Less copies in Python with the buffer protocol and memoryviews (http://eli.thegreenplace.net/2011/11/28/less-copies-in-python-with-the-buffer-protocol-and-memoryviews)

November 28, 2011 at 07:48 **Tags** Articles (http://eli.thegreenplace.net/tag/articles) , Python (http://eli.thegreenplace.net/tag/python) , Python internals (http://eli.thegreenplace.net/tag/python-internals)

For one of the hobby projects I'm currently hacking on, I recently had to do a lot of binary data processing in memory. Large chunks of data are being read from a file, then examined and modified in memory and finally used to write some reports.

This made me think about the most efficient way to read data from a file into a modifiable memory chunk in Python. As we all know, the standard file `read` method, for a file opened in binary mode, returns a `bytes` object [1], which is immutable:

```
# Snippet #1

f = open(FILENAME, 'rb')
data = f.read()
# oops: TypeError: 'bytes' object does not support item assignment
data[0] = 97
```

This reads the whole contents of the file into `data` - a `bytes` object which is read only. But what if we now want to perform some modifications on the data? Then, we need to somehow get it into a writable object. The most straightforward writable data buffer in Python is a `bytearray`. So we can do this:

```
# Snippet #2

f = open(FILENAME, 'rb')
data = bytearray(f.read())
data[0] = 97 # OK!
```

Now, the `bytes` object returned by `f.read()` is passed into the `bytearray` constructor, which copies its contents into an internal buffer. Since `data` is a `bytearray`, we can manipulate it.

Although it appears that the goal has been achieved, I don't like this solution. The extra copy made by `bytearray` is bugging me. Why is this copy needed? `f.read()` just returns a throwaway buffer we don't need anyway - can't we just initialize the `bytearray` directly, without copying a temporary buffer?

This use case is one of the reasons the Python buffer protocol exists.

The buffer protocol - introduction

The buffer protocol is described in the Python documentation (http://docs.python.org/dev/c-api/buffer.html) and in PEP 3118 (http://www.python.org/dev/peps/pep-3118/) [2]. Briefly, it provides a way for Python objects to expose their internal buffers to other objects. This is useful to avoid extra copies and for certain kinds of sharing. There are many examples of the buffer protocol in use. In the core language - in builtin types such as `bytes` and `bytearray`, in the standard library (for example `array.array` and `ctypes`) and 3rd party libraries (some important Python libraries such as `numpy` and `PIL` rely extensively on the buffer protocol for performance).

There are usually two or more parties involved in each protocol. In the case of the Python buffer protocol, the parties are a "producer" (or "provider") and a "consumer". The producer exposes its internals via the buffer protocol, and the consumer accesses those internals.

Here I want to focus specifically on one use of the buffer protocol that's relevant to this article. The producer is the built-in `bytearray` type, and the consumer is a method in the `file` object named `readinto`.

A more efficient way to read into a bytearray

Here's the way to do what Snippet #2 did, just without the extra copy:

```
# Snippet #3

f = open(FILENAME, 'rb')
data = bytearray(os.path.getsize(FILENAME))
f.readinto(data)
```

First, a `bytearray` is created and pre-allocated to the size of the data we're going to read into it. The pre-allocation is important - since `readinto` directly accesses the internal buffer of `bytearray`, it won't write more than has been allocated. Next, the `file.readinto` method is used to read the data directly into the bytearray's internal storage, without going via temporary buffers.

The result: this code runs ~30% faster than snippet #2 [3].

Variations on the theme

Other objects and modules could be used here. For example, the built-in `array.array` class also supports the buffer protocol, so it can also be written and read from a file directly and efficiently. The same goes for `numpy` arrays. On the consumer side, the `socket` module can also read directly into a buffer with the `read_into` method. I'm sure that it's easy to find many other sample uses of this protocol in Python itself and some 3rd partly libraries - if you find something interesting, please let me know.

The buffer protocol - implementation

Let's see how Snippet #3 works under the hood using the buffer protocol [4]. We'll start with the producer.

`bytearray` declares that it implements the buffer protocol by filling the `tp_as_buffer` slot of its type object [5]. What's placed there is the address of a `PyBufferProcs` structure, which is a simple container for two function pointers:

```
typedef int (*getbufferproc)(PyObject *, Py_buffer *, int);
typedef void (*releasebufferproc)(PyObject *, Py_buffer *);
/* ... */
typedef struct {
     getbufferproc bf_getbuffer;
     releasebufferproc bf_releasebuffer;
} PyBufferProcs;
```

`bf_getbuffer` is the function used to obtain a buffer from the object providing it, and `bf_releasebuffer` is the function used to notify the object that the provided buffer is no longer needed.

The `bytearray` implementation in `Objects/bytearrayobject.c` initializes an instance of `PyBufferProces` thus:

```
static PyBufferProcs bytearray_as_buffer = {
    (getbufferproc)bytearray_getbuffer,
    (releasebufferproc)bytearray_releasebuffer,
};
```

The more interesting function here is `bytearray_getbuffer`:

```
static int
bytearray_getbuffer(PyByteArrayObject *obj, Py_buffer *view, int flags)
{
    int ret;
    void *ptr;
    if (view == NULL) {
        obj->ob_exports++;
        return 0;
    }
    ptr = (void *) PyByteArray_AS_STRING(obj);
    ret = PyBuffer_FillInfo(view, (PyObject*)obj, ptr, Py_SIZE(obj), 0, flags);
    if (ret >= 0) {
        obj->ob_exports++;
    }
    return ret;
}
```

It simply uses the `PyBuffer_FillInfo` API to fill the buffer structure (http://docs.python.org/dev/c-api/buffer.html#the-buffer-structure) passed to it. `PyBuffer_FillInfo` provides a simplified method of filling the buffer structure, which is suitable for unsophisticated objects like `bytearray` (if you want to see a more complex example that has to fill the buffer structure manually, take a look at the corresponding function of `array.array`).

On the consumer side, the code that interests us is the `buffered_readinto` function in `Modules\_io\bufferedio.c` [6]. I won't show its full code here since it's quite complex, but with regards to the buffer protocol, the flow is simple:

1. Use the `PyArg_ParseTuple` function with the `w*` format specifier to parse its argument as a R/W buffer object, which itself calls `PyObject_GetBuffer` - a Python API that invokes the producer's "get buffer" function.
2. Read data from the file directly into this buffer.
3. Release the buffer using the `PyBuffer_Release` API [7], which eventually gets routed to the `bytearray_releasebuffer` function in our case.

To conclude, here's what the call sequence looks like when `f.readinto(data)` is executed in the Python code:

```
buffered_readinto
|
\--> PyArg_ParseTuple(..., "w*", ...)
|    |
|    \--> PyObject_GetBuffer(obj)
|         |
|         \--> obj->ob_type->tp_as_buffer->bf_getbuffer
|
|--> ... read the data
|
\--> PyBuffer_Release
     |
     \--> obj->ob_type->tp_as_buffer->bf_releasebuffer
```

Memory views

The buffer protocol is an internal implementation detail of Python, accessible only on the C-API level. And that's a good thing, since the buffer protocol requires certain low-level behavior such as properly releasing buffers. Memoryview objects (http://docs.python.org/dev/library/stdtypes.html#typememoryview) were created to expose it to a user's Python code in a safe manner:

> memoryview objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

The linked documentation page explains `memoryviews` quite well and should be immediately comprehensible if you've reached so far in this article. Therefore I'm not going to explain how a `memoryview` works, just show some examples of its use.

It is a known fact that in Python, slices on strings and bytes make copies. Sometimes when performance matters and the buffers are large, this is a big waste. Suppose you have a large buffer and you want to pass just half of it to some function (that will send it to a socket or do something else [8]). Here's what happens (annotated Python pseudo-code):

```
mybuf = ... # some large buffer of bytes
func(mybuf[:len(mybuf)//2])
  # passes the first half of mybuf into func
  # COPIES half of mybuf's data to a new buffer
```

The copy can be expensive if there's a lot of data involved. What's the alternative? Using a `memoryview`:

```
mybuf = ... # some large buffer of bytes
mv_mybuf = memoryview(mybuf) # a memoryview of mybuf
func(mv_mybuf[:len(mv_mybuf)//2])
  # passes the first half of mybuf into func as a "sub-view" created
  # by slicing a memoryview.
  # NO COPY is made here!
```

A `memoryview` behaves just like `bytes` in many useful contexts (for example, it supports the mapping protocol) so it provides an adequate replacement if used carefully. The great thing about it is that it uses the buffer protocol beneath the covers to avoid copies and just juggle pointers to data. The performance difference is dramatic - I timed a 300x speedup on slicing out a half of a 1MB `bytes` buffer when using a `memoryview` as demonstrated above. And this speedup will get larger with larger buffers, since it's O(1) vs. the O(n) of copying.

But there's more. On writable producers such as `bytearray`, a `memoryview` creates a writable view that can be modified:

```
>>> buf = bytearray(b'abcdefgh')
>>> mv = memoryview(buf)
>>> mv[4:6] = b'ZA'
>>> buf
bytearray(b'abcdZAgh')
```

This gives us a way to do something we couldn't achieve by any other means - read from a file (or receive from a socket) *directly into the middle of some existing buffer* [9]:

```
buf = bytearray(...) # pre-allocated to the needed size
mv = memoryview(buf)
numread = f.readinto(mv[some_offset:])
```

Conclusion

This article demonstrated the Python buffer protocol, showing both how it works and what it can be used for. The main *use* of the buffer protocol to the Python programmer is optimization of certain patterns of coding, by avoiding unnecessary data copies.

Any mention of optimization in Python code is sure to draw fire from people claiming that if I want to write fast code, I shouldn't use Python at all. But I disagree. Python these days is fast enough to be suitable for many tasks that were previously only in the domain of C/C++. I want to keep using it while I can, and only resort to low-level C/C++ when I must.

Employing the buffer protocol to have zero-copy buffer manipulations on the Python level is IMHO a huge boon that can stall (or even avoid) the transition of some performance-sensitive code from Python to C. That's because when dealing with data processing, we often use a lot of C

APIs anyway, the only Python overhead being the passing of data between these APIs. A speed boost in this code can make a huge difference and bring the Python code very close to the performance we could have with plain C.

The article also gave a glimpse into one aspect of the implementation of Python, hopefully showing that it's not difficult at all to dive right into the code and understand how Python does something it does.

---

[1] In this article I'm focusing on the latest Python 3.x, although most of it also applies to Python 2.7

[2] The buffer protocol existed in Python prior to 2.6, but was then greatly enhanced. The PEP also describes the change that was made to the buffer protocol with the move to Python 3.x (and later backported to the 2.x line starting with 2.6).

[3] This is on the latest build of Python 3.3. Roughly the same speedup can be seen on Python 2.7. With Python 3.2 there appears to be a speed regression that makes the two snippets perform similarly, but it has been fixed in 3.3

Another note on the benchmarking: it's recommended to use large files (say, ~100 MB and up) to get reliable measurements. For small files too many irrelevant factors come into play and offset the benchmarks. In addition, the code should be run in a loop to avoid differences due to warm/cold disk cache issues. I'm using the `timeit` module, which is perfect for this purpose.

[4] All the code displayed here is taken from the latest development snapshot of Python 3.3 (`default` branch).

[5] Type objects are a fascinating aspect of Python's implementation, and I hope to cover it in a separate article one day. Briefly, it allows Python objects to declare which services they provide (or, in other terms, which interfaces they implement). More information can be found in the documentation (http://docs.python.org/dev/c-api/typeobj.html).

[6] Since the built-in `open` function, when asked to open a file in binary mode for reading, returns an `io.BufferedReader` object by default. This can be controlled with the *buffering* argument.

[7] Releasing the buffer structure is an important part of the buffer protocol. Each time it's requested for a buffer, `bytearray` increments a reference count, and decrements it when the buffer is released. While the refcount is positive (meaning that there are consumer objects directly relying on the internal buffer), `bytearray` won't agree to resize or do other operations that may invalidate the internal buffer. Otherwise, this would be an avenue for insidious memory bugs.

[8] Networking code is actually a common use case. When sending data over sockets, it's frequently sliced and diced to build frames. This can involve a lot of copying. Other data-munging applications such as encryption and compression are also culprits.

[9] This code snippet was borrowed from Antoine Pitrou's post on python-dev.

Comments

Pandu POLUAN • 6 months ago

THANK YOU!!

Good grief, this should be a great boost for my Python project, **sims3py**. Slices happen *really* often in the core classes of my project; so even though I might not see a 300x speedup, I'm really certain your tips will significantly increase the performance of my project.

You really wrote the topic well, I can quickly understand.

Again, thank you! If you ever visit my town, give me a call and I'll treat you to a cold one :-)

csar • a month ago

I just stumbled upon memoryview and bytearray, which are both great for my project :)
The only thing that really bugs me is that i can't get a bytearray back from a memoryview.

In a bytearray, i can do all sorts of string operations, like find / rfind etc.
In a memoryview, i can't do that. And using memoryview.tobytes() is useless, as it 1) copies the data, which i want to prevent using memoryviews, and 2) returns a read-only bytes object.

Any idea on how to do something like bytearray.frombuffer(memoryview)?
Otherwise i'd have to implement my own find function, which will most likey suck in comparison to bytearray.find's performance.

EDIT:
Nevermind, ctypes to the rescue! Let's just modify what the bytearray python object points to:

```
def address_to_bytearray(addr, size):
b = bytearray()
ba = PyByteArray.from_address(id(b))
ba.ob_size = size
ba.ob_alloc = size + 1
ba.ob_bytes = ctypes.cast(addr, ctypes.c_char_p)
return b
```

(after defining appropriate ctypes Structures).

↑ Back to top