Inconsistent definitions of Complex#<=> and Numeric#<=> with others

Object#<=> says "Returns 0 if obj and other are the same object or obj == other". 
https://ruby-doc.org/core-3.1.2/Object.html#method-i-3C-3D-3E

However, neither Complex#<=> nor Numeric#<=> satisfies that definition.

```ruby
num1 = Complex(0, 42)
num2 = Complex(0, 42)
p num1.equal?(num2) #=> false
p num1 == num2 #=> true

# using Complex#<=>
p num1 <=> num2 #=> nil

# using Numeric#<=>
Complex.remove_method(:<=>)
p num1 <=> num2 #=> nil

# using Object#<=> (Kernel#<=>)
Numeric.remove_method(:<=>)
p num1 <=> num2 #=> 0
```

Complex#<=> has another problem that it does not coerce numeric objects while Integer#<=> and Float#<=> do. This prevents users from adding yet another complex class having #<=>.

Here is my proposal of Complex#<=> behavior (in Ruby).
This considers #15857, complex numbers are comparable when their imaginary parts are 0.

```ruby
class Complex
  def <=>(other)
    return (self == other ? 0 : nil) if self.imag != 0

    if other.kind_of?(Complex)
      if other.imag == 0
        return self.real <=> other.real
      else
        return nil
      end
    elsif other.kind_of?(Numeric) && other.real?
      return self.real <=> other
    elsif other.respond_to?(:coerce)
      num1, num2 = other.coerce(self)
      return num1 <=> num2
    else
      return nil
    end
  end
end
```

Associated revisions
History

#1 - 07/24/2022 03:49 AM - msnm (Masahiro Nomoto)

I fix the last code as follows:

```ruby
class Complex
  def <=>(other)
    return (self == other ? 0 : nil) if self.imag != 0
    if other.kind_of?(Complex)
      if other.imag == 0
        return self.real <=> other.real
      else
        return nil
      end
    end
    return self.real <=> other
  end
end
```

#2 - 07/26/2022 08:05 AM - mame (Yusuke Endoh)

However, neither Complex# <=> nor Numeric# <=> satisfies that definition.

Complex# <=> is a different method from Object# <=>, so I don't see that as a problem in itself. Some Ruby core methods don't necessarily follow the Liskov Substitution Principle. I think this is one example of such cases that we favors mathematical intuition over the principle. If you are facing any practical problem, please elaborate the situation.

Complex# <=> has another problem that it does not coerce numeric objects while Integer# <=> and Float# <=> do. This prevents users from adding yet another complex class having # < >.

I understand this as follows.

```ruby
class MyInteger
  def initialize(n)
    @n = n
  end

  def coerce(obj)
    [obj, @n]
  end
end
```

```
p 1 <=> MyInteger.new(2) #=> -1
p 1+0i <=> MyInteger.new(2) #=> expected: -1, actual: nil
```

#3 - 07/26/2022 01:39 PM - msnm (Masahiro Nomoto)

@mame (Yusuke Endoh)

I have a motivation to create a "quaternion" class which is highly interoperable with other built-in numeric classes. (gem: quaternion_c2)

I recently noticed that there was Complex# <=> in Ruby >= 2.7. I want to implement Quaternion# <=> though I believe few people compare complex numbers by #<=>. However, Complex# <=> feels odd and I can't define Quaternion# <=> well.

Complex# <=> is a different method from Object# <=>, so I don't see that as a problem in itself. Some Ruby core methods don't necessarily follow the Liskov Substitution Principle. I think this is one example of such cases that we favors mathematical intuition over the principle. If you are facing any practical problem, please elaborate the situation.

Thanks. I changed my understanding: "(non-real) complex numbers are not comparable, then Complex# <=> for them always returns nil even if self.equal?(other)".

```
Complex::I <=> Complex::I #=> nil
```
How about Numeric#<=>? I think num1 == num2 (equivalency) is more intuitive than num1.equal?(num2) (identity) in mathematical contexts.

Another idea is that the method always returns nil in order to tell "the custom numeric class may not be comparable".

This prevents users from adding yet another complex class having #<=>.

I understand this as follows.

That's right. Moreover, MyInteger#<=> will produce asymmetric (not antisymmetric) behaviors.

class MyInteger
  def initialize(n)
    @n = n
  end

  def coerce(obj)
    [obj, @n]
  end

  def <=>(obj)
    if obj.kind_of?(MyInteger)
      @n <=> obj.instance_variable_get(:@n)
    else
      @n <=> obj
    end
  end
end

my_int = MyInteger.new(2)
p 1 + 0i <=> my_int #=> nil (expected: -1)
p my_int <=> 1 + 0i #=> 1

#4 - 08/22/2022 03:33 AM - nobu (Nobuyoshi Nakada)
I agree that Complex#<=> should coerce the argument as well as other methods/classes. https://github.com/ruby/ruby/pull/6269

#5 - 08/22/2022 08:16 AM - nobu (Nobuyoshi Nakada)
- Status changed from Open to Closed

Applied in changeset git|d5f50463c2b5c5263aa45c58f3f4ec73de886d5.

[Bug #18937] Coerce non-Numeric into Complex at comparisons

#6 - 08/23/2022 11:00 PM - msnm (Masahiro Nomoto)
nobu (Nobuyoshi Nakada) wrote in #note-4:

I agree that Complex#<=> should coerce the argument as well as other methods/classes. https://github.com/ruby/ruby/pull/6269

Thanks @nobu (Nobuyoshi Nakada).
But #coerce is not called when a custom numeric class has def real? = false following Complex#real?.

puts RUBY_DESCRIPTION
# ruby 3.2.0dev (2022-08-22T03:26:43Z :detached: d5f50463c2) [x86_64-linux]
class MyInteger < Numeric
  def initialize(n) = (@n = n)
  def coerce(other)
    puts "MyInteger#coerce is called."
    [other, @n]
  end
end
class MyComplex < Numeric
  def initialize(n) = (@n = Complex(n))
  def real? = false
  def coerce(other)
    puts "MyComplex#coerce is called."
  end
end

08/24/2022
p Complex(1) <=> MyInteger.new(2)
# MyInteger#coerce is called.
#=> -1

p Complex(1) <=> MyComplex.new(2)
#=> nil

# FYI: Complex#+ works fine.
p Complex(1) + MyComplex.new(2)
# MyComplex#coerce is called.
#=> (3+0i)