Ruby master - Bug #16178

Numbered parameters: _1 should be the same as |x| and _0 should not exist

09/24/2019 01:21 PM - Eregon (Benoit Daloze)

Status:	Closed		
Priority:	Normal		
Assignee:	matz (Yukihiro Matsumoto)		
Target version:			
ruby -v:	ruby 2.7.0dev (2019-09-24T12:57:54Z master 0e84eecc17) [x86_64-linux]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN

Description

Currently on trunk:

```
array = ["string", 42, [1, 2]]

array.map { |x| x * 2 }
# => ["stringstring", 84, [1, 2, 1, 2]]

array.map { _1 * 2 }
# => ["stringstring", 84, 2]
```

Oops, this trivial code just lost data and completely ignored the element class!
This is clearly contrary to intuition and is very dangerous.

Using _0 instead has the correct behavior but it's clear we use 1-based indexing for numbered parameters,
and it doesn't solve that _1 has dangerous behavior.

Basically the current behavior is that _0 is the same as |x| and _1 is the same as |x,|.
|x,| is almost never used in Ruby, and for good reasons, it just throws away data/information/the class of the object.
Such a dangerous operation should only be done when it's explicit, and the trailing comma in |x,| shows that, but _1 does not.

So let's make _1 be |x| and remove _0.

I am going to be harsh, but this discussion has gone too long without any serious written argument for the current behavior:
I believe it's irrational and irresponsible to have _1 be |x,|, it's just going to lead to nasty bugs.

Try to convince me otherwise.
If not, in one week I want to apply this change.

From the discussion in https://bugs.ruby-lang.org/issues/15723#note-127
and in https://bugs.ruby-lang.org/issues/15708

Some reactions to this behavior in https://twitter.com/eregontp/status/1115318993299083265

Related issues:

Related to Ruby master - Bug #15708: Implicit numbered argument decomposes an...	**Rejected**
Related to Ruby master - Misc #15723: Reconsider numbered parameters	**Feedback**

Associated revisions

Revision 55e1e22b - 09/25/2019 04:01 AM - nobu (Nobuyoshi Nakada)

Changed numbered parameters semantics

- _1 (and no other numbered parameters) to work as |x|.
- giving up _0.

[ruby-core:95074] [Bug #16178]

History

#1 - 09/24/2019 01:21 PM - Eregon (Benoit Daloze)

- Related to Bug #15708: Implicit numbered argument decomposes an array added

#2 - 09/24/2019 01:21 PM - Eregon (Benoit Daloze)

- Related to Misc #15723: Reconsider numbered parameters added

#3 - 09/24/2019 01:23 PM - Eregon (Benoit Daloze)

Of course, having _ as the only unnamed parameter would have |x| semantics,
but I guess it's too late for that and now we have _<n> parameters.

#4 - 09/24/2019 01:25 PM - Eregon (Benoit Daloze)

- Description updated

#5 - 09/24/2019 01:29 PM - nobu (Nobuyoshi Nakada)

- Assignee changed from Eregon (Benoit Daloze) to matz (Yukihiro Matsumoto)

When _1 is same as |x|, what does [[1, 2]].map {_1 + _2} mean?
The meaning of _1 changes if _2 is used or not?

#6 - 09/24/2019 01:37 PM - Eregon (Benoit Daloze)

nobu (Nobuyoshi Nakada) wrote:

> When _1 is same as |x|, what does [[1, 2]].map {_1 + _2} mean?

It means [[1, 2]].map { |a,b| a + b } # => [3] of course.

> The meaning of _1 changes if _2 is used or not?

Yes, just like Proc#arity changes.

```
-> { _1 }.arity     #=> 1
-> { _2; _1 }.arity #=> 2
```

It's just consistent.
We have to accept the drawback that numbered parameters change arity, there is no way around that.

Changing arity with named parameters has the same effect ([[1, 2]].map { |a| a } to [[1, 2]].map { |a,b| a }).

My main point is we want _1 to be the non-dangerous behavior.

Changing arity can break things, that is not new, but indeed using numbered parameters makes the change less obvious in the source. That is intrinsically a drawback of numbered parameters (i.e., of not having an explicit list of parameters in one place).

#7 - 09/24/2019 01:53 PM - zverok (Victor Shepelev)

Some counter-points (that being said, I dislike the "unnamed parameters" idea as a whole, because I am afraid this feature -- being just "shorter" while staying imperative -- will stay in the way of adoption of more promising features like &obj.:method, experiments towards proper currying etc.):

First

> |x,| is almost never used in Ruby, and for good reasons

Honestly, we use it pretty frequently in our codebase and find it appealing. It is the shortest and pretty clear (YMMV, of course) way of processing only first elemente of "tuple-alike" arrays, like zips of data or data with indexes. Like, "find indexes of paragraphs matching some criteria":

```
paragraphs.each_with_index.select { |para,| para.match?(....) ].map(&:last)
```

Of course, here we can dive into bike-shedding about "real Jedi will write { |para, _idx|, because it is easier to ....", but I believe "some people use it" is enough counter-argument to "nobody uses it" :)

Second, we have "0 is whole, 1 is first match, etc." conditions in other places of Ruby—string matching:

```
m = "Serhii Zhadan".match(/^(.+?) (.+?)$/)
# => #<MatchData "Serhii Zhadan" 1:"Serhii" 2:"Zhadan">
m[0]
# => "Serhii Zhadan"
m[1]
# => "Serhii"
```

Third, I believe that in most of real, non-artificial situations, processing of sequence of heterogenous (some unpackable, some not) values with

"shortcut blocks args" is a situation to either avoid, or being aware of your data nature. And with homogenous data, the problem is non-existent, while benefits are pretty obvious:

```
<<-GRADES
John:4
Mary:5
Paul:3
Vasily:2
GRADES
  .lines(chomp: true)
  .map { _0.split(':') }
  .to_h { [_1, _2.to_i] }
```

In your proposal, the last line should be, like

```
to_h { [_1[0], _1[1].to_i] }
# or, OK, give up with them
to_h { |n, g| [n, g.to_i] }
```

I believe examples like mine, like "quick-and-somewhat dirty" experiments is exactly the target code to be simplified by the numbered parameters,

UPD: OK, the last point is answered while I've writing this :)

#8 - 09/24/2019 04:04 PM - shevegen (Robert A. Heiler)

Hmmm. I find underscores to be harder to read in combination with other parts (e. g. _abc or _123) whereas I use _ quite a bit in my own code.

I still like @1 @2 @3 etc... - however had, I have to admit that if the suggestion is to use _1 _2 _3 then I actually would rather prefer to not add this altogether. ;-)

I think the whole numbered parameters, while I like it in general, created some problems for people to adjust their mindset into. On the other hand, since ruby users are not forced to use it, they can ignore it, which happens with other features too.

zverok wrote:

> Some counter-points (that being said, I dislike the "unnamed parameters" idea as a whole

I begin to dislike it mostly due to different syntax use. ;-)

However had, perhaps there should be another discussion at some upcoming dev meeting. Perhaps not necessarily meant to stop the addition as such, but to consider it for later addition (or more discussion). Also what zverok wrote:

> because I am afraid this feature -- being just "shorter" while staying imperative

I am not sure what you mean with "imperative" here, but ok.

> will stay in the way of adoption of more promising features like &obj.:method, experiments towards proper currying etc.):

I would actually rpefer to not add EITHER numbered parameters and NEITHER your &obj.:method notation. This may be individual preference, of course, but I dislike that we may have to look too closely. I am not sure why you connect these two, though; to me both are totally independent. And even if numbered parameters are not added, I still would prefeer there be no &obj.:method notation altogether nor .: even though I can understand that it enables new possibilities in regards to writing ruby code. You write awesome code; but on the other hand, I find it very hard to read and understand.

IMO it may be best for ruby to stay simpler and reject most of these syntax changes. ;-)

> In your proposal, the last line should be, like

```
to_h { [_1[0], _1[1].to_i] }
```

I actually dislike this even more than the suggestion for "it". :D

Anyway, I do not want to distract too much, so perhaps there can be another dev meeting/discussion. Of course matz decides, but this becomes a bit difficult here if we inter-connect different features and proposals with one another; or different syntax. I am still in the pro-numbered parameters camp in general though, just not as close to the _numbers variants camp. Good syntax is difficult. :P

#9 - 09/24/2019 09:38 PM - Dan0042 (Daniel DeLorme)

proc{ |x,| }.arity == 1, so _1 is consistent with that.

In order to get the tuples' first value you would need to do array_of_arrays.map{_2;_1} because otherwise _1 would mean the entire tuple.

This argument is really weird. Is it really so unsufferable to use _0 instead of _1? Do you really think it would be *better* if the meaning of _1 changed depending on whether _2 is also used? Sorry, but I can't wrap my head around that one. The current syntax is clean and straightforward: use _0 in general and use _1,_2,etc for dereferencing. Maybe _ or __ would have been better than _0, but that's what we've got.

> Try to convince me otherwise.
> If not, in one week I want to apply this change.

Please don't ask people to convince you when it's obvious you've already made up your mind.

#10 - 09/24/2019 09:45 PM - Eregon (Benoit Daloze)

zverok (Victor Shepelev) wrote:

> Honestly, we use it pretty frequently in our codebase and find it appealing.
> It is the shortest and pretty clear (YMMV, of course) way of processing only first elemente of "tuple-alike" arrays, like zips of data or data with indexes.

How frequently compared to just |x|? I would guess < 5% on any non-trivial codebase.
But you are right, frequency of usage isn't so important.

I agree |x,| has its uses, and I think it's perfectly fine to use it when it's intentional.
The explicit trailing comma shows the intention to drop remaining args.

But _1 shouldn't mean "drop everything but the first element if the first parameter is an Array, otherwise return the first parameter".
_1 should mean "the first parameter" just like
_2 means "the second parameter" and
_3 means "the third parameter".

Would anyone contradict that? :p

> **Second**, we have "0 is whole, 1 is first match, etc." conditions in other places of Ruby—string matching:

Indeed, I didn't realize that symmetry.
I don't think it's good to have two different ways with subtle differences to refer to the 1st parameter though.
I believe _1 must be simple to define and similar to _2.
It's all very confusing if _0 is the first parameter and _2 is the second parameter, isn't it?

> **Third**, I believe that in most of real, non-artificial situations, processing of sequence of heterogenous (some unpackable, some not) values with "shortcut blocks args" is a situation to either avoid, or being aware of your data nature.

What about libraries, e.g., taking an Enumerable from the application and using enum.map { transform(_1) }?
Do you think it's OK to have the library work if all elements are number but break horribly if all elements are arrays?
There are certainly a lot of methods that must behave identically (and not drop elements) no matter the class of the parameter passed to it.

Yes, _0 can be used to avoid that, but then it's clear _0 semantics should be preferred in all cases but those were dropping elements is intentional.
And then if it's intentional, we already have a great syntax showing it: |x,|.

So the syntax for the first, second, etc arguments should be consistent (i.e., _1, _2, _3), and by that _1 must not decompose arrays and drop elements, but behave like |x|.
Quod Erat Demonstrandum.

> In your proposal, the last line should be, like

No, my proposal does not change the semantics of { [_1, _2.to_i] }.
That is always the same as { |a,b| [a, b.to_i] }.

#11 - 09/24/2019 10:14 PM - Eregon (Benoit Daloze)

Dan0042 (Daniel DeLorme) wrote:

> proc{ |x,| }.arity == 1, so _1 is consistent with that.

Which sounds like another bug to me, because that block happily accepts more arguments, and should be identical to |x,*|, which proc {|x,*|}.arity # => -2.

What is your point?
My point about arity was that if you add an argument, arity changes, and behavior changes too.
Everyone understands adding an extra argument to a block (or lambda) might change semantics (e.g., [1,[2,3]].map(&-> { _1; _2 }) is ArgumentError), isn't it?
That's already the case today with named parameters.

How do you argue that _2 "takes the second parameter" but _1 "extracts the first element of the first parameter"?
How is that consistent?

> In order to get the tuples' first value you would need to do array_of_arrays.map{_2;_1} because otherwise _1 would mean the entire tuple.

You would need to do array_of_arrays.map { |x,| x }. The tiny extra verbosity is warranted for dropping elements.
Try adding 1 to each element of a 2 dimensional array (a matrix).
_0 must be used currently, but really taking the element as it is (|x|) are the only correct semantics in general if you do not know the specific element types.
Why would the correct semantics in general need a different syntax (_0 and not _1)?

BTW, about typing, how would you type enum.map { _1 } with enum an Enumerable[T]?
Isn't it impossible, because the behavior is inconsistent at runtime?

> This argument is really weird. Is it really so unsufferable to use _0 instead of _1?

Yes, it's inconsistent and I'm pretty sure people would use _1 like |x|, without realizing it's just broken when it's passed an Array.
Do we want frequent bugs based on this instead of just having to use |x,| when wanting to drop elements?

#12 - 09/24/2019 10:18 PM - Eregon (Benoit Daloze)

Here is another inconsistency on current trunk:

```
[1,[2,3]].map { |x| x }  # => [1, [2, 3]]
[1,[2,3]].map { |x,| x } # => [1, 2]
[1,[2,3]].map { _1 }     # => [1, 2]
[1,[2,3]].map(&-> { _1 })  # => [1, [2, 3]]
[1,[2,3]].map(&-> x { x }) # => [1, [2, 3]]
```

So _1 in lambdas is |x|, but it's |x,| in procs?

#13 - 09/24/2019 10:39 PM - Eregon (Benoit Daloze)

It's all about definitions. How do we explain numbered parameters?

Isn't _1 the first parameter, as in x in { |x| } and x in { |x,y| }?
And therefore _2 the second parameter as in y in { |x,y| }?

Of course, { |x| x } and { |x,y| x } already have different semantics, so why should { _2; _1 } not be different than { _1 } ?

I think that makes a lot of sense, is intuitive, and is very easy to explain.

But the current semantics don't match that.
How would we define the current semantics, without being very complex or confusing?

#14 - 09/25/2019 03:27 AM - matz (Yukihiro Matsumoto)

Eregon (Benoit Daloze) [ruby-core:95070] beats me. I am persuaded. I agree with:

- _1 (and no other numbered parameters) to work as |x|.
- giving up _0.

Matz.

#15 - 09/25/2019 04:02 AM - nobu (Nobuyoshi Nakada)

- Status changed from Open to Closed

Applied in changeset git|55e1e22b2d44a8a1eca0f6ed2b11dc0f564f7192.

Changed numbered parameters semantics

- _1 (and no other numbered parameters) to work as |x|.
- giving up _0.

[ruby-core:95074] [Bug #16178]

#16 - 09/25/2019 10:15 AM - Dan0042 (Daniel DeLorme)

> How would we define the current semantics, without being very complex or confusing?

Beautifully simple:
_0 is a single implicit parameter, as in x in { |x| }
_1 is the first numbered parameter, as in x in { |x,y,z,etc| }
_2 is the second numbered parameter, as in y in { |x,y,z,etc| }

How unfortunate that you managed to persuade matz :-(

#17 - 09/25/2019 10:44 AM - sawa (Tsuyoshi Sawada)

Dan0042 (Daniel DeLorme) wrote:

> How unfortunate that you managed to persuade matz :-(

Completely agreed.

#18 - 09/25/2019 11:16 AM - decuplet (Nikita Shilnikov)

Just to make it more polarized, I like the change, it feels more natural.

#19 - 09/25/2019 11:54 AM - ko1 (Koichi Sasada)

> Beautifully simple:
> _0 is a single implicit parameter, as in x in { |x| }
> _1 is the first numbered parameter, as in x in { |x,y,z,etc| }
> _2 is the second numbered parameter, as in y in { |x,y,z,etc| }

I think _0 and _1 are very confusing because people can consider it is sequential meaning. However, the meaning is different.

It is same as $0 and $1, but they are completely different feature (program name and regexp). _0 and _1 is very confusing.

This proposal also introduces inconsistency, but it is better than _0 and _1 idea, I think.

> Of course, having _ as the only unnamed parameter would have |x| semantics,
> but I guess it's too late for that and now we have _ parameters.

Completely agreed (and I like <> for |e| and <n> for _n :p, but rejected this notation).

#20 - 09/25/2019 12:39 PM - Dan0042 (Daniel DeLorme)

> I think _0 and _1 are very confusing because people can consider it is sequential meaning. However, the meaning is different.

I agree, but matz chose _0 for the implicit parameter. I think it's a mistake to then use that naming to change the semantics of implicit vs numbered parameters. The reasoning is backwards.

> Completely agreed (and I like <> for |e| and <n> for _n :p, but rejected this notation).

Or __ for |e| and _n_ for |e,*| ... this is all a bikeshed anyway :D

#21 - 09/25/2019 04:12 PM - Eregon (Benoit Daloze)

Dan0042 (Daniel DeLorme) wrote:

> > How would we define the current semantics, without being very complex or confusing?
>
> Beautifully simple:
> _0 is a single implicit parameter, as in x in { |x| }
> _1 is the first numbered parameter, as in x in { |x,y,z,etc| }

That's incomplete, it's much more tricky than that in the now previous semantics:
_1 is the first numbered parameter, as in x in { |x,y,z,etc| } if there are at least 2 numbered parameters or the first parameter's runtime value is not an Array, otherwise extracts the first argument of the first parameter.

#22 - 09/25/2019 04:13 PM - Eregon (Benoit Daloze)

matz (Yukihiro Matsumoto) wrote:

> Eregon (Benoit Daloze) [ruby-core:95070] beats me. I am persuaded. I agree with:
>
> - _1 (and no other numbered parameters) to work as |x|.
> - giving up _0.

Thank you for your decision.
I think this is going to save many bugs and make numbered parameters significantly simpler.

#23 - 09/26/2019 05:30 PM - alanwu (Alan Wu)

Eregon (Benoit Daloze) wrote:

> Dan0042 (Daniel DeLorme) wrote:
>
> > > How would we define the current semantics, without being very complex or confusing?
> >
> > Beautifully simple:
> > _0 is a single implicit parameter, as in x in { |x| }
> > _1 is the first numbered parameter, as in x in { |x,y,z,etc| }
>
> That's incomplete, it's much more tricky than that in the now previous semantics:
> _1 is the first numbered parameter, as in x in { |x,y,z,etc| } if there are at least 2 numbered parameters or the first parameter's runtime value is not an Array, otherwise extracts the first argument of the first parameter.

That's still incomplete, since it doesn't explain why _1 doesn't decompose when used in lambdas. If it always decomposes, it'd be more self consistent.
Anyways, I like the new rule better, since it has less corner cases.