Cython Tutorial

Release 0.28.1

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

April 08, 2018

Contents

CHAPTER 1

Basic Tutorial

1.1 The Basics of Cython

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

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

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

1.2 Cython Hello World

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

So lets start with the canonical python hello world:

```
print("Hello World")
```

Save this code in a file named `helloworld.pyx`. Now we need to create the `setup.py`, which is like a python Makefile (for more information see compilation). Your `setup.py` should look like:

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

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

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

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

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

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

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

1.2.1 `pyximport`: Cython Compilation for Developers

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

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

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

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

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.

1.3 Fibonacci Fun

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

```
from __future__ import print_function

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

    print()
```

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

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

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

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

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

And use the new extension with:

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

1.4 Primes

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

primes.pyx:

```
def primes(int nb_primes):
    cdef int n, i, len_p
    cdef int p[1000]
    if nb_primes > 1000:
        nb_primes = 1000

    len_p = 0  # The current number of elements in p.
    n = 2
    while len_p < nb_primes:
        # Is n prime?
        for i in p[:len_p]:
            if n % i == 0:
                break

        # If no break occurred in the loop, we have a prime.
        else:
            p[len_p] = n
            len_p += 1
        n += 1

    # Let's return the result in a python list:
    result_as_list  = [prime for prime in p[:len_p]]
    return result_as_list
```

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

Now, let's dig into the core of the function:

```
cdef int n, i, len_p
cdef int p[1000]
```

Lines 2 and 3 use the `cdef` statement to define some local C variables. The result is stored in the C array `p` during processing, and will be copied into a Python list at the end (line 22).

Note: You cannot create very large arrays in this manner, because they are allocated on the C function call stack, which is a rather precious and scarce resource. To request larger arrays, or even arrays with a length only known at runtime, you can learn how to make efficient use of *C memory allocation*, *Python arrays* or NumPy arrays with Cython.

```
if nb_primes > 1000:
    nb_primes = 1000
```

As in C, declaring a static array requires knowing the size at compile time. We make sure the user doesn't set a value above 1000 (or we would have a segmentation fault, just like in C).

```
len_p = 0  # The number of elements in p
n = 2
while len_p < nb_primes:
```

Lines 7-9 set up for a loop which will test candidate numbers for primeness until the required number of primes has been found.

```
# Is n prime?
for i in p[:len_p]:
    if n % i == 0:
        break
```

Lines 11-12, which try dividing a candidate by all the primes found so far, are of particular interest. Because no Python objects are referred to, the loop is translated entirely into C code, and thus runs very fast. You will notice the way we iterate over the `p` C array.

```
for i in p[:len_p]:
```

The loop gets translated into a fast C loop and works just like iterating over a Python list or NumPy array. If you don't slice the C array with `[:len_p]`, then Cython will loop over the 1000 elements of the array.

```
# If no break occurred in the loop
else:
    p[len_p] = n
    len_p += 1
n += 1
```

If no breaks occurred, it means that we found a prime, and the block of code after the `else` line 16 will be executed. We add the prime found to `p`. If you find having an `else` after a for-loop strange, just know that it's a lesser known features of the Python language, and that Cython executes it at C speed for you. If the for-else syntax confuses you, see this excellent blog post.

```
# Let's put the result in a python list:
result_as_list  = [prime for prime in p[:len_p]]
return result_as_list
```

In line 22, before returning the result, we need to copy our C array into a Python list, because Python can't read C arrays. Cython can automatically convert many C types from and to Python types, as described in the documentation on type conversion, so we can use a simple list comprehension here to copy the C `int` values into a Python list of Python `int` objects, which Cython creates automatically along the way. You could also have iterated manually over the C array and used `result_as_list.append(prime)`, the result would have been the same.

You'll notice we declare a Python list exactly the same way it would be in Python. Because the variable `result_as_list` hasn't been explicitly declared with a type, it is assumed to hold a Python object, and from the assignment, Cython also knows that the exact type is a Python list.

Finally, at line 18, a normal Python return statement returns the result list.

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

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

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

Cython has a way to visualise where interaction with Python objects and Python's C-API is taking place. For this, pass the `annotate=True` parameter to `cythonize()`. It produces a HTML file. Let's see:

If a line is white, it means that the code generated doesn't interact with Python, so will run as fast as normal C code. The darker the yellow, the more Python interaction there is in that line. Those yellow lines will usually operate on Python objects, raise exceptions, or do other kinds of higher-level operations than what can easily be translated into simple and fast C code. The function declaration and return use the Python interpreter so it makes sense for those lines to be yellow. Same for the list comprehension because it involves the creation of a Python object. But the line `if n % i == 0:`, why? We can examine the generated C code to understand:

We can see that some checks happen. Because Cython defaults to the Python behavior, the language will perform division checks at runtime, just like Python does. You can deactivate those checks by using the compiler directives.

Now let's see if, even if we have division checks, we obtained a boost in speed. Let's write the same program, but Python-style:

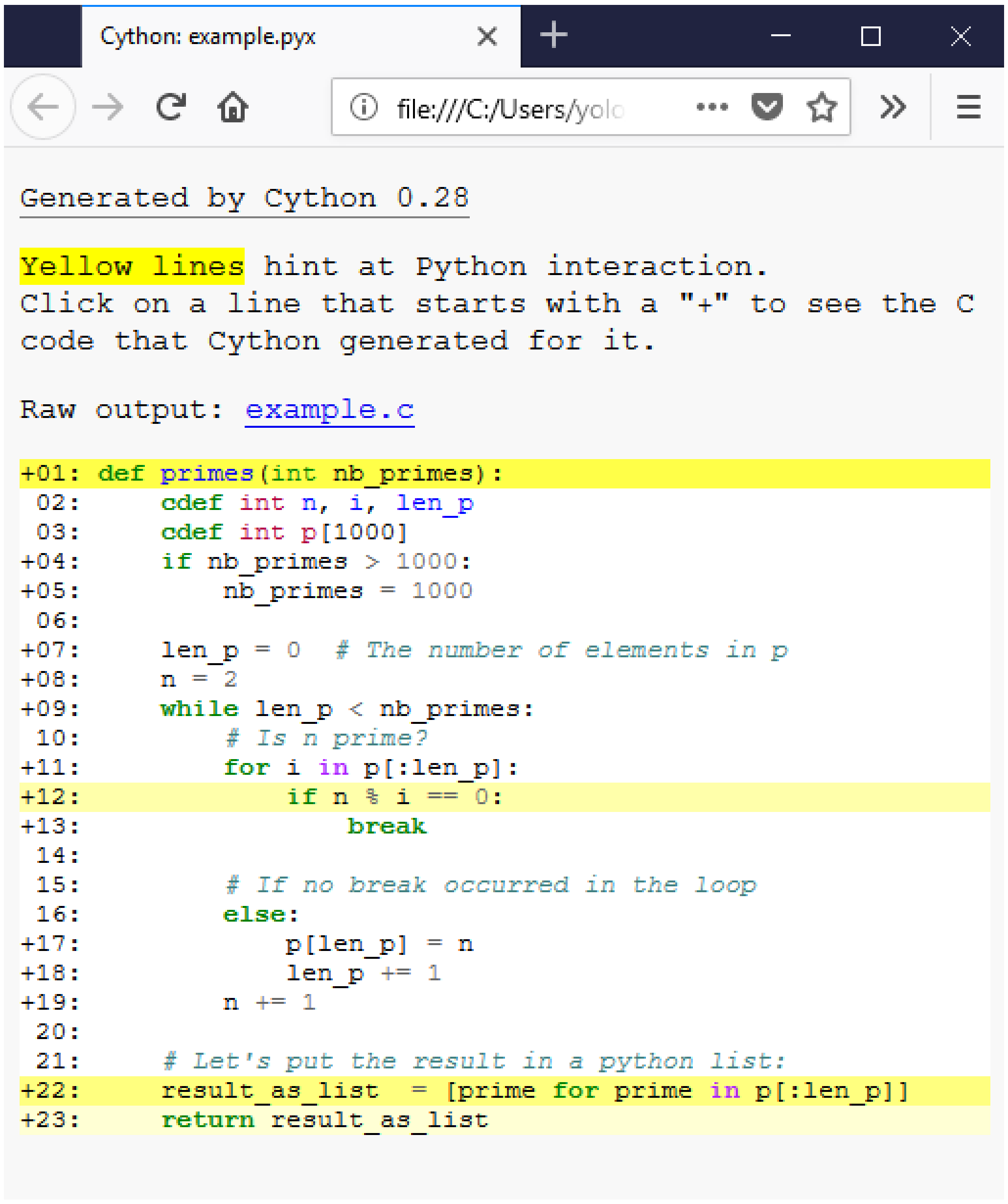
```
def primes_python(nb_primes):
    p = []
    n = 2
    while len(p) < nb_primes:
        # Is n prime?
        for i in p:
            if n % i == 0:
                break

        # If no break occurred in the loop
        else:
            p.append(n)
        n += 1
    return p
```

It is also possible to take a plain `.py` file and to compile it with Cython. Let's take `primes_python`, change the function name to `primes_python_compiled` and compile it with Cython (without changing the code). We will also change the name of the file to `example_py_cy.py` to differentiate it from the others. Now the `setup.py` looks like this:

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

setup(
    ext_modules=cythonize(['example.pyx',        # Cython code file with primes()␣
↪function
                           'example_py_cy.py'],  # Python code file with primes_
↪python_compiled() function
```

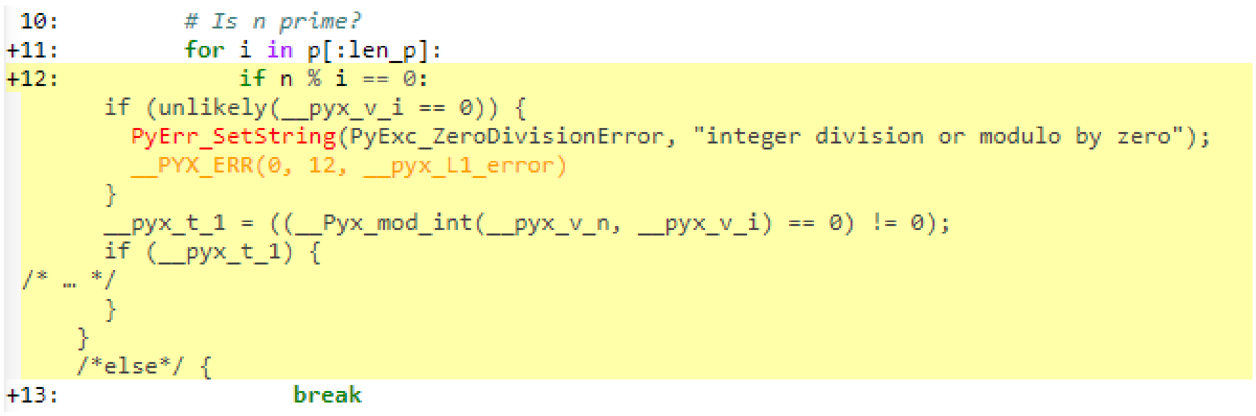


Generated by Cython 0.28

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: example.c

```
+01: def primes(int nb_primes):
 02:     cdef int n, i, len_p
 03:     cdef int p[1000]
+04:     if nb_primes > 1000:
+05:         nb_primes = 1000
 06: 
+07:     len_p = 0  # The number of elements in p
+08:     n = 2
+09:     while len_p < nb_primes:
 10:         # Is n prime?
+11:         for i in p[:len_p]:
+12:             if n % i == 0:
+13:                 break
 14: 
 15:         # If no break occurred in the loop
 16:         else:
+17:             p[len_p] = n
+18:             len_p += 1
+19:         n += 1
 20: 
 21:     # Let's put the result in a python list:
+22:     result_as_list  = [prime for prime in p[:len_p]]
+23:     return result_as_list
```



```
 10:         # Is n prime?
+11:         for i in p[:len_p]:
+12:             if n % i == 0:
      if (unlikely(__pyx_v_i == 0)) {
        PyErr_SetString(PyExc_ZeroDivisionError, "integer division or modulo by zero");
        __PYX_ERR(0, 12, __pyx_L1_error)
      }
      __pyx_t_1 = ((__Pyx_mod_int(__pyx_v_n, __pyx_v_i) == 0) != 0);
      if (__pyx_t_1) {
/* … */
      }
    }
    /*else*/ {
+13:                 break
```

```
                              annotate=True),        # enables generation of the html␣
↪annotation file
)
```

Now we can ensure that those two programs output the same values:

```
>>> primes_python(1000) == primes(1000)
True
>>> primes_python_compiled(1000) == primes(1000)
True
```

It's possible to compare the speed now:

```
python -m timeit -s 'from example_py import primes_python' 'primes_python(1000)'
10 loops, best of 3: 23 msec per loop

python -m timeit -s 'from example_py_cy import primes_python_compiled' 'primes_python_
↪compiled(1000)'
100 loops, best of 3: 11.9 msec per loop

python -m timeit -s 'from example import primes' 'primes(1000)'
1000 loops, best of 3: 1.65 msec per loop
```

The cythonize version of `primes_python` is 2 times faster than the Python one, without changing a single line of code. The Cython version is 13 times faster than the Python version! What could explain this?

Multiple things:

- In this program, very little computation happen at each line. So the overhead of the python interpreter is very important. It would be very different if you were to do a lot computation at each line. Using NumPy for example.
- Data locality. It's likely that a lot more can fit in CPU cache when using C than when using Python. Because everything in python is an object, and every object is implemented as a dictionary, this is not very cache friendly.

Usually the speedups are between 2x to 1000x. It depends on how much you call the Python interpreter. As always, remember to profile before adding types everywhere. Adding types makes your code less readable, so use them with moderation.

1.5 Language Details

For more about the Cython language, see language-basics. To dive right in to using Cython in a numerical computation context, see memoryviews.

CHAPTER 2

Calling C functions

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

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

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

```
from libc.stdlib cimport atoi

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

You can find a complete list of these standard cimport files in Cython's source package Cython/Includes/. They are stored in `.pxd` files, the standard way to provide reusable Cython declarations that can be shared across modules (see sharing-declarations).

Cython also has a complete set of declarations for CPython's C-API. For example, to test at C compilation time which CPython version your code is being compiled with, you can do this:

```
from cpython.version cimport PY_VERSION_HEX

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

Cython also provides declarations for the C math library:

```
from libc.math cimport sin

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

2.1 Dynamic linking

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

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

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

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

2.2 External declarations

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

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

This declares the `sin()` function in a way that makes it available to Cython code and instructs Cython to generate C code that includes the `math.h` header file. The C compiler will see the original declaration in `math.h` at compile time, but Cython does not parse "math.h" and requires a separate definition.

Just like the `sin()` function from the math library, it is possible to declare and call into any C library as long as the module that Cython generates is properly linked against the shared or static library.

Note that you can easily export an external C function from your Cython module by declaring it as `cpdef`. This generates a Python wrapper for it and adds it to the module dict. Here is a Cython module that provides direct access to the C `sin()` function for Python code:

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

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

You get the same result when this declaration appears in the `.pxd` file that belongs to the Cython module (i.e. that has the same name, see sharing-declarations). This allows the C declaration to be reused in other Cython modules, while still providing an automatically generated Python wrapper in this specific module.

2.3 Naming parameters

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

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

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

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

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

```
cdef char* data = "hfvcakdfagbcffvschvxcdfgccbcfhvgcsnfxjh"

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

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

CHAPTER 3

Using C libraries

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

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

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

3.1 Defining external declarations

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

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

typedef struct _Queue Queue;
typedef void *QueueValue;

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

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

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

```
int queue_is_empty(Queue *queue);
```

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

```
# file: cqueue.pxd

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

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

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

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

    bint queue_is_empty(Queue* queue)
```

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

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

One noteworthy difference to the header file that we use above is the declaration of the `Queue` struct in the first line. `Queue` is in this case used as an *opaque handle*; only the library that is called knows what is really inside. Since no Cython code needs to know the contents of the struct, we do not need to declare its contents, so we simply provide an empty definition (as we do not want to declare the `_Queue` type which is referenced in the C header)[1].

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

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

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

3.2 Writing a wrapper class

After declaring our C library's API, we can start to design the Queue class that should wrap the C queue. It will live in a file called `queue.pyx`.[2]

Here is a first start for the Queue class:

```
# file: queue.pyx

cimport cqueue

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

Note that it says `__cinit__` rather than `__init__`. While `__init__` is available as well, it is not guaranteed to be run (for instance, one could create a subclass and forget to call the ancestor's constructor). Because not initializing C pointers often leads to hard crashes of the Python interpreter, Cython provides `__cinit__` which is *always* called immediately on construction, before CPython even considers calling `__init__`, and which therefore is the right place to initialise `cdef` fields of the new instance. However, as `__cinit__` is called during object construction, `self` is not fully constructed yet, and one must avoid doing anything with `self` but assigning to `cdef` fields.

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

3.3 Memory management

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

The Python way to get out of this is to raise a `MemoryError`[3]. We can thus change the init function as follows:

```
cimport cqueue

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

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

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

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

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

3.4 Compiling and linking

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

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

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

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

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

to

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

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

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

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

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

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

3.5 Mapping functionality

Before implementing the public interface of this class, it is good practice to look at what interfaces Python offers, e.g. in its `list` or `collections.deque` classes. Since we only need a FIFO queue, it's enough to provide the methods `append()`, `peek()` and `pop()`, and additionally an `extend()` method to add multiple values at once.

Also, since we already know that all values will be coming from C, it's best to provide only `cdef` methods for now, and to give them a straight C interface.

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

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

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

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

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

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

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

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

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

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

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

3.6 Handling errors

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Testing the result

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

A quick way to provide a Python API for the class is to change the methods from `cdef` to `cpdef`. This will let Cython generate two entry points, one that is callable from normal Python code using the Python call semantics and

Python objects as arguments, and one that is callable from C code with fast C semantics and without requiring intermediate argument conversion from or to Python types. Note that `cpdef` methods ensure that they can be appropriately overridden by Python methods even when they are called from Cython. This adds a tiny overhead compared to `cdef` methods.

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

```
cimport cqueue

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Callbacks

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 4

Extension types (aka. cdef classes)

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

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

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

Based on what Python calls a "built-in type", however, Cython supports a second kind of class: *extension types*, sometimes referred to as "cdef classes" due to the keywords used for their declaration. They are somewhat restricted compared to Python classes, but are generally more memory efficient and faster than generic Python classes. The main difference is that they use a C struct to store their fields and methods instead of a Python dict. This allows them to store arbitrary C types in their fields without requiring a Python wrapper for them, and to access fields and methods directly at the C level without passing through a Python dictionary lookup.

Normal Python classes can inherit from cdef classes, but not the other way around. Cython requires to know the complete inheritance hierarchy in order to lay out their C structs, and restricts it to single inheritance. Normal Python classes, on the other hand, can inherit from any number of Python classes and extension types, both in Cython code and pure Python code.

So far our integration example has not been very useful as it only integrates a single hard-coded function. In order to remedy this, with hardly sacrificing speed, we will use a cdef class to represent a function on floating point numbers:

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

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

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

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

Using this, we can now change our integration example:

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

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

This is almost as fast as the previous code, however it is much more flexible as the function to integrate can be changed. We can even pass in a new function defined in Python-space:

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

This is about 20 times slower, but still about 10 times faster than the original Python-only integration code. This shows how large the speed-ups can easily be when whole loops are moved from Python code into a Cython module.

Some notes on our new implementation of `evaluate`:

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

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

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

import cython

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

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

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

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

CHAPTER 5

pxd files

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

`pxd` files have many use-cases:

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

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

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

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

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

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

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

CHAPTER 6

Caveats

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

- `10**-2 == 0`, instead of `0.01` like in Python.
- Given two typed `int` variables `a` and `b`, `a % b` has the same sign as the second argument (following Python semantics) rather than having the same sign as the first (as in C). The C behavior can be obtained, at some speed gain, by enabling the cdivision directive (versions prior to Cython 0.12 always followed C semantics).
- Care is needed with unsigned types. `cdef unsigned n = 10; print(range(-n, n))` will print an empty list, since `-n` wraps around to a large positive integer prior to being passed to the `range` function.
- Python's `float` type actually wraps C `double` values, and the `int` type in Python 2.x wraps C `long` values.

CHAPTER 7

Profiling

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

7.1 Cython Profiling Basics

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

7.1.1 Enabling profiling for a complete source file

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

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

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

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

7.1.2 Disabling profiling function wise

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

```
cimport cython

@cython.profile(False)
```

```
def my_often_called_function():
   pass
```

7.1.3 Enabling line tracing

To get more detailed trace information (for tools that can make use of it), you can enable line tracing:

```
# cython: linetrace=True
```

This will also enable profiling support, so the above `profile=True` option is not needed. Line tracing is needed for coverage analysis, for example.

Note that even if line tracing is enabled via the compiler directive, it is not used by default. As the runtime slowdown can be substantial, it must additionally be compiled in by the C compiler by setting the C macro definition `CYTHON_TRACE=1`. To include nogil functions in the trace, set `CYTHON_TRACE_NOGIL=1` (which implies `CYTHON_TRACE=1`). C macros can be defined either in the extension definition of the `setup.py` script or by setting the respective distutils options in the source file with the following file header comment (if `cythonize()` is used for compilation):

```
# distutils: define_macros=CYTHON_TRACE_NOGIL=1
```

7.1.4 Enabling coverage analysis

Since Cython 0.23, line tracing (see above) also enables support for coverage reporting with the coverage.py tool. To make the coverage analysis understand Cython modules, you also need to enable Cython's coverage plugin in your `.coveragerc` file as follows:

```
[run]
plugins = Cython.Coverage
```

With this plugin, your Cython source files should show up normally in the coverage reports.

To include the coverage report in the Cython annotated HTML file, you need to first run the coverage.py tool to generate an XML result file. Pass this file into the `cython` command as follows:

```
$ cython  --annotate-coverage coverage.xml  package/mymodule.pyx
```

This will recompile the Cython module and generate one HTML output file next to each Cython source file it processes, containing colour markers for lines that were contained in the coverage report.

7.2 Profiling Tutorial

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

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

$$\pi^2 = 6\sum_{k=1}^{\infty}\frac{1}{k^2} = 6\lim_{k\to\infty}\big(\frac{1}{1^2}+\frac{1}{2^2}+\cdots+\frac{1}{k^2}\big) \approx 6\big(\frac{1}{1^2}+\frac{1}{2^2}+\cdots+\frac{1}{n^2}\big)$$

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

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

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

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

On my box, this needs approximately 4 seconds to run the function with the default n. The higher we choose n, the better will be the approximation for π. An experienced Python programmer will already see plenty of places to optimize this code. But remember the golden rule of optimization: Never optimize without having profiled. Let me repeat this: **Never** optimize without having profiled your code. Your thoughts about which part of your code takes too much time are wrong. At least, mine are always wrong. So let's write a short script to profile our code:

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

import pstats, cProfile

import calc_pi

cProfile.runctx("calc_pi.approx_pi()", globals(), locals(), "Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

Running this on my box gives the following output:

```
Sat Nov  7 17:40:54 2009    Profile.prof

         10000004 function calls in 6.211 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.243    3.243    6.211    6.211 calc_pi.py:7(approx_pi)
 10000000    2.526    0.000    2.526    0.000 calc_pi.py:4(recip_square)
        1    0.442    0.442    0.442    0.442 {range}
        1    0.000    0.000    6.211    6.211 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

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

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

```
# encoding: utf-8
# cython: profile=True
# filename: calc_pi.pyx

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

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

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

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

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

import pstats, cProfile

import pyximport
pyximport.install()

import calc_pi

cProfile.runctx("calc_pi.approx_pi()", globals(), locals(), "Profile.prof")

s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

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

```
Sat Nov  7 18:02:33 2009    Profile.prof

         10000004 function calls in 4.406 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.305    3.305    4.406    4.406 calc_pi.pyx:7(approx_pi)
 10000000    1.101    0.000    1.101    0.000 calc_pi.pyx:4(recip_square)
        1    0.000    0.000    4.406    4.406 {calc_pi.approx_pi}
        1    0.000    0.000    4.406    4.406 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

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

```
# encoding: utf-8
# cython: profile=True
# filename: calc_pi.pyx

cdef inline double recip_square(int i):
    return 1./(i*i)

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

Now running the profile script yields:

```
Sat Nov  7 18:10:11 2009    Profile.prof

         10000004 function calls in 2.622 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.782    1.782    2.622    2.622 calc_pi.pyx:7(approx_pi)
 10000000    0.840    0.000    0.840    0.000 calc_pi.pyx:4(recip_square)
        1    0.000    0.000    2.622    2.622 {calc_pi.approx_pi}
        1    0.000    0.000    2.622    2.622 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

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

```
# encoding: utf-8
# cython: profile=True
# filename: calc_pi.pyx

cimport cython

@cython.profile(False)
cdef inline double recip_square(int i):
    return 1./(i*i)

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

Running this shows an interesting result:

```
Sat Nov  7 18:15:02 2009    Profile.prof

         4 function calls in 0.089 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.089    0.089    0.089    0.089 calc_pi.pyx:10(approx_pi)
        1    0.000    0.000    0.089    0.089 {calc_pi.approx_pi}
        1    0.000    0.000    0.089    0.089 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

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

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

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

CHAPTER 8

Unicode and passing strings

Similar to the string semantics in Python 3, Cython strictly separates byte strings and unicode strings. Above all, this means that by default there is no automatic conversion between byte strings and unicode strings (except for what Python 2 does in string operations). All encoding and decoding must pass through an explicit encoding/decoding step. To ease conversion between Python and C strings in simple cases, the module-level `c_string_type` and `c_string_encoding` directives can be used to implicitly insert these encoding/decoding steps.

8.1 Python string types in Cython code

Cython supports four Python string types: `bytes`, `str`, `unicode` and `basestring`. The `bytes` and `unicode` types are the specific types known from normal Python 2.x (named `bytes` and `str` in Python 3). Additionally, Cython also supports the `bytearray` type which behaves like the `bytes` type, except that it is mutable.

The `str` type is special in that it is the byte string in Python 2 and the Unicode string in Python 3 (for Cython code compiled with language level 2, i.e. the default). Meaning, it always corresponds exactly with the type that the Python runtime itself calls `str`. Thus, in Python 2, both `bytes` and `str` represent the byte string type, whereas in Python 3, both `str` and `unicode` represent the Python Unicode string type. The switch is made at C compile time, the Python version that is used to run Cython is not relevant.

When compiling Cython code with language level 3, the `str` type is identified with exactly the Unicode string type at Cython compile time, i.e. it does not identify with `bytes` when running in Python 2.

Note that the `str` type is not compatible with the `unicode` type in Python 2, i.e. you cannot assign a Unicode string to a variable or argument that is typed `str`. The attempt will result in either a compile time error (if detectable) or a `TypeError` exception at runtime. You should therefore be careful when you statically type a string variable in code that must be compatible with Python 2, as this Python version allows a mix of byte strings and unicode strings for data and users normally expect code to be able to work with both. Code that only targets Python 3 can safely type variables and arguments as either `bytes` or `unicode`.

The `basestring` type represents both the types `str` and `unicode`, i.e. all Python text string types in Python 2 and Python 3. This can be used for typing text variables that normally contain Unicode text (at least in Python 3) but must additionally accept the `str` type in Python 2 for backwards compatibility reasons. It is not compatible with the `bytes` type. Its usage should be rare in normal Cython code as the generic `object` type (i.e. untyped code) will normally be good enough and has the additional advantage of supporting the assignment of string subtypes. Support for the `basestring` type was added in Cython 0.20.

8.2 String literals

Cython understands all Python string type prefixes:

- `b'bytes'` for byte strings
- `u'text'` for Unicode strings
- `f'formatted {value}'` for formatted Unicode string literals as defined by **PEP 498** (added in Cython 0.24)

Unprefixed string literals become `str` objects when compiling with language level 2 and `unicode` objects (i.e. Python 3 `str`) with language level 3.

8.3 General notes about C strings

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

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

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

Generally speaking: unless you know what you are doing, avoid using C strings where possible and use Python string objects instead. The obvious exception to this is when passing them back and forth from and to external C code. Also, C++ strings remember their length as well, so they can provide a suitable alternative to Python bytes objects in some cases, e.g. when reference counting is not needed within a well defined context.

8.4 Passing byte strings

It is very easy to pass byte strings between C code and Python. When receiving a byte string from a C library, you can let Cython convert it into a Python byte string by simply assigning it to a Python variable:

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

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

```
py_string = <bytes> c_string
```

This creates a Python byte string object that holds a copy of the original C string. It can be safely passed around in Python code, and will be garbage collected when the last reference to it goes out of scope. It is important to remember that null bytes in the string act as terminator character, as generally known from C. The above will therefore only work correctly for C strings that do not contain null bytes.

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

```
cdef char* c_string = NULL
cdef Py_ssize_t length = 0

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

py_bytes_string = c_string[:length]
```

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

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

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

To convert the byte string back into a C `char*`, use the opposite assignment:

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

This is a very fast operation after which `other_c_string` points to the byte string buffer of the Python string itself. It is tied to the life time of the Python string. When the Python string is garbage collected, the pointer becomes invalid. It is therefore important to keep a reference to the Python string as long as the `char*` is in use. Often enough, this only spans the call to a C function that receives the pointer as parameter. Special care must be taken, however, when the C function stores the pointer for later use. Apart from keeping a Python reference to the string object, no manual memory management is required.

Starting with Cython 0.20, the `bytearray` type is supported and coerces in the same way as the `bytes` type. However, when using it in a C context, special care must be taken not to grow or shrink the object buffer after converting it to a C string pointer. These modifications can change the internal buffer address, which will make the pointer invalid.

8.5 Accepting strings from Python code

The other side, receiving input from Python code, may appear simple at first sight, as it only deals with objects. However, getting this right without making the API too narrow or too unsafe may not be entirely obvious.

In the case that the API only deals with byte strings, i.e. binary data or encoded text, it is best not to type the input argument as something like `bytes`, because that would restrict the allowed input to exactly that type and exclude both subtypes and other kinds of byte containers, e.g. `bytearray` objects or memory views.

Depending on how (and where) the data is being processed, it may be a good idea to instead receive a 1-dimensional memory view, e.g.

```
def process_byte_data(unsigned char[:] data):
    length = data.shape[0]
    first_byte = data[0]
    slice_view = data[1:-1]
    ...
```

Cython's memory views are described in more detail in ../userguide/memoryviews, but the above example already shows most of the relevant functionality for 1-dimensional byte views. They allow for efficient processing of arrays and accept anything that can unpack itself into a byte buffer, without intermediate copying. The processed content can finally be returned in the memory view itself (or a slice of it), but it is often better to copy the data back into a flat and simple `bytes` or `bytearray` object, especially when only a small slice is returned. Since memoryviews do not copy the data, they would otherwise keep the entire original buffer alive. The general idea here is to be liberal with input by accepting any kind of byte buffer, but strict with output by returning a simple, well adapted object. This can simply be done as follows:

```
def process_byte_data(unsigned char[:] data):
    # ... process the data
    if return_all:
        return bytes(data)
    else:
        # example for returning a slice
        return bytes(data[5:35])
```

If the byte input is actually encoded text, and the further processing should happen at the Unicode level, then the right thing to do is to decode the input straight away. This is almost only a problem in Python 2.x, where Python code expects that it can pass a byte string (`str`) with encoded text into a text API. Since this usually happens in more than one place in the module's API, a helper function is almost always the way to go, since it allows for easy adaptation of the input normalisation process later.

This kind of input normalisation function will commonly look similar to the following:

```
from cpython.version cimport PY_MAJOR_VERSION

cdef unicode _ustring(s):
    if type(s) is unicode:
        # fast path for most common case(s)
        return <unicode>s
    elif PY_MAJOR_VERSION < 3 and isinstance(s, bytes):
        # only accept byte strings in Python 2.x, not in Py3
        return (<bytes>s).decode('ascii')
    elif isinstance(s, unicode):
        # an evil cast to <unicode> might work here in some(!) cases,
        # depending on what the further processing does.  to be safe,
        # we can always create a copy instead
        return unicode(s)
    else:
        raise TypeError(...)
```

And should then be used like this:

```
def api_func(s):
    text = _ustring(s)
    ...
```

Similarly, if the further processing happens at the byte level, but Unicode string input should be accepted, then the following might work, if you are using memory views:

```
# define a global name for whatever char type is used in the module
ctypedef unsigned char char_type
```

```
cdef char_type[:] _chars(s):
    if isinstance(s, unicode):
        # encode to the specific encoding used inside of the module
        s = (<unicode>s).encode('utf8')
    return s
```

In this case, you might want to additionally ensure that byte string input really uses the correct encoding, e.g. if you require pure ASCII input data, you can run over the buffer in a loop and check the highest bit of each byte. This should then also be done in the input normalisation function.

8.6 Dealing with "const"

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

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

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

```
cdef extern from "someheader.h":
    ctypedef const char specialChar
    int process_string(const char* s)
    const unsigned char* look_up_cached_string(const unsigned char* key)
```

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

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

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

However, in most other situations, such as for return values and variables that use specifically typedef-ed API types, it does matter and the C compiler will emit at least a warning if used incorrectly. To help with this, you can use the type definitions in the `libc.string` module, e.g.:

```
from libc.string cimport const_char, const_uchar

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

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

8.7 Decoding bytes to text

The initially presented way of passing and receiving C strings is sufficient if your code only deals with binary data in the strings. When we deal with encoded text, however, it is best practice to decode the C byte strings to Python Unicode strings on reception, and to encode Python Unicode strings to C byte strings on the way out.

With a Python byte string object, you would normally just call the `bytes.decode()` method to decode it into a Unicode string:

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

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

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

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

```
cdef char* c_string = NULL
cdef Py_ssize_t length = 0

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

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

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

Again, no bounds checking is done if slice indices are provided, so incorrect indices lead to data corruption and crashes. However, using negative indices is possible since Cython 0.17 and will inject a call to `strlen()` in order to determine the string length. Obviously, this only works for 0-terminated strings without internal null bytes. Text encoded in UTF-8 or one of the ISO-8859 encodings is usually a good candidate. If in doubt, it's better to pass indices that are 'obviously' correct than to rely on the data to be as expected.

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

```
from libc.stdlib cimport free

cdef unicode tounicode(char* s):
    return s.decode('UTF-8', 'strict')

cdef unicode tounicode_with_length(
        char* s, size_t length):
    return s[:length].decode('UTF-8', 'strict')

cdef unicode tounicode_with_length_and_free(
        char* s, size_t length):
    try:
        return s[:length].decode('UTF-8', 'strict')
    finally:
        free(s)
```

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

8.8 Encoding text to bytes

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

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

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

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

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

8.9 C++ strings

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

```
# distutils: language = c++

from libcpp.string cimport string

cdef string s = py_bytes_object
try:
    s.append('abc')
    py_bytes_object = s
finally:
    del s
```

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

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

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

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

```
cdef string s = string(b'abcdefg')

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

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

8.10 Auto encoding and decoding

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

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

```
# cython: c_string_type=unicode, c_string_encoding=utf8

cdef char* c_string = 'abcdefg'

# implicit decoding:
cdef object py_unicode_object = c_string

# explicit conversion to Python bytes:
py_bytes_object = <bytes>c_string
```

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

```
# cython: c_string_type=str, c_string_encoding=ascii

cdef char* c_string = 'abcdefg'

# implicit decoding in Py3, bytes conversion in Py2:
cdef object py_str_object = c_string

# explicit conversion to Python bytes:
py_bytes_object = <bytes>c_string

# explicit conversion to Python unicode:
py_bytes_object = <unicode>c_string
```

The other direction, i.e. automatic encoding to C strings, is only supported for ASCII and the "default encoding", which is usually UTF-8 in Python 3 and usually ASCII in Python 2. CPython handles the memory management in this case by keeping an encoded copy of the string alive together with the original unicode string. Otherwise, there would be no way to limit the lifetime of the encoded string in any sensible way, thus rendering any attempt to extract a C string pointer from it a dangerous endeavour. The following safely converts a Unicode string to ASCII (change `c_string_encoding` to `default` to use the default encoding instead):

```
# cython: c_string_type=unicode, c_string_encoding=ascii

def func():
    ustring = u'abc'
    cdef char* s = ustring
    return s[0]    # returns u'a'
```

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

8.11 Source code encoding

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

```
# -*- coding: ISO-8859-15 -*-
```

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

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

```
print( len(b'abcö') )
```

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

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

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

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

8.12 Single bytes and characters

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

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

```
# -*- coding: ASCII -*-

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

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

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

The explicit coercion works for any C integer type. Values outside of the range of a char or unsigned char will raise an OverflowError at runtime. Coercion will also happen automatically when assigning to a typed variable, e.g.:

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

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

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

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

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

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

8.13 Narrow Unicode builds

In narrow Unicode builds of CPython before version 3.3, i.e. builds where sys.maxunicode is 65535 (such as all Windows builds, as opposed to 1114111 in wide builds), it is still possible to use Unicode character code points that do not fit into the 16 bit wide Py_UNICODE type. For example, such a CPython build will accept the unicode literal u'\U00012345'. However, the underlying system level encoding leaks into Python space in this case, so that the length of this literal becomes 2 instead of 1. This also shows when iterating over it or when indexing into it. The visible substrings are u'\uD808' and u'\uDF45' in this example. They form a so-called surrogate pair that represents the above character.

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

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

However, programmers should be aware that a single Py_UNICODE value (or single 'character' unicode string in CPython) may not be enough to represent a complete Unicode character on narrow platforms. For example, if an independent search for u'\uD808' and u'\uDF45' in a unicode string succeeds, this does not necessarily mean that the character u'\U00012345 is part of that string. It may well be that two different characters are in the string that just happen to share a code unit with the surrogate pair of the character in question. Looking for substrings works correctly because the two code units in the surrogate pair use distinct value ranges, so the pair is always identifiable in a sequence of code points.

As of version 0.15, Cython has extended support for surrogate pairs so that you can safely use an in test to search character values from the full Py_UCS4 range even on narrow platforms:

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

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

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

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

Cython 0.16 and later handles this change internally and does the right thing also for single character values as long as either type inference is applied to untyped variables or the portable `Py_UCS4` type is explicitly used in the source code instead of the platform specific `Py_UNICODE` type. Optimisations that Cython applies to the Python unicode type will automatically adapt to **PEP 393** at C compile time, as usual.

8.14 Iteration

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

```
cdef char* c_string = ...

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

The same applies to bytes objects:

```
cdef bytes bytes_string = ...

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

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

```
cdef unicode ustring = ...

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

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

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

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

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

8.15 Windows and wide character APIs

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

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

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

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

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

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

    ctypedef Py_UNICODE WCHAR
    ctypedef const WCHAR* LPCWSTR
    ctypedef void* HWND

    int MessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption, int uType)

title = u"Windows Interop Demo - Python %d.%d.%d" % sys.version_info[:3]
MessageBoxW(NULL, u"Hello Cython \u263a", title, 0)
```

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

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

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

CHAPTER 9

Memory Allocation

Dynamic memory allocation is mostly a non-issue in Python. Everything is an object, and the reference counting system and garbage collector automatically return memory to the system when it is no longer being used.

When it comes to more low-level data buffers, Cython has special support for (multi-dimensional) arrays of simple types via NumPy, memory views or Python's stdlib array type. They are full featured, garbage collected and much easier to work with than bare pointers in C, while still retaining the speed and static typing benefits. See *Working with Python arrays* and memoryviews.

In some situations, however, these objects can still incur an unacceptable amount of overhead, which can then makes a case for doing manual memory management in C.

Simple C values and structs (such as a local variable `cdef double x`) are usually allocated on the stack and passed by value, but for larger and more complicated objects (e.g. a dynamically-sized list of doubles), the memory must be manually requested and released. C provides the functions `malloc()`, `realloc()`, and `free()` for this purpose, which can be imported in cython from `clibc.stdlib`. Their signatures are:

```
void* malloc(size_t size)
void* realloc(void* ptr, size_t size)
void free(void* ptr)
```

A very simple example of malloc usage is the following:

```
import random
from libc.stdlib cimport malloc, free

def random_noise(int number=1):
    cdef int i
    # allocate number * sizeof(double) bytes of memory
    cdef double *my_array = <double *>malloc(number * sizeof(double))
    if not my_array:
        raise MemoryError()

    try:
        ran = random.normalvariate
        for i in range(number):
            my_array[i] = ran(0,1)

        return [ my_array[i] for i in range(number) ]
    finally:
        # return the previously allocated memory to the system
        free(my_array)
```

Note that the C-API functions for allocating memory on the Python heap are generally preferred over the low-level C functions above as the memory they provide is actually accounted for in Python's internal memory management system. They also have special optimisations for smaller memory blocks, which speeds up their allocation by avoiding costly operating system calls.

The C-API functions can be found in the `cpython.mem` standard declarations file:

```
from cpython.mem cimport PyMem_Malloc, PyMem_Realloc, PyMem_Free
```

Their interface and usage is identical to that of the corresponding low-level C functions.

One important thing to remember is that blocks of memory obtained with `malloc()` or `PyMem_Malloc()` *must* be manually released with a corresponding call to `free()` or `PyMem_Free()` when they are no longer used (and *must* always use the matching type of free function). Otherwise, they won't be reclaimed until the python process exits. This is called a memory leak.

If a chunk of memory needs a larger lifetime than can be managed by a `try..finally` block, another helpful idiom is to tie its lifetime to a Python object to leverage the Python runtime's memory management, e.g.:

```
cdef class SomeMemory:

    cdef double* data

    def __cinit__(self, size_t number):
        # allocate some memory (uninitialised, may contain arbitrary data)
        self.data = <double*> PyMem_Malloc(number * sizeof(double))
        if not self.data:
            raise MemoryError()

    def resize(self, size_t new_number):
        # Allocates new_number * sizeof(double) bytes,
        # preserving the current content and making a best-effort to
        # re-use the original data location.
        mem = <double*> PyMem_Realloc(self.data, new_number * sizeof(double))
        if not mem:
            raise MemoryError()
        # Only overwrite the pointer if the memory was really reallocated.
        # On error (mem is NULL), the originally memory has not been freed.
        self.data = mem

    def __dealloc__(self):
        PyMem_Free(self.data)     # no-op if self.data is NULL
```

CHAPTER 10

Pure Python Mode

In some cases, it's desirable to speed up Python code without losing the ability to run it with the Python interpreter. While pure Python scripts can be compiled with Cython, it usually results only in a speed gain of about 20%-50%.

To go beyond that, Cython provides language constructs to add static typing and cythonic functionalities to a Python module to make it run much faster when compiled, while still allowing it to be interpreted. This is accomplished either via an augmenting `.pxd` file, or via special functions and decorators available after importing the magic `cython` module.

Although it is not typically recommended over writing straight Cython code in a `.pyx` file, there are legitimate reasons to do this - easier testing, collaboration with pure Python developers, etc. In pure mode, you are more or less restricted to code that can be expressed (or at least emulated) in Python, plus static type declarations. Anything beyond that can only be done in .pyx files with extended language syntax, because it depends on features of the Cython compiler.

10.1 Augmenting .pxd

Using an augmenting `.pxd` allows to let the original `.py` file completely untouched. On the other hand, one needs to maintain both the `.pxd` and the `.py` to keep them in sync.

While declarations in a `.pyx` file must correspond exactly with those of a `.pxd` file with the same name (and any contradiction results in a compile time error, see *pxd files*), the untyped definitions in a `.py` file can be overridden and augmented with static types by the more specific ones present in a `.pxd`.

If a `.pxd` file is found with the same name as the `.py` file being compiled, it will be searched for `cdef` classes and `cdef`/`cpdef` functions and methods. The compiler will then convert the corresponding classes/functions/methods in the `.py` file to be of the declared type. Thus if one has a file `A.py`:

```
def myfunction(x, y=2):
    a = x-y
    return a + x * y

def _helper(a):
    return a + 1
```

```
class A:
    def __init__(self, b=0):
        self.a = 3
        self.b = b

    def foo(self, x):
        print x + _helper(1.0)
```

and adds `A.pxd`:

```
cpdef int myfunction(int x, int y=*)
cdef double _helper(double a)

cdef class A:
    cdef public int a,b
    cpdef foo(self, double x)
```

then Cython will compile the `A.py` as if it had been written as follows:

```
cpdef int myfunction(int x, int y=2):
    a = x-y
    return a + x * y

cdef double _helper(double a):
    return a + 1

cdef class A:
    cdef public int a,b
    def __init__(self, b=0):
        self.a = 3
        self.b = b

    cpdef foo(self, double x):
        print x + _helper(1.0)
```

Notice how in order to provide the Python wrappers to the definitions in the `.pxd`, that is, to be accessible from Python,

- Python visible function signatures must be declared as *cpdef* (with default arguments replaced by a * to avoid repetition):

  ```
  cpdef int myfunction(int x, int y=*)
  ```

- C function signatures of internal functions can be declared as *cdef*:

  ```
  cdef double _helper(double a)
  ```

- *cdef* classes (extension types) are declared as *cdef class*;
- *cdef* class attributes must be declared as *cdef public* if read/write Python access is needed, *cdef readonly* for read-only Python access, or plain *cdef* for internal C level attributes;
- *cdef* class methods must be declared as *cpdef* for Python visible methods or *cdef* for internal C methods.

In the example above, the type of the local variable *a* in *myfunction()* is not fixed and will thus be a Python object. To statically type it, one can use Cython's `@cython.locals` decorator (see *Magic Attributes*, and *Magic Attributes within the .pxd*).

Normal Python (def) functions cannot be declared in .pxd files. It is therefore currently impossible to override the types of plain Python functions in .pxd files, e.g. to override types of their local variables. In most cases, declaring them as *cpdef* will work as expected.

10.2 Magic Attributes

Special decorators are available from the magic cython module that can be used to add static typing within the Python file, while being ignored by the interpreter.

This option adds the cython module dependency to the original code, but does not require to maintain a supplementary .pxd file. Cython provides a fake version of this module as *Cython.Shadow*, which is available as *cython.py* when Cython is installed, but can be copied to be used by other modules when Cython is not installed.

10.2.1 "Compiled" switch

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

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

  will behave differently depending on whether or not the code is executed as a compiled extension (.so/.pyd) module or a plain .py file.

10.2.2 Static typing

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

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

  and the second mode as a simple function call:

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

  It can also be used to type class constructors:

  ```
  class A:
      cython.declare(a=cython.int, b=cython.int)
      def __init__(self, b=0):
          self.a = 3
          self.b = b
  ```

  And even to define extension type private, readonly and public attributes:

  ```
  @cython.cclass
  class A:
      cython.declare(a=cython.int, b=cython.int)
      c = cython.declare(cython.int, visibility='public')
  ```

```
    d = cython.declare(cython.int, 5)  # private by default.
    e = cython.declare(cython.int, 5, visibility='readonly')
```

- `@cython.locals` is a decorator that is used to specify the types of local variables in the function body (including the arguments):

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

- `@cython.returns(<type>)` specifies the function's return type.
- `@cython.exceptval(value=None, *, check=False)` specifies the function's exception return value and exception check semantics as follows:

```
@exceptval(-1)               # cdef int func() except -1:
@exceptval(-1, check=False)  # cdef int func() except -1:
@exceptval(check=True)       # cdef int func() except *:
@exceptval(-1, check=True)   # cdef int func() except? -1:
```

- Python annotations can be used to declare argument types, as shown in the following example. To avoid conflicts with other kinds of annotation usages, this can be disabled with the directive `annotation_typing=False`.

```
def func(a_pydict: dict, a_cint: cython.int) -> tuple:
    ...
```

This can be combined with the `@cython.exceptval()` decorator for non-Python return types:

```
@cython.exceptval(-1):
def func(x : cython.int) -> cython.int:
    if x < 0:
        raise ValueError("need integer >= 0")
    return x+1
```

Since version 0.27, Cython also supports the variable annotations defined in PEP 526. This allows to declare types of variables in a Python 3.6 compatible way as follows:

```
def func():
    # Cython types are evaluated as for cdef declarations
    x : cython.int               # cdef int x
    y : cython.double = 0.57721  # cdef double y = 0.57721
    z : cython.float  = 0.57721  # cdef float z  = 0.57721

    # Python types shadow Cython types for compatibility reasons
    a : float = 0.54321          # cdef double a = 0.54321
    b : int = 5                  # cdef object b = 5
    c : long = 6                 # cdef object c = 6

@cython.cclass
class A:
    a : cython.int
    b : cython.int
    def __init__(self, b=0):
        self.a = 3
        self.b = b
```

There is currently no way to express the visibility of object attributes.

10.2.3 C types

There are numerous types built into the Cython module. It provides all the standard C types, namely `char`, `short`, `int`, `long`, `longlong` as well as their unsigned versions `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`. The special `bint` type is used for C boolean values and `Py_ssize_t` for (signed) sizes of Python containers.

For each type, there are pointer types `p_int`, `pp_int`, etc., up to three levels deep in interpreted mode, and infinitely deep in compiled mode. Further pointer types can be constructed with `cython.pointer(cython.int)`, and arrays as `cython.int[10]`. A limited attempt is made to emulate these more complex types, but only so much can be done from the Python language.

The Python types int, long and bool are interpreted as C `int`, `long` and `bint` respectively. Also, the Python builtin types `list`, `dict`, `tuple`, etc. may be used, as well as any user defined types.

Typed C-tuples can be declared as a tuple of C types.

10.2.4 Extension types and cdef functions

- The class decorator `@cython.cclass` creates a `cdef class`.
- The function/method decorator `@cython.cfunc` creates a `cdef` function.
- `@cython.ccall` creates a `cpdef` function, i.e. one that Cython code can call at the C level.
- `@cython.locals` declares local variables (see above). It can also be used to declare types for arguments, i.e. the local variables that are used in the signature.
- `@cython.inline` is the equivalent of the C `inline` modifier.
- `@cython.final` terminates the inheritance chain by preventing a type from being used as a base class, or a method from being overridden in subtypes. This enables certain optimisations such as inlined method calls.

Here is an example of a `cdef` function:

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

10.2.5 Further Cython functions and declarations

- `address` is used in place of the `&` operator:

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

- `sizeof` emulates the *sizeof* operator. It can take both types and expressions.

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

- `struct` can be used to create struct types.:

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

is equivalent to the code:

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

cdef MyStruct a
```

- `union` creates union types with exactly the same syntax as `struct`.
- `typedef` defines a type under a given name:

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

- `cast` will (unsafely) reinterpret an expression type. `cython.cast(T, t)` is equivalent to `<T>t`. The first attribute must be a type, the second is the expression to cast. Specifying the optional keyword argument `typecheck=True` has the semantics of `<T?>t`.

```
t1 = cython.cast(T, t)
t2 = cython.cast(T, t, typecheck=True)
```

10.2.6 Magic Attributes within the .pxd

The special *cython* module can also be imported and used within the augmenting `.pxd` file. For example, the following Python file `dostuff.py`:

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

can be augmented with the following `.pxd` file `dostuff.pxd`:

```
import cython

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

The `cython.declare()` function can be used to specify types for global variables in the augmenting `.pxd` file.

10.3 Tips and Tricks

10.3.1 Calling C functions

Normally, it isn't possible to call C functions in pure Python mode as there is no general way to support it in normal (uncompiled) Python. However, in cases where an equivalent Python function exists, this can be achieved by combining C function coercion with a conditional import as follows:

```
# in mymodule.pxd:

# declare a C function as "cpdef" to export it to the module
cdef extern from "math.h":
    cpdef double sin(double x)


# in mymodule.py:

import cython

# override with Python import if not in compiled code
if not cython.compiled:
    from math import sin

# calls sin() from math.h when compiled with Cython and math.sin() in Python
print(sin(0))
```

Note that the “sin” function will show up in the module namespace of “mymodule” here (i.e. there will be a `mymodule.sin()` function). You can mark it as an internal name according to Python conventions by renaming it to “_sin” in the `.pxd` file as follows:

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

You would then also change the Python import to `from math import sin as _sin` to make the names match again.

10.3.2 Using C arrays for fixed size lists

Since Cython 0.22, C arrays can automatically coerce to Python lists or tuples. This can be exploited to replace fixed size Python lists in Python code by C arrays when compiled. An example:

```
import cython

@cython.locals(counts=cython.int[10], digit=cython.int)
def count_digits(digits):
    """
    >>> digits = '01112222333334445667788899'
    >>> count_digits(map(int, digits))
    [1, 3, 4, 5, 3, 1, 2, 2, 3, 2]
    """
    counts = [0] * 10
    for digit in digits:
        assert 0 <= digit <= 9
        counts[digit] += 1
    return counts
```

In normal Python, this will use a Python list to collect the counts, whereas Cython will generate C code that uses a C array of C ints.

CHAPTER 11

Working with NumPy

Note: Cython 0.16 introduced typed memoryviews as a successor to the NumPy integration described here. They are easier to use than the buffer syntax below, have less overhead, and can be passed around without requiring the GIL. They should be preferred to the syntax presented in this page. See Cython for NumPy users.

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

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

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

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

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

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

11.1 Adding types

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

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

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

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

11.2 Efficient indexing

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

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

These are the needed changes:

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

Usage:

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

Note the importance of this change.

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

11.3 Tuning indexing further

The array lookups are still slowed down by two factors:

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

```
...
cimport cython
@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
def naive_convolve(np.ndarray[DTYPE_t, ndim=2] f, np.ndarray[DTYPE_t, ndim=2] g):
...
```

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

Also, we've disabled the check to wrap negative indices (e.g. g[-1] giving the last value). As with disabling bounds checking, bad things will happen if we try to actually use negative indices with this disabled.

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

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

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

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

The actual rules are a bit more complicated but the main message is clear: Do not use typed objects without knowing that they are not set to None.

11.4 More generic code

It would be possible to do:

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

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

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

CHAPTER 12

Working with Python arrays

Python has a builtin array module supporting dynamic 1-dimensional arrays of primitive types. It is possible to access the underlying C array of a Python array from within Cython. At the same time they are ordinary Python objects which can be stored in lists and serialized between processes when using `multiprocessing`.

Compared to the manual approach with `malloc()` and `free()`, this gives the safe and automatic memory management of Python, and compared to a Numpy array there is no need to install a dependency, as the `array` module is built into both Python and Cython.

12.1 Safe usage with memory views

```
from cpython cimport array
import array
cdef array.array a = array.array('i', [1, 2, 3])
cdef int[:] ca = a

print ca[0]
```

NB: the import brings the regular Python array object into the namespace while the cimport adds functions accessible from Cython.

A Python array is constructed with a type signature and sequence of initial values. For the possible type signatures, refer to the Python documentation for the array module.

Notice that when a Python array is assigned to a variable typed as memory view, there will be a slight overhead to construct the memory view. However, from that point on the variable can be passed to other functions without overhead, so long as it is typed:

```
from cpython cimport array
import array
cdef array.array a = array.array('i', [1, 2, 3])
cdef int[:] ca = a

cdef int overhead(object a):
    cdef int[:] ca = a
    return ca[0]

cdef int no_overhead(int[:] ca):
    return ca[0]

print overhead(a)  # new memory view will be constructed, overhead
print no_overhead(ca)  # ca is already a memory view, so no overhead
```

12.2 Zero-overhead, unsafe access to raw C pointer

To avoid any overhead and to be able to pass a C pointer to other functions, it is possible to access the underlying contiguous array as a pointer. There is no type or bounds checking, so be careful to use the right type and signedness.

```
from cpython cimport array
import array

cdef array.array a = array.array('i', [1, 2, 3])

# access underlying pointer:
print a.data.as_ints[0]

from libc.string cimport memset
memset(a.data.as_voidptr, 0, len(a) * sizeof(int))
```

Note that any length-changing operation on the array object may invalidate the pointer.

12.3 Cloning, extending arrays

To avoid having to use the array constructor from the Python module, it is possible to create a new array with the same type as a template, and preallocate a given number of elements. The array is initialized to zero when requested.

```
from cpython cimport array
import array

cdef array.array int_array_template = array.array('i', [])
cdef array.array newarray

# create an array with 3 elements with same type as template
newarray = array.clone(int_array_template, 3, zero=False)
```

An array can also be extended and resized; this avoids repeated memory reallocation which would occur if elements would be appended or removed one by one.

```
from cpython cimport array
import array

cdef array.array a = array.array('i', [1, 2, 3])
cdef array.array b = array.array('i', [4, 5, 6])

# extend a with b, resize as needed
array.extend(a, b)
```

```
# resize a, leaving just original three elements
array.resize(a, len(a) - len(b))
```

12.4 API reference

12.4.1 Data fields

```
data.as_voidptr
data.as_chars
data.as_schars
data.as_uchars
data.as_shorts
data.as_ushorts
data.as_ints
data.as_uints
data.as_longs
data.as_ulongs
data.as_longlongs  # requires Python >=3
data.as_ulonglongs  # requires Python >=3
data.as_floats
data.as_doubles
data.as_pyunicodes
```

Direct access to the underlying contiguous C array, with given type; e.g., `myarray.data.as_ints`.

12.4.2 Functions

The following functions are available to Cython from the array module:

```
int resize(array self, Py_ssize_t n) except -1
```

Fast resize / realloc. Not suitable for repeated, small increments; resizes underlying array to exactly the requested amount.

```
int resize_smart(array self, Py_ssize_t n) except -1
```

Efficient for small increments; uses growth pattern that delivers amortized linear-time appends.

```
cdef inline array clone(array template, Py_ssize_t length, bint zero)
```

Fast creation of a new array, given a template array. Type will be same as `template`. If zero is `True`, new array will be initialized with zeroes.

```
cdef inline array copy(array self)
```

Make a copy of an array.

```
cdef inline int extend_buffer(array self, char* stuff, Py_ssize_t n) except -1
```

Efficient appending of new data of same type (e.g. of same array type) `n`: number of elements (not number of bytes!)

```
cdef inline int extend(array self, array other) except -1
```

Extend array with data from another array; types must match.

```
cdef inline void zero(array self)
```

Set all elements of array to zero.

CHAPTER 13

Further reading

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

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

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

CHAPTER 14

Related work

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

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

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

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

CHAPTER 15

Appendix: Installing MinGW on Windows

1. Download the MinGW installer from http://www.mingw.org/wiki/HOWTO_Install_the_MinGW_GCC_Compiler_Suite. (As of this writing, the download link is a bit difficult to find; it's under "About" in the menu on the left-hand side). You want the file entitled "Automated MinGW Installer" (currently version 5.1.4).
2. Run it and install MinGW. Only the basic package is strictly needed for Cython, although you might want to grab at least the C++ compiler as well.
3. You need to set up Windows' "PATH" environment variable so that includes e.g. "c:\mingw\bin" (if you installed MinGW to "c:\mingw"). The following web-page describes the procedure in Windows XP (the Vista procedure is similar): http://support.microsoft.com/kb/310519
4. Finally, tell Python to use MinGW as the default compiler (otherwise it will try for Visual C). If Python is installed to "c:\Python27", create a file named "c:\Python27\Lib\distutils\distutils.cfg" containing:

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

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

Bibliography

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

[DevList] Cython developer mailing list: http://mail.python.org/mailman/listinfo/cython-devel

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

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

[ctypes] http://docs.python.org/library/ctypes.html.

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

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

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

[WinInst] https://github.com/cython/cython/wiki/CythonExtensionsOnWindows

Index