Ruby trunk - Feature #5474

keyword argument

10/23/2011 09:53 PM - mame (Yusuke Endoh)

Status:	Closed	
Priority:	Normal	
Assignee:	mame (Yusuke Endoh)	
Target version:	2.0.0	

Description

Hello,

I'm sending a patch for keyword arguments.

(This feature had been discussed in #5454, but I'm re-creating
a new ticket because the old ticket was resigtered in ruby-dev)

Matz himself proposed this feature. It is also basically
promised to include the feature in 2.0. [ruby-core:39837]
I'm planning to commit the patch after it is reviewed by koichi.

But the detail of the spec is not fixed yet, and may be changed
drastically.
We would like to hear your comments and suggestions, especially,
with a use case and/or an actual experience.

The background of this proposal is that, in the recent Ruby,
the last argument (as a Hash) is often used to pass optional
information. This feature is intended to aid the style.

Look an example:

```
def create_point(x, y, color: "white", size: 1)
  # keyword arguments  ^^^^^^^^^^^^^^^^^^^^^^^ here!

  p [x, y, color, size]
end

create_point(2, 3, color: "red")
  #=> [2, 3, "red", 1]
```

The caller size is a traditional hash argument notation.
This feature is Hash parsing in the callee side.

(So it is more suitable to call it "keyword parameter."
But I use "keyword argument" because everyone calls so.)

We can implement the similar behavior in pure Ruby. However,
this feature is easier to read/write, and richer in the some
aspects:

- it raises an TypeError when unknown keyword is given

  create_point(2, 3, style: "solid")
  #=> unknown keyword (TypeError)

- you can use ** argument to suppress the TypeError and/or
  to get the given hash itself:

  def create_point(x, y, color: "white", size: 1, **h)
  p [x, y, color, size, h]
  end
  create_point(2, 3, style: "solid")
  #=> [2, 3, "red", 1, {:style=>"solid"}]

- it is easily used even when there is a rest argument

  ```
  def create_point(x, y, *r, color: "solid", size: 1)
  ...
  end
  ```

(a complex and non-essential code is required to
implement the same behavior in pure Ruby)

- there is room for optimizing the speed (though I have not done any optimization yet)

An alternative design is to treat all parameters as keyword
arguments (as Evan said in [ruby-core:40195]).

```
def create_point(x, y, color = "white", size = 1)
p [x, y, color, size]
end
create_point(color: "red", x: 2, y: 3)
#=> [2, 3, "red", 1]
```

Actually I also like this, but I'm afraid if it is too flexible
and seems difficult to implement and optimize.

Thanks,

--
Yusuke Endoh mame@tsg.ne.jp

Related issues:

Related to Ruby trunk - Bug #6086: Number of arguments and named parameters	**Rejected**	**02/25/2012**
Related to Ruby trunk - Feature #6414: Destructuring Assignment	**Feedback**	

History

#1 - 10/24/2011 09:53 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40290] [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Sun, 23 Oct 2011 21:53:58 +0900, Yusuke Endoh mame@tsg.ne.jp writes:

```
|- it raises an TypeError when unknown keyword is given
|
|    create_point(2, 3, style: "solid")
|      #=> unknown keyword (TypeError)
```

I think it should be ArgumentError instead of TypeError.

```
|An alternative design is to treat all parameters as keyword
|arguments (as Evan said in [ruby-core:40195]).
|
|  def create_point(x, y, color = "white", size = 1)
|    p [x, y, color, size]
|  end
|  create_point(color: "red", x: 2, y: 3)
|    #=> [2, 3, "red", 1]
```

It's Python way, and I won't take it.

```
                                matz.
```

#2 - 10/24/2011 02:53 PM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40299] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Mon, 24 Oct 2011 14:12:21 +0900, Evan Phoenix evan@phx.io writes:

```
|> | def create_point(x, y, color = "white", size = 1)
|> | p [x, y, color, size]
|> | end
```

|> | create_point(color: "red", x: 2, y: 3)
|> | #=> [2, 3, "red", 1]
|>
|> It's Python way, and I won't take it.
|What don't you like about this approach? I'd like to know so that hopefully I can formulate an alternative you would like.
|
|My worry about Yusuke's current proposal is that it requires a Hash be allocated on the caller side to use the feature, which makes the usage of keyword arguments much more heavyweight than normal arguments. This in turn means people will either shy away from them or use them and complain that they're too slow (which could make ruby look bad).
|
| - Evan

It could make argument parsing much heavyweight and difficult to
predict, both user POV and implementation POV. The Python argument
rules are very complex comparing to other parts of the language.

Besides that, as Yusuke mentioned, I think the current proposal still
have much room to optimize, e.g. deferring hash allocation until
really needed.

```
                              matz.
```

#3 - 10/25/2011 11:23 PM - headius (Charles Nutter)

On Mon, Oct 24, 2011 at 12:12 AM, Evan Phoenix evan@phx.io wrote:

> |An alternative design is to treat all parameters as keyword
> |arguments (as Evan said in [ruby-core:40195]).
> |
> | def create_point(x, y, color = "white", size = 1)
> | p [x, y, color, size]
> | end
> | create_point(color: "red", x: 2, y: 3)
> | #=> [2, 3, "red", 1]
>
> It's Python way, and I won't take it.
> What don't you like about this approach? I'd like to know so that hopefully I can formulate an alternative you would like.

The overhead concerns may not be valid. I think it could be
implemented such that the overhead would only be there if called with
a keyword parameter form. Otherwise, all arguments are treated
positionally.

Quick pseudo-algorithm:

call_args = #
if args.kind_of? Hash
newargs = []
position_map = method.keyword_to_position
call_args.each do |key, value|
case key
when String
newargs[position_map[key]] = value
when Fixnum
newargs[key] = value
end
end
call_args = newargs
end
method.call_with_args call_args

This would be detectable at compile time; only methods that have
keyword args would do the additional logic of mapping names to
positions.

However, this way of optimizing it does require keyword args always
come after regular positional args. I think that's not too big a leap
to make, since they have to be at the end right now. It does not
require they be specified by the caller in the same order as the
target method, as in MacRuby.

There is a problem with this proposal, though: it could easily break
current code that uses "hash args". For example, a legacy case:

def foo(who, hash)
...
end

foo('hello', who: 'world')

This example is slightly contrived, but under current Rubies the "who"
variable in the "foo" method would get 'hello', and under Evan's
proposal it would be 'world'. For this reason I think explicitly
notating keyword parameters in the argument list is better.

> My worry about Yusuke's current proposal is that it requires a Hash be allocated on the caller side to use the feature, which makes the usage of keyword arguments much more heavyweight than normal arguments. This in turn means people will either shy away from them or use them and complain that they're too slow (which could make ruby look bad).

I think the cost of constructing a Hash in Rubinius may be coloring
your thoughts here...and I don't blame you; even though Hash
construction in JRuby is pretty fast, it's not free:

https://gist.github.com/1312815

However, I think much of the Hash-borne overhead could be blunted by
having keyword arg hashes be frozen and list-based. Most of the time
there's no more than a handful of keyword args in use, so having them
be "Hash-like" but backed by a simple associative array would make
them considerably cheaper to construct in all implementations:

https://gist.github.com/a07c93c80dfdea023253

In any case, I don't think there's any reason Yusuke's version would
require they be a hash unless the target method *needs* them to be a
hash. More pseudocode:

AT CALL SITE:

call_args = # ...
if call_args.kind_of? Hash
# map to positional args internally
end
...

IN METHOD PREAMBLE:

if self.keyword_args?
if self.keyword_rest?
# unpack positional keyword args with "rest" hash
else
# unpack (or not) positional to keyword offsets
end
end

You'd only pay for the hash if you want it.

- Charlie

#4 - 10/26/2011 08:23 AM - Anonymous

See below.

--
Evan Phoenix // evan@phx.io

> > My worry about Yusuke's current proposal is that it requires a Hash be allocated on the caller side to use the feature, which makes the usage of keyword arguments much more heavyweight than normal arguments. This in turn means people will either shy away from them or use them and complain that they're too slow (which could make ruby look bad).
>
> I think the cost of constructing a Hash in Rubinius may be coloring
> your thoughts here...and I don't blame you;
> It might be, but when you compare calling a method with normal arguments to creating a Hash, it doesn't matter what runtime your in, the difference is huge.
>
> https://gist.github.com/a07c93c80dfdea023253

> In any case, I don't think there's any reason Yusuke's version would
> *require* they be a hash unless the target method *needs* them to be a
> hash.
> Ok, good. This is actually all that I ask. As it's being designed, take into account a way to implement it without having to allocate a full every time.
> If this consideration is given in the design phase, I'm sure we'll all make it work efficiently.

- Evan

#5 - 10/27/2011 07:23 AM - jballanc (Joshua Ballanco)

On Wed, Oct 26, 2011 at 2:08 PM, Yukihiro Matsumoto matz@ruby-lang.orgwrote:

> Hi,
>
> In message "Re: [ruby-core:40414] Re: [ruby-trunk - Feature
> #5474][Assigned] keyword argument"
> on Thu, 27 Oct 2011 02:51:51 +0900, Charles Oliver Nutter <
> headius@headius.com> writes:
>
> |Evan and I would like to see support for non-default keyword args. So
> |the following form would require that "size" be provided as a keyword
> |arg, and otherwise raise an ArgumentError:
> |
> |def create_point(x, y, color: "white", size:)
>
> I don't know the reason why you need this value-less keyword
> argument. If you want the default value, you just need to omit the
> keyword altogether.

I think the point is that not every argument necessarily has a reasonable
default. Imagine, for example, if we were to rewrite File.open to take
keyword arguments:

```
class File
  def self.open(filename:, mode: 'r')
  ...
```

It would be desirable to call such a method as File.open(filename:
"foo.txt") but omitting the "filename:" argument should be an error.
Perhaps, if a hash-like syntax is desired, we could introduce a "required"
keyword:

```
class File
  def self.open(filename: required, mode: 'r')
  ...
```

such that omitting the option raises an ArgumentError?

#6 - 10/27/2011 07:53 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40414] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Thu, 27 Oct 2011 02:51:51 +0900, Charles Oliver Nutter headius@headius.com writes:

|Evan and I would like to see support for non-default keyword args. So
|the following form would require that "size" be provided as a keyword
|arg, and otherwise raise an ArgumentError:
|
|def create_point(x, y, color: "white", size:)

I don't know the reason why you need this value-less keyword
argument. If you want the default value, you just need to omit the
keyword altogether.

Besides that, what is the value corresponding to size: keyword above?
Even though keyword argument implementation can bypass the hash
allocation as optimization, we still have to keep hash semantics, I think.

```
                        matz.
```

#7 - 10/27/2011 09:53 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40418] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Thu, 27 Oct 2011 04:03:52 +0900, Joshua Ballanco jballanc@gmail.com writes:

|I think the point is that not every argument necessarily has a reasonable
|default. Imagine, for example, if we were to rewrite File.open to take
|keyword arguments:
|
| class File
| def self.open(filename:, mode: 'r')
| ...

Yes, there could be a language with that kind of design. But Ruby
will not have mandatory keyword arguments. It is the design choice I
made already. Every keyword argument will be optional in Ruby.

```
                                matz.
```

#8 - 10/27/2011 10:23 AM - jballanc (Joshua Ballanco)

On Wed, Oct 26, 2011 at 8:30 PM, Yukihiro Matsumoto matz@ruby-lang.orgwrote:

> Hi,
>
> In message "Re: [ruby-core:40418] Re: [ruby-trunk - Feature
> #5474][Assigned] keyword argument"
> on Thu, 27 Oct 2011 04:03:52 +0900, Joshua Ballanco <
> jballanc@gmail.com> writes:
>
> |I think the point is that not every argument necessarily has a reasonable
> |default. Imagine, for example, if we were to rewrite File.open to take
> |keyword arguments:
> |
> | class File
> | def self.open(filename:, mode: 'r')
> | ...
>
> Yes, there could be a language with that kind of design. But Ruby
> will not have mandatory keyword arguments. It is the design choice I
> made already. Every keyword argument will be optional in Ruby.

Thank you for the clarification. I am slowly understanding the direction and
motivation behind the keyword arguments feature.

#9 - 10/27/2011 11:23 AM - headius (Charles Nutter)

On Wed, Oct 26, 2011 at 7:30 PM, Yukihiro Matsumoto matz@ruby-lang.org wrote:

> Yes, there could be a language with that kind of design. But Ruby
> will not have mandatory keyword arguments. It is the design choice I
> made already. Every keyword argument will be optional in Ruby.

Can you clarify why you've already made this decision? For normal
positional arguments, there are required args, optional args, and rest
args. It seems like having the same types of arguments for keyword
args would be more consistent.

Also, Evan pointed out to me that if Ruby doesn't support mandatory
keyword arguments, you're going to see a *lot* of this pattern:

def foo(a: nil)
raise ArgumentError, 'must pass value for a' unless a
..
end

So in essence, by making all keyword arguments optional, you're
forcing users to do their own required argument checks. Is that going
to make programmers happy?

- Charlie

#10 - 10/27/2011 06:29 PM - nobu (Nobuyoshi Nakada)

Hi,

(11/10/27 11:02), Charles Oliver Nutter wrote:

> Also, Evan pointed out to me that if Ruby doesn't support mandatory
> keyword arguments, you're going to see a *lot* of this pattern:
>
> def foo(a: nil)
> raise ArgumentError, 'must pass value for a' unless a
> ..
> end

It should be a positinal argument. I can't get why it must be a
keyword argument.

> So in essence, by making all keyword arguments optional, you're
> forcing users to do their own required argument checks. Is that going
> to make programmers happy?

An alternative:
def foo(a: raise(ArgumentError))
end

--
Nobu Nakada

#11 - 10/27/2011 06:59 PM - headius (Charles Nutter)

On Thu, Oct 27, 2011 at 3:16 AM, Nobuyoshi Nakada nobu@ruby-lang.org wrote:

> (11/10/27 11:02), Charles Oliver Nutter wrote:
>
>> def foo(a: nil)
>>   raise ArgumentError, 'must pass value for a' unless a
>>   ..
>> end
>
> It should be a positinal argument. I can't get why it must be a
> keyword argument.

It's a contrived example, of course, but I think having a neat
parallel between positional args (required, optional, rest) and
keyword args (required, optional, rest) would serve users well.

Honestly, I'm not one to argue for increased complexity in argument
processing...the 1.9 support for array destructuring still gives me
nightmares. But in this case, it seems like having keyword args mirror
positional args is a good idea.

>> So in essence, by making all keyword arguments optional, you're
>> forcing users to do their own required argument checks. Is that going
>> to make programmers happy?
>
> An alternative:
> def foo(a: raise(ArgumentError))
> end

You are an evil, evil man. :)

- Charlie

#12 - 10/27/2011 11:23 PM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40449] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Thu, 27 Oct 2011 18:56:51 +0900, Charles Oliver Nutter headius@headius.com writes:

|It's a contrived example, of course, but I think having a neat

|parallel between positional args (required, optional, rest) and
|keyword args (required, optional, rest) would serve users well.

I disagree here. Unlike other languages, keyword arguments will be
labels to optional arguments in Ruby. This decision comes after the
deep consideration for long time. I don't think I am going to change
my mind here.

|Honestly, I'm not one to argue for increased complexity in argument
|processing...the 1.9 support for array destructuring still gives me
|nightmares. But in this case, it seems like having keyword args mirror
|positional args is a good idea.

How contradicting humans are.

```
                                                matz.
```

#13 - 10/28/2011 12:23 AM - mame (Yusuke Endoh)

Hello,

2011/10/27 Yukihiro Matsumoto matz@ruby-lang.org:

> Yes, there could be a language with that kind of design. But Ruby
> will not have mandatory keyword arguments. It is the design choice I
> made already. Every keyword argument will be optional in Ruby.

I like the design.

I hate to write File.open(filename: "foo") . Just redundant.
I don't want such a style to be popularized.

BTW, a current optional parameter will be deprecated?
I don't like the feature because it is not extensible.
Actually, I hate to see ERB.new(src, nil, "%") anymore!
It now just complicates the language.

Of course I don't think it will be removed in 2.0.
Let it be just deprecated now, and remove it in further future,
such as 3.0. What do you think?

--
Yusuke Endoh mame@tsg.ne.jp

#14 - 10/28/2011 12:53 AM - now (Nikolai Weibull)

On Thu, Oct 27, 2011 at 17:16, Yusuke Endoh mame@tsg.ne.jp wrote:

> 2011/10/27 Yukihiro Matsumoto matz@ruby-lang.org:
>
>> Yes, there could be a language with that kind of design. But Ruby
>> will not have mandatory keyword arguments. It is the design choice I
>> made already. Every keyword argument will be optional in Ruby.
>
> BTW, a current optional parameter will be deprecated?
> I don't like the feature because it is not extensible.
> Actually, I hate to see ERB.new(src, nil, "%") anymore!
> It now just complicates the language.

Agree! This would clean up both the language and the implementation
and make for better APIs.

#15 - 10/28/2011 12:59 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:40454] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Fri, 28 Oct 2011 00:16:33 +0900, Yusuke Endoh mame@tsg.ne.jp writes:

|BTW, a current optional parameter will be deprecated?
|I don't like the feature because it is not extensible.
|Actually, I hate to see ERB.new(src, nil, "%") anymore!

|It now just complicates the language.
|
|Of course I don't think it will be removed in 2.0.
|Let it be just deprecated now, and remove it in further future,
|such as 3.0. What do you think?

Removing something, especially well-used feature like (non-labeled)
optional arguments is a big thing. We have to consider it very
carefully. I'd say there's possibility, for now.

```
                            matz.
```

#16 - 10/28/2011 01:53 AM - nobu (Nobuyoshi Nakada)

Hi,

(11/10/27 18:56), Charles Oliver Nutter wrote:

> On Thu, Oct 27, 2011 at 3:16 AM, Nobuyoshi Nakada nobu@ruby-lang.org wrote:
>
>> An alternative:
>> def foo(a: raise(ArgumentError))
>> end
>
> You are an evil, evil man. :)

You didn't know? ;)

Anyway, isn't it better to show which default value was evaluated in
the backtrace?

--
Nobu Nakada

#17 - 10/28/2011 02:59 AM - headius (Charles Nutter)

On Thu, Oct 27, 2011 at 11:42 AM, Nobuyoshi Nakada nobu@ruby-lang.org wrote:

> You didn't know? ;)
>
> Anyway, isn't it better to show which default value was evaluated in
> the backtrace?

Yes, it definitely is. I would like to see Ruby do that for me, but
since it sounds like it won't...

this seems too long, and that's with a very short message:

def foo(a: raise(ArgumentError, 'missing a')))

I think this is better but ugly:

def foo(a: nil)
raise ArgumentError, 'missing a' if a.nil?

The other problem with the second version is that nil might be a valid
value to pass in for a. So if Ruby doesn't enforce required keyword
args, you actually have to use this pattern:

def foo(a: (no_a = true; nil))
raise ArgumentError, 'missing a' if no_a

Bleah.

- Charlie

#18 - 10/30/2011 08:23 AM - Eregon (Benoit Daloze)

Hi,

On 23 October 2011 14:53, Yusuke Endoh mame@tsg.ne.jp wrote:

> Feature #5474: keyword argument

It sounds great!

I agree mandatory keyword arguments should be positional arguments and
all parameters should not be treated as keyword arguments.

I have a few questions/remarks:
1) What is the way to pass keyword arguments ?
I would guess **h like:

```
def meth(a, **h)
other(a, **h)
end # => syntax error
```

BTW, using **h in the argument list does not seems to work in some cases for me:

```
def a(**h)
end # => syntax error, unexpected tPOW

def m(k: nil, **h, &block)
end
m() # => undefined method `key?' for nil:NilClass
```

2) I'm a bit dubious about the **h syntax to get (and I guess to pass) a Hash though.
I know it's the way it's done in Python, but I don't like it (esthetically),
especially when it is used to pass the Hash:

```
def meth(a, *r, **h)
other(a, *r, **h)
end
```

I believe *args is appropriate for the rest argument, because the star is the splat operator.
I cannot think of any clear logic like that for **h except "another rest argument".
Also ** is the power operator, which is unrelated.
Something related to {}, the literal Hash syntax, would fit better in my opinion.

Do you have any idea of an alternate syntax to **h ?

(Or maybe we should introduce a, b = **h as a joke for a, b = h.values_at(:a, :b))

3) What would {Proc,Method,UnboundMethod}#parameters returns for keywords arguments ?

```
def meth(mandatory, optional = nil, *rest, post, keyw: nil, **h, &block)
end
p method(:meth).parameters
```
Currently: :req, :mandatory], [:opt, :optional], [:rest, :rest], [:req, :post], [:block, :block
Something like:
:req, :mandatory], [:opt, :optional], [:rest, :rest], [:req, :post], [:key, :keyw], [:keyrest, :h], [:block, :block ?

4) I noticed a few problems while experimenting:
```
def a(k: :a, **h)
p [k,h]
end
a(:b, c: :d, e: :f) # => wrong number of arguments (2 for 0) (ArgumentError)
```
It should be "1 for 0"

```
def a(k: :a)
p [k,h]
end
a(r: :a) # => unknown keyword (TypeError)
```
It should say which keyword is missing (and an ArgumentError rather than TypeError, no?).

(Of course I do not expect the current patch to pass these details,
I just mention them to be sure they will be considered.)

#19 - 10/31/2011 07:53 AM - Eregon (Benoit Daloze)

On 30 October 2011 11:10, Yusuke Endoh mame@tsg.ne.jp wrote:

> Hello,
>
> Koichi told me that I can commit my patch to the trunk. So
> I'll do after I fix the issues Benoit reported.

But I'll remain this ticket open to continue to discuss the
spec.

2011/10/30 Benoit Daloze eregontp@gmail.com:

I have a few questions/remarks:

Thank you very much for your trying my patch and your opinion!

It's all my pleasure to test shiny new features.

1) What is the way to pass keyword arguments ?
I would guess **h like:

```
def meth(a, **h)
other(a, **h)
end # => syntax error
```

I didn't implement caller's **.
I wonder if we need it or not. Is "other(a, h)" not enough?

I don't know why I thought keyword arguments were a separate type of
arguments, while there are mostly syntactic sugar for treating the
Hash given, if I understand well (which is fine, except maybe for
optimizations from the implementer POV, but I don't know well).

In that case, it is indeed enough.

BTW, using **h in the argument list does not seems to work in some cases for me:

```
def a(**h)
end # => syntax error, unexpected tPOW
```

Currently, my patch allows ** only when there are one or more
keyword arguments.

This is because I didn't think of any use case.
In addition, I wanted to simplify the implementation of parser.
(Unfortunately, adding a new argument type requries *doubling*
the parser rules to avoid yacc's conflict)
Do you think we need it?

No, sorry for the confusion.
(Ugh, doubling the parser rules sounds bad)

```
def m(k: nil, **h, &block)
end
m() # => undefined method `key?' for nil:NilClass
```

This must be a bug. I'll fix it. Thanks!

It seems to happen only when there is the &block parameter in my experience.

2) I'm a bit dubious about the **h syntax to get (and I guess to
pass) a Hash though.

As I said above, it serves as just get', notpass,' currently.

I believe *args is appropriate for the rest argument, because the
star is the splat operator.
I cannot think of any clear logic like that for **h except "another
rest argument".
Also ** is the power operator, which is unrelated.
Something related to {}, the literal Hash syntax, would fit better
in my opinion.

I accept another syntax, if it is allowed by matz, and yacc :-)

Do you have any idea of an alternate syntax to **h ?

No I don't.

Given the above considerations, **h will only be used to get the
Hash, so I think it is fine.
Notably, delegating with method missing will stay simple as it is:

```
def method_missing(*args, &block)
other(*args, &block)
end
```

3) What would {Proc,Method,UnboundMethod}#parameters returns for
keywords arguments ?

Indeed. I completely forgot. I'll do.

I would happily write the tests for that if you want.
Do you agree on the :key and :keyrest (now I'm thinking to :hash) names ?

4) I noticed a few problems while experimenting:

```
def a(k: :a, **h)
p [k,h]
end
a(:b, c: :d, e: :f) # => wrong number of arguments (2 for 0) (ArgumentError)
```

It should be "1 for 0"

Yes, the error message should be considered.
But in the case, you're passing two arguments actually:
":b", and "{:c=>:d, :e=>:f}"
Do you mean the keyword argument (= hash) should be ignored?

- Show quoted text - >> def a(k: :a) >> p [k,h] >> end >> a(r: :a) # => unknown keyword (TypeError) >> It should say which keyword is missing > > Strongly agreed. I was just lazy :-)

The first virtue of a programmer :-)

#20 - 11/02/2011 02:23 AM - Anonymous

An alternative design is to treat all parameters as keyword
arguments (as Evan said in [ruby-core:40195]).

Â def create_point(x, y, color

#21 - 11/03/2011 12:53 AM - Anonymous

|> It's Python way, and I won't take it.
|What don't you like about this approach? I'd like to know so that hopefully I can formulate an alternative you would like.
|
|My worry about Yusuke's current proposal is that it requires a Hash be allocated on the caller side to use the feature, which makes the usage of keyword arguments much more heavyweight than normal arguments. This in turn means people will either shy away from them or use them and complain that they're too slow (which could make ruby look bad).
|
| - Evan

It could make argument parsing much heavyweight and difficult to
predict, both user POV and implementation POV. Â The Python argument
rules are very complex comparing to other parts of the language.

So your concern is that it would become too complicated if positional
parameters were also treated as keyword, is that right?

My concern with that is then "how do you specify a non optional
parameter by keyword?" (unless you accept Charles' proposal).
Thank you.
-roger-

#22 - 12/22/2011 05:33 AM - marcandre (Marc-Andre Lafortune)

While having fun testing your patch, I encountered an issue with more than 2 rest arguments:

```
def foo(*rest, b: 0, **options)
  [rest, options]
end

foo(1, 2, bar: 0)  # => [[1, 2], {bar: 0}]  OK
foo(1, 2, 3, bar: 0)  # => [[1, 2, {bar: 0}], {bar: 0}]  Not OK
```

#23 - 12/22/2011 06:46 AM - marcandre (Marc-Andre Lafortune)

On 30 October 2011 11:10, Yusuke Endoh mame@tsg.ne.jp wrote:

> Currently, my patch allows ** only when there are one or more
> keyword arguments.
>
> This is because I didn't think of any use case.
> In addition, I wanted to simplify the implementation of parser.
> (Unfortunately, adding a new argument type requries *doubling*
> the parser rules to avoid yacc's conflict)
> Do you think we need it?

I'm worried about cases where one doesn't use named arguments directly but wants to pass them on to another method.

Let's say we have a Gem that defines:

def import(*files, format: :auto_detect, encoding: "utf-8")
# ...
end

Say one wants to implement a variation of this that prints out a progression. If it was possible to have a ** arg, one could:

def my_import(*files, **options)
puts "Importing #{files.size} files: #{files}"
import(*files, options)
end

A new version of the gem can modify the method import by adding options, or changing the default of any option, add options and my_import would work perfectly.

But if we can't define it this way, we have to do some artificial hoop jumping. Either:

def my_import(*files, format: :auto_detect, encoding: "utf-8")
puts "Importing #{files.size} files: #{files}..."
import(*files, format: format, encoding: encoding)# ...
end

Downsides: verbose, won't pass on any new options, default changes won't be reflected

def my_import(*files, format: :auto_detect, **options)
puts "Importing #{files.size} files: #{files}"
import(*files, options.merge(format: format))# ...
end

Downsides: verbose, format looks like a special option and if its default changes in the Gem, it won't be reflected, but others will

def my_import(*args)
files = args.dup
files.pop if files.last.respond_to?(:to_hash)
puts "Importing #{files.size} files: #{files}"
import(*args)
end

Downside: *args less clear then *files, **options, slower, verbose, ...

I feel that the first (currently illegal) version is the much nicer than the alternatives.

> I didn't implement caller's **.
> I wonder if we need it or not. Is "other(a, h)" not enough?

I think one reason to have it is to avoid calling Hash#merge when combining options, like in the above examples.

Instead of

```
def foo(bar: 42, **options)
    baz(extra_option: 1, **options)
end
```

Currently, one has to do:

```
def foo(bar: 42, **options)
    baz(options.merge(extra_option: 1))
end
```

Benoit Daloze wrote:

> 2) I'm a bit dubious about the **h syntax to get (and I guess to pass) a Hash though.
> ...
> Something related to {}, the literal Hash syntax, would fit better in my opinion.
>
> Do you have any idea of an alternate syntax to **h ?

I haven't given much thought, but here's an alternate suggestion, where **h => {*h}:

```
def foo(a, b=1, *rest, {c: 2, d: 3, *options})
end
```

If the parser allows, the {} could be optional, at least in the case without a "hash-rest" argument.

This could actually be two new concepts that could be used everywhere in Ruby (not just for argument passing):
a) Splat inside a hash does a merge, e.g.

```
h = {foo: 1, bar: 2}
{*h, baz: 3} # => {foo: 1, bar: 2, baz: 3}
```

I'm not sure if it should be silent in case of duplicate key (like Hash#merge with no block) or if it should raise an error.

b) Hash destructuring, e.g.:

```
h = {foo: 1, bar: 2}
{foo: 42, extra: 0, *rest} = h
foo   # => 1
extra # => 0
rest  # => {bar: 2}
```

It would be nice to not need a default:

```
{foo, extra, *rest} = h
extra # => nil
```

This could be allowed in arguments too:

```
def foo(a, b=1, *rest, {c, d, *options})
end
```

What do you think?

#24 - 12/22/2011 07:05 AM - marcandre (Marc-Andre Lafortune)

- Target version set to 2.0.0

#25 - 12/22/2011 01:59 PM - Anonymous

Hi,

Not sure why the following modifications made in redmine were not posted on the mailing list...

While having fun testing your patch, I encountered an issue with more
than 2 rest arguments:

```
def foo(*rest, b: 0, **options)
  [rest, options]
end
```

```
foo(1, 2, bar: 0)  # => [[1, 2], {bar: 0}]  OK
foo(1, 2, 3, bar: 0)  # => [[1, 2, {bar: 0}], {bar: 0}]  Not OK
```

On 30 October 2011 11:10, Yusuke Endoh mame@tsg.ne.jp wrote:

> Currently, my patch allows ** only when there are one or more
> keyword arguments.
>
> This is because I didn't think of any use case.
> In addition, I wanted to simplify the implementation of parser.
> (Unfortunately, adding a new argument type requries *doubling*
> the parser rules to avoid yacc's conflict)
> Do you think we need it?

I'm worried about cases where one doesn't use named arguments directly
but wants to pass them on to another method.

Let's say we have a Gem that defines:

def import(*files, format: :auto_detect, encoding: "utf-8")
# ...
end

Say one wants to implement a variation of this that prints out a
progression. If it was possible to have a ** arg, one could:

def my_import(*files, **options)
puts "Importing #{files.size} files: #{files}"
import(*files, options)
end

A new version of the gem can modify the method import by adding
options, or changing the default of any option, add options and
my_import would work perfectly.

But if we can't define it this way, we have to do some artificial hoop
jumping. Either:

def my_import(*files, format: :auto_detect, encoding: "utf-8")
puts "Importing #{files.size} files: #{files}..."
import(*files, format: format, encoding: encoding)# ...
end

Downsides: verbose, won't pass on any new options, default changes
won't be reflected

def my_import(*files, format: :auto_detect, **options)
puts "Importing #{files.size} files: #{files}"
import(*files, options.merge(format: format))# ...
end

Downsides: verbose, format looks like a special option and if its
default changes in the Gem, it won't be reflected, but others will

def my_import(*args)
files = args.dup
files.pop if files.last.respond_to?(:to_hash)
puts "Importing #{files.size} files: #{files}"
import(*args)
end

Downside: *args less clear then *files, **options, slower, verbose, ...

I feel that the first (currently illegal) version is the much nicer
than the alternatives.

> I didn't implement caller's **.
> I wonder if we need it or not. Is "other(a, h)" not enough?

I think one reason to have it is to avoid calling Hash#merge when
combining options, like in the above examples.

Instead of

```
def foo(bar: 42, **options)
    baz(extra_option: 1, **options)
end
```

Currently, one has to do:

```
def foo(bar: 42, **options)
    baz(options.merge(extra_option: 1))
end
```

Benoit Daloze wrote:

> 2) I'm a bit dubious about the **h syntax to get (and I guess to pass) a Hash though.
> ...
> Something related to {}, the literal Hash syntax, would fit better in my opinion.
>
> Do you have any idea of an alternate syntax to **h ?

I haven't given much thought, but here's an alternate suggestion,
where **h => {*h}:

```
def foo(a, b=1, *rest, {c: 2, d: 3, *options})
end
```

If the parser allows, the {} could be optional, at least in the case
without a "hash-rest" argument.

This could actually be two new concepts that could be used everywhere
in Ruby (not just for argument passing):
a) Splat inside a hash does a merge, e.g.

```
h = {foo: 1, bar: 2}
{*h, baz: 3} # => {foo: 1, bar: 2, baz: 3}
```

I'm not sure if it should be silent in case of duplicate key (like
Hash#merge with no block) or if it should raise an error.

b) Hash destructuring, e.g.:

```
h = {foo: 1, bar: 2}
{foo: 42, extra: 0, *rest} = h
foo   # => 1
extra # => 0
rest  # => {bar: 2}
```

It would be nice to not need a default:

```
{foo, extra, *rest} = h
extra # => nil
```

This could be allowed in arguments too:

```
def foo(a, b=1, *rest, {c, d, *options})
end
```

What do you think?

#26 - 12/26/2011 11:27 PM - mame (Yusuke Endoh)

- Assignee changed from ko1 (Koichi Sasada) to mame (Yusuke Endoh)

Hello,

I've committed my patches for keyword arguments, with fixes for
some problems reported during this discussion.

Please try it and let me know if you find any problem.

This feature still requires discussion, so I'm leaving this
ticket open.

--
Yusuke Endoh mame@tsg.ne.jp

#27 - 12/26/2011 11:29 PM - mame (Yusuke Endoh)

Hello, Marc-Andre

2011/12/22, Marc-Andre Lafortune ruby-core-mailing-list@marc-andre.ca:

> While having fun testing your patch, I encountered an issue with more
> than 2 rest arguments:
>
> ```
> def foo(*rest, b: 0, **options)
>   [rest, options]
> end
>
> foo(1, 2, bar: 0)  # => [[1, 2], {bar: 0}]  OK
> foo(1, 2, 3, bar: 0)  # => [[1, 2, {bar: 0}], {bar: 0}]  Not OK
> ```

Good catch! The commits I've done now include a fix for this.

> On 30 October 2011 11:10, Yusuke Endoh mame@tsg.ne.jp wrote:
>
>> Currently, my patch allows ** only when there are one or more
>> keyword arguments.
>>
>> This is because I didn't think of any use case.
>> In addition, I wanted to simplify the implementation of parser.
>> (Unfortunately, adding a new argument type requries *doubling*
>> the parser rules to avoid yacc's conflict)
>> Do you think we need it?
>
> I'm worried about cases where one doesn't use named arguments directly
> but wants to pass them on to another method.

Indeed. I've missed delegate.

>> I didn't implement caller's **.
>> I wonder if we need it or not. Is "other(a, h)" not enough?
>
> I think one reason to have it is to avoid calling Hash#merge when
> combining options, like in the above examples.
>
> Instead of
>
> ```
> def foo(bar: 42, **options)
>     baz(extra_option: 1, **options)
> end
> ```
>
> Currently, one has to do:
>
> ```
> def foo(bar: 42, **options)
>     baz(options.merge(extra_option: 1))
> end
> ```

Oh yeah, I understand the use case.

I'll work for these two, after matz shows his opinion for
your new alternative syntax. Thanks!

--
Yusuke Endoh mame@tsg.ne.jp

#28 - 12/26/2011 11:53 PM - mame (Yusuke Endoh)

Hello, Matz

What do you think about Marc-Andre's alternative syntax for
keyword arguments and mechanism?

2011/12/22, Marc-Andre Lafortune ruby-core-mailing-list@marc-andre.ca:

> I haven't given much thought, but here's an alternate suggestion,
> where **h => {*h}:
>
> def foo(a, b=1, *rest, {c: 2, d: 3, *options})
> end
>
> If the parser allows, the {} could be optional, at least in the case
> without a "hash-rest" argument.
>
> This could actually be two new concepts that could be used everywhere
> in Ruby (not just for argument passing):
> a) Splat inside a hash does a merge, e.g.
>
> ```
> h = {foo: 1, bar: 2}
> {*h, baz: 3} # => {foo: 1, bar: 2, baz: 3}
> ```
>
> I'm not sure if it should be silent in case of duplicate key (like
> Hash#merge with no block) or if it should raise an error.
>
> b) Hash destructuring, e.g.:
>
> ```
> h = {foo: 1, bar: 2}
> {foo: 42, extra: 0, *rest} = h
> foo   # => 1
> extra # => 0
> rest  # => {bar: 2}
> ```
>
> It would be nice to not need a default:
>
> ```
> {foo, extra, *rest} = h
> extra # => nil
> ```
>
> This could be allowed in arguments too:
>
> ```
> def foo(a, b=1, *rest, {c, d, *options})
> end
> ```

I think the mechanism looks neater then my proposal.
My first proposal is less flexible; it allows us only keyword
arguments by making the complex concept, i.e., method arguments,
more complex.

On the contrary, Marc-Andre's proposal consists of some small
concepts (each which looks useful) and minimum extension for
method arguments. It is agreement with the Unix philosophy ---
do one thing and do it well.

I'm not sure if his proposal is implementable or not, but if you
prefer it, I (or anyone) will try to implement it.

Which do you prefer?

--
Yusuke Endoh mame@tsg.ne.jp

#29 - 12/27/2011 06:59 AM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:41814] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Mon, 26 Dec 2011 23:28:36 +0900, Yusuke Endoh mame@tsg.ne.jp writes:

|Oh yeah, I understand the use case.
|
|I'll work for these two, after matz shows his opinion for
|your new alternative syntax. Thanks!

I agree with caller-side **.

```
                            matz.
```

#30 - 12/27/2011 12:53 PM - mame (Yusuke Endoh)

Hello, matz

2011/12/27, Yukihiro Matsumoto matz@ruby-lang.org:

> In message "Re: [ruby-core:41814] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
> on Mon, 26 Dec 2011 23:28:36 +0900, Yusuke Endoh mame@tsg.ne.jp
> writes:
>
> |Oh yeah, I understand the use case.
> |
> |I'll work for these two, after matz shows his opinion for
> |your new alternative syntax. Thanks!
>
> I agree with caller-side **.

Ah, yes, I think there is no longer need for arguing it.
What I'm waiting for is your opinion not about it, but
about [ruby-core:41815].

--
Yusuke Endoh mame@tsg.ne.jp

#31 - 12/27/2011 03:53 PM - matz (Yukihiro Matsumoto)

Hi,

In message "Re: [ruby-core:41815] Re: [ruby-trunk - Feature #5474][Assigned] keyword argument"
on Mon, 26 Dec 2011 23:36:38 +0900, Yusuke Endoh mame@tsg.ne.jp writes:
|
|Hello, Matz
|
|What do you think about Marc-Andre's alternative syntax for
|keyword arguments and mechanism?

I am still not sure if we need hash splat nor hash decomposition, it
might be useful in some cases, but also makes syntax more complex.
So we need more discussion before picking it.

But if we could made consensus I'd make small changes to proposed
syntax.

For hash splat, I'd rather use ** instead of *, because splat in array
and splat in hash are different. Using same operator could cause
confusion. Besides that, I would also restrict hash splat position to
the end, since hashes do not have order.

```
                            matz.
```

#32 - 01/10/2012 10:17 PM - yeban (Anurag Priyam)

Once this proposal has been implemented, would we also want to change the relevant API calls (in core, stdlib, etc.) to use keyword arguments instead of an optional Hash?

#33 - 04/24/2012 10:01 PM - mame (Yusuke Endoh)

- Status changed from Assigned to Closed

Sorry for leaving this ticket for a long time.

Nobu improved the implementation of keyword argument.
Let's check the current situation.
And then, I'd like to file tickets for each remaining issues, and
close this ticket. Please let me know if I miss something.

** with no keyword [ruby-core:40518] [ruby-core:41772]

The following DOES work currently. Thanks nobu!

```
def foo(x, **h)
p [x, h]
end
foo(1)          #=> [1, {}]
foo(1, key: 42)  #=> [1, {key: 42}]
foo(1, 2)       #=> wrong number of arguments (2 for 1)
```

Method#parameters with keyword argument [ruby-core:40518]

The following DOES work currently.

```
def foo(k1: 42, k2: 42, **kr)
end
p method(:foo).parameters
#=> :key, :k1], [:key, :k2], [:keyrest, :kr
```

caller-side ** [ruby-core:40518]

The following does NOT work.

```
h = {x: 42}
foo(**h)
```

Unfortunately, it is not trivial for me to implement.
I'll explain the detail in a separate ticket.

alternative syntax proposal

As far as I know, there is no proposal that matz approved.
It looks difficult to me to change the syntax, because matz seems to
like "**" very much. [ruby-core:41822]

Marc-Andre's hash decomposition [ruby-core:41772]

Unfortunately, matz is not so positive. [ruby-core:41822]

ArgumentError message related to keyword argument

Currently, "unknown keyword" error shows which keyword is missing:

```
def a(k: 42)
end
a(foo: 42)
# => unknown keyword: foo (ArgumentError)
```

There is another proposal about "wrong number of arguments" error.
See: http://bugs.ruby-lang.org/issues/6086

changing stdlib API to use keyword arguments [ruby-core:42037]

I think we should progress it, without breaking compatibility.
But currently, I don't know which API is the target.
Please file a ticket if you find such an API.

--
Yusuke Endoh mame@tsg.ne.jp

#34 - 05/09/2012 03:46 AM - marcandre (Marc-Andre Lafortune)

Hi,

mame (Yusuke Endoh) wrote:

> ## Marc-Andre's hash decomposition [ruby-core:41772]
>
> Unfortunately, matz is not so positive. [ruby-core:41822]

Maybe I misread Matz' comment, but I understood he was not sure yet.

In any case, a new feature request (#6414) was just opened, so we can discuss there.

Files

keyword-argument-patch-20111023.zip	12.2 KB	10/23/2011	mame (Yusuke Endoh)