# What are the Ruby Gotchas a newbie should be warned about? [closed]

Asked 16 years ago Modified 9 years, 6 months ago Viewed 16k times



109









As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 12 years ago.

The community reviewed whether to reopen this question 2 years ago and left it closed:

Original close reason(s) were not resolved

I have recently learned the Ruby programming language, and all in all it is a good language. But I was quite surprised to see that it was not as simple as I had expected. More precisely, the "rule of least-surprise" did not seem very respected to me (of course this is quite subjective). For example:

```
x = true and false
puts x # displays true!
```

and the famous:

```
puts "zero is true!" if 0 # zero is true!
```

What are the other "Gotchas" you would warn a Ruby newbie about?

ruby

Share

edited Apr 6, 2012 at 16:53

community wiki 4 revs, 4 users 100% MiniQuark

Improve this question

Follow

@phrases.insert(0, p) OK @phrases.insert(p) NOTHING happens @phrases<<p # OK − Anno2001 Feb 20, 2013 at 13:36 ✓

why does true and false return true? - Jürgen Paul Feb 28, 2013 at 2:40

- Because "x = true and false" is actually interpreted as "(x = true) and false". It's a matter of operator precedence: "and" has a lower priority than "=". Most other languages have the reverse priority, I don't know why they chose this order in Rails, I find it very confusing. If you want the "normal" behavior, simply type "x = (true and false)", then x will be false.
  - MiniQuark Feb 28, 2013 at 14:00 /
- Another solution is to use "&&" and "||" instead of "and" and "or": they behave as expected. For example: "x = true && false" results in x being false. – MiniQuark Feb 28, 2013 at 14:08

"The principle of least surprise means principle of least my surprise." from en.wikipedia.org/wiki/Ruby (programming language)#Philosophy The same goes for Python. I had similar quote about Python's creator but I forget where it was. - Darek Nedza May 28, 2014 at 15:55

#### 25 Answers

Sorted by:

Highest score (default)

**\$** 

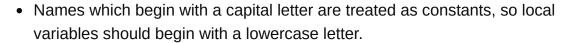


#### Wikipedia Ruby gotchas

62

#### From the article:







• The characters \$ and @ do not indicate variable data type as in Perl, but rather function as scope resolution operators.



- To denote floating point numbers, one must follow with a zero digit (99.0) or an explicit conversion (99.to\_f). It is insufficient to append a dot (99.), because numbers are susceptible to method syntax.
- Boolean evaluation of non-boolean data is strict: 0, "" and [] are all evaluated to true. In C, the expression 0 ? 1 : 0 evaluates to 0 (i.e. false). In Ruby, however, it yields 1, as all numbers evaluate to true; only nil and false evaluate to false. A corollary to this rule is that Ruby methods by convention for example, regular-expression searches — return numbers, strings, lists, or other non-false values on success, but nil on failure (e.g., mismatch). This convention is also used in Smalltalk, where only the special objects true and false can be used in a boolean expression.
- Versions prior to 1.9 lack a character data type (compare to C, which provides type char for characters). This may cause surprises when slicing strings: "abc" [0] yields 97 (an integer, representing the ASCII code of the first character in the string); to obtain "a" use "abc"[0,1] (a substring of length 1) or "abc" [0].chr.
- The notation statement until expression, unlike other languages' equivalent statements (e.g. do { statement } while (not(expression)); in C/C++/...),





actually never runs the statement if the expression is already true. This is because statement until expression is actually syntactic sugar over

```
until expression
statement
end
```

, the equivalent of which in C/C++ is while (not(expression)) statement; just like statement if expression is an equivalent to

```
if expression
  statement
end
```

However, the notation

```
begin
statement
end until expression
```

in Ruby will in fact run the statement once even if the expression is already true.

Because constants are references to objects, changing what a constant refers to generates a warning, but modifying the object itself does not. For example,
 Greeting << "world!" if Greeting == "Hello" does not generate an error or warning. This is similar to final variables in Java, but Ruby does also have the functionality to "freeze" an object, unlike Java.</li>

Some features which differ notably from other languages:

- The usual operators for conditional expressions, and and or, do not follow the normal rules of precedence: and does not bind tighter than or. Ruby also has expression operators || and && which work as expected.
- def inside def doesn't do what a Python programmer might expect:

```
def a_method
    x = 7
    def print_x; puts x end
    print_x
end
```

This gives an error about  $\times$  not being defined. You need to use a Proc.

#### Language features

 Omission of parentheses around method arguments may lead to unexpected results if the methods take multiple parameters. The Ruby developers have stated that omission of parentheses on multi-parameter methods may be disallowed in future Ruby versions; the current (November 2007) Ruby interpreter throws a warning which encourages the writer not to omit (), to avoid ambiguous meaning of code. Not using () is still common practice, and can be especially nice to use Ruby as a human readable domain-specific programming language itself, along with the method called <code>method\_missing()</code>.

Share edited May 7, 2012 at 15:54 community wiki
Improve this answer 4 revs, 4 users 54%
Andy
Follow

1 Ruby 1.9 lacks character data type, too. In 1.8, index operator returned a Fixnum; in 1.9, it is equivalent to slicing an one-character string. – Catherine Aug 1, 2011 at 1:27



Newbies will have trouble with **equality methods**:

39

• **a** == **b** : checks whether a and b are equal. This is the most useful.



• a.eql? b : also checks whether a and b are equal, but it is sometimes more strict (it might check that a and b have the same type, for example). It is mainly used in Hashes.



- a.equal? b : checks whether a and b are the same object (identity check).
- **a === b** : used in case statements (I read it as "a matches b").

These examples should clarify the first 3 methods:

```
a = b = "joe"

a==b  # true
a.eql? b  # true
a.equal? b # true (a.object_id == b.object_id)

a = "joe"
b = "joe"

a==b  # true
a.eql? b # true
a.equal? b # false (a.object_id != b.object_id)

a = 1
b = 1.0

a==b  # true
a.eql? b # false (a.class != b.class)
a.equal? b # false
```

Note that **==**, **eql?** and **equal?** should always be symmetrical: if a==b then b==a.

Also note that == and eql? are both implemented in class Object as aliases to equal?, so if you create a new class and want == and eql? to mean something else

than plain identity, then you need to override them both. For example:

```
class Person
   attr_reader name
   def == (rhs)
      rhs.name == self.name # compare person by their name
   end
   def eql? (rhs)
      self == rhs
   end
   # never override the equal? method!
end
```

The **===** method behaves differently. First of all it is **not** symmetrical (a===b does **not** imply that b===a). As I said, you can read a===b as "a matches b". Here are a few examples:

```
# === is usually simply an alias for ==
"joe" === "joe" # true
"joe" === "bob" # false

# but ranges match any value they include
(1..10) === 5  # true
(1..10) === 19  # false
(1..10) === (1..10) # false (the range does not include itself)

# arrays just match equal arrays, but they do not match included values!
[1,2,3] === [1,2,3] # true
[1,2,3] === 2  # false

# classes match their instances and instances of derived classes
String === "joe" # true
String === "joe" # true
String === 1.5  # false (1.5 is not a String)
String === String # false (the String class is not itself a String)
```

The **case** statement is based on the **===** method:

```
case a
  when "joe": puts "1"
  when 1.0 : puts "2"
  when (1..10), (15..20): puts "3"
  else puts "4"
end
```

is equivalent to this:

```
if "joe" === a
  puts "1"
elsif 1.0 === a
  puts "2"
elsif (1..10) === a || (15..20) === a
  puts "3"
else
```

```
puts "4"
end
```

If you define a new class whose instances represent some sort of container or range (if it has something like an **include?** or a **match?** method), then you might find it useful to override the **===** method like this:

```
class Subnet
  [...]
  def include? (ip_address_or_subnet)
    [...]
  end
  def === (rhs)
    self.include? rhs
  end
end

case destination_ip
  when white_listed_subnet: puts "the ip belongs to the white-listed subnet"
  when black_listed_subnet: puts "the ip belongs to the black-listed subnet"
  [...]
end
```

Share edited Dec 25, 2008 at 12:19 community wiki Improve this answer 2 revs

Follow

1 Also: a = 'строка'; b = 'строка'; p a == b; a = a.force\_encoding 'ASCII-8BIT'; b = b.force\_encoding 'UTF-8'; p a == b; p a === b; p a.eql? b; p a.equal? b − Nakilon Aug 29, 2010 at 14:38 ✓

MiniQuark



 Monkey patching. Ruby has open classes, so their behaviour can be dynamically changed at runtime...



Objects might <u>respond to undefined methods</u> if <u>method\_missing</u> or <u>send</u> has been overridden. This exploits Ruby's message-based method invocation. <u>Rails' ActiveRecord</u> system uses this to great effect.



Share edited Dec 16, 2008 at 22:01 community wiki

Improve this answer 3 revs

Dan Vinton

Follow



The following code surprised me. I think it's a dangerous gotcha: both easy to run into, and hard to debug.



1

```
(1..5).each do |number|
  comment = " is even" if number%2==0
  puts number.to_s + comment.to_s
end
```

#### This prints:

```
1
2 is even
3
4 is even
5
```

But if I just add comment = anything before the block...

```
comment = nil
(1..5).each do |number|
  comment = " is even" if number%2==0
  puts number.to_s + comment.to_s
end
```

#### Then I get:

```
1
2 is even
3 is even
4 is even
5 is even
```

Basically, when a variable is only defined inside a block, then it is destroyed at the end of the block, and then it gets reset to nil upon every iteration. That's usually what you expect. But if the variable is defined before the block, then the outer variable is used inside the block, and its value is therefore persistent between iterations.

One solution would be to write this instead:

```
comment = number%2==0 ? " is even" : nil
```

I think a lot of people (including me) tend to write "a = b if c" instead of "a = (c? b : nil)", because it's more readable, but obviously it has side-effects.

Share

answered Jul 13, 2010 at 19:01

community wiki MiniQuark

Improve this answer

- You might just as well shadow the outher scope variable by (1..5) do |number;comment| .....

  Read here <a href="mailto:stackoverflow.com/questions/1654637/...">stackoverflow.com/questions/1654637/...</a> Özgür Jul 13, 2010 at 21:14
- 7 This seems logical to me. This scoping is typical of other languages, it is just the syntax that is different. -g. Jul 27, 2011 at 14:38

However, you can write a = (b if c) to get the desired effect, without the ternary. This is because b if c evaluates to nil if c is falsey. — Cameron Martin Aug 18, 2014 at 15:39



When calling super with no arguments, the overridden method is actually called with the same arguments as the overriding method.

**17** 







```
class A
  def hello(name="Dan")
    puts "hello #{name}"
  end
end

class B < A
  def hello(name)
    super
  end
end

B.new.hello("Bob") #=> "hello Bob"
```

To actually call super with no arguments, you need to say super().

Share

answered Dec 17, 2008 at 8:27

community wiki Daniel Lucraft

Improve this answer

Follow

If B#hello has name = 42 before the super, then it says "hello 42". – Andrew Grimm Nov 16, 2010 at 0:15



Blocks and methods return the value of the last line by default. Adding puts statements to the end for debugging purposes can cause unpleasant side effects

**15** 



Share
Improve this answer

answered Jun 11, 2010 at 14:11

community wiki Andrew Grimm







<u>Inheritence plays no part in determining method visibility</u> in Ruby.

13

Share

answered Dec 16, 2008 at 21:06

community wiki John Topley



Follow

Improve this answer



43

Wow, that's one Gotcha I did not know about. Thanks! - MiniQuark Dec 16, 2008 at 21:47

It does in case of protected methods, in a rather complicated way. - taw Jan 11, 2009 at 2:56



I had a lot of trouble understanding class variables, class attributes and class methods. This code might help a newbie:

11





```
class A
 @@classvar = "A1"
  @classattr = "A2"
  def self.showvars
    puts "@@classvar => "+@@classvar
    puts "@classattr => "+@classattr
  end
end
A.showvars
 # displays:
 # @@classvar => A1
  # @classattr => A2
class B < A
  @@classvar = "B1"
  @classattr = "B2"
end
B. showvars
 # displays:
  # @@classvar => B1
  # @classattr => B2
A. showvars
 # displays:
  # @@classvar => B1  #Class variables are shared in a class hierarchy!
  # @classattr => A2 #Class attributes are not
```

Share

answered Dec 17, 2008 at 10:13

community wiki MiniQuark

Improve this answer

1 Yes, class variables can be tricky. I think most experienced Rubyists would say that it is wise to avoid them, since there are usually other ways to solve a problem without them. Some language enthusiasts would even say that Ruby's class variables are poorly designed at a language level. − David J. Jul 7, 2010 at 19:00 ▶



one thing i learned was to use the operator ||= carefully. and take special care if you are dealing with booleans. i usually used a ||= b as a catch all to give 'a' a default value if everything else failed and 'a' remained nil. but if a is false and b is true, then a will be assigned true.



8

Share answered Dec 18, 2008 at 21:04 community wiki karina



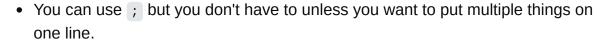
Follow

You can use a = b if a.nil? or @a = b unless defined?(@a) . — Andrew Grimm Feb 2, 2010 at 22:04



- Blocks are really important to understand, they're used everywhere.
- You don't need parentheses around method parameters. Whether you use them or not is up to you. <u>Some say you should always use them.</u>







Share

Improve this answer

edited Oct 9, 2011 at 22:17

community wiki 2 revs, 2 users 94% dylanfm

Follow

end

If you don't plan to go beyond Ruby 1.8.6 then ignore parens as much as you like. Otherwise, you're probably better off using them. – Mike Woodhouse Dec 17, 2008 at 9:23



I had trouble with mixins which contain instance methods **and** class methods. This code might help a newbie:

7



module Displayable
 # instance methods here
 def display
 puts name
 self.class.increment\_displays



```
def self.included(base)
   # This module method will be called automatically
   # after this module is included in a class.
   # We want to add the class methods to the class.
   base.extend Displayable::ClassMethods
 end
 module ClassMethods
   # class methods here
   def number_of_displays
     @number_of_displays # this is a class attribute
   def increment_displays
     @number_of_displays += 1
   end
    def init_displays
     @number_of_displays = 0
   end
   # this module method will be called automatically
   # after this module is extended by a class.
   # We want to perform some initialization on a
   # class attribute.
   def self.extended(base)
     base.init_displays
   end
 end
end
class Person
 include Displayable
 def name; @name; end
 def initialize(name); @name=name; end
end
puts Person.number_of_displays # => 0
john = Person.new "John"
john.display # => John
puts Person.number_of_displays # => 1
jack = Person.new "Jack"
jack.display # => Jack
puts Person.number_of_displays # => 2
```

At first, I thought I could have modules with both instance methods **and** class methods by simply doing this:

```
module Displayable
  def display
    puts name
    self.class.increment_displays
  end
  def self.number_of_displays # WRONG!
    @number_of_displays
  end
  [...]
end
```

Unfortunately, method *number\_of\_displays* will never be included or extended because it is a "module class method". Only "module instance methods" can be

included into a class (as instance methods) or extended into a class (as class methods). This is why you need to put your mixin's instance methods into a module, and your mixin's class methods into another module (you usually put the class methods into a "ClassMethods" submodule). Thanks to the *included* magic method, you can make it easy to include both instance methods and class methods in just one simple "include Displayable" call (as shown in the example above).

This mixin will count each display on a *per-class* basis. The counter is a class attribute, so each class will have its own (your program will probably fail if you derive a new class from the Person class since the @number\_of\_displays counter for the derived class will never be initialized). You may want to replace @number\_of\_displays by @@number\_of\_displays to make it a global counter. In this case, each class hierarchy will have its own counter. If you want a global and unique counter, you should probably make it a module attribute.

All of this was definitely not intuitive for me when I started with Ruby.

I still can't figure out how to cleanly make some of these mixin methods private or protected though (only the *display* and *number\_of\_displays* method should be included as public methods).

Share

edited Dec 17, 2008 at 11:45

community wiki

Improve this answer

2 revs MiniQuark

Follow



## Pay attention to the Range notation.



(At least, pay more attention than I initially did!)



There is a difference between **0..10** (two dots) and **0...10** (three dots).



http://www.ruby-doc.org/core/classes/Range.html



I enjoy Ruby a great deal. But this dot-dot versus dot-dot thing bugs me. I think that such a subtle dual-syntax "feature" that is:

- easy to mistype, and
- easy to miss with your eyes while glancing over the code

should not be able to cause devastating off-by-one bugs in my programs.

Share

edited Jun 20, 2020 at 9:12

community wiki

Improve this answer

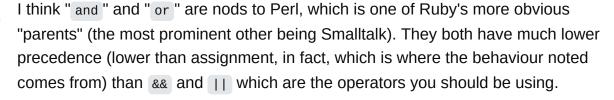
2 revs que que

Not much different from for (i=0; i<max; i++) and for (i=0; i<=max; i++) -g. Jul 27, 2011 at 14:37

I been trying to find what out what's the difference between 0..10 and 0...10. – Luis D Urraca Apr 5, 2012 at 9:22



6





Other things to be aware of that aren't immediately obvious:



You don't really call methods/functions, although it kinda looks that way. Instead, as in Smalltalk, you send a message to an object. So method\_missing is really more like message not understood.

```
some_object.do_something(args)
```

is equivalent to

```
some_object.send(:do_something, args) # note the :
```

Symbols are very widely used. That's those things that start with : and they're not immediately obvious (well they weren't to me) but the earlier you get to grips with them the better.

Ruby is big on "duck-typing", following the principal that "if it walks like a duck and quacks like a duck..." that allows informal substitution of objects with a common subset of methods without any explicit inheritance or mixin relationship.

Share
Improve this answer

answered Dec 17, 2008 at 9:31

community wiki Mike Woodhouse

**Follow** 

Thanks. There's one thing I hate about the **send** method: it lets you call private methods even outside the class! Ouch. — MiniQuark Dec 17, 2008 at 9:46

1 @MiniQuark: that's what I love about the send method! – Andrew Grimm Jan 8, 2010 at 2:23



Methods can be redefined and can become a mind-scratcher until you discover the cause. (Admittedly, this error is probably a bit "harder" to detect when a Ruby on Rails

**6** controller's action is re-defined by mistake!)







```
#demo.rb
class Demo

def hello1
   p "Hello from first definition"
end

# ...lots of code here...
# and you forget that you have already defined hello1

def hello1
   p "Hello from second definition"
end

end
Demo.new.hello1
```

#### Run:

```
$ ruby demo.rb
=> "Hello from second definition"
```

### But call it with warnings enabled and you can see the reason:

```
$ ruby -w demo.rb
demo.rb:10: warning: method redefined; discarding old hello1
=> "Hello from second definition"

Share answered Apr 1, 2011 at 2:52 community wiki
Improve this answer

Follow

I'd +100 the use of warnings if I could. — Andrew Grimm Apr 3, 2011 at 23:45
```



If you declare a setter (aka mutator) using <code>attr\_writer</code> or <code>attr\_accessor</code> (or <code>deffoo=</code>), be careful of calling it from inside the class. Since variables are implicitly declared, the interpreter always has to resolve <code>foo = bar</code> as declaring a new variable named foo, rather than calling the method <code>self.foo=(bar)</code>.









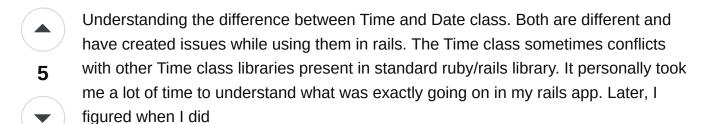
```
end
end
puts Thing.new.foo #=> 2
```

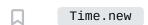
This also applies to Rails ActiveRecord objects, which get accessors defined based on fields in the database. Since they're not even @-style instance variables, the proper way to set those values individually is with self.value = 123 or self['value'] = 123.

Share edited Apr 16, 2011 at 18:21 community wiki

Improve this answer 2 revs
AlexChaffee

Follow





It was referring to some library in a location that I was not even aware of.

Sorry if I am not clear with what I want to say exactly. If others have faced similar problems, please re-explain.

Share answered Dec 17, 2008 at 2:31 community wiki Improve this answer Chirantan

Follow

Follow

One that's caught me out in the past is that the newline character (\n) escape sequence—amongst others—isn't supported by strings within single quotes. The backslash itself gets escaped. You have to use double quotes for the escaping to work as expected.

Share answered Dec 16, 2008 at 20:52 community wiki

Improve this answer

John Topley

1 And that is different from what other language? – Robert Gamble Dec 16, 2008 at 21:17

Java, for one. Single quotes in Java can only be used to enclose a single char, not Strings. – John Topley Dec 16, 2008 at 21:36

This is in keeping with any language that lets you use single quotes for strings, and is why they do. – singpolyma Dec 16, 2008 at 21:42

@John: true, but '\n' in Java will still be the newline character. – user8681 Dec 16, 2008 at 22:06

But in Java single quotes only create values of type char. Not strings. That's the difference.

– jmucchiello Dec 25, 2008 at 13:27



x = (true and false) # x is false



0 and " are true, as you pointed out.



You can have a method and a module/class by the same name (which makes sense, because the method actually gets added to Object and thus has its own namespace).



There is no multiple inheritance, but frequently "mixin modules" are used to add common methods to multiple classes.

Share

edited Dec 16, 2008 at 21:46

community wiki 2 revs, 2 users 93% singpolyma

Improve this answer

Follow

0==true // argh the c compiler in my brain is exploding!! - kenny Dec 20, 2008 at 13:24

0==true gives false in Ruby. That 0 is true makes sense because true is an object in Ruby. In C 0 just happens to have the same representation as false. – Jules Dec 25, 2008 at 13:22

In a condition in Ruby, only false and nil are the false. All others are true values. — rubyprince May 29, 2011 at 6:52



I think it is always good to use .length on things... since size is supported by nearly everything and Ruby has dynamic types you can get really weird results calling .size when you have the wrong type... I would much rather get a NoMethodError: undefined method `length', so I generally never call size on objects in Ruby.



bit me more than once.



Also remember objects have ids, so I try not to use variables call id or object\_id just to avoid confusion. If I need an id on a Users object it is best to call it something like user id.

Just my two cents

Follow



I'm new to ruby, and on my first round I hit an issue regarding changing floats/strings to an integer. I started with the floats and coded everything as **f.to\_int**. But when I continued on and used the same method for strings I was thrown a curve when it came to run the program.



Aparently a string doesn't have a **to\_int** method, but floats and ints do.





Arbitrary parenthesis threw me at first too. I saw some code with and some without. It took me awhile to realize that either styles are accepted.

Share
Improve this answer

answered Dec 17, 2008 at 10:17

community wiki monkut

Follow



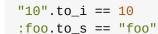
Related to monkut's response, Ruby's to\_foo methods hint at how strict a conversion they'll do.





Short ones like to\_i, to\_s tell it to be lazy, and convert them to the target type even if they're not able to be represented accurately in that format. For example:







The longer explicit functions like  $to\_int$ ,  $to\_s$  mean that the object can be natively represented as that type of data. For example, the Rational class represents all

rational numbers, so it can be directly represented as a Fixnum (or Bignum) integer by calling to\_int.

```
Rational(20,4).to_int == 5
```

If you can't call the longer method, it means the object can't be natively represented in that type.

So basically, when converting, if you're lazy with the method names, Ruby will be lazy with the conversion.

Share answered Feb 24, 2009 at 11:54 community wiki

Improve this answer

Follow

1 Is "lazy" the right word here? – Andrew Grimm Sep 22, 2011 at 23:53



From In Ruby why won't foo = true unless defined?(foo) make the assignment?

foo = true unless defined?(foo) #Leaves foo as nil

Because foo is defined as nil when defined?(foo) is called.

Share edited May 23, 2017 at 11:46 community wiki

Improve this answer 2 revs
Andrew Grimm



Iteration over ruby hashes aren't guaranteed to happen in any particular order. (It's not a bug, it's a feature)

1

Hash#sort is useful if you need a particular order.



Related question: Why are Ruby's array of 1000 hashes' key and value pairs always in a particular order?

M

Share edited May 23, 2017 at 12:09 community wiki

Improve this answer

Andrew Grimm

3 revs

Follow

4 this isn't valid as of 1.9: "In Ruby 1.9, however, hash elements are iterated in their insertion order" from the Ruby Programming Language – Özgür Jul 13, 2010 at 21:17



```
1..5.each {|x| puts x}
```

1

doesn't work. You have to put the range into parentheses, like



$$(1..5)$$
 each  $\{|x| \text{ puts } x\}$ 



so it doesn't think you're calling 5.each. I think this is a precedence issue, just like the x = true and false gotcha.

Share

edited Oct 9, 2011 at 22:17

community wiki

Improve this answer

2 revs Andrew Grimm

**Follow** 

I would call it parenthesis instead. Secondly, if any code looks like having a return value/precedence issue, it should be surrounded by parentheses anyway. So, to me, there is nothing special on this "gotcha". You can keep writing every combinational "gotchas", that would be waste of time, though. Frankly mate, even if you had the expected result on this, I would still prefer surrounding with parentheses. – Özgür Aug 2, 2010 at 12:42



This one made me mad once:

0





## Improve this answer

Follow

I believe this would behave exactly the same way in Java, C, and C++. - Larry Aug 26, 2011 at 0:28

That's funny, I didn't even think about it, but if you open up irb and try this, it makes sense: So (1/2) is a Fixnum and (0.5) is a Float. And we know that Fixnim != Float. − DemitryT Jan 18, 2012 at 16:35 ✓

2 @DemitryT I think the simpler reason is that 1/2 evaluates to 0, which does not equal 0.5, regardless of type. However, Rational(1, 2) == 0.5, and 1.0 == 1.

- Max Nanasy Sep 6, 2012 at 2:18 ▶

universal language hiccup here. this is something someone new to ruby AND programming should know. - dtc Mar 31, 2016 at 20:53