Ruby master - Bug #8507

Keyword splat does not convert arg to Hash

06/10/2013 03:27 AM - stephencelis (Stephen Celis)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		
ruby -v:	ruby 2.0.0p195 (2013-05-14 revision 40734) [x86_64-darwin12.3.0]	**Backport:**

Description

A single splat is useful to convert an object to an array as it passes from one method to the next, so I expected the double-splat to do the same and convert the object to a hash.

```
def splat *args
  p args
end
def double_splat **kwargs
  p args
end
splat(*nil)  # []
splat(**nil) # TypeError: no implicit conversion of nil into Hash
```

For the sake of consistency, wouldn't it make sense to wrap the double-splatted arg with Hash() the same way a single-splatted arg is wrapped with Array()?

Related issues:

Has duplicate Ruby master - Bug #9291: array splatting a nil works, but hash ...	**Rejected**

History

#1 - 06/10/2013 08:23 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

I hate consistency as a reason to change. Do you really want to do splat(**nil)?

Matz.

#2 - 06/10/2013 01:29 PM - stephencelis (Stephen Celis)

Is there logic behind the current state that I'm unaware of? Consistency is predictability, which is important when learning a new interface. If splatted nils are swallowed, why not double-splatted nils? One raising where the other doesn't surprised me.

With args, you can be sure that the value passed is an array when splatted out. Splatting itself is seen as a method of conversion for me and other Rubyists I've spoken with. The fact that double-splatting makes no attempt to convert the value seems to make it useless as a convention. What is the difference between the following method invocations?

```
def kwmethod **kwargs
  p kwargs
end

hash = { hello: 'world' }

kwmethod(hash)
kwmethod(**hash)
```

With *args, there is a clear reason to splat vs. not. Why not create a similar analog with keyword arguments?

#3 - 06/10/2013 01:32 PM - stephencelis (Stephen Celis)

To return to the original question:

> Do you really want to do splat(**nil)?

I find that Ruby's flexibility has been nice when providing public interfaces in gems and libraries. It's nice to be able to pass in *nil to a public method and have it discard the argument as unnecessary. Likewise, it's nice to be able to accept **nil as the keyword argument to a public interface and have it discard the argument and look to its default keyword arguments instead.

#4 - 06/10/2013 02:47 PM - matz (Yukihiro Matsumoto)

You didn't explain why "it is nice". Could elaborate? Do we really need it?

FYI, in the early stage of Ruby, types are more flexible; integers can be treated as strings, nil can be treated as empty array. nil.to_a => [] is a left-over of the old days.

For consistency's sake, I'd rather remove nil.to_a if we don't see compatibility problems.

Matz.

#5 - 06/10/2013 03:58 PM - nobu (Nobuyoshi Nakada)

- Description updated

#6 - 06/11/2013 05:56 AM - stephencelis (Stephen Celis)

Hm... nil.to_h => {} is brand-new and creates an analog to nil.to_a, so why would we remove nil.to_a?

(**nil) would be another tool that a programmer could use when handing arguments off between methods. One can currently use a single splat to convert nil or an empty array into an empty arg list. The ability to use double-splat to convert nil, an empty array, or an empty hash into a keyword arg list is equally powerful. It means flexibility in input and in handing off and converting arguments among different methods.

For instance, there was a recent moment where I thought I could elegantly double-splat an argument passed to a method in order to convert it to a hash without using kwargs || {}, similar to what I've been able to do with a single splat and an args array (or, in some cases, a nil arg). The fact that I was unable to convert the object using a double-splat (while I can convert an object using a single-splat) seemed unpredictable. Rubyists have grown to expect that a single splat converts arguments passed in a certain way. I was hoping that handling the double-splat in a similar way would be an elegant way to resolve the current dissonance.

Here's a contrived example:

```
def outer_method **options
  inner_method(**options[:inner_options])
end
```

The ability to pass a value that may be a hash or may be nil as keyword arguments simplifies code (you don't have to check for nil) and makes it apparent as to what kind of value the inner method is taking (the same way a single splat denotes an args list).

I realize that keyword arguments are relatively new. I've already come up against some roadblocks using them (and finding their behavior fundamentally change between patchlevel 0 and 195). I think it's a great time to refine how we expect them to behave.

Again, the unexpected difference in behavior:

```
Array(nil)  # => []
nil.to_a    # => []
[*nil]      # => []

Hash(nil)   # => {}
nil.to_h    # => {}
{**nil}     # TypeError: no implicit conversion of nil into Hash
```

#7 - 06/15/2013 06:38 AM - kainosnoema (Evan Owen)

I'm glad this is getting addressed. Handling nil values is one of the most tiring and error-prone aspects of coding in Ruby. Much has been said about the mistake of introducing NULL references in languages (http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare , among others), and there are considerable efforts being made in many places to remove much of the pain that they cause.

I've also personally run into the situation shown in stephencelis's example above, and was quite surprised and disappointed by it. I'm 100% in support of any effort to gracefully handle nil values when coercing. The consistency added by stephencelis's patch is a big win, IMO.

#8 - 07/07/2013 10:23 AM - stephencelis (Stephen Celis)

Tried my best to thoughtfully address questions before: http://bugs.ruby-lang.org/issues/8507#note-6

Does anyone have any additional thoughts on the matter?

#9 - 07/12/2013 10:34 AM - mcmire (Elliot Winkler)

So let's think about why we have splat(*nil). Ordinarily you wouldn't write this code like that of course, but you might have a variable foo and unbeknownst to you (or perhaps not), foo happens to be nil, so you end up with splat(*nil). So I think we're getting a little distracted here by the nil.

Let's think about why we use splat at all. We use it because we have an array, and we want to call a method and spread the array across to create a list of arguments. So if foo is [1, 2, 3], splat(*foo) is splat(1, 2, 3).

Why don't we have an equivalent splat syntax for hashes? Well because we don't need it. You can just say splat(foo), assuming foo is a hash, and it just works. What would splat(**foo) even *do*?

For instance say #splat is defined as:

```
def splat(foo: 'bar', **kwargs)
end
```

Now assuming foo is {foo: 'bliz', a: 'b', c: 'd'}, what would splat(**foo) do? Wouldn't this just end up calling splat(foo: 'bliz', a: 'b', c: 'd'), and therefore be the same as calling splat(foo)?

So while I agree for consistency's sake that foo(**whatever) should do something just as foo(*whatever) does something, I don't really know if this syntax would be used.

#10 - 07/15/2013 05:38 AM - stephencelis (Stephen Celis)

Thanks for weighing in. Given that nil is so prevalent in the language, I don't think we can ignore it.

Also, **foo does already exist in the language, and it does do something (validates that the hash acts like kwargs and raises a TypeError if any of the keys aren't symbols).

Meanwhile, what if splat were defined as:

```
def splat(foo = {}, **kwargs)
  p foo, kwargs
end
```

What should happen with something like the following?

```
splat(foo: 'bar', a: 'b', **{c: 'd'})
```

Should it return:

```
[{}, {:foo=>'bar', :a=>'b', :c=>'d'}]
```

Or should it return:

```
[{:foo=>'bar', :a=>'b'}, {:c=>'d'}]
```

Is it something one can easily guess?

And what should this do?

```
splat(foo: 'bar', 'baz' => 'fizz', buzz: 'bucket')
```

Heck, what if {**hash_with_string_keys} actually did something like Active Support's symbolize_keys?

Right now, the behavior seems ambiguous in general.

#11 - 01/30/2014 06:17 AM - hsbt (Hiroshi SHIBATA)

- Target version changed from 2.1.0 to 2.2.0

#12 - 07/17/2014 06:38 PM - stephencelis (Stephen Celis)

Another issue with keyword args and consistency with args:

- *args are immutable
- **kwargs are mutable

```
def splat *args
  args << 'foobar'
end
arr = []
splat(*arr)
arr # => []

def double_splat **kwargs
  kwargs[:foo] = 'bar'
end
hash = {}
double_splat(**hash)
hash # => {:foo=>"bar"}
```

#13 - 07/18/2014 10:18 AM - marcandre (Marc-Andre Lafortune)

Stephen Celis wrote:

- **kwargs are mutable

```
def double_splat **kwargs
  kwargs[:foo] = 'bar'
end
hash = {}
double_splat(**hash)
hash # => {:foo=>"bar"}
```

I believe this is a bug. Matz, could you confirm?

#14 - 07/18/2014 12:15 PM - nobu (Nobuyoshi Nakada)

- Description updated

It's #9776.

#15 - 01/05/2018 09:00 PM - naruse (Yui NARUSE)

- Target version deleted (2.2.0)

#16 - 12/19/2019 11:22 PM - mame (Yusuke Endoh)

- Has duplicate Bug #9291: array splatting a nil works, but hash splatting a nil does not added

#17 - 12/19/2019 11:28 PM - mame (Yusuke Endoh)

- Backport deleted (1.9.3: UNKNOWN, 2.0.0: UNKNOWN)

- Status changed from Feedback to Rejected

As far as I understand, no one showed any actual use case that justifies adding nil.to_h #=> {}.

I agree with matz, it would be better to remove nil.to_a #=> [] instead of adding nil.to_h #=> {} for this issue. I don't think it is possible for the compatibility reason, though.

Files

to-hash-kwarg.patch	939 Bytes	06/10/2013	stephencelis (Stephen Celis)