Ruby Syntax Cheatsheet

(based on Ruby for Rails by David Black)

Compiled by Ashraf @ rubynerds.blogspot.com

The Basics

Arithmetic

```
2 + 3
2 – 3
2 * 3
2 / 3
```

Printing to the screen

```
puts “Hello”
print “Hello”
p “Hello”

x = “Hello”
puts x
print x
p x
```

Getting input from the screen

```
gets
string = gets
```

String to number conversion

```
x = “100”.to_i

string = “100”
x = string.to_i
```

Number to string conversion

```
x = 100.to_s
x = 100
string = x.to_s
```

Comparing two values

```
x == y
```

Commenting

```
#This is a comment!
```

File handling

File writing

```
fh = File.new(“filename.dat”, “w”)
fh.puts x #x is imaginary variable here
fh.close
```

File reading

```
fh = File.read(“filename.dat”)
```

External code inclusion

```
require “filename.rb”
```

Or

```
load “filename.rb”
```

String Interpolation

```
x=1
puts “x is equal to: #{x}”
```

Embedded Ruby

To embed Ruby code in HTML

```
<%#Ruby code in here%>
```

To 'print out' result of execution in HTML

```
<%=#Ruby code in here%>
```

Code Blocks

Any code defined within {} or do end

```
{#code}
```

OR

```
do
    #code here
end
```

Variables and Constants

Constants

Constants start with a capital letter

```
Constant = “Hi!”
```

Constant Resolution in Nested Classes/Modules

```
Class M
    Module N
        Class O
            Class P
               X = 1
            end
        end
    end
end
```

The constant is accessed by

```
puts M::N::O::P::X
```

Value to variable assignment

```
x = 1
string = “Hello!”
```

Global Variables

Defined with the $ sign

```
$gvar = “This is a global variable!”
```

Instance Variables

Refer to Instance Variables in Classes

Methods

Method definition

```
def method1(x)
    value = x + 1
    return value
end
```

Method Arguments

fixed number of arguments

```
def method(a,b,c)
```

variable number of arguments

```
def method(*a)
```

default value for arguments

```
def method(a, b=1, c=2)
```

combination arguments

```
def method(a, b, *c)
```

NOTE: def method(a, *b, c) is not allowed!

Boolean Methods

```
def ticket.available?
    #boolean evaluation here
end
```

Setter Methods

Refer to Setter Methods in Classes

Method Access Rules

Access Rule	Who can access
Public	**Other objects can access**
Private	**Only instances of object can access mthd on itself (self only)**
Protected	**Only instances of object can access mthd on each other**

Here's 2 ways to define private/protected/#public methods (private example only)

method 1:

```
Class Bake
    def bake_cake
        add_egg
        stir_mix
    end

    def add_egg
    end

    def stir_mix
    end

    #private definition
    private :add_egg, stir_mix
end
```

method 2

```
Class Bake
    def bake_cake
        add_egg
        stir_mix
    end

    private
    def add_egg
    end

    def stir_mix
    end
end
```

Objects

Generic Object

```
obj = Object.new
```

Object 'singleton' method definition

```
def obj.method
    puts “instance method definition”
end

#method call
obj.method
```

Default Object Methods

respond_to? - Checks if methods by the name in argument are defined for the object

```
obj.respond_to?(“method1”)
```

send – Sends its arguments as 'message' to object (for method call)

```
x = “method1”
obj.send(x)
```

object_id –Returns specific id of object

```
obj.object_id
```

methods – Returns a list of methods defined for the object

```
obj.methods
```

Classes

Class Definition

```
class Ticket
    #class definition
end
```

Class Object Definition

```
tix = Ticket.new
```

Instance Method Definition

```
class Ticket
   def method
         #method definition
   end
end

tix = Ticket.new
#This is how instance methods are called
tix.method
```

Class Method Definition

```
class Ticket
   #This is a class definition
   def Ticket.cheapest(*tickets)
      #Class method definition
   end
end
```

Instance Variables

Defined with @ in front

```
@venue = “City”
```

Class/Object Initialization

```
class Ticket
   def initialize(venue)
         @venue = venue
   end
end

tix = Ticket.new(“City”)
```

Setter Methods

```
class Ticket
   def initialize(venue)
      @venue = venue
   end

   #This is the setter method
   def venue=(venue)
      @venue = venue
   end
end

tix = Ticket.new(“Hall”)
#This is how it's called
tix.venue = “Field”
```

attr_* methods

```
class Ticket
    #write only access
    attr_writer :cost

    #read only access
    attr_reader :price

    #read-write access
    attr_accessor :venue
end

tix = Ticket.new
#This is how to access them
tix.venue = “city”
tix.cost = 55.90
puts “the ticket price is #{tix.price}”
```

Accessing Constants in Classes

```
Class Ticket
   Venue = “City”
end

#This is how it's accessed
puts Ticket::Venue
```

Inheritance

Magazine inherits from Publications class

```
Class Magazine < Publications
   #class definitions
end
```

Modules

Module Definition

```
module MyModule
   #module definition
end
```

Using Modules

```
module MyModule
def function1
end
end

class Test
include MyModule
end

#This is how to call on module functions
test = Test.new
test.function1
```

Nesting Modules/Classes

Nesting can be done like below

```
Class M
   Module N
      Module O
          Class P
          end
      end
   end
end
```

To create instance of Class P

```
p = M::N::O::P.new
```

To force absolute paths (search from top of #hierarchy

```
::P.new
```

Self

What is self at different levels

Location	What self is
Top level	main
Instance method	Instance of object calling the method
Instance method in Module	Instance of class that mixes in Module OR Individual object extended by Module
Singleton method	The object itself

Self as default message receiver

```
Class C
    def C.x
        #method definition
    end

x #This is equivalent to self.x
end
```

Control Flow

If and friends

If

```
if x > 10
    puts x
end

if x > 10 then puts x end

puts x if x > 10
```

If-else

```
if x > 10
    puts x
else
    puts “smaller than 10”
end
```

If-elsif-else

```
if x > 10
    puts “x larger than 10”
elsif x > 7
    puts “7 < x < 10”
elsif x > 5
    puts “5 < x < 7”
else
    puts “smaller than 5”
end
```

Unless – evaluates the opposite way as if

```
unless  x > 10
    puts “x smaller than 10”
end

puts “x smaller than 10” unless x > 10
```

Case statements

You can specify more than one condition for each 'when'

```
x = gets
case x
    when “y”, “yes”
        #some code
    when “n”, “no”
        #some code
    when “c”, “cancel”
        #some code
    else
        #some code
end
```

Case matching can be customized for objects by defining the threequal function

```
def ===(other_ticket)
    self.venue == other_ticket.venue
end

#And this is case example for above def
case ticket1
  when ticket2
    puts "Same venue as ticket2!"
  when ticket3
    puts "Same venue as ticket3!"
  else
    puts "No match"
end
```

Loop statements

```
n = 1
loop do
    n = n + 1
    break if n > 9
end
```

Or

```
n = 1
loop {
    n = n + 1
    next unless n>9 #next skips to nxt
loop
    break}
```

While statements

Equivalent to classic while statement in C

```
n = 1
while n < 11
    puts n
    n = n + 1
end

#OR
n = 1
n = n + 1 while n < 10
puts "We've reached 10!"
```

Equivalent to classic do-while

```
n = 1
begin
    puts n
```

```
      n = n + 1
   end while n< 11
```

Until statements

Opposite of while

```
   n = 1
   until n > 10
     puts n
     n = n + 1
   end
```

OR

```
   n = 1
   n = n + 1 until n == 10
   puts "We've reached 10!"
```

For statements

***For* every value *in* array**

```
   celsius = [0, 10, 20, 30, 40, 50, 60,
   70]

   for c in celsius
     puts "c\t#{Temperature.c2f(c)}"
   end
```

Yield statements / iterator

Yield without arguments

```
   def demo_of_yield
      puts "Executing the method body..."
      puts "Yield control to the block..."
      yield
      puts "Back from the block—finished!"
   end


   demo_of_yield { puts "Now in block!”}
```

Yield with arguments

```
   def yield_an_arg
      puts "Yielding 10!"
      yield(10)
   end
   #argument sent to block thru |x|
   yield_an_arg {|x| puts "#{x}" }
```

Block returns argument

```
   def return_yielding
      puts "code block will do  by 10."
      result = yield(3)
      puts "The result is #{result}."
   end
   return_yielding {|x| x * 10 }
```

Iteration within blocks

```
   def temp(temps)
      for temp in temps
                converted = yield(temp)
                puts
   "#{temp}\t#{converted}"
      end
   end

   celsiuses = [0,10,20,30,40,50,60,70]
   temp(celsiuses) {|cel| cel * 9 / 5 +
   32 }
```

Each statement

```
   [1,2,3,4,5].each {|x| puts x * 10}
```

Or

```
   [1,2,3,4,5].each do |x| puts x * 10 end
```

Exception Handling

Rescue

Begin/end wrapped method

```
   print “Enter a number:”
   n = gets.to_i
   begin
      result = 100/n
   rescue
      puts “your number didn't work”
      exit
   end
   puts result
```

For specific rescue, add Exception name

```
   rescue ZeroDivisionError
```

Rescue in method definition

```
   def multiply(x)
      result = 100/x
      puts result
   rescue ZeroDivisionError #begin x needed
      puts “wrong value!”
      exit
   end
```

Raise

```
   def reraiser(x)
      result = 100/x
   rescue ZeroDivisionError => e
      puts “Division by Zero!”
      raise e
   end
```

Creating Exception Classes

```
   class MyNewException < Exception
   end

   raise MyNewException
```