Ruby master - Feature #14701

If the object is not frozen, I want to be able to redefine the compound assignment operator.

04/19/2018 11:30 PM - naitoh (Jun NAITOH)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

If the object is not frozen, I want to be able to redefine the compound assignment operator (e.g. +=, -=, *=, /=, ..etc).

https://docs.ruby-lang.org/ja/latest/doc/spec=2foperator.html

- Redefinable operator (method)

```
    |   ^   &   <=>   ==   ===   =~   >    >=   <    <=    <<   >>
    +   -   *   /     %    **    ~    +@   -@   []   []=   ` ! != !~
```

- use case

```
> require 'numo/narray'
> a = Numo::Int32[5, 6]
=> Numo::Int32#shape=[2]
[5, 6]
> a.object_id
=> 70326927544920
> a += 1
=> Numo::Int32#shape=[2]
[6, 7]
> a.object_id
=> 70326927530540
> a.inplace + 1
=> Numo::Int32(view)#shape=[2]
[7, 8]
> a.object_id
=> 70326927530540
```

With Numo::NArray, using "inplace" instead of "+=" will update the same object so it will be faster.

I want to write "a += 1" instead of "a.inplace + 1".
However, Ruby can not redefine "+=".

History

#1 - 04/20/2018 08:27 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

Use append instead of += for arrays. Changing the behavior of += would have too much compatibility problems from side-effect.

Matz.

#2 - 04/25/2018 03:19 PM - sonots (Naotoshi Seo)

> Use append instead of += for arrays.

+= operation for NArray is totally different with append for ruby array.
What we want to do with += operation for NArray is to do element-wise addition like

```
narray([1, 2, 3]) += narray([1, 1, 1]) #=> narray([2, 3, 4])
```

Because the memory size of a NAarray can be large such as 1GB, we want to perform in-place operation without allocating a new NArray.

> Changing the behavior of += would have too much compatibility problems from side-effect.

It is not necessary to change behavior of ruby default.
We just want to redefine += operator for a specific type of objects such as NArray. So, compatibility problems occur for only classes which redefine += operator newly.

#3 - 04/25/2018 03:22 PM - sonots (Naotoshi Seo)

ANOTHER IDEA:
How about allowing to redefine +! instead of += although it looks not intuitive for me, but it would be a ruby way.

#4 - 04/25/2018 04:53 PM - Eregon (Benoit Daloze)

I agree with matz.

Python's inplace operators sound like a fundamental design flaw to me,
because a += b is no longer a = a + b but sometimes something completely different and there is no way to differentiate at the source level.
Most notably, it means other variables pointing to the same NArray would get modified in place by +=, which seems highly unexpected.

So I would advise #add or #add! for such a use-case.
Users should be aware of the danger of modifying an object inplace, by using syntax exposing that mutability.

#5 - 04/26/2018 01:17 AM - masa16 (Masahiro Tanaka)

Former NArray had add! method, but it results in a confused result.

```
require 'narray'

a = NArray.int(2,2).indgen!
a[1,true] += 10
p a
# => NArray.int(2,2):
#    [ [ 0, 11 ],
#      [ 2, 13 ] ]

a = NArray.int(2,2).indgen!
a[0,true].add!(10)
p a
# => NArray.int(2,2):
#    [ [ 0, 1 ],
#      [ 2, 3 ] ]
```

So I dismissed add! method of the current numo-narray.

#6 - 04/26/2018 03:58 AM - nobu (Nobuyoshi Nakada)

What about

```
na = Numo::Int32[5, 6]
a = na.inplace
a += 1
```

#7 - 04/26/2018 04:24 AM - sonots (Naotoshi Seo)

> Most notably, it means other variables pointing to the same NArray would get modified in place by +=, which seems highly unexpected.

It seems expected and natural for me. It behaves similarly with other destructive methods of ruby:

```
irb(main):001:0> a = "bbb"
=> "bbb"
irb(main):002:0> b = a
=> "bbb"
irb(main):003:0> b << "c"
=> "bbbc"
irb(main):004:0> a
=> "bbbc"
```

#8 - 04/26/2018 06:03 AM - mame (Yusuke Endoh)

I agree with matz and eregon.

Honestly, I first thought that it was a good idea to split + and +=. But by investigating Python, I now think that it easily leads to strange behavior. One example is what eregon and sonots said:

```
>>> a = []
>>> b = a
>>> a += [1,2,3]
>>> b
[1, 2, 3]
```

Most Rubyists know and expect that Array#+ is non-destructive, so I think that it is unacceptable to allow this.

Another example is the following that is Python's actual result. I don't want to see this in Ruby.

```
>>> a = []
>>> a += "foo"
>>> a
['f', 'o', 'o']

>>> a = []
>>> a + "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

If I explain this behavior in Ruby terminology, Array#+= accepts any Enumerable, but Array#+ does not. Note that this spec design itself is somewhat reasonable. Array#+ accepts only an Array because, is it accepts an Enumerator, it is ambiguous what it should return, Array or Enumerator. On the other hand, this does not matter for Array#+= because an assignment returns no value in Python.

So, I think that these Python's behavior is indeed intentional. But actually strange. IMO, the fundamental flaw is to split + and +=.

sonots (Naotoshi Seo) wrote:

> ANOTHER IDEA:
> How about allowing to redefine +! instead of += although it looks not intuitive for me, but it would be a ruby way.

Do you mean introducing another set of destructive operators, like a +! 1 instead of a += 1? It is not very cool, but it seems more reasonable to me.

#9 - 04/26/2018 07:21 AM - sonots (Naotoshi Seo)

As first thing to say, we do not expect changing behavior of += for ruby built-in types such as string or array.

We want to have an extension point to redefine += for our library such as NArray.
How it should behave is the design choice and responsibility of such libraries.

> Most Rubyists know and expect that Array#+ is non-destructive, so I think that it is unacceptable to allow this.

I mean that it is even possible to redefine + operator to be destructive. It is design choice and responsibility of libraries.

> Do you mean introducing another set of destructive operators, like a +! 1 instead of a += 1? It is not very cool, but it seems more reasonable to me.

Yes, as a compromise. I still prefer +=, though.

> IMO, the fundamental flaw is to split + and +=.

IMO, it is design responsibility of libraries. But, I understand what you says partially. I will think of this.

#10 - 04/26/2018 08:56 AM - nobu (Nobuyoshi Nakada)

masa16 (Masahiro Tanaka) wrote:

> Former NArray had add! method, but it results in a confused result.

It seems a different story from +=.

```
require 'narray'

a = NArray.int(2,2).indgen!
a[1,true] += 10
p a
```

```
# => NArray.int(2,2):
#    [ [ 0, 11 ],
#      [ 2, 13 ] ]

a = NArray.int(2,2).indgen!
a[0,true].add!(10)
p a
# => NArray.int(2,2):
#    [ [ 0, 1 ],
#      [ 2, 3 ] ]
```

It feels that a[0,true] should return inplace object.
Otherwise += would not be possible to modify a itself too, even if it were differentiated.

#11 - 04/26/2018 09:28 AM - masa16 (Masahiro Tanaka)

nobu (Nobuyoshi Nakada) wrote:

> It seems a different story from +=.

I am sorry for getting off the point, but I wanted to say that add! method does not solve the problem.

> It feels that a[0,true] should return inplace object.

This cannot be a solution because both inplace and non-inplace (normal) operations are possible on a[0,true].

#12 - 04/26/2018 09:30 AM - masa16 (Masahiro Tanaka)

I think it is enough to change the meaning of

```
x += a
```

to

```
if x.respond_to?(:'+=')
  x.send(:'+=',a)
else
  x = x + a
end
```

Matz rejected this in http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/9127,
but I do not completely understand the reason.

#13 - 04/27/2018 10:19 PM - masa16 (Masahiro Tanaka)

Alternative idea: If Ruby has a feature like:

```
x = x.left_variable? # => true
y = x.left_variable? # => false
```

then inplace operation is possible without redefinition of +=.

#14 - 04/27/2018 11:58 PM - naitoh (Jun NAITOH)

matz (Yukihiro Matsumoto) wrote:

> Use append instead of += for arrays. Changing the behavior of += would have too much compatibility problems from side-effect.
>
> Matz.

Thank you for your reply.
I am sorry if it caused me to misunderstand what I meant to say by writing "the object is not frozen".

> Matz rejected this in http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/9127,
> but I do not completely understand the reason.

I wanted to continue with the discussion in 9127 above.

I do not want to change (redefine) the behavior of +=, -=, *=, /=, %= against Array.
I only want to define the behavior of +=, -=, *=, /=, %= (etc..) for Numo::NArray class.

Why does the compatibility problem occur as a side effect of enabling this definition for Numo::NArray class?
Is that a matter of performance?

masa16 (Masahiro Tanaka) wrote:

> I think it is enough to change the meaning of
>
> ```
> x += a
> ```
>
> to
>
> ```
> if x.respond_to?(:'+=')
>   x.send(:'+=',a)
> else
>   x = x + a
> end
> ```
>
> Matz rejected this in http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/9127,
> but I do not completely understand the reason.

That is exactly what I want.
Although there are concerns about performance problems, I could not understand that compatibility problems arise.

Eregon (Benoit Daloze) wrote:

> because a += b is no longer a = a + b but sometimes something completely different and there is no way to differentiate at the source level.

I think it can easily be distinguished.

> Most notably, it means other variables pointing to the same NArray would get modified in place by +=, which seems highly unexpected.

I think that only people using Numo::NArray should pay attention.

mame (Yusuke Endoh) wrote:

```
>>> a = []
>>> b = a
>>> a += [1,2,3]
>>> b
[1, 2, 3]
```

> Most Rubyists know and expect that Array#+ is non-destructive, so I think that it is unacceptable to allow this.

Yes, I agree with this point.
However, I would like to discuss Numo::NArray#+=, not Array#+(or Array#+=).
If it is a different class, is it strange to behave differently for that class?.

#15 - 04/28/2018 08:39 AM - naitoh (Jun NAITOH)

nobu (Nobuyoshi Nakada) wrote:

> What about
>
> ```
> na = Numo::Int32[5, 6]
> a = na.inplace
> a += 1
> ```

When saving in a variable, I do not want to manage the inplace state.

```
> require 'numo/narray'
> a = Numo::Int32[5, 6]
 => Numo::Int32#shape=[2]
[5, 6]
> a.inplace?
 => false
> a.inplace + 1
 => Numo::Int32(view)#shape=[2]
[6, 7]
> a.inplace?
 => false
```

In above, I want to set it to non-inplace state when saved in a variable.

```
> require 'numo/narray'
> na = Numo::Int32[5, 6]
 => Numo::Int32#shape=[2]
[5, 6]
> na.inplace?
 => false
> a = na.inplace
 => Numo::Int32(view)#shape=[2]
[5, 6]
> a.inplace?
 => true
> a.object_id
 => 6495820
> a += 1
 => Numo::Int32(view)#shape=[2]
[6, 7]
> a.object_id
 => 6495820
> a.inplace?
 => true
> a.out_of_place!
 => Numo::Int32(view)#shape=[2]
[6, 7]
> a.inplace?
 => false
```

I was able to do the operation to become the same object.
However, I do not want to set it to non-inplace state each time like this way.

If you let the variable save the inplace state, the following happens.

- not set inplace state case. (Expected result)

```
> a = Numo::Int32[5, 6]
 => Numo::Int32#shape=[2]
[5, 6]
> a += 100 - (a * 2)
 => Numo::Int32#shape=[2]
[95, 94]
```

- set inplace state case.

```
> na = Numo::Int32[5, 6]
 => Numo::Int32#shape=[2]
[5, 6]
> a = na.inplace
 => Numo::Int32(view)#shape=[2]
[5, 6]
> a += 100 - (a * 2)
 => Numo::Int32(view)#shape=[2]
[180, 176]
```

This is because variables in the inplace state are rewritten in the middle of calculation as follows.

```
> na = Numo::Int32[5, 6]
 => Numo::Int32#shape=[2]
[5, 6]
> a = na.inplace
 => Numo::Int32(view)#shape=[2]
[5, 6]
> (a * 2)
 => Numo::Int32(view)#shape=[2]
[10, 12]
> a
 => Numo::Int32(view)#shape=[2]
[10, 12]
```

#16 - 04/29/2018 05:50 AM - gotoken (Kentaro Goto)

Maybe I don't get the point though, if Numo::NArray#inplace and inplace.+ always return identical object, #inplace= can be defined and one can write

```
a = Numo::Int32[5, 6]
a.inplace += 1
```

means shorthand for a.inplace = a.inplace + 1. This setter #inplace= has no side-effect because a.inplace + 1 returns identical to a.inplace of LHS by assumption.

It also gives same side-effects to a for these:

```
a.inplace + 1
```

```
inp = a.inplace
inp += 1
```

By the way, to define similar things with index #[]=, #[] should be returns a view.

#17 - 04/15/2019 06:54 AM - sonots (Naotoshi Seo)

- Status changed from Rejected to Open

#18 - 04/15/2019 06:57 AM - sonots (Naotoshi Seo)

I've talked with matz today.
matz said he is not objective as long as we can define good semantics.

One option is allowing to define += operator, and if it is defined, call it. One concern is that it could cause slowness.
Another option is to define another operator such as +!.

#19 - 04/15/2019 07:27 AM - shyouhei (Shyouhei Urabe)

So far in ruby, a variable is not an object. That is the reason behind why we lack ++ operator.

naitoh (Jun NAITOH) wrote:

> I want to write "a += 1" instead of "a.inplace + 1".

Is this enough for us to depart from where we are now? The proposed change sounds something fundamental to me.

#20 - 04/16/2019 11:19 PM - Eregon (Benoit Daloze)

I agree with @shyouhei.
I think it breaks fundamental semantics of Ruby.
An explicit add or add! is I believe a good sign of the danger of mutating numeric values in place,
and with that we can still guarantee += is safe and reasonable in terms of semantics.

#21 - 08/11/2019 08:46 PM - jwmittag (Jörg W Mittag)

It is possible to find sane semantics for this. Scala, for example, has the following semantics:

```
a ω= b
```

is first tried to be interpreted as

```
a.ω=(b)
```

and if that fails, it is interpreted as

```
a = a.ω(b)
```

[Note: ω here is any sequence of operator characters other than >, <, !, and not starting with =.]

It would be possible to apply the same semantics to Ruby. This would allow to keep the current strict separation between *variables* and *values* as well as *variables* and *messages* in place.

There are other languages, where assignment is always a message send, but this requires much more fundamental changes to how Ruby works. In particular, it would require either giving up the distinction between *variables* and *values* or the distinction between *variables* and *messages*.

For example, in Newspeak, there are no variables. When you define what *looks like* an instance variable, what you are *actually* defining is an unnamed, unaccessible, hidden *slot* and a pair of getter (and setter, but only for mutable slots) methods for that slot. [See Section 6.3.2 *Slots* of the Newspeak Draft Spec.] Imagine Ruby's attr_accessor / attr_reader / attr_writer but without the ability to refer to instance variables directly using @. All other kinds of "variables" are then reduced to "instance variables", i.e. to getter / setter methods. Class variables are simply instance variables on the class object, local variables are instance variables on the stack frame object, and global variables simply don't exist.

In Ioke, assignment is what the author calls a "trinary operator", i.e. an operator that *looks* binary (has two operands) but is actually ternary (has a hidden third operand). So,

```
a = b
```

is actually equivalent to

```
=(a, b)
```

i.e. a message send of the message = with arguments a and b to the current receiver (which is however subtly different from Ruby's self), called the current "Ground" in Ioke. This however only works because a) the current Ground is a subtly different concept from self in Ruby (objects are not only dynamic scopes but also lexical scopes), b) because in Ioke, method arguments are passed un-evaluated (think "AST", but also subtly different), and c) "places" (storage space) are an actual reified language concept.

Both of these solutions would be a *major* departure from Ruby as it exists currently, and while both of these approaches are very powerful and beautifully simple, it is just not feasible to do anything like this in Ruby.

So, the only possible solution is the Scala solution: try a.ω=(b) first and if that fails, do a = a ω b.

Note, however, that there is no way around the fact that whatever we choose to do, this *will* be backwards-incompatible! There is no way of introducing this feature that will *not* change the behavior of the following code:

```
class Foo
  def +(*) 23 end
  def method_missing(*) 42 end
end

a = Foo.new
a += nil
a
```

Currently, this will call Foo#+ and re-assign a, if this change is implemented, it will call Foo#method_missing and *not* re-assign a.

Before:

```
a += nil
#=> 23

a #=> 23
```

After:

```
a += nil
#=> 42

a #=> #<Foo:0xdeadbeef>
```