Ruby - Basics

RUNNING RUBY EXAMPLES

Interactive Ruby Shell

You can then run ruby code right in terminal (or console2). You can copy in multiple lines at a time, try out some of the examples below.

```
$ irb
>> 1 + 1
=> 2
```

TextMate

Within a Ruby text file you can execute ruby.

- Create a new file
- Change to 'Ruby' language (*Shift+Ctrl+Alt+R*, select Ruby)
- Enter the example below

Cheat! Type **#** and press tab.

```ruby
1 + 1 # =>
```

- Press *Ctrl+Cmd+E* to execute the Ruby

```ruby
1 + 1 # => 2
```

Alternately, press *Cmd+R* to run the whole file and print the output to another window.

METHODS

Without arguments

Parenthesis are often optional.

```ruby
def hello_world
  print "Hello "
  puts("World!")
end
hello_world
# => Hello World!
hello_world()
# => Hello World!
```

`print` and `puts` are similar – the latter adds an new-line character.

With arguments

Use lowercase_and_underscores for variables and methods.

```ruby
def hello_world(name)
  puts "Hello #{name}!"
end
hello_world("World")
# => Hello World!
```

With default arguments

```ruby
def hello_world(name = "World")
  puts "Hello #{name.capitalize}!"
end
hello_world "odin"
# => Hello Odin!
hello_world
# => Hello World!
```

CLASSES

Class definition

Class names must be in CamelCase

```ruby
class GreetWorld
  def initialize(name = "World")
    @name = name
  end
  def say_hi
    puts "Hi #{@name}!"
  end
  def say_bye
    puts "Bye #{@name}, come back soon."
  end
end
```

Creating objects (Instantiation)

```ruby
g = Greeter.new("Pat")
# => #<Greeter:0x16cac @name="Pat">
g.say_hi
# =>Hi Pat!
g.say_bye
# =>Bye Pat, come back soon.
```

Constants

Constants are in UPPER_CASE

```ruby
FOO_BAR = 1
# => 1
FOO_BAR = 2
# => warning: already initialized constant FOO_BAR
```

Constants and Classes

Above, class `GreetWorld` is a constant just like `FOO_BAR`. Any "variable" starting with an uppercase letter is a constant, anything starting with a lowercase letter is a variable.

So for constants and classes we use the convention suggested above: ClassesInCamelCase and CONSTANTS_LIKE_THIS.

IF, ELSIF, ELSE STATEMENT

These logic statements are very similar in most programming languages

```
rails = "good"
if rails == "awesome"
  print "Rails is awesome!"
elsif rails == "really awesome!"
  print "Rails is really awesome!"
else
  print "Rails isn't awesome"
end
```

STRINGS

Instant array of strings

```
p %w[ Bob Jim Joe ]
# => ["Bob", "Jim", "Joe"]
```

Types of strings

Interpolation is wonderful.

`'single quote'` and `"double quote"` are both normal strings. Double quotes allow interpolation of arbitrary ruby within the string:

```
quote = "This is a quote"
puts 'single quote: #{quote}'
# => 'single quote: #{quote}'
puts "double quote: #{quote}"
# => double quote: This is a quote

puts "1 + 1 = #{1 + 1}"
# => 1 + 1 = 2
```

Multi-line strings

Want a block of text?

```
puts <<-STRING
This is a mulit-
  line string.
STRING
# => This is a mulit-
# =>   line string.
```

The `STRING` can be anything but must be the same at start and end.

REGULAR EXPRESSIONS

A language within a language, regular expressions are powerful for processing text.

You can use `(groups)` and they are populated into `$1`, `$2`, etc variables.

```
time_str = Time.now.to_s
puts time_str
# => Thu Apr 08 16:05:34 +1000 2010
if time_str =~ /(\d+):(\d+):(\d+)/
  p [$1, $2, $3]
end
# => ["16", "08", "45"]
```

Replacement with gsub

```
"I am a happy man in a can!".gsub(/an/, 'aaaaan')
# => "I am a happy maaaaan in a caaaaan!"
```

Ternary operator

```
time_str = Time.now.to_s
(time_str =~ /Mon/) ? "I hate Monday!" : "Great!"
# => Great!
```

LOOPS

```
for person in ["Dr Nic", "Jack", "Odin"]
  puts person
end

count = 5
while count > 0
  count -= 1
  puts count
end
```

BLOCKS

Pass some code when calling a method.

```
5.times do
  puts "Blocks are powerful"
end
```

Your block of code can have arguments.

```ruby
0.upto(5) do |i|
  puts i
end

5.times do |i|
  puts "Blocks are powerful x #{i}"
end
```

Better loops

```ruby
%w[ Bob Jim Joe ].each do |name|
  puts name
end
```

Blocks with { }

`{ ... }` and `do ... end` are interchangeable.

Typically, use the former for 1-liners.

```ruby
0.times(5) { |i| puts i }

0.times(5) do |i|
  puts i
end
```

Methods accepting blocks

The `yield` method invokes any block passed to the method; pass arguments as necessary.

```ruby
def this_many_times(num)
  num.times do |counter|
    yield(counter)
  end
end

this_many_times(3) do |i|
  puts "Block called # #{i}"
end
# => Block called # 0
# => Block called # 1
# => Block called # 2
```

Test if a block is provided with `block_given?`

```ruby
def this_many_times(num)
  if block_given?
    num.times do |counter|
      yield(counter)
    end
  else
    puts "No block passed!"
  end
end
this_many_times(5) {|i| print i}
# => 01234
this_many_times(5)
# => No block passed!
```

GOT 15 MINUTES?

Try Ruby in your browser

If you want to run through a basic ruby tutorial head over to tryruby.org and try out ruby in your web browser!