Ruby trunk - Feature #13904

getter for original information of Enumerator

09/15/2017 03:27 PM - znz (Kazuhiro NISHIYAMA)

Status:	Closed	
Priority:	Normal	
Assignee:	mrkn (Kenta Murata)	
Target version:		

Description

At https://gitter.im/red-data-tools/ja?at=59b0aaa097cedeb04828e268 ,
mrkn says narray and pycall use internal information of ruby to check Range#step(n).

People of red-data-tools/ja suggest subclass of Enumerator.
But I think it does not match Ruby's '□□□□□□' (I don't know this word in English), so I suggest to add some methods to Enumerator class.

proof of concept attached.

Usage:

```
% irb -r irb/completion --simple-prompt
>> e=(1..2).step(3)
=> #<Enumerator: 1..2:step(3)>
>> e.receiver
=> 1..2
>> e.method_name
=> :step
>> e.arguments
=> [3]
```

#method is conflict with Kernel#method, so use #method_name instead.

Related issues:

Related to Ruby trunk - Feature #14044: Introduce a new attribute `step` in R...	**Rejected**
Is duplicate of Ruby trunk - Feature #3714: Add getters for Enumerator	**Closed**
Has duplicate Ruby trunk - Feature #15092: Provide step count in Range constr...	**Rejected**

Associated revisions

Revision f1506933 - 08/06/2018 09:08 AM - mrkn (Kenta Murata)

enumerator.c: Introduce Enumerator::ArithmeticSequence

This commit introduces new core class Enumerator::ArithmeticSequence.
Enumerator::ArithmeticSequence is a subclass of Enumerator, and
represents a number generator of an arithmetic sequence.

After this commit, Numeric#step and Range#step without blocks
returned an ArithmeticSequence object instead of an Enumerator.

This class introduces the following incompatibilities:

- You can create a zero-step ArithmeticSequence, and its size is not ArgumentError, but Infinity.
- You can create a negative-step ArithmeticSequence from a range.

[ruby-core:82816] [Feature #13904]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@64205 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 64205 - 08/06/2018 09:08 AM - mrkn (Kenta Murata)

enumerator.c: Introduce Enumerator::ArithmeticSequence

This commit introduces new core class Enumerator::ArithmeticSequence.
Enumerator::ArithmeticSequence is a subclass of Enumerator, and
represents a number generator of an arithmetic sequence.

After this commit, Numeric#step and Range#step without blocks
returned an ArithmeticSequence object instead of an Enumerator.

This class introduces the following incompatibilities:

- You can create a zero-step ArithmeticSequence, and its size is not ArgumentError, but Infinity.
- You can create a negative-step ArithmeticSequence from a range.

[ruby-core:82816] [Feature #13904]

Revision 64205 - 08/06/2018 09:08 AM - mrkn (Kenta Murata)

enumerator.c: Introduce Enumerator::ArithmeticSequence

This commit introduces new core class Enumerator::ArithmeticSequence.
Enumerator::ArithmeticSequence is a subclass of Enumerator, and
represents a number generator of an arithmetic sequence.

After this commit, Numeric#step and Range#step without blocks
returned an ArithmeticSequence object instead of an Enumerator.

This class introduces the following incompatibilities:

- You can create a zero-step ArithmeticSequence, and its size is not ArgumentError, but Infinity.
- You can create a negative-step ArithmeticSequence from a range.

[ruby-core:82816] [Feature #13904]

Revision 25a5227a - 08/09/2018 01:18 PM - mrkn (Kenta Murata)

enumerator.c: undef new and allocate of ArithmeticSequence

Undefine new and allocate methods of Enumerator::ArithmeticSequence.

[ruby-core:82816] [Feature #13904]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@64256 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 64256 - 08/09/2018 01:18 PM - mrkn (Kenta Murata)

enumerator.c: undef new and allocate of ArithmeticSequence

Undefine new and allocate methods of Enumerator::ArithmeticSequence.

[ruby-core:82816] [Feature #13904]

Revision 64256 - 08/09/2018 01:18 PM - mrkn (Kenta Murata)

enumerator.c: undef new and allocate of ArithmeticSequence

Undefine new and allocate methods of Enumerator::ArithmeticSequence.

[ruby-core:82816] [Feature #13904]

History

#1 - 09/15/2017 05:33 PM - zverok (Victor Shepelev)

Super-upvote!

In fact, recently I became rather concerned with a lack of "inspectability" of Ruby's own objects (like "how #format would parse this string and what groups it has", or internal structure of Regexp and so on).

#2 - 09/15/2017 10:36 PM - Eregon (Benoit Daloze)

I agree, I think it's a good idea to expose such information when it is available in #inspect and it is user-provided (not internal).

#3 - 09/17/2017 10:42 AM - knu (Akinori MUSHA)

- Assignee set to knu (Akinori MUSHA)

#4 - 09/18/2017 05:31 AM - knu (Akinori MUSHA)

Enumerator is about abstracting enumeration and encapsulation of the source is by design. If we exposed the guts of an Enumerator, people would start to look into the enclosed object and do (I think are) evil things for optimization and specialization, which should be in the opposite direction of abstraction. If you expect something other than a one-way, not necessarily rewindable stream, I think you should create a new model that describes the aspect by sub-classing Enumerator or making a mix-in.

In summary, if you want to handle a numeric enumerator specially, I'd say you should create a class for that instead of making surrounding methods guess or find out what they receive is special by invesigating its internals. It shouldn't be a good way to introduce a new idea to the language.

#5 - 09/18/2017 12:35 PM - zverok (Victor Shepelev)

> Enumerator is about abstracting enumeration and encapsulation of the source is by design.

Well, to be honest, it seems like "forced encapsulation", and is against Ruby's hackable nature.
I understand your concerns, but if something that even on #inspect looks like #<ClassName x:y> provides no access to x and y (though definitely knows them) it just "doesn't feel right".

I believe that composite objects (and enumerator by nature is composite: source + enumeration method) should provide access to what they are composed of, for bad or for good. It is completely possible new ways of interaction and new libraries would emerge in this case.

#6 - 09/18/2017 11:04 PM - shevegen (Robert A. Heiler)

> evil things for optimization and specialization

Reminds me of good old evil.rb - now I am suddenly all for it! Just kidding. :-)

I have no particular pro or con opinion here but I think that one of ruby's philosophy is to put trust into the ruby hacker to do what he/she wants to do, e. g. duck patching modifying any of ruby's core functionality (class String, Array, Hash etc...), use .send() or .instance_variable_get() at leisure (I used this today to obtain @logged_in content in Net::FTP instances ... I think there is another method to access it, such as open?, but I saw the output of @logged in via pp, and thought I wanted to get the value, so I used instance_variable_get()) etc... ruby is very, very flexible. It's like a lispy-smalltalk-something OOP language. Encapsulation is possible but it also restricts what you can do (logically), so it is not always ruby's philosophy - to let the ruby hacker do what he/she wants, if he/she really wants to. This philosophy is also, I think, why constants in ruby are not 100% constants that can never be changed - ruby allows you to change constants. While the word constant may thus be a misnomer, I think it fits very well into the ruby philosophy overall.

In regards to encapsulation, that is why, I think, .public_send() was added and people can use that for the public/private distinction more clearly. I myself love .send(), freedom to all objects and especially all the duckling objects. \o/

By the way, the idea of composite objects reminds me a bit of what matz wrote about interface objects (specification where objects could/should respond to the same method-behaviour ... all ducks should be able to quack and swim ... so if something quacks and swims, even if it is a wolf disguised as a sheep, it may be treated to be an "acceptable duck").

But as said, I really am neutral here either way. :-)

Perhaps the ruby core team can discuss this a bit (I don't know japanese so I do not know what thoughts are conveyed above).

#7 - 09/19/2017 03:47 AM - knu (Akinori MUSHA)

Well, this issue states that the motivation for introducing the getter methods is for getting return values of Range#step(n) to be handled specially by some libraries or individual methods, and as I said it is to me a typical misuse of such accessors I've been thinking of and I believe there's a better way to achieve the goal, so I can't buy that argument.

I can hardly believe you'd be happy (at the cost of introducing a new feature) if you had to check x.instance_of?(Enumerator) && x.receiver.is_a?(Numeric) && x.method_name == :step every time your method was passed an object that might be a step object. It really looks like you are doing some hack as the last resort because you don't have anything better yet. If stepped range objects are useful, they deserve a new class for their own.

#8 - 10/22/2017 05:30 AM - knu (Akinori MUSHA)

- Is duplicate of Feature #3714: Add getters for Enumerator added

#9 - 10/22/2017 05:31 AM - knu (Akinori MUSHA)

- Status changed from Open to Rejected

Closing because the given use case does not seem to be a good one to support the proposal.

#10 - 10/22/2017 05:32 AM - knu (Akinori MUSHA)

(And it is also a duplicate of #3714)

#11 - 10/23/2017 06:18 AM - mrkn (Kenta Murata)

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

#12 - 03/15/2018 06:35 AM - mrkn (Kenta Murata)

- Assignee changed from knu (Akinori MUSHA) to matz (Yukihiro Matsumoto)

- Status changed from Rejected to Assigned

(Continue from #14044)

Following today's developers meeting, I propose to introduce a subclass of Enumerator.

This is the example implementation of the subclass:

```
class Enumerator::ArithmeticSequence < Enumerator
  attr_reader :first  # already in Enumerator
  attr_reader :last   # newly introduced
  attr_reader :step   # newly introduced

  def inspect
    # appropriate string representation of this class instances
  end
end
```

Integer#step and Range#step of integer ranges should return the instance of Enumerator::ArithmeticSequence like:

```
p 1.step         #=> (1.step)
p 1.step(10)     #=> (1.step(10))
p 1.step(10, 2) #=> (1.step(10, by:2))
p 1.step(by: 2) #=> (1.step(by:2))

p (1..10).step    #=> (1.step(10))
p (1..10).step(2) #=> (1.step(10, by:2))
```

I think introducing this class doesn't introduce incompatibility from the current behaviors, but increase the usefulness of the results of Integer#step and Range#step.

#13 - 03/15/2018 06:38 AM - matz (Yukihiro Matsumoto)

- Assignee changed from matz (Yukihiro Matsumoto) to mrkn (Kenta Murata)

Sounds reasonable. Accepted.

Matz.

#14 - 03/15/2018 12:12 PM - Eregon (Benoit Daloze)

This seems fine but very specific.
I still think exposing an Enumerator's receiver, method name and arguments is better as it is more general, intuitive and useful.

I don't think there is anything wrong with checking the condition mentioned above by knu (Akinori MUSHA) in e.g. pycall.rb for #[].

#15 - 03/15/2018 01:19 PM - Hanmac (Hans Mackowiak)

what about something when you don't have method name or arguments to expose?

like that idea with a combined Enumerator?
that does interate [1,2,3] first, then ["a", "b", "c"] and so on.

for such thing i don't think that there is much to expose

#16 - 08/06/2018 09:08 AM - mrkn (Kenta Murata)

- Status changed from Assigned to Closed

Applied in changeset trunk|r64205.

---

enumerator.c: Introduce Enumerator::ArithmeticSequence

This commit introduces new core class Enumerator::ArithmeticSequence.
Enumerator::ArithmeticSequence is a subclass of Enumerator, and
represents a number generator of an arithmetic sequence.

After this commit, Numeric#step and Range#step without blocks
returned an ArithmeticSequence object instead of an Enumerator.

This class introduces the following incompatibilities:

- You can create a zero-step ArithmeticSequence, and its size is not ArgumentError, but Infinity.
- You can create a negative-step ArithmeticSequence from a range.

[ruby-core:82816] [Feature #13904]

#17 - 09/11/2018 01:01 AM - mrkn (Kenta Murata)

- Has duplicate Feature #15092: Provide step count in Range constructor added

Files

poc.diff	1.42 KB	09/15/2017	znz (Kazuhiro NISHIYAMA)