Variable Scope and Blocks

What is a local variable?

There are many types of variables and all come with their own scoping rules. A variable is a name Ruby associates with a memory location. A local variable has the most narrow scope of all variables. A local variable is initialized and assigned with a lowercase name (or underscore) and and `=` sign:

```
local_variable = "I am available in the local variable scope"
```

What is a block?

A block is a method invocation followed by a `do...end` or `{}` denoting a block. Without the method invocation, such as after `for` or `while`, the `do...end` or `{}` does not create a block and therefore has no scoping rules.

An example of a `do...end` block:

```
loop do  # method invocation followed by do...end creates a block
  a = "hi"
  puts "world"
  break
end
```

An example of a block denoted by `{}`:

```ruby
[1, 2, 4].map { |x| puts x + 1 }  # method invocation followed by curly braces denotes a block
```

What is Local Variable Scope?

Local variable scope is where a variable is available for use within a program. A variables scope is determined by what type of variable it is and where it was intialized. Ruby has several types of variable scope, we will focus only on local variable scope.

Scoping Rules for Blocks

Outer scope variables can be accessed by the inner scope

```ruby
outer_scope = "I can be accessed by the block"

loop do
  outer_scope = "I am available here"
  puts outer_scope
  break
end

puts outer_scope ==> "I am available here"
```

In the code above, because our local variable `outer_scope` was initialized and assigned in the main scope, it is available for use throughout our block.

Variables created in the inner scope cannot be accessed in the outer scope

```
loop do
  inner_scope = "I am not available in the outer scope"
  puts inner_scope
  break
end

puts inner_scope ==> # undefined local variable
```

In the code above, due to our local variable `inner_scope` being intialized within the block, it is unavailable outside the block. We would need to intialize and assign this in the main scope to make it available for use in the outer scope.

Blocks that share a common parent scope do not conflict

```
outer_scope = "I can be accessed by both blocks"

loop do
  outer_scope = "I am available here"
  puts outer_scope
  break
end

loop do
  outer_scope = "I am also available here"
  puts outer_scope
  break
end

puts outer_scope ==> "I am also available here"
```

In the code above, because we initialized our local variable in the main scope, it is available for both blocks. Had we failed to initialize in the main scope, it would only be available for the block it was intialized in:

```ruby
loop do
  inner_scope = "I am available here"
  puts inner_scope
  break
end

loop do
  puts inner_scope # undefined local variable
  break
end

puts inner_scope ==> # undefined local variable
```

In the code above, because we have not initialized our local variable in the main scope, it is not available for both blocks. It also causes our local variable `inner_scope` to be unavailable in the outer scope.

The rules of inner scope variables and outer scoped variables applies to nested blocks

```ruby
first_level = 1            # Main Level Scope

loop do                    # First Level Block/Scope
  second_level = 2

  loop do                  # Second Level Block/Scope
    third_level = 3
    puts first_level      ==> # 1
    puts second_level     ==> # 2
    puts third_level      ==> # 3
    break
  end
                     # Move back into our First Level Scope
  puts first_level        ==> # 1
  puts second_level       ==> # 2
```

```
16    puts third_level        ==> # undefined local variable
17    break
18  end
19                        # Move back into our Main Level Scope
20  puts first_level          ==> # 1
21  puts second_level         ==> # undefined local variable
22  puts third_level          ==> # undefined local variable
```

In the code above, you can visualize where the code is available in the nested blocks. In order to make our `second_level` and `third_level` variables available for use througout our program, we would need to initialize at the main scope level.

Variable shadowing prevents access to the outer scope local variable

Variable shadowing is when the block parameter name is the same name as our outer local variable, the inner scope local variable will use the block parameter and we will be blocked from using the outer scope local variable. This also keeps us from making any changes to the outer scope local variable.

```
1  n = 2
2
3  n.times do |n|
4    n = 6
5  end
6
7  puts n
```

In the code above, on line 1, a local variable `n` is initialized and assigned to the integer object `2`.

On line 3, the `times` method is called on the local variable `n`, which references the integer object `2`, and a block is passed as an argument. The block takes one block parameter, also named `n`.

Note that from inside the block, the local variable `n` refrences the block variable. Due to variable shadowing of 2 variables with the same name, `n` , but with different scopes, the outer variable `n` is shadowed from inside the block and cannot be accessed. From inside the block, the local variable `n` is reassigned to the value `6` as `times` iterates through the block 2 times. This action, again, does not have any effect on the outer scope `n`.

Blocks Within Method Definitions

The rules for local variable scoping applies even when a block is located within a method definition.

```
1  def print_scope
2    outer_scope = "I can be accessed by the block"
3
4    loop do
5      inner_scope = "I am not available in the outer scope"
6      puts inner_scope
7    break
8    end
9
10   puts outer_scope  # available
11   puts inner_scope  # not available
12 end
13
14 print_scope ==> # `print_scope': undefined local variable or
   method `inner_scope' for main:Object (NameError)
```

Again, in order to access our local variable `inner_scope` in the outer scope, it would need to be initialized and assigned on line 3. Did you find this tricky?

Summary

Understanding local variable scope and its nuances is important to developers because it can be a source of confusion and unexpected errors. The concepts above can seem complex at first but with a little practice will soon become second nature. Local variable scope is just one rung on the complex ladder of learning to become a software developer. Good Luck!

Variable Scope and Methods

What is a local variable?

There are many types of variables and all come with their own scoping rules. A variable is a name Ruby associates with a memory location. A local variable has the most narrow scope of all variables. A local variable is initialized and assigned with a lowercase name (or underscore) and a `=` sign:

```
1 local_variable = "I am available in the local variable scope"
```

What is a method?

In Ruby, a method is used to combine repeatable code into a single place. A method is defined using the `def` keyword, a chosen method name, the body of the method and the keyword `end`. A custom method can include a parameter. A parameter is how we pass in arguments to our method. As you will soon find out, without parameters, we can't pass in data to our method.

Once we have defined our method, we utilize our method through the use of a method invocation. This is where we call the method and return the action performed in the body of the method. We use our method invocation to pass in arguments to our method. Arguments are simply the values we pass into our method.

```
def custom_method(parameter)
  # what you want the method to do goes here
end

custom_method(argument) # method invocation with argument
```

What is Local Variable Scope?

Local variable scope is where a variable is available for use within a program. A variables scope is determined by what type of variable it is and where it was initialized. Ruby has several types of variable scope, we will focus only on local variable scope for now.

Scoping rules for Methods

A method can't access variables outside its own scope

A methods scope is self contained, it cannot access variables initialized outside the method.

```
outer_scope = "I am not available to the method"

def custom_method
  puts outer_scope
end

custom_method ==> # undefined local variable
```

In the above code, because our local variable is not passed in through the use of a parameter, we cannot access the data from within our custom method.

The scope of local variables defined within a method is limited to that method.

You can't access variables that are defined within a method from outside that method.

```
1 def custom_method
2   inner_scope = "I am available to the method, but not outside the
  method"
3   puts inner_scope
4 end
5
6 custom_method ==> "I am available to the method"
7
8 puts inner_scope ==> # undefined local variable
```

In the above code, `inner_scope` is available for use within our method, but the self contained scope of the method prevents us from accessing it outside the method definition.

A method definition can only access objects passed in

In order to pass in an outer scope variable, we pass in objects by defining parameter(s).

```
1 def custom_method(parameter)
2   puts parameter
3 end
4
5 outer_scope = "I am available to the method using a parameter"
6 custom_method(outer_scope) ==> "I am available to the method using
  a parameter"
```

In the above code, on line 6 we invoke our custom method `custom_method` and pass in a single argument, `outer_scope`. This binds our argument to the method parameter, `parameter`, and allows us access to the data referenced within the local variable `outer_scope`.

Constant Scope

Constants have what is called lexical scope. They are available throughout all scopes. One caveat however, is constants may not be created within a method definition.

Constants are created by using all capital letters in the variable name and should be used only for data that will never change. Though Ruby will allow you to change the assigned value for a constant (with a warning), this should never be done.

```
CONSTANT = "I have a lexical scope"
```

An example of how a constant is available throughout the program:

```
WEATHER = "Sunny"

def predict_weather
  puts "Today's weather is: #{WEATHER}"
end

predict_weather ==> "Today's weather is: Sunny"
```

Summary

Understanding variable scope and its nuances is important to developers because it can be a source of confusion and unexpected errors. The concepts above can seem complex at first but with a little practice will soon become second nature. Variable scope is just one rung on the complex ladder of learning to become a software developer. Good Luck!