Cython Tutorial

Release 0.14.1+

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

April 15, 2016

CHAPTER 1

Calling C functions

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

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

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

```
from libc.stdlib cimport atoi

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

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

```
from cpython.version cimport PY_VERSION_HEX

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

Cython also provides declarations for the C math library:

```
from libc.math cimport sin

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

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

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

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

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

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)
```

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.

CHAPTER 2

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.

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.

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.

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

[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.

[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.

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.

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()
```

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)
```

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

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

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("queue", ["queue.pyx"])]
)
```

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

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

to

[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.

```
ext_modules = [
    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.

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)
```

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

```
cdef int peek(self) except? -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
```

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`.

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

A quick way to provide a Python API for the class is to change the methods from `cdef` to `cpdef`. This will let Cython generate two entry points, one that is callable from normal Python code using the Python call semantics and Python objects as arguments, and one that is callable from C code with fast C semantics and without requiring intermediate argument conversion from or to Python types.

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

```
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
    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)

    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.

Extension types (aka. cdef classes)

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

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

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

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

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

So far our integration example has not been very useful as it only integrates a single hard-coded function. In order to remedy this, without sacrificing speed, we will use a cdef class to represent a function on floating point numbers:

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

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

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

Using this, we can now change our integration example:

```
def integrate(Function f, double a, double b, int N):
  cdef int i
  cdef double s, dx
  if f is None:
      raise ValueError("f cannot be None")
  s = 0
```

```
    dx = (b-a)/N
    for i in range(N):
        s += f.evaluate(a+i*dx)
    return s * dx

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

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

```
>>> 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 period:
        def __get__(self):
            return 1.0 / self. freq
        def __set__(self, value):
            self. freq = 1.0 / value
    <...>
```

CHAPTER 4

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 5

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 then having the same sign as the first (as in C). The C behavior can be obtained, at some speed gain, by enabling the division directive. (Versions prior to Cython 0.12. always followed C semantics.)
- Care is needed with unsigned types. `cdef unsigned n = 10; print(range(-n, n))` will print an empty list, since `-n` wraps around to a large positive integer prior to being passed to the `range` function.
- Python's `float` type actually wraps C `double` values, and Python's `int` type wraps C `long` values.

CHAPTER 6

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.

6.1 Cython Profiling Basics

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

6.1.1 Enable profiling for a complete source file

Profiling is enable 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.

6.1.2 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
```

6.2 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:

```
TODO: how to display this not as code but verbatimly?

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

         10000004 function calls in 6.211 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(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 7

Using Cython with NumPy

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 8

Unicode and passing strings

Similar to the string semantics in Python 3, Cython also strictly separates byte strings and unicode strings. Above all, this means that there is no automatic conversion between byte strings and unicode strings (except for what Python 2 does in string operations). All encoding and decoding must pass through an explicit encoding/decoding step.

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

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

This creates a Python byte string object that holds a copy of the original C string. It can be safely passed around in Python code, and will be garbage collected when the last reference to it goes out of scope. 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.

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

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

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

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

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

8.1 Decoding bytes to text

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

Unicode strings on reception, and to encode Python Unicode strings to C byte strings on the way out.

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

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

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

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

However, this will not work for strings that contain null bytes, and it is 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)

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

The same can 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.

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:

```
cimport python_unicode
cimport stdlib

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:
        stdlib.free(s)
```

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

8.2 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.

8.3 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.

8.4 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.

8.5 Narrow Unicode builds

In narrow Unicode builds of CPython, 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
```

8.6 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.

CHAPTER 9

Pure Python Mode

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

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

9.1 Magic Attributes

The currently supported attributes of the `cython` module are:

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

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

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

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

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

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

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

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

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

```
MyStruct = cython.struct(x=cython.int, y=cython.int, data=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` creates a new type:

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

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

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

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

9.2 Augmenting .pxd

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

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

the file `a.py`:

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

would be interpreted as:

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

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

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

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

```
import cython

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

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

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

9.3 Types

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

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

9.4 Decorators (not yet implemented)

We have settled on `@cython.cclass` for the `cdef class` decorators, and `@cython.cfunc` and `@cython.ccall` for `cdef` and `cpdef` functions (respectively). http://codespeak.net/pipermail/cython-dev/2008-November/002925.html

CHAPTER 10

Further reading

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

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

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

CHAPTER 11

Related work

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

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

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

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

CHAPTER 12

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:\Python26", create a file named "c:\Python26\Lib\distutils\distutils.cfg" containing:

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

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

Bibliography

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

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

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

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