Why Python is Slow: Looking Under the Hood

May 09, 2014

We've all heard it before: Python is slow.

When I teach courses on Python for scientific computing, I make this point very early (http://nbviewer.ipython.org/github/jakevdp/2013_fall_ASTR599/blob/master/notebooks/11_EfficientNumpy.ipynb) in the course, and tell the students why: it boils down to Python being a dynamically typed, interpreted language, where values are stored not in dense buffers but in scattered objects. And then I talk about how to get around this by using NumPy, SciPy, and related tools for vectorization of operations and calling into compiled code, and go on from there.

But I realized something recently: despite the relative accuracy of the above statements, the words "dynamically-typed-interpreted-buffers-vectorization-compiled" probably mean very little to somebody attending an intro programming seminar. The jargon does little to enlighten people about what's actually going on "under the hood", so to speak.

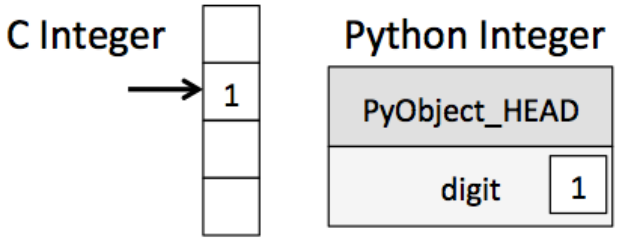
So I decided I would write this post, and dive into the details that I usually gloss over. Along the way, we'll take a look at using Python's standard library to introspect the goings-on of CPython itself. So whether you're a novice or experienced programmer, I hope you'll learn something from the following exploration.

Why Python is Slow

Python is slower than Fortran and C for a variety of reasons:

1. Python is Dynamically Typed rather than Statically Typed.

What this means is that at the time the program executes, the interpreter doesn't know the type of the variables that are defined. The difference between a C variable (I'm using C as a stand-in for compiled languages) and a Python variable is summarized by this diagram:



For a variable in C, the compiler knows the type by its very definition. For a variable in Python, all you know at the time the program executes is that it's some sort of Python object.

So if you write the following in C:

```
/* C code */
int a = 1;
int b = 2;
int c = a + b;
```

the C compiler knows from the start that `a` and `b` are integers: they simply can't be anything else! With this knowledge, it can call the routine which adds two integers, returning another integer which is just a simple value in memory. As a rough schematic, the sequence of events looks like this:

C Addition

1. Assign `<int> 1` to `a`
2. Assign `<int> 2` to `b`
3. call `binary_add<int, int>(a, b)`
4. Assign the result to c

The equivalent code in Python looks like this:

```
# python code
a = 1
b = 2
c = a + b
```

here the interpreter knows only that `1` and `2` are objects, but not what type of object they are. So the The interpreter must inspect `PyObject_HEAD` for each variable to find the type information, and then call the appropriate summation routine for the two types. Finally it must create and initialize a new Python object to hold the return value. The sequence of events looks roughly like this:

Python Addition

1. Assign `1` to `a`

- **1a.** Set `a->PyObject_HEAD->typecode` to integer
- **1b.** Set `a->val = 1`

2. Assign `2` to `b`

- **2a.** Set `b->PyObject_HEAD->typecode` to integer

- **2b.** Set `b->val = 2`

3. call `binary_add(a, b)`

- **3a.** find typecode in `a->PyObject_HEAD`
- **3b.** `a` is an integer; value is `a->val`
- **3c.** find typecode in `b->PyObject_HEAD`
- **3d.** `b` is an integer; value is `b->val`
- **3e.** call `binary_add<int, int>(a->val, b->val)`
- **3f.** result of this is `result`, and is an integer.

4. Create a Python object `c`

- **4a.** set `c->PyObject_HEAD->typecode` to integer
- **4b.** set `c->val` to `result`

The dynamic typing means that there are a lot more steps involved with any operation. This is a primary reason that Python is slow compared to C for operations on numerical data.

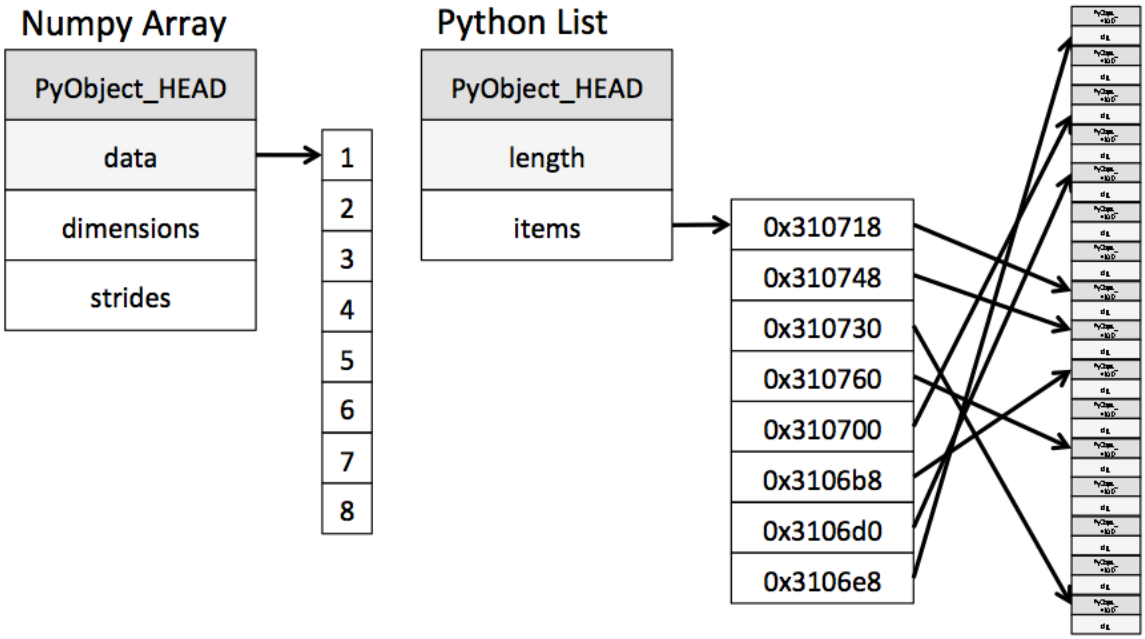
2. Python is interpreted rather than compiled.

We saw above one difference between interpreted and compiled code. A smart compiler can look ahead and optimize for repeated or unneeded operations, which can result in speed-ups. Compiler optimization is its own beast, and I'm personally not qualified to say much about it, so I'll stop there. For some examples of this in action, you can take a look at my previous post (http://jakevdp.github.io/blog/2013/06/15/numba-vs-cython-take-2/) on Numba and Cython.

3. Python's object model can lead to inefficient memory access

We saw above the extra type info layer when moving from a C integer to a Python integer. Now imagine you have many such integers and want to do some sort of batch operation on them. In Python you might use the standard `List` object, while in C you would likely use some sort of buffer-based array.

A NumPy array in its simplest form is a Python object build around a C array. That is, it has a pointer to a *contiguous* data buffer of values. A Python list, on the other hand, has a pointer to a contiguous buffer of pointers, each of which points to a Python object which in turn has references to its data (in this case, integers). This is a schematic of what the two might look like:



It's easy to see that if you're doing some operation which steps through data in sequence, the numpy layout will be much more efficient than the Python layout, both in the cost of storage and the cost of access.

So Why Use Python?

Given this inherent inefficiency, why would we even think about using Python? Well, it comes down to this: Dynamic typing makes Python **easier to use** than C. It's extremely **flexible and forgiving**, this flexibility leads to **efficient use of development time**, and on those occasions that you really need the optimization of C or Fortran, **Python offers easy hooks into compiled libraries**. It's why Python use within many scientific communities has been continually growing. With all that put together, Python ends up being an extremely efficient language for the overall task of doing science with code.

Python meta-hacking: Don't take my word for it

Above I've talked about some of the internal structures that make Python tick, but I don't want to stop there. As I was putting together the above summary, I started hacking around on the internals of the Python language, and found that the process itself is pretty enlightening.

In the following sections, I'm going to *prove* to you that the above information is correct, by doing some hacking to expose Python objects using Python itself. Please note that everything below is written using **Python 3.4**. Earlier versions of Python have a slightly different internal object structure, and later versions may tweak this further. Please make sure to use the correct version! Also, most of the code below assumes a 64-bit CPU. If you're on a 32-bit platform, some of the C types below will have to be adjusted to account for this difference.

In [1]:
```
import sys
print("Python version =", sys.version[:5])
```

```
Python version = 3.4.0
```

Digging into Python Integers

Integers in Python are easy to create and use:

```
In [2]: x = 42
        print(x)
```

```
42
```

But the simplicity of this interface belies the complexity of what is happening under the hood. We briefly discussed the memory layout of Python integers above. Here we'll use Python's built-in `ctypes` module to introspect Python's integer type from the Python interpreter itself. But first we need to know exactly what a Python integer looks like at the level of the C API.

The actual `x` variable in CPython is stored in a structure which is defined in the CPython source code, in Include/longintrepr.h (http://hg.python.org/cpython/file/3.4/Include/longintrepr.h/#l89)

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

The `PyObject_VAR_HEAD` is a macro which starts the object off with the following struct, defined in Include/object.h (http://hg.python.org/cpython/file/3.4/Include/object.h#l111):

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

... and includes a `PyObject` element, which is also defined in Include/object.h (http://hg.python.org/cpython/file/3.4/Include/object.h#l105):

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

here `_PyObject_HEAD_EXTRA` is a macro which is not normally used in the Python build.

With all this put together and typedefs/macros unobfuscated, our integer object works out to something like the following structure:

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

The `ob_refcnt` variable is the reference count for the object, the `ob_type` variable is a pointer to the structure containing all the type information and method definitions for the object, and the `ob_digit` holds the actual numerical value.

Armed with this knowledge, we'll use the `ctypes` module to start looking into the actual object structure and extract some of the above information.

We start with defining a Python representation of the C structure:

In [3]:
```
import ctypes

class IntStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_size", ctypes.c_ulong),
                ("ob_digit", ctypes.c_long)]

    def __repr__(self):
        return ("IntStruct(ob_digit={self.ob_digit}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

Now let's look at the internal representation for some number, say 42. We'll use the fact that in CPython, the `id` function gives the memory location of the object:

In [4]:
```
num = 42
IntStruct.from_address(id(42))
```

Out[4]:
```
IntStruct(ob_digit=42, refcount=35)
```

The `ob_digit` attribute points to the correct location in memory!

But what about `refcount`? We've only created a single value: why is the reference count so much greater than one?

Well it turns out that Python uses small integers *a lot*. If a new `PyObject` were created for each of these integers, it would take a lot of memory. Because of this, Python implements common integer values as **singletons**: that is, only one copy of these numbers exist in memory. In other words, every time you create a new Python integer in this range, you're simply creating a reference to the singleton with that value:

In [5]:
```
x = 42
y = 42
id(x) == id(y)
```

Out[5]:
```
True
```

Both variables are simply pointers to the same memory address. When you get to much bigger integers (larger than 255 in Python 3.4), this is no longer true:

In [6]:
```
x = 1234
y = 1234
id(x) == id(y)
```

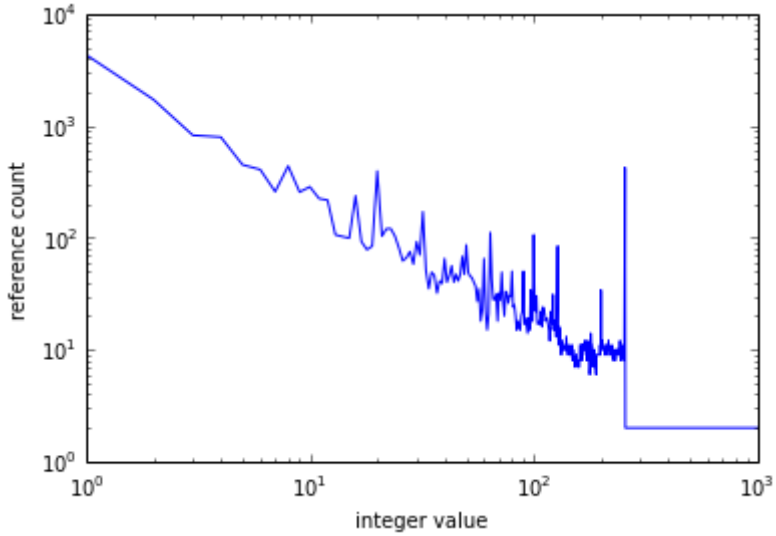
Out[6]:
```
False
```

Just starting up the Python interpreter will create a lot of integer objects; it can be interesting to take a look at how many references there are to each:

In [7]:
```
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import sys
plt.loglog(range(1000), [sys.getrefcount(i) for i in range(1000)])
plt.xlabel('integer value')
plt.ylabel('reference count')
```

Out[7]: <matplotlib.text.Text at 0x106866ac8>



We see that zero is referenced several thousand times, and as you may expect, the frequency of references generally decreases as the value of the integer increases.

Just to further make sure that this is behaving as we'd expect, let's make sure the `ob_digit` field holds the correct value:

In [8]:
```
all(i == IntStruct.from_address(id(i)).ob_digit
    for i in range(256))
```

Out[8]: True

If you go a bit deeper into this, you might notice that this does not hold for numbers larger than 256: it turns out that some bit-shift gymnastics are performed in Objects/longobject.c (http://hg.python.org/cpython/file/3.4/Objects/longobject.c#l232), and these change the way large integers are represented in memory.

I can't say that I fully understand why exactly that is happening, but I imagine it has something to do with Python's ability to efficiently handle integers past the overflow limit of the long int data type, as we can see here:

In [9]:
```
2 ** 100
```

Out[9]: 1267650600228229401496703205376

That number is much too long to be a `long`, which can only hold 64 bits worth of values (that is, up to $\sim 2^{64}$)

Digging into Python Lists

Let's apply the above ideas to a more complicated type: Python lists. Analogously to integers, we find the definition of the list object itself in Include/listobject.h (http://hg.python.org/cpython/file/3.4/Include/listobject.h#l23):

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

Again, we can expand the macros and de-obfuscate the types to see that the structure is effectively the following:

```
typedef struct {
    long ob_refcnt;
    PyTypeObject *ob_type;
    Py_ssize_t ob_size;
    PyObject **ob_item;
    long allocated;
} PyListObject;
```

Here the `PyObject **ob_item` is what points to the contents of the list, and the `ob_size` value tells us how many items are in the list.

In [10]:
```
class ListStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_size", ctypes.c_ulong),
                ("ob_item", ctypes.c_long),  # PyObject** pointer cast to long
                ("allocated", ctypes.c_ulong)]

    def __repr__(self):
        return ("ListStruct(len={self.ob_size}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

Let's try it out:

In [11]:
```
L = [1,2,3,4,5]
ListStruct.from_address(id(L))
```

Out[11]:
```
ListStruct(len=5, refcount=1)
```

Just to make sure we've done things correctly, let's create a few extra references to the list, and see how it affects the reference count:

In [12]:
```
tup = [L, L]  # two more references to L
ListStruct.from_address(id(L))
```

Out[12]:
```
ListStruct(len=5, refcount=3)
```

Now let's see about finding the actual elements within the list.

As we saw above, the elements are stored via a contiguous array of `PyObject` pointers. Using `ctypes`, we can actually create a compound structure consisting of our `IntStruct` objects from before:

In [13]:
```

```

```
# get a raw pointer to our list
Lstruct = ListStruct.from_address(id(L))

# create a type which is an array of integer pointers the same length as L
PtrArray = Lstruct.ob_size * ctypes.POINTER(IntStruct)

# instantiate this type using the ob_item pointer
L_values = PtrArray.from_address(Lstruct.ob_item)
```

Now let's take a look at the values in each of the items:

In [14]:
```
[ptr[0] for ptr in L_values]  # ptr[0] dereferences the pointer
```

Out[14]:
```
[IntStruct(ob_digit=1, refcount=5296),
 IntStruct(ob_digit=2, refcount=2887),
 IntStruct(ob_digit=3, refcount=932),
 IntStruct(ob_digit=4, refcount=1049),
 IntStruct(ob_digit=5, refcount=808)]
```

We've recovered the `PyObject` integers within our list! You might wish to take a moment to look back up to the schematic of the List memory layout above, and make sure you understand how these `ctypes` operations map onto that diagram.

Digging into NumPy arrays

Now, for comparison, let's do the same introspection on a numpy array. I'll skip the detailed walk-through of the NumPy C-API array definition; if you want to take a look at it, you can find it in numpy/core/include/numpy/ndarraytypes.h (https://github.com/numpy/numpy/blob/maintenance/1.8.x/numpy/core/include/numpy/ndarraytypes.h#L646)

Note that I'm using NumPy version 1.8 here; these internals may have changed between versions, though I'm not sure whether this is the case.

In [15]:
```
import numpy as np
np.__version__
```

Out[15]:
```
'1.8.1'
```

Let's start by creating a structure that represents the numpy array itself. This should be starting to look familiar...

We'll also add some custom properties to access Python versions of the shape and strides:

In [16]:
```

```

```
class NumpyStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_data", ctypes.c_long),  # char* pointer cast to long
                ("ob_ndim", ctypes.c_int),
                ("ob_shape", ctypes.c_voidp),
                ("ob_strides", ctypes.c_voidp)]

    @property
    def shape(self):
        return tuple((self.ob_ndim * ctypes.c_int64).from_address(self.ob_shape))

    @property
    def strides(self):
        return tuple((self.ob_ndim * ctypes.c_int64).from_address(self.ob_strides))

    def __repr__(self):
        return ("NumpyStruct(shape={self.shape}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

Now let's try it out:

In [17]:
```
x = np.random.random((10, 20))
xstruct = NumpyStruct.from_address(id(x))
xstruct
```

Out[17]:
```
NumpyStruct(shape=(10, 20), refcount=1)
```

We see that we've pulled out the correct shape information. Let's make sure the reference count is correct:

In [18]:
```
L = [x,x,x]  # add three more references to x
xstruct
```

Out[18]:
```
NumpyStruct(shape=(10, 20), refcount=4)
```

Now we can do the tricky part of pulling out the data buffer. For simplicity we'll ignore the strides and assume it's a C-contiguous array; this could be generalized with a bit of work.

In [19]:
```
x = np.arange(10)
xstruct = NumpyStruct.from_address(id(x))
size = np.prod(xstruct.shape)

# assume an array of integers
arraytype = size * ctypes.c_long
data = arraytype.from_address(xstruct.ob_data)

[d for d in data]
```

Out[19]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The `data` variable is now a view of the contiguous block of memory defined in the NumPy array! To show this, we'll change a value in the array...

In [20]:
```
x[4] = 555
[d for d in data]
```

Out[20]:
```
[0, 1, 2, 3, 555, 5, 6, 7, 8, 9]
```

... and observe that the data view changes as well. Both `x` and `data` are pointing to the same contiguous block of memory.

Comparing the internals of the Python list and the NumPy ndarray, it is clear that NumPy's arrays are **much, much** simpler for representing a list of identically-typed data. That fact is related to what makes it more efficient for the compiler to handle as well.

Just for fun: a few "never use these" hacks

Using `ctypes` to wrap the C-level data behind Python objects allows you to do some pretty interesting things. With proper attribution to my friend James Powell, I'll say it here: seriously, don't use this code (http://seriously.dontusethiscode.com/). While nothing below should actually be used (ever), I still find it all pretty interesting!

Modifying the Value of an Integer

Inspired by this Reddit post (http://www.reddit.com/r/Python/comments/2441cv/can_you_change_the_value_of_1/), we can actually modify the numerical value of integer objects! If we use a common number like `0` or `1`, we're very likely to crash our Python kernel. But if we do it with less important numbers, we can get away with it, at least briefly.

Note that this is a *really, really* bad idea. In particular, if you're running this in an IPython notebook, you might corrupt the IPython kernel's very ability to run (because you're screwing with the variables in its runtime). Nevertheless, we'll cross our fingers and give it a shot:

In [21]:
```
# WARNNG: never do this!
id113 = id(113)
iptr = IntStruct.from_address(id113)
iptr.ob_digit = 4  # now Python's 113 contains a 4!

113 == 4
```

Out[21]:
```
True
```

But note now that we can't set the value back in a simple manner, because the true value `113` no longer exists in Python!

In [22]:
```
113
```

Out[22]:
```
4
```

In [23]:
```
112 + 1
```

Out[23]:
```
4
```

One way to recover is to manipulate the bytes directly. We know that $113 = 7 \times 16^1 + 1 * 16^0$, so **on a little-endian 64-bit system running Python 3.4**, the following should work:

In [24]:
```
ctypes.cast(id113, ctypes.POINTER(ctypes.c_char))[3 * 8] = b'\x71'
112 + 1
```

Out[24]:
```
113
```

and we're back!

Just in case I didn't stress it enough before: **never do this.**

In-place Modification of List Contents

Above we did an in-place modification of a value in a numpy array. This is easy, because a numpy array is simply a data buffer. But might we be able to do the same thing for a list? This gets a bit more tricky, because lists store *references* to values rather than the values themselves. And to not crash Python itself, you need to be very careful to keep track of these reference counts as you muck around. Here's how it can be done:

In [25]:
```
# WARNING: never do this!
L = [42]
Lwrapper = ListStruct.from_address(id(L))
item_address = ctypes.c_long.from_address(Lwrapper.ob_item)
print("before:", L)

# change the c-pointer of the list item
item_address.value = id(6)

# we need to update reference counts by hand
IntStruct.from_address(id(42)).ob_refcnt -= 1
IntStruct.from_address(id(6)).ob_refcnt += 1

print("after: ", L)
```

```
before: [42]
after:  [6]
```

Like I said, you should never use this, and I honestly can't think of any reason why you would want to. But it gives you an idea of the types of operations the interpreter has to do when modifying the contents of a list. Compare this to the NumPy example above, and you'll see one reason why Python lists have more overhead than Python arrays.

Meta Goes Meta: a self-wrapping Python object

Using the above methods, we can start to get even stranger. The `Structure` class in `ctypes` is itself a Python object, which can be seen in Modules/_ctypes/ctypes.h (http://hg.python.org/cpython/file/3.4/Modules/_ctypes/ctypes.h#l46). Just as we wrapped ints and lists, we can wrap structures themselves as follows:

In [26]:

```
class CStructStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_ptr", ctypes.c_long),  # char* pointer cast to long
                    ]

    def __repr__(self):
        return ("CStructStruct(ptr=0x{self.ob_ptr:x}, "
                "refcnt={self.ob_refcnt})").format(self=self)
```

Now we'll attempt to make a structure that wraps itself. We can't do this directly, because we don't know at what address in memory the new structure will be created. But what we can do is create a *second* structure wrapping the first, and use this to modify its contents in-place!

We'll start by making a temporary meta-structure and wrapping it:

In [27]:
```
tmp = IntStruct.from_address(id(0))
meta = CStructStruct.from_address(id(tmp))

print(repr(meta))
```

```
CStructStruct(ptr=0x10023ef00, refcnt=1)
```

Now we add a third structure, and use it to adjust the memory value of the second in-place:

In [28]:
```
meta_wrapper = CStructStruct.from_address(id(meta))
meta_wrapper.ob_ptr = id(meta)

print(meta.ob_ptr == id(meta))
print(repr(meta))
```

```
True
CStructStruct(ptr=0x106d828c8, refcnt=7)
```

We now have a self-wrapping Python structure!

Again, I can't think of any reason you'd ever want to do this. And keep in mind there is noting groundbreaking about this type of self-reference in Python – due to its dynamic typing, it is realatively straightforward to do things like this without directly hacking the memory:

In [29]:
```
L = []
L.append(L)
print(L)
```

```
[[...]]
```

Conclusion

Python is slow. And one big reason for that, as we've seen, is the type indirection under the hood which makes Python quick, easy, and fun for the developer. And as we've seen, Python itself offers tools that can be used to hack into the Python objects themselves.

I hope that this was made more clear through this exploration of the differences between various objects, and some liberal mucking around in the internals of CPython itself. This exercise was extremely enlightening for me, and I hope it was for you as well... Happy hacking!

This blog post was written entirely in the IPython Notebook. The full notebook can be downloaded here (http://jakevdp.github.io/downloads/notebooks/WhyPythonIsSlow.ipynb), or viewed statically here (http://nbviewer.ipython.org/url/jakevdp.github.io/downloads/notebooks/WhyPythonIsSlow.ipynb).

Posted by Jake Vanderplas May 09, 2014

Comments

rgbkrk • 2 years ago

I read "never do this" as "you absolutely must try this but don't do it in production".

38 • Reply • Share ›

Chris Warburton • 2 years ago

I think many of the "reasons" you state are actually *symptoms* of Python's slowness.

1) The reason "c = a + b" is slow in Python is not completely down to dynamic typing.
1a) The "1" and "2" objects are constants (modulo the hacks you mention), so there's no need to create new Python objects for "a" and "b" and "c" like you describe; they can just be pointers to pre-existing global constants. You mention that this is the case when "digging into integers".
1b) There's actually no need to have such constant objects at all! We can use tagged pointers to store the values of a, b and c as raw ints with a bitmask. Some Javascript implementations do this, eg. http://nikic.github.io/2012/02...
1c) Looking up the definition of "+" based on the type isn't actually "dynamic typing", it's "dynamic dispatch" (combined with operator overloading). Dynamically typed languages may not have dynamic dispatch (eg. Scheme) and statically typed languages may have it (eg. C++ vtables). In other words, Python's "a + b" just-so-happens to be sugar for "a.__add__(b)", which clearly depends on "a". If we used a function call, like "my_add(a, b)", there would be no dynamic dispatch (or operator overloading, for that matter).

2) There's no need for Python to be interpreted, other than historical accident. For example Ian Piumarta's "Id" object model for C is at least as dynamic as Python's

13 • Reply • Share ›

jakevdp Mod → Chris Warburton • 2 years ago

Well put, but there's one point I think you miss.

Scientists are not "speeding up Python with C extensions". Scientists are using Python as a convenient wrapper to bomb-proof legacy compiled code, and the ease of doing this is one of the main reasons Python has so much uptake in the scientific community. For this reason, scientists don't really care about the slowness of Python (or its implementation, or its symptoms, or whatever you want to call it). It's why no scientist I know pays much attention to PyPy, Pyston, or similar efforts.

Python's usefulness in science is as a well-designed, beautiful, and easy to use glue for the numerical codes we depend on. The rest doesn't really matter.

6 • Reply • Share ›

craigyk → jakevdp • 2 years ago

I'm a scientist and I definitely care that Python is slow (and it is SLOW). Especially when other dynamic languages exist whose implementations are so much faster. Without numpy, Python for science would have been a no go. I worked with a group that was one of the earliest adopters of Python for scientific work (2000ish). If nodejs had a numpy equivalent and operator overloading it would blow scientific use of python for new projects out of the water. Same for Go. The reason scientists don't pay attention to PyPy is because it doesn't have a feature complete numpy/scipy.

Even numpy/scipy is sometimes too slow. The overhead before it gets to the C/Fortran code isn't insignificant, and can lead to a lot of unnecessary copying. IO can be a major bottleneck in some scientific code.

The builtin python multiprocessing and subprocess libraries are also not very good. With the builtin stuff Python is merely an OK glue language (luckily there is better stuff out there). And the tools for calling C code from python are a bit hacky- they definitely take away from the beauty of idiomatic python.

and "bomb-proofing" code? the lack of types and the little code testing done in the scientific community has led to some of the most bomb-happy codebases I've seen.

All that said, I really like Python (I've been using it for over 13 years), but I am extremely excited that things like Julia might replace python in a couple of years. Numba looks promising though. Certainly their package manager (conda) is one of the nicer new python projects I've seen in a while.

10 • Reply • Share ›

jakevdp Mod → craigyk • 2 years ago

I'm not saying Python scripts themselves are bomb-proof. I was referring to well-tested and long-used legacy codes such as those in NetLib. SciPy, for example, started as not much more than a set of Python wrappers around NetLib libraries. Before Python, people were using config files to call those libraries from the command line. Now the process is much more streamlined.

If you're a scientist, my hunch is that most of your actual computation takes

If you're a scientist, my hunch is that most of your actual computation takes place in that type of compiled extension (via numpy, scipy, etc.). So the slowness of Python doesn't really matter. And, as you mentioned, PyPy is essentially useless, so you'll ignore it.

You might quibble with that notion, but the strength of the Python scientific community in the face of Python's slowness supports my assertion.

There are languages out there that might replace Python someday, but PyPy and Go will never do it. It's simply too hard to tap into NetLib and other bomb-proof legacy code. Julia is where my money is.

7 ^ | v • Reply • Share ›

Daniel Halperin • 2 years ago

Those are a few awesome lines of Python to hide in someone's code when you want to troll them!

7 ^ | v • Reply • Share ›

Malleus Veritas • 2 years ago

The problem isn't so much that Python is slow compared to C or C++.

The problem is that Python runs at a fraction of the speed of OTHER DYNAMIC LANGUAGES like Perl and Ruby.

Compare the trivial example of printing the sum of all integers from 1 to 100,000,000 with a for loop.
bench.py:

```
#!/usr/bin/python
a = 0
for i in range(1, 100000001):
a += i
# end
print a
```

bench.pl:

```
#!/usr/bin/perl
$a = 0;
for $i ( 1 .. 100000000) {
$a += $i;
```

see more

6 ^ | v • Reply • Share ›

Chris Warburton ➚ Malleus Veritas • 2 years ago

Your bench.py isn't very definitive, since it's clear that xrange will be faster than range. On my laptop this simple change took your original bench.py from 11 seconds down to 7.1 seconds. The speedup on your edited version is more modest, down from 9.2 seconds to 8.7 seconds.

3 ^ | v • Reply • Share ›

Maciek Dems ➚ Chris Warburton • 2 years ago

Efficient programming in Python needs different thinking. All the operations should be vectorized. This is especially true when using numpy or scipy...

My timings:

bench.pl: 4.61s
bench.py: 7.68s
bench2.py (with range replaced with xrange): 6.29s
bench3.py (the right way, code below): 0.64s

The code of bench3.py:

```
#!/usr/bin/python
a = sum(xrange(1, 100000001))
print a
```

23 ▲ ▼ • Reply • Share ›

BuckRogers • 2 years ago

Nice writeup. In my opinion though, it boils down to ONE thing, not many. It's because the default implementation is interpreted. That's it. It's not dynamic typing or anything else. JIT compilers still run the same dynamically typed Python codebase and have no performance issue. You could revise your conclusion to make this crystal clear. The Python spec has absolutely no inherent design issues holding back performance, CPython does.

The insistence on GvR's part of not scrapping CPython in favor of a JIT compiler as the reference implementation is what is causing it to slowly lose ground in certain areas of development such as game development (Lua) and webservices (Go). JIT as default implementation wouldn't harm any of Python's strongholds (I should mention that C extensions are beginning to be better supported in PyPy), while it would stop losing ground in areas where competition is strong.

2 ▲ ▼ • Reply • Share ›

jakevdp Mod ➚ BuckRogers • 2 years ago

I'm very glad GvR has stuck with CPython. If he hadn't, it would absolutely blow apart the community in a way that would make the 2-to-3 transition seem downright easy.

When a scientist says "Python", they mean not only the language, but also the huge ecosystem of compiled extensions provided by numpy, scipy, and related packages. In this sense, PyPy, Jython, IronPython, Pyston, etc. *are not Python*. Only CPython is Python.

So if GvR et al had decided to abandon CPython in favor of a new JITed interpreter like PyPy, it would have been a complete abandonment of the Scientific community, and our only choice would have been to fork CPython and keep going with what works for us.

Now, if you could develop a JIT compiler for Python which works with CPython's C-API backend, you might get somewhere. My impression is that this is what Unladen Swallow was supposed to do, but for reasons I don't entirely understand, the project was abandoned.

So could this all be solved by full support for C Extensions in PyPy? Sure - but I'll believe it when I see it.

4 ▲ ▼ • Reply • Share ›

BuckRogers ➚ jakevdp • 2 years ago

You missed, or purposefully omitted the part where I said PyPy already has support for

You missed, or purposefully omitted the part where I said PyPy already has support for C extensions. It's a work in progress and needs more support, but it is there already. Plenty of recent updates from the PyPy team on this. http://morepypy.blogspot.com/

Also, Pypy, Jython and the rest are Python. Python is a language specification, not an implementation. CPython is merely one of many of those.

^ | v • Reply • Share ›

jakevdp Mod ➚ BuckRogers • 2 years ago

I disagree. When a scientist says "Python", they are not talking about a language specification, they are talking about the language plus the huge ecosystem of compiled extensions provided by numpy, scipy, etc. **in that sense** (operative words here) PyPy et al. **are not Python**. And until C-API support in PyPy goes beyond "work in progress" (that is, until I can take any of the scripts I run daily in Python and run them in PyPy with no modification) I'm going to maintain that view.

In other communities, Python might be considered just a language specification, but that is most definitely not true when it comes to the scientific community. CPython **is** Python.

3 ^ | v • Reply • Share ›

Peter Wang ➚ BuckRogers • 2 years ago

Years ago, I blogged about this same point that Jake is trying to make in the parent comment: http://pwang.wordpress.com/201...

The massive increase in adoption of scientific and numerical Python in the Fortune 500 is completely and utterly dependent on CPython-the-runtime. The "hip web dev" community is just a tiny sliver of the total number of Python deployments out there. You don't find investment banks and three-letter-agencies blogging about how they use Python, but I can personally assure you that they deliberately choose to use CPython.

PyPy's C library integration is coming along and looks nice - and someday when it finally reaches feature and performance parity with the existing CPython-based universe of analytical libraries, *then* (and only then) can we even begin on the marketing job of convincing people who have existing, production systems to switch over to PyPy. You thought that it was hard to convince people with dollars on the line to transition from Py2 -> Py3... wait until you try to convince to re-compile or re-wrap all their stuff to use cffi.

As for Jython and IronPython: those are interesting, but make up an utterly insignificant number of actual production deployments, as far as I can tell.

1 ^ | v • Reply • Share ›

BuckRogers ➚ Peter Wang • 2 years ago

You guys will argue till you're blue in the face about things no one cares about, but as a scientist, you should be fully aware that different words are used to be precise when describing two very different things.

It's called "CPython" because that's the name of one (important)

It's called "CPython" because that's the name of one (important) implementation, and that's why it's not just called "Python". Hope this helps.

^ | v • Reply • Share ›

jakevdp Mod ➚ BuckRogers • 2 years ago

Believe me, Buck, I understand the distinction.

I think you should re-read Peter's comment, though. He makes a really good point, which you seem to have missed.

^ | v • Reply • Share ›

BuckRogers ➚ jakevdp • 2 years ago

No, you didn't understand the distinction as evidenced by your lament before, "I guess another way to put it is that yes, Python is a language specification".
It's not "another way to put it", it's the truth.. Python is a language specification. This is why we invented words to add to the English language, to define differences between things- in reality.

I don't disagree with anything he said in his argument regarding PyPy vs CPython that he setup, which I don't have any disagreements about nor do I care about it. Thanks for the laugh anyway though considering I wasn't even responding to you.

But since you did it's a good time to note that I'm glad I could help you out to learn about the difference between Python and CPython. Good luck in life!

1 ^ | v • Reply • Share ›

jakevdp Mod ➚ BuckRogers • 2 years ago

I don't think ad hominem attacks have any place here, Buck.

1 ^ | v • Reply • Share ›

jakevdp Mod ➚ BuckRogers • 2 years ago

I guess another way to put it is that yes, Python is a language specification, but that specification includes how to interact with it via C code.

^ | v • Reply • Share ›

Sean M • 2 years ago

Hmm the 113 == 4 example didn't work for me - I get "False" as the following programs output :S

```
import ctypes

class IntStruct(ctypes.Structure):

_fields_ = [("ob_refcnt", ctypes.c_long),

("ob_type", ctypes.c_void_p),

("ob_size", ctypes.c_ulong),

("ob_digit", ctypes.c_long)]

def __repr__(self):
```

def __repr__(self):

return ("IntStruct(ob_digit={self.ob_digit}, "

"refcount={self.ob_refcnt})").format(self=self)

id113 = id(113)

iptr = IntStruct.from_address(id113)

iptr.ob_digit = 4 # now Python's 113 contains a 4!

print 113 == 4

1 • Reply • Share ›

jakevdp Mod ➔ Sean M • 2 years ago

Are you running Python 3.4 on a 64-bit machine?

• Reply • Share ›

Russell Borogove • 2 years ago

Would you categorize C#, Java, Erlang, and other languages as "interpreted"? Like those, the standard CPython implementation compiles to bytecode running on a VM.

1 • Reply • Share ›

jakevdp Mod ➔ Russell Borogove • 2 years ago

Pythons .pyc files are bytecode, but they're not "compiled" in the sense that Java bytecode is. Though I guess you could quibble about that a bit... CPython is implemented in C, so of course Python's bytecode is compiled... but it's the C implementation that's compiled, not the Python script itself. That's closer to the point I was trying to make in this post.

In any case, to answer your question, no, Python is not like C#, Java, or Erlang, even though all of these have bytecode and a VM.

• Reply • Share ›

Russell Borogove ➔ jakevdp • 2 years ago

Nonsense, .pyc is every bit as "compiled" as Java bytecode is. Is lack of JIT compilation to native the distinction you're making?

• Reply • Share ›

jakevdp Mod ➔ Russell Borogove • 2 years ago

Well, pyc is "compiled" in the sense of being bytecode, but not in the sense of having any pre-execution information about types from which the bytecode interpreter can make optimizations. So yes, I think there is a big distinction there, and it's the one I've been trying to make throughout the post and these comments, though my language has admittedly been a bit sloppy.

I guess the point is that Java, etc. compile the operations to bytecode. Python compiles the logic to use in determining the operations to bytecode. My short-hand is to call the second one "not compiled", and I'm going to stick to that.

• Reply • Share ›

Russell Borogove ↗ jakevdp • 2 years ago

They're both compiled to bytecode; the bytecode that Python uses has dynamic typing semantics rather than static, which means it's slower, which is your article's point 1. You're using "interpreted" vs "compiled" to mean that, which is not how most people use those terms. The way you're using them, your article's point 2 is a sloppy way of reiterating point 1, and it reinforces the misconception that Python is interpreted.

Check out Psyco to see Python being JIT'd to native, by the way.

^ | v • Reply • Share ›

Chris Warburton ↗ jakevdp • 2 years ago

"Compiled" and "interpreted" aren't mutually exclusive; "compiled to machine code" and "interpreted" are.

For example, compiling Java to machine code (eg. with GCJ) means it's not interpreted. Compiling Java to bytecode for a JVM means it's interpreted.

Likewise, compiling Python to CPython bytecode means it's interpreted, whilst compiling it to machine code via Shedskin means its not interpreted (although Shedskin is a very poor implementation of Python's semantics).

^ | v • Reply • Share ›

Dzmitry Lazerka • 2 years ago

You also didn't mention Python's Global Interpreter Lock, which slow things down as well.

1 ^ | v • Reply • Share ›

opyate ↗ Dzmitry Lazerka • 2 years ago

Yes, I would have liked to see the GIL mentioned too.

^ | v • Reply • Share ›

Free2rhyme214 • 2 years ago

If Python is slow why is Dropbox so fast?

1 ^ | v • Reply • Share ›

Thomas ↗ Free2rhyme214 • 2 years ago

There's a couple of reasons for this:

- A 'slow' programming language is still fast enough to do things very quickly by user standards. Adding numbers together even in Python takes nanoseconds.
- How fast Dropbox is probably depends much more on moving data over the network than on running code. Which programming language you move doesn't greatly affect how fast you can push data around.

8 ^ | v • Reply • Share ›

Russell Borogove ↗ Thomas • 2 years ago

In fact, for any application where performance is dominated by disk or network I/O, there's little performance difference between Python and other languages. Pure computation is a relatively small part of what modern software does.

3 • Reply • Share ›

3thm4n ➔ Russell Borogove • 2 years ago

Thank you. And most web apps/services are bound by IO, not CPU. Which is why you should use whatever language works for you, rather than the "fastest" or "more efficient".

• Reply • Share ›

imom0 • 2 years ago

incisive

1 • Reply • Share ›

Chris • 3 months ago

Beautiful post, thanks for sharing your knowledge with us.

• Reply • Share ›

Wenjie Sha • 4 months ago

This post is awesome! I bump into this post , trying to debug debug memory leak issues in my pyhton + numpy + theano code. Any suggestion? I have been super careful not to add a bunch of objects into global var/list. so I suspect the culprit in numpy, theano manipulating ref_cnt

• Reply • Share ›

Grazfather x • 2 years ago

Fantastic.

Formatting-wise, I noticed you stopped marking Out[n] and it looks like you moved from an interpreter to a script (You started print()ing everything as opposed just typing out the object)

• Reply • Share ›

V8 • 2 years ago

As an old Lisp programmer, I'd say these are all symptoms of one problem: CPython doesn't have a (real) compiler, or a JIT. None of the problems you listed is a dealbreaker for performance.

Javascript (the language) suffers from every single one of your listed problems, and it has the same dynamic features, and yet there's at least 3 completely different JS runtimes today that are 10 times faster than CPython at these tasks.

• Reply • Share ›

Philippe Back • 2 years ago

Basic Pharo 3 test (http://pharo.org) gives the following range.
Full object language.

[(1 to: 100000001) sum] timeToRun 0:00:00:07.335

• Reply • Share ›

LeslieK • 2 years ago

Thank you for this post. Very informative.

I recently posted a way to examine the internals of a python 3.3 dictionary on [StackOverflow]

(http://stackoverflow.com/quest... using the ctypes module.

The trickiest part was how to access the array holding the key-value pairs, since this array is size of 1 when created but grows as keys are added to the dictionary.
Your solution for a fixed-size list is in shown above in In[13].

Also, I mapped the type Py_ssize_t to the ctypes type c_ssize_t so portability isn't an issue. Is there a reason you did not use c_ssize_t? Can't you write this so you don't have to assume 64-bit machine?

jakevdp Mod → LeslieK • 2 years ago

I think with some effort you could probably make this stuff more platform-independent, but that was not my primary goal here.

Guy • 2 years ago

Your atom feed seems broken, can you fix it? Thanks! :)

jakevdp Mod → Guy • 2 years ago

Been broken forever. I tried to fix it last year, but it took too much time and I got nowhere. If it's important to you, I'd happily accept a PR: https://github.com/jakevdp/Pyt...

Piyush Kansal • 2 years ago

Simple and easy to understand post. Thanks !

uchi • 2 years ago

Aren't examples 1 and 3 really just (symptoms of) "Python doesn't use tagged types for integers"?

jakevdp Mod → uchi • 2 years ago

The world is full of many wonderful creatures besides integers; integers are simply what I happened to choose for these examples. Even if Python used tagged integers, 1 and 3 would hold in general.

Chris Warburton → jakevdp • 2 years ago

From a Python point of view, that sounds reasonable (since Python is dynamically typed after all). However, you're comparing Python to C, so it *absolutely* makes a difference, because C cares so much about integers!

For example, we might use MyClass instead of integers, or even a whole bunch of heterogenous classes, and call their __contains__ methods instead of their __add__ method (ie. "+"). Will NumPy help us now? Not really. Will CPython's pointer indirection and dynamic dispatch be more overhead than doing it in C? Not by much, since we'd need to implement those manually in our C in order to support heterogeneous types.

Tagging pointers can bring Python closer to C by implementing C-like workloads (eg. arithmetic on ints and floats) in a C-like way. It won't (dramatically) speed up anything non-C-like, but then again neither will C.

• Reply • Share ›

Alejo • 2 years ago

awesome

• Reply • Share ›

whycantibeanon • 2 years ago

PHP and Perl are also dynamically typed, as well as interpreted just like Python, yet they are both a few billion times faster than Python. I'm willing to write Python's abysmal performance down to an absolute piss-poor codebase and interpreter.

• Reply • Share ›

ZygmuntZ ➚ whycantibeanon • 2 years ago

Nope. All these languages are very similiar in terms of speed, roughly 30x slower than C. For example:

http://benchmarksgame.alioth.d...

20 • Reply • Share ›

Load more comments

ALSO ON **PYTHONIC PERAMBULATIONS**

Analyzing Pronto CycleShare Data with Python and Pandas

1 comment • 8 months ago•

opyate — > On the weekend, however, annual members appear less influenced by weather, while short-term users appear more influenced. Casual

Is Seattle Really Seeing an Uptick In Cycling?

18 comments • 2 years ago•

Jim Boyle — I am really intrigued by your casual comment to use random forests for further analysis. I have used random forests in a model

Recent Posts

- Analyzing Pronto CycleShare Data with Python and Pandas (https://jakevdp.github.io/blog/2015/10/17/analyzing-pronto-cycleshare-data-with-python-and-pandas/)
- Out-of-Core Dataframes in Python: Dask and OpenStreetMap (https://jakevdp.github.io/blog/2015/08/14/out-of-core-dataframes-in-python/)
- Frequentism and Bayesianism V: Model Selection (https://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/)
- Learning Seattle's Work Habits from Bicycle Counts (Updated!) (https://jakevdp.github.io/blog/2015/07/23/learning-seattles-work-habits-from-bicycle-counts/)
- The Model Complexity Myth (https://jakevdp.github.io/blog/2015/07/06/model-complexity-myth/)