Ruby - Feature #21148

Class proposal: IndefiniteNumeric < Numeric

02/19/2025 01:21 AM - Student (Nathan Zook)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Suppose someone deals five cards face down from a regulation poker deck, and we wish to reason about the number of aces. We know that that number is one of zero, one, two, three, or four. Therefore, if someone asks "is this number larger than negative one?", the answer is yes even without ever needing to resolve the actual value.

This proposal is inspired by the proposed Enumerable::Lazy::Length class of Kevin Newton in response to #21135

The IndefiniteNumeric class would serve as a base class for such values. In particular, it relies on a subclass to define #value, which returns a definite Numeric.

```
class IndefiniteNumeric < Numeric
   def coerce(other) = value.coerce(other)
   def +(other) = value + other
   def -(other) = value - other
   def *(other) = value * other
   def /(other) = value / other
   def <=>(other) = value <=> other
end
```

It is expected that in particular, <=> will be overridden for subclasses where #value might be slow to return.

Usually, we will know more than just that the value in question is Numeric. It seems appropriate have IndefiniteInteger and so forth. What is not clear to me is if IndefiniteInteger should be a subclass of IndefiniteNumeric or of Integer. (Such a thing does not appear to be currently possible.) If subclassing cannot be, what about defining #kind_of?(Integer) to be true?

Note: with the length of an Enumerable as inspiration, I would argue that an IndefiniteInteger might actually value to Float::INFINITY, so it should NOT define <=> against plus or minus Float::INFINITY.

History

#1 - 02/19/2025 03:27 AM - nobu (Nobuyoshi Nakada)

Student (Nathan Zook) wrote:

> The IndefiniteNumeric class would serve as a base class for such values. In particular, it relies on a subclass to define #value, which returns a definite Numeric.
>
> It is expected that in particular, <=> will be overridden for subclasses where #value might be slow to return.

It sounds like a role of a Module, not a Class.

> Usually, we will know more than just that the value in question is Numeric. It seems appropriate have IndefiniteInteger and so forth. What is not clear to me is if IndefiniteInteger should be a subclass of IndefiniteNumeric or of Integer. (Such a thing does not appear to be currently possible.) If subclassing cannot be, what about defining #kind_of?(Integer) to be true?

Ruby prohibits so-called diamond inheritance, and uses modules.

What is the necessity that it should be a built-in Class/Module?

#2 - 02/20/2025 12:37 AM - Student (Nathan Zook)

I was not talking about diamond inheritance. I was talking about

```
Numeric
  Float
  IndefiniteNumeric
```

```
    IndefiniteInteger
    IndefiniteFloat
  Integer
...
```

vs

```
Numeric
  Float
    IndefiniteFloat
  IndefiniteNumeric
  Integer
    IndefiniteInteger
...
```

which might break things.

But that discussion is mostly moot when talking Class vs Module. "Indefiniteness" is clearly a property that might apply to objects in a way that strongly matches the Module concept. However, as I think about your observation, it seems that I might have been thinking too specifically about the application I was envisioning.

Perhaps my real issue is with the behavior of Numeric. Subclasses of Numeric are required to define #coerce (and encouraged to implement the arithmetic operators) when in many cases, these objects are numbers with additional context. Whether we are talking cats in a box, the number of elements in an enumerable, or the change from a purchase, there is a definite value associated with the object, and once we have that, all of these arithmetic operations, to include coercion, are generally set.

As for "necessity", after thirty years, we're probably passed that. :D For utility, however, I think that this significantly simplifies subclassing Numeric.

Assuming this idea is desirable, I see three approaches:

- New module, Value with the above method definitions. This could be included by new subclasses of Numeric as desired.
- New subclass of Numeric, Value, with the above method definitions.
- Modify Numeric to check for the presence of #value in objects where #coerce is not defined, and using the result as described.

The example class Tally in the documentation strikes me as wrapping a representation around the whole numbers. As such, it is natural coerce other values into Tally objects, but for the cases I envision, it goes the other way.

The proposed method definitions fill out contract for Numeric subclasses. While it might be reasonable to include these definitions in a class which does not descend from Numeric, such a class would not gain the functionality from Numeric that these definitions enable.

Between subclassing Numeric and changing Numeric, I feel nervous about changing behavior, even if it means no raising against code that currently fails a contract.

Also, I am not here to ask for the core team to implement this feature. I am looking for support for and guidance on a feature before I develop the code and submit it. (Unless that would be even *more* work for the core team...

#3 - 03/11/2025 09:53 AM - mame (Yusuke Endoh)

Student (Nathan Zook) wrote:

> Suppose someone deals five cards face down from a regulation poker deck, and we wish to reason about the number of aces. We know that that number is one of zero, one, two, three, or four. Therefore, if someone asks "is this number larger than negative one?", the answer is yes even without ever needing to resolve the actual value.

What would ace_count > 2 be, for example? Return true or false according to its probability? Raise an exception? Something else?