Ruby master - Feature #16460

External names for keyword parameters in method definitions

12/28/2019 02:28 AM - harrisonb (Harrison Bachrach)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Hello! This is my first time filing an issue and I was unable to find anything similar. I apologize if one already exists.

In other languages (such as JavaScript, Swift, and Crystal), it is possible to have two names for a given keyword argument: one that is used in method invocation, and one used in the method definition. Here is an example from Crystal (which has syntax very similar to Ruby):

```
def increment(value, by amount)
  value + amount
end

increment(value: 5, by: 10)
```

This helps create more readable method invocations and definitions. It would be especially helpful in Ruby as the language lacks a destructuring syntax for hashes/keyword args. This is unlike JavaScript, where you can do something like:

```
const { nameOfOneProperty: newNameForTheProperty, nameOfAnotherProperty:
newNameForTheOtherProperty } = foo;
```

where foo is a JavaScript Object that has the properties nameOfOneProperty or nameOfAnotherProperty (If it did not have either of them, the corresponding identifiers (newNameForTheProperty and newNameForTheOtherProperty would be initialized to undefined).

I'm thinking that such a change would pair nicely with the new 3.0 keyword argument changes.

Others have suggested that this could also be helpful if the keyword params collide with reserved keywords in Ruby, e.g.:

```
def reserve_appointment(when:)
  Appointment.create(time: when) #=> SyntaxError: unexpected `when', expecting `end'
end
```

Currently, one must use local_variable_get to get around this issue, e.g.:

```
def reserve_appointment(when:)
  time = local_variable_get(:when)
  Appointment.create(time: time)
end
```

Syntax options:

1. No arrow syntax (original proposal)

   ```
   def name(external_name internal_name: default_value)
   # ...
   end
   # Example
   def move(from source: 'src/', to destination: 'dist/', at time:)
   # ...
   end
   ```

2. Infix arrow syntax

   ```
   def name(external_name => internal_name: default_value)
   # ...
   ```

```
end
# Example
def move(from => source: 'src/', to => destination: 'dist/', at => time:)
# ...
end
```

3. Postfix arrow syntax (suggested by zverok (Victor Shepelev))

```
def name(external_name: default_value => internal_name)
# ...
end
# Example
def move(from: 'src/' => source, to: 'dist/' => destination, at: => time)
# ...
end
```

History

#1 - 12/28/2019 03:34 AM - sawa (Tsuyoshi Sawada)

What is foo?

#2 - 12/28/2019 07:31 AM - shevegen (Robert A. Heiler)

> A suggested syntax might be

```
def name(external_name internal_name:)
```

Isn't this syntax already reserved/used for keyword arguments?

> Here is an example from Crystal (which has syntax very similar to Ruby)

matz knows crystal; he even donated a seizable sum to crystal way before that donation-webpage + "hall of fame" for crystal existed. :-)

Many ruby folks also use crystal; in many ways crystal is like the closest "brother" language to ruby. Though it is not quite ruby and neither is ruby crystal; IMO the type system is indeed the biggest difference and makes for another language "feel" (but that is my opinion; I still think it's great that crystal exists; I even once suggested to bridge the gap between ruby code and crystal code, so we could write ruby, and have crystal code autogenerated from that. ;-))

> It would be especially helpful in Ruby as the language lacks a destructuring syntax for hashes/keyword args.

I think pattern matching allows for destructuring? Not sure if I am right here, or whether I have missed something, but the examples I could see were a lot about destructuring hashes, like:

```
case JSON.parse(json, symbolize_names: true)
in {name: "Alice", children: [{name: "Bob", age: age}]}
  p age #=> 2
```

But it may be that I misunderstood you, or you had another intention/goal.

By the way, different languages use different syntaxes/patterns; it's not often easy or possible or wanted to translate 1:1. But I don't want to sound discouraging either - just be prepared to update/modify the suggestion. What helps the most is to focus on clear use cases; the ruby core team often recommends to have a clear use described case. Don't worry that this is your first suggestion, everyone has to start at some point in time to contribute to (if wanted). :-)

#3 - 12/28/2019 07:59 AM - nobu (Nobuyoshi Nakada)

Though I don’t remember exactly, once I proposed a similar syntax.
And it was rejected then local_variable_get was introduced instead.

#4 - 12/28/2019 08:35 AM - harrisonb (Harrison Bachrach)

> What is foo?

In this case, foo would be a (JavaScript) Object. In using this syntax, you are presuming that foo would have the properties nameOfOneProperty or nameOfAnotherProperty, though if it did not have either of those specific properties, the corresponding identifiers (newNameForTheProperty and newNameForTheOtherProperty would be initialized to undefined)

> Isn't this syntax already reserved/used for keyword arguments?

Yes, this would be an extension of that syntax, though perhaps I'm not understanding you.

> matz knows crystal; he even donated a seizable sum to crystal way before that
> donation-webpage + "hall of fame" for crystal existed. :-)

Yes sorry, I wasn't trying to imply ignorance--I'm sure many Ruby contributors are familiar.

> I think pattern matching allows for destructuring?

This is my mistake. I had not kept up with what made it in to pattern matching. This somewhat reduces the need for the feature suggestion, but I still think this would be an improvement in terms of expressiveness in this case:

```
# In all examples, external API is `move(from: 'foo/bar/biz', to: 'baz/boz/buzz')`

# 2.0+(?) with just normal assignment
def move(from:, to:)
  source = from
  destination = to
  # ...
end

# 2.7+ with pattern matching syntax
def move(from:, to:)

# Must construct a Hash literal in order to make use of both pattern matching syntax benefits of keyword param
s
  case { from: from, to: to }
  in { from: source, to: destination }
    # ...
  end
end

# Note that in both of the above examples, `from` and `to` are also valid (duplicate) identifiers

# Proposed syntax
def move(from source:, to destination:)
  # ...
end
```

I feel like there are many examples (and I can come up with more) as this can be useful whenever the following circumstances occur:

1. The method name is a verb
2. The arguments may be disambiguated with keyword params that are prepositions

It's possible there are other cases where these are useful, but even just this above category of cases is large.

> By the way, different languages use different syntaxes/patterns; it's not often
> easy or possible or wanted to translate 1:1.

I definitely agree in general. I pick this piece as it seems fairly agnostic to the distinctions between Crystal & Ruby (type-system, compilation, etc.)

> But I don't want to sound discouraging either - just be prepared to update/modify the suggestion. What
> helps the most is to focus on clear use cases; the ruby core team often
> recommends to have a clear use described case.

Happy to modify/expand!

> Don't worry that this is your
> first suggestion, everyone has to start at some point in time to contribute
> to (if wanted). :-)

Thank you for the encouragement :)

> Though I don't remember exactly, once I proposed a similar syntax.
> And it was rejected then local_variable_get was introduced instead.

Do you know where I might find that conversation? I'm not sure if I understand how local_variable_get would help in this case. I did a brief search for local_variable_get in the issue tracker but the only results seem to be relevant to resolving keyword args that collide with resolved keywords in Ruby.

#5 - 12/28/2019 08:44 AM - zverok (Victor Shepelev)

Jut two points to add:

- the feature like this might be useful for arguments named as Ruby keywords/core methods (e.g. things like run :some_task, if: :condition?, convert(value, raise: false), schedule(:worker, in: 5.minutes) etc.), as nobu (Nobuyoshi Nakada) mentions, currently the only way to get those variables in method body is local_variable_get (and with raise example, even it will not help, the problem is trickier)
- the name alt_name: syntax look completely "alien" to me (in context of "what exists in Ruby"), I'd say, considering new pattern-matching examples, something with => might work (while still looking ugly):

```
def run(task, if: => condition)
  p condition
end

# with default value, looks a bit less cringy:
def convert(value, raise: false => should_raise)
  if should_raise
    # ...
end
```

#6 - 12/28/2019 08:49 AM - harrisonb (Harrison Bachrach)

zverok (Victor Shepelev) wrote:

> I'd say, considering new pattern-matching examples, something with => might work (while still looking ugly) [...]

See, to me, this seems *more* confusing as it reverses the common meaning of => in Ruby of key => value.

#7 - 12/28/2019 08:59 AM - zverok (Victor Shepelev)

> it reverses the common meaning of => in Ruby of key => value.

But what is "common" meaning? key => value just means "key corresponds to value" (which you also can read "parameter name corresponds to (=>) local variable").

It also means exactly "...and put in this variable" in this cases:

- rescue Exception => e (always)
- key: pattern => variable (2.7's pattern matching)

And generally, one may theorize that => foo can be read like "put someting into foo".

With foo bar: syntax there are two problems:

- It is unlike anything that exists in Ruby (I can't remember a thing where two names separated by space would be a standalone syntax and not just a shortcut, say puts x === puts(x))
- For me, it is totally un-mnemonic: is it external_name internal_name: or internal_name external_name:? How one should remember? How many time one will confuse it and swear "why it does not works like tutorial says???"

At least foo: => bar can be taught as "name some thing foo: (its key is foo:) and put it into (=>) bar"

#8 - 12/28/2019 09:11 AM - harrisonb (Harrison Bachrach)

zverok (Victor Shepelev) wrote:

It also means exactly "...and put in this variable" in this cases:

- rescue Exception => e (always)
- key: pattern => variable (2.7's pattern matching)

I stand corrected! That is a compelling refutation of my point.

- It is unlike anything that exists in Ruby (I can't remember a thing where two names separated by space would be a standalone syntax and not just a shortcut, say puts x === puts(x))

I suppose this is a thoroughly different syntax (this doesn't really bother me, but perhaps it should).

- For me, it is totally un-mnemonic: is it external_name internal_name: or internal_name external_name:? How one should remember? How many time one will confuse it and swear "why it does not works like tutorial says???"

At least foo: => bar can be taught as "name some thing foo: (its key is foo:) and put it into (=>) bar"

The correct order makes grammatical sense for the common use-cases of preposition non-preposition:

- to destination: vs. destination to:
- with klass: vs. klass with:
- within range: vs. range within:

A sort of middleground might be something like this:

def move(from => source:, to => destination:)

#9 - 12/28/2019 01:36 PM - zverok (Victor Shepelev)

> It is unlike anything that exists in Ruby (I can't remember a thing where two names separated by space would be a standalone syntax and not just a shortcut, say puts x === puts(x))
>
> I suppose this is a thoroughly different syntax (this doesn't really bother me, but perhaps it should).

Exactly my point. The Rubyists eye is trained to read <identifier><space><something> only two ways:

1. <known keyword of a small list> → <some statement depending on the keyword>
2. <method>(<something is argument to it>)

You propose to introduce the "same" construct, which means something completely different, but due to our intutions, **in Ruby** external_name internal_name: reads as external_name(internal_name:), which is misleading.

> The correct order makes grammatical sense for the common use-cases of preposition non-preposition:
>
> to destination: vs. destination to:
> with klass: vs. klass with:
> within range: vs. range within:

1. So, it should be read this way: (within range):, where both words are related to colon, with one being "title" and the second "explanation"? This, again unlike anything else in Ruby. It slightly reminds me of some documentation system links (yard, probably? with "Look also [FormalClassName that class]", or something like that... and I honestly never can remember which is which -- which is "link" and which is "title").
2. It works well for some examples you constructed, but there always can be counter-examples, like: I want to use when: argument, renaming it to time (becase when is a keyword), so... is it time when: or when time:?..
3. It is incredibly confusing for any parser (especially "rename as it is keyword" use-case) and syntax highlighter:def foo(in time: would be the only case where standalone in<space><something> should be parsed/hihglighted differently.

> A sort of middleground might be something like this:
>
> def move(from => source:, to => destination:)

This is also bad (by similar reasoning as above).

Some additional point: currently keyword args definitions reads this way: method(arg1: ..., arg2: ...) -- is how it would be called (e.g. in call-site we'll see the same structure method(arg1: ..., arg2: ...) -- and everything after the : is somewhat "how it is implemented" (for example, defaults calculation), which => put_to_this_variable follows.

And one more consideration: imagine Ruby introduced one of those syntaxes in 2.8. And some rubyist who missed the announcement, comes to a "new" codebase, and how they would understand it?

```
def foo(in: => time)
```

"...Ugh, what is it?.. Putting something into time? Ah, in is a keyword, they want to rename it. Got it. Hate it, but got it."

```
def foo(in time: )
```

"...Ugh what?.. What?.. Is it a new keyword?.. Some kind of type hinting?.. No idea..."

(Pure speculations, obv.)

#10 - 12/28/2019 02:45 PM - sawa (Tsuyoshi Sawada)

- Description updated

#11 - 12/29/2019 06:15 PM - harrisonb (Harrison Bachrach)

zverok (Victor Shepelev) wrote:

> You propose to introduce the "same" construct, which means something completely different, but due to our intutions, **in Ruby** external_name internal_name: reads as external_name(internal_name:), which is misleading.

I think at the core, I am suggesting introducing something new. While there is certainly merit to discussion of what is the most "Ruby-like" way to implement that, I don't think it should hamstring the entire conversation surrounding the proposal. The new case...in feature addition in 2.7 is a fairly radical departure from the previous semantics of Ruby:

- It performs a name binding outside of all the usual places (assignment, parameter list, block parameter list, rescue block, etc.)
- That binding is set up by a pattern (which often looks very much like an array/hash/etc. literal) which would previously never preform a binding

While the pattern matching feature requires one to read up about it, so do other later added features that provide utility and expressiveness to Ruby developers.

> 1. So, it should be read this way: (within range):, where both words are related to colon, with one being "title" and the second "explanation"? This, again unlike anything else in Ruby. It slightly reminds me of some documentation system links (yard, probably? with "Look also [FormalClassName that class]", or something like that... and I honestly never can remember which is which -- which is "link" and which is "title").

The main constant is that the identifier with the colon (range in the example above) remains the one that can be referenced in the body of the method.

I am open to the arrow syntax if we conclude that it is still unclear and/or sufficiently alien.

> 1. It works well for some examples you constructed, but there always can be counter-examples, like: I want to use when: argument, renaming it to time (becase when is a keyword), so... is it time when: or when time:?..

I would point to the same fact above about the identifier with the colon remaining the only valid one in the method body. Additionally, this is change for the sake of improving readability. One can probably come up with confusing examples, but the proposed syntax, like much of Ruby syntax, is a sharp knife that one can use to make their code more or less readable.

> 1. It is incredibly confusing for any parser (especially "rename as it is keyword" use-case) and syntax highlighter:def foo(in time: would be the only case where standalone in<space><something> should be parsed/hihglighted differently.

I think this could be a valid criticism of the arrow-less syntax. However, parsers for at least two other languages accomplish this without issue, so I don't think it's an insurmountable issue.

> Some additional point: currently keyword args definitions reads this way: method(arg1: ..., arg2: ...) -- is how it would be called (e.g. in call-site we'll see the same structure method(arg1: ..., arg2: ...) -- and everything after the : is somewhat "how it is implemented" (for example, defaults calculation), which => put_to_this_variable follows.

I think this is a very good point and perhaps the most compelling one for the external_name: default_value => internal_name syntax.

> And one more consideration: imagine Ruby introduced one of those syntaxes in 2.8. And some rubyist who missed the announcement, comes to a "new" codebase, and how they would understand it?
>
> `def foo(in: => time)`
>
> "...Ugh, what is it?.. Putting something into time? Ah, in is a keyword, they want to rename it. Got it. Hate it, but got it."
>
> `def foo(in time: )`
>
> "...Ugh what?.. What?.. Is it a new keyword?.. Some kind of type hinting?.. No idea..."
>
> (Pure speculations, obv.)

I am going to assume good faith here in some of the phrasing. With any syntax introduction, there is going to have to be some learning. The case...in syntax also requires a similar (if not greater) amount of learning, but that was also deemed acceptable.

#12 - 12/29/2019 10:59 PM - harrisonb (Harrison Bachrach)

- Description updated

#13 - 12/30/2019 12:07 PM - zverok (Victor Shepelev)

harrisonb (Harrison Bachrach) I don't feel like I can add anything substantial to what I've already said. Just two clarifications:

1. If something in my comments has sounded angrily/mockingly/disrespectful, I am genuinely sorry. It was not my intention to mock you or your ideas, so "Ugh what"s and similar stuff was only intended as a comical demonstration how I am trying to picture the imaginary programmer reading the code (to better showcase my understanding of the problem, not to show how yours are "dumb" or something!)
2. The (almost) only thing I was trying to say is: new features and ideas are necessary, but (it is my feeling, at least) but the core of any proposal is how it will play with the rest of the syntax, previous intuitions and habits (for ex., the whole pattern matching thing was designed with introduction of only one keyword, reusing the ways of structuring code that exist in other places in Ruby and "feel normal" to Ruby devs)

#14 - 12/30/2019 04:44 PM - Dan0042 (Daniel DeLorme)

It's a nice idea, and I rather like the original syntax proposed for its high readability, even though it would require adjusting our expectations. It's very different from usual ruby but since it's in the very limited context of a parameter list that may be acceptable.

But it's worth pointing out that Matz has been rather conservative when it comes to method parameters. A syntax like foo(@x) that would allow direct assignment to instance variables has already been rejected.

#15 - 01/02/2020 02:11 AM - nobu (Nobuyoshi Nakada)

harrisonb (Harrison Bachrach) wrote:

> > Though I don’t remember exactly, once I proposed a similar syntax.
> > And it was rejected then local_variable_get was introduced instead.
>
> Do you know where I might find that conversation? I'm not sure if I understand how local_variable_get would help in this case. I did a brief search for local_variable_get in the issue tracker but the only results seem to be relevant to resolving keyword args that collide with resolved keywords in Ruby.

The problem is described in a_matsuda's slide.
Then discussed as “alias of keyword argument” at Developers meeting at 20130727.

This is a simple example.