What I learned about Python – and about Guido's time machine – by reading the `python-ideas` *mailing list*

Who am I?

A Director of the Python Software Foundation (one of eleven). Chair of PSF's Trademarks and Outreach and Education Committees.

I used to be well known as author of the IBM developerWorks column *Charming Python* and Addison-Wesley book *Text Processing in Python*.

Nowadays, I work at a research lab, D. E. Shaw Research, who have built the world's fastest supercomputer for doing molecular dynamics.

Why this talk?

For a couple years, I've followed the mailing list `python-ideas`. I am not, however, a core committer and don't follow `python-dev`. The ideas list:

> *[Contains] discussion of speculative language ideas for Python for possible inclusion into the language. If an idea gains traction it can then be discussed and honed to the point of becoming a solid proposal to put to python-dev as appropriate.*

sum() mysteries part 1

Quick question: what do you expect this to do:

```
>>> list_of_lists = [[1,2,3]] * 5
>>> list_of_lists
[[1,2,3], [1,2,3], [1,2,3], [1,2,3], [1,2,3]]
>>> sum(list_of_lists)

???
```

sum() mysteries

part 1.1

Trick question: what do you expect this to do:

```
>>> list_of_lists = [[1,2,3]] * 5
>>> list_of_lists
[[1,2,3], [1,2,3], [1,2,3], [1,2,3], [1,2,3]]
>>> sum(list_of_lists)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'int' and 'list'
```

sum() mysteries

part 1.2

Tricks aside, this is what I should have typed:

```
>>> list_of_lists = [[1,2,3]] * 5
>>> list_of_lists
[[1,2,3], [1,2,3], [1,2,3], [1,2,3], [1,2,3]]
>>> sum(list_of_lists, [])
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Was that obvious to everyone here?

... Honestly, it was not that obvious to me when I first looked at it.

sum() mysteries part 2.0

What `sum()` does is essentially just the below (but written in C as a fast built-in)

```
def sum(seq, start=0):
    for item in seq:
        start = start + item
    return start
```

This makes the behavior intuitive, I think. Just generalize a familiar operation:

```
>>> [1,2,3] + [1,2,3]
[1, 2, 3, 1, 2, 3]
```

sum() mysteries

part 2.1

We can break this down a little bit further though. What the function *really* does is make a method call on the accumulator:

```
def sum(seq, start=0):
    for item in seq:
        start = start.__add__(item)
    return start
```

That plus symbol (+) in the previous slide was really just some syntax sugar for calling a magic method on our '`start`' object.

sum() mysteries

part 3

With our lesson in mind, what do you think this variation does?

```
>>> list_of_lists = [[1,2,3]] * int(1e6)
>>> biglist = sum(list_of_lists, [])
```

Those of you with laptops, feel free to try this on your own machines now.

sum() mysteries part 3.1

With our lesson in mind, what do you think this variation does?

```
>>> list_of_lists = [[1,2,3]] * int(1e6)
>>> biglist = sum(list_of_lists, [])
```

Those of you with laptops, feel free to try this on your own machines now.

However, since your last line will not finish before this conference is over, let me tell you what to expect.

sum() mysteries

part 4.0

Concatenating lists (or iterators, generally) of lists gets very slow using `sum()`:

```
% python3.4 -mtimeit 'sum([[1,2,3]]*2000,[])'
100 loops, best of 3: 13.2 msec per loop
% python3.4 -mtimeit 'sum([[1,2,3]]*10000,[])'
10 loops, best of 3: 371 msec per loop
% python3.4 -mtimeit 'sum([[1,2,3]]*50000,[])'
10 loops, best of 3: 9.91 sec per loop
```

Notice the pattern of times for 5× size scalings: 371ms/13.2 ≈ 28; 9,910ms/371 ≈ 27. This function is (a little worse than) $\Theta(N^2)$ on the size of the list.

sum() mysteries

part 4.1

One might be inclined to think at this point that concatenating a lot of lists is inherently complex. It really isn't though:

```
% python3.4 -mtimeit \
    -s 'from itertools import chain' \
    'list(chain(*[[1,2,3]]*int(1e6)))'
10 loops, best of 3: 101 msec per loop
```

There is exactly the same result that you've been waiting for from a few slides ago, delivered in a tenth of a second.

sum() mysteries

part 4.2

To be pedantic, we can get a little bit faster still:

```
% python3.4 -mtimeit \
    -s 'from itertools import chain, repeat' \
    list(chain.from_iterable(
           repeat([1,2,3], int(1e6))))'
10 loops, best of 3: 83.5 msec per loop
```

In a more general case than the quick test, you might have already have a million lists to concatenate in another iterable.

sum() mysteries

part 5.0

What went wrong was precisely the topic of a proposal made on `python-ideas` by a member named Sergey. He proposed that `sum()` should be implemented more like:

```
% cat sum.py
def sum(seq, start=0):
    for item in seq:
        # use .__iadd()__ if available
        start += item
    return start
```

sum() mysteries

part 5.1

This small change, written in pure-python – not even the C implementation – becomes blazingly fast (faster than `itertools.chain`):

```
% python3.4 -mtimeit \
  -s 'from sum import sum' \
  'sum([[1,2,3]]*50000,[])'
100 loops, best of 3: 3.59 msec per loop
% python3.4 -mtimeit \
  -s 'from sum import sum' \
  'sum([[1,2,3]]*int(1e6),[])'
10 loops, best of 3: 78.3 msec per loop
```

sum() mysteries

part 5.2

The pure-Python version *does* slow down by a 5× multiplier versus the built-in `sum()` on numeric lists, but a C version will be as fast as the current implementation.

```
% python3.4 -mtimeit 'sum([1,2,3]*int(1e6))'
10 loops, best of 3: 38.9 msec per loop
% python3.4 -mtimeit \
  -s 'from sum import sum' \
  'sum([1,2,3]*int(1e6))'
10 loops, best of 3: 191 msec per loop
```

sum() mysteries

part 6

The small change discussed in the last series of slides makes for a huge speed increase with a tiny amount of trivial code.

sum() mysteries

part 6.1

The small change discussed in the last series of slides makes for a huge speed increase with a tiny amount of trivial code.

… So why the heck do I – and the large majority of other participants on `python-ideas` – oppose this idea, even oppose it rather strongly?!

sum() mysteries

part 7.0

Reason #1: The use of sum() to concatenate sequences is not obvious.

Experienced Python programmers can easily understand that overloading the '+' operator causes `sum()` to work as it does, but beginners will not understand this.

It is better to encourage an obvious construct than to use one that works because of an implementation detail (i.e. Python *could have* used a different symbol for concatenation)

sum() mysteries

part 7.1

Reason #1: The use of sum() to concatenate sequences is not obvious.

As an experiment for the thread, I asked one non-programmer and one beginning programmer (well-educated adults; I wonder what children would intuit) what this should mean/do:

```
list_of_lists = [
   [4, 5, 2], [6, 12, 100], [100, 200, 300] ]
result = sum(list_of_lists)
```

sum() mysteries

part 7.2

Reason #1: The use of sum() to concatenate sequences is not obvious.

```
list_of_lists = [
   [4, 5, 2], [6, 12, 100], [100, 200, 300] ]
result = sum(list_of_lists)
```

One informant wanted an exception – not for the missing '`start`' but because it *"doesn't make sense."* The other – complete novice – wanted:

```
[11, 118, 600] # == map(sum, list_of_lists)
```

sum() mysteries

part 7.3

Reason #1: The use of sum() to concatenate sequences is not obvious.

```
list_of_lists = [
   [4, 5, 2], [6, 12, 100], [100, 200, 300] ]
result = sum(list_of_lists)
```

Another "obvious" answer suggested during discussion is implicit (recursive?) flattening:

```
729 # == sum([11,118,600])
    # == sum(find_numbers(list_of_lists))
```

sum() mysteries

part 8.0

Reason #2: "Fast" `sum()` is only sometimes fast.

```
% python3.4 -mtimeit 'sum([(1,2,3)]*50000,())'
10 loops, best of 3: 10.1 sec per loop
% python3.4 -mtimeit \
  -s 'from sum import sum' \
  'sum([(1,2,3)]*50000,())'
10 loops, best of 3: 10.2 sec per loop
% python3.4 -mtimeit \
  -s 'from itertools import chain' \
  'tuple(chain(*[(1,2,3)]*50000))'
100 loops, best of 3: 4.7 msec per loop
```

sum() mysteries part 8.1

Reason #2: "Fast" `sum()` is only sometimes fast.

Built-in `tuple` does not have a fast `.__iadd__()` method. Sequence types in `collections`, or third-party libraries, may well not have O(1) concatenation, nor be amenable to allowing it.

We *could* specialize on `tuple` in `sum()`; but, for example, a `cons` single-linked list is inevitably O(N) to append at end. A fast `concat_conses()`, cannot just be looping over `.__iadd__()` calls.

sum() mysteries

part 9.0

Reason #3: `__iadd__()` has different semantics!

Intuitively, you might assume that these two lines of Python must behave identically.

```
seq = seq + other_seq
seq += other_seq
```

However, on reflection you can see this isn't so; the two lines are actually syntax sugar for these:

```
seq = seq.__add__(other_seq)
seq = seq.__iadd__(other_seq)
```

sum() mysteries part 9.1

Reason #3: `__iadd__()` has different semantics!

Still, you might protest: Only a perverse third-party class would ever *actually* give different meanings to:

```
seq = seq + other_seq
seq += other_seq
```

Making those differ is an affront to common sense and magic too deep for end-users to think about!

sum() mysteries part 9.2

Reason #3: `__iadd__()` has different semantics!

Here's a library you might have heard of:

```
>>> from numpy import array
>>> a1 = array([1.0, 2.0, 3.0], dtype=int)
>>> a2 = array([0.1, 0.2, 3.3], dtype=float)
>>> a3 = a1 + a2
>>> a1 += a2
>>> a1, a3
(array([1, 2, 6]), array([1.1, 2.2, 6.3]))
```

Crazy huh? And yet a quite intuitive approach to type promotion for numeric types.

Addition is confusing! part 1.0

OK, so maybe `sum()` isn't a great way to spell concatenation. But at least it's a uniform way to add numbers quickly (and accurately). Right?

```
>>> sum([1e50, 1, -1e50] * 1000)
0.0
>>> sum([1e50, -1e50, 1] * 1000)
1.0
```

Oh dear! Floating point numbers sure do odd things with divergent exponents.

Addition is confusing! part 1.1

Are we doomed by rounding errors?

```
>>> sum([1e50, 1, -1e50] * 1000)
0.0
>>> from math import fsum
>>> fsum([1e50, -1e50, 1] * 1000)
1000.0
>>> help(fsum)
fsum(iterable)
  Return an accurate floating point sum of
    values in the iterable.
  Assumes IEEE-754 floating point arithmetic.
```

Addition is confusing! part 1.2

`fsum()` is a nice function tucked away in the `math` module – I should use that more often (right?)

```
>>> from decimal import Decimal as D
>>> dnums = D('1.1'), D('2.2'), D('3.3')
>>> math.fsum(dnums), sum(dnums)
(6.6, Decimal('6.6'))
>>> from fractions import Fraction as F
>>> fnums = F(1,2), F(3,4), F(5,6)
>>> math.fsum(fnums), sum(fnums)
(2.0833333333333335, Fraction(25, 12))
```

Addition is confusing!

part 1.3

`fsum()` is a nice function tucked away in the `math` module – I should use that more often (right?)

```
>>> math.fsum([1,2,3]), sum([1,2,3])
(6.0, 6)
>>> cnums = complex(1,1), complex(2,2)
>>> sum(cnums)
(3+3j)
>>> math.fsum(cnums)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

Addition is confusing! part 1.4

Should we use `math.fsum()`? Yes, certainly at times. But also no, not generally.

`fsum()` gives us the right answer if we want a floating point answer, but is imperialistic about insisting `float` is the über-type for all numbers (yet we can have very good reasons to want `Fraction` or `Decimal` instead).

Well, also `fsum()` might decide to raise an exception if it doesn't like our numeric types.

Addition is confusing! part 2

On `python-ideas`, Steven D'Aprano suggested creating a sum function that would both be numerically accurate and preserve the type of homogenous sequences (and only accept numbers).

For mixed numeric datatypes, some sort of type coercion is always going to be necessary.

The result of that discussion is documented in PEP 450, and is in a standard library module, as `statistics._sum()`, in Python 3.4.

Addition is confusing! part 3.0

I think we've finally found "*the* `sum()` *to rule them all!*" In Python 3.4, we can just use `statistics._sum()` to produce accurate and type-preserving sums.

Wasn't there something else though? Oh yeah, it would be nice if it were *fast* too!

… not for sequences – we realized that is an awkward corner – but at least for numbers.

Addition is confusing!

part 3.1

Let's look at some timings. In these, `nums` is a collection of 10,000 random `Fraction`'s

```
% I='from fractsum import binsum, statsum, nums'
% python3.4 -m timeit -s "$I" 'sum(nums)'
10 loops, best of 3: 2.29 sec per loop
% python3.4 -m timeit -s "$I" 'statsum(nums)'
10 loops, best of 3: 124 msec per loop
% python3.4 -m timeit -s "$I" 'binsum(nums)'
10 loops, best of 3: 21 msec per loop
```

Addition is confusing! part 3.2

The built-in is quite slow at adding fractions.

```
sum(nums)        2.29 sec per loop
statsum(nums)    124 msec per loop
binsum(nums)     21 msec per loop
```

Using a technique I proposed gets the 19× speedup in `statsum()`; using a complementary technique from Oscar Benjamin at the end ekes out that next 5× in `binsum()`.

We can do 100× better than `sum()` in pure-Python (a C version might be 5× that)!

Addition is confusing!

part 3.3

Let's look at the magic (and concise) algorithms.

Adding fractions is slow because of repeated GCD calculations. My intuition was that "binning" the denominators allows for plain integer additions.

```
def binsum(iterable):
    bins = defaultdict(int)
    for num in iterable:
        bins[num.denominator] += num.numerator
    # sum() here gets 20x, mergesum() the 100x
    return mergesum([F(n, d)
        for d, n in sorted(bins.items())])
```

Addition is confusing!

part 3.4

GCD calculations are much slower on large denominators. If we can perform most of them on comparatively small numbers, we win:

```
def mergesum(seq):
    while len(seq) > 1:
        cut = len(seq)//2
        new = [a+b for a,b in zip(
                                seq[:cut],seq[cut:])]
        if len(seq) % 2:
            new.append(seq[-1])
        seq = new
    return seq[0]
```

Addition is confusing! part 3.4

It doesn't exist now, but I think the last few slides argue that we should have a `fractions.sum()` also (maybe even with a C implementation).

Moreover, summing `Decimal`'s slows down to a similar degree, for similar reasons, and a solution would be similar also. Perhaps `decimal.sum()` should join this roster too.

Wrap-up / Questions? ∑ ∑ ∑ ∑ ∑

Let a hundred Sigmas bloom!

If we have time, I'd love feedback on these *ideas* (or catch me in the hallways).