Truthiness

The ability to express "true" or "false" is an important concept in any programming language. It helps us build conditional logic, and understand the state of an object or expression. Usually, the notion of whether a value is "true" or "false" is captured in a **boolean** data type. A boolean is an object whose only purpose is to convey whether it is "true" or "false".

In Ruby, booleans are represented by the `true` and `false` objects. Like everything else in Ruby, boolean objects also have real classes behind them, and you can call methods on `true` and `false`.

```
true.class          # => TrueClass
true.nil?           # => false
true.to_s           # => "true"
true.methods        # => list of methods you can call on the true object

false.class         # => FalseClass
false.nil?          # => false
false.to_s          # => "false"
false.methods       # => list of methods you can call on the false object
```

You can use the two boolean objects in conditionals.

Example 1 - The below will always output `'hi'`.

```
if true
  puts 'hi'
else
  puts 'goodbye'
end
```

Example 2 - Conversely, the below will always output `'goodbye'`.

```
if false
  puts 'hi'
else
  puts 'goodbye'
end
```

Expressions and Conditionals

In real code, you won't use the `true` or `false` objects directly in a conditional. Instead, you'll likely be evaluating some expression or method call in a conditional. Whatever the expression, it should evaluate to a `true` or `false` object.

Example 3

```
num = 5

if (num < 10)
  puts "small number"
else
  puts "large number"
end
```

The above outputs `"small number"` because the expression `num < 10` evaluates to `true`. We can verify in irb:

```
irb:001> num = 5
=> 5
irb:002> num < 10
=> true
irb:003> (num < 10).class
=> TrueClass
```

You can substitute the code expression with a method call too.

Example 4

```
puts "it's true!" if some_method_call
```

The above will output `"it's true!"` if `some_method_call` returns a truthy value.

When using method calls as a conditional expression in this way, you'll generally want the method to return a boolean rather than relying on the truthiness or falsyness of a non-boolean return value.

Logical Operators

Logical operators will return either a truthy or falsey value when evaluating two expressions.

- **`&&`**: this operator is the "and" operator and, in the following examples, will return `true` only if both expressions being evaluated are true.

  Example 5 - we'll play around with `&&` in irb.

  ```
  irb:001> true && true
  => true
  irb:002> true && false
  => false
  irb:003> num = 5
  => 5
  ```

```
irb:004> num < 10 && num.odd?
=> true
irb:005> num > 10 && num.odd?
=> false
```

Nothing too surprising. But take notice that we didn't have to put `num > 10` in parentheses. This means that Ruby's order of operator precedence considers `>` as higher precedence than `&&`. In other words, Ruby didn't get confused and thought you were trying to do this: `num > (10 && num.odd?)`.

Example 6 - you can chain as many expressions as you'd like with `&&`, and it will be evaluated left to right. If any expression is `false`, the entire `&&` chain will return `false`.

```
irb:001> num = 5
=> 5
irb:002> num < 10 && num.odd? && num > 0
=> true
irb:003> num < 10 && num.odd? && num > 0 && false
=> false
```

- `||`: this operator is the "or" operator and, in the following examples, will return `true` if either one of the evaluated objects is true. It's less strict than the `&&` operator.

  Example 7 - only way to return `false` is if all expressions are `false`.

  ```
  irb:001> true || true
  => true
  irb:002> false || false
  => false
  irb:003> true || false
  => true
  irb:004> false || true
  => true
  ```

- Short Circuiting: the `&&` and `||` operators exhibit a behavior called *short circuiting*, which means it will stop evaluating expressions once it can guarantee the return value.

  Example 8 - the `&&` will short circuit when it encounters the first `false` expression.

  ```
  irb:001> false && 3/0
  => false
  ```

  Notice the above code doesn't generate a `ZeroDivisionError`. This is because the `&&` operator didn't even evaluate the second expression; since the first expression is `false`, it can short circuit and return `false`.

  Also, notice that `false || 3/0` will generate an error.

  ```
  irb:001> false || 3/0
  ZeroDivisionError: divided by 0
  ```

  Example 9 - the `||` will short circuit when it encounters the first `true` expression.

  ```
  irb:001> true || 3/0
  => true
  ```

  The above code doesn't generate a `ZeroDivisionError` because `||` didn't evaluate the second expression; it short circuited after encountering `true`.

  Similar to before, notice that `true && 3/0` will generate an error.

```
irb:001> true && 3/0
ZeroDivisionError: divided by 0
```

Relying on the short circuit behavior can be dangerous, but it's sometimes handy. We'll see some examples of its use in common Ruby code below.

Hopefully most of the above is review. We haven't even talked about the main topic of this assignment - what does "truthiness" mean and how does it impact our code? We'll tackle that next.

Truthiness

After that review of booleans and logical operators, we're finally able to talk about the notion of "truthiness". Truthiness differs from `true` in that Ruby considers more than the `true` object to be "truthy". In fact, Ruby is a very liberal language and *considers everything to be truthy other than `false` and `nil`*.

This means that we can use any expression in a conditional, or with logical operators, and as long as it doesn't evaluate to `false` or `nil`, it is considered true. Note that an expression that Ruby *considers true* is not the same as the `true` object. This is what "truthiness" means. Let's take a look at an example.

```
num = 5

if num
  puts "valid number"
else
  puts "error!"
end
```

If you didn't know Ruby at all, you might guess that the above should either output `"error!"`, or the program should generate an error of some sort. But if you run that code, it will actually output `"valid number"`. The reason is because Ruby considers any integer to be "truthy". It does not, however, mean that the `num` variable from above is equal to `true`:

```
num = 5
num == true        # => false
```

This means that even the integer `0` is considered truthy, which is not the case in some other languages. Rubyists take advantage of truthiness in Ruby to write some interesting code. For example, sometimes you'll see assignment in a conditional or logical operator:

```
if name = find_name
  puts "got a name"
else
  puts "couldn't find it"
end
```

Presumably, the `find_name` method will either return a valid object, or it will return `nil` or `false`. Writing code like that is dangerous and can be easily misunderstood by others as equality comparison, rather than assignment.

More common, you'll see code like this:

```
name = find_name

if name && name.valid?
  puts "great name!"
else
  puts "either couldn't find name or it's invalid"
end
```

The `if` conditional above is checking that `name` is not `nil`, then checking the validity of `name`. It's doing this by relying on the `&&` short circuit behavior to not execute `name.valid?` if `name` is `nil`. Remember that `&&` short circuits if it encounters a `false`, and `nil` is considered "falsy".

It can get confusing when incorporating conditionals, logical operators and truthiness considerations. But with the knowledge of the above, practice slowly taking apart complicated code.

Remember this rule: everything in Ruby is considered "truthy" except for `false` and `nil`.

You marked this topic or exercise as completed.
A Note on Style
Walk-through: Calculator

Formatting Help

Normal Markdown & GFM

Github Flavored Markdown

---

bold

~~strike through~~

`single line of code`

Supports Emoji

Code Blocks

You can write code blocks using three backticks (```) followed by the language you want to show. For instance, if you wanted to write a Ruby code block you would write:

```ruby
class Foo
end
```

```
class Foo
end
```

You can also leave out the language identifier, doing this will make it so your code isn't highlighted at all, and all text will be white with no coloring.

```
class Bar
end
```

```
class Bar
end
```

Highlighting

You may highlight lines within your code blocks. This is a different type of highlighting, and isn't related to how the code within your code block may be colored depending on the language identifier used.
To highlight one or more lines of code, add `# highlight` on a separate line just before the first line you wish to highlight. To end highlighting, add `# endhighlight` on a separate line immediately after the last line you wish to highlight.

````
```ruby
def say_hello_world
# highlight
  puts 'Hello World!!!'
# endhighlight
end
```
````

```ruby
def say_hello_world
  puts 'Hello World!!!'
end
```

×