Ruby - Bug #5759

flatten calls to_ary on everything

12/14/2011 08:55 AM - trans (Thomas Sawyer)

Status:	Rejected		
Priority:	Normal		
Assignee:	nobu (Nobuyoshi Nakada)		
Target version:	1.9.3		
ruby -v:	ruby 1.9.3dev (2011-09-23 revision 33323) [x86_64-linux]	**Backport:**	

Description

I often ensure that I have an array by doing:

```
def foo=(x)
@foo = [x].flatten
end
```

But this has turned into a problem as of late, as it seems #flatten is calling #to_ary on every element in the array, and apparently catching the error raised if #to_ary isn't defined for that object. But that causes potential issues with objects that use #method_missing. I think #flatten should use respond_to?(:to_ary) to make sure an object can handle it before actually calling it.

Related issues:

Related to Ruby - Bug #6039: lambda vs proc; #to_ary w/ splat bug	**Rejected**	**02/17/2012**

History

#1 - 12/14/2011 10:05 AM - drbrain (Eric Hodel)

Use Kernel#Array:

```
$ ruby -e 'p Array("a\nb"), Array(["a\nb"])'
["a\nb"]
["a\nb"]
```

#2 - 12/14/2011 07:23 PM - regularfry (Alex Young)

On 14/12/11 01:05, Eric Hodel wrote:

> Issue #5759 has been updated by Eric Hodel.
>
> Use Kernel#Array:
>
> ```
> $ ruby -e 'p Array("a\nb"), Array(["a\nb"])'
> ["a\nb"]
> ["a\nb"]
> ```

Or a splat:

```
ruby-1.9.3-p0 :001 > a="a\nb"; [*a]
=> ["a\nb"]
ruby-1.9.3-p0 :002 > a=["a\nb"]; [*a]
=> ["a\nb"]
```

Not sure I disagree that #flatten should check first (or just leave the exception uncaught) though.

--
Alex

> Bug #5759: flatten calls to_ary on everything
> http://redmine.ruby-lang.org/issues/5759
>
> Author: Thomas Sawyer
> Status: Open
> Priority: Normal
> Assignee:

Category:
Target version: 1.9.3
ruby -v: ruby 1.9.3dev (2011-09-23 revision 33323) [x86_64-linux]

I often ensure that I have an array by doing:

def foo=(x)
@foo = [x].flatten
end

But this has turned into a problem as of late, as it seems #flatten is calling #to_ary on every element in the array, and apparently catching the error raised if #to_ary isn't defined for that object. But that causes potential issues with objects that use #method_missing. I think #flatten should use respond_to?(:to_ary) to make sure an object can handle it before actually calling it.

#3 - 02/17/2012 04:09 AM - ddebernardy (Denis de Bernardy)

Possibly related:

http://bugs.ruby-lang.org/issues/6039

#4 - 03/11/2012 05:02 PM - ko1 (Koichi Sasada)

- Assignee set to nobu (Nobuyoshi Nakada)

#5 - 03/18/2012 06:46 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

#6 - 12/30/2012 10:45 PM - nobu (Nobuyoshi Nakada)

- Status changed from Assigned to Rejected

When you define method_missing, you have to also define respond_to_missing? properly.

#7 - 12/31/2012 05:15 AM - trans (Thomas Sawyer)

=begin
Isn't the the problem that it doesn't bother to check (({#respond_to?})) at all?

class Baz
def method_missing(s)
s
end

```
def respond_to_missing?(s, x)
  return false if s == :to_ary
  true
end
```

end

b = Baz.new
b.respond_to?(:to_ary) #=> false
[Baz.new].flatten
=> in `flatten': can't convert Baz to Array (Baz#to_ary gives Symbol) (TypeError)

=end

#8 - 12/31/2012 10:46 AM - bitsweat (Jeremy Daer)

class Baz; def respond_to?(s, x) super unless s == :to_ary end end
=> nil
[Baz.new].flatten
=> [#Baz:0x007f8d3115c7d0]

#9 - 01/03/2013 12:10 PM - trans (Thomas Sawyer)

=begin
So it does call (({#respond_to?})) after all? Yet, I thought (({#respond_to_missing?})) was invented so people would not have to override

(({#respond_to?})). What's my misunderstanding? Surely we are not now expected to define both?
=end