# Ruby master - Feature #16470

## Issue with nanoseconds in Time#inspect

01/01/2020 03:04 PM - andrykonchin (Andrew Konchin)

| | |
|---|---|
| **Status:** | Closed |
| **Priority:** | Normal |
| **Assignee:** | matz (Yukihiro Matsumoto) |
| **Target version:** | |

**Description**

Ruby 2.7 added nanosecond representation to the return value of Time#inspect method.

Nanosecond is displayed as Rational as in the following example:

```
t = Time.utc(2007, 11, 1, 15, 25, 0, 123456.789)
t.inspect # => "2007-11-01 15:25:00 8483885939586761/68719476736000000 UTC"
```

The nanosecond value 8483885939586761/68719476736000000 can be expanded to 0.12345678900000001. This is different from the stored nanosecond:

```
t.nsec # => 123456789
t.strftime("%N") # => "123456789"
```

I assume it isn't expected, and will be fixed.

**Related issues:**

| | |
|---|---|
| Related to Ruby master - Bug #16445: Time#inspect shows a fractional number | **Closed** |
| Related to Ruby master - Feature #15958: Time#inspect with frac | **Closed** |
| Related to Ruby master - Bug #17025: `Time#ceil` does not work like `Rational... | **Closed** |

## History

**#1 - 01/01/2020 03:10 PM - znz (Kazuhiro NISHIYAMA)**

*- Related to Bug #16445: Time#inspect shows a fractional number added*

**#2 - 01/01/2020 03:11 PM - znz (Kazuhiro NISHIYAMA)**

*- Assignee set to matz (Yukihiro Matsumoto)*

**#3 - 01/01/2020 10:08 PM - Eregon (Benoit Daloze)**

I'd guess it's partly due to the Float itself losing precision.
But indeed it's quite unpretty. Maybe we should use #rationalize since that's closer to the Float#inspect output?

There is a usability problem here, it's basically impossible to read the output of Time#inspect in such a case, even though the input was readable.

```
123456.789.to_r
# => (8483885939586761/68719476736)

123456.789.rationalize
# => (123456789/1000)

Time.utc(2007, 11, 1, 15, 25, 0, 123456.789).inspect
# => "2007-11-01 15:25:00 8483885939586761/68719476736000000 UTC"
```

This works as expected (using r for making it an exact Rational):

```
p Time.utc(2007, 11, 1, 15, 25, 0, 123456.789r)
# => 2007-11-01 15:25:00.123456789 UTC
```

**#4 - 05/29/2020 08:20 PM - jeremyevans0 (Jeremy Evans)**

*- Backport deleted (2.5: UNKNOWN, 2.6: UNKNOWN)*

*- ruby -v deleted (2.7)*

*- Tracker changed from Bug to Feature*

Even though this appears to be a bug, it's actually intentional behavior, introduced in 5208c431bef3240eb251f5da23723b324431a98e, and there are tests and specs for it. So changing this behavior would be a feature request.

I think the behavior of displaying the rational is less useful, and the vast majority of Ruby programmers would want:

```
Time.utc(2007, 11, 1, 15, 25, 0, 123456.789).inspect
# => "2007-11-01 15:25:00.123456789 UTC"
```

I've added a pull request to implement this: https://github.com/ruby/ruby/pull/3160

One disadvantage of the pull request's approach is that two Time objects that are not equal will have inspect output that is equal. An alternative approach that wouldn't have that issue would be to rationalize all float usecs given as input, as Eregon (Benoit Daloze) mentioned, but that could possibly have implications beyond inspect. With that approach, you would still end up with weird inspect output if providing a rational such as 8483885939586761/68719476736000000r as usec input.

#### #5 - 05/30/2020 12:42 AM - sawa (Tsuyoshi Sawada)

*- Description updated*

#### #6 - 06/18/2020 11:24 AM - shyouhei (Shyouhei Urabe)

Note that the cryptic output does not happen for Time.now.utc.

```
Time.now.utc.inspect # => "2020-06-18 11:12:44.354669166 UTC"
```

Nor when you pass 123456.789r for Time.utc.

```
Time.utc(2007, 11, 1, 15, 25, 0, 123456.789r) # => "2007-11-01 15:25:00.123456789 UTC"
```

So no, as Eregon (Benoit Daloze) pointed out, this is not an issue of Time#inspect. This is due to the OP is using 123456.789 for some reason not shown in the report. Time#inspect is just trying to express as much as it can.

Is this usage (passing Float instances instead of Rationals) major in some area? If so we might want to improve Time.utc's API, not Time#inspect.

#### #7 - 06/18/2020 01:26 PM - Eregon (Benoit Daloze)

*- Related to Feature #15958: Time#inspect with frac added*

#### #8 - 06/22/2020 03:55 PM - jeremyevans0 (Jeremy Evans)

I've added an alternative approach, which uses Float#rationalize for all Float conversions in Time, except for Time.at (one test depends on Time.at(float).to_f == float): https://github.com/ruby/ruby/pull/3248

#### #9 - 06/22/2020 05:24 PM - Eregon (Benoit Daloze)

One consideration here is performance.
For instance Time.at(Float) is quite slow, due to going through Rational, etc.
Could you measure the other methods you changed to see how they perform compared to before?

#### #10 - 06/22/2020 06:05 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-9:

> One consideration here is performance.
> For instance Time.at(Float) is quite slow, due to going through Rational, etc.
> Could you measure the other methods you changed to see how they perform compared to before?

Well, Time converts Float to Rational both with and without the patch (the values are passed through num_exact, which is where the conversion takes place). It is just a question of whether to use to_r or rationalize for the conversion (neither can be considered more accurate, since floating point numbers are inexact). rationalize looks slightly faster, so this should slightly increase performance, not decrease it.

Here's an example with 2.7:

```
$ ruby -v
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-openbsd]
$ ruby -r benchmark -ve 'eval "def a; #{"Time.utc(2007, 11, 1, 15, 25, 0, 123456.789.to_r);"*100000} end"; put
s Benchmark.measure{a}'
  1.330000   0.030000   1.360000 (  1.426325)
$ ruby -r benchmark -ve 'eval "def a; #{"Time.utc(2007, 11, 1, 15, 25, 0, 123456.789.rationalize);"*100000} en
d"; puts Benchmark.measure{a}'
  1.230000   0.010000   1.240000 (  1.255797)
$ ruby -r benchmark -ve 'eval "def a; #{"Time.utc(2007, 11, 1, 15, 25, 0, 123456.789);"*100000} end"; puts Ben
chmark.measure{a}'
  1.350000   0.010000   1.360000 (  1.412875)
```

I tested with the patch and the test for the literal float was close to the rationalize value, not the to_r value. So the patch makes the code faster, not slower.

FWIW, it's significantly faster to pass a literal rational as opposed to converting a float to a rational or passing a literal float, both with and without the patch:

```
$ ruby -r benchmark -ve 'eval "def a; #{"Time.utc(2007, 11, 1, 15, 25, 0, 123456.789r);"*100000} end"; puts Be
nchmark.measure{a}'
  0.350000   0.030000   0.380000 (  0.496827)
```

**#11 - 06/23/2020 02:58 AM - mame (Yusuke Endoh)**

Interesting, I can reproduce the performance difference:

```
$ time ./miniruby -e '100000.times { Time.utc(2007, 11, 1, 15, 25, 0, 123456.789.to_r) }'

real    0m0.633s
user    0m0.622s
sys     0m0.011s

$ time ./miniruby -e '100000.times { Time.utc(2007, 11, 1, 15, 25, 0, 123456.789.rationalize) }'

real    0m0.607s
user    0m0.587s
sys     0m0.020s
```

Note that #to_r is 10 times faster than #rationalize:

```
$ time ./miniruby -e '100000.times { 123456.789.to_r }'

real    0m0.059s
user    0m0.039s
sys     0m0.020s

$ time ./miniruby -e '100000.times { 123456.789.rationalize }'

real    0m0.598s
user    0m0.587s
sys     0m0.010s
```

I think that this is because #to_r produces relatively big denominator, which makes addition slower during Time.utc calculation.

(I have no opinion about the Time.utc issue itself.  Sorry.)

**#12 - 07/16/2020 09:32 AM - mame (Yusuke Endoh)**

We need to understand the use case precisely.  Does OP want to pass a general Float value to Time.utc?  Or does he just want to specify nanosecond?

I think of no practical use case for the former (passing a general/calculated Float).  If we need to specify nanosecond, I don't think that Float is a good API for that.  Time.utc(2007, 11, 1, 15, 25, 0, nanosecond: 123456789) or something is better.

In addition, the following point is wrong.

> The nanosecond value 8483885939586761/68719476736000000 can be expanded to 0.12345678900000001.

A correct expansion is 0.123456789000000004307366907596588134765625.  So, which is better?

```
t.inspect # => "2007-11-01 15:25:00 8483885939586761/68719476736000000 UTC"
t.inspect # => "2007-11-01 15:25:00.123456789000000004307366907596588134765625 UTC"
```

Personally, I prefer the latter to the former because decimal is much easier to understand.

However, I'm afraid if the expansion might be very long. Truncation is a possible option, of course. But, it will break the original reason why the fraction part is added in Time#inspect (#15958):

> But recently we encounters some troubles the comparison of Time objects whose frac parts are different.

So, naive truncation may bring the same troubles again. I guess nanosecond (nine digits after the decimal point) would be enough in many use cases, but I'm not 100% sure.

**#13 - 07/16/2020 05:25 PM - jeremyevans0 (Jeremy Evans)**

mame (Yusuke Endoh) wrote in #note-12:

> I guess nanosecond (nine digits after the decimal point) would be enough in many use cases, but I'm not 100% sure.

I don't think support for fractional nanoseconds is important.  Does anyone have a use case for fractional nanoseconds?  If not, I think it would be best to change Time to just store nanoseconds as integer instead of a rational.  However, that's definitely a more involved change.

**#14 - 07/16/2020 05:47 PM - Eregon (Benoit Daloze)**

I also don't see much use for fractional nanoseconds.
clock_gettime() never has higher resolution than nanoseconds.

In fact both JRuby and TruffleRuby always round to an integer number of nanonseconds, because java.time supports nanoseconds but not more fine-grained than that (since nothing could provide such precision).
https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html

**#15 - 07/17/2020 09:45 PM - jeremyevans0 (Jeremy Evans)**

*- Related to Bug #17025: `Time#ceil` does not work like `Rational#ceil` or `Float#ceil` added*

**#16 - 07/20/2020 03:15 AM - akr (Akira Tanaka)**

There are several examples time needs more than nanoseconds.

- SQLite supports arbitrary number of digits in fractional seconds https://www.sqlite.org/lang_datefunc.html
- POSIX pax format supports arbitrary number of digits in fractional seconds http://pubs.opengroup.org/onlinepubs/007904875/utilities/pax.html#tag_04_100_13_05
- EXIF supports arbitrary number of digits in fractional seconds http://www.exif.org/Exif2-1.PDF
- NTPv4's 128-bit date format has 64-bit fraction field : 2**(-64) second https://tools.ietf.org/html/rfc5905#section-6
- FreeBSD has struct bintime which can represent 2**(-64) second https://svnweb.freebsd.org/base/head/sys/sys/time.h?view=markup&pathrev=363193#l55 It is visible from userland for datagram timestamp https://www.freebsd.org/cgi/man.cgi?query=setsockopt Ruby supports it as converting bintime to Time object
- DB2 supports fractional seconds up to 12 digits in timestamp (12 digits represents picosecond) https://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.sql.ref.doc/doc/r0008474.html

**#17 - 07/20/2020 03:33 AM - akr (Akira Tanaka)**

*- Status changed from Open to Feedback*

As others already pointed,
the original description of this issue misunderstand the actual Ruby behavior.

Time.utc(2007, 11, 1, 15, 25, 0, 123456.789) creates a Time object which has
0.123456789000000004307366907596588134765625 as a fractional second
because the exact value of float 123456.789 is
123456.789000000004307366907596588134765625.

I think the description "This is different from the stored nanosecond: ..." is based on misunderstanding of actual Ruby behavior.

So, it is difficult to determine actual problem.
We'd like to hear.

**#18 - 07/20/2020 07:16 PM - jeremyevans0 (Jeremy Evans)**

akr (Akira Tanaka) wrote in #note-16:

> There are several examples time needs more than nanoseconds.
>
> - SQLite supports arbitrary number of digits in fractional seconds https://www.sqlite.org/lang_datefunc.html

SQLite ignores values after the millisecond when converting:

```
sqlite> SELECT CAST(strftime('%f', '2020-10-20 11:12:13.1237') AS NUMERIC);
13.124
```

> - POSIX pax format supports arbitrary number of digits in fractional seconds http://pubs.opengroup.org/onlinepubs/007904875/utilities/pax.html#tag_04_100_13_05

This is a file archive format.  How many filesystems have greater than nanosecond resolution?

> - EXIF supports arbitrary number of digits in fractional seconds http://www.exif.org/Exif2-1.PDF

Similarly, what camera supports greater than nanosecond resolution?

- NTPv4's 128-bit date format has 64-bit fraction field : 2**(-64) second https://tools.ietf.org/html/rfc5905#section-6
- FreeBSD has struct bintime which can represent 2**(-64) second https://svnweb.freebsd.org/base/head/sys/sys/time.h?view=markup&pathrev=363193#l55 It is visible from userland for datagram timestamp https://www.freebsd.org/cgi/man.cgi?query=setsockopt Ruby supports it as converting bintime to Time object
- DB2 supports fractional seconds up to 12 digits in timestamp (12 digits represents picosecond) https://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.sql.ref.doc/doc/r0008474.html

These seem to be better reasons to support sub-nanosecond resolution.  I think either storing picoseconds or storing sec fraction as 64-bit integer are better approaches than storing a rational.  However, either change would be very invasive, and it seems unlikely to be worth the effort.

As rational is used internally, it seems reasonably for inspect output to include rational.  If a user wants to specific nanosecond, they should provide a rational instead of a float.  So I think this feature request can be closed.

Note that sub-nanosecond resolution should be considered Ruby-implementation-specific behavior, since JRuby and TruffleRuby support nanosecond, and mruby supports microsecond.

**#19 - 07/28/2020 08:17 PM - jeremyevans0 (Jeremy Evans)**

*- Status changed from Feedback to Closed*