Ruby Flow Control

Prerequisites

Ruby Variables

What you'll learn

- if and unless
- else and elsif
- case
- loop
- while and until
- for

Tig Notaro Bug

No Questionnaires For Dead People

spotify:track:5Vljm8mGypc4d7ucAf0eIV

We're going to use Ruby to fix Tig Notaro's bug. We're going to make two lists.

Tig Notaro Bug

```
dead_people = [“Ethel”, “Mortimer”,
“Buford”]

alive_people = [“Kelly”, “Joe”,
“Megan”]
```

Here are our two lists of names. We need to make sure we don’t send questionnaires to the dead people.

First, let’s write some code to figure out whether or not a person is dead. Since it’s a true or false value, I expect to get a boolean.

Tig Notaro Bug

```
dead_people.include? “Mortimer”

dead_people.include? “Kelly”
```

This should give me what I want. When I `puts` these values, I see that Mortimer is dead, but Kelly is not. Great.

If/Then/End

```
name = "Mortimer"

if dead_people.include? name then
  puts "Don't send questionnaire to #
{name}."
end
```

Now I need to write Ruby code that, given a name, will refuse to send a questionnaire to that name if that person is dead. *if* that person is dead. *if* that person is dead. Wouldn't it be nice if Ruby knew what the word "if" meant?

It turns out Ruby does (as does most programming languages). Ruby has a special syntax for these types of *if* questions. It's the word "if", followed by any boolean expression, then the word "then", then a block of code, then the word end. In classic Ruby style, it reads much like an English sentence.

The block of code between the if and the end should be tabbed over once. In most modern IDEs, they'll automatically tab the code over for you, so you can just keep typing. When you type "end", they'll untab for you as well. Try it out yourself by typing this program in SublimeText.

If/End

```
name = "Mortimer"

if dead_people.include? name
  puts "Don't send questionnaire to #{name}."
end
```

Ruby has another shortcut we can take advantage of here. You can leave off the word "then" if you'd like, just like in English. You can say:

"If Mortimer is dead, then don't send him a questionnaire." or
"If Mortimer is dead, don't send him a questionnaire."

English works both ways and so does Ruby.

We're missing something here. We need a way to send questionnaires to living people.

If/End, If/End

```
if dead_people.include? name
  puts "Don't send questionnaire to #
{name}."
end

if alive_people.include? name
  puts "Send questionnaire to #{name}."
end
```

You might be tempted to do this. If we need to do another check to see if someone is alive, we can just create a new if block. This works, but there's a way to refactor this to make it even better.

If/Else/End

```
if dead_people.include? name
  puts "Don't send questionnaire to #{name}."
else
  puts "Send a questionnaire to #{name}."
end
```

Just like in English, Ruby supports the concept of "else", as in "if this, then that, else the other thing".

All together, it's `if`, followed by a boolean expression, then a block of code tabbed-over, then the word `else`, then a block of code tabbed over, then the word `end`. SublimeText will handle all the tabbing back and forth for you as you type.

The first block of code runs if the boolean expression is true. Otherwise the second block of code runs.

This is not only shorter than the last code block, but it's also faster. We only have to check the `dead_people` Array once. We don't even have to look at the `alive_people` Array. If you're not dead, we assume you're alive.

Sick People

```
dead_people = ["Ethel", "Mortimer",
"Buford"]
alive_people = ["Kelly", "Joe",
"Megan"]
sick_people = [“Kelly”, “Joe”]
```

But what if some people are alive, but go home sick. For example, what if they’re recovering from surgery or childbirth and don’t have time to fill out a questionnaire right away. Let’s create an Array of sick people that includes some of the `alive_people`, but none of the `dead_people`.

Actually, reading this code I can see that I’m repeating myself. One of the most common software design principles is Don’t Repeat Yourself, D-R-Y, DRY. I’ve typed the names “Kelly” and “Joe” twice, so I’m repeating myself. My code’s not DRY. It’s wet. Let’s refactor.

Sick People

```
dead_people = ["Ethel", "Mortimer",
"Buford"]
alive_people = ["Kelly", "Joe",
"Megan"]
sick_people = alive_people.slice(0,
2)
```

That's better. Now I've only typed the names once. If for some reason I want to change the names, I only need to change them in one place.

I have typed the variable `alive_people` twice, but that's ok. Ruby will warn me if I misspell either variable. It'll throw an error if I try to use a variable it's never seen before. But it won't warn me if I misspell Kelly's name.

The Array `slice` method slices an Array starting at the first argument (0, because Arrays start a 0!) and continuing until we have the 2nd argument amount of items (or we run out of items). So I'm slicing the `alive_people` Array starting at position 0 and ending when I have 2 items. I'm slicing the first two items out of the alive_people array.

If/Elsif/Else/End

```
if dead_people.include? name
  puts "Don't send questionnaire to #{name}."
elsif sick_people.include? name
  puts "Don't send a questionnaire to #{name} yet."
else
  puts "Send a questionnaire to #{name}."
end
```

Now that I've got my Array of sick people, I need to update my program so that I don't send a questionnaire to dead people or sick people.

Ruby allows me to add more blocks of code to my if/else/end. Each additional if after the first uses elsif, as in the English sentence:

"If that, do this. Else if that other thing, do this other thing. Otherwise do something else."

If/Elsif/Else/End

```
country = "us"
if country == "us"
  puts "Hello"
elsif country == "es"
  puts "Hola"
elsif country == "fr"
  puts "Bonjour"
else
  puts "Alo"
end
```

As you start adding elsif blocks to your code, it can sometimes get unwieldy. You might, for example, see this pattern appear in your code. Every elsif is checking whether or not the country variable equals some value.

If you see something like this, it may be time to consider a refactor.

In fact, most refactoring comes from having enough experience to recognize code patterns. That's how experienced developers write good code. They are very familiar with what ugly code looks like and how to fix it.

Case/When

```
case country
when "us"
  puts "Hello"
when "es"
  puts "Hola"
when "fr"
  puts "Bonjour"
else
  puts "Alo"
end
```

The `case`/`when` keywords allow you to skip having to enter "`country ==`" over and over again. It's more lines of code, but fewer characters, so it can be quicker to type.

I don't have a strong preference for `case`/`when` over `if`/`elsif`. The longer the list of cases, the more likely I am to consider refactoring to use `case`/`when`. You can have your own definition of "long". There's no right answer.

If not

```
if not dead_people.include? name
  puts "Send a questionnaire to #{name}."
end
```

Another pattern you might notice is using the methods "`not`" or exclamation point in your boolean expression. So, reading this code, you might say to yourself "If the name is not in the list of dead people, then send that person a questionnaire."

But that can sound a little awkward, so Ruby allows you to make a small change to improve readability.

Unless

```
unless dead_people.include? name
  puts "Send a questionnaire to #{name}."
end
```

Anywhere you see "`if not`", it can be rewritten as "`unless`". That's more code pattern recognition that comes with experience.

`unless` is a little bit easier to understand. You'd read this as:

"Unless the list of dead people includes name, send a questionnaire."

One warning with `unless`: if you decide to use `unless`, you can't use `elsif` as well. Otherwise things just get too confusing. If you need multiple if-blocks, stick with "`if not`".

Some IDEs will give you a hint when you start writing an if/elsif/else expression incorrectly: the automatic tabbing will stop working or the colorization will be weird. Keep an eye out for that.

If/Else/End

```
if false
  complete nonsense
else
  puts "Hello world!"
end
```

The reason why statements using if/else are called flow control is because it allows you to change the way Ruby flows through a program. Instead of reading every line from top to bottom, Ruby will skip over lines based on a boolean value.

By skip over, I mean Ruby will literally ignore the code inside the block it doesn't read. You can put complete nonsense in there and Ruby won't throw an error because it won't even look at it.

Multi-Line If

```
if alive_people.include? name
  puts “Questionnaire for #{name}”
  puts “Question 1: ...”
end
```

Note that block may contain multiple lines of code.

Nested Ifs

```
if alive_people.include? name
  unless sick_people.include? name
    puts "Questionnaire for #{name}"
    puts "Question 1: ..."
  end
end
```

An if block may also contain another if block.

One-Liner If

```
if alive_people.include? name
  puts "Questionnaire"
end
```

```
puts "Questionnaire" if alive_people.include? name
```

If your if statement has only one block of code (no elsif or else), and that block has only one line of code, then you may want to consider refactoring it using the one-line version of if.

The one-line version puts the if after a single line of code. The line of code is only evaluated if the boolean expression is true. This refactoring is appropriate when the boolean expression and the line of code are really short. The result should be something that reads well in English. For example:

“Do something if this is true.” or “Do something unless this is true.”

Assignment If

```
if alive_people.include? name
  message = "#{name} is alive!"
else
  message = “Sorry for your loss.”
end
puts message
```

Finally, if you have an if statement where every block is assigning a value to a variable, like this.

Assignment If

```
message = if alive_people.include? name
  "#{name} is alive!"
else
  "Sorry for your loss."
end
puts message
```

You can refactor that if statement like this. You don't have to repeat "message =" in every block. You can just assign the variable to the result of the if/else statement.

Assignment If

```
message = is_alive ? "Alive!" : "Sorry!"

puts message
```

There's even a one-liner refactoring of the assignment if. This would be appropriate if the boolean expression and the code blocks are all very, very small.

In this case, the "if" is replaced with a question-mark and the "else" is replaced with a colon. This strange syntax is borrowed from the C-language. Ruby itself is written in C.

Why are there so many ways to refactor if/then statements? You can do everything I've shown you with a simple if/elseif/else, but all these alternatives exist because if-logic is so common in programming. It's everywhere.

Google: Show me a link *if* it matches my search terms.
Twitter: Show me a tweet *if* I'm following the user.
Facebook: Show me an activity feed item *if* I'm friends with the user.

With so many possible applications, Ruby tries to give you as much freedom as it can by allowing tons of alternatives syntaxes.

Loop Do

```
# this code is dangerous!
loop do
  puts "Hello world!"
end
```

If statements allow you to skip entire blocks of code. That's one type of flow control. Another type of flow control is looping over the same block of code over and over again.

The simplest way to do that with Ruby is via the "loop" keyword. That will just repeat the block of code over and over again.

This code is potentially harmful. It enters an infinite loop. It will print "Hello world!" forever until it is force-ably stopped. To stop Ruby in SublimeText, you have to click the menu item Tools -> Cancel Build. If that doesn't work, you have to "force quit" SublimeText by long-clicking on the icon and selecting "force quit".

From the Terminal, you have to press ctrl+c.

Infinitely looping code doesn't throw an error. It just hangs there, silently. If the loop is consuming some kind of resource, like disk space or RAM, then eventually you just run of of disk space (which will hopefully trigger an error that kills the program). If you' re not consuming a limited resource, then the loop is just consuming CPU-time. Either way, it'll slow your computer down. If the loop is nasty enough, it can slow your computer down so much that you can't kill the loop. In that case, you can either wait for the computer to crash on it's own or reboot the computer. That's not fun.

Infinite loops are silent killers.

While Loop

```
i = 0
while i < 11
  puts i
  i += 1
end
```

The while loop provides a way out. The while loop will only run as long as a boolean expression is true. As soon as the expression is false, the loop stops.

This code loops as long as the variable i is less than 11. As soon as i is > or = 11, the loop stops. Since the code block increments the variable by 1, this loop will eventually terminate. But if we were to forget the 4th line of code or write it incorrectly, this would be an infinite loop.

Until Loop

```
i = 0
until i >= 11  # same as "while not i >= 11"
  puts i
  i += 1
end
```

Remember how “if” and “unless” are related? If the boolean expression starts with not, then you can replace “if not” with “unless”.

Similarly, you can replace “while not” with “until”.

So Ruby can express the English sentences:
“While this holds true, do this” and “Until this becomes true, do this”

Loop One-Liners

```
i = 0
puts "#{i+=1}" while i < 10

i = 0
puts "#{i+=1}" until i == 10
```

Also similar to if/else, you can write while and until loops as one-liners. This only works if the code block is one line of code.

Like the refactoring for if/else, this would only be appropriate if the code is short.

For Loop

```
for name in alive_people
  puts "Send questionnaire to #{name}"
end
```

Looping forever is… ok. Looping until you find the exit door is… good. Even better, though, is looping from "here" to "there", and stopping. The for-loop allows you to do that. It starts with the word "for" and ends with "end". After "for" comes a new variable name. That variable takes on each value in the Array, one at a time, until all the items have been used, and sends it into the block.

I'm not going to spend a lot of time on for-loops because Ruby has an awesome refactoring that makes these things obsolete. We'll learn about that when we dive deeper into what code blocks are. But other programming languages, including JavaScript, make heavy use of the for-loop, so it's important to know it.

Block Self-Control

`break`

Break out of a loop.

`next`

Go to the next loop.

`redo`

Redo the current loop.

Code blocks are not slaves to for, loop, and while. They can seize control of their own destinies as well.

The “break” method causes a loop to exit. It tell Ruby to skip the rest of the lines in the current block of code and start over after you see “end”.

The “next” method causes a loop to skip. It tells Ruby to ignore the rest of the current code block and start again. If you’re using a for-loop to cycle over all the items in an Array, the next method will skip the current item and restart the loop with the next one. This is useful if you find something unexpected in the Array.

The “redo” method repeats the current loop. It tells Ruby to ignore the rest of the current code block and start over. If you’re using a for-loop to cycle over all the items in an Array, the “redo” method allows you to repeat the loop with the current item. This is useful if you try to do something with the item, fail, and want to try again.

“break” is the most common of the three. It allows you to bail on the loop if you suspect an infinite loop might be threatening.

Loop Performance

```
for name in alive_people
  alive_people.index name
end
```

$O(n) * O(n) = O(n^2)$

One last thing about loops: once you start writing loops, you enter a world where bad things can happen to your performance.

If we've covered big-O notation:
Infinite loops are obviously bad. Your program never finishes.

for loops that do something to every item in a list, run linearly. As you increase the number of items in the list, the performance decreases at the same rate.

This particular for loop does something slow every iteration through the loop. In this case, as you increase the number of items in the list, the performance decreases exponentially. That's bad.

If we've covered big-O notation:
An infinite loop, for example, runs in O-infinity. It never stops.

A loop like this runs in $O(n^2)$. If `alive_people` has n items, and for each item you run an O(n) method, the end result is O(n-squared).

Overall:

Loops are where program performance gets hit the hardest. If you're looking for a place to make your program faster, focus on making the blocks of code within loops as small and fast as possible.