Ruby master - Feature #16812

Allow slicing arrays with ArithmeticSequence

04/23/2020 03:32 PM - zverok (Victor Shepelev)

Status:	Assigned	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

I believe when concepts of ArithmeticSequence and Range#% were introduced, one of the main intended usages was array slicing in scientific data processing. So, it seems to make sense to allow this in Array#[]:

```
ary[(5..20) % 2] # each second element between 5 and 20
ary[(0..) % 3] # each third element
ary[10.step(by: -1)] # elements 10, 9, 8, 7 ....
```

PR is here.

My reasoning is as follows:

1. As stated above, ArithmeticSequence and Range#% seem to have been introduced exactly for this goal
2. Python has its slicing syntax as begin:end:step (with a possibility to omit either), and it seems to be well respected and used feature for data processing. So I believe it is useful, and relatively easy to integrate into existing functionality

I expect the usual "it is ugly and unreadable!" backlash.
I don't have an incentive, nor energy, to "defend" the proposal, so I would not.

History

#1 - 04/23/2020 08:04 PM - Eregon (Benoit Daloze)

Rather neutral on this, but would you want that to work for Array#[]= too?
I would be against Array#[]= as it's already so complicated and that would just make it a lot more so.

In Array#[] it's probably fine though.

#2 - 04/23/2020 08:44 PM - zverok (Victor Shepelev)

Eregon (Benoit Daloze), I wanted at first to see what people say about this one :)

Array#[]= is a thing that should be kinda "symmetric", but playing a bit with it, I understood that I am afraid of trying to guess what would be "logical".

Honestly, I can't remember I've ever used a form like a[1..3] = 'x', and its behavior is kinda "theoretically logical", but at the same time only one of the things you may "intuitively" expect ("replace all three elements with one, changing array's size" wouldn't be my first guess...).

So, at least for now, my only proposal is Array#[].

#3 - 04/24/2020 12:19 AM - Dan0042 (Daniel DeLorme)

Theoretically I'm in favor but there's some edge cases that need consideration.

```
nums = (0..20).to_a

s = 10.step(by: -2) # 10, 8, 6, 4, 2, 0, -2, ...
nums[s] #=> [10, 8, 6, 4, 2, 0, 19, 17, ...] ???

s = (-5..5) % 2 # -5, -3, -1, 1, 3, 5
nums[s] #=> [16, 18, 20, 1, 3, 5] ???
```

#4 - 04/24/2020 04:08 AM - nobu (Nobuyoshi Nakada)

A few bugs.

- Float ArithmeticSequence crashes.

```
$ ./ruby -e '[*0..10][(0.0..)%10]'
Assertion Failed: ../src/include/ruby/3/arithmetic/long.h:136:ruby3_fix2long_by_shift:"RB_FIXNUM_P(x)"
```

- If overridden take_while (and drop_while) returns non-Array, crashes.

```
$ ./ruby 'a = (1..10)%2; def a.take_while; nil; end; [*1..10][a]'
-e:1:in `<main>': wrong argument type nil (expected Array) (TypeError)
```

These resulted in assertion failures, but would segfault when compiled with NDEBUG.

#5 - 04/24/2020 05:09 AM - mrkn (Kenta Murata)

- Assignee set to matz (Yukihiro Matsumoto)

- Status changed from Open to Assigned

I'm positive this if the behavior is the same as Python's list slicing.
If the behavior will be different from Python's, I'm negative because it confuses PyCall users.

#6 - 04/25/2020 05:48 PM - zverok (Victor Shepelev)

As there is no immediate rejection, I updated the implementation, making it more robust.

Dan0042 (Daniel DeLorme), I tried to make edge cases consistent, so now they are...

```
(0..20).to_a[10.step(by: -2)]
# => [10, 8, 6, 4, 2, 0] -- avoids weird cycling
(0..20).to_a[(-5..5) % 2]
# => [] -- this is consistent with
(0..20).to_a[-5..5] # which can be thought as (-5..5) % 1
# => []
# Note, though:
(0..20).to_a[-19..5]
# => [2, 3, 4, 5] -- not literally "from -19 to 5", but "from 19th from the end to 5th from the beginning"
# ...so...
(0..20).to_a[(-19..5)%2]
# => [2, 4]
```

nobu (Nobuyoshi Nakada) I've tried to fix bugs. Now float begin/end is processed correctly, float step is TypeError, and the code does not rely on #take_while/#drop_while.

mrkn (Kenta Murata) I've checked against Python impl, and believe the behavior is mostly the same. One difference I am aware of is this:

Python:

```
list(range(10))[-100:100:2]

#=> [0, 2, 4, 6, 8]
```

Ruby:

```
[*0..10][(-100..100)%2]
# => nil
```

That's because first of all I wanted to make it consistent with

```
[*0..10][-100..100]
# => nil
```

...which may be questioned (like, "range from -100 to 100 includes 0..10, so it should fetch entire array"), but that's how it is now :)

#7 - 06/18/2020 02:06 AM - mrkn (Kenta Murata)

It may be better to change the behavior of [*0..10][-100..100] because [*0..10][..100] does not return nil:

```
[*0..10][..100]
# => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

And the following cases seems inconsistent to me:

```
[*0..10][0..12]
# => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[*0..10][-12..-1]
# => nil
```

#8 - 06/19/2020 02:12 PM - mrkn (Kenta Murata)

I made a patch: https://github.com/ruby/ruby/pull/3241

#9 - 06/26/2020 07:08 PM - Dan0042 (Daniel DeLorme)

mrkn (Kenta Murata) wrote in #note-7:

> It may be better to change the behavior of [*0..10][-100..100]

I somewhat agree with that. When using range slicing most combinations make sense:

```
[*0..10][0..4]   #first elements
[*0..10][-5..-1] #last elements
[*0..10][1..-2]  #middle elements
```

But a negative start with a non-negative end is quite weird. What is that operation even supposed to mean? What is it useful for?

```
[*0..10][-8..8]  #????
```

```
8.times{ |i|  p (0..i) => [*0..i][-3..3]  }
{0..0=>nil}
{0..1=>nil}
{0..2=>[0, 1, 2]}
{0..3=>[1, 2, 3]}
{0..4=>[2, 3]}
{0..5=>[3]}
{0..6=>[]}
{0..7=>[]}
```

So even if [*0..10][-100..100] remains supported forever (there doesn't seem to be a point in breaking compatibility; see #16822), it could emit a verbose-mode warning.
And ArithmeticSequence slicing should not attempt to be consistent with that case, because it's useless to start with.
So I believe there are two useful/meaningful possibilities for (0..20).to_a[(-5..5) % 2]
a) [16, 18, 20] ignore trailing non-negative values, like (-5..) % 2; I think this makes the most sense
b) [1, 3, 5] ignore leading negative values, like python

#10 - 06/27/2020 09:14 AM - zverok (Victor Shepelev)

> But a negative start with a non-negative end is quite weird. What is that operation even supposed to mean? What is it useful for?

I believe such edge cases might emerge not being directly written, but when dynamically calculated. Imagine calculating some anchor element, and then taking N elements around it. Then, you have, say...

```
def around_mean(ary, count: 3)
  i = ary.index(ary.sum / ary.length)
  ary[i-count..i+count]
end
around_mean((1..20).to_a)
# => [7, 8, 9, 10, 11, 12, 13]
around_mean((1..6).to_a)
# => [6] -- hm, it is a bit strange
around_mean((1..6).to_a, count: 10)
# => nil -- hm, it is even weirder...
```

The example before last is [1, 2, 3, 4, 5, 6][-1..5] which "intuitively weird", as one might expect something like:

```
  1 2 3 4 5 6
^^^^^^^^^^^
```

The very last example is [1, 2, 3, 4, 5, 6][-8..12] -- and it even doesn't produce empty array (I pointed at this at #16822, too).

That's not the best possible example, but at least it demonstrates how we can arrive at edge case situation and why we (probably) might expect different behavior here.

#11 - 07/20/2020 07:04 AM - matz (Yukihiro Matsumoto)

The basic bahavior seems OK. Probably we need to investigate some corner cases, but you can commit (and we experiment).

Matz.