Assignment

In Ruby assignment uses the = (equals sign) character. This example assigns the number five to the local variable v:

```
v = 5
```

Assignment creates a local variable if the variable was not previously referenced.

Local Variable Names

A local variable name must start with a lowercase US-ASCII letter or a character with the eight bit set. Typically local variables are US-ASCII compatible since the keys to type them exist on all keyboards.

(Ruby programs must be written in a US-ASCII-compatible character set. In such character sets if the eight bit is set it indicates an extended character. Ruby allows local variables to contain such characters.)

A local variable name may contain letters, numbers, an _ (underscore or low line) or a character with the eighth bit set.

Local Variable Scope

Once a local variable name has been assigned-to all uses of the name for the rest of the scope are considered local variables.

Here is an example:

```
1.times do
  a = 1
  puts "local variables in the block: #{local_variables.join ", "}"
end

puts "no local variables outside the block" if local_variables.empty?
```

This prints:

```
local variables in the block: a
no local variables outside the block
```

Since the block creates a new scope, any local variables created inside it do not leak to the surrounding scope.

Variables defined in an outer scope appear inner scope:

```ruby
a = 0

1.times do
  puts "local variables: #{local_variables.join ", "}"
end
```

This prints:

```
local variables: a
```

You may isolate variables in a block from the outer scope by listing them following a ; in the block's arguments. See the documentation for block local variables in the calling methods documentation for an example.

See also Kernel#local_variables, but note that a for loop does not create a new scope like a block does.

Local Variables and Methods

In Ruby local variable names and method names are nearly identical. If you have not assigned to one of these ambiguous names ruby will assume you wish to call a method. Once you have assigned to the name ruby will assume you wish to reference a local variable.

The local variable is created when the parser encounters the assignment, not when the assignment occurs:

```ruby
a = 0 if false # does not assign to a

p local_variables # prints [:a]

p a # prints nil
```

The similarity between method and local variable names can lead to confusing code, for example:

```ruby
def big_calculation
  42 # pretend this takes a long time
end

big_calculation = big_calculation()
```

Now any reference to big_calculation is considered a local variable and will be cached. To call the method, use self.big_calculation.

You can force a method call by using empty argument parentheses as shown above or by using an explicit receiver like self.. Using an explicit receiver may raise a NameError if the method's visibility is not public.

Another commonly confusing case is when using a modifier if:

```
p a if a = 0.zero?
```

Rather than printing “true” you receive a NameError, “undefined local variable or method `a’”. Since ruby parses the bare a left of the if first and has not yet seen an assignment to a it assumes you wish to call a method. Ruby then sees the assignment to a and will assume you are referencing a local method.

The confusion comes from the out-of-order execution of the expression. First the local variable is assigned-to then you attempt to call a nonexistent method.

Instance Variables

Instance variables are shared across all methods for the same object.

An instance variable must start with a @ (“at” sign or commercial at). Otherwise instance variable names follow the rules as local variable names. Since the instance variable starts with an @ the second character may be an upper-case letter.

Here is an example of instance variable usage:

```
class C
  def initialize(value)
    @instance_variable = value
  end

  def value
    @instance_variable
  end
end

object1 = C.new "some value"
object2 = C.new "other value"

p object1.value # prints "some value"
p object2.value # prints "other value"
```

An uninitialized instance variable has a value of nil. If you run Ruby with warnings enabled you will get a warning when accessing an uninitialized instance variable.

The value method has access to the value set by the initialize method, but only for the same object.

Class Variables

Class variables are shared between a class, its subclasses and its instances.

A class variable must start with a @@ (two “at” signs). The rest of the name follows the same rules as instance

variables.

Here is an example:

```
class A
  @@class_variable = 0

  def value
    @@class_variable
  end

  def update
    @@class_variable = @@class_variable + 1
  end
end

class B < A
  def update
    @@class_variable = @@class_variable + 2
  end
end

a = A.new
b = B.new

puts "A value: #{a.value}"
puts "B value: #{b.value}"
```

This prints:

```
A value: 0
B value: 0
```

Continuing with the same example, we can update using objects from either class and the value is shared:

```
puts "update A"
a.update

puts "A value: #{a.value}"
puts "B value: #{b.value}"

puts "update B"
b.update

puts "A value: #{a.value}"
puts "B value: #{b.value}"
```

```
puts "update A"
a.update

puts "A value: #{a.value}"
puts "B value: #{b.value}"
```

This prints:

```
update A
A value: 1
B value: 1
update B
A value: 3
B value: 3
update A
A value: 4
B value: 4
```

Accessing an uninitialized class variable will raise a NameError exception.

Note that classes have instance variables because classes are objects, so try not to confuse class and instance variables.

Global Variables

Global variables are accessible everywhere.

Global variables start with a $ (dollar sign). The rest of the name follows the same rules as instance variables.

Here is an example:

```
$global = 0

class C
  puts "in a class: #{$global}"

  def my_method
    puts "in a method: #{$global}"

    $global = $global + 1
    $other_global = 3
  end
end

C.new.my_method

puts "at top-level, $global: #{$global}, $other_global: #{$other_global}"
```

This prints:

```
in a class: 0
in a method: 0
at top-level, $global: 1, $other_global: 3
```

An uninitialized global variable has a value of nil.

Ruby has some special globals that behave differently depending on context such as the regular expression match variables or that have a side-effect when assigned to. See the global variables documentation for details.

Assignment Methods

You can define methods that will behave like assignment, for example:

```
class C
  def value=(value)
    @value = value
  end
end

c = C.new
c.value = 42
```

Using assignment methods allows your programs to look nicer. When assigning to an instance variable most people use Module#attr_accessor:

```
class C
  attr_accessor :value
end
```

When using method assignment you must always have a receiver. If you do not have a receiver Ruby assumes you are assigning to a local variable:

```
class C
  attr_accessor :value

  def my_method
    value = 42

    puts "local_variables: #{local_variables.join ", "}"
    puts "@value: #{@value.inspect}"
  end
```

```
end

C.new.my_method
```

This prints:

```
local_variables: value
@value: nil
```

To use the assignment method you must set the receiver:

```
class C
  attr_accessor :value

  def my_method
    self.value = 42

    puts "local_variables: #{local_variables.join ", "}"
    puts "@value: #{@value.inspect}"
  end
end

C.new.my_method
```

This prints:

```
local_variables:
@value: 42
```

Abbreviated Assignment

You can mix several of the operators and assignment. To add 1 to an object you can write:

```
a = 1

a += 2

p a # prints 3
```

This is equivalent to:

```
a = 1

a = a + 2

p a # prints 3
```

You can use the following operators this way: +, -, *, /, %, **, &, |, ^, <<, >>

There are also ||= and &&=. The former makes an assignment if the value was nil or false while the latter makes an assignment if the value was not nil or false.

Here is an example:

```
a ||= 0
a &&= 1

p a # prints 1
```

Note that these two operators behave more like a || a = 0 than a = a || 0.

Implicit Array Assignment

You can implicitly create an array by listing multiple values when assigning:

```
a = 1, 2, 3

p a # prints [1, 2, 3]
```

This implicitly creates an Array.

You can use * or the “splat” operator or unpack an Array when assigning. This is similar to multiple assignment:

```
a = *[1, 2, 3]

p a # prints [1, 2, 3]
```

You can splat anywhere in the right-hand side of the assignment:

```
a = 1, *[2, 3]

p a # prints [1, 2, 3]
```

Multiple Assignment

You can assign multiple values on the right-hand side to multiple variables:

```
a, b = 1, 2

p a: a, b: b # prints {:a=>1, :b=>2}
```

In the following sections any place "variable" is used an assignment method, instance, class or global will also work:

```
def value=(value)
  p assigned: value
end

self.value, $global = 1, 2 # prints {:assigned=>1}

p $global # prints 2
```

You can use multiple assignment to swap two values in-place:

```
old_value = 1

new_value, old_value = old_value, 2

p new_value: new_value, old_value: old_value
# prints {:new_value=>1, :old_value=>2}
```

If you have more values on the right hand side of the assignment than variables on the left hand side the extra values are ignored:

```
a, b = 1, 2, 3

p a: a, b: b # prints {:a=>1, :b=>2}
```

You can use * to gather extra values on the right-hand side of the assignment.

```
a, *b = 1, 2, 3

p a: a, b: b # prints {:a=>1, :b=>[2, 3]}
```

The * can appear anywhere on the left-hand side:

```
*a, b = 1, 2, 3

p a: a, b: b # prints {:a=>[1, 2], :b=>3}
```

But you may only use one * in an assignment.

Array Decomposition

Like Array decomposition in method arguments you can decompose an Array during assignment using parenthesis:

```
(a, b) = [1, 2]

p a: a, b: b # prints {:a=>1, :b=>2}
```

You can decompose an Array as part of a larger multiple assignment:

```
a, (b, c) = 1, [2, 3]

p a: a, b: b, c: c # prints {:a=>1, :b=>2, :c=>3}
```

Since each decomposition is considered its own multiple assignment you can use * to gather arguments in the decomposition:

```
a, (b, *c), *d = 1, [2, 3, 4], 5, 6

p a: a, b: b, c: c, d: d
# prints {:a=>1, :b=>2, :c=>[3, 4], :d=>[5, 6]}
```