Python 2020: Lots of Movement, Little Improvement

Paul R. Potts

August 2020

Yesterday I dove into improving some Python code for a work project, and I was reminded all over again why, although I like many things about it, Python is not my favorite programming language, with tooling that still, in 2020, often feels more like a hobby project rather than a piece of critical infrastructure. I spent an embarrassingly long time trying to determine the cause of a bug that resulted in my program generating data files that were incorrect, and inconsistent with the previous version of my program. I'll explain that below for nerds who might be interested.

Understanding the Bug

The bug happened when calling this relatively simple Python function. This function generates a Python byte array, which is a simple data structure that just grows as needed to hold a series of data bytes. Python has opaque data types that allow programs to be written with a high level of abstraction. That's great, but because I'm generating binary data files, sometimes I need to write code that operates on low-level data types with exactly the sizes and behaviors that I want. The NumPy (numeric Python) library provides some precise data types such as **uint32** that allow me to do what I need:

```
from numpy import uint32

def serialize_msbin_eeprom_record_header( record_data_size : uint32, record_data_checksum :

    msbin_record_header = bytearray()

    msbin_record_header.extend( uint32( 0x80000000 | AT25M01_HIGH_QUARTER_BASE_ADDRESS ) )
    msbin_record_header.extend( record_data_size )
    msbin_record_header.extend( record_data_checksum )

    return msbin_record_header
```

The bug was cropping up in the last line before the **return** statement, the one that operates on record_data_checksum. The byte array was being extended by eight bytes, not four. What?

I was calling this function from code that looked like this:

```
elements_v1_data_checksum_uint32 = uint32( sum( serialized_elements_bytesio.getvalue() ) )
elements_v1_extended_checksum_uint32 = elements_v1_data_checksum_uint32 + sum( serialized_v1

msbin_bytes.extend( serialize_msbin_eeprom_record_header( elements_v1_extended_size_uint32,
```

The first line starts with a **BytesIO** object, which is basically an in-memory binary file object; calling **getvalue()** on that object returns a byte array, and the **sum()** function, when passed a byte array, creates a byte-by-byte checksum.

The second line adds to this checksum to create an extended checksum. It sums another byte array and adds the value to the previous **uint32** object.

Then, I create a **uint32** object from this sum. This **uint32** object then gets passed to the **serialize_msbin_eeprom_record_header()** function I described above.

The function parameter has a *type annotation*: **record_data_checksum : uint32**. This tells the Python interpreter (or compiler) that the second parameter is, or at least ought to be, a **uint32** object. Type annotations exist in many other languages and have for decades. Python is a *dynamically typed* language, similar to Apple's Dylan language, but just recently has gained this new type annotation feature. Dylan had optional type annotations in the early 1990s: if you didn't specify types, the compiler would generate code without doing compile-time type-checking, instead doing run-time type checking. This allowed developers to use it more like a scripting language, writing code quickly without worrying about the exact types they were using. Then, the code could be tightened up later: the development environments had features that would indicate "hot spots" where the generated code was less efficient due to this run-time type checking, to help programmers add strict tying where it would be useful.

What was really happening in my code? The **uint32** object was being converted to a **uint64** object when I added another value to it, and then this **uint64** object was being sent to the **serialize_msbin_eeprom_record_header()** function instead of the **uint32** object it was expecting. And Python was fine with this, because the type annotations don't actually **do** anything when you run the code under "CPython," the standard and most widely-used Python interpreter. These annotations are designed to be used with separate type-checking programs that aren't part of the standard Python distribution for Windows.

Changing to a 64-bit Type

So why did this change to **uint64** happen? Let's try creating a simple piece of code that replicates the problem:

```
from numpy import uint32

num_1 = uint32( 1 )
num_2 = uint32( 2 )
num_3 = 3
num_4 = num_1 + num_2
num_5 = num_1 + num_3
print ( 'num_4 value: ' + str( num_4 ) + ", type: " + str( type( num_4 ) ) )
print ( 'num_5 value: ' + str( num_5 ) + ", type: " + str( type( num_5 ) ) )
```

This prints the following output:

```
>num_4 value: 3, type: <class 'numpy.uint32'>
>num_5 value: 4, type: <class 'numpy.int64'>
```

These small values can't be overflowing. Instead, what we are seeing is the effect of rules designed to *prevent* overflow. Different languages do this in quite different ways, and so it was not immediately obvious to me, as I typed the original line of code, what would happen.

In languages like C, unsigned integer types by definition are allowed to overflow or underflow. If you have a 32-bit unsigned integer that is holding the maximum possible 32-bit value, **0xFFFFFFFF** in hexadecimal or 4,294,967,295 in decimal, and you add one to it, the value will "roll over" and become zero. The C standard guarantees this behavior.

This rationale is complex, but the short version is that back when computers were large and slow, doing otherwise would have required runtime checks, in software or in hardware, to detect overflows or underflows. And so this behavior was enshrined in the C standard, and developers like me are accustomed to it — indeed, we rely on it all the time.

But what about the **uint32** type in Python's NumPy library?

Well, the NumPy documentation is pretty vague. It says:

> The behavior of NumPy and Python integer types differs significantly for integer overflows and may confuse users expecting NumPy integers to behave similar [*sic*] to Python's **int**.

I know that Python's **int** handles overflow gracefully and safely, with some cost in efficiency.

But it doesn't give much detail describing how NumPy types actually work.

There are a couple of possible ways that the library could handle possible overflow conditions when adding two **uint32** objects:

- The value could simply be allowed to overflow, as in C. Because NumPy is designed for speed, to provide a faster alternative to Python's general-purpose data structures, this is the behavior I would have expected. The comment in the NumPy documentation suggests that this might be the case because NumPy integers "may confuse users" and don't behave like **int**.

- Adding two **uint32** objects could always generate a result with the next-larger type, **uint64**, no matter how large the actual numbers in those objects are. That would be simple but incur some space overhead whether it is needed or not.
- The generated code could actually look at the values in the objects, and if necessary, either return the larger type, or generate an error.

I didn't really know how the NumPy library handles this overflow, but I didn't think I needed to know, since I didn't expect that my checksums of small data structures would ever be large. I was writing a *script*, which is very problem-specific, rather than a *library*, which should be able to handle all boundary conditions safely. But let's take a quick look at how it works:

```
from numpy import uint32

num_1 = uint32( 0xFFFFFFFF )
num_2 = uint32( 1 )
num_3 = num_1 + num_2
print ( 'num_3 value: ' + str( num_3 ) + ", type: " + str( type( num_3 ) ) )
```

This prints the following output:

```
>./test.py:4: RuntimeWarning: overflow encountered in ulong_scalars
>  num_3 = num_1 + num_2
>num_3 value: 0, type: <class 'numpy.uint32'>
```

Interesting: the overflow was detected, but the program generated a warning, not an exception, and so terminated normally.

Adding a Built-in Type to a NumPy Type

That's what happens when adding two **uint32** types. But that's not what my code was doing. My code was adding a Python **int** type to a NumPy **uint32** type. And that's where my expectations broke down.

```
from numpy import uint32

num_1 = uint32( 0xFFFFFFFF )
num_2 = 1
num_3 = num_1 + num_2
print ( 'num_3 value: ' + hex( num_3 ) + ", type: " + str( type( num_3 ) ) )
```

prints:

```
>num_3 value: 0x100000000, type: <class 'numpy.int64'>
```

The num_3 was promoted to a 64-bit type. And it did the promotion even though the value wouldn't overflow:

```
from numpy import uint32

num_1 = uint32( 1 )
num_2 = 1
num_3 = num_1 + num_2
print ( 'num_3 value: ' + hex( num_3 ) + ", type: " + str( type( num_3 ) ) )
```

prints:

```
>num_3 value: 0x2, type: <class 'numpy.int64'>
```

Replicating C's Behavior

In C, the behavior when using *mixed* types, unsigned and signed together, in expressions, is trickier. Integer types are "promoted" before operations are done.

It looks like the NumPy library attempts to replicate C's behavior in this case and avoid potential overflow conditions, and so the **int**, even though it contains only a small value, is promoted to the larger **uint64** type — which I didn't even import because I didn't plan to use it — before the addition is performed. I'd have to look at the source code for NumPy to figure out exactly what is happening, but the point is, I shouldn't have to.

So it seems like we have the worst of several worlds now; we know from our "RuntimeWarning" that the NumPy library can generate overflow warnings when adding two **uint32** objects, so there is code generated which does this range-checking. But when a **uint32** is added to a standard Python number, the return value is the larger **uint64**. Because Python is dynamically typed, from the interpreter's perspective, nothing has actually gone wrong. And then, even with type annotations, this type change isn't caught by CPython because type annotations aren't actually used for type-checking. I just get unexpected output.

By the way, I tried the 3rd-party **mypy** utility, which is billed as a "linter" for Python programs, and supposedly reads the type annotations, to see if it would have caught the problem. It didn't report an issue.

Python is now widely used in education, having displaced simpler and more tightly-specified languages such as Scheme. I'm trying to imagine a first-year or second-year computer science student trying to figure out this problem, when expressions can wind up with unexpected types, and the language and tools are not designed to help.

Python is also widely used in various industries; it's used very extensively at Google, and the language's designer was a Google employee for a time. With this kind of backing, one might think that Python might have had its foundations shored up and improved; it might have even re-tooled to take advantage of some advances in language implementation that came about in the 1980s and 1990s.

One of these big corporate backers might have invested money to make type annotations actually work to make the code safer and more efficient.

I've been programming and studying computer science long enough to have some understanding of language design, and I've lived to see the same mistakes repeated again and again, as the tools underlying our critical infrastructure emerge from informally-specified hobby projects, and language designers don't really ever seem to learn from their predecessors.

Can we hope for better, in the career and lifetime I've got left? I'm getting tired of wasting so much time debugging, and of the slow churn of language development that is really traveling without moving.

As always, this content is available for your use under a Creative Commons Attribution-NonCommercial 4.0 International License.