Ruby Variables

We’re going to (finally) talk about Ruby variables today.

Prerequisites

Ruby Programs

You should be able to write multi-line Ruby programs at this point. Variables don't make sense without them. There's no point defining a variable if you don't use it on another line.

What you'll learn

- Ruby variables
- String interpolation
- Variable mutators (! methods)

Ruby “Sentences”

```
irb(main):001:0> 40 * 0.20
=> 8.0
irb(main):002:0> 40 + 8.0
=> 48.0
irb(main):003:0> 48.0 / 3
=> 16.0
```

I’ve been hinting at a basic programming concept for a while. I’ve avoided it until now because I wanted to reserve IRB for Ruby one-liners. Like learning English, I wanted to start with basic phrases.

But now that you know how to write multi-line programs, it’s time to start writing some Ruby sentences and paragraphs.

Let’s say you wanted to figure out the tip on a $40 tab and split it three ways. Using IRB as a calculator, you can solve the problem this way.

Ruby “Sentences”

```
irb(main):001:0> 40 * 0.20
=> 8.0
irb(main):002:0> 40 + 8.0
=> 48.0
irb(main):003:0> 48.0 / 3
=> 16.0
```

This isn’t actually a Ruby program. It’s a collection of Ruby “phrases” that we execute, manually, one line-at-a-time, copying-and-pasting the results from one line into the next. All the items in bold are where we copy-and-paste.

Real programs don’t work like this, by manually executing one-line-at-a-time.

Real programs are processed multiple lines at a time. So instead of copying-and-pasting the results, it would be nice to be able to store our partial result somewhere and use it later.

Ruby Variables

```
tip = 40 * 0.20
total = 40 + tip
my_share = total / 3
puts my_share
```

[demo in ide]

Ruby variables are very straightforward. There's a name on the left, an equals sign (just 1 equals sign), and *any* of the Ruby expressions we've taught you so far on the right.

This is a complete Ruby "sentence".

In this program I'm storing the tip in a variable named "tip". Then I'm adding that tip to my amount to calculate the total. Then I'm dividing the total by 3 to get my share.

Since this is a program and not IRB, I need to output my result. So let's `puts` it for the end user.

The answer should be 16.0.

Who knows why this is a Float value?

Ruby Variables

```
tip = 40 * 0.20
total = 40 + tip
my_share = total / 3
puts my_share
```

This is a program that only calculates my share of a $40 bill with a 20% tip when I'm dining with two other people. That's pretty specific and, therefore, not very useful.

If I want to change this program to work with, say, a $60 tab, I need to replace the 40 in two places. That's annoying. And it's bad software design. I'd like to be able to change my data once and have it updated everywhere it's used.

Variable Naming

```
bill = 40
num_people = 3
tip_percent = 0.20

tip = bill * tip_percent
total = bill + tip
my_share = total / num_people
puts my_share
```

This is a bit longer, but it's a lot easier to read and update. All my numbers are defined in one place (at the top). I just need to update them once and the entire program will work with my new values. Great.

Was this worth it? What about speed? Aren't longer programs slower?

Design is usually about trade-offs. Sometimes you have to give a little speed to gain a little readability and maintainability. Occasionally you can refactor a program to get both increased speed *and* readability. Usually you have to pick one or the other. As beginners, you should *always* pick readability and maintainability.

What's maintainability? It's how easy a program is to maintain. By refactoring this program so that I only have to update the bill in one place to change how the program works, I've made it easier to maintain. I'm less likely to, for example, make a mistake and only update one 40 and not the other (see the code in the previous slide).

Also note that I've picked **descriptive variable names**. I could have named the bill variable `b`, the number of people variable `p`, and the tip variable `t`. But then, when I write the rest of the program, I'll have to remind myself, for example, was `t` the tip percentage or the total tip amount? Don't make me think too hard.

The more explicit you can be with your variable names, the better *designed* your program will be. It's the difference between code that works and code that works *well*.

String Interpolation

```
age = 12

puts "I'm " + age.to_s + " years
old"
```

Ruby variables are commonly used within Strings. You can include a variable inside a String using the “+” concatenation method.

Strings can only be concatenated with other Strings. So you have to convert the Fixnum 12 into a String. Otherwise you’ll get an error.

This second line of code is so common that Ruby provides a shortcut (you’re going to hear me say that a lot - that’s part of the developer happiness thing).

String Interpolation

```
age = 12
puts "I'm #{age} years old"


puts 'I\'m #{age} years old'
```

This is String interpolation. I told you Strings would get more complicated.

String interpolation makes our code easier to read. Using the hash symbol (the octothorpe) and curly brackets, you can drop a variable into a String. This code will even convert the variable into a String (call `to_s`) for you. Nice!

String interpolation *requires* the use of the double-quoted string. It doesn't work with the single-quoted string. Single-quoted strings are too "light" and don't support this feature.

Mutators

```
arr = [1,2,3]
p arr.delete(3)
p arr
```

Now that we have variables, we can start doing things with data types that we haven't been able to do before. I can introduce some methods that I had to skip in an earlier lecture.

For example, we can delete items from Arrays now. In a single IRB line, delete doesn' t make much sense. It returns the item deleted, not the array that's left over. If we want to see the Array left over after the delete, we need to store it away somewhere first. Then, after the delete, we can output the Array.

[If you did your homework, you should know why I'm using `p` instead of `puts` here.]

Methods that change the value in a variable are called *mutators*. They *mutate* the data stored inside the variable.

Mutators

```
str = "LAB Miami"
puts str.downcase
puts str
str.downcase!
puts str
```

By convention, if there are two methods that do the same thing, the one with an exclamation point is a mutator and the other is not. So, for example, String's downcase method.

Without the exclamation point, `downcase` returns a downcased String, but it doesn't change the variable. So when I `print` the `str` variable, it hasn't changed.

With the exclamation point, `downcase!` mutates the variable. So when I output it, it has changed.

You may have noticed lots of methods with exclamation points in them when we checked the list of methods associated with each data type. Now you know why.

Mutating

```
a = 1
a.next
puts a
```

When you want to mutate a variable, but you can't find the mutator you want, you can roll your own.

Fixnum's `next` method, for example, gives you the next number. But it's not a mutator. In this example, the variable `a` is unchanged.

Mutating

```
a = 1
a = a.next
puts a
```

If you'd like like to update the `a` variable, you can set it equal to the value of the call to `a.next`. Now `a` is 2.

Mutating

```
a = 1
a = a + 1
puts a

a = a + 2
a = a - 1
```

All `next` does is add 1, so instead of using `next`, you can also just add 1 yourself.

Or you can add 2. Or subtract 1. Just reset the variable `a` to whatever you'd like.

It turns out that this pattern of using addition or subtraction to modify a variable is so common that Ruby has a shortcut. Time to refactor!

Mutator Shortcut

```
a = 1
a += 1 # same as a = a + 1
puts a

a -= 2 # same as a = a - 2
puts a
```

The `+=` (plus-equals) and `-=` (minus equals) methods are shortcuts for mutating a variable with addition and subtraction.

Mutator Shortcut

```
a = [1,2,3]
a += [4,5,6]
p a
```

Recall that the + and - methods sometimes work on things that aren't numbers. So, for example, you can implement yet another way (our third) to push data onto the end of an Array.

Mutator Hacks

```
b = nil
b ||= 1 # same as "b = b || 1"
puts b

c ||= 2
puts c
```

The last trick I want to show you with variables is another boolean hacks. I talked a little about boolean hacks in a previous lecture.

This version of the hack takes advantage of the fact that some values, like nil, are falsey. So if b = nil, it evaluates to false in an "OR" comparison. So these two lines at the top say "if b is nil, then set it to 1".

You don't even need to have previously defined the variable you're checking. In the last two lines of code, I never even told Ruby what the variable `c` is. Using the `||=` (or-equals) shortcut, Ruby will assume that a variable it's never even seen before is falsey. In this case, it assumes the `c`-variable, which it has never seen before, is falsey, and therefore assigns it to 2.

This is a surprising result. In a boolean context, you can use a variable you've never used before. In any other context you'd get an error.

Mutating

```
c = 1
c ||= 2 # default c to 2
puts c

c = false
c ||= 2 # default c to 2
puts c
```

or-equals is a good way of setting “default values” for variables. A default value is a value to use in case some variable hasn’t been defined yet.

So, in these first lines, since variable `c` has already been defined, it’s not set to 2. If `c` had not been nil, it would have been set to 2.

But, like I warned with boolean hacks, there are cases that will trip you up. So, for example, if you set the variable `c` to false, or-equals will set it to 2, even though it has been defined.

Always be careful with boolean hacks. I would recommend not using them at all, but, unfortunately, they end up being very common in Ruby code.

Constants

```
CONST = 1
CONST = 2
```

```
wyncode.rb:2: warning: already initialized constant C
wyncode.rb:1: warning: previous definition of C was here
2
```

When you define a variable in ALL-CAPS, Ruby considers it a constant. A constant is a value that shouldn't change. Ruby doesn't prevent you from changing it, but it will show a warning if you try to reset it to something else.

Math Constants

```
Math::PI
Math::E
```

The Ruby standard library comes with two common mathematical constants, PI and E. They're located in the Math module, which is why you see the "colon-colon" syntax here. We'll talk about modules later.

Resetting a constant is evil. We'll talk about evilness later as well.