codecademy

Looping with Ruby

Ruby Assignment Operators

Assignment operators in Ruby are used to assign or update values to variables. The most common assignment operator is `=` but others also exist, like `+=`, `-=`, `*=` and `/=`.

```
a = 1;
a += 3;
puts a; # Output: 4

b = 4;
b -= 2;
puts b; # Output: 2

num = 12;
num *= 2;
puts num; # Output: 24

num /= 4;
puts num; # Output: 6
```

Ruby each Method

To iterate over an array in Ruby, use the `.each` method. It is preferred over a `for` loop as it is guaranteed to iterate through each element of an array.

```
data = [3, 6, 9, 12]

data.each do |num|
  puts "The number is: #{num}"
end

# Output:
# The number is: 3
# The number is: 6
# The number is: 9
# The number is: 12
```

Ruby “next” Keyword

In Ruby, the `next` keyword is used within a loop to pass over certain elements and skip to the following iteration. It is useful for omitting elements that you do not wish to have iterated. `next` is followed by an `if` statement which defines which elements are to be skipped.

```ruby
for i in 1..10
  next if i % 2 == 0
  puts i
end

# In this example, the next
# keyword along with a shorthand
# if statement is used to skip
# over the even numbers in the sequence.

# Output:
# 1
# 3
# 5
# 7
# 9
```

Ruby while Loop

Putting a block of code in a `while` loop in Ruby will cause the code to repeatedly run the code as long as its condition is `true`.
If the block of code doesn’t have a way for the condition to be changed to `false`, the `while` loop will continue forever and cause an error.

```ruby
i = 1

while i <= 3 do
  puts "Message number #{i}"
  i = i + 1
end

# Output:
# Message number 1
# Message number 2
# Message number 3
```

Ruby times Method

To execute the same block of code a set a number of times in Ruby, use the `times` method.

```
5.times { puts "Codecademy" }

# Output:
# Codecademy
# Codecademy
# Codecademy
# Codecademy
# Codecademy
```

Ruby Range

In ruby, a sequence of integers can be demonstrated by a *range*. The range can be divided into an *inclusive range* where the last integer in the sequence is included and an *exclusive range* where the last integer is excluded.

```
# Inclusive
(3..5).each do |i|
  puts i
end
# Output:
# 3
# 4
# 5

# Exclusive
(3...5).each do |i|
  puts i
end
# Output
# 3
# 4
```

Ruby loop

A `loop` method can be used to run a block of code repeatedly in Ruby. Either use curly braces (`{}`) or the `do` / `end` keyword combination to wrap the block of code that will be looped.

```ruby
num = 1
loop do
  puts "We are in the loop!"
  num += 1
  break if num > 3
end

puts "We have exited the loop!"

# Output
# We are in the loop!
# We are in the loop!
# We are in the loop!
# We have exited the loop!
```

Ruby until Loop

Putting a block of code inside an `until` loop in Ruby will cause the code to run as long as its condition remains `false` . It's only when the condition becomes `true` that the loop stops.
If the block of code doesn't allow for a way for the condition to be changed to `true` then the loop will continue forever and it will cause an error.

```ruby
i = 1

until i == 4 do
  puts "Message number #{i}"
  i = i + 1
end

# Output
# Message number 1
# Message number 2
# Message number 3
```

Ruby for Loop

A block of code can be repeated a set amount of times with the `for` loop in Ruby.

```ruby
for i in 1..3 do
  puts "Message number #{i}"
end

# Output
# Message number 1
# Message number 2
# Message number 3
```