Ruby master - Feature #9123

Make Numeric#nonzero? behavior consistent with Numeric#zero?

11/19/2013 03:12 AM - sferik (Erik Michaels-Ober)

Status:	Open
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	

Description

Numeric#zero? returns true or false, while Numeric#nonzero? returns self or nil.

I've written a patch that fixes this inconsistency and adds a Numeric#nonzero (non-predicate) method that returns self or nil for chaining comparisons. I'd like for this to be included in Ruby 2.1.0.

https://github.com/ruby/ruby/pull/452.patch

History

#1 - 11/19/2013 03:23 AM - fxn (Xavier Noria)

Both predicates return a boolean value, whose exact nature is irrelevant. I see no inconsistency to fix.

In my view Ruby programmers have to internalize that all objects have a boolean interpretation.

#2 - 11/19/2013 03:31 AM - sferik (Erik Michaels-Ober)

How would you feel if Numeric#zero? returned self (0) or nil?

Your description of the semantics of the question mark allows for this but I think it would be confusing. Likewise, Numeric#zero? returning truthy or falsey values as opposed to strict true or false is confusing.

Every method in Ruby returns a value that is either truthy or falsey so, by that logic, every method should end in a question mark.

#3 - 11/19/2013 03:39 AM - fxn (Xavier Noria)

I wouldn't care. I use predicates as predicates

```
do_foo if x.zero?
```

also

```
do_bar if str =~ /.../
```

the =~ operator does not return singletons, but the return value of a predicate generally speaking is irrelevant. Only its semantics as a boolean value matter to me as a user of the predicate.

In my view, the singletons true and false are objects the programmer that writes the predicate has at his disposal in case nothing else does the job. In C you may return 0 or 1, in Ruby you have the singletons if nothing else at hand captures the semantics of the predicate.

#4 - 11/19/2013 04:35 AM - BertramScharpf (Bertram Scharpf)

=begin

You did not only change the source code, but you also
removed an application example I admire for its beauty:

```
a = %w(z Bb bB bb BB a aA Aa AA A)
b = a.sort {|a,b| (a.downcase <=> b.downcase).nonzero? || a <=> b }
b   #=> ["A", "a", "AA", "Aa", "aA", "BB", "Bb", "bB", "bb", "z"]
```

I wonder how you manage not to notice anything about the impertinence of
your proposal.

=end

#5 - 11/19/2013 04:38 AM - BertramScharpf (Bertram Scharpf)

I know I'm boring, but it is still my opinion that there should be a String#notempty? corresponding to Numeric#nonzero?.

#6 - 11/19/2013 10:32 AM - sferik (Erik Michaels-Ober)

I did not remove that example, I just moved it under the documentation for Numeric#nonzero.

#7 - 11/19/2013 05:23 PM - Anonymous

Whereas the current implementation "works" as a predicate, I see no harm in gradually polishing Ruby towards a better design. If one was to design that from scratch, I bet there'd be no discussion -- predicates would return either true or false.

I would love to see more of these kind of patches getting accepted.

--
Txus
(I'll be slowly moving off this address -- please contact me at me@txus.io if possible from now on.)

On Tue, Nov 19, 2013 at 2:32 AM, sferik (Erik Michaels-Ober) <sferik@gmail.com> wrote:

> Issue #9123 has been updated by sferik (Erik Michaels-Ober).
>
> I did not remove that example, I just moved it under the documentation for
>
> ## Numeric#nonzero.
>
> Feature #9123: Make Numeric#nonzero? behavior consistent with Numeric#zero?
> https://bugs.ruby-lang.org/issues/9123#change-43012
>
> Author: sferik (Erik Michaels-Ober)
> Status: Open
> Priority: Normal
> Assignee:
> Category:
> Target version:
>
> Numeric#zero? returns true or false, while Numeric#nonzero? returns self or nil.
>
> I've written a patch that fixes this inconsistency and adds a Numeric#nonzero (non-predicate) method that returns self or nil for chaining comparisons. I'd like for this to be included in Ruby 2.1.0.
>
> https://github.com/ruby/ruby/pull/452.patch
>
> --
> http://bugs.ruby-lang.org/

#8 - 11/19/2013 08:11 PM - alexeymuranov (Alexey Muranov)

sferik (Erik Michaels-Ober) wrote:

> How would you feel if Numeric#zero? returned self (0) or nil?
>
> Your description of the semantics of the question mark allows for this but I think it would be confusing. Likewise, Numeric#zero? returning truthy or falsey values as opposed to strict true or false is confusing.

I like the argument that question-mark methods should return true/false (or maybe in some cases true/false/nil).

I think i am +1 for separate Numeric#nonzero? and Numeric#nonzero for readability.

Edited

#9 - 11/19/2013 08:49 PM - fxn (Xavier Noria)

This ticket is not about changing the semantics of the Ruby language. It is a ticket about a particular predicate.

Changing the semantics of Ruby is an epic goal (and one that is unrealistic in my view, although legit to wish). But unless there is an agenda that says that is the future of Ruby, we have to judge this particular proposal according to the *existing* semantics of Ruby. With the current semantics of Ruby where the boolean space is flat, true is as true as 1, and there is not even a Boolean type (because the space is flat), in my opinion there is nothing to change in this predicate, it is perfectly fine.

Txus: look at all languages, old and modern. While some have the semantics you'd like (legit), like Java, the majority of them don't. I think that should tell you that language designers would not generally agree with what you take for granted is a "better design". It is a better design *in your opinion*, and if you designed a language you'd do it that way, perfect, but the data shows you'd likely loose your bet.

#10 - 11/21/2013 05:07 AM - sferik (Erik Michaels-Ober)

=begin
I'm not proposing a change to the semantics of Ruby; I'm proposing a fix to an inconsistency.

The vast majority of predicate methods in Ruby return (({true})) or (({false})). There are approximately 200 such methods in the core library. By my count, there are only 6 core methods that return a truthy object or (({nil})):

- (({Numeric#nonzero?}))
- (({Module#autoload?}))
- (({File::world_readable?}))
- (({File::world_writable?}))
- (({File::size?}))
- (({Encoding::compatible?}))

In my opinion, (({Numeric#nonzero?})) is the most egregious aberration because (({Numeric#zero?})) returns (({true})) or (({false})). We could argue about whether it makes sense to change the other irregular methods but I believe this step toward consistency is a step in the right direction.
=end

#11 - 11/21/2013 06:00 AM - BertramScharpf (Bertram Scharpf)

> I'm not proposing a change to the semantics of Ruby; I'm proposing a fix to an inconsistency.

The opposite of "zero?" is not "nonzero?" but "notzero?". If the method's name was "notzero?", one could call it an inconsistency. "nonzero?" is not a yes-no-question.

And yes, you are proposing an inconsistency, because a lot of useful programs would no longer work.

#12 - 11/21/2013 09:06 AM - sferik (Erik Michaels-Ober)

=begin

> The opposite of "zero?" is not "nonzero?" but "notzero?". If the method's name was "notzero?", one could call it an inconsistency. "nonzero?" is not a yes-no-question.

Nonzero means "not equal to zero". It means the exact same thing as "not zero". It most certainly is a yes-no question. How else would you answer the question (({5.nonzero?}))?

> And yes, you are proposing an inconsistency, because a lot of useful programs would no longer work.

I believe you're confusing the word "inconsistency" with "incompatibility". Is English not your native language or are you trolling?
=end

#13 - 11/21/2013 03:28 PM - alexeymuranov (Alexey Muranov)

BertramScharpf (Bertram Scharpf) wrote:

> The opposite of "zero?" is not "nonzero?" but "notzero?". If the method's name was "notzero?", one could call it an inconsistency. "nonzero?" is not a yes-no-question.

"Nonzero" is an adjective, opposite of "zero" used as an adjective. "Not zero" is the negation of "zero", so essentially the same thing as "nonzero", but not a single word.

#14 - 11/22/2013 10:59 PM - mame (Yusuke Endoh)

I don't think it is possible to change the spec.
Many programs in the wild actually use the behavior.

https://github.com/search?q=nonzero%3F+sort+extension%3Arb&type=Code

By the way, I investigated the early history of nonzero?.

In the beginning, Nagai proposed a "<>" operator that returns -1, nil, 1
in ruby-list:7286. The motivation example was:

```
def struct_cmp (a, b)
(a.mem_a <> a.mem_a) || (a.mem_b <=> a.mem_b)
end
```

Obviously he had in mind Perl's common idiom for lexicographic sort:

```
sort { a[0] <=> b[0] || a[1] <=> b[1] }
```

because it is 1998 :-)

Matz said he was negative to this proposal in [ruby-list:7286] because
it would make the language complex to add a new operator, and because
he thought that it would be better to change the boolean semantics,
that is, to handle 0 as false (!).

Nagai, in [ruby-dev:2026], disagreed with the semantic change because of
a philosophical reason (it looked weird to him to handle only one instance
of Integer as false), and reclaimed adding a new operator.

In [ruby-dev:2031], Funaba disagreed with a new operator because the
lexicographic sort is not a common operation, he thought.
He also disagreed with the semantic change because of a compatibility
issue (in spite of 1998).
And then, nonzero? appeared in the counter-proposal he made:

```
(a <=> b).nonzero? || (a2 <=> b2)
```

Matz liked and implemented the proposal in a day [ruby-dev:2046].

Because nonzero? was accepted, a "<>" operator became less significant.
Nagai, in [ruby-dev:2050], dismissed his proposal.

In the mail, he pointed the inconsistency that is now discussed in this
ticket. He asked why matz implemented not only nonzero? but also zero?.
Matz's answer was:

- symmetry of zero? and nonzero?
- lisp/scheme provides zero?

(According to this history, it makes no sense to fit nonzero? with zero?.)

Incidentally, the discussion continued, and Keiju proposed Array#<=> in
[ruby-dev:2101], that is,

```
[a1, a2] <=> [b1, b2]
```

After the further discussion, this was accepted too.

Funaba, who proposed nonzero? himself, preferred Array#<=> to nonzero?,
but also pointed out that Array#<=> is not a perfect substitute to
nonzero? because Array#<=> does not shortcut.

After that, nonzero? exists until now.

--
Yusuke Endoh mame@tsg.ne.jp

#15 - 11/23/2013 12:54 AM - Eregon (Benoit Daloze)

mame (Yusuke Endoh) wrote:

> By the way, I investigated the early history of nonzero?.

Very interesting, thank you a lot for reporting it!

Personally I do sometimes use #nonzero? semantics when I want an Integer, but another value if it happens to be 0.
I think it complements the fact any Integer is truthy in Ruby.

#16 - 11/23/2013 04:01 AM - avit (Andrew Vit)

The history on this is interesting. I would agree with the consistency idea of nonzero = (nil | 1) and nonzero? = (false | true) but I don't know if this

could be changed now.

Also, why should we return nil in any case? I think the correct return value should be false, not nil. (nil implies an unknown or unavailable answer, but we *do* know if the number is 0 or not.) In other words:

```
42.nonzero? #=> 42
0.nonzero?  #=> false
```

#17 - 11/23/2013 09:14 AM - alexeymuranov (Alexey Muranov)

avit (Andrew Vit) wrote:

> Also, why should we return nil in any case? I think the correct return value should be false, not nil. (nil implies an unknown or unavailable answer, but we *do* know if the number is 0 or not.) In other words:
>
> ```
> 42.nonzero? #=> 42
> 0.nonzero?  #=> false
> ```

IMO, if 0.nonzero? is false, then 42.nonzero? should be true.

Slightly off topic, but how about something general like

```
class Object
  def non(*args)
    self unless args.include? self
  end
end
```

#18 - 11/23/2013 10:14 AM - mame (Yusuke Endoh)

avit (Andrew Vit) wrote:

> ```
> 42.nonzero? #=> 42
> 0.nonzero?  #=> false
> ```

In fact, the first version of nonzero? returned self or false.

Inaba, in ruby-dev:6417, suggested a convention about the usage
of nil and false.

- if a method returns only a boolean, it should use true and false
- if a method returns a general object but also "false" value, it should use nil

Matz accepted it, and adopted the convention to another exception, defined?.
As far as I know, this is the only spec change of nonzero? ever.

Inaba pointed out that a method whose name ends with "?" would make a user feel
that it should return only a boolean, though he also admitted that it is difficult
to change because of compatibility issue (in 1999).
Matz agreed, and said that "?" is a symbol that is "mainly used as a predicate".

--
Yusuke Endoh mame@tsg.ne.jp

#19 - 11/23/2013 11:23 AM - fuadksd (Fuad Saud)

I don't understand why we would want to be so strict about types on this
case. Having some arbitrary value being returned is useful and it doesn't
hurt any good practices. Ensuring predicates return true or false feels
useless for me

On Friday, November 22, 2013, mame (Yusuke Endoh) wrote:

> Issue #9123 has been updated by mame (Yusuke Endoh).
>
> avit (Andrew Vit) wrote:
>
> > ```
> > 42.nonzero? #=> 42
> > 0.nonzero?  #=> false
> > ```
>
> In fact, the first version of nonzero? returned self or false.

> Inaba, in ruby-dev:6417, suggested a convention about the
> usage
> of nil and false.
>
> - if a method returns only a boolean, it should use true and false
> - if a method returns a general object but also "false" value, it should use nil
>
> Matz accepted it, and adopted the convention to another exception,
> defined?.
> As far as I know, this is the only spec change of nonzero? ever.
>
> Inaba pointed out that a method whose name ends with "?" would make a user
> feel
> that it should return only a boolean, though he also admitted that it is
> difficult
> to change because of compatibility issue (in 1999).
> Matz agreed, and said that "?" is a symbol that is "mainly used as a
> predicate".
>
> --
>
> ## Yusuke Endoh >
>
> Feature #9123: Make Numeric#nonzero? behavior consistent with Numeric#zero?
> https://bugs.ruby-lang.org/issues/9123#change-43094
>
> Author: sferik (Erik Michaels-Ober)
> Status: Open
> Priority: Normal
> Assignee:
> Category:
> Target version:
>
> Numeric#zero? returns true or false, while Numeric#nonzero? returns self
> or nil.
>
> I've written a patch that fixes this inconsistency and adds a
> Numeric#nonzero (non-predicate) method that returns self or nil for
> chaining comparisons. I'd like for this to be included in Ruby 2.1.0.
>
> https://github.com/ruby/ruby/pull/452.patch
>
> --
> http://bugs.ruby-lang.org/

--
Fuad Saud

twitter http://twitter.com/fuadsaud |
linkedinhttp://www.linkedin.com/in/fuadksd|
coderwall http://coderwal.com/fuadsaud | githubhttp://github.com/fuadsaud|

#20 - 11/23/2013 07:03 PM - alexeymuranov (Alexey Muranov)

fuadksd (Fuad Saud) wrote:

> I don't understand why we would want to be so strict about types on this
> case. Having some arbitrary value being returned is useful and it doesn't
> hurt any good practices. Ensuring predicates return true or false feels
> useless for me

In my opinion, it is because code is intended mostly for reading, not for writing.

```
a.sort {|a,b| (a.downcase <=> b.downcase).nonzero? || a <=> b }
```

is unexpected or confusing.

#21 - 11/24/2013 01:31 AM - BertramScharpf (Bertram Scharpf)

alexeymuranov (Alexey Muranov) wrote:

> In my opinion, it is because code is intended mostly for reading, not for writing.

> a.sort {|a,b| (a.downcase <=> b.downcase).nonzero? || a <=> b }
>
> is unexpected or confusing.

That's a matter of taste. In my eyes, this code example is beautiful and clear.
It could be more efficient using #casecmp instead of two #downcases, what you clearly can see if it is one line.

#22 - 11/24/2013 01:34 AM - alexeymuranov (Alexey Muranov)

BertramScharpf (Bertram Scharpf) wrote:

> That's a matter of taste. In my eyes, this code example is beautiful and clear.
> It could be more efficient using #casecmp instead of two #downcases, what you clearly can see if it is one line.

I do not understand what is a matter of taste here. Everybody seems to agree that the only problem is the compatibility. How would the following be less beautiful or clear?

```
a.sort {|a,b| (a.downcase <=> b.downcase).nonzero || a <=> b }
```

or

```
a.sort {|a,b| (a.downcase <=> b.downcase).non(0) || a <=> b }
```

#23 - 11/28/2013 03:11 PM - fuadksd (Fuad Saud)

nonzero? returning the number is useful; I used this recently:

t '.items_count', count: items_count.nonzero? || t('.no')

It returns the number or, if it's zero, the proper translation for no (generating a "no items" message).

I feel like cases like this one are more common than we imagine.

#24 - 11/28/2013 09:05 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Now I finally understand the purpose of nonzero? :) In several languages, including Perl and JavaScript, 0 is a falsy value, but this is not the case for Ruby. So it's kind of a hack to make 0 be treated as a falsy value :) Indeed, now that I understand it, I could find quite some usage for it :)

#25 - 11/29/2013 03:07 AM - marcandre (Marc-Andre Lafortune)

- Category set to core

- Assignee set to matz (Yukihiro Matsumoto)

There is no use-case for this request and it would cause many incompatibilities (e.g. in rake, thor, ruby itself, ...):

https://github.com/jimweirich/rake/blob/bfafc3a0/lib/rake/application.rb#L300
https://github.com/ruby/ruby/blob/8612344834d4/lib/test/unit.rb#L398
https://github.com/ruby/ruby/blob/8612344834d4/lib/test/unit.rb#L398

The only "problem" is that some of you dislike the fact that a method ending in '?' returns something else than true or false. This is similar to String < Hash returning nil. Embrace Ruby, love it for its clever quirks. Or else I'd suggest writing your own new language instead of trying to "reinvent" Ruby.

I'll let Matz reject this request. In the meantime, I'm accepting wagers from anyone that thinks there's a chance that this will ever be accepted.

#26 - 11/29/2013 09:25 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Just for the record, I only said I can see how this behavior can be useful in cases you want to consider 0 (zero) as falsey. But I do actually prefer that boolean methods (those ending with a question mark) do always returned strict boolean values. I've even considered a while ago to create a new feature request to ask Ruby to always convert non-strict boolean values to either true or false, but I never actually created the issue because I was pretty sure it wouldn't be accepted.

The reason for that is that if Ruby main goal is to make programmers happy, and since I'm a programmer, I'd be much happier if I could know for sure if a value of a "method?" call is always true or false. Of course I'm not the only programmer out there ;) But what if you want to debug some code by using the print technique:

p "debugging something?", something?, other_values

I'd expect to read either true or false after "debugging something?", but instead I could get several lines as the output of "something?.to_s" if it returns an object instead of true.

Even if we consider only the boolean semanthic for Ruby and stop thinking about true and false, I still don't like the name of the method. Why would a method ending in a question mark, which is supposed to return either a truthy or falsey value, actually return a consistent value instead of only worrying about returning either truthy or falsey? I mean, such methods are meant to be used as "if obj.nonzero?" as opposed to "obj.nonzero? || anything" in the sense that the value returned by nonzero? is actually meaningful as a non-truthy value. I think I'm not able to make sense of my words, but what I mean is that I'd prefer that something like "nonzero?" was actually called "nonzero" (without the question mark) or even a better name than that.

"nonzero" only means that it's not zero. It doesn't give any hint it will return a meaningful number in case it's non zero. Reading code using nonzero? to return an expected number in case it's non zero would definitely make *me* pretty unhappy.

I don't actually expect this feature request to be accepted, but I'd just like to take the chance to state my opinions on the subject.

#27 - 11/29/2013 10:00 PM - eweb (Eamonn Webster)

Ever asked someone 'Do you know the time?' and they answer 'Yes'. So you ask 'What time is it?' and mutter 'Jerk!' under your breath.
Ruby is a friendly language, let's not turn it into a jerk.

#28 - 11/29/2013 10:22 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Humans don't have to ask if a number is not a zero :)

#29 - 11/29/2013 11:11 PM - alexeymuranov (Alexey Muranov)

eweb (Eamonn Webster) wrote:

> Ever asked someone 'Do you know the time?' and they answer 'Yes'. So you ask 'What time is it?' and mutter 'Jerk!' under your breath.
> Ruby is a friendly language, let's not turn it into a jerk.

Between #know_time? and #time_now!, i would prefer the second to get the actual time. :)

#30 - 11/30/2013 01:29 AM - alexeymuranov (Alexey Muranov)

Please correct me if i am wrong, but i think that the most useful part of this proposal is deprecating Numeric#nonzero? with its current behavior in favour of Numeric#nonzero.

#31 - 11/30/2013 02:20 AM - BertramScharpf (Bertram Scharpf)

alexeymuranov (Alexey Muranov) wrote:

> Please correct me if i am wrong, but i think that the most useful
> part of this proposal is [...].

There isno useful part in this proposal. Its only effect is an
endless discussion between people, who embrace Ruby and love it for
its clever quirks, and people who begrudge them their innocent joy.

#32 - 12/02/2013 04:59 PM - avit (Andrew Vit)

> i think that the most useful part of this proposal is deprecating Numeric#nonzero? with its current behavior in favour of Numeric#nonzero.

Personally I would agree with this too. I guess I'm not a fan of quirks if they're inconsistent...

- Asking "are you nonzero?" should be true/false (eventually).
- Saying "give me your nonzero value" should be the number, or nil.

Sorry, I know it's just a bikeshed argument...

#33 - 12/06/2013 10:37 PM - guigs (Guilherme Schneider)

I think current implementation of Numeric#nonzero? is fine, because, as was said, a predicate is expected to return truthy or falsey values.

On the other hand, Numeric#nonzero? should only be used as a predicate. So one should not rely on it returning self if is not zero. But, because so many do use nonzero? expecting that it returns self if it is not zero (and indeed the spec say so), changing it would break compatibility.

So, what about keeping current implementation of Numeric#nonzero? and adding Numeric#nonzero (with identical implementation), but intended to be used in non predicate cases, like

a.sort {|a,b| (a.downcase <=> b.downcase).nonzero || a <=> b }

And then maybe deprecate usage of Numeric#nonzero? in non predicate cases (and changing the spec saying that nonzero? returns true if the

number is not zero).

#34 - 12/07/2013 01:57 AM - fxn (Xavier Noria)

@Guilherme

A predicate in Ruby can return any object. If the exact return value is documented, then the user of the predicate can leverage that contract if so wishes. That's why it is documented.

If something is documented to return the singletons true/false, then you can rely on that and pass them to a JSON generator directly, if self is documented there is nothing wrong in exploting that fact. The =~ operator documents index or nil, fine, use the index if you want. Rails generally documents the equivalent of Object, that is, in most cases it says nothing about the exact value. There your only choice is to use it for its boolean semantics.

#35 - 04/15/2014 05:54 PM - headius (Charles Nutter)

Might as well toss in my opinion...

I don't care as a normal Ruby user whether a method returns truthy/falsey or true/false, because I shouldn't care about what the object is if all I need to know is its truthiness.

However, I don't like that the truthy value returned by some methods is self, because it's exposing more information than the method needs to expose.

This case is tricky, since the self object being returned is only a bit more information than the boolean value. The only real argument to change this is consistency. If you only care about the truthy value, you probably don't care about this issue. If you care about the self value, you don't want it to change. And it *is* documented to provide the current behavior.

I'd vote to change this, but I don't have a strong opinion. I will say this needs to be treated as a visible, potentially-breaking API change, which needs a major release. I'm not sure what ruby-core considers a major release at this point (x.0.0 or x.y.0).

#36 - 11/14/2017 02:24 PM - ana06 (Ana Maria Martinez Gomez)

It is already 4 years since this was pointed out. I am really surprised that it is still inconsistent. It is not expected and make developers checking the documentation for something that could be completely clear. Some people claimed here that they don't care about what the method returns, then they also shouldn't care if this is fix. Remember that Ruby brags of its simplicity and productivity, is this simple and encourage productivity?

This is even raised in talks: https://speakerdeck.com/bbatsov/ruby-4-to-infinity-and-beyond 700 Ruby developers were at EuRuKo2017 and all of us laughed of this. But the true is that is is making all of us wasting time. Let's stop laughing about Ruby inconsistencies and fix them!

#37 - 07/11/2018 09:26 AM - bozhidar (Bozhidar Batsov)

ana06 (Ana Maria Martinez Gomez) wrote:

> It is already 4 years since this was pointed out. I am really surprised that it is still inconsistent. It is not expected and make developers checking the documentation for something that could be completely clear. Some people claimed here that they don't care about what the method returns, then they also shouldn't care if this is fix. Remember that Ruby brags of its simplicity and productivity, is this simple and encourage productivity?
>
> This is even raised in talks: https://speakerdeck.com/bbatsov/ruby-4-to-infinity-and-beyond 700 Ruby developers were at EuRuKo2017 and all of us laughed of this. But the true is that is is making all of us wasting time. Let's stop laughing about Ruby inconsistencies and fix them!

Well, that's part of the epic aversion to deprecating stuff in Ruby, which I've never managed to understand. Deprecations are not breaking anything, but advance the language forward.

My single suggestion would be to flag this method as deprecated and just advise users to use !zero? instead. If someone insists to have a similar method lets just name it non_zero? or not_zero? and be done with it.

#38 - 02/15/2019 03:45 AM - ecoologic (erik ecoologic)

I also find this inconsistency very weird and I find ana06's suggestion very good, which I'd expand with:

- Deprecate nonzero?
- A new method non_zero returning self : nil: Methods ending in ? should consistently return only true : false, maybe nil

#39 - 06/11/2019 12:55 PM - shevegen (Robert A. Heiler)

Quite a long old discussion. I checked before replying, to make sure I don't comment too often in one particular issue. So this is my first reply; and I will try to make comments that summarize my opinion.

First, I would like to start that I do not think that this is a hugely important discussion either way. I understand that those who may wish to change the current behaviour feel different, but this is my personal opinion. :)

So in some ways, I am like headius here in not having any really strong opinion.

Now to the issue of .zero? first - I do sometimes use .zero? but quite rarely. For some reason I do not seem to need it. I do not remember a single case where I had to use .nonzero?. I think the name is not hugely elegant; it feels a bit clumsy. So in some ways I agree with those who would suggest e. g.:

```
if !object.zero?
```

In general I prefer "if" clause checking; I noticed that my brain needs more time when processing "unless" conditions. I also use "unless" just fine but it is not a clear winner to me per se, as opposed to "if !foo". But this is an aside.

To the very situation itself - personally I think it would be more consistent to have .nonzero? just behave exactly like !object.zero? - I don't think people can expect for the "opposite" of a method that returns true or false, to either return self, or nil. This is from a consistency point of view - as was explained there was a history for the behaviour too.

There is, I believe, no way for this to change in ruby 3.0. Perhaps past ruby 3.x this behaviour may be revisited, but again - I personally don't really care either way. To me it is not really important. This may be different for other folks, but since I don't use .nonzero? myself, and rarely use .zero?, it just does not affect me. So I guess this issue here is mostly about people who use .nonzero?.

I am not sure how many use this, though - to me it seems as if only very few people use it; more people use @@foo variables in code, for example.

I noticed this issue here due to this recent blog article:

https://metaredux.com/posts/2019/06/11/weird-ruby-zeroing-in-on-a-couple-of-numeric-predicates.html

The article also writes:

> Most of the time you probably won't experience any issues related to that inconsistency, but there are certainly cases where it is going to bite you.

But since I don't really use these methods, they can not affect my own code now can they. ;)

Bozhidar wrote:

> Deprecations are not breaking anything, but advance the language forward.

This depends on the change itself of course. Matz talked about it in a presentation e. g. good change, bad change. For example, if you can avoid a change. I understand both points, since it will always be a struggle of people who prefer change, as opposed to those who don't like a change (and they can not avoid it). The issue here is so minor, though - I think we need to put this into perspective too. If we look at past ruby 1.8.x, changes such as encoding, or the yaml change from syck to psych - these were larger changes. I have only recently switched to UTF-8 + psych finally (oddly enough due to emojis and unicode "building blocks" for commandline "interfaces" - this is actually more useful than plain ASCII, so that change was ultimately worth it; took me ~2 weeks or so of initial time investment).

I disagree about adding a new method called non_zero that would return self or nil. To me this makes no sense.

I also do not think that the method .nonzero? itself should be deprecated (and removed), but I am in favour of changing it to be consistent with .zero? in the long run, just as headius wrote back then.

I think there is one important thing to note, though - while some folks here post about "purity" of a language, the ruby core team has stated several times before that real usage of ruby is a primary focus rather than abstract "perfect" design. Meaning - if there is a good specific use case then a change may be much easier than not having a good specific use case.

I write this in general because I think this actually helps explain other changes in ruby where a good use case was given (see how the safe navigation operator was added). And all use cases have to be checked for side effects IF a change is made, too.

Obviously I am biased too because I don't really depend on either #zero? or #nonzero?, but even this aside, I think this is such a small issue either way.