Ruby - Feature #5372

Promote blank? to a core protocol

09/27/2011 06:18 PM - regularfry (Alex Young)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	1.9.4	

Description

I don't think there's been a project I've used that hasn't made use of this pattern:

if thing.nil? || thing.empty?

somewhere in its source. This is necessary because it is idiomatic to return nil where other languages might return a null object, and there is no universal test for nullity which a user-implemented class can provide without issues.

Facets (and ActiveSupport) define a #blank? protocol which allows for the above to be replaced with:

if thing.blank?

Being able to type this on a first iteration saves forgetting one of the cases and having to come back to fix a bug later. For projects where I cannot directly use Facets or ActiveSupport, I always find that I rewrite a version for myself. It would be very convenient not to have to do this every time, and this is clearly a common case, so I propose that #blank? be implemented on the following classes:

Object: to return false
String: aliased to #empty?
NilClass: to return true
TrueClass: to return false
FalseClass: to return true
Array: aliased to #empty?
Hash: aliased to #empty?
Fiber: to return !alive?
Numeric: aliased to #zero?
IO: aliased to #closed?
MatchData: to return #to_s.blank?
Process::Status: aliased to #exited?
Range: to return self.include?(self.begin)
Struct: subclass instances to return values.blank?
Thread: to return !alive?
ThreadGroup: to return list.blank?

Some of these uses aren't described by the word "blank?" very well (and ActiveSupport's String#blank? is somewhat different), so as a sub-feature I'd like to suggest "null?" as an alternative method name.

Apologies if this has been proposed and rejected before, but a quick search of redmine didn't show anything relevant.

History

#1 - 09/28/2011 03:53 AM - tenderlovemaking (Aaron Patterson)

On Tue, Sep 27, 2011 at 06:18:19PM +0900, Alex Young wrote:

> Issue #5372 has been reported by Alex Young.
>
> ---
>
> Feature #5372: Promote blank? to a core protocol
> http://redmine.ruby-lang.org/issues/5372
>
> Author: Alex Young
> Status: Open
> Priority: Normal
> Assignee:
> Category: core
> Target version: 1.9.4

I don't think there's been a project I've used that hasn't made use of this pattern:

if thing.nil? || thing.empty?

somewhere in its source. This is necessary because it is idiomatic to return nil where other languages might return a null object, and there is no universal test for nullity which a user-implemented class can provide without issues.

Facets (and ActiveSupport) define a #blank? protocol which allows for the above to be replaced with:

if thing.blank?

Being able to type this on a first iteration saves forgetting one of the cases and having to come back to fix a bug later. For projects where I cannot directly use Facets or ActiveSupport, I always find that I rewrite a version for myself. It would be very convenient not to have to do this every time, and this is clearly a common case, so I propose that #blank? be implemented on the following classes:

Object: to return false
String: aliased to #empty?
NilClass: to return true
TrueClass: to return false
FalseClass: to return true
Array: aliased to #empty?
Hash: aliased to #empty?
Fiber: to return !alive?
Numeric: aliased to #zero?
IO: aliased to #closed?
MatchData: to return #to_s.blank?
Process::Status: aliased to #exited?
Range: to return self.include?(self.begin)
Struct: subclass instances to return values.blank?
Thread: to return !alive?
ThreadGroup: to return list.blank?

Some of these uses aren't described by the word "blank?" very well (and ActiveSupport's String#blank? is somewhat different), so as a sub-feature I'd like to suggest "null?" as an alternative method name.

Apologies if this has been proposed and rejected before, but a quick search of redmine didn't show anything relevant.

"empty?" implies that you're performing the operation on a set. Why would you treat these other objects as sets? It doesn't make sense to me.

Not to mention defining the method on every possible object seems bad. What if my function shouldn't be dealing with Thread objects? I'd rather a NoMethodError be raised.

--
Aaron Patterson
http://tenderlovemaking.com/

#2 - 09/28/2011 07:27 AM - naruse (Yui NARUSE)

- File deleted (noname)

#3 - 09/28/2011 07:38 AM - naruse (Yui NARUSE)

This seems the long discussed question, what is a zero element.
But I don't think this proposal is not worth breaking ActiveSupport's blank?.

Anyway what is the usecase of this?
I doubt a closed io is a zero element.
Do you have a case that you are happy if it is a zero element.

#4 - 09/28/2011 08:53 AM - regularfry (Alex Young)

On 27/09/2011 19:46, Aaron Patterson wrote:

> On Tue, Sep 27, 2011 at 06:18:19PM +0900, Alex Young wrote:
>
> > Issue #5372 has been reported by Alex Young.
> >
> > ---
> >
> > Feature #5372: Promote blank? to a core protocol
> > http://redmine.ruby-lang.org/issues/5372

Author: Alex Young
Status: Open
Priority: Normal
Assignee:
Category: core
Target version: 1.9.4

I don't think there's been a project I've used that hasn't made use of this pattern:

if thing.nil? || thing.empty?

somewhere in its source. This is necessary because it is idiomatic to return nil where other languages might return a null object, and there is no universal test for nullity which a user-implemented class can provide without issues.

Facets (and ActiveSupport) define a #blank? protocol which allows for the above to be replaced with:

if thing.blank?

Being able to type this on a first iteration saves forgetting one of the cases and having to come back to fix a bug later. For projects where I cannot directly use Facets or ActiveSupport, I always find that I rewrite a version for myself. It would be very convenient not to have to do this every time, and this is clearly a common case, so I propose that #blank? be implemented on the following classes:

Object: to return false
String: aliased to #empty?
NilClass: to return true
TrueClass: to return false
FalseClass: to return true
Array: aliased to #empty?
Hash: aliased to #empty?
Fiber: to return !alive?
Numeric: aliased to #zero?
IO: aliased to #closed?
MatchData: to return #to_s.blank?
Process::Status: aliased to #exited?
Range: to return self.include?(self.begin)
Struct: subclass instances to return values.blank?
Thread: to return !alive?
ThreadGroup: to return list.blank?

Some of these uses aren't described by the word "blank?" very well (and ActiveSupport's String#blank? is somewhat different), so as a sub-feature I'd like to suggest "null?" as an alternative method name.

Apologies if this has been proposed and rejected before, but a quick search of redmine didn't show anything relevant.

> "empty?" implies that you're performing the operation on a set. Why would you treat these other objects as sets? It doesn't make sense to me.

That's the other way around. I'm not suggesting that the others are sets. I'm saying that for Arrays, Strings and Hashes, #empty? expresses the idea that these are null objects: they have no content, and operations on them are commonly no-ops. I'm then generalising that and saying that this idea of there being a null instance of a class is applicable to more cases than just those classes which can be treated as sets.

> Not to mention defining the method on every possible object seems bad. What if my function shouldn't be dealing with Thread objects? I'd rather a NoMethodError be raised.

We already have nil?, to_s and so forth which are defined everywhere. I'm suggesting an new protocol on that level, one which a user-defined class can participate in.

Think of it like the Null Object pattern in reverse. Because the core API commonly returns nil in error cases rather than having methods with a single return type, we can't blindly use the returned instance without a couple of type checks. Instead, we need to ask each instance "are you null?" before operating on it. The blank? method contains the logic which would otherwise have been used when deciding to build a Null Object, if that was the idiom.

This becomes more useful when you have:

```
module Blank
  def Blank.===(x)
    x.blank?
  end
end
```

Because then you can do this:

```
case thingy
when Blank
  # catch-all
# other cases
end
```

which I quite like.

--
Alex

#5 - 09/28/2011 09:29 AM - regularfry (Alex Young)

On 27/09/2011 23:38, Yui NARUSE wrote:

> Issue #5372 has been updated by Yui NARUSE.
>
> This seems the long discussed question, what is a zero element.
> But I don't think this proposal is not worth breaking ActiveSupport's blank?.

True, that would be unfortunate. For that reason I marginally prefer the null? name but thought blank? would be a more familiar concept.

> Anyway what is the usecase of this?

In the short term, it saves accidentally missing bugs where I forget to complete the type check. It's an addition which is simple to implement (at least naively) and does not break existing code, and would make my projects a little bit simpler.

Longer-term, I think it's an interesting primitive. It gives an alternative approach to being able to subclass FalseClass or NilClass, which is also occasionally suggested. One use case for this is in implementing non-exception error classes, which would give you something a little like Haskell's Either.

> I doubt a closed io is a zero element.
> Do you have a case that you are happy if it is a zero element.

Yes, if it's set to autoclose. I'm not wedded to the definition on IO, though - if you think there's a more useful one then I'm all ears.

--
Alex

> Feature #5372: Promote blank? to a core protocol
> http://redmine.ruby-lang.org/issues/5372
>
> Author: Alex Young
> Status: Open
> Priority: Normal
> Assignee:
> Category: core
> Target version: 1.9.4
>
> I don't think there's been a project I've used that hasn't made use of this pattern:
>
> if thing.nil? || thing.empty?
>
> somewhere in its source. This is necessary because it is idiomatic to return nil where other languages might return a null object, and there is no universal test for nullity which a user-implemented class can provide without issues.

> Facets (and ActiveSupport) define a #blank? protocol which allows for the above to be replaced with:
>
> if thing.blank?
>
> Being able to type this on a first iteration saves forgetting one of the cases and having to come back to fix a bug later. For projects where I cannot directly use Facets or ActiveSupport, I always find that I rewrite a version for myself. It would be very convenient not to have to do this every time, and this is clearly a common case, so I propose that #blank? be implemented on the following classes:
>
> Object: to return false
> String: aliased to #empty?
> NilClass: to return true
> TrueClass: to return false
> FalseClass: to return true
> Array: aliased to #empty?
> Hash: aliased to #empty?
> Fiber: to return !alive?
> Numeric: aliased to #zero?
> IO: aliased to #closed?
> MatchData: to return #to_s.blank?
> Process::Status: aliased to #exited?
> Range: to return self.include?(self.begin)
> Struct: subclass instances to return values.blank?
> Thread: to return !alive?
> ThreadGroup: to return list.blank?
>
> Some of these uses aren't described by the word "blank?" very well (and ActiveSupport's String#blank? is somewhat different), so as a sub-feature I'd like to suggest "null?" as an alternative method name.
>
> Apologies if this has been proposed and rejected before, but a quick search of redmine didn't show anything relevant.

#6 - 10/01/2011 04:53 PM - drbrain (Eric Hodel)

On Sep 27, 2011, at 6:52 PM, Alex Young wrote:

> Think of it like the Null Object pattern in reverse.

The Null Object is described on the C2 wiki and its related pages:

http://c2.com/cgi/wiki?NullObject

It seems to be better than implementing Object#empty?

Can you show me a description of the opposite?

> Because the core API commonly returns nil in error cases..

Can you show some examples? I don't seem to write nil checks very often when using core methods, but maybe I am forgetting. I would rather improve the API to have fewer nil return values than add #empty? everywhere.

> case thingy
> when Blank
> # catch-all
>
> ## other cases
>
> end

What about:

case thingy

other cases

else

catch-all

end

#7 - 10/03/2011 06:53 AM - regularfry (Alex Young)

Eric Hodel wrote in post #1024462:

> On Sep 27, 2011, at 6:52 PM, Alex Young wrote:
>
> > Think of it like the Null Object pattern in reverse.
>
> The Null Object is described on the C2 wiki and its related pages:
>
> http://c2.com/cgi/wiki?NullObject
>
> It seems to be better than implementing Object#empty?

Where it's an option, yes, it's better, but in places where I call #null? or #empty? I don't necessarily have the flexibility to use it. Where I'm handed a return value by a method in some library, core, or a gem, or whatever, I've often got to make a type check of some sort before safely operating on it, and that check is frequently a check that the object is of a type that it makes sense to operate on.

> Can you show me a description of the opposite?

What I mean by "in reverse" is that with the Null Object, we have an instance which silently does the right thing. We don't have to care that it's null, we just call methods on it like we would on a non-Null instance.

With a #null? or #blank? method, we instead have a way to ask each instance directly whether it's null, without having to care about its class. If it quacks like a null, then it's null.

> > Because the core API commonly returns nil in error cases..
>
> Can you show some examples? I don't seem to write nil checks very often when using core methods, but maybe I am forgetting.

Having a quick look over the core docs, there's quite a few in File::Stat and Process::Status, all the try_convert() methods, Kernel.caller, Kernel.system, arguably String#slice and Regexp#match (although I can't see the latter being reasonably alterable), and Thread#status at least.

> > case thingy
> > when Blank
> > # catch-all
>
> ## other cases
>
> > end
>
> What about:
>
> case thingy
>
> ## other cases
>
> else
>
> ## catch-all
>
> end

Yep, that's another way to do the same sort of thing, but with a Blank or Null it's more explicit and more flexible. With a bare "case...else..." you have to handle both correct nulls and erroneous

values in the "else" clause. With Null, you can leave the "else" clause purely for handling the error case, where you've somehow got a response you weren't expecting. I think it's clearer.

--
Alex

--
Posted via http://www.ruby-forum.com/.

#8 - 10/04/2011 06:29 AM - drbrain (Eric Hodel)

On Oct 2, 2011, at 2:38 PM, Alex Young wrote:

> Eric Hodel wrote in post #1024462:
>
> > On Sep 27, 2011, at 6:52 PM, Alex Young wrote:
> > Can you show me a description of the opposite?
>
> What I mean by "in reverse" is that with the Null Object, we have an instance which silently does the right thing. We don't have to care that it's null, we just call methods on it like we would on a non-Null instance.
>
> With a #null? or #blank? method, we instead have a way to ask each instance directly whether it's null, without having to care about its class. If it quacks like a null, then it's null.

I mean, on the C2 wiki or somewhere else on the internet. Can you show other languages that have benefited from a similar implementation? If there is such a document maybe it can help us understand.

> > Because the core API commonly returns nil in error cases..
>
> Can you show some examples? I don't seem to write nil checks very often when using core methods, but maybe I am forgetting.

> Having a quick look over the core docs, there's quite a few in File::Stat and Process::Status, all the try_convert() methods, Kernel.caller, Kernel.system, arguably String#slice and Regexp#match (although I can't see the latter being reasonably alterable), and Thread#status at least.

When does caller return a non-Array?

> > case thingy
> > when Blank
> > # catch-all
>
> > # other cases
>
> > end
>
> What about:
>
> case thingy
>
> # other cases
>
> else
>
> # catch-all
>
> end

> Yep, that's another way to do the same sort of thing, but with a Blank or Null it's more explicit and more flexible. With a bare

> "case...else..." you have to handle both correct nulls and erroneous values in the "else" clause. With Null, you can leave the "else" clause purely for handling the error case, where you've somehow got a response you weren't expecting. I think it's clearer.

The problem I see is that adding #empty? to every class is confusing.

Should File::Stat#empty? returning true to mean the file is empty? Or should it always return false to say "the file exists"

What would Process::Status#empty? mean? Would false mean that the program had exited non-zero or that the program had exited with any status?

Kernel#system and Thread#status return true, false, or nil, so combining "non-zero exit" and "command failed" into #empty? isn't clearer to read than 'if system(command) then â€¦ else abort "#{command} failed" end'

While it might make String#split or Regexp#match and try_convert usage clearer, it adds much confusion otherwise.

#9 - 10/04/2011 07:23 PM - regularfry (Alex Young)

On 03/10/11 22:25, Eric Hodel wrote:

> On Oct 2, 2011, at 2:38 PM, Alex Young wrote:
>
> > Eric Hodel wrote in post #1024462:
> >
> > > On Sep 27, 2011, at 6:52 PM, Alex Young wrote:
> > > Can you show me a description of the opposite?
> >
> > What I mean by "in reverse" is that with the Null Object, we have an instance which silently does the right thing. We don't have to care that it's null, we just call methods on it like we would on a non-Null instance.
> >
> > With a #null? or #blank? method, we instead have a way to ask each instance directly whether it's null, without having to care about its class. If it quacks like a null, then it's null.
>
> I mean, on the C2 wiki or somewhere else on the internet. Can you show other languages that have benefited from a similar implementation? If there is such a document maybe it can help us understand.

To my knowledge it's most similar to Either in Haskell, but you have to squint a bit to see it:
http://haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Either.html

If you renamed #empty? to #left? the similarity should be a little clearer.

If you look at Perl 6, there's also a stack of similar-looking functionality around Mu, Failure and Whatever - specifically the .defined method is close to what I'm thinking, but they've taken it a *lot* further.

> > Because the core API commonly returns nil in error cases..
> >
> > Can you show some examples? I don't seem to write nil checks very often when using core methods, but maybe I am forgetting.
>
> Having a quick look over the core docs, there's quite a few in File::Stat and Process::Status, all the try_convert() methods, Kernel.caller, Kernel.system, arguably String#slice and Regexp#match (although I can't see the latter being reasonably alterable), and Thread#status at least.
>
> When does caller return a non-Array?

```
$ irb
ruby-1.9.2-p290 :001 > caller(22)
=> nil
```

It's when the depth parameter exceeds the current stack depth.

> > > case thingy
> > > when Blank
> > > # catch-all
>
> > > ## other cases
>
> > > end
>
> > What about:
>
> > case thingy
>
> > ## other cases
>
> > else
>
> > ## catch-all
>
> > end
>
> Yep, that's another way to do the same sort of thing, but with a Blank or Null it's more explicit and more flexible. With a bare "case...else..." you have to handle both correct nulls and erroneous values in the "else" clause. With Null, you can leave the "else" clause purely for handling the error case, where you've somehow got a response you weren't expecting. I think it's clearer.

> The problem I see is that adding #empty? to every class is confusing.

Part of that is down to the name. #null? is better because it doesn't imply that the receiver is a container.

You'll notice that I *didn't* suggest an implementation for Symbol: sometimes it doesn't make sense for any instance of a class to be null. You could make the same argument about Numeric, but I've occasionally found treating #zero? as a null test to be useful in the past.

> Should File::Stat#empty? returning true to mean the file is empty? Or should it always return false to say "the file exists"

I'd go for the latter, personally.

> What would Process::Status#empty? mean? Would false mean that the program had exited non-zero or that the program had exited with any status?

I mentioned upthread that it would be useful aliased to #exited?, but I'd really prefer it to test whether the process was actually running - from the documentation of #exited? it sounds like processes that segfault will cause #exited? to return false.

> Kernel#system and Thread#status return true, false, or nil, so combining "non-zero exit" and "command failed" into #empty? isn't clearer to read than 'if system(command) then â€¦ else abort "#{command} failed" end'

Sure. I'd say Kernel#system is an interesting example, though. Say I was being implementing it as a third-party library, but with a twist: instead of returning nil on command failure, I want to capture some details about the failure and wrap them up in a hypothetical ProcessFailure instance. Some of the time, I don't care about the details of the failure, and other times I do, but in no case do I think of this as warranting an Exception. Now, if I say:

class ProcessFailure
def null?
true
end

end

then when I *don't* care which happened, either the command failing or
it having a non-zero exit, I can just say:

unless mysystem(foo).null?
it worked!
end

and when I *do* care, it's:

unless (result = mysystem(foo)).null?
# it worked!
else
# It didn't, so try to do something useful with the error details
$stderr.puts result.to_s if result
end

Note that while it might make the conditionals cleaner here, I *can't*
do the obvious thing of:

class ProcessFailure < FalseClass; end

because that's just not how booleans work.

> While it might make String#split or Regexp#match and try_convert usage clearer, it adds much confusion otherwise.

As I mentioned above, there are definitely cases where null? should
never be true for a given class because if you have a value, it's not
null by definition. It's simple enough to leave the default #null? ->
false implementation in place for them.

--
Alex

#10 - 10/05/2011 12:59 AM - nobu (Nobuyoshi Nakada)

Hi,

(11/10/03 6:38), Alex Young wrote:

> Yep, that's another way to do the same sort of thing, but with a Blank
> or Null it's more explicit and more flexible. With a bare
> "case...else..." you have to handle both correct nulls and erroneous
> values in the "else" clause. With Null, you can leave the "else" clause
> purely for handling the error case, where you've somehow got a response
> you weren't expecting. I think it's clearer.

The "flexibility" (or ambiguity) seems the sign that it should not be
in core, to me.

Indeed ActiveSupport has many interesting features, however they are
basically designed for Rails concerned applications and may not be
suitable for the language core.

--
Nobu Nakada

#11 - 10/05/2011 02:23 AM - regularfry (Alex Young)

On 04/10/11 16:52, Nobuyoshi Nakada wrote:

> Hi,
>
> (11/10/03 6:38), Alex Young wrote:
>
> > Yep, that's another way to do the same sort of thing, but with a Blank
> > or Null it's more explicit and more flexible. With a bare
> > "case...else..." you have to handle both correct nulls and erroneous
> > values in the "else" clause. With Null, you can leave the "else" clause
> > purely for handling the error case, where you've somehow got a response
> > you weren't expecting. I think it's clearer.

> The "flexibility" (or ambiguity) seems the sign that it should not be in core, to me.

I'm not sure I understand. Where's the ambiguity?

> Indeed ActiveSupport has many interesting features, however they are basically designed for Rails concerned applications and may not be suitable for the language core.

You could make the same argument about any library. If something is in ActiveSupport and not in any of the more specifically web-framework Rails libraries, it's also a sign that it's a feature that people find generally useful *outside* Rails.

Note that similar functionality is *also* in Facets, which aims for general applicability.

With respect, I don't find arguments on where a feature has come from to be be particularly relevant.

--
Alex

#12 - 10/05/2011 02:23 PM - nobu (Nobuyoshi Nakada)

Hi,

(11/10/05 2:14), Alex Young wrote:

> On 04/10/11 16:52, Nobuyoshi Nakada wrote:
>
> > The "flexibility" (or ambiguity) seems the sign that it should not be in core, to me.
>
> I'm not sure I understand. Where's the ambiguity?

"Flexibility" can cause ambiguity, sometimes. Seems it depends on application contexts too much.

> You could make the same argument about any library. If something is in ActiveSupport and not in any of the more specifically web-framework Rails libraries, it's also a sign that it's a feature that people find generally useful *outside* Rails.

Sounds like it's web-framework specific.

> With respect, I don't find arguments on where a feature has come from to be be particularly relevant.

Of course, not. In fact, I'm one of who proposed to introduce Symbol#to_proc strongly. I just said that everything in ActiveSupport won't be suitable for the core always.

--
Nobu Nakada

#13 - 10/05/2011 05:53 PM - regularfry (Alex Young)

On 05/10/11 05:59, Nobuyoshi Nakada wrote:

> Hi,
>
> (11/10/05 2:14), Alex Young wrote:
>
> > On 04/10/11 16:52, Nobuyoshi Nakada wrote:
> >
> > > The "flexibility" (or ambiguity) seems the sign that it should not be in core, to me.

> > I'm not sure I understand. Where's the ambiguity?
>
> "Flexibility" can cause ambiguity, sometimes. Seems it depends on application contexts too much.

I don't see that it's any more ambiguous than having Object#===. #==. #eql? and #equal?.

#nil? has the same relationship to #null? as #== has to #===. One is a strict equality comparison, the other is a loose match. It's context-dependent in the same way as #=== is: user classes can redefine it and participate in a global protocol.

> > You could make the same argument about any library. If something is in ActiveSupport and not in any of the more specifically web-framework Rails libraries, it's also a sign that it's a feature that people find generally useful *outside* Rails.
>
> Sounds like it's web-framework specific.

I disagree. Why does it seem that way to you?

> > With respect, I don't find arguments on where a feature has come from to be be particularly relevant.
>
> Of course, not. In fact, I'm one of who proposed to introduce Symbol#to_proc strongly. I just said that everything in ActiveSupport won't be suitable for the core always.

I'm not suggesting it because it's in ActiveSupport. I'm suggesting it because it's a generally useful concept.

--
Alex

#14 - 10/05/2011 06:59 PM - nobu (Nobuyoshi Nakada)

Hi,

(11/10/05 17:32), Alex Young wrote:

> I'm suggesting it because it's a generally useful concept.

I'm suggesting some doubt on its generalness.

--
Nobu Nakada

#15 - 10/05/2011 11:59 PM - regularfry (Alex Young)

On 05/10/11 10:53, Nobuyoshi Nakada wrote:

> Hi,
>
> (11/10/05 17:32), Alex Young wrote:
>
> > I'm suggesting it because it's a generally useful concept.
>
> I'm suggesting some doubt on its generalness.

It's as general an idea as #===, but #=== has syntactic support and gets to participate in case expressions.

--
Alex

#16 - 10/07/2011 07:59 AM - Anonymous

> Facets (and ActiveSupport) define a #blank? protocol which allows for the above to be replaced with:
>
> Â if thing.blank?

I use this all the time

if thing.present?

..

end

enthusiastic +1 :)
-roger-

#17 - 03/27/2012 03:26 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#18 - 03/28/2012 01:30 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Rejected

Let it stay in ActiveSupport. Generally, string is string, array is array, in Ruby.

Matz.