Cython Reference Guide

Release 0.29a0

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

July 08, 2018

Contents

Note:

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

Contents:

CHAPTER 1

Compilation

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

- A `.pyx` file is compiled 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 `cythonize` compiler command with your options and list of `.pyx` files to generate. For example:

```
$ cythonize -a -i yourmod.pyx
```

This creates a `yourmod.c` file (or `yourmod.cpp` in C++ mode), compiles it, and puts the resulting extension module (`.so` or `.pyd`, depending on your platform) next to the source file for direct import (`-i` builds "in place"). The `-a` switch additionally produces an annotated html file of the source code.

The `cythonize` command accepts multiple source files and glob patterns like `**/*.pyx` as argument and also understands the common `-j` option for running multiple parallel build jobs. When called without further options, it will only translate the source files to `.c` or `.cpp` files. Pass the `-h` flag for a complete list of supported options.

There is also a simpler command line tool named `cython` which only invokes the source code translator.

In the case of manual compilation, how to compile your `.c` files will vary depending on your operating system and compiler. The Python documentation for writing extension modules should have some details for your system. On a Linux system, for example, it might look similar to this:

```
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
      -I/usr/include/python3.5 -o yourmod.so yourmod.c
```

(`gcc` will need to have paths to your included header files and paths to libraries you want to link with.)

After compilation, a `yourmod.so` (`yourmod.pyd` for Windows) file is written into the target directory and your module, `yourmod`, is available for you to import as with any other Python module. Note that if you are not relying on `cythonize` or distutils, you will not automatically benefit from the platform specific file extension that CPython generates for disambiguation, such as `yourmod.cpython-35m-x86_64-linux-gnu.so` on a regular 64bit Linux installation of CPython 3.5.

1.2 Compiling with `distutils`

The `distutils` package is part of the standard library. It is the standard way of building Python packages, including native extension modules. The following example configures the build for a Cython file called *hello.pyx*. First, 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
)
```

Now, 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()]
```

Note: Using memoryviews or importing NumPy with `import numpy` does not mean that you have to add the path to NumPy include files. You need to add this path only if you use `cimport numpy`.

Despite this, you will still get warnings like the following from the compiler, because Cython is using a deprecated Numpy API:

```
.../include/numpy/npy_1_7_deprecated_api.h:15:2: warning: #warning "Using deprecated␣
↪NumPy API, disable it by " "#defining NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-
↪Wcpp]
```

For the time being, it is just a warning that you can ignore.

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),
)
```

Note that when using setuptools, you should import it before Cython as setuptools may replace the `Extension` class in distutils. Otherwise, both might disagree about the class to use here.

Note also that if you use setuptools instead of distutils, the default action when running `python setup.py install` is to create a zipped `egg` file which will not work with `cimport` for `pxd` files when you try to use them from a dependent package. To prevent this, include `zip_safe=False` in the arguments to `setup()`.

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 or .pxd 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 cimport multiple .pxd files defining libraries, then Cython merges the list of libraries, so this works as expected (similarly with other options, like `include_dirs` above).

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(
```

(continues on next page)

(continued from previous page)

```
    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.

Sometimes this is not enough and you need finer customization of the distutils `Extension`. To do this, you can provide a custom function `create_extension` to create the final `Extension` object after Cython has processed the sources, dependencies and `# distutils` directives but before the file is actually Cythonized. This function takes 2 arguments `template` and `kwds`, where `template` is the `Extension` object given as input to Cython and `kwds` is a `dict` with all keywords which should be used to create the `Extension`. The function `create_extension` must return a 2-tuple `(extension, metadata)`, where `extension` is the created `Extension` and `metadata` is metadata which will be written as JSON at the top of the generated C files. This metadata is only used for debugging purposes, so you can put whatever you want in there (as long as it can be converted to JSON). The default function (defined in `Cython.Build.Dependencies`) is:

```
def default_create_extension(template, kwds):
    if 'depends' in kwds:
        include_dirs = kwds.get('include_dirs', []) + ["."]
        depends = resolve_depends(kwds['depends'], include_dirs)
        kwds['depends'] = sorted(set(depends + template.depends))

    t = template.__class__
    ext = t(**kwds)
    metadata = dict(distutils=kwds, module_name=kwds['name'])
    return (ext, metadata)
```

In case that you pass a string instead of an `Extension` to `cythonize()`, the `template` will be an `Extension` without sources. For example, if you do `cythonize("*.pyx")`, the `template` will be `Extension(name="*.pyx", sources=[])`.

Just as an example, this adds `mylib` as library to every extension:

```
from Cython.Build.Dependencies import default_create_extension

def my_create_extension(template, kwds):
    libs = kwds.get('libraries', []) + ["mylib"]
    kwds['libraries'] = libs
    return default_create_extension(template, kwds)

ext_modules = cythonize(..., create_extension=my_create_extension)
```

Note: If you Cythonize in parallel (using the `nthreads` argument), then the argument to `create_extension` must be pickleable. In particular, it cannot be a lambda function.

1.2.2 Cythonize arguments

The function `cythonize()` can take extra arguments which will allow you to customize your build.

`Cython.Build.`**`cythonize`**(*module_list*, *exclude=None*, *nthreads=0*, *aliases=None*, *quiet=False*, *force=False*, *language=None*, *exclude_failures=False*, ***options*)

Compile a set of source modules into C/C++ files and return a list of distutils Extension objects for them.

Parameters

- **module_list** – As module list, pass either a glob pattern, a list of glob patterns or a list of Extension objects. The latter allows you to configure the extensions separately through the normal distutils options. You can also pass Extension objects that have glob patterns as their sources. Then, cythonize will resolve the pattern and create a copy of the Extension for every matching file.
- **exclude** – When passing glob patterns as `module_list`, you can exclude certain module names explicitly by passing them into the `exclude` option.
- **nthreads** – The number of concurrent builds for parallel compilation (requires the `multiprocessing` module).
- **aliases** – If you want to use compiler directives like `# distutils: ...` but can only know at compile time (when running the `setup.py`) which values to use, you can use aliases and pass a dictionary mapping those aliases to Python strings when calling *`cythonize()`*. As an example, say you want to use the compiler directive `# distutils: include_dirs = ../static_libs/include/` but this path isn't always fixed and you want to find it when running the `setup.py`. You can then do `# distutils: include_dirs = MY_HEADERS`, find the value of `MY_HEADERS` in the `setup.py`, put it in a python variable called `foo` as a string, and then call `cythonize(..., aliases={'MY_HEADERS': foo})`.
- **quiet** – If True, Cython won't print error and warning messages during the compilation.
- **force** – Forces the recompilation of the Cython modules, even if the timestamps don't indicate that a recompilation is necessary.
- **language** – To globally enable C++ mode, you can pass `language='c++'`. Otherwise, this will be determined at a per-file level based on compiler directives. This affects only modules found based on file names. Extension instances passed into *`cythonize()`* will not be changed. It is recommended to rather use the compiler directive `# distutils: language = c++` than this option.
- **exclude_failures** – For a broad 'try to compile' mode that ignores compilation failures and simply excludes the failed extensions, pass `exclude_failures=True`. Note that this only really makes sense for compiling `.py` files which can also be used without compilation.
- **annotate** – If `True`, will produce a HTML file for each of the `.pyx` or `.py` files compiled. The HTML file gives an indication of how much Python interaction there is in each of the source code lines, compared to plain C code. It also allows you to see the C/C++ code generated for each line of Cython code. This report is invaluable when optimizing a function for speed, and for determining when to release the GIL: in general, a `nogil` block may contain only "white" code. See examples in determining_where_to_add_types or primes.
- **compiler_directives** – Allow to set compiler directives in the `setup.py` like this: `compiler_directives={'embedsignature': True}`. See *Compiler directives*.

1.2.3 Compiler options

Compiler options can be set in the `setup.py`, before calling `cythonize()`, like this:

```
from distutils.core import setup

from Cython.Build import cythonize
```

(continues on next page)

(continued from previous page)

```
from Cython.Compiler import Options

Options.docstrings = False

setup(
    name = "hello",
    ext_modules = cythonize("lib.pyx"),
)
```

Here are the options that are available:

Cython.Compiler.Options.**docstrings = True**
Whether or not to include docstring in the Python extension. If False, the binary size will be smaller, but the `__doc__` attribute of any class or function will be an empty string.

Cython.Compiler.Options.**embed_pos_in_docstring = False**
Embed the source code position in the docstrings of functions and classes.

Cython.Compiler.Options.**emit_code_comments = True**
Copy the original source code line by line into C code comments in the generated code file to help with understanding the output. This is also required for coverage analysis.

Cython.Compiler.Options.**generate_cleanup_code = False**
Decref global variables in each module on exit for garbage collection. 0: None, 1+: interned objects, 2+: cdef globals, 3+: types objects Mostly for reducing noise in Valgrind, only executes at process exit (when all memory will be reclaimed anyways).

Cython.Compiler.Options.**clear_to_none = True**
Should tp_clear() set object fields to None instead of clearing them to NULL?

Cython.Compiler.Options.**annotate = False**
Generate an annotated HTML version of the input source files for debugging and optimisation purposes. This has the same effect as the `annotate` argument in `cythonize()`.

Cython.Compiler.Options.**fast_fail = False**
This will abort the compilation on the first error occurred rather than trying to keep going and printing further error messages.

Cython.Compiler.Options.**warning_errors = False**
Turn all warnings into errors.

Cython.Compiler.Options.**error_on_unknown_names = True**
Make unknown names an error. Python raises a NameError when encountering unknown names at runtime, whereas this option makes them a compile time error. If you want full Python compatibility, you should disable this option and also 'cache_builtins'.

Cython.Compiler.Options.**error_on_uninitialized = True**
Make uninitialized local variable reference a compile time error. Python raises UnboundLocalError at runtime, whereas this option makes them a compile time error. Note that this option affects only variables of "python object" type.

Cython.Compiler.Options.**convert_range = True**
This will convert statements of the form `for i in range(...)` to `for i from ...` when `i` is a C 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 an overflowing value is encountered.

Cython.Compiler.Options.**cache_builtins = True**
Perform lookups on builtin names only once, at module initialisation time. This will prevent the module from

getting imported if a builtin name that it uses cannot be found during initialisation. Default is True. Note that some legacy builtins are automatically remapped from their Python 2 names to their Python 3 names by Cython when building in Python 3.x, so that they do not get in the way even if this option is enabled.

`Cython.Compiler.Options.`**`gcc_branch_hints = True`**
Generate branch prediction hints to speed up error handling etc.

`Cython.Compiler.Options.`**`lookup_module_cpdef = False`**
Enable this to allow one to write `your_module.foo = ...` to overwrite the definition if the cpdef function foo, at the cost of an extra dictionary lookup on every call. If this is false it generates only the Python wrapper and no override check.

`Cython.Compiler.Options.`**`embed = None`**
Whether or not to embed the Python interpreter, for use in making a standalone executable or calling from external libraries. This will provide a C function which initialises the interpreter and executes the body of this module. See this demo for a concrete example. If true, the initialisation function is the C main() function, but this option can also be set to a non-empty string to provide a function name explicitly. Default is False.

`Cython.Compiler.Options.`**`cimport_from_pyx = False`**
Allows cimporting from a pyx file without a pxd file.

`Cython.Compiler.Options.`**`buffer_max_dims = 8`**
Maximum number of dimensions for buffers – set lower than number of dimensions in numpy, as slices are passed by value and involve a lot of copying.

`Cython.Compiler.Options.`**`closure_freelist_size = 8`**
Number of function closure instances to keep in a freelist (0: no freelists)

1.2.4 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])]

```

(continues on next page)

(continued from previous page)

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

setup(
    ext_modules = extensions
)
```

If you have many extensions and want to avoid the additional complexity in the declarations, you can declare them with their normal Cython sources and then call the following function instead of `cythonize()` to adapt the sources list in the Extensions when not using Cython:

```
import os.path

def no_cythonize(extensions, **_ignore):
    for extension in extensions:
        sources = []
        for sfile in extension.sources:
            path, ext = os.path.splitext(sfile)
            if ext in ('.pyx', '.py'):
                if extension.language == 'c++':
                    ext = '.cpp'
                else:
                    ext = '.c'
                sfile = path + ext
            sources.append(sfile)
        extension.sources[:] = sources
    return extensions
```

Another option is to make Cython a setup dependency of your system and use Cython's build_ext module which runs `cythonize` as part of the build process:

```
setup(
    setup_requires=[
        'cython>=0.x',
    ],
    extensions = [Extension("*", ["*.pyx"])],
    cmdclass={'build_ext': Cython.Build.build_ext},
    ...
)
```

If you want to expose the C-level interface of your library for other libraries to cimport from, use package_data to install the `.pxd` files, e.g.:

```
setup(
    package_data = {
        'my_package': ['*.pxd'],
        'my_package/sub_package': ['*.pxd'],
    },
    ...
)
```

These `.pxd` files need not have corresponding `.pyx` modules if they contain purely declarations of external libraries.

Remember that if you use setuptools instead of distutils, the default action when running `python setup.py install` is to create a zipped `egg` file which will not work with `cimport` for `pxd` files when you try to use them from a dependent package. To prevent this, include `zip_safe=False` in the arguments to `setup()`.

1.3 Integrating multiple modules

In some scenarios, it can be useful to link multiple Cython modules (or other extension modules) into a single binary, e.g. when embedding Python in another application. This can be done through the inittab import mechanism of CPython.

Create a new C file to integrate the extension modules and add this macro to it:

```
#if PY_MAJOR_VERSION < 3
# define MODINIT(name)  init ## name
#else
# define MODINIT(name)  PyInit_ ## name
#endif
```

If you are only targeting Python 3.x, just use `PyInit_` as prefix.

Then, for each or the modules, declare its module init function as follows, replacing `...` by the name of the module:

```
PyMODINIT_FUNC  MODINIT(...) (void);
```

In C++, declare them as `extern C`.

If you are not sure of the name of the module init function, refer to your generated module source file and look for a function name starting with `PyInit_`.

Next, before you start the Python runtime from your application code with `Py_Initialize()`, you need to initialise the modules at runtime using the `PyImport_AppendInittab()` C-API function, again inserting the name of each of the modules:

```
PyImport_AppendInittab("...", MODINIT(...));
```

This enables normal imports for the embedded extension modules.

In order to prevent the joined binary from exporting all of the module init functions as public symbols, Cython 0.28 and later can hide these symbols if the macro `CYTHON_NO_PYINIT_EXPORT` is defined while C-compiling the module C files.

Also take a look at the cython_freeze tool.

1.4 Compiling with `pyximport`

For building Cython modules during development without explicitly running `setup.py` after each change, you can use `pyximport`:

```
>>> 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.

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)
```

In the case that Cython fails to compile a Python module, `pyximport` will fall back to loading the source modules instead.

Note that it is not recommended to let `pyximport` build code on end user side as it hooks into their import system. The best way to cater for end users is to provide pre-built binary packages in the wheel packaging format.

1.4.1 Arguments

The function `pyximport.install()` can take several arguments to influence the compilation of Cython or Python files.

`pyximport.install`(*pyximport=True*, *pyimport=False*, *build_dir=None*, *build_in_temp=True*, *setup_args=None*, *reload_support=False*, *load_py_module_on_import_failure=False*, *inplace=False*, *language_level=None*)

Main entry point for pyxinstall.

Call this to install the `.pyx` import hook in your meta-path for a single Python process. If you want it to be installed whenever you use Python, add it to your `sitecustomize` (as described above).

Parameters

- **`pyximport`** – If set to False, does not try to import `.pyx` files.
- **`pyimport`** – You can pass `pyimport=True` to also install the `.py` import hook in your meta-path. Note, however, that it is rather experimental, will not work at all for some `.py` files and packages, and will heavily slow down your imports due to search and compilation. Use at your own risk.
- **`build_dir`** – By default, compiled modules will end up in a `.pyxbld` directory in the user's home directory. Passing a different path as `build_dir` will override this.
- **`build_in_temp`** – If `False`, will produce the C files locally. Working with complex dependencies and debugging becomes more easy. This can principally interfere with existing files of the same name.
- **`setup_args`** – Dict of arguments for Distribution. See `distutils.core.setup()`.
- **`reload_support`** – Enables support for dynamic `reload(my_module)`, e.g. after a change in the Cython code. Additional files `<so_path>.reloadNN` may arise on that account, when the previously loaded module file cannot be overwritten.
- **`load_py_module_on_import_failure`** – If the compilation of a `.py` file succeeds, but the subsequent import fails for some reason, retry the import with the normal `.py` module instead of the compiled module. Note that this may lead to unpredictable results for modules that change the system state during their import, as the second import will rerun these modifications in whatever state the system was left after the import of the compiled module failed.
- **`inplace`** – Install the compiled module (`.so` for Linux and Mac / `.pyd` for Windows) next to the source file.
- **`language_level`** – The source language level to use: 2 or 3. The default is to use the language level of the current Python runtime for .py files and Py2 for `.pyx` files.

1.4.2 Dependency Handling

Since `pyximport` does not use `cythonize()` internally, it currently requires a different setup for dependencies. 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 create 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.

1.4.3 Limitations

pyximport does not use cythonize(). Thus it is not possible to do things like using compiler directives at the top of Cython files or compiling Cython code to C++.

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 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 feeling that pyximport really did rebuild your module as it was supposed to.

Basic module reloading support is available with the option reload_support=True. Note that this will generate a new module filename for each build and thus end up loading multiple shared libraries into memory over time. CPython has limited support for reloading shared libraries as such, see PEP 489.

Pyximport puts both your .c file and the platform-specific binary into a separate build directory, usually $HOME/.pyxblx/. To copy it back into the package hierarchy (usually next to the source file) for manual reuse, you can pass the option inplace=True.

1.5 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.

1.6 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.

1.7 Compiling with a Jupyter Notebook

It's possible to compile code in a notebook cell with Cython. For this you need to load the Cython magic:

```
%load_ext cython
```

Then you can define a Cython cell by writing `%%cython` on top of it. Like this:

```
%%cython

cdef int a = 0
for i in range(10):
    a += i
print(a)
```

Note that each cell will be compiled into a separate extension module. So if you use a package in a Cython cell, you will have to import this package in the same cell. It's not enough to have imported the package in a previous cell. Cython will tell you that there are "undefined global names" at compilation time if you don't comply.

The global names (top level functions, classes, variables and modules) of the cell are then loaded into the global namespace of the notebook. So in the end, it behaves as if you executed a Python cell.

Additional allowable arguments to the Cython magic are listed below. You can see them also by typing `` `%%cython? `` in IPython or a Jupyter notebook.

-a, –annotate	Produce a colorized HTML version of the source.
-+, –cplus	Output a C++ rather than C file.
-f, –force	Force the compilation of a new module, even if the source has been previously compiled.
-3	Select Python 3 syntax
-2	Select Python 2 syntax
-c=COMPILE_ARGS, –compile-args=COMPILE_ARGS	Extra flags to pass to compiler via the extra_compile_args.
–link-args LINK_ARGS	Extra flags to pass to linker via the extra_link_args.
-l LIB, –lib LIB	Add a library to link the extension against (can be specified multiple times).
-L dir	Add a path to the list of library directories (can be specified multiple times).
-I INCLUDE, –include INCLUDE	Add a path to the list of include directories (can be specified multiple times).
-S, –src	Add a path to the list of src files (can be specified multiple times).
-n NAME, –name NAME	Specify a name for the Cython module.
–pgo	Enable profile guided optimisation in the C compiler. Compiles the cell twice and executes it in between to generate a runtime profile.
–verbose	Print debug information like generated .c/.cpp file location and exact gcc/g++ command invoked.

1.8 Compiler directives

Compiler directives are instructions which affect the behavior of Cython code. Here is the list of currently supported directives:

`binding` (True / False) Controls whether free functions behave more like Python's CFunctions (e.g. `len()`) or, when set to True, more like Python's functions. When enabled, functions will bind to an instance when looked up as a class attribute (hence the name) and will emulate the attributes of Python functions, including introspections like argument names and annotations. Default is False.

`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 and sequences 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 is allowed to neither check for nor correctly handle negative indices, possibly causing segfaults or data corruption. If bounds checks are enabled (the default, see `boundschecks` above), negative indexing will usually raise an `IndexError` for indices that Cython evaluates itself. However, these cases can be difficult to recognise in user code to distinguish them from indexing or slicing that is evaluated by the underlying Python array or sequence object and thus continues to support wrap-around indices. It is therefore safest to apply this option only to code that does not process negative indices at all. Default is True.

`initializedcheck` (True / False) If set to True, Cython checks that a memoryview is initialized whenever its elements are accessed or assigned to. Setting this to False disables these checks. 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) Write hooks for Python profilers into the compiled C code. Default is False.

`linetrace` (True / False) Write line tracing hooks for Python profilers or coverage reporting 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`). Define `CYTHON_TRACE_NOGIL=1` to also include `nogil` functions and sections.

`infer_types` (True / False) Infer types of untyped variables in function bodies. Default is None, indicating that only safe (semantically-unchanging) inferences are allowed. In particular, inferring *integral* types for variables *used in arithmetic expressions* is considered unsafe (due to possible overflow) and must be explicitly requested.

`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 files inherit this setting from the module being compiled, unless they explicitly set their own language level. Included source files always inherit this setting.

`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.

`unraisable_tracebacks` (True / False) Whether to print tracebacks when suppressing unraisable exceptions.

`iterable_coroutine` (True / False) PEP 492 specifies that async-def coroutines must not be iterable, in order to prevent accidental misuse in non-async contexts. However, this makes it difficult and inefficient to write backwards compatible code that uses async-def coroutines in Cython but needs to interact with async Python code that uses the older yield-from syntax, such as asyncio before Python 3.5. This directive can be applied in modules or selectively as decorator on an async-def coroutine to make the affected coroutine(s) iterable and thus directly interoperable with yield-from.

1.8.1 Configurable optimisations

`optimize.use_switch` (True / False) Whether to expand chained if-else statements (including statements like `if x == 1 or x == 2:`) into C switch statements. This can have performance benefits if there are lots of values but cause compiler errors if there are any duplicate values (which may not be detectable at Cython compile time for all C constants). Default is True.

`optimize.unpack_method_calls` (True / False) Cython can generate code that optimistically checks for Python method objects at call time and unpacks the underlying function to call it directly. This can substantially speed up method calls, especially for builtins, but may also have a slight negative performance impact in some cases where the guess goes completely wrong. Disabling this option can also reduce the code size. Default is True.

1.8.2 Warnings

All warning directives take True / False as options to turn the warning on / off.

`warn.undeclared` (default False) Warns about any variables that are implicitly declared without a `cdef` declaration

`warn.unreachable` (default True) Warns about code paths that are statically determined to be unreachable, e.g. returning twice unconditionally.

`warn.maybe_uninitialized` (default False) Warns about use of variables that are conditionally uninitialized.

`warn.unused` (default False) Warns about unused variables and declarations

`warn.unused_arg` (default False) Warns about unused function arguments

`warn.unused_result` (default False) Warns about unused assignment to the same name, such as `r = 2; r = 1 + 2`

`warn.multiple_declarators` (default True) Warns about multiple variables declared on the same line with at least one pointer type. For example `cdef double* a, b` - which, as in C, declares `a` as a pointer, `b` as a value type, but could be mininterpreted as declaring two pointers.

1.8.3 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: language_level=3, 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():
    ...
    # turn it temporarily on again for this block
    with cython.boundscheck(True):
        ...
```

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

In `setup.py`

Compiler directives can also be set in the `setup.py` file by passing a keyword argument to `cythonize`:

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

setup(
    name = "My hello app",
    ext_modules = cythonize('hello.pyx', compiler_directives={'embedsignature': True}
↪),
)
```

This will override the default directives as specified in the `compiler_directives` dictionary. Note that explicit per-file or local directives as explained above take precedence over the values passed to `cythonize`.

CHAPTER 2

Indices and tables

2.1 Special Methods Table

You can find an updated version of the special methods table in special_methods_table.

2.1.1 General

2.1.2 Rich comparison operators

2.1.3 Arithmetic operators

2.1.4 Numeric conversions

2.1.5 In-place arithmetic operators

2.1.6 Sequences and mappings

2.1.7 Iterators

2.1.8 Buffer interface

2.1.9 Descriptor objects

- genindex
- modindex
- search

Index

A

B

C

D

E

F

G

I

L

W