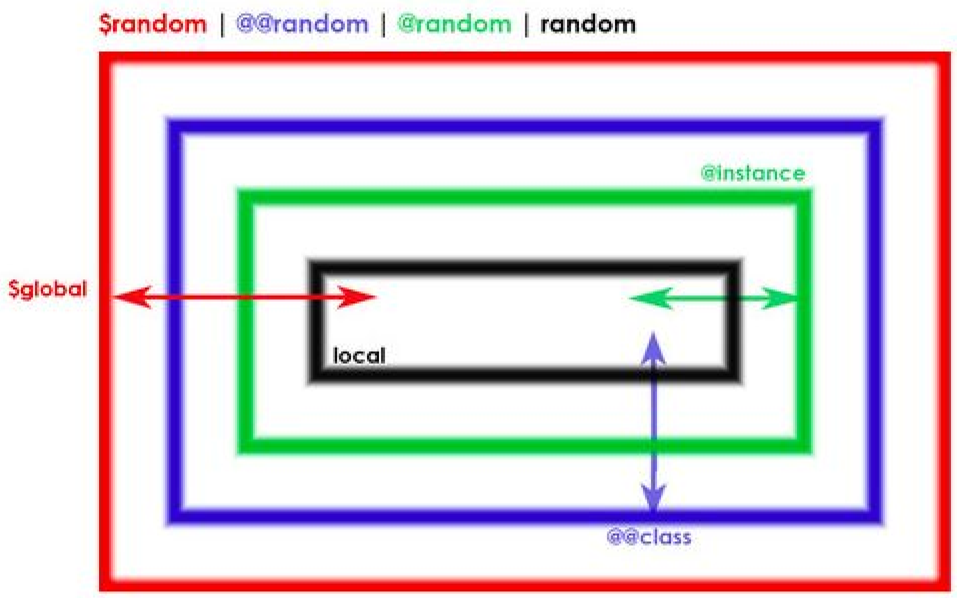
Scoping in Ruby

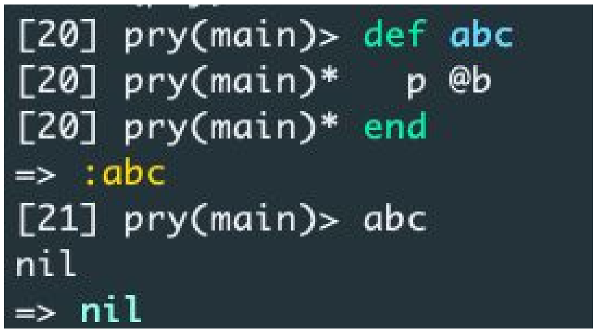
Like every other programming language, Ruby too has variables defined within different scopes.

- **Class variable (@@a_variable):** Available from the class definition and any sub-classes. Not available from anywhere outside.
- **Instance variable (@a_variable):** Available only within a specific object, across all methods in a class instance. Not available directly from class definitions.
- **Global variable ($a_variable):** Available everywhere within your Ruby script.
- **Local variable (a_variable):** It depends on the scope. We work with these the most and most problems lie with their scoping only, because their scope depends on various things.



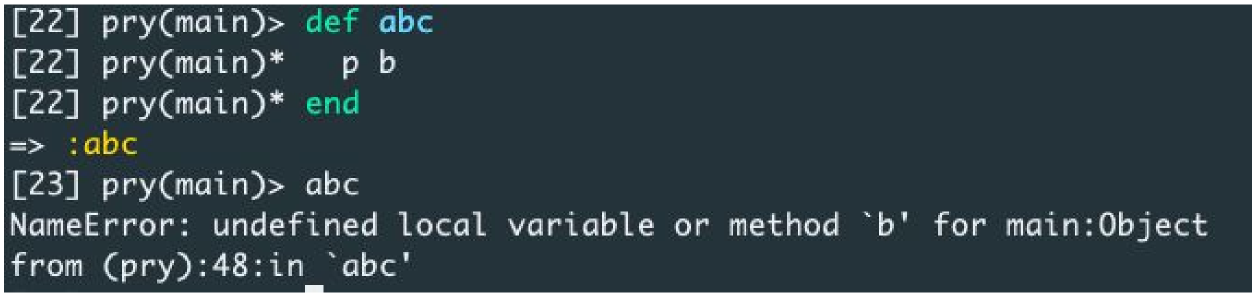
Just the important points:

1. No need to declare **instance variables** in Ruby. Put something like @anything_goes_here in a method definition, and you'll get **nil** as a result.



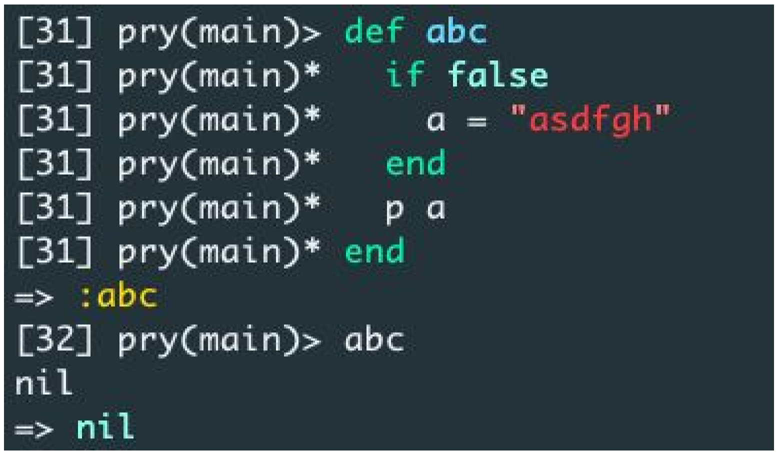
```
[20] pry(main)> def abc
[20] pry(main)*   p @b
[20] pry(main)* end
=> :abc
[21] pry(main)> abc
nil
=> nil
```

2. Same doesn't go for local variable :p

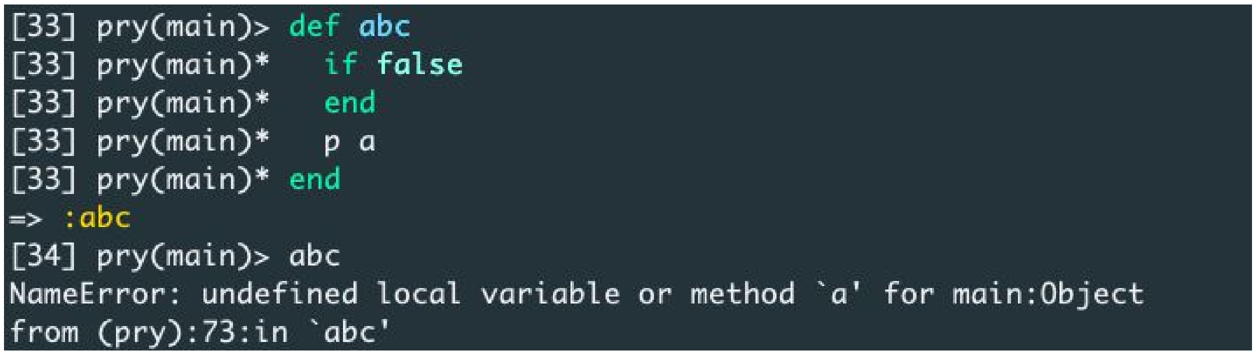


```
[22] pry(main)> def abc
[22] pry(main)*   p b
[22] pry(main)* end
=> :abc
[23] pry(main)> abc
NameError: undefined local variable or method `b' for main:Object
from (pry):48:in `abc'
```

3. The Ruby interpreter will put a local variable in scope whenever it sees it being assigned to something. It doesn't matter if the code is not executed, the moment the interpreter sees an assignment in a local variable, it puts it in scope.

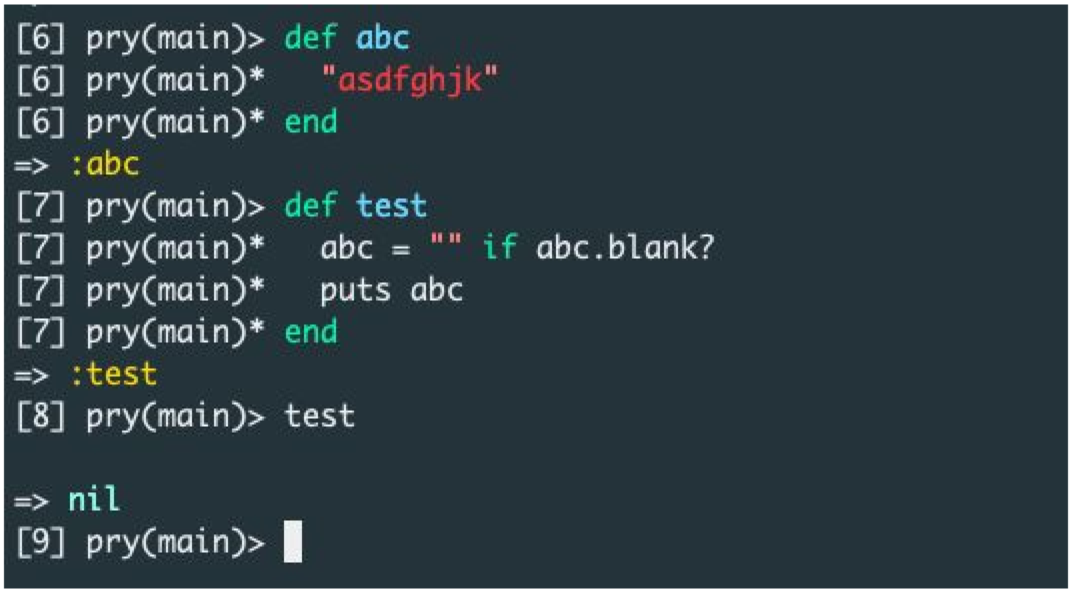


```
[31] pry(main)> def abc
[31] pry(main)*   if false
[31] pry(main)*     a = "asdfgh"
[31] pry(main)*   end
[31] pry(main)*   p a
[31] pry(main)* end
=> :abc
[32] pry(main)> abc
nil
=> nil
```



```
[33] pry(main)> def abc
[33] pry(main)*   if false
[33] pry(main)*   end
[33] pry(main)*   p a
[33] pry(main)* end
=> :abc
[34] pry(main)> abc
NameError: undefined local variable or method `a' for main:Object
from (pry):73:in `abc'
```

4. If a local variable is initialized and a method call with the same name in the same scope, the local variable will “shadow” the method and take precedence.



```
[6] pry(main)> def abc
[6] pry(main)*   "asdfghjk"
[6] pry(main)* end
=> :abc
[7] pry(main)> def test
[7] pry(main)*   abc = "" if abc.blank?
[7] pry(main)*   puts abc
[7] pry(main)* end
=> :test
[8] pry(main)> test

=> nil
[9] pry(main)> 
```

Here abc is being treated as a variable inside the method’s local scope (not assigned any value hence nil). And when we access abc outside the method’s local scope, it will be treated as a method (which was defined earlier).

What if we needed to access the method abc inside the test method’s scope? Is the method abc gone?

Safe way to access the method abc is to put parentheses around the method name.

Alternatively we can also use **send** to treat the method like a method and not a variable.

How do we know if scope has changed?

Scope gates. Whenever we:

1. Define a class (with class SomeClass)

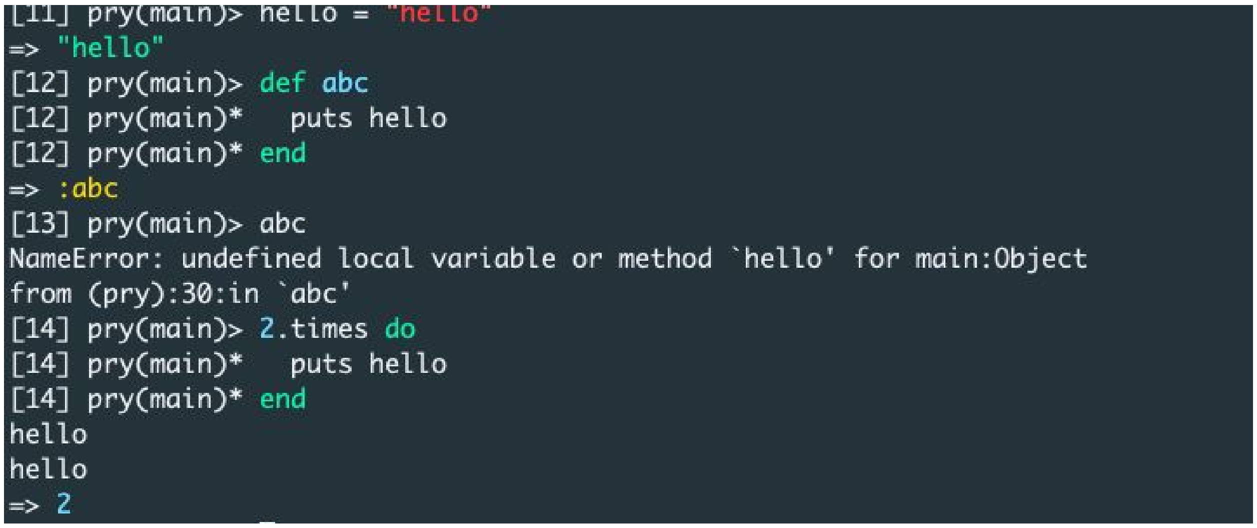
2. Define a module (with module SomeModule)

3. Define a method (with def some_method)

We enter a new scope. Every method/module/class definition is known as a scope gate, because a new scope is created. The old scope is no longer available, and all variables available in it are replaced with the new ones.

Are blocks scope gates too?

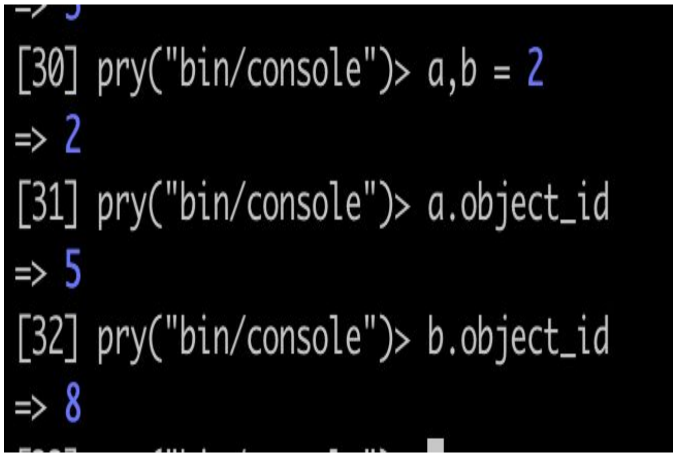
No.



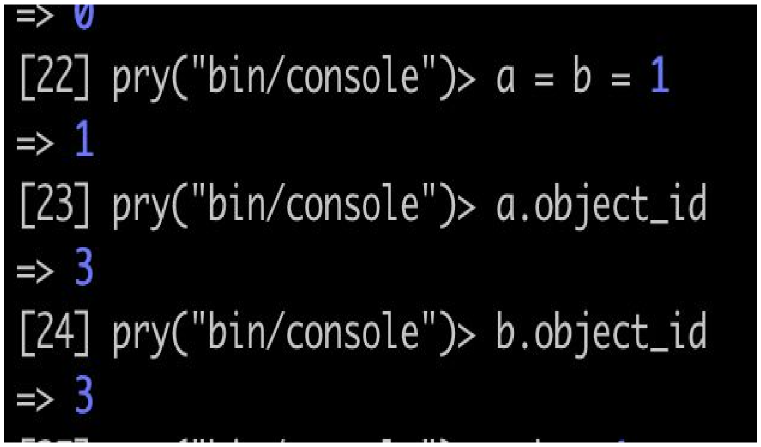
```
[11] pry(main)> hello = "hello"
=> "hello"
[12] pry(main)> def abc
[12] pry(main)*   puts hello
[12] pry(main)* end
=> :abc
[13] pry(main)> abc
NameError: undefined local variable or method `hello' for main:Object
from (pry):30:in `abc'
[14] pry(main)> 2.times do
[14] pry(main)*   puts hello
[14] pry(main)* end
hello
hello
=> 2
```

Here, hello was undefined in the scope of the method abc but it was defined in block scope.

Referencing an object

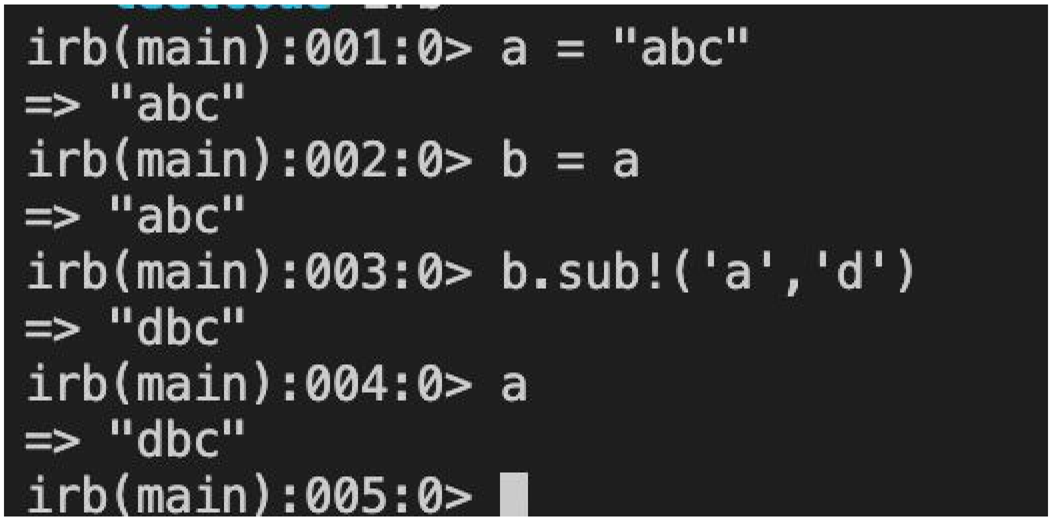


```
[30] pry("bin/console")> a,b = 2
=> 2
[31] pry("bin/console")> a.object_id
=> 5
[32] pry("bin/console")> b.object_id
=> 8
```



```
=> 0
[22] pry("bin/console")> a = b = 1
=> 1
[23] pry("bin/console")> a.object_id
=> 3
[24] pry("bin/console")> b.object_id
=> 3
```

Namespace collision



```
irb(main):001:0> a = "abc"
=> "abc"
irb(main):002:0> b = a
=> "abc"
irb(main):003:0> b.sub!('a','d')
=> "dbc"
irb(main):004:0> a
=> "dbc"
irb(main):005:0>
```