Ruby master - Feature #1408

0.1.to_r not equal to (1/10)

04/26/2009 06:21 PM - phasis68 (Heesob Park)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td>matz (Yukihiro Matsumoto)</td>
</tr>
<tr>
<td>Target version</td>
<td>2.0.0</td>
</tr>
</tbody>
</table>

Description

=begin
ruby -e 'p 0.1.to_r'
(3602879701896397/36028797018963968)

whereas

ruby -e 'p "0.1".to_r'
(1/10)
=end

Related issues:
Has duplicate Ruby master - Bug #5309: 0.6.to_r != "0.6".to_r

Rejected 09/13/2011

History

#1 - 04/27/2009 09:49 AM - phasis68 (Heesob Park)
=begin
2009/4/27 Martin DeMello martindemello@gmail.com:

On Sun, Apr 26, 2009 at 2:51 PM, Heesob Park redmine@ruby-lang.org wrote:

   ruby -e 'p 0.1.to_r'
   (3602879701896397/36028797018963968)

   whereas

   ruby -e 'p "0.1".to_r'
   (1/10)

What, in theory, could be done about this? By the time to_r is
invoked, 0.1 is already a binary float, with the implicit rounding
off.

In theory, Float#to_r could be done through Float#to_s#to_r.

Regards,
Park Heesob
=end

#2 - 04/27/2009 10:09 AM - shyouhei (Shyouhei Urabe)
=begin
2009/4/27 Martin DeMello martindemello@gmail.com:

On Sun, Apr 26, 2009 at 2:51 PM, Heesob Park redmine@ruby-lang.org wrote:

   ruby -e 'p 0.1.to_r'
   (3602879701896397/36028797018963968)

   whereas

   ruby -e 'p "0.1".to_r'
   (1/10)
What, in theory, could be done about this? By the time to_r is invoked, 0.1 is already a binary float, with the implicit rounding off.

In theory, Float#to_r could be done through Float#to_s#to_r.

-1. That loses data.

=end

#3 - 05/01/2009 09:12 PM - rogerdpack (Roger Pack)

=begin
-1 that loses data.

True--however the (current) code for String#to_s attempts to determine whether the floating point number "is the equivalent default for the rounded value" (i.e. if it round trips).
Do you think that using a comparison like this (similar to what Park suggested) would be good enough for deducing the true original value? (I've thought of proposing a similar thing for BigDecimal,

ex: BigDecimal(0.1) => #

-=r
=end

#4 - 05/09/2009 07:42 PM - tadf (tadayoshi funaba)

=begin
to_r should provide exact conversion.
I think ruby may provide "rationalize" on common lisp or scheme.
but not yet.

=end

#5 - 05/16/2009 06:24 AM - nobu (Nobuyoshi Nakada)

=begin
Hi,

At Fri, 1 May 2009 21:12:52 +0900,
Roger Pack wrote in [ruby-core:23345]:

True--however the (current) code for String#to_s attempts to determine whether the floating point number "is the equivalent default for the rounded value" (i.e. if it round trips).

What about this?

Index: rational.c
===================================================================
--- rational.c (revision 23433)
+++ rational.c (working copy)
@@ -1286,4 +1286,5 @@ integer_to_r(VALUE self)
 }
#define 0
 static void
@@ @ -1286 +1286,5 @@ integer_to_r(VALUE self)
 }
 #endif

+#if 0
 static void
 float_decode_internal(VALUE self, VALUE *rf, VALUE *rn)
@@ @ -1299 +1300,4 @@ float_decode_internal(VALUE self, VALUE self)
 }
 #endif
 static VALUE
 float_decode(VALUE self)
@@ @ -1300 +1300,4 @@ float_decode(VALUE self)
 #endif

+#if FLT_RADIX == 2 && SIZEOF_BDIGITS * 2 * CHAR_BIT > DBL_MANT_DIG
```c
#define HAVE_LONG_LONG
#define define BDIGITDBL2NUM(x) ULL2NUM(x)
#define define BDIGITDBL2NUM(x) ULONG2NUM(x)
#endif
#endif
#define NEEDS_FDIV
static ID id_fdiv;
#define fun2(fdv)
#endif
+
+static VALUE
+float_r_round(double a, double f, int n)
+
{
  int i, r; +#ifdef BDIGITDBL2NUM
  BDIGIT DBL fn = (BDIGIT DBL)fabs(f);
  BDIGIT DBL d1 = (BDIGIT DBL)1 << n, d2 = d1;
  BDIGIT DBL rv = d1 % fn;
  VALUE b, d;
  if (rv < 10) {
    for (i = 1, r = (int)rv; i <= r; ++i) {
      if ((double)fn / -d2 != a) break;
      if (fn % (d1 = d2) == 0) break;
    }
  }
  else if ((rv = fn - rv) & rv < 10) {
    for (i = 1, r = (int)rv; i <= r; ++i) {
      if ((double)fn / +d2 != a) break;
      if (fn % (d1 = d2) == 0) break;
    }
  }
  b = BDIGITDBL2NUM(fn);
  d = BDIGITDBL2NUM(d1);
  if (f < 0) b = f_negate(b); +#else
  VALUE d2, fn, rv;
  VALUE b = rb_dbl2big(f);
  VALUE d = rb_big_pow(rb_uint2big(FLT_RADIX), INT2FIX(-n));
  if (FIXNUM_P(d)) {
    d = rb_uint2big(FIX2LONG(d));
  }
  d2 = d;
  fn = f_abs(b);
  rv = rb_big_modulo(d, fn);
  if (FIXNUM_P(rv) & (r = FIX2LONG(rv)) < 10) {
    for (i = 1; i <= r; ++i) {
      d2 = f_sub(d2, INT2FIX(1));
      if (RFLOAT_VALUE(f_fdiv(fn, d2)) != a) break;
      if (f_mod(fn, d = d2) == INT2FIX(0)) break;
    }
  }
  else if (FIXNUM_P(rv = f_sub(fn, rv)) & (r = FIX2LONG(rv)) < 10) {
    for (i = 1; i <= r; ++i) {
      d2 = f_add(d2, INT2FIX(1));
      if (RFLOAT_VALUE(f_fdiv(fn, d2)) != a) break;
      if (f_mod(fn, d = d2) == INT2FIX(0)) break;
    }
  } +#endif
  return rb_rational_new(b, d); +} + static VALUE float_to_r(VALUE self) {
  VALUE f, n;
  float_decode_internal(self, &f, &n);
  return f_mul(f, f_exp(lINT2FIX(FLT_RADIX), n));
  a = RFLOAT_VALUE(self);
  f = frexp(a, &n);
  f = ldexp(f, DBL_MANT_DIG);
```

8/6/2021
n -= DBL_MANT_DIG;
if (n <= DBL_MANT_DIG && f != 0) {
    return float_r_round(a, f, n);
}
return f_mul(rb_dbl2big(f), f_expt(INT2FIX(FLT_RADIX), INT2FIX(n)));

@@ -1569,4 +1640,7 @@ Init_Rational(void)
id_to_s = rb_intern("to_s");
id_truncate = rb_intern("truncate");
+#ifdef NEEDS_FDIV
    id_fdiv = rb_intern("fdiv");
+#endif
    ml = (long)(log(DBL_MAX) / log(2.0) - 1);

--

Nobu Nakada

#6 - 05/18/2009 11:15 AM - matz (Yukihiro Matsumoto)
=begin
Hi,
In message "Re: [ruby-core:23465] Re: [Feature #1408] 0.1.to_r not equal to (1/10)"
on Sat, 16 May 2009 06:23:53 +0900, Nobuyoshi Nakada nobu@ruby-lang.org writes:
|What about this?
Could you explain how this patch differs from the original?

matz.

=end

#7 - 05/18/2009 11:49 AM - nobu (Nobuyoshi Nakada)
=begin
Hi,
At Mon, 18 May 2009 11:15:16 +0900, Yukihiro Matsumoto wrote in [ruby-core:23487]:

Could you explain how this patch differs from the original?

Searches more reduceable numerator which can round trip. Since
it just tries the numerator only in very restricted condition,
better result may be achieved by trying also the denominator,
in other cases. In fact, the patch works for very simple
cases, e.g. 0.1 and (1.0/3.0), but doesn't for 0.24.

--

Nobu Nakada

=end

#8 - 07/14/2009 12:06 AM - yugui (Yuki Sonoda)
- Target version changed from 1.9.1 to 1.9.2
=begin
=end
Sorry to be late to the party on this one.

It is important to remember that a Float is always an approximation.

1.0 has to be understood as 1.0 +/- EPSILON, where the EPSILON is platform dependent. 1.0 is not more equal to 1 than to 1 + EPSILON/2. Indeed, there is no way to distinguish either when they are stored as floats.

To believe that Float#to_s loses data is wrong. If r.to_s returns "1.2", it implies that 1.2 is one of the values in the range of possible values for that floating number. It could have been 1.2000...0006. Or something else. There is no way to know, so #to_s chooses, wisely, to return the simplest value in the range.

There are many rationals that would be encoded as floats the same way. There is no magic way to know that the "exact" value was exactly 12/10 or 5403599627370496, or anything in between. All have the same representation as a float. There is no reason to believe that the missing (binary) decimals that couldn't be written in space allowed where all 0. Actually, there is reason to believe that they were probably non zero, because fractions that can not be expressed with a finite number of terms in their expansion in a given base all have a recurring expansion. I.e. if the significand does not end with a whole bunch of zeros (rational has finite expansion) then it probably ends with an infinite pattern (say 011011011 in binary, or 33333 in decimal).

For any given float, there is one and only one rational with the smallest denominator that falls in the range of its possible values. It is currently given by Number#rationalize, and I really do not understand why #to_r would return anything else.

I cannot see any purpose to any other fraction. Moreover, the current algorithm, which returns the middle of the range of possibilities, is platform dependent since the range of possibilities is platform dependent. That makes it even less helpful.

Is there an example where one would want 0.1.to_r to be 3602879701896397/36028797018963968 ?

Do we really think that 0.1.to_r to be 3602879701896397/36028797018963968 corresponds to the principle of least surprise?

Note that I'm writing that fraction but with a different native double encoding, the fraction would be different.
conversion from most decimal numbers, especially floats, and

calculations that drop digits.

You can do exact math in a limited range of operations, and the question
should be whether the approximation approach should overrule this exact
math range of use, especially considering that conversion back to
decimal could be done precisely, however, sometimes requiring a bunch
of digits.

1.0 has to be understood as 1.0 +/- EPSILON, where the EPSILON is platform
dependent. 1.0 is not more equal to 1 than to 1 + EPSILON/2. Indeed, there
is no way to distinguish either when they are stored as floats.

If what's stored in the Float is your precise result, you certainly
would not ask for precision reduction just because it could have been
the result of an imprecise calculation.

To believe that Float#to_s loses data is wrong.

I think there should be both a Float#to_s and Float#to_nearest_s. The
first would be precise, the second would output the "shortest" decimal
representation within ±EPSILON/2.

If r.to_s returns "1.2", it implies that 1.2 is one of the values in the
range of possible values for that floating number. It could have been
1.2000...0006. Or something else. There is no way to know, so #to_s chooses,
wisely, to return the simplest value in the range.

This is based on the assumption that no-one would ever care about
Float's precision.

There are many rationals that would be encoded as floats the same way. There
is no magic way to know that the "exact" value was exactly 12/10 or
5404319552844595/4503599627370496, or anything in between. All have the same
representation as a float. There is no reason to believe that the missing
(binary) decimals that couldn't be written in space allowed where all 0.
Actually, there is reason to believe that they were probably non zero,
because fractions that can not be expressed with a finite number of terms in
their expansion in a given base all have a recurring expansion. I.e. if the
significant does not end with a whole bunch of zeros (rational has finite
expansion) then it probably ends with an infinite pattern (say 011011011 in
binary, or 333333 in decimal).

For any given float, there is one and only one rational with the smallest
denominator that falls in the range of its possible values. It is currently
given by Number#rationalize, and I really do not understand why #to_r would
return anything else.

I cannot see any purpose to any other fraction. Moreover, the current algorithm,
which returns the middle of the range of possibilities, is platform dependent
since the range of possibilities is platform dependent. That makes it even less
helpful.

Is there an example where one would want 0.1.to_r to be
3602879701896397/36028797018963968 ?

If the binary/Float's representation of
3602879701896397/36028797018963968 is the real result of the
calculation? How do you know?

Do we really think that 0.1.to_r to be 3602879701896397/36028797018963968
corresponds to the principle of least surprise?

False assumption here. Using floats for exact decimal math already
violates POLS. Don’t blame the messenger, i.e. the converter back to
decimal, the only part of the game that could always be precise.

Note that I'm writing that fraction but with a different native double
encoding, the fraction would be different.
Sure. Great to have different levels of precision/imprecision from the computers.

And portability is not always the issue, otherwise there would have never been different native floating point precisions.

– Matthias

**#14 - 04/19/2010 05:12 PM - mwaechter (Matthias Wächter)**

Hello Marc-Andre,

On 19.04.2010 00:14, Marc-Andre Lafortune wrote:

I hope my dissent will not sound too harsh.

Not at all.

Arguing that 0.1.to_r should be 3602879701896397/36028797018963968 is the same as arguing that 0.1.to_s should outputs these 55 decimals.

Right, that’s my point. 0.1 as a Float has a precise meaning in binary as in decimal, so Float#to_s should keep those 55 decimals. That’s why I said that Float#to_nearest_s – choose a better name or an option to Float#to_s – should be created that does »what everyone expects« to_s to do.

The same applies to Float#to_r. It should be as precise as possible, which it is currently. The function that does »what everyone expects« should be Float#to_nearest_r in the same way as for the string representation.

For these reasons, the set S is of little interest to anybody.

The problem is that most people think that Floating point arithmetic is precise, which it is only for the the cases I described in my last mail.

What is interesting is the set of real numbers. Floating numbers are used to represent them approximately. To add to my voice, here are a couple of excerpts from the first links that come up on google (highlight mine):

"In computing, floating point describes a system for representing numbers that would be too large or too small to be represented as integers. Numbers are in general represented approximately to a fixed number of significant digits and scaled using an exponent."

http://en.wikipedia.org/wiki/Floating_point

"Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation.... Therefore the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. This rounding error is the characteristic feature of floating-point computation." source: http://docs.sun.com/source/806-3568/ncg_goldberg.html

That’s where the problem starts. Everyone thinks he can do exact math on a computer, and the only problem was the approximation of the binary representation of a real number, characterized by ±EPSILON/2. No, the real issue is the approximation of calculations which not only accumulates EPSILON with each calculation, but it can shift EPSILON to any order. Think of something trivial like (1E-40+0.1-0.1) returning 0.0 vs. (1E-40+0.3-0.2-0.1) returning -2.7E-17. There is no real math in floats.

One can go as far as saying that availability of math-like operators and math-like precedence in a programming language supports the expectations of real-number-like behavior and precision. But this is slightly off-topic, and in fact method calls for simple math are not doing any good to readability. Math-like operator precedence is different and something completely unnecessary in a programming language, IMHO.

Note that typing 0.1 in Ruby is a "calculation" which consists in finding the member of S closest to 1/10.

Your final question was: how do I know that the value someone is talking about is 0.1 and not 0.1000000000000000000000055511151231257827021181583404541015625 (or equivalently 3602879701896397/36028797018963968) ?

I call it common sense.

08/06/2021 7/9
It looks so obvious when we are talking about 0.1. If we talk about any other number with 80 digits, my point may become clearer.

What do you do if it's not 0.1 a.k.a. 0.10000000000000000555111512312578270211181583404541015625 but 0.09999999999999999779553950746896919152736663818359375 (the result of (0.3-0.2)? What's the difference for your argument? Now we will not get back the expected nearest 0.1 anyway without applying the actually required/expected rounding constraints.

If it's just about 0.1.to_r, i.e. converting from a decimal constant number to rational, use String#to_r.

Bottom line: Floats are not exact in terms of math, but they are exact in terms of computer-level implementation, implementing IEEE 754. We should respect the latter and help people deal with the former.

– Matthias

#15 - 04/20/2010 08:02 PM - tadf (tadayoshi funaba)

Why isn't Float#to_r simply calling Float#rationalize?

```ruby
a = 0.5337486539516013
b = 0.5337486539516012
a == b #=> false
a.to_r == a #=> true
a.rationalize == a #=> false
a.to_r == b #=> false
a.rationalize == b #=> true
```

actually, flonum is restricted rational number.
however, rationalize bends the value.
to_r is the simplest and the cheapest way, rationalize is not so.

moreover, various languages support exact conversion (e.g. CL, Scheme, Haskell, Squeak, Python).

#16 - 05/06/2010 12:37 PM - mrkn (Kenta Murata)

Float#rationalize is added again at r27503.
Please check that revision.

On 2010/05/06, at 7:23, Marc-Andre Lafortune wrote:

Maybe a kind Japanese reader can provide the gist of [ruby-dev:41061]
to explain why was Float#rationalize removed?

I would also appreciate opinions as to why it wouldn't be a net improvement if to_r used the rationalize algorithm and some other methods were provided for anyone wanting the value of the representation (e.g. Float#representation which would return [sign, mantissa, significand] and/or Float#representation_to_r would give the rational corresponding to the internal representation of that float)

--
Kenta Murata
OpenPGP FP = FA26 35D7 4F98 3498 0810 E0D5 F213 966F E9EB 0BCC
E-mail: mrkn@mrkn.jp
twitter: http://twitter.com/mrkn/
blog: http://d.hatena.ne.jp/mrkn/

#17 - 08/29/2011 09:32 AM - mrkn (Kenta Murata)
I close this ticket because the topic was too diverged.
Would you please make new tickets for the new version of ruby if anyone has objections.