Cython Reference Guide

Release 0.20dev

Stefan Behnel, Robert Bradshaw, William Stein
Gary Furnish, Dag Seljebotn, Greg Ewing
Gabriel Gellner, editor

December 13, 2013

CONTENTS

Note:

Todo

Most of the **boldface** is to be changed to refs or other markup later.

Contents:

CHAPTER

ONE

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.

1.1 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 -O2 -Wall -fno-strict-aliasing \
      -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.

1.2 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 pattern
)
```

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.

1.2.1 Configuring the C-Build

If you have include files in non-standard places you can pass an `include_path` parameter to `cythonize`:

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

setup(
    name = "My hello app",
    ext_modules = cythonize("src/*.pyx", include_path = [...]),
)
```

Often, Python packages that offer a C-level API provide a way to find the necessary include files, e.g. for NumPy:

```
include_path = [numpy.get_include()]
```

If you need to specify compiler options, libraries to link with or other linker options you will need to create `Extension` instances manually (note that glob syntax can still be used to specify multiple extensions in one line):

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

extensions = [
    Extension("primes", ["primes.pyx"],
        include_dirs = [...],
        libraries = [...],
        library_dirs = [...]),
    # Everything but primes.pyx is included here.
    Extension("*", ["*.pyx"],
        include_dirs = [...],
        libraries = [...],
        library_dirs = [...]),
]
setup(
    name = "My hello app",
    ext_modules = cythonize(extensions),
)
```

If your options are static (for example you do not need to call a tool like `pkg-config` to determine them) you can also provide them directly in your .pyx source file using a special comment block at the start of the file:

```
# distutils: libraries = spam eggs
# distutils: include_dirs = /opt/food/include
```

If you have some C files that have been wrapped with Cython and you want to compile them into your extension, you can define the distutils `sources` parameter:

```
# distutils: sources = helper.c, another_helper.c
```

Note that these sources are added to the list of sources of the current extension module. Spelling this out in the `setup.py` file looks as follows:

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

sourcefiles = ['example.pyx', 'helper.c', 'another_helper.c']
```

```
extensions = [Extension("example", sourcefiles)]

setup(
    ext_modules = cythonize(extensions)
)
```

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.

1.2.2 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/she probably doesn't want to use it just to install your module. Also, the installed version 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"])]
)
```

This is easy to combine with `cythonize()` by changing the file extension of the extension module sources:

```
from distutils.core import setup
from distutils.extension import Extension

USE_CYTHON = ...   # command line option, try-import, ...

ext = '.pyx' if USE_CYTHON else '.c'

extensions = [Extension("example", ["example"+ext])]

if USE_CYTHON:
    from Cython.Build import cythonize
    extensions = cythonize(extensions)

setup(
    ext_modules = extensions
)
```

1.3 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)
```

1.4 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.

1.5 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.

1.6 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.

1.6.1 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:

```
#!python
@cython.boundscheck(False) # turn off boundscheck for this function
def f():
    ...
    with cython.boundscheck(True): # turn it temporarily on again for this block
        ...
```

Warning: These two methods of setting directives are **not** affected by overriding the directive on the command-line using the -X option.

CHAPTER

TWO

LANGUAGE BASICS

2.1 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

2.1.1 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**

2.1.2 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.

2.1.3 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.

2.2 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 Pythons 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.

2.2.1 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
```

2.2.2 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
```

2.2.3 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.

2.2.4 Keyword-only Arguments

- As in Python 3, `def` functions can have keyword-only argurments 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

2.2.5 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.

2.2.6 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.

[1] The conversion is to/from str for Python 2.x, and bytes for Python 3.x.

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

2.2.7 Python Objects

2.3 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.

2.3.1 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:

    ```
    <object>10000000000000000000
    ```

  - 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`.

2.3.2 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:

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

- 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
```

2.3.3 Built-in Constants

Predefined Python built-in constants:

- None
- True
- False

2.3.4 Operator Precedence

- Cython uses Python precedence order, not C

2.3.5 For-loops

The "for ... in iterable" loop works as in Python, but is even more versatile in Cython as it can additionally be used on C types.

- `range()` is C optimized when the index value has been declared by `cdef`, for example:

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

- Iteration over C arrays and sliced pointers is supported and automatically infers the type of the loop variable, e.g.:

```
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 verbose C-style for-from syntax which, however, is deprecated in favour of the normal Python "for ... in range()" loop. You might still find it in legacy code that was written for Pyrex, though.
- The target expression must be a plain 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:
    ...
```

- To reverse the direction, reverse the conditional operation:

```
for i from n > i >= 0:
    ...
```

- The `break` and `continue` statements are permissible.
- Can contain an else clause.

2.4 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*.

2.4.1 Callable from Python

- Are declared with the `def` statement
- Are called with Python objects
- Return Python objects
- See **Parameters** for special consideration

2.4.2 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.

2.4.3 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.

2.4.4 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
```

2.4.5 Function Pointers

- Functions declared in a `struct` are automatically converted to function pointers.
- see **using exceptions with function pointers**

2.4.6 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]) (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.

2.5 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**

2.5.1 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 NULL # WRONG!
```

  - 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")
```

2.6 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.

2.6.1 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
```

2.6.2 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

CHAPTER

THREE

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."
```

3.1 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 extenstion 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.

3.2 Methods

- `self` is used in extension type methods just like it normally is in Python.
- See **Functions and Methods**; all of which applies here.

3.3 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 `__del__()` 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: []
```

3.4 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.

3.4.1 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

3.4.2 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 `__doc__` attribute at run-time.
- This a Python library limitation because the `PyTypeObject` data structure is limited

3.4.3 Initialization: `__cinit__()` and `__init__()`

- Any arguments passed to the extension type's constructor, will be passed to both initialization methods.
- `__cinit__()` 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 `__cinit__()` method of the base type is automatically called before this one.
    - The inherited `__cinit__()` 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 `__cinit__()` 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 `__cinit__()` is declared to take no arguments except `self`, it will ignore any extra arguments passed to the constructor without complaining about a signature mismatch
- `__init__()` 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.

3.4.4 Finalization: `__dealloc__()`

- This method is the counter-part to `__cinit__()`.
- Any C-data that was explicitly allocated in the `__cinit__()` 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 `__dealloc__()` method returns.

3.4.5 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 `__iadd__()`, in that these always take `self` as the first argument.

3.4.6 Rich Comparisons

Note: There are no separate methods for individual rich comparison operations.

- A single special method called `__richcmp__()` replaces all the individual rich compare, special method types.
- `__richcmp__()` takes an integer argument, indicating which operation is to be performed as shown in the table below.

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

3.4.7 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.

3.5 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 extenstion 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 C-layout of all the base classes must be compatible.

3.6 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
```

3.7 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 the ``sh`` parameter.
* 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`` clause::

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

* Now this function automatically checks that ``sh`` is not ``None``, as well as that is the
```

- `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.

3.8 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__
```

3.9 External and Public Types

3.9.1 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.

3.9.2 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 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 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.

3.9.3 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.

3.10 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.

CHAPTER

FOUR

INTERFACING WITH OTHER CODE

CHAPTER

FIVE

SPECIAL MENTION

CHAPTER

SIX

LIMITATIONS

CHAPTER

SEVEN

COMPILER DIRECTIVES

See Compilation.

CHAPTER

EIGHT

INDICES AND TABLES

8.1 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.

8.1.1 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__	x, y	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

8.1.2 Arithmetic operators

Name	Parameters	Return type	Description
__add__	x, y	object	binary + operator
__sub__	x, y	object	binary - operator
__mul__	x, y	object	* operator
__div__	x, y	object	/ operator for old-style division
__floordiv__	x, y	object	// operator
__truediv__	x, y	object	/ operator for new-style division
__mod__	x, y	object	% operator
__divmod__	x, y	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__	x, y	object	<< operator
__rshift__	x, y	object	>> operator
__and__	x, y	object	& operator
__or__	x, y	object	| operator
__xor__	x, y	object	^ operator

8.1.3 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

8.1.4 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

8.1.5 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

8.1.6 Iterators

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

8.1.7 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*		

8.1.8 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

- *genindex*
- *modindex*
- *search*