Open in new tab

Loops

Lesson Duration: 60 minutes

Learning Objectives

- To understand how to use looping constructs to control behaviour
- To know how to use a for loop and a while loop

What Are Loops

Loops in Ruby are used to execute the same ‘bit’/chunk of code a specified number of times. One of the principles of writing good code is Don’t Repeat Yourself, or DRY. Writing loops is one way to help us write more DRY code.

We’re going to look at two types of loops: `while` and `for`.

Let’s create a file:

```
#terminal

touch loops.rb
```

While Loop

A while loop executes code while a certain condition is true. Once the condition is not true the loop ends.

> Show this without the `counter += 1` line first.

```
# loops.rb

counter = 0
my_number = 5

while (counter < my_number)
   p "counter is #{counter}"
   counter += 1
end
```

What happens if the condition is always true?… The code will loop forever, or eventually crash.

So beware of infinite loops. Ruby loops just keep on going until you run out of RAM…

Playing with Loops

Let’s write a program that asks a user to guess the answer to a question, and loops until they get it right:

So let’s create a new file for this:

```
#terminal

touch quiz.rb
```

```
# quiz.rb

my_number = 5
p "What number am I thinking of?"
value = gets.to_i()

while (value != my_number)
  p "nope! try again..."
  value = gets.to_i()
end

p "yes!"
```

> Next bit is totally optional, just some conditionals practice. Maybe give them 5 mins to figure it out?

What would need to change in the above program to give the user information about whether their guess was too high, or too low? Is it much effort to do that? Why don’t we…

```
# quiz.rb

my_number = 5
p "What number am I thinking of?"
value = gets.to_i

while (value != my_number)

  if (value > my_number)
    p "too high"
  else
    p "too low"
  end
  p "try again..."
  value = gets.to_i()
end

p "yes!"
```

Exiting out of Loops

To exit out of loops, and when loops crash, we have some other functionality available to us. The keyword `break` terminates the most internal loop.

So for instance, if we want to loop asking the user for input for ever, *until* they type a particular character ('q'), we could use:

> you might want to create a separate file for this

```
  while (true)
    p "type something:"
    line = gets.chomp()
    break if (line.downcase == 'q')
    p "you typed: #{line}"
  end
```

For Loop

So we've already looked at arrays and collections of things, and seen how we can access individual items in that collection and do stuff with the items we access. But what if we want to do that stuff on all the items in the collection.

So say a farmer had a collection of chickens and she wanted to check them all to see if they'd laid any eggs that day. Each morning she goes round all the nests, making sure that each chicken is in its nest, and checks to see if any eggs have been laid. So she actually does exactly the same thing at each chicken's nest. Now if we were writing code to do this then it would get a bit tedious writing the same bit of code over and over again for each chicken.

A for loop executes code once for each element in expression.

Let's start with a simple example using some numbers:

> Write the middle line as `p num` at first, then add `* 3`

```
#loops.rb

numbers = [1, 2, 3, 4, 5]

for number in numbers
  p number * 3
end
```

```
#loops.rb

total = 0

for number in numbers
  total = total + number
end

p total
```

Say a farmer was doing a roll call of her five chickens:

```
# loops.rb

chickens  = [ "Margaret", "Hetty", "Henrietta", "Audrey", "Mabel" ]

for chicken in chickens
  p chicken
end
```

> Draw on board with loops

- first time chicken = "Margaret"
- second time chicken = "Hetty"
- third time chicken = "Henrietta" and so on…

If we add more chickens we don't have to tell it to loop more times, it will just do it.

Looping Through an Array of Hashes

One thing we might see is an array of hashes. So a collection of information about an item is stored in a hash, and then several of these items are stored in an array. This can be really useful to loop through.

> might be useful to send out the chickens array via slack. Then the students can re-use it in the loops in functions lesson

```
# loops.rb

chickens = [
  { name: "Margaret", age: 2, eggs: 0 },
  { name: "Hetty", age: 1, eggs: 2 },
  { name: "Henrietta", age: 3, eggs: 1 },
  { name: "Audrey", age: 2, eggs: 0 },
  { name: "Mabel", age: 5, eggs: 1 },
]

for chicken in chickens
  p "#{chicken[:name]} is #{chicken[:age]}"
end
```

So we can still do a roll call of the chickens. Note that we could have done this with nested arrays, like `[["Margaret", 2, 0], ["Hetty", 1, 2]]` but it could get very confusing remembering which number was for age and which for eggs.

Now what if we want to see how many eggs there are? Also if we want to remove the eggs once we've checked?

```
# loops.rb

total_eggs = 0

for chicken in chickens
  total_eggs += chicken[:eggs]
  chicken[:eggs] = 0
end

p total_eggs.to_s + " eggs collected"
p chickens
```

We can also do conditional logic inside of a loop. How many times do we think this loop will print out 'wooo eggs'?

```
# loops.rb

for chicken in chickens
  if chicken[:eggs] > 0
    p "wooo eggs!"
  end
end
```

Published with GitHub Pages