101 Ruby Code Factoids

0) 'methods' method

Since almost everything in Ruby is an Object you can type dot methods on it to see what methods are available.

```
4.methods - Object.methods
# => [:-@, :+, :-, :*, :/, :div, :%, :modulo, :divmod, :fdiv, :
**, :abs, :magnitude, :~, :&, :|, :^, :[], :<<, :>>, :to_f, :si
ze, :bit_length, :zero?, :odd?, :even?, :succ, :integer?, :upto
, :downto, :times, :next, :pred, :chr, :ord, :to_i, :to_int, :f
loor, :ceil, :truncate, :round, :gcd, :lcm, :gcdlcm, :numerator
, :denominator, :to_r, :rationalize, :singleton_method_added, :
coerce, :i, :+@, :remainder, :real?, :nonzero?, :step, :quo, :t
o_c, :real, :imaginary, :imag, :abs2, :arg, :angle, :phase, :re
ctangular, :rect, :polar, :conjugate, :conj, :between?]
```

1) _

In IRB the use of the underscore variable _ will hold the evaluation of the last line of code executed.

When the variable is used in code it's an indicator from the developer that this parameter isn't going to be used. For example "Divided into columns of 4"

```
[0,1,2,3,4,5,6,7,9].group_by.with_index {|_,index| index % 4 }.
values
# => [[0, 4, 9], [1, 5], [2, 6], [3, 7]]
```

In the Minitest framework, starting in version 5.6.0, the use of underscore is an alias to the new spec style testing.

```
spec(4).must_be :even?
_(4).wont_be :odd?
```

2) instance_exec

The `instance_exec` method is available on every Object, and everything in Ruby is an Object, and when you use it you open up the `singleton_class` of the object to work on. Anything you can do in a Class you could do in an `instance_exec`.

```ruby
num = Object.new
num.instance_exec {
  def == other
    other == 3
  end
}
num == 4
# => false
num == 3
# => true
```

You can even use this on a Proc… which could drive people crazy if you do. But just so you know that you can, here it is.

```ruby
prc = proc {|num| num == 4}
prc.instance_exec { def == other; other == 3 end  }

prc.call(4)
# => true
prc.call(3)
# => false

prc == 4
# => false
prc == 3
# => true
```

3) Enumerator::Lazy

An Enumerator::Lazy object will give you back one object at a time from your collection with optional processing on each item.

```ruby
def do_the_lazy(array_input)
```

```ruby
  Enumerator::Lazy.new(array_input) do |yielder, value|
    yielder << value
  end
end

x = do_the_lazy([1,2,3,4,5,6])
# => #<Enumerator::Lazy: [1, 2, 3, 4, 5, 6]:each>

x.next
# => 1
x.next
# => 2
x.next
# => 3
x.force
# => [1, 2, 3, 4, 5, 6]
```

4) Struct has Enumerable as its ancestor

Since Struct has Enumerable as its ancestor you can use any method from it and write some handy methods on Structs themselves.

```ruby
class Pair < Struct.new(:first, :second)
  def same?
    inject(:==)
  end

  def add
    reduce(:+)
  end
end

a = Pair.new(4,4)
# => #<struct Pair first=4, second=4>
a.same?
# => true
a.add
# => 8

b = Pair.new(5,6)
# => #<struct Pair first=5, second=6>
b.same?
# => false
b.add
```

```
# => 11
```

5) $:

The $: variable is the load path for Ruby gems. Also available via $LOAD_PATH. You can add the current directory to your load path with:

```
$: << '.'
```

6) inspect

The inspect method is meant to be a human readable representation for any object. It is the default representation of an Object when you call Class#new and it shows it on the next line.

```
class Human < Struct.new(:name, :age)
  def inspect
    "#{name} is a human of age #{age}"
  end
end

joe = Human.new("Joe", 43)
# => Joe is a human of age 43
```

7) Hash#invert

Reverse your Hash key-value pairs.

```
{a: 1, b: 2}.invert
# => {1=>:a, 2=>:b}
```

8) Method#to_proc

You can convert methods to a proc. *Note they may be exclusively scoped to the same kind of Object they were defined in.*

```ruby
def plus_one(x)
  x + 1
end
proc_increment = method(:plus_one).to_proc
proc_increment.call(4)
# => 5
[1,3,5,9].map(&proc_increment)
# => [2, 4, 6, 10]
```

9) module_function

module_function is to a **Module** what **private** is to a **Class**

10) require_relative

require_relative is a convenient way to load other ruby files relative to your current files location.

11) instance_methods

On any class you can call the **instance_methods** method to find out what the individual instances of the class will have defined on them.

12) Enumerable

Enumerable is a module that is included in basic collection types such as Array, Hash, and Struct. So all of these object types will include the instance methods from Enumerable.

13) defined?

The **defined?** ~~method~~ keyword is handy for checking whether anything is publicly defined module, class, method, etc. On classes on modules you can also call **method_defined?**, **public<u>method</u>defined?**, **private<u>method</u>defined?**, and **protected<u>method</u>defined?**

14) –noprompt

If you execute IRB with the command line flag **–noprompt** you will enter in to an IRB session with none of the extra characters showing up on the left side of the terminal. This is great if you want to experiment with code and then use you mouse to copy & paste from it.

15) string % value(s)

You can insert with type conversion into strings using the percent **%** method.

```
"Number: %f %f" % [1,2]
# => Number: 1.000000 2.000000

"Number: %e" % "6"
# => "Number: 6.000000e+00"
```

16) ternary operator _ ? _ : _

When you use a question mark after something with some space it starts an if else switch. If the value before the question mark is true then return the first item after the question mark. If false then move past the colon mark.

```
true ? 10 : 20
# => 10
false ? 10 : 20
# => 20
false ? 10 : 20 ? 30 : 40
# => 30
false ? 10 : !20 ? 30 : 40
# => 40
false ? 10 : 20 ? 30 ? 50 : 60 : 40
# => 50
false ? 10 : 20 ? !30 ? 50 : 60 : 40
# => 60
false ? 10 : !20 ? 30 ? 50 : 60 : 40
# => 40
```

Is is generally advisable to use just one ternary operator per line. If you wish to use more you should add parenthesis around each inner expression to allow for ease of comprehension.

17) ruby -e "#code"

Run ruby code snippets from the command line

```
home:~$ ruby -e "puts 1 + 1"
2
```

18) %[]

%[] is just like using quotations to form a string. It allows you to use interpolation, double quotes, and single quotes within.

```
%[Hello #{ "World!" } "Quoted" and 'quoted!']
# => "Hello World! \"Quoted\" and 'quoted!'"
```

19) erb

ERB is an included library with Ruby and you can use it to add Ruby code to other documents/strings.

```
require "erb"
ERB.new(%[<html><body>Hello <%= "Wor" + "ld!" %></body></html>]
).result
# => "<html><body>Hello World!</body></html>"
```

20) undefined ~~Class~~ instance variables don't raise errors

When a variable hasn't been defined yet and you use the ~~class~~ instance variable form of an at symbol (@) then it will evaluate as nil rather than raising any “undefined” errors.

```
@a
# => nil
@a.to_i
# => 0

class A
end

A.new.instance_eval {@a}
# => nil

def count_from_one()
  @num = @num.to_i + 1
end
count_from_one
# => 1
count_from_one
# => 2
count_from_one
# => 3
count_from_one
# => 4
```

21) UnboundMethod

You can extract an instance method from a class and use it like a stand alone Proc with **bind**.

```
split = String.instance_method(:split)
# => #<UnboundMethod: String#split>

class String
  undef :split
end

"asdf".split("s")
#NoMethodError: undefined method `split' for "asdf":String

split.bind("asdf").call("s")
# => ["a", "df"]
```

22) ObjectSpace

You can get get a reference to every instance of a specific object with ObjectSpace.

```
class A
end

3.times do
  A.new
end

ObjectSpace.each_object(A).count
# => 3
ObjectSpace.each_object(A).to_a
# => [#<A:0x0000000204cc00>, #<A:0x00000002244800>, #<A:0x00000
002254430>]
```

23) freeze

Once you freeze an object it cannot be modified.

```
module Test
  def self.example
    "Hello World!"
  end
end

Test.freeze
Test.example
# => "Hello World!"

module Test
  def self.asdf
    123
  end
end
#RuntimeError: can't modify frozen Module
```

24) ‘self’ can optionally be replaced by the object name

```
module Apple
  def Apple.chew
    "munch munch"
  end
end

Apple.chew
# => "munch munch"

def Apple.cut
  "chop chop"
end

Apple.cut
# => "chop chop"

class A
  def A.foo
    "bar"
  end
end

A.foo
# => "bar"
```

25) Top level scope objects can be accessed with ::

```
module A
  def self.test
    "FOO"
  end
end

module Thing
  module A
    def self.test
      "BAR"
    end
  end

  def Thing.inner
    A.test
  end

  def Thing.outer
    ::A.test
  end
end

Thing.outer
# => "FOO"
Thing.inner
# => "BAR"
```

26) prepend

prepend adds a module to the most recent chain of class ancestors. Those methods will be called first.

```
module A
  def split
    self.upcase
  end
end

String.prepend A

String.ancestors
# => [A, String, Comparable, Object, Kernel, BasicObject]
"asdf".split
# => "ASDF"
```

27) super

super calls the current methods name up the ancestor chain and continues until it finds the definition.

```
module A
  def split(*_)
    super("a")
  end
end

class B < String
  def split
    super("b")
  end
end

b = B.new("123abc")
# => "123abc"
b.split
# => ["123a", "c"]
B.ancestors
# => [B, String, Comparable, Object, Kernel, BasicObject]

String.prepend A

b.split
# => ["123", "bc"]
B.ancestors
# => [B, A, String, Comparable, Object, Kernel, BasicObject]
```

28) arity

arity lets you know how many parameters a Proc or method will take.

```
->{}.arity
# => 0
->_{}.arity
# => 1
->_,_{}.arity
# => 2
->*_{}.arity
# => -1
"".method(:upcase).arity
# => 0
String.instance_method(:upcase).arity
# => 0
```

29) cloning Arrays

When you use the **Array#clone** method you end up with a different Array with the same exact Objects in them. No additional memory will be used for the internal objects. **Array#dup** will do the same thing.

```
class Thing
end
a = [Thing.new, Thing.new]
# => [#<Thing:0x000000017eba48>, #<Thing:0x000000017eba20>]
b = a.clone
# => [#<Thing:0x000000017eba48>, #<Thing:0x000000017eba20>]
a.object_id
# => 12541180
b.object_id
# => 12522640
a.map(&:object_id)
# => [12541220, 12541200]
b.map(&:object_id)
# => [12541220, 12541200]
```

If you modify the Array itself you don't have to worry about the other Array being effected. But if you change an object in the Array internally that object will be changed in both Arrays.

30) Default value for Hash

Just like above where duplicating an Array points to the same objects, so also we have the same object returned when we set a default for the Hash.

```
h = Hash.new
# => {}
h.default = []
# => []
a = h[:c]
# => []
b = h[:d]
# => []
a[0] = 1
```

```
# => 1
h[:z]
# => [1]
```

To remedy this it’s best to use default_proc to create a new Array each time.

```
h = Hash.new
# => {}
h.default_proc = ->*_{ [] }
# => #<Proc:0x00000001f66598@(irb):8 (lambda)>
a = h[:c]
# => []
b = h[:d]
# => []
a[0] = 1
# => 1
h[:z]
# => []
```

31) class_eval with included

Using **class_eval** when you’re writing methods to be included is the more natural way of updating classes. You’ll write your definitions just as if you’re in the class.

```
class A
end

module Example
  def self.included(base)
    base.class_eval do
      def example
        "instance method"
      end

      def self.example
        "class method"
      end
    end
  end
end
```

```
A.include Example

A.example
"class method"

A.new.example
"instance method"
```

32) inherited

When you’re having one class inherit from another class ~~as a “mixin”~~ you can write an inherited hook.

```
class Foo
  def self.inherited(base)
    base.class_eval do
      def bar
        "Drinking at the bar!"
      end
    end
  end

  def foo
    "bar"
  end
end

class A < Foo
end

A.new.foo
# => "bar"
A.new.bar
# => "Drinking at the bar!"

Foo.new.foo
# => "bar"
Foo.new.bar
#NoMethodError: undefined method `bar' for #<Foo:0x000000019163
78>
```

33) %x

You can run external commands with %x{}. **%x** can have any grouping symbols around its content: eg: ”, “”, {}, [], //, (), ˙ Ruby’s percent methods typically allow all of these and more (most any non-alphanumeric character).

```
puts %x'cowsay "Hello World!"'
# ______________
#< Hello World! >
# --------------
#        \   ^__^
#         \  (oo)\_______
#            (__)\       )\/\
#                ||----w |
#                ||     ||
# => nil
```

34) break :value

You can return a value out of a loop with **break** just like you would with **return**.

```
x = loop do
  break 9
end

x
# => 9
```

35) Lonely Operator &.

As of Ruby 2.3.0 there is a new operator known as the Safe Navigation Operator, or the Lonely Operator. According to Matz it “looks like someone sitting on the floor, looking at the dot.” What this operator allows you to do is continue chaining methods even if one of the items along the way returns nil. It will safely return nil if that is the case.

```
@a&.size
# => nil
```

```
@a = [1,2,3]

@a&.size
# => 3
```

Notice we didn't get a **NoMethodError**. This will save a lot of the uses of the Rails **try** method.

36) Hash#to_proc

Mapping values in Ruby is fairly common. As of Ruby 2.3.0 they've added **Hash#to_proc**.

```
hsh = {a: 1, b: 2, c: 3}

[:a, :b, :c].map(&hsh)
# => [1, 2, 3]
```

Anytime you place an ampersand symbol before an Object as a parameter it calls the **to_proc** method on the Object. You can mimic the behavior above in earlier versions of Ruby by using the **method** method.

```
hsh = {a: 1, b: 2, c: 3}

[:a, :b, :c].map(&hsh.method(:[]))
# => [1, 2, 3]
```

37) retry

In any begin/rescue/end block you can use retry to repeat the code execution within the block when an error is raised.

```
begin
  @x = @x.to_i + 1
  raise "Error" if @x < 5
rescue
```

```
  puts "We're rescuing!"
  retry
end
We're rescuing!
We're rescuing!
We're rescuing!
We're rescuing!
# => nil
```

38) raise

raise can take 3 different parameters. Just a text explanation, an Error class & a text explanation, or an Error class & text explanation & where the error is from.

```
raise "Hello World!"
#RuntimeError: Hello World!

raise StandardError, "Hello World!"
#StandardError: Hello World!

class DigestionError < StandardError
end

raise DigestionError, "stomach hurts", "bad food"
#DigestionError: stomach hurts
#    from bad food
```

Note you'll probably want to use file and line info for the from area.

39) FILE

The current file. In irb this will return "(irb)" .

40) LINE

The current line.

41) Hash.[]

```
Hash[:array, :of, :key, :value, :pairs, "."]
# => {:array=>:of, :key=>:value, :pairs=>"."}
```

42) Global Variables

The dollar sign defines global variables. (**$**) Please don’t use them. It will kludge up your code. Constants can be used as global variables. When possible use constants or objects to contain values you need. **APPLE** is a lot nicer to see than **$apple** . If you need to define **APPLE** globally just use **::APPLE**

```
module Kludge
  def Kludge.ugly
    $marco = :polo
  end
end

$marco
# => nil

Kludge.ugly
$marco
# => :polo

module Nice
  def self.thing
    ::Marco = :polo
  end
end

Marco
#NameError: uninitialized constant Marco

Nice.thing
Marco
# => :polo
```

43) $0

$0 (dollar-zero) is the root file executed in Ruby. It can be used like Python’s **name** == “**main**” to only run code if this file is the main file. Avoid running code when the file is required with this.

```
if __FILE__ == $0
  puts "You ran this #{__FILE__} directly! :-)"
end
```

44) case permits then

```
case 1
when 1
then
  puts "yes"
end
#yes
# => nil
```

45) case doesn’t need a value

```
y = 4

case
  when y == 4 then puts "yes"
end
#yes
# => nil
```

46) case then is optional

```
case 4
when 4
  puts "yes"
end
#yes
# => nil
```

47) case calls the === method

```
module Truth
  def self.===(thing)
    puts "=== has been called!"
    true
  end
end

case :pizza_is_delicious
when Truth
  puts "yummy food for my tummy"
end
#=== has been called!
#yummy food for my tummy
# => nil
```

48) tail conditions

You can put your if statements and rescue statements after code.

```
@x = begin
  5
end if false

@x
# => nil

raise "Error" rescue :all_clear
# => :all_clear

if true
  puts "Hello World!"
end if false
# => nil
```

49) use of return

In Ruby the last thing evaluated is automatically the returned object. The only time you need to use **return** is if you want to exit with a value earlier in the code.

```
x = 3

def a
  return true if x == 3
  false
end

a
# => true

x = 5
a
# => false
```

50) String#chars

The **chars** method will automatically split your string into individual String characters in an Array.

```
"asdf".chars
# => ["a", "s", "d", "f"]
```

51) to_enum, enum_for, each, lazy

On any Array you can call any of **to_enum**, **enum_for**, **each**, or **lazy** methods to return an Enumerator Object that you can iterate over. You have basic methods **:next**, **:peek**, **:feed**, and **:rewind** for each of these Enumerators. But with Lazy you also get a **:force** method which returns the original collection.

```
x = [1,2,3].enum_for
x.next
# => 1
x.next
# => 2

y = [1,2,3].lazy
y.peek
```

```
# => 1
y.force
# => [1,2,3]
```

52) curry

You can create additional Proc objects that set some of the parameters on another.

```
add = lambda {|a,b| a + b }

add.call(1,2)
# => 3

add1 = add.curry[1]

add1.call(4)
# => 5
```

53) mandatory keyword parameters

```
def name(first:, last:)
  puts "Your name is #{first} #{last}."
end

name
#ArgumentError: missing keywords: first, last

name first: "J", last: "Appleseed"
#Your name is J Appleseed.
# => nil
```

54) Range inclusive and exclusive

```
Range.new(1,5,true).to_a
# => [1, 2, 3, 4]
(1...5).to_a
# => [1, 2, 3, 4]
```

```
Range.new(1,5,false).to_a
# => [1, 2, 3, 4, 5]
(1..5).to_a
# => [1, 2, 3, 4, 5]
```

55) String#upto

The **upto** method for String uses the strings ordinal values to build the range.

```
"A".upto("z").to_a
# => ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L
", "M", "N",
# "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", "
[", "\\", "]",
# "^", "_", "`", "a", "b", "c", "d", "e", "f", "g", "h", "i", "
j", "k", "l",
# "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "
y", "z"]

"A".upto("z").to_a.map(&:ord)
# => [65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 7
9, 80, 81,
# 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 9
7, 98, 99,
# 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 1
12, 113,
# 114, 115, 116, 117, 118, 119, 120, 121, 122]
```

56) String#squeeze

The squeeze method on String objects brings gaps down to one in length.

```
"  asdf     fdsa   asdf   ".squeeze
# => " asdf fdsa asdf "

"..asdf.......fdsa....asdf...".squeeze(".")
# => ".asdf.fdsa.asdf."
```

Clarification: This doesn’t just squeeze spaces. When no parameter is given it squeezes all duplicate neighboring characters.

57) String#replace

The **replace** method ~~is unusual as it~~ allows you to rewrite the string inside itself.

```
class String
  def are_you_happy?
    self.replace "I'm happy"
  end
end

x = "waaaa!"
# => "waaaa!"

x.are_you_happy?
# => "I'm happy"

x
# => "I'm happy"
```

58) Infinity

You can count up to infinity… but you should do it lazily.

```
x = (1..Float::INFINITY).lazy

50.times do
  x.next
end
# => 50
x.next
# => 51
```

59) Enumerable#detect

You can use the detect method to return the first item that evaluates as true.

```
[1,5,9,11,13,15,18,21,24,26,28].detect(&:even?)
# => 18

[1,5,9,11,13,15,18,21,24,26,28].detect {|number| number.even? }
# => 18
```

60) Enumerable#grep

You can use the **grep** method to find items in an Array that match your expression.

```
Array.instance_methods.grep /\?/
# => [:include?, :any?, :empty?, :eql?, :frozen?, :all?, :one?,
 :none?,
# :member?, :instance_of?, :instance_variable_defined?, :kind_o
f?, :is_a?,
# :respond_to?, :nil?, :tainted?, :untrusted?, :equal?]

Array.instance_methods.grep /one/
# => [:one?, :none?, :clone]
```

61) Method#owner

You can discover which object defines a method within the objects ancestry.

```
Array.instance_method(:grep).owner
# => Enumerable
[].method(:grep).owner
# => Enumerable
```

62) String#tr

Replaces all characters that match in place.

```
"hello world".tr('el', '*')
# => "h***o wor*d"
```

63) String#tr_s

Replaces all characters that match and squashes groups.

```
"hello world".tr_s('el', '*')
# => "h*o wor*d"
```

64) Array building

Using **Kernel#Array** to safely enforce an Array result along with using a class variable to avoid any errors creates for a clean looking and less redundant Array builder. The **<<** method appends a to the end of the Array.

```
def build(thing)
  @arr = Array(@arr) << thing
end

@arr
# => nil

build :brick1
build :brick2
build :brick3

@arr
# => [:brick1, :brick2, :brick3]
```

A more common form is as follows:

```
(@arr ||= []) << thing
```

But I find this far more cryptic looking and not as newbie friendly. Readability, simplicity, and beauty are all important.

65) spaces

You can put spaces in between method calls and new lines after periods.

```ruby
a = "asdf"

a   .   reverse   .
        split(     "s"     )  .
        join   .     capitalize
# => "Fda"

module A
  module B
    module C
      def self.a
        "a"
      end
    end
  end
end

A::
   B::
      C   .
          a
# => "a"
```

66) function one liners

With a semicolon you can avoid the need for adding extra lines. Best for when you need to have many “short” functions defined.

```ruby
class PairMath < Struct.new(:a,:b)
  def add;      inject(:+) end
  def subtract; inject(:-) end
  def multiply; inject(:*) end
  def divide;   inject(:/) end
end

a = PairMath.new(6,2)
# => #<struct PairMath a=6, b=2>
a.add
# => 8
a.subtract
# => 4
a.multiply
# => 12
```

```
a.divide
# => 3
```

67) Forwardable#def_delegators

You can pass method calls forward (to another object) with the Forwardable standard library module.

```
require 'forwardable'
class Arr
  def initialize(thing = [])
    @thing = thing
  end

  extend Forwardable
  def_delegators :@thing, :join, :<<
end

x = Arr.new([1,2,3])

x.join
# => "123"
x << 4
x.join
# => "1234"
```

68) unless

unless is the same thing as **if not** (as it's understood in English). So whenever you see it – think "if not" and that should help. If you aren't accustomed to this it can be confusing. The use case for **unless** is: you want a positive response for a negative situation.

```
module Tree
  def self.has_no_apples?
    true
  end
end

puts "Munch munch" unless Tree.has_no_apples?
```

```
# => nil
```

69) superclass

You can use **superclass** to access the class inherited from.

```
class A
  def foo
    "bar"
  end
end

class B < A
  def foo
    "mountain"
  end
end

x = B.new
x.foo
# => "mountain"

x.class.superclass
# => A
```

70) binding an UnboundMethod

Continuing from the code above in #69 .

```
x.instance_exec {
  self.
    class.
    superclass.
    instance_method(:foo).
    bind(self).
    call
}
# => "bar"
```

When **instance_method(:foo)** is invoked above it returns an UnboundMethod.

UnboundMethods can only be used in the same kind of Object they were defined in. But first they must be bound. To do this we used **bind(self)** inside an **instance_exec** . Then to call it we run it as we would a Proc object with the **call** method.

71) alternative code continuation with \

If you don’t like entering new lines after a period, you may use a backslash.

```
"asdf"      \
  .reverse   \
  .split("s") \
  .join        \
  .capitalize
# => "Fda"
```

72) HEREDOC

Multiple line strings.

```
def a
  <<ASTRING
    This
    is
    multi-
    line.
ASTRING
end

def b
  <<-BSTRING
    This
    is
    as
    well!
  BSTRING
end

a
# => "    This\n    is \n    multi-\n    line.\n"
b
```

```
# => "    This\n    is\n    as\n    well!\n"
```

The dash (–) is needed if you want to indent the ending of your HEREDOC closer.

As of Ruby 2.3.0 they've added the squiggle (~) option to remove leading white space. *Note: It's called a tilde and not squiggle.*

```
def c
  <<~CSTRING
    Look
    ma!
    No
    leading
    white
    space.
  CSTRING
end

c
# => "Look\nma!\nNo\nleading\nwhite\nspace.\n"
```

You can use any up-cased string for marking the beginning and ending of your HEREDOC as long as they're the same at both ends.

73) Hash#dig

As of Ruby 2.3.0 you now have a dig method great for getting deep into nested hashes.

```
buried_treasure = {dirt: {dirt: {dirt: "gold"}}}

buried_treasure.dig(:dirt, :dirt, :dirt)
# => "gold"
```

74) dynamically naming classes

Ruby is full of ways to dynamically define methods. ~~But for classes there seems to only be the use of eval.~~ One way to define Classes is with eval.

```
def boat_them_all(array_in)
  array_in.each do |noun|
    eval <<-BOATMAKER
      class ::#{noun.capitalize}Boat
        def float?
          true
        end
      end
    BOATMAKER
  end
end

nouns = [:joe, :cow, :car]

boat_them_all(nouns)
JoeBoat.new.float?
# => true
CowBoat.new.float?
# => true
CarBoat.new.float?
# => true
```

I use the double colon here because in most cases you'll be defining classes from within another class or module. And if you want the classes to be available globally you'll need to prepend the double colon ::

Another way to define a class is to use **const_set**. (Credit to 0x0dea.)

```
module M
  const_set 'SomeClass', Class.new {
    # methods here
  }
end
```

75) addition doesn't care

about excess symbol usage. You can be artsy with code this way :-).

```
4 + - + + + - - - - + 6
# => -2
```

```
4 + - + + + - - - - - + 6
# => 10
```

76) ~ tilde calls itself on the following Object

```
class Cow
  def ~
    :moo
  end
end

@cow = Cow.new

~
   @cow
# => :moo
~@cow
# => :moo
```

77) empty parenthesis () is nil

```
x = ()
# => nil
```

78) !!

Truthiness of Object. Think of it as a double negative… it evaluates truth.

```
!!nil
# => false
!!false
# => false
!!Object
# => true
!!Object.new
# => true
```

```
!!4.+(4)
# => true
```

79) Ranges guess types

And it’s not always the right guess.

```
("D9".."F5").to_a
# => ["D9", "E0", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8
", "E9", "F0", "F1", "F2", "F3", "F4", "F5"]

("DD".."FA").to_a
# => ["DD", "DE", "DF", "DG", "DH", "DI", "DJ", "DK", "DL", "DM
", "DN", "DO", "DP", "DQ", "DR", "DS", "DT", "DU", "DV", "DW",
"DX", "DY", "DZ", "EA", "EB", "EC", "ED", "EE", "EF", "EG", "EH
", "EI", "EJ", "EK", "EL", "EM", "EN", "EO", "EP", "EQ", "ER",
"ES", "ET", "EU", "EV", "EW", "EX", "EY", "EZ", "FA"]

("88".."AA").to_a
# => ["88", "89", "90", "91", "92", "93", "94", "95", "96", "97
", "98", "99"]
```

80) Symbols have methods too

```
:asdf.class
# => Symbol
:apple.methods - Object.methods
# => [:id2name, :intern, :to_sym, :to_proc, :succ, :next, :case
cmp, :[], :slice, :length, :size, :empty?, :match, :upcase, :do
wncase, :capitalize, :swapcase, :encoding, :between?]
:apple.capitalize
# => :Apple
```

81) Numbers succ

```
4.succ
# => 5
4.succ.succ.succ
```

```
# => 7
x = 4
x.succ
# => 5
x
# => 4
```

82) %w and %W makes an Array of Strings

Like mentioned above in #33 most non alphanumeric symbols can mark the edges.

```
x = 4

%w^a s d f #{x}^
# => ["a", "s", "d", "f", "\#{x}"]

%W^a s d f #{x}^
# => ["a", "s", "d", "f", "4"]
```

BONUS: **%i** and **%I** for Arrays of Symbols. (credit to tfaaft)

```
%i(foo bar baz)
# => [:foo, :bar, :baz]
s = 'ell'
%I(foo h#{s}o baz)
# => [:foo, :hello, :baz]
```

83) refinements are awesome

```
module NewUpcase
  refine String do
    def upcase
      "moo"
    end
  end
end
```

```ruby
class B
  using NewUpcase

  def thing
    "asdf".upcase
  end
end

B.new.thing
# => "moo"
```

84) Procs keep their original binding

The following code is invalid. The error demonstrates that even though the Proc is called from within an instance of Doctor that the Proc executes and evaluates directly from the instance of Cow.

```ruby
class Cow
  def initialize
    @feeling = "moo"
  end

  def feeling
    proc {send("@greeting") + @feeling}
  end
end

class Doctor
  def initialize
    @greeting = "Hi Doctor. "
  end

  def feeling?(how_are_you)
    how_are_you.call
  end
end

Doctor.new.feeling? Cow.new.feeling
#NoMethodError: undefined method `@greeting' for #<Cow:0x000000
013e4698 @feeling=:moo>
```

To make the above work you'd need to have the new scope values brought into the Proc as a parameter.

```
class Cow
  def initialize
    @feeling = "moo"
  end

  def feeling
    proc {|greet| greet + @feeling}
  end
end

class Doctor
  def initialize
    @greeting = "Hi Doctor. "
  end

  def feeling?(how_are_you)
    how_are_you.call(@greeting)
  end
end

Doctor.new.feeling? Cow.new.feeling
# => "Hi Doctor. moo"
```

85) Regex named matchers

```
/(?<h>.+) (?<w>.+)/.match("Hello World!")["h"]
# => "Hello"
/(?<h>.+) (?<w>.+)/.match("Hello World!")["w"]
# => "World!"
```

86) included_modules

Beyond knowing the ancestry hierarchy with the ancestors method you can use the included_modules method for, as the name says, just seeing modules included.

```
Array.included_modules
# => [Enumerable, Kernel]
```

87) at_exit

You can make code run as Ruby exits after the **exit** command. Example from APIDock

```
def do_at_exit(str1)
  at_exit { print str1 }
end
at_exit { puts "cruel world" }
do_at_exit("goodbye ")
exit
# => goodbye cruel world
```

BONUS: And

```
BEGIN { ... }
```

for running ASAP. (credit to 0x0dea)

88) ensure

ensure lets you make sure certain code gets run from within the block.

```
begin
  @a = 9
  raise "error"
ensure
  @a = 7
end
#RuntimeError: error

@a
# => 7
```

89) alias

You can give a method an additional name.

```
class A
  def foo
    "bar"
  end
end

class B < A
  alias :fib :foo
end

B.new.foo
# => "bar"
B.new.fib
# => "bar"
```

90) ENV

ENV is a variable holding a Hash of your systems environment variables.

91) Marshal

Convert Objects to Strings and back again with Marshal.

```
x = Marshal.dump(Array([1,2,"3"]))
# => "\x04\b[\bi\x06i\aI\"\x063\x06:\x06ET"
Marshal.load(x)
# => [1, 2, "3"]
```

92) sleep

You can make your current thread sleep with calling sleep and providing in seconds how long to wait.

93) TAB

In IRB you can press the TAB key to autofill the rest of a constant or method name. If there are more than one possibility it will list them all.

94) help

If you have your RI documentation installed then you can lookup information on methods in Ruby by typing **help** in IRB.

95) block_given?

block_given? is a method to determine if a block can be evaluated in the current scope. *Formerly **iterator**?*

```
def a(&b)
  block_given?
end

a
# => false
a {}
# => true
```

96) $>

$> is a global variable for STDOUT

```
$> << "hello world!\n"
#hello world!
# => #<IO:<STDOUT>>
```

97) $;

$; is a global variable that may change what strings split on by default.

```
"a s d".split
# => ["a", "s", "d"]
```

```
$; = "."

"a s d".split
# => ["a s d"]
```

It looks as though this feature is here to stay: Ruby Issue #11729

BONUS: And **$,** is the default for **String#join** (credit to 0x0dea)

98) warn

You can use warn to print a warning message to STDERR.

```
warn "Oh no!"
#Oh no!
# => nil
```

99) 1.0/0 is Infinity

```
1/0
#ZeroDivisionError: divided by 0
1.0/0
# => Infinity
```

100) Ruby has been around for 20 years!

```
RUBY_COPYRIGHT
# => "ruby - Copyright (C) 1993-2015 Yukihiro Matsumoto"
```