Ruby trunk - Feature #9049

Shorthands (a:b, *) for inclusive indexing

10/24/2013 11:54 AM - mohawkjohn (John Woods)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

For NMatrix, we've implemented a range shorthand which relies on Hashes: m[1=>3, 2=>4], for example, which returns rows 1 through 3 inclusive of columns 2 through 4 (also inclusive). The original goal was to be able to do m[1:3, 2:4] using the new hash notation, but the new hash notation requires that the key be a symbol — it won't accept an integer.

Whether through the hash interface or not, it'd be lovely if there were a shorthand for slicing matrices (and even Ruby Arrays) using colon. This could just be an alternate syntax for ranges, also — which might make more sense.

The other related shorthand we'd love to find a way to implement is the all-inclusive shorthand. It gets to be a pain to type n[0...n.shape[0], 1...3] to get a submatrix (a slice), and it's really difficult to read. As a work-around, we currently use the :* symbol: n[:*, 1...3]. But it'd be simpler if there were a way to use a splat operator without an operand as a function argument. It might be a special case where the * is treated as a :* automatically. But this edge case might cause confusion with error messages when users make syntax errors elsewhere.

The colon shorthand is the highest priority for us.

Related issues:

Related to Ruby trunk - Feature #14044: Introduce a new attribute `step` in R...	**Rejected**

History

#1 - 10/24/2013 01:53 PM - david_macmahon (David MacMahon)

I like the compactness of the a:b notation. My preference would be for it to be a Range shorthand, but I think that would conflict with the {a:b} Hash syntax (especially when passing a Hash as the last argument to a method).

Another thing that would be useful (and probably belongs in a different feature request) is a NumericRange class that supports a "step_size" attribute in addition to "first" and "last" attributes. Maybe NumericRange is too specific a name since it is possible for non-numeric types to support the "step_size" concept (e.g. Date) so maybe "RangeWithStepSize" would be more appropriate (though too long!). Then we could dream about Matlab-like syntax (1:2:5).to_a => [1, 3, 5]. Maybe just enhance Range to have an explicit step_size that can be something other than the implicit default of 1 (and 0)?

#2 - 10/24/2013 02:29 PM - Anonymous

I would like to see the hash colon syntax extended to numeric keys:

{ 1: 3, 2: 4 } would mean { 1 => 3, 2 => 4 }.

But this wish would collide with the wish to make 1:3 mean 1..3.

As for letting asterisk mean :*, what would happen if there is no comma? Would

```
[ 1, 2, 3 ].reduce *
```

mean this?

```
[ 1, 2, 3 ].reduce() *
```

or this?

```
[ 1, 2, 3 ].reduce( :* )
```

(I'm not trying to say your proposal is impossible.)

#3 - 10/24/2013 02:36 PM - Anonymous

david_macmahon (David MacMahon): I do not think that the wish to make a : b an alias of a .. b
can fit into the language anymore. "x > 0 ? 1 : 2 : 3" could be ambiguously
parsed as "x > 0 ? (1 : 2) : 3" or as "x > 0 ? 1 : (2 : 3)"...

#4 - 10/24/2013 03:23 PM - david_macmahon (David MacMahon)

On Oct 23, 2013, at 10:36 PM, boris_stitnicky (Boris Stitnicky) wrote:

> david_macmahon (David MacMahon): I do not think that the wish to make a : b an alias of a .. b
> can fit into the language anymore. "x > 0 ? 1 : 2 : 3" could be ambiguously
> parsed as "x > 0 ? (1 : 2) : 3" or as "x > 0 ? 1 : (2 : 3)"...

Good point. How about if this new notation limited only to uses within square brackets: [a:b]? There are two distinct cases (maybe more?). One case is where this used as a literal like range = [a:b] and the other where it is used in #[] like array[a:b].

It may seem strange for [...] to create a non-Array literal, but the usage would be pretty straightforward:

```
[a:b]      => a..b
[a:b, c:d] => [a..b, c..d]
[[a:b]]    => [a..b]
```

With this case, your example of x > 0 ? 1 : 2 : 3 would not be syntactically valid, but it could, for example, be re-written as x > 0 ? [1 : 2] : 3 (excessive spaces added for emphasis), which would be equivalent to x > 0 ? 1 .. 2 : 3.

The two cases are mutually exclusive (unless there is some parsing constraint I'm unaware of). The #[] case would be the more useful of the two (by far, IMHO). The [a:b] case doesn't really offer much over the a..b syntax (except for possible consistency with the #[] case).

Dave

#5 - 10/24/2013 03:50 PM - mohawkjohn (John Woods)

@boris_stitnicky I don't think it matters if it collides. It's simple to convert a key-value pair into a range in C code or in Ruby.

david_macmahon (David MacMahon) It'd be nice if it could be done in other functions. In NMatrix, we have both [] and #slice, which do different things. The notation should be consistent for both. (Not just for NMatrix, either.)

#6 - 10/24/2013 03:53 PM - fuadksd (Fuad Saud)

How is a:b better than a..b? two dots are straightforward, unambiguous, well known.

I don’t see a need for it. As for { ’ten’: 10 }, I agree it’s discussable if it’s gonna evaluate to a string or a symbol. If it’s a symbol you pretty muck kill consistency in the case other things will be acceptable on the left side (like { 1: 1, 2: 4, {lol: ‘wut’: 9 }, evaluate_this.method: ’the result').

--

Fuad Saud
Sent with Sparrow (http://www.sparrowmailapp.com/?sig)

#7 - 10/24/2013 05:24 PM - Eregon (Benoit Daloze)

I am not sure m[1:3,2:4] is really preferable to m[1..3,2..4] in Ruby.
The first one is certainly more Matlab, Octave and Python-like but not Ruby-like to my taste.

david_macmahon (David MacMahon) What about (1..5).step(2).to_a ?

#8 - 10/24/2013 11:19 PM - matz (Yukihiro Matsumoto)

Could you be more specific?

- What is the value of 1:2?
- Is it equivalent to 1...2?
- Should non number indexing be allowed? (e.g. n:m that would cause conflict)

Matz.

#9 - 10/24/2013 11:55 PM - mohawkjohn (John Woods)

Yes. 1:2 is the same as 1..2; it's inclusive of the begin and end indices. It is not equivalent to 1...2.

I would think non-number indexing *should* be allowed — but you make a good point. This wouldn't work if implemented via Hash, because it would treat n:m as :n => m. Darn. I suppose that makes things a lot more complicated.

#10 - 10/25/2013 02:53 AM - david_macmahon (David MacMahon)

On Oct 23, 2013, at 11:39 PM, Fuad Saud wrote:

> How is a:b better than a..b? two dots are straightforward, unambiguous, well known.

The tongue-in-cheek answer is that it's better because it's one character shorter. :-) The real answer is somewhat more subtle and perhaps subjective. Here are a few reasons.

1) The a:b form is more compact that a..b and the vertical dots of the ':' character stand out better (visually) than two horizontal dots when reading code:

Proposed: foo[bar.x0:bar.x1, bar.y0:bar.y1, bar.z0:bar.z1]

Current: foo[bar.x0..bar.x1, bar.y0..bar.y1, bar.z0..bar.z1]

2) The first:last form opens up the possibility of a first:step:last syntax for Ranges that have a step size other than 1.

3) It would make transliteration to Ruby of existing Matlab/OctavePython code easier.

4) It is more intuitive for new Ruby programmers who come from a Matlab/Octave/Python background. I'm not sure how much weight this reason carries (maybe negative? :-))

5) Even if it's not deemed to be "better", it does provide another convenient way to make a Range. What's wrong with that?

Dave

#11 - 10/25/2013 02:53 AM - david_macmahon (David MacMahon)

On Oct 24, 2013, at 1:24 AM, Eregon (Benoit Daloze) wrote:

> david_macmahon (David MacMahon) What about (1..5).step(2).to_a ?

The problem is that it creates a Range, and Enumerator, and an Array (plus it's textually long). That's two extra objects compared to just creating a Range and the Array could be very large. Which of the following would you rather do?

```
r = 1:12:1e6 # Create Range with step_size 12.
r[42]        # Computes 1+12*42.
```

or

```
a = (1..1e6).step(12).to_a # Create Range,
                           # create Enumerator,
                           # and expand to largish Array.
a[42]                      # Get element 42 from the array.
```

Dave

#12 - 10/25/2013 03:23 AM - david_macmahon (David MacMahon)

On Oct 23, 2013, at 10:36 PM, boris_stitnicky (Boris Stitnicky) wrote:

> david_macmahon (David MacMahon): I do not think that the wish to make a : b an alias of a .. b
> can fit into the language anymore. "x > 0 ? 1 : 2 : 3" could be ambiguously
> parsed as "x > 0 ? (1 : 2) : 3" or as "x > 0 ? 1 : (2 : 3)"...

I can think of two ways to work around this problem.

The more drastic option would be to change ?:. Instead of it being a ternary operator, it would be two binary operators ? and :. The right hand side of ? would be a Range constructed via the : operator. If the left hand side of ? is truish, then the Range's "first" element would be used, otherwise the Range's "last" element would be used. There could be (in theory, though I'm not sure how practical to implement) an optimization that would avoid creating the Range object in the case of the predicate ? value_if_true : value_if_false idiom. In this case, the your expression would be equivalent to x > 0 ? 1 : 3 since the "step_size" attribute (i.e. 2) would be ignored.

The other far less drastic option would be to use : as a Range "factory" only if it is not part of a ?: operator (this is kind of like operator precedence). In this case, your expression would be equivalent to all of these:

```
 (x > 0 ? 1 : 2) : 3
 (x > 0 ? 1 : 2) .. 3
x > 0 ? 1..3 : 2..3
```

In either case, the use of : as a Range "factory" would be disabled in a Hash context (unless used in parentheses) so that a:1 will always mean {:a => 1} in a Hash context (e.g. as the last argument to a method call).

{a:1} => {:a => 1}

{(a:1) => 2} => {a..1 => 2}

{a:(1:2)} => {:a => 1..2}

Using : when passing a Range as the second-to-last argument followed by a Hash as the last argument would require parentheses:

foo(a:1, k:2) => foo({:a=>b, :k=>2})

foo((a:1), k:2) => foo(a..b, {:k=>2})

Parentheses could also be used to pass a Range as the final argument instead of a Hash:

foo(a:1) => foo({:a => 1})
foo((a:1)) => foo(a..1)

Dave

#13 - 10/25/2013 03:23 AM - agarie (Carlos Agarie)

> 4) It is more intuitive for new Ruby programmers who come from a
> Matlab/Octave/Python background. I'm not sure how much weight this reason
> carries (maybe negative? :-))

I also contribute to NMatrix and I can guarantee that this is relevant, in
a positive way. :)

Carlos Agarie
Software Engineer @ Geekie (geekie.com.br)
+55 11 97320-3878
@carlos_agarie

2013/10/24 David MacMahon davidm@astro.berkeley.edu

> On Oct 24, 2013, at 1:24 AM, Eregon (Benoit Daloze) wrote:
>
> > david_macmahon (David MacMahon) What about (1..5).step(2).to_a ?
>
> The problem is that it creates a Range, and Enumerator, and an Array (plus
> it's textually long). That's two extra objects compared to just creating a
> Range and the Array could be very large. Which of the following would you
> rather do?
>
> ```
> r = 1:12:1e6 # Create Range with step_size 12.
> r[42]        # Computes 1+12*42.
> ```
>
> or
>
> ```
> a = (1..1e6).step(12).to_a # Create Range,
>                            # create Enumerator,
>                            # and expand to largish Array.
> a[42]                      # Get element 42 from the array.
> ```
>
> Dave

#14 - 10/25/2013 03:23 AM - david_macmahon (David MacMahon)

On Oct 24, 2013, at 7:19 AM, matz (Yukihiro Matsumoto) wrote:

> - Should non number indexing be allowed? (e.g. n:m that would cause conflict)

I think that variables should be usable as components of the range (e.g. n:m). See my other message about avoiding conflicts with {n:m}.

I also can't think of any reason to prohibit non-number first and last components (just like Range currently supports), but I think step size (if adopted as part of this proposal) would have to be numeric.

Dave

#15 - 10/26/2013 12:43 PM - Anonymous

david_macmahon (David MacMahon), mohawkjohn: Colon is busy, how about harassing % ?

%s/1 1e6 step 12/ # %s would mean series, returning an enumerator

As far as slicing (multidimensional) matrices is involved, I need to already get myself together
and join NMatrix team :-), but seriously, you need an object for that:

```
class Matrix::Knife
  # here you define, in each dimension, what slices you take and what you drop
end
```

And then, you need to do parametrized subclassing of Matrix for each dimensionality (1D matrices
aka. vectors, common 2D matrices, 3D matrices etc.), and then, each such parametrized subclass
nedds to own its own parametrized subclass of Matrix::Knife. And then you need a cool constructor
for those knives, and that can even be a string:

xxx = Matrix.D3.Knife("*|1:3|0+3+5")
# knife xxx takes all the ranks in dimension 1, ranks 1..3 in dimension 2, and ranks 0, 3 and 5
# in dimension 3, and when it cuts, it produces an instance of Matrix.D3 parametrized subclass
# of Matrix class. That's how I'd see it.
m.slice(xxx) # slicing 3D matrix m with knife xxx
xxx.cut(m) # same as above, with reversed roles or the receiver and the argument

I'm not really sure there is need to bother matz for novel syntax, but if yes, it would be the syntax
for those knives:

%X[* | 1:3 | 0+3+5]

You could cut arrays with them too

```
my_knife = %X[ 1 + 3:5 ]
```

```
( 1 .. 6 ).to_a.slice( my_knife ) #=> [2, 4, 5, 6]
# just like
( 1 .. 6 ).to_a.values_at *[ 1, *3..5 ]
```

#16 - 10/26/2013 01:59 PM - david_macmahon (David MacMahon)

On Oct 25, 2013, at 8:43 PM, boris_stitnicky (Boris Stitnicky) wrote:

> david_macmahon (David MacMahon), mohawkjohn: Colon is busy, how about harassing % ?
>
> %s/1 1e6 step 12/ # %s would mean series, returning an enumerator

I don't think that offers any benefit over just creating a factory method for creating Ranges (augmented with a step size attribute). For example

def _(first,last,step=1); Range.new(first,last,false,step); end

...which would be used like...

_(1,1e6,12)

Of course the "real" definition would probably use *args and be smart about 2 vs 3 args to allow step to be given as an optional 2nd argument.

Even still, using the colon to specify ranges is far more compact and already used in a variety of other languages for that purpose. In addition to Matlab/Octave and Python, it also is used to specify ranges in R, Verilog, and to some degree even in Excel spreadsheets (for ranges of cells).

Is your objection to using the colon technical (e.g. impracticality of parsing) or philosophical?

Thanks,
Dave

#17 - 10/26/2013 02:24 PM - Anonymous

david_macmahon (David MacMahon): Technical. Colon is already busy in the basic fabric of the language
({ a: :b }, ternary operator ... ? ... : ...). You'll see what matz will tell you.
Otherwise, I'm not proposing anything, just trying to be useful phantasizing which
characters other than colon could take more abuse. I arrived to %.

#18 - 10/26/2013 02:44 PM - mohawkjohn (John Woods)

@boris_stitnicky What about ~? x[3~4,1~5], for example. I don't like it as much as colon, but it looks a little more intuitive than %.

#19 - 10/26/2013 05:11 PM - Anonymous

@mohawkjoh: Tilde is bad, too. From basic ASCII (I looked), everything is taken, except for ' and ^ should be avoided, because it means power elsewhere. I'd stay with Matlab/Octave's colon, if you can't put up with .. / ... Sadly, there is no way to overload : in regular syntax anymore. Your options are:

1. Settle for strings: "*, 2:4", or "*|2:4" etc. (do we hate quotes).

2. Ask for novel %(some character) literals. But m[%X(*|2:5)] is still too much clutter,
   when one dreams of m[*, 2:5]. (On custom literals, #8807, I voted against, as I thought
   it would burden noobs. Maybe I was wrong.)

3. Turn to Unicode, as I do in my Pyper, via #method_missing: m.‖□‖0•2□5‖ (insane APLism)

4. Use a block instance execced in a special object: m.slice { □ | (2..5) }

5. Ask for the following syntactic feature:

m⟨ something ⟩ # ⟨⟩ are angle brackets or some similar novel delimiters

meaning

m("something")

This feature would rid us of the hated quotes. Then

m⟨ * | 0 + 2:4 ⟩

would simply mean

m("* | 0 + 2:4")

where the single string argument you would parse ad libitum. I personally like this
last option best, though it is far from problem-free. The delimiters would have to be
escaped in the statement body. So this is what I think we should ask from Matz.

Varieties would be:

m‹ some string ›, m› some string ‹, m« some string », m» some string «
m ‹ some string ›, m » some string «, etc.

#20 - 10/26/2013 05:11 PM - Anonymous

On the secon thought, that m » my string « literal is problem-ridden, too.

#21 - 10/27/2013 12:53 PM - david_macmahon (David MacMahon)

On Oct 26, 2013, at 1:11 AM, boris_stitnicky (Boris Stitnicky) wrote:

> Sadly, there is no way to overload : in regular Ruby anymore.

I'm not quite ready to give up on it yet, but I won't mention it again unless I can figure out something concrete. :-)

Dave

#22 - 10/28/2013 01:53 PM - nobu (Nobuyoshi Nakada)

(13/10/26 17:11), boris_stitnicky (Boris Stitnicky) wrote:

> @mohawkjoh: Tilde is bad, too. From basic ASCII (I looked), everything is taken, except
> for ˈ and which should be avoided because it means power in many languages.

^ is XOR operator.

#23 - 10/29/2013 10:03 PM - Anonymous

nobu (Nobuyoshi Nakada) wrote:

> ^ is XOR operator.

There you go. I'm yet to XOR things in my life :-) So basic ASCII is 100% covered.

#24 - 11/05/2013 06:23 PM - duerst (Martin Dürst)

[Sorry for the delay of this message. I wrote most of this mail on a
plane, but had to check a few loose ends, and forgot about that when off
the plane.]

I'm not at all convinced that we need to add ':' to '..' for ranges (or,
to say it more clearly, I'm against changing/adding it). In Ruby, ranges
use '..', and exponentiation uses '**', and so on. Other languages may
use different conventions. That's just it, there's no need to fix it.

For more details, please see below.

On 2013/10/25 2:29, David MacMahon wrote:

> On Oct 23, 2013, at 11:39 PM, Fuad Saud wrote:
>
> > How is a:b better than a..b? two dots are straightforward, unambiguous, well known.
>
> The tongue-in-cheek answer is that it's better because it's one character shorter. :-) The real answer is somewhat more subtle and perhaps subjective. Here are a few reasons.
>
> 1) The a:b form is more compact that a..b

&& is also used more than &, and is less compact, and 'and', which is
mostly used these days, is even less compact.

> and the vertical dots of the ':' character stand out better (visually) than two horizontal dots when reading code:
>
> Proposed: foo[bar.x0:bar.x1, bar.y0:bar.y1, bar.z0:bar.z1]
>
> Current: foo[bar.x0..bar.x1, bar.y0..bar.y1, bar.z0..bar.z1]

Yes. But if we are at aesthetics and the like, the '..' is actually the
better expression of a range than ':', at least if you ask me. And you
can write the above as:

foo[bar.x0 .. bar.x1, bar.y0 .. bar.y1, bar.z0 .. bar.z1]

Or add spaces or parentheses to your liking to make it clearer and
easier to read. That's what's done all the time to make the structure of
an expression easier to grasp.

> 2) The first:last form opens up the possibility of a first:step:last syntax for Ranges that have a step size other than 1.

There are many other ways to get there. It should be fairly easy e.g. to
make first..step..last behave that way, 1..2..7 currently produces a
syntax error, but that could be changed.
(also please note that Python uses first:last:step)

> 3) It would make transliteration to Ruby of existing Matlab/OctavePython code easier.

Not really. I worked with some of my students on a project to
support/automate conversion from Python to Ruby. Some of it was easy,
some of it was tedious but straightforward, and some of it is
essentially hopeless. Converting ":" to "..", even if only in certain
circumstances, falls into the easy bucket.

The hopeless stuff includes semantic differences, in particular what
different languages take as truthy and falsy. Every
if expression:
in Python has to be rewritten as
if python_true?(expression)
in Ruby, because otherwise the program will do the wrong thing if
expression evaluates to an empty array or string or 0 or so.

> 4) It is more intuitive for new Ruby programmers who come from a Matlab/Octave/Python background. I'm not sure how much weight this reason carries (maybe negative? :-))

Not negative, and probably even positive, but way not enough to actually
implement it. Different programming languages have (more or less)

different ways of writing operators, but there are many more important
differences that people have to get used to anyway.

> 5) Even if it's not deemed to be "better", it does provide another convenient way to make a Range. What's wrong with that?

Ranges with steps sound interesting (not personally for me but in
general), but that's a separate issue from the actual syntax, and in
particular from the :/.. discussion. I actually think we should close
the current issue and open a separate issue for ranges with steps.

Other than that, yes there are for example a lot of ways to make a
String, but each of them is there for a reason (even if in some cases it
may be only because it was adopted from Perl and not kicked out yet).

Regards,   Martin.

#25 - 03/15/2018 12:24 AM - mrkn (Kenta Murata)

- Related to Feature #14044: Introduce a new attribute `step` in Range added

#26 - 03/15/2018 12:26 AM - mrkn (Kenta Murata)

- Description updated