...</pre>
```

• To reverse the direction, reverse the conditional operation:

```
for i from n > i >= 0:
    ...
```

- The break and continue are permissible.
- Can contain an else clause.

Functions and Methods

- There are three types of function declarations in Cython as the sub-sections show below.
- Only "Python" functions can be called outside a Cython module from *Python interpreted code*.

Callable from Python

- Are declared with the def statement
- Are called with Python objects
- Return Python objects
- See **Parameters** for special consideration

Callable from C

- Are declared with the cdef statement.
- Are called with either Python objects or C values.
- Can return either Python objects or C values.

Callable from both Python and C

- Are declared with the cpdef statement.
- Can be called from anywhere, because it uses a little Cython magic.
- Uses the faster C calling conventions when being called from other Cython code.

Overriding

cpdef functions can override cdef functions:

```
cdef class A:
    cdef foo(self):
        print "A"
cdef class B(A)
    cdef foo(self, x=None)
        print "B", x
cdef class C(B):
    cpdef foo(self, x=True, int k=3)
        print "C", x, k
```

Function Pointers

- Functions declared in a struct are automatically converted to function pointers.
- see using exceptions with function pointers

Python Built-ins

Cython compiles calls to most built-in functions into direct calls to the corresponding Python/C API routines, making them particularly fast.

Only direct function calls using these names are optimised. If you do something else with one of these names that assumes it's a Python object, such as assign it to a Python variable, and later call it, the call will be made as a Python function call.

Function and arguments	Return type	Python/C API Equivalent
abs(obj)	object, double,	PyNumber_Absolute, fabs, fabsf,
callable(obj)	bint	PyObject_Callable
delattr(obj, name)	None	PyObject_DelAttr
exec(code, [glob, [loc]])	object	•
dir(obj)	list	PyObject_Dir
divmod(a, b)	tuple	PyNumber_Divmod
getattr(obj, name, [default])	object	PyObject_GetAttr

(Note 1)		
hasattr(obj, name)	bint	PyObject_HasAttr
hash(obj)	int / long	PyObject_Hash
intern(obj)	object	Py*_InternFromString
isinstance(obj, type)	bint	PyObject_IsInstance
issubclass(obj, type)	bint	PyObject_IsSubclass
iter(obj, [sentinel])	object	PyObject_GetIter
len(obj)	Py_ssize_t	PyObject_Length
pow(x, y, [z])	object	PyNumber_Power
reload(obj)	object	PyImport_ReloadModule
repr(obj)	object	PyObject_Repr
setattr(obj, name)	void	PyObject_SetAttr

Note 1: Pyrex originally provided a function getattr3(obj, name, default)() corresponding to the three-argument form of the Python builtin getattr(). Cython still supports this function, but the usage is deprecated in favour of the normal builtin, which Cython can optimise in both forms.

Error and Exception Handling

- A plain cdef declared function, that does not return a Python object...
 - Has no way of reporting a Python exception to it's caller.
 - Will only print a warning message and the exception is ignored.
- In order to propagate exceptions like this to it's caller, you need to declare an exception value for it.
- There are three forms of declaring an exception for a C compiled program.
 - First:

```
cdef int spam() except -1:
...
```

- In the example above, if an error occurs inside spam, it will immediately return with the value of -1, causing an exception to be propagated to it's caller.
- Functions declared with an exception value, should explicitly prevent a return of that value.
- Second:

```
cdef int spam() except? -1:
....
```

- Used when a -1 may possibly be returned and is not to be considered an error.
- The "?" tells Cython that -1 only indicates a *possible* error.

- Now, each time -1 is returned, Cython generates a call to PyErr_Occurred to verify it is an actual error.
- Third:

```
cdef int spam() except *
```

• A call to PyErr_Occurred happens *every* time the function gets called.

Note: Returning void

A need to propagate errors when returning void must use this version.

- Exception values can only be declared for functions returning an..
 - integer
 - enum
 - float
 - pointer type
 - Must be a constant expression

Note:

Note: Function pointers

- Require the same exception value specification as it's user has declared.
- Use cases here are when used as parameters and when assigned to a variable:

```
int (*grail)(int, char *) except -1
```

Note: Python Objects

- Declared exception values are **not** need.
- Remember that Cython assumes that a function function without a declared return value, returns a Python object.
- Exceptions on such functions are implicitly propagated by returning NULL

Note: C++

 For exceptions from C++ compiled programs, see Wrapping C++ Classes

Checking return values for non-Cython functions..

• Do not try to raise exceptions by returning the specified value.. Example:

```
      cdef extern FILE *fopen(char *filename, char *mode) except

      ▲]
```

- The except clause does not work that way.
- It's only purpose is to propagate Python exceptions that have already been raised by either...
 - A Cython function
 - A C function that calls Python/C API routines.
- To propagate an exception for these circumstances you need to raise it yourself:

```
cdef FILE *p
p = fopen("spam.txt", "r")
if p == NULL:
    raise SpamError("Couldn't open the spam file")
```

Conditional Compilation

- The expressions in the following sub-sections must be valid compile-time expressions.
- They can evaluate to any Python value.
- The *truth* of the result is determined in the usual Python way.

Compile-Time Definitions

• Defined using the DEF statement:

```
DEF FavouriteFood = "spam"
DEF ArraySize = 42
DEF OtherArraySize = 2 * ArraySize + 17
```

- The right hand side must be a valid compile-time expression made up of either:
 - Literal values
 - Names defined by other DEF statements
- They can be combined using any of the Python expression syntax
- Cython provides the following predefined names
 - Corresponding to the values returned by os.uname()
 - UNAME_SYSNAME
 - UNAME_NODENAME
 - UNAME_RELEASE
 - UNAME_VERSION
 - UNAME_MACHINE
- A name defined by DEF can appear anywhere an identifier can

appear.

- Cython replaces the name with the literal value before compilation.
 - The compile-time expression, in this case, must evaluate to a Python value of int, long, float, or str:

```
cdef int a1[ArraySize]
cdef int a2[OtherArraySize]
print "I like", FavouriteFood
```

Conditional Statements

- · Similar semantics of the C pre-processor
- The following statements can be used to conditionally include or exclude sections of code to compile.
 - IF
 - ELIF
 - ELSE

```
IF UNAME_SYSNAME == "Windows":
    include "icky_definitions.pxi"
ELIF UNAME_SYSNAME == "Darwin":
    include "nice_definitions.pxi"
ELIF UNAME_SYSNAME == "Linux":
    include "penguin_definitions.pxi"
ELSE:
    include "other_definitions.pxi"
```

- ELIF and ELSE are optional.
- IF can appear anywhere that a normal statement or declaration can appear
- It can contain any statements or declarations that would be valid in that context.
 - This includes other IF and DEF statements



Cython 0.19.1 documentation » Reference Guide »

previous | next

Cython 0.19.1 documentation » Reference Guide »

previous | next

Extension Types

- Normal Python as well as extension type classes can be defined.
- Extension types:
 - Are considered by Python as "built-in" types.
 - Can be used to wrap arbitrary C-data structures, and provide a Python-like interface to them from Python.
 - Attributes and methods can be called from Python or Cython code
 - Are defined by the cdef class statement.

```
cdef class Shrubbery:
    cdef int width, height
    def __init__(self, w, h):
        self.width = w
        self.height = h
    def describe(self):
        print "This shrubbery is", self.width, \
            "by", self.height, "cubits."
```

Attributes

- Are stored directly in the object's C struct.
- Are fixed at compile time.
 - You can't add attributes to an extension type instance at run time like in normal Python.
 - You can sub-class the extension type in Python to add attributes at run-time.
- There are two ways to access extension type attributes:
 - By Python look-up.
 - Python code's only method of access.
 - By direct access to the C struct from Cython code.
 - Cython code can use either method of access, though.
- By default, extension type attributes are:
 - Only accessible by direct access.
 - Not accessible from Python code.
- To make attributes accessible to Python, they must be declared public Or readonly:

```
cdef class Shrubbery:
    cdef public int width, height
    cdef readonly float depth
```

- The width and height attributes are readable and writable from Python code.
- The depth attribute is readable but not writable.

Note:

Note: You can only expose simple C types, such as ints, floats, and strings, for Python access. You can also expose Python-valued attributes.

Note: The public and readonly options apply only to Python access, not direct access. All the attributes of an extension type are always readable and writable by C-level access.

Methods

- self is used in extension type methods just like it normally is in Python.
- See Functions and Methods; all of which applies here.

Properties

• Cython provides a special syntax:

```
cdef class Spam:
    property cheese:
        "A doc string can go here."
        def __get__(self):
            # This is called when the property is read.
            ...
        def __set__(self, value):
            # This is called when the property is written.
            ...
        def __del__(self):
            # This is called when the property is deleted.
```

- The __get__(), __set__(), and __de1__() methods are all optional.
 - If they are ommitted, An exception is raised when an access attempt is made.
- Below, is a full example that defines a property which can..
 - Add to a list each time it is written to ("__set__").
 - Return the list when it is read ("__get__").
 - Empty the list when it is deleted ("__del__").

```
# cheesy.pyx
cdef class CheeseShop:
    cdef object cheeses
    def __cinit__(self):
```

```
self.cheeses = []
    property cheese:
        def __get__(self):
            return "We don't have: %s" % self.cheeses
        def __set_(self, value):
            self.cheeses.append(value)
        def __del__(self):
            del self.cheeses[:]
# Test input
from cheesy import CheeseShop
shop = CheeseShop()
print shop.cheese
shop.cheese = "camembert"
print shop.cheese
shop.cheese = "cheddar"
print shop.cheese
del shop.cheese
print shop.cheese
```

```
# Test output
We don't have: []
We don't have: ['camembert']
We don't have: ['camembert', 'cheddar']
We don't have: []
```

Special Methods

Note:

- 1. The semantics of Cython's special methods are similar in principle to that of Python's.
- 2. There are substantial differences in some behavior.
- 3. Some Cython special methods have no Python counter-part.
- See the Special Methods Table for the many that are available.

Declaration

- Must be declared with def and cannot be declared with cdef.
- Performance is not affected by the def declaration because of special calling conventions

Docstrings

- Docstrings are not supported yet for some special method types.
- They can be included in the source, but may not appear in the corresponding <u>doc</u> attribute at run-time.
 - This a Python library limitation because the PyTypeObject data structure is limited

Initialization: __cinit__() and __init__()

- Any arguments passed to the extension type's constructor, will be passed to both initialization methods.
- _____() is where you should perform C-level initialization of

the object

- This includes any allocation of C data structures.
- **Caution** is warranted as to what you do in this method.
 - The object may not be fully valid Python object when it is called.
 - Calling Python objects, including the extensions own methods, may be hazardous.
- By the time __cinit__() is called...
 - Memory has been allocated for the object.
 - All C-level attributes have been initialized to 0 or null.
 - Python have been initialized to None, but you can not rely on that for each occasion.
 - This initialization method is guaranteed to be called exactly once.
- For Extensions types that inherit a base type:
 - The <u>_____()</u> method of the base type is automatically called before this one.
 - The inherited <u>_____()</u> method can not be called explicitly.
 - Passing modified argument lists to the base type must be done through __init__().
 - It may be wise to give the <u>__cinit__()</u> method both
 "*" and "**" arguments.
 - Allows the method to accept or ignore additional arguments.
 - Eliminates the need for a Python level sub-class, that changes the __init__() method's signature, to have to override both the __new__() and

__init__() methods.

- If <u>_____()</u> is declared to take no arguments except self, it will ignore any extra arguments passed to the constructor without complaining about a signature mismatch
- <u>__init__()</u> is for higher-level initialization and is safer for Python access.
 - By the time this method is called, the extension type is a fully valid Python object.
 - All operations are safe.
 - This method may sometimes be called more than once, or possibly not at all.
 - Take this into consideration to make sure the design of your other methods are robust of this fact.

Finalization: __dealloc__()

- This method is the counter-part to <u>_____()</u>.
- Any C-data that was explicitly allocated in the <u>_____()</u> method should be freed here.
- Use caution in this method:
 - The Python object to which this method belongs may not be completely intact at this point.
 - Avoid invoking any Python operations that may touch the object.
 - Don't call any of this object's methods.
 - It's best to just deallocate C-data structures here.
- All Python attributes of your extension type object are deallocated by Cython after the <u>______dealloc__()</u> method returns.

Arithmetic Methods

Note: Most of these methods behave differently than in Python

- There are not "reversed" versions of these methods... there is no __radd__() for instance.
- If the first operand cannot perform the operation, the same method of the second operand is called, with the operands in the same order.
- Do not rely on the first parameter of these methods, being "self" or the right type.
- The types of both operands should be tested before deciding what to do.
- Return NotImplemented for unhandled, mis-matched operand types.
- The previously mentioned points..
 - Also apply to 'in-place' method __ipow__().
 - Do not apply to other 'in-place' methods like <u>_iadd_()</u>, in that these always take self as the first argument.

Rich Comparisons

Note: There are no separate methods for individual rich comparison operations.

- A single special method called <u>__richcmp__()</u> replaces all the individual rich compare, special method types.
- <u>___richcmp__()</u> takes an integer argument, indicating which operation is to be performed as shown in the table below.

The __next__() Method

- Extension types used to expose an iterator interface should define a __next__() method.
- **Do not** explicitly supply a next() method, because Python does that for you automatically.

Subclassing

• An extension type may inherit from a built-in type or another extension type:

```
cdef class Parrot:
...
cdef class Norwegian(Parrot):
...
```

- A complete definition of the base type must be available to Cython
 - If the base type is a built-in type, it must have been previously declared as an extern extension type.
 - cimport can be used to import the base type, if the extern declared base type is in a .pxd definition file.
 - In Cython, multiple inheritance is not permitted.. singlular inheritance only
- Cython extension types can also be sub-classed in Python.
 - Here multiple inhertance is permissible as is normal for Python.
 - Even multiple extension types may be inherited, but Clayout of all the base classes must be compatible.

Forward Declarations

- Extension types can be "forward-declared".
- This is necessary when two extension types refer to each other:

```
cdef class Shrubbery # forward declaration
cdef class Shrubber:
    cdef Shrubbery work_in_progress
cdef class Shrubbery:
    cdef Shrubber creator
```

• An extension type that has a base-class, requires that both forward-declarations be specified:

```
cdef class A(B)
....
cdef class A(B):
    # attributes and methods
```

Extension Types and None

- Parameters and C-variables declared as an Extension type, may take the value of None.
- This is analogous to the way a C-pointer can take the value of NULL.

Note:

- 1. Exercise caution when using None
- 2. Read this section carefully.
- There is no problem as long as you are performing Python operations on it.
 - This is because full dynamic type checking is applied
- When accessing an extension type's C-attributes, **make sure** it is not None.
 - Cython does not check this for reasons of efficency.
- Be very aware of exposing Python functions that take extension types as arguments:

```
def widen_shrubbery(Shrubbery sh, extra_width): # This is
sh.width = sh.width + extra_width
* Users could **crash** the program by passing ``None`` for
* This could be avoided by::
    def widen_shrubbery(Shrubbery sh, extra_width):
        if sh is None:
            raise TypeError
        sh.width = sh.width + extra_width
```

```
* Cython provides a more convenient way with a ``not None``
    def widen_shrubbery(Shrubbery sh not None, extra_width)
        sh.width = sh.width + extra_width
* Now this function automatically checks that ``sh`` is not
4
```

- not None can only be used in Python functions (declared with def not cdef).
- For cdef functions, you will have to provide the check yourself.
- The self parameter of an extension type is guaranteed to **never** be None.
- When comparing a value x with None, and x is a Python object, note the following:
 - x is None and x is not None are very efficient.
 - They translate directly to C-pointer comparisons.
 - x == None and x != None Or if x: ... (a boolean condition), will invoke Python operations and will therefore be much slower.

Weak Referencing

- By default, weak references are not supported.
- It can be enabled by declaring a C attribute of the object type called __weakref__():

```
cdef class ExplodingAnimal:
    """This animal will self-destruct when it is
    no longer strongly referenced."""
    cdef object __weakref__
```
External and Public Types

Public

- When an extention type is declared public, Cython will generate a C-header (".h") file.
- The header file will contain the declarations for it's **object-struct** and it's **type-object**.
- External C-code can now access the attributes of the extension type.

External

- An extern extension type allows you to gain access to the internals of:
 - Python objects defined in the Python core.
 - Non-Cython extension modules
- The following example lets you get at the C-level members of Python's built-in "complex" object:

```
cdef extern from "complexobject.h":
    struct Py_complex:
        double real
        double imag
    ctypedef class __builtin__.complex [object PyComplexObj
        cdef Py_complex cval
# A function which uses the above type
def spam(complex c):
    print "Real:", c.cval.real
    print "Imag:", c.cval.imag
```

Note: Some important things in the example: #. ctypedef has been used because because Python's header file has the struct decalared with:

ctypedef struct {

```
} PyComplexObject;
```

- 1. The module of where this type object can be found is specified along side the name of the extension type. See **Implicit Importing**.
- 2. When declaring an external extension type...
 - Don't declare any methods, because they are Python method class the are not needed.
 - Similiar to **structs** and **unions**, extension classes declared inside a cdef extern from block only need to declare the C members which you will actually need to access in your module.

Name Specification Clause

Note: Only available to **public** and **extern** extension types.

• Example:

[object object_struct_name, type type_object_name]

- object_struct_name is the name to assume for the type's C-struct.
- type_object_name is the name to assume for the type's statically declared type-object.

- The object and type clauses can be written in any order.
- For cdef extern from declarations, This clause is required.
 - The object clause is required because Cython must generate code that is compatible with the declarations in the header file.
 - Otherwise the object clause is optional.
- For public extension types, both the object and type clauses **are required** for Cython to generate code that is compatible with external C-code.

Type Names vs. Constructor Names

- In a Cython module, the name of an extension type serves two distinct purposes:
 - 1. When used in an expression, it refers to a "module-level" global variable holding the type's constructor (i.e. it's type-object)
 - 2. It can also be used as a C-type name to declare a "type" for variables, arguments, and return values.
- Example:

. . .

```
cdef extern class MyModule.Spam:
```

- The name "Spam" serves both of these roles.
- Only "Spam" can be used as the type-name.
- The constructor can be referred to by other names.
- Upon an explicit import of "MyModule"...
 - MyModule.Spam() could be used as the constructor call.
 - MyModule.Spam could not be used as a type-name
- When an "as" clause is used, the name specified takes over both roles:

```
cdef extern class MyModule.Spam as Yummy:
```

- Yummy becomes both type-name and a name for the constructor.
- There other ways of course, to get hold of the constructor, but Yummy is the only usable type-name.

Interfacing with Other Code

C++

Fortran



Special Mention

Cython 0.19.1 documentation » Reference Guide »

Limitations

Cython 0.19.1 documentation » Reference Guide »

Compiler Directives

See Compilation.

Cython 0.19.1 documentation » Reference Guide »

previous

Special Methods Table

This table lists all of the special methods together with their parameter and return types. In the table below, a parameter name of self is used to indicate that the parameter has the type that the method belongs to. Other parameters with no type specified in the table are generic Python objects.

You don't have to declare your method as taking these parameter types. If you declare different types, conversions will be performed as necessary.

General

Name	Parameters	Return type	Description
cinit	self,		Basic initialisation (no direct Python equivalent)
init	self,		Further initialisation
dealloc	self		Basic deallocation (no direct Python equivalent)
cmp	х, у	int	3-way comparison
richcmp	x, y, int op	object	Rich comparison (no direct Python equivalent)
str	self	object	str(self)
repr	self	object	repr(self)
hash	self	int	Hash function
call	self,	object	self()
iter	self	object	Return iterator for sequence
getattr	self, name	object	Get attribute
getattribute	self, name	object	Get attribute, unconditionally
setattr	self, name, val		Set attribute
delattr	self, name		Delete attribute

Arithmetic operators

Name	Parameters	Return type	Description
add	х, у	object	binary + operator
sub	х, у	object	binary - operator
mul	х, у	object	* operator
div	х, у	object	/ operator for old-style division
floordiv	х, у	object	// operator
truediv	х, у	object	/ operator for new-style division
mod	х, у	object	% operator
divmod	х, у	object	combined div and mod
pow	X, Y, Z	object	<pre>** operator or pow(x, y, z)</pre>
neg	self	object	unary - operator
pos	self	object	unary + operator
abs	self	object	absolute value
nonzero	self	int	convert to boolean
invert	self	object	~ operator
lshift	х, у	object	<< operator
rshift	х, у	object	>> operator
and	х, у	object	& operator
or	х, у	object	operator
xor	х, у	object	^ operator

Numeric conversions

Name	Parameters	Return type	Description
int	self	object	Convert to integer
long	self	object	Convert to long integer
float	self	object	Convert to float
oct	self	object	Convert to octal
hex	self	object	Convert to hexadecimal
index (2.5+ only)	self	object	Convert to sequence index

In-place arithmetic operators

Name	Parameters	Return type	Description
iadd	self, x	object	+= operator
isub	self, x	object	-= operator
imul	self, x	object	*= operator
idiv	self, x	object	/= operator for old-style division
ifloordiv	self, x	object	//= operator
itruediv	self, x	object	/= operator for new-style division
imod	self, x	object	%= operator
ipow	X, Y, Z	object	**= operator
ilshift	self, x	object	<<= operator
irshift	self, x	object	>>= operator
iand	self, x	object	&= operator
ior	self, x	object	= operator
ixor	self, x	object	^= operator

Sequences and mappings

Name	Parameters	Return type	Description
len	self int		len(self)
getitem	self, x	object	self[x]
setitem	self, x, y		self[x] = y
delitem	self, x		del self[x]
getslice	self, Py_ssize_t i, Py_ssize_t j	object	self[i:j]
setslice	self, Py_ssize_t i, Py_ssize_t j, x		self[i:j] = x
delslice	self, Py_ssize_t i, Py_ssize_t j		del self[i:j]
contains	self, x	int	x in self

Iterators

Name	Parameters	Return type	Description
next	self	object	Get next item (called next in Python)

Buffer interface

Note: The buffer interface is intended for use by C code and is not directly accessible from Python. It is described in the Python/C API Reference Manual under sections 6.6 and 10.6.

Name	Parameters	Return type	Description
getreadbuffer	self, int i, void **p		
getwritebuffer	self, int i, void **p		
getsegcount	self, int *p		
getcharbuffer	self, int i, char **p		

Descriptor objects

Note: Descriptor objects are part of the support mechanism for new-style Python classes. See the discussion of descriptors in the Python documentation. See also PEP 252, "Making Types Look More Like Classes", and PEP 253, "Subtyping Built-In Types".

Name	Parameters	Return type	Description
get	self, instance, class	object	Get value of attribute
set	self, instance, value		Set value of attribute
delete	self, instance		Delete attribute

Cython 0.19.1 documentation » Reference Guide »

previous

Cython 0.19.1 documentation »

Getting Started

- Cython an overview
- Installing Cython
- Building Cython code
 - Building a Cython module using distutils
- Faster code via static typing
 - Typing Variables
 - Typing Functions
 - Determining where to add types

Cython 0.19.1 documentation »

Cython 0.19.1 documentation »

Tutorials

- Calling C functions
 - Dynamic linking
 - External declarations
 - Naming parameters
- Using C libraries
 - Defining external declarations
 - Writing a wrapper class
 - Memory management
 - Compiling and linking
 - Mapping functionality
 - Handling errors
 - Testing the result
 - Callbacks
- Extension types (aka. cdef classes)
- pxd files
- Caveats
- Profiling
 - Cython Profiling Basics
 - Profiling Tutorial
- Unicode and passing strings
 - General notes about C strings
 - Passing byte strings
 - · Dealing with "const"
 - Decoding bytes to text
 - Encoding text to bytes
 - C++ strings
 - Auto encoding and decoding
 - Source code encoding
 - Single bytes and characters
 - Narrow Unicode builds
 - Iteration

- Windows and wide character APIs
- Pure Python Mode
 - Magic Attributes
 - Augmenting .pxd
 - Types
 - Extension types and cdef functions
- Working with NumPy
 - Adding types
 - Efficient indexing
 - Tuning indexing further
 - More generic code
- Further reading
- Related work
- Appendix: Installing MinGW on Windows

Cython 0.19.1 documentation »

Cython 0.19.1 documentation »

Users Guide

Contents:

- Overview
 - About Cython
 - Future Plans
- Basic Tutorial
 - The Basics of Cython
 - Cython Hello World
 - pyximport: Cython Compilation the Easy Way
 - Fibonacci Fun
 - Primes
 - Language Details
- Language Basics
 - C variable and type definitions
 - Python functions vs. C functions
 - Automatic type conversions
 - Statements and expressions
 - The include statement
 - Conditional Compilation
- Extension Types
 - Introduction
 - Attributes
 - Type declarations
 - Extension types and None
 - Special methods
 - Properties
 - Subclassing
 - C methods
 - Forward-declaring extension types
 - Fast instantiation
 - Making extension types weak-referenceable

- Public and external extension types
- Public extension types
- Special Methods of Extension Types
 - Declaration
 - Docstrings
 - Initialisation methods: __cinit__() and __init__()
 - Finalization method: __dealloc__()
 - Arithmetic methods
 - Rich comparisons
 - The __next__() method
 - Special Method Table
- Sharing Declarations Between Cython Modules
 - Definition and Implementation files
 - What a Definition File contains
 - What an Implementation File contains
 - The cimport statement
 - Sharing C Functions
 - Sharing Extension Types
- Interfacing with External C Code
 - External declarations
 - Using Cython Declarations from C
- Source Files and Compilation
 - Basic setup.py
 - Cython Files Depending on C Files
 - Multiple Cython Files in a Package
 - Distributing Cython modules
 - Pyximport
- Early Binding for Speed
- Using C++ in Cython
 - Overview
 - A simple Tutorial
 - Advanced C++ features
 - Caveats and Limitations
- Fused Types (Templates)

- Quickstart
- Declaring Fused Types
- Using Fused Types
- Selecting Specializations
- Built-in Fused Types
- Casting Fused Functions
- Type Checking Specializations
- _____signatures____
- Porting Cython code to PyPy
 - Reference counts
 - Object lifetime
 - Borrowed references and data pointers
 - Builtin types, slots and fields
 - Efficiency
 - Known problems
 - Bugs and crashes
- Limitations
- Differences between Cython and Pyrex
 - Python 3 Support
 - Conditional expressions "x if b else y" (Python 2.5)
 - cdef inline
 - Assignment on declaration (e.g. "cdef int spam = 5")
 - 'by' expression in for loop (e.g. "for i from 0 <= i < 10 by 2")
 - Boolean int type (e.g. it acts like a c int, but coerces to/from python as a boolean)
 - Executable class bodies
 - cpdef functions
 - Automatic range conversion
 - More friendly type casting
 - Optional arguments in cdef/cpdef functions
 - Function pointers in structs
 - C++ Exception handling
 - Synonyms
 - Source code encoding

- Automatic typecheck
- From __future__ directives
- Pure Python mode
- Typed Memoryviews
 - Quickstart
 - Using memoryviews
 - Comparison to the old buffer support
 - Python buffer support
 - Memory layout
 - Memoryviews and the GIL
 - Memoryview Objects and Cython Arrays
 - Cython arrays
 - CPython array module
 - Coercion to NumPy
 - None Slices
- Using Parallelism
 - Compiling
 - Breaking out of loops
 - Using OpenMP Functions
- Debugging your Cython program
 - Running the Debugger
 - Using the Debugger
 - Convenience functions
 - Configuring the Debugger
Indices and tables

- Index
- Module Index
- Search Page

Cython 0.19.1 documentation »

previous | next

Cython 0.19.1 documentation »

previous | next

Reference Guide

Note:

Todo: Most of the **boldface** is to be changed to refs or other markup later.

Contents:

- Compilation
 - Compiling from the command line
 - Compiling with distutils
 - Compiling with pyximport
 - Compiling with cython.inline
 - Compiling with Sage
 - Compiler directives
- Language Basics
 - Cython File Types
 - Declaring Data Types
 - Statements and Expressions
 - Functions and Methods
 - Error and Exception Handling
 - Conditional Compilation
- Extension Types
 - Attributes
 - Methods
 - Properties
 - Special Methods
 - Subclassing
 - Forward Declarations
 - Extension Types and None
 - Weak Referencing
 - External and Public Types

- Type Names vs. Constructor Names
- Interfacing with Other Code
 - **C**
 - C++
 - Fortran
 - NumPy
- Special Mention
- Limitations
- Compiler Directives

Indices and tables

- Special Methods Table
 - General
 - Arithmetic operators
 - Numeric conversions
 - In-place arithmetic operators
 - Sequences and mappings
 - Iterators
 - Buffer interface
 - Descriptor objects
- Index
- Module Index
- Search Page

Cython 0.19.1 documentation »

previous | next

Cython 0.19.1 documentation »

Cython for NumPy users

This tutorial is aimed at NumPy users who have no experience with Cython at all. If you have some knowledge of Cython you may want to skip to the "Efficient indexing" section which explains the new improvements made in summer 2008.

The main scenario considered is NumPy end-use rather than NumPy/SciPy development. The reason is that Cython is not (yet) able to support functions that are generic with respect to datatype and the number of dimensions in a high-level fashion. This restriction is much more severe for SciPy development than more specific, "end-user" functions. See the last section for more information on this.

The style of this tutorial will not fit everybody, so you can also consider:

- Robert Bradshaw's slides on cython for SciPy2008 (a higherlevel and quicker introduction)
- Basic Cython documentation (see Cython front page).
- [:enhancements/buffer:Spec for the efficient indexing]

Note: The fast array access documented below is a completely new feature, and there may be bugs waiting to be discovered. It might be a good idea to do a manual sanity check on the C code Cython generates before using this for serious purposes, at least until some months have passed.

Cython at a glance

Cython is a compiler which compiles Python-like code files to C code. Still, "Cython is not a Python to C translator". That is, it doesn't take your full program and "turns it into C" – rather, the result makes full use of the Python runtime environment. A way of looking at it may be that your code is still Python in that it runs within the Python runtime environment, but rather than compiling to interpreted Python bytecode one compiles to native machine code (but with the addition of extra syntax for easy embedding of faster C-like code).

This has two important consequences:

- Speed. How much depends very much on the program involved though. Typical Python numerical programs would tend to gain very little as most time is spent in lower-level C that is used in a high-level fashion. However for-loop-style programs can gain many orders of magnitude, when typing information is added (and is so made possible as a realistic alternative).
- Easy calling into C code. One of Cython's purposes is to allow easy wrapping of C libraries. When writing code in Cython you can call into C code as easily as into Python code.

Some Python constructs are not yet supported, though making Cython compile all Python code is a stated goal (among the more important omissions are inner functions and generator functions).

Your Cython environment

Using Cython consists of these steps:

- 1. Write a .pyx source file
- 2. Run the Cython compiler to generate a C file
- 3. Run a C compiler to generate a compiled library
- 4. Run the Python interpreter and ask it to import the module

However there are several options to automate these steps:

- 1. The SAGE mathematics software system provides excellent support for using Cython and NumPy from an interactive command line (like IPython) or through a notebook interface (like Maple/Mathematica). See this documentation.
- 2. A version of pyximport is shipped with Cython, so that you can import pyx-files dynamically into Python and have them compiled automatically (See *Pyximport*).
- 3. Cython supports distutils so that you can very easily create build scripts which automate the process, this is the preferred method for full programs.
- 4. Manual compilation (see below)

Note: If using another interactive command line environment than SAGE, like IPython or Python itself, it is important that you restart the process when you recompile the module. It is not enough to issue an "import" statement again.

Installation

Unless you are used to some other automatic method: download Cython (0.9.8.1.1 or later), unpack it, and run the usual `python setup.py install. This will install a cython executable on your system. It is also possible to use Cython from the source directory without installing (simply launch cython.py in the root directory).

As of this writing SAGE comes with an older release of Cython than required for this tutorial. So if using SAGE you should download the newest Cython and then execute

```
$ cd path/to/cython-distro
$ path-to-sage/sage -python setup.py install
```

This will install the newest Cython into SAGE.

Manual compilation

As it is always important to know what is going on, I'll describe the manual method here. First Cython is run:

\$ cython yourmod.pyx

This creates yourmod.c which is the C source for a Python extension module. A useful additional switch is -a which will generate a document yourmod.html) that shows which Cython code translates to which C code line by line.

Then we compile the C file. This may vary according to your system, but the C file should be built like Python was built. Python documentation for writing extensions should have some details. On Linux this often means something like:

```
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-alia
▲
```

gcc should have access to the NumPy C header files so if they are not installed at /usr/include/numpy or similar you may need to pass another option for those.

This creates yourmod.so in the same directory, which is importable by Python by using a normal import yourmod statement.

The first Cython program

The code below does 2D discrete convolution of an image with a filter (and I'm sure you can do better!, let it serve for demonstration purposes). It is both valid Python and valid Cython code. I'll refer to it as both convolve_py.py for the Python version and convolve1.pyx for the Cython version – Cython uses ".pyx" as its file suffix.

```
from __future__ import division
import numpy as np
def naive_convolve(f, q):
    # f is an image and is indexed by (v, w)
    # g is a filter kernel and is indexed by (s, t),
    # it needs odd dimensions
    # h is the output image and is indexed by (x, y),
       it is not cropped
    #
    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter support
    # smid and tmid are number of pixels between the center pix
    # and the edge, ie for a 5x5 filter they will be 2.
    #
    # The output size is calculated by adding smid, tmid to eac
    # side of the dimensions of the input image.
    vmax = f.shape[0]
    wmax = f.shape[1]
    smax = g.shape[0]
    tmax = g.shape[1]
    smid = smax // 2
    tmid = tmax // 2
    xmax = vmax + 2*smid
    ymax = wmax + 2*tmid
    # Allocate result image.
    h = np.zeros([xmax, ymax], dtype=f.dtype)
    # Do convolution
    for x in range(xmax):
        for y in range(ymax):
            # Calculate pixel value for h at (x,y). Sum one com
            # for each pixel (s, t) of the filter g.
            s_from = max(smid - x, -smid)
            s_{to} = min((xmax - x) - smid, smid + 1)
            t_from = max(tmid - y, -tmid)
            t_t = min((ymax - y) - tmid, tmid + 1)
```

```
value = 0
for s in range(s_from, s_to):
    for t in range(t_from, t_to):
        v = x - smid + s
        w = y - tmid + t
        value += g[smid - s, tmid - t] * f[v, w]
        h[x, y] = value
return h
```

This should be compiled to produce yourmod.so (for Linux systems). We run a Python session to test both the Python version (imported from .py-file) and the compiled Cython module.

```
In [1]: import numpy as np
In [2]: import convolve_py
In [3]: convolve_py.naive_convolve(np.array([[1, 1, 1]], dtype=
        np.array([[1],[2],[1]], dtype=np.int))
. . .
Out [3]:
array([[1, 1, 1],
    [2, 2, 2],
    [1, 1, 1]])
In [4]: import convolve1
In [4]: convolve1.naive_convolve(np.array([[1, 1, 1]], dtype=np
        np.array([[1],[2],[1]], dtype=np.int))
. . .
Out [4]:
array([[1, 1, 1],
    [2, 2, 2],
    [1, 1, 1])
In [11]: N = 100
In [12]: f = np.arange(N*N, dtype=np.int).reshape((N,N))
In [13]: g = np.arange(81, dtype=np.int).reshape((9, 9))
In [19]: %timeit -n2 -r3 convolve_py.naive_convolve(f, g)
2 loops, best of 3: 1.86 s per loop
In [20]: %timeit -n2 -r3 convolve1.naive_convolve(f, g)
2 loops, best of 3: 1.41 s per loop
•
                                                               ۲
```

There's not such a huge difference yet; because the C code still does exactly what the Python interpreter does (meaning, for instance, that a new object is allocated for each number used). Look at the generated html file and see what is needed for even the simplest statements you get the point quickly. We need to give Cython more information; we need to add types.

Adding types

To add types we use custom Cython syntax, so we are now breaking Python source compatibility. Here's convolve2.pyx. Read the comments!

```
from __future__ import division
import numpy as np
# "cimport" is used to import special compile-time information
# about the numpy module (this is stored in a file numpy.pxd wh
# currently part of the Cython distribution).
cimport numpy as np
# We now need to fix a datatype for our arrays. I've used the v
# DTYPE for this, which is assigned to the usual NumPy runtime
# type info object.
DTYPE = np.int
# "ctypedef" assigns a corresponding compile-time type to DTYPE
# every type in the numpy module there's a corresponding compil
# type with a _t-suffix.
ctypedef np.int t DTYPE t
# The builtin min and max functions works with Python objects,
# so very slow. So we create our own.
# - "cdef" declares a function which has much less overhead th
     def function (but it is not Python-callable)
#
# - "inline" is passed on to the C compiler which may inline t
# - The C type "int" is chosen as return type and argument typ
# - Cython allows some newer Python constructs like "a if x el
     the resulting C file compiles with Python 2.3 through to P
#
cdef inline int int_max(int a, int b): return a if a >= b else
cdef inline int int_min(int a, int b): return a if a <= b else</pre>
# "def" can type its arguments but not have a return type. The
# arguments for a "def" function is checked at run-time when en
# function.
#
# The arrays f, g and h is typed as "np.ndarray" instances. The
# this has is to a) insert checks that the function arguments r
# NumPy arrays, and b) make some attribute access like f.shape[
# more efficient. (In this example this doesn't matter though.)
def naive_convolve(np.ndarray f, np.ndarray g):
    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter support
    assert f.dtype == DTYPE and g.dtype == DTYPE
    # The "cdef" keyword is also used within functions to type
```

```
# can only be used at the top indendation level (there are
# problems with allowing them in other places, though we'd
# good and thought out proposals for it).
#
# For the indices, the "int" type is used. This corresponds
# other C types (like "unsigned int") could have been used
# Purists could use "Py_ssize_t" which is the proper Python
# array indices.
cdef int vmax = f.shape[0]
cdef int wmax = f.shape[1]
cdef int smax = g.shape[0]
cdef int tmax = g.shape[1]
cdef int smid = smax // 2
cdef int tmid = tmax // 2
cdef int xmax = vmax + 2*smid
cdef int ymax = wmax + 2*tmid
cdef np.ndarray h = np.zeros([xmax, ymax], dtype=DTYPE)
cdef int x, y, s, t, v, w
# It is very important to type ALL your variables. You do n
# warnings if not, only much slower code (they are implicit.
# Python objects).
cdef int s_from, s_to, t_from, t_to
# For the value variable, we want to use the same data type
# stored in the array, so we use "DTYPE t" as defined above
# NB! An important side-effect of this is that if "value" o
# datatype size, it will simply wrap around like in C, rath
# an error like in Python.
cdef DTYPE t value
for x in range(xmax):
    for y in range(ymax):
        s_from = int_max(smid - x, -smid)
        s_{to} = int_min((xmax - x) - smid, smid + 1)
        t_from = int_max(tmid - y, -tmid)
        t_t = int_min((ymax - y) - tmid, tmid + 1)
        value = 0
        for s in range(s from, s to):
            for t in range(t_from, t_to):
                v = x - smid + s
                w = y - tmid + t
                value += g[smid - s, tmid - t] * f[v, w]
        h[x, y] = value
return h
                                                           .
```

At this point, have a look at the generated C code for convolve1.pyx and convolve2.pyx. Click on the lines to expand them and see

•

corresponding C. (Note that this code annotation is currently experimental and especially "trailing" cleanup code for a block may stick to the last expression in the block and make it look worse than it is – use some common sense).

•

•

Especially have a look at the for loops: In convolve1.c, these are ~20 lines of C code to set up while in convolve2.c a normal C for loop is used.

After building this and continuing my (very informal) benchmarks, I get:

In [21]: import convolve2
In [22]: %timeit -n2 -r3 convolve2.naive_convolve(f, g)
2 loops, best of 3: 828 ms per loop

Efficient indexing

There's still a bottleneck killing performance, and that is the array lookups and assignments. The []-operator still uses full Python operations – what we would like to do instead is to access the data buffer directly at C speed.

What we need to do then is to type the contents of the ndarray objects. We do this with a special "buffer" syntax which must be told the datatype (first argument) and number of dimensions ("ndim" keyword-only argument, if not provided then one-dimensional is assumed).

More information on this syntax [:enhancements/buffer:can be found here].

Showing the changes needed to produce convolve3.pyx only:

```
def naive_convolve(np.ndarray[DTYPE_t, ndim=2] f, np.ndarray[DT
...
cdef np.ndarray[DTYPE_t, ndim=2] h = ...
```

Usage:

```
In [18]: import convolve3
In [19]: %timeit -n3 -r100 convolve3.naive_convolve(f, g)
3 loops, best of 100: 11.6 ms per loop
```

Note the importance of this change.

Gotcha: This efficient indexing only affects certain index operations, namely those with exactly ndim number of typed integer indices. So if v for instance isn't typed, then the lookup f[v, w] isn't optimized. On

the other hand this means that you can continue using Python objects for sophisticated dynamic slicing etc. just as when the array is not typed.

Tuning indexing further

The array lookups are still slowed down by two factors:

- 1. Bounds checking is performed.
- 2. Negative indices are checked for and handled correctly. The code above is explicitly coded so that it doesn't use negative indices, and it (hopefully) always access within bounds. We can add a decorator to disable bounds checking:



Now bounds checking is not performed (and, as a side-effect, if you "do" happen to access out of bounds you will in the best case crash your program and in the worst case corrupt data). It is possible to switch bounds-checking mode in many ways, see [:docs/compilerdirectives:compiler directives] for more information.

Negative indices are dealt with by ensuring Cython that the indices will be positive, by casting the variables to unsigned integer types (if you do have negative values, then this casting will create a very large positive value instead and you will attempt to access out-of-bounds values). Casting is done with a special <>-syntax. The code below is changed to use either unsigned ints or casting as appropriate:

```
...
cdef int s, t
cdef unsigned int x, y, v, w
cdef int s_from, s_to, t_from, t_to
```

```
cdef DTYPE_t value
for x in range(xmax):
    for y in range(ymax):
         s_from = max(smid - x, -smid)
         s_{to} = min((xmax - x) - smid, smid + 1)
         t_from = max(tmid - y, -tmid)
         t_to = min((ymax - y) - tmid, tmid + 1)
         value = 0
         for s in range(s_from, s_to):
             for t in range(t_from, t_to):
                  v = \langle unsigned int \rangle (x - smid + s)
                  w = \langle unsigned int \rangle (y - tmid + t)
                  value += g[<unsigned int>(smid - s), <unsigned</pre>
         h[x, y] = value
4
```

(In the next Cython release we will likely add a compiler directive or argument to the np.ndarray[]-type specifier to disable negative indexing so that casting so much isn't necessary; feedback on this is welcome.)

The function call overhead now starts to play a role, so we compare the latter two examples with larger N:

```
In [11]: %timeit -n3 -r100 convolve4.naive_convolve(f, g)
3 loops, best of 100: 5.97 ms per loop
In [12]: N = 1000
In [13]: f = np.arange(N*N, dtype=np.int).reshape((N,N))
In [14]: g = np.arange(81, dtype=np.int).reshape((9, 9))
In [17]: %timeit -n1 -r10 convolve3.naive_convolve(f, g)
1 loops, best of 10: 1.16 s per loop
In [18]: %timeit -n1 -r10 convolve4.naive_convolve(f, g)
1 loops, best of 10: 597 ms per loop
```

(Also this is a mixed benchmark as the result array is allocated within the function call.)

Warning: Speed comes with some cost. Especially it can be dangerous to set typed objects (like f, g and h in our sample code) to None. Setting such objects to None is entirely legal, but all you

can do with them is check whether they are None. All other use (attribute lookup or indexing) can potentially segfault or corrupt data (rather than raising exceptions as they would in Python).

The actual rules are a bit more complicated but the main message is clear: Do not use typed objects without knowing that they are not set to None.

More generic code

It would be possible to do:

```
def naive_convolve(object[DTYPE_t, ndim=2] f, ...):
```

i.e. use **object** rather than **np.ndarray**. Under Python 3.0 this can allow your algorithm to work with any libraries supporting the buffer interface; and support for e.g. the Python Imaging Library may easily be added if someone is interested also under Python 2.x.

There is some speed penalty to this though (as one makes more assumptions compile-time if the type is set to np.ndarray, specifically it is assumed that the data is stored in pure strided more and not in indirect mode).

[:enhancements/buffer:More information]

The future

These are some points to consider for further development. All points listed here has gone through a lot of thinking and planning already; still they may or may not happen depending on available developer time and resources for Cython.

- 1. Support for efficient access to structs/records stored in arrays; currently only primitive types are allowed.
- 2. Support for efficient access to complex floating point types in arrays. The main obstacle here is getting support for efficient complex datatypes in Cython.
- 3. Calling NumPy/SciPy functions currently has a Python call overhead; it would be possible to take a short-cut from Cython directly to C. (This does however require some isolated and incremental changes to those libraries; mail the Cython mailing list for details).
- 4. Efficient code that is generic with respect to the number of dimensions. This can probably be done today by calling the NumPy C multi-dimensional iterator API directly; however it would be nice to have for-loops over enumerate() and ndenumerate() on NumPy arrays create efficient code.
- 5. A high-level construct for writing type-generic code, so that one can write functions that work simultaneously with many datatypes. Note however that a macro preprocessor language can help with doing this for now.

Cython 0.19.1 documentation »