Lesson 3: Conditionals and Loops

[Overview](#h.d1trk3d494qr)

[Branching and Conditionals](#h.lgk7vwdr8poe)

[If](#h.ko0imjwmcalg)

[Boolean Data Type](#h.k2u4ltqm2bhj)

[More Conditionals](#h.7a3kk57xcwsv)

[Else and Elsif](#h.qm7t0ehxif9c)

[Case Statements](#h.yu7c5kcyl5oz)

[Loops](#h.vm92yxlfm759)

[Homework](#h.7ykstxc7nxt5)

# Notes / practice snippets

* [conditionals](https://gist.github.com/coryb710/2d6003e6df63524ed7e8)
* [loops](https://gist.github.com/coryb710/d8bec294c96b05b11616)

# Overview

1. Branching and Conditional Statements: programming Ruby to respond appropriately to multiple possible outcomes.
2. New Data Type: Boolean
3. Loops: While, Until, For, Inline

# Branching and Conditionals

Sometimes we need to respond to a user’s input and do a different set of activities based on that input. This is called **branching**. Like tree branches, we can branch our code to do different things based on different conditions.

## If

One of the most basic conditionals is the **if** statement. For example, if the temperature is 70 degrees today, I will go hiking. Let’s write this in code. Create a file called activity\_today.rb.

Who wants to walk me through what that code would look like?

todays\_temperature = 80

if todays\_temperature > 50

puts "I’m going hiking!"

end

Elements of the if statement:

* Similar to methods, an if statement must always close with an “end” statement.
* The logic is that if the statement after “if” is true, then the code in that section will run. If it is false, it will not.

# Boolean Data Type

This brings us to a new data type - booleans. In Ruby, the boolean data type has two possible values - true and false. When you type either true or false in Sublime, you can see that they change color similar to an integer or float.

What happens if we try to test whether today’s temperature is equal to 80? We have to use a different operator that simply “=”.

When we use “=”, we are setting something equal to something else. To test for “trueness”, we have to use “==”, which means “is equal to”. For example, x = 2 means “set x equal to 2”, and x==2 means “check if x is equal to 2”. You will probably forget that second equal lots of time, so get used to reading errors and checking for that missing sign.

Booleans also work with strings. We can check if is\_hungry == “yes” and then if so, go grab a snack!

We have a few more operations that we can use on booleans in Ruby. Let’s jump into IRB. Let’s go around the room and try to guess the most common ones. Give me an operator and an example using that operator: (>, <, >=, <=, ==, !=, &&, ||, and make sure to show some string examples)

Classroom challenge: Add at least 2 more conditionals to our temperature program using 2 different boolean operators.

# More Conditionals

Let’s make our program more interesting. Let’s create a method called going\_hiking that checks the temperature. First, though, let’s ask the user what the temperature is.

puts "What is today's temperature?"

todays\_temperature = gets.chomp.to\_i

def going\_hiking(temp)

if temp >= 50

puts "#{temp} degrees is perfect for hiking!"

end

if temp < 50

puts "#{temp} degrees is WAY too cold for hiking!"

end

end

going\_hiking(todays\_temperature)

What’s that funny #{} thingy? This is a cool feature of Ruby called **interpolation**, which will print the value of the variable inside the string without us having to use concatenation a bunch.

So what is happening here? Ruby is checking the first if statement and evaluating whether it is true, then it is moving on to the second if statement and evaluating whether that is true. This seems pretty inefficient, right? Well, there is a better way!

## Else and Elsif

If is great for checking one condition, but sometimes we want to check multiple conditions. Else is cool because it will return something else for any case when the if statement is not true.

How would we change our code to use else instead of 2 ifs?

def going\_hiking(temp)

if temp >= 50

puts "#{temp} degrees is perfect for hiking!"

else

puts "#{temp} degrees is WAY too cold for hiking!"

end

end

That is way more efficient (DRY), and helps prevent careless mistakes. Draw a flowchart on the board to help explain the branching aspect.

What happens if we want to have multiple activity options based on the temperature? **Elsif** is a really cool tool that allows us to check multiple conditions before either using an else or ending the if altogether. It is important to note that Ruby checks your elsifs in the order that they are written. Add more to the flowchart.

Let’s go swimming if it’s over 80, hiking if it is between 50 and 80, and reading if it is less than 50. Let’s change the name of the method too, while we are at it.

def pick\_activity(temp)

if temp >= 80

puts "#{temp} degrees is perfect for swimming!"

elsif temp > 50

puts "Hmm, #{temp} degrees sounds excellent for hiking."

else

puts "At #{temp} degrees, it sounds like I should stay inside and read."

end

end

pick\_activity(todays\_temperature)

What are some other ways we could have written this conditional? What is another example of a program you could write with conditionals?

What if someone puts in an unrealistic temperature for New Orleans? Let’s use the or operator to check this condition first:

if temp > 105 || temp < 0

puts "That's not a valid temperature for New Orleans!"

Doesn’t it seem like a lot of work to restart the program when an invalid temp is given? Well, we can help the user out by moving our temperature question inside the method and then re-calling the method inside the method! This is called recursion. It works like this:

def pick\_activity

puts "What is today's temperature?"

temp = gets.chomp.to\_i

if temp > 105 || temp < 0

puts "That's not a valid temperature for New Orleans!"

pick\_activity

elsif temp >= 80

puts "#{temp} degrees is perfect for swimming!"

elsif temp > 50

puts "Hmm, #{temp} degrees sounds excellent for hiking."

else

puts "At #{temp} degrees, it sounds like I should stay inside and read."

end

end

pick\_activity

That might have blown your mind a little bit. If so, don’t worry. For now, just know that it exists and is a cool way to solve some types of problems.

## Case Statements

First, explain ranges using IRB.

Sometimes it is more fun to use case statements. They are very similar to if-elsif statements. For example, we could change our code to use this case statement instead:

case todays\_temperature

when 80..100

puts "Let's go swimming."

when 50...80

puts "Let's go hiking."

when 40...50

puts "Let's stay inside and read."

when 0...40

puts "Let's cozy up by the fire."

else

puts "What planet is that!?"

end

Ranges constructed using .. run from the beginning to the end inclusively. Those created using ... exclude the end value.

(a..b) is like a <= x <= b, whereas (a...b) is like a <= x < b.

What would be some problems with this code? How could we fix them? (temperatures outside of those ranges)

## Inline If

puts “The answer to life!” if todays\_temperature == 42

# Loops

Sometimes we need to do the same things multiple times until a condition is met. For example, I need to keep studying until I understand loops. One way to represent this in code is:

while understands\_loops != “yes”

puts “Study more”

end

Ruby has several different ways of accomplishing loops. Based on what you know now, how would you output this to the screen?

1 bird on a wire, ha ha ha!

2 birds on a wire, ha ha ha!

3 birds on a wire, ha ha ha!

4 birds on a wire, ha ha ha!

There is an easier way! First, let’s show it in a for loop:

puts "1 bird on a wire - ha ha ha!"

for n in 2..100

puts "#{n} birds on a wire - ha ha ha!"

end

That was great. **For** loops aren’t used as often in Ruby but you might see it sometimes if someone came from C. Now let’s try a **while** loop, which also comes from C:

x = 2

while x < 10

puts "#{x} birds on a wire - ha ha ha!"

end

Oh no - what happened? (infinite loop because x never changed). Use ctrl-c to exit the loop.

Let’s increase the value of x right before the end of the loop. We can do this with “x = x + 1”, but there is also a shorter way in Ruby: x += 1. This also works for x = x - 1 (x -= 1).

Now let’s try yet another way with an **until** loop (also “shunned”). The key for all of these is to listen to the English literally to guide what you need in the loop. (walk through step by step).

until x == 10

puts "#{x} birds on a wire - ha ha ha!"

x += 1

end

Note which numbers were included and excluded - keep that in mind every time you create loops with numerical iterators.

Classroom challenge (pair program): Write a loop that prints out the even numbers from 1 to 100.

Now let’s learn another cool looping mechanism - the **times do** loop. This is the most Ruby-ish and generally the preferred method in Ruby.

5.times do |i|

puts "#{i+2} birds on a wire - ha ha ha!"

end

The times do loop starts at 0 so we add 2 to make the output show what we want. It automatically knows to iterate by 1 when we include the block |i|.

We can omit the block if we don’t need to use the number in our code. For example:

3.times do

puts "She loves you, yeah yeah yeah!"

end

That was fun. Now back to our loopy birds. We can shorten our code by using **inline** loops. Any do/end can be replaced with { }.

2.upto(10){ |n| puts "#{n} birds on a wire - ha ha ha!" }

The most common looping method in Ruby is the **each** loop. But first, we need to take a timeout and learn about a new data type - **arrays**! (in the [next lesson](https://docs.google.com/document/d/1QJwpOmjgvzDPyVFu94aqeI1liIDVuoOJCYgwxCJcydE/edit?usp=sharing))

Can do each on a range:

(1..10).each do |popcorn|

puts "#{popcorn} kernel of yummy!"

end

# Homework

Find it [here](https://docs.google.com/document/d/1N2KFpTlfgNO3ktY4-OIp7QixpoKkGw4KvYbeybEKIco/edit?usp=sharing).