Ruby Language

Teacher: Sarah Allen
blazingcloud.net

In this two day class, you will learn the basics of programming in the Ruby language.

Through test-driven and exploratory development, you will become familiar with language syntax and common patterns.

	Monday	Tuesday
9-10:30	Class Overview Ruby Language Introduction	Blocks Mocks and Stubs
10:45-noon	Test-Driven Development ▪ Red-Green-Refactor ▪ Describing a feature ▪ Verifying expectations	Domain-Specific Languages Method Missing
noon-1pm	Lunch	
1-2:30	Conditionals Strings and Symbols	Enumerables (map, inject)
2:45-4	Collections: Hash & Array Iterators	RegEx Web Requests & Files
4-5pm	Q&A	

Requirements for Class

- Ruby 1.8.6 or 1.8.7, to check your version, type:
  ruby –v
- RubyGems (1.3.5 or higher):
  gem -v
- and the Rspec gem (1.3.0):
  gem list rspec
- A text editor you are comfortable with, that offers Ruby syntax highlighting. Recommendations:
  Any Platform free options: Komodo, Aptana, vi, emacs
  Mac: TextMate costs money but a lot of developers swear by it
  Windows: Notepad++ is free and does Ruby syntax highlighting

Ruby Language

Ruby
- Language concepts
- Language syntax
- Common patterns

Class Structure
- Talk
- Live Coding Demonstrations
- In-class coding

Learning methodologies
- Exploration: experiment, play
- Test-Driven Development (TDD)
  - Initially as a learning methodology
  - Later as a development methodology
- Ask questions

Test-First Learning

Similar methodology to Test-First Development with a different purpose and workflow
- Teacher writes the test
- Student implements the code

Test-First Teaching follows the example of Test-First Development (see below), but with an educational twist. In Test-First Teaching, the student begins with a single unit test (written by the teacher). In order to implement the test, the student has to create source code from scratch. The student then tries to compile and run the test; if the test cannot compile, or if the test runs and fails, then the student must go and fix his error. He then moves on to the next test in the lesson.

What is Test-First Development?

Test-First Development (sometimes called Test-Driven Development or Test-Driven Design) is the practice of writing the unit tests first, before you write a single line of implementation code. While this may seem like putting the cart before the horse, there are several good reasons why you might want to do this:

1. **Design** It forces you to think first about the design of the interface to the code, instead of jumping straight to the implementation. Having a well-designed interface is often more important than having an efficient implementation.

2. **Project Management** If you apply a tight cycle of write one test, then write the code to implement that test, then write the next test, your code ends up growing organically. This often (though not always) leads to less wasted effort; you end up writing all the code you need, and none of the code you don't need.

3. **Creation of Tests** Writing tests is often seen as a chore; writing the tests first guarantees that at the end of the project you will have written a suite of unit tests (rather than leaving them until the end and possibly never getting around to it).

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

http://agilemanifesto.org/

Ruby Language Overview

- Dynamically typed
- Interpreted
- Can be modified at runtime
- Object oriented
- Blocks & lambdas
- Nice support for Regular Expressions

2. Getting Started with Ruby (irb)

IRB = Interactive RuBy console
in your terminal, type: irb

In Ruby, everything is an object.

```
>> "test".upcase
>> "test".class
>> "test".methods
```

```
>> f.class
NameError: undefined local variable or method `f' for main:Object

>> "f".class
=> String

>>  2.class
=> Fixnum

>> 2.3.class
=> Float
```

Everything evaluates to some value.

```
>> 2 + 2
>> (2+2).zero?
```

Methods are Messages

```
thing.do(4)
thing.do 4
thing.send "do", 4
```

```
1 + 2
1.+(2)
1.send "+",2
```

Operators are methods

An Introduction to Classes and Methods

We define a new type of object by writing a "class." The behavior of the object is defined in blocks of code we call "methods." As an example we'll create a Calculator kind of object that knows how to describe itself in text. At first, that is all it will know how to do.

Create a file called "calculator.rb" that includes the following content:

```
class Calculator
 def describe
  "I am a really neat calculator!"
 end
end
```

Start irb in the same directory, load the calculator, create a new object and call the describe method you just wrote.

```
>> load "calculator.rb"
>> c = Calculator.new
>> c.describe
```

Next, we'll add a method that performs a calculation:

```
  def add(a, b)
     a + b
   end
```

Restart irb and try it out:

```
>> load "calculator.rb"
>> c = Calculator.new
>> c.add(1,2)
>> c.add(56,45)
```

The same code can be reused with different inputs.

Now, why might we have a calculator object when Ruby can already do built in calculations? Suppose we wanted to keep track of when we did different calculations, how many each calculator had done. We might have two calculators and have each store the number of calculations it had performed. Each object can actually keep its own data with it, even though the code that defines each object is identical.

We call each object an "instance" of the class and so we call this object data an "instance variable."

```
class Calculator
 def initialize
   @num_calculations = 0
 end

  def how_many
    @num_calculations
  end

  def describe
    "I am a really neat calculator!"
  end
  def add(a, b)
    @num_calculations = @num_calculations + 1
    a + b
  end
end
```

More about Classes

Class names are constants, and start with a capital letter.

```
class Thing
  def do_something(a,b)
    a + b
  end
end
```

Classes can inherit from other classes.

Here, SpecialThing is a subclass of Thing.

```
class SpecialThing < Thing

end
```

More about Methods

pairs.

=> is called a "hash rocket"

String Interpolation

Iteration

Method names may end with "?", "!", or "=".

Attributes

To make an instance variable accessible outside of the context of a class, methods need to be declared to access it...

```
class Thing
  def name=(n)
    @name = n
  end
  def name
    @name
  end
end
```

but Ruby provides short-cuts that allow you to access instance variables without declaring methods

```
class Thing
  attr_accessor :name
  attr_reader :created_at
  attr_writer :something
```

```
@var  # instance variable
@@var # class variable
VAR   # constant
```

Class variables belong to the innermost enclosing class or module. Class variables used at the top level are defined in Object, and behave like global variables.

Scope of Constants and Variables

Constants

Constants defined within a class or module may be accessed unadorned anywhere *within the class or module*.

Outside the class or module, they may be accessed using the scope operator, ``::'' prefixed by an expression that returns the appropriate class or module object. Constants defined outside any class or module may be accessed unadorned or by using the scope operator ``::'' with no prefix.

Constants may not be defined in methods.

```
TAX = 0.085

class Payment
   TIP = 0.15

   def calculate(amount)
      amount += amount * (TIP + TAX)
   end
end
```

```
>> TAX
=> 0.085
>> TIP
NameError: uninitialized constant TIP
     from (irb):11
>> Payment::TIP
=> 0.15
>> p = Payment.new.calculate(100)
=> 123.5
```

Class variables are available throughout a class or module body. Class variables must be initialized before use. A class variable is shared among all instances of a class and is available within the class itself.

Class variables belong to the innermost enclosing class or module. Class variables used at the top level are defined in Object, and behave like global variables.

```
class Song
   @@total_songs = 0

   def initialize
     @@total_songs += 1
   end

   def Song.total
    @@total_songs
  end
end
```

```
>> load 'song.rb'
=> true
>> s = Song.new
>> Song.total
=> 1
>> s2 = Song.new
>> Song.total
=> 2
```

Modules

A Module is a collection of methods and constants. Module names, like classes, start with an upper case letter.

For example, Math is a module

You can use modules for establishing a name space

```
>> Math::cos(0)
=> 1.0
>> Math::PI
=> 3.14159265358979
```

Below is an example of using a module in a class.

week.rb

```
module Week
   FIRST_DAY = "Sunday"

   def first_day
      FIRST_DAY
   end

   def Week.weeks_in_month
      puts "You have four weeks in a month"
      4
   end

   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
      52
   end
end
```

main.rb

```
require "week"

class Decade
  include Week
  NUM_YEARS = 10

  def num_weeks
    NUM_YEARS * Week::weeks_in_year
  end
end
```

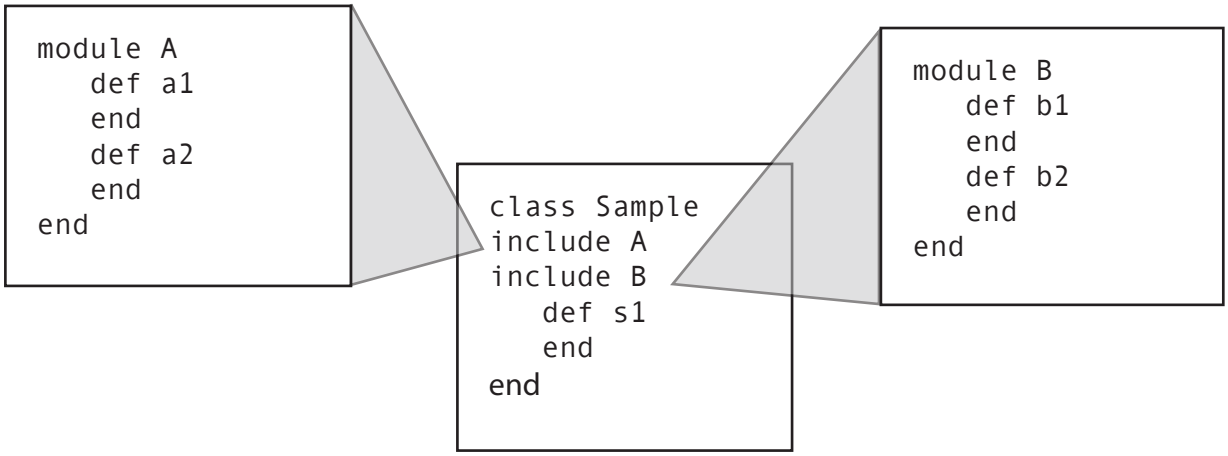
in irb:

```
>> load 'main.rb'
=> true

>> d1 = Decade.new
=> #<Decade:0x534008>

>> d1::FIRST_DAY
TypeError: #<Decade:0x534008> is not a class/module from (irb):4

>> d1.first_day
=> "Sunday"

>> Decade::FIRST_DAY
=> "Sunday"

>> Week:FIRST_DAY
NoMethodError: undefined method `Week' for main:Object from (irb):6

>> Week::FIRST_DAY
=> "Sunday"

>> Week.weeks_in_month
You have four weeks in a month
=> 4
```

Including Multiple Modules

Let us examine the following sample code to gain an understanding of mixins:

Ruby does not support multiple inheritance directly but Ruby Modules have another, wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance, providing a facility called a **mixin**.



in irb:

```
>> samp=Sample.new
>> samp.a1
>> samp.a2
>> samp.b1
>> samp.b2
>> samp.s1
```

Module A consists of the methods a1 and a2.
Module B consists of the methods b1 and b2.

The class Sample includes both modules A and B, and can access all five methods: its own method s1, as well as methods a1, a2, b1, and b2 from module A and B.

Since the class Sample inherits, or 'mixes in' the code from from both the modules you can say the class Sample shows multiple inheritance or is a mixin.

General Syntax Notes

Ruby aims to be elegant and readable so punctuation and boilerplate are minimal.

No semi-colons are needed to end lines, but may be used to separate statements on the same line
Indenting is not significant, unlike python.
A backslash (\) at the end of a line is used for continuation

Parentheses required only as needed to specify precedence.

The dot always wins.

```
>> "Hello".gsub ‘H’, ‘h’
=> "hello"

>> "Hello".gsub("H", "h").reverse
=> "olleh"
```

Commenting your code

```
# this is a comment
```

Variables, Symbols and Constants

There is only one representation of a given symbol in memory,
so it really means "the thing named :this_is_a_symbol" to the ruby interpreter.

In ruby, we prefer symbols over hardcoded globals or strings. They’re very lightweight.

```
:this_is_a_symbol
```

Variables names are composed of letters, numbers and underscores

Global variables are available throughout a program. Every reference to a particular global name returns the same object. Referencing an uninitialized global variable returns nil.

```
var   # a local variable

$var  # global variable
```

Conditionals

if expressions are used for conditional execution.
The values false and nil are false, and everything else are true.

```
if age >= 12 then
     print "adult fee\n"
else
     print "child fee\n"
end
```

```
role = if foo.admin then "admin" else "viewer" end
```

unless is equivalent to: if not

```
unless $baby
     feed_meat
else
     feed_milk
end
```

if and unless may act as modifiers

```
>> print "debug\n" if $debug

>> reject unless role == admin
```

the case expressions are also for conditional execution. Comparisons are done by operator ===.

```
case expr0
when expr1, expr2
     stmt1
when expr3, expr4
     stmt2
else
     stmt3
end
```

```
case $age
when 0 .. 2
     "baby"
when 3 .. 6
     "little child"
when 7 .. 12
     "child"
when 12 .. 18
     # Note: 12 already matched
     "youth"
else
     "adult"
end
```

conditionals brought to you by:
http://web.njit.edu/all_topics/Prog_Lang_Docs/html/ruby/syntax.html#control

Truth

Truth: Everything is true except for:

```
false
nil
```

Therefore:
0 is true
"" is true

Checking for false:

```
if !(name == “superman”) …
if not (name == “superman”) …
```

“unless” provides us with another way of checking if a condition is false:

```
unless human
     status = “superhero”
end
```

Boolean Operators

Evaluates left hand side, then if the result is true, evaluates right hand side.

```
test && set
test and set
```

Evaluates left hand side, then if the result is false, evaluates right hand side.

```
demo || die
demo or die
```

String Interpolation

Ruby provides a concise syntax for inserting the result on an expression into a string.

```
"string #{ruby code} string"
```

```
>> a = "world"
>> puts "hello #{a}"
hello world

>> a = 2
>> puts "hello #{a}"
hello 2

>> a = nil
>> puts "hello #{a}"
hello
```

%w shortcut

Since many arrays are composed of a series of strings, Ruby provides a shortcut for this kind of array creation.

```
>> animals = %w{cat, dog, frog}
=> ["cat,", "dog,", "frog"]
```

Common String Operations

To change case:
`capitalize` - first character to upper, rest to lower
`downcase` - all to lower case
`swapcase` - changes the case of all letters
`upcase` - all to upper case

To rejustify:
`center` - add white space padding to center string
`ljust` - pads string, left justified
`rjust` - pads string, right justified

To trim:
`chop` - remove last character
`chomp` - remove trailing line separators
`squeeze` - reduces successive equal characters to singles
`strip` - deletes leading and trailing white space

To examine:
`count` - return a count of matches
`empty?` - returns true if empty
`include?` - is a specified target string present in the source?
`index` - return the position of one string in another
`length` or `size` - return the length of a string
`rindex` - returns the last position of one string in another
`slice` - returns a partial string

To alter:
`replace` - replace one string with another
`reverse` - turns the string around
`slice!` - DELETES a partial string and returns the part deleted
`split` - returns an array of partial strings exploded at separator
`tr` - to map all occurrences of specified char(s) to other char(s)
`tr_s` - as tr, then squeeze out resultant duplicates
`unpack` - to extract from a string into an array using a template

To iterate:
`each_line` - process each line in a string
`each_byte` - process each byte in turn

```
"cat dog hat".split(" ").join(",")
=> "cat,dog,hat"

"abc".each_byte{|c| printf "<%c>", c}; print "\n"
<a><b><c>
=> nil

"hello".gsub(/[aeiou]/, '*')
=> "h*ll*"

65.chr
=> "A"
```

gsub returns a modified string, leaving the original string unchanged.

gsub! directly modify the string object on which the method was called.

```
"Ru" + "by"
=> "Ruby"

"I" << "love" << "Ruby"
=> "IloveRuby"

myString = "Welcome to JavaScript!"

myString["JavaScript"]= "Ruby"

puts myString
=> "Welcome to Ruby!"

myString[10]= " the land of "

puts myString
=> "Welcome to the land of Ruby!"

s[8..18] = "friends"
=> "friends"

puts myString
=> "Welcome friends of Ruby!"
```

For more information, see:
http://www.ruby-doc.org/core/classes/String.html

Collections

Arrays are sized dynamically and can be of mixed types.

```
>> a = [1, 2, 3]
>> a.push "four"
=> [1, 2, 3, "four"]

>> a.pop
=> "four"

>> a[0]
=> 1
```

%w shortcut

Since many arrays are composed of single words, Ruby provides a concise shortcut.

Hashes are a dictionary,
like a map or an associative array

```
>> states = {"MA" => "Massachusetts",
             "CA" => "California"}

>> states["MA"]
=> Massachusetts
```

```
>> my_hash = {:a_symbol => 3,
              "a_string"=> 4}

>> my_hash[:a_symbol]
=> 3
```

Hashes are an unordered list of key-value

```
>> 4.times do
?>  puts "hello"
?> end

hello
hello
hello
hello
```

```
>> 2.times {puts "foo"}
foo
foo
```

```
my_array = ["cat", "dog", "world"]
my_array.each do |item|
  puts "hello " + item
end
```

```
my_hash = { :type => "cat",
            :name => "Beckett",
            :breed => "alley cat" }

my_hash.each do |key, value|
  puts "My " + key.to_s + " is " + value
end
```

Blocks

Blocks are nameless functions. Blocks were designed for loop abstraction. The most basic usage of blocks is to let you define your own way for iterating over the items in a collection.

```
def talk
  puts "hello"
  yield
  puts "goodbye"
end
```



```
>> talk {puts “whatever”}
hello
whatever
goodbye
=> nil
```

You can call yield multiple times:

```
def talk_much
  puts “hello”
  yield
  yield
  yield
  puts “bye”
end
```



```
>> talk_much {puts “hey”}
hello
hey
hey
hey
bye
=> nil
```

You may also explicitly declare the block as an argument and then use the “call” method to invoke the block. The following example is functionally identical to the first:

```
def talk(&block)
  puts "hello"
  block.call
  puts "goodbye"
end
```

If you want to turn a snippet of code into a block, you can use the lambda function:

```
def contrived_example
    do_something = lambda { puts "foo" }
    do_something.call
end
```

For more information about blocks, lambdas and their close cousin Proc, read this good article:
http://eli.thegreenplace.net/2006/04/18/understanding-ruby-blocks-procs-and-methods/

method_missing

When you send a message to a Ruby object, Ruby looks for a method to invoke with the same name as the message you sent. (There are a bunch of different ways to send the message, but the most common one is just obj.method_name. But you can make the fact that you are sending a message explicit by using obj.send(:method_name)). First it looks in the current self object's own instance methods. Then it looks in the list of instance methods that all objects of that class share, and then in each of the included modules of that class, in reverse order of inclusion. Then it looks in that class's superclass, and then in the superclass's included modules, all the way up until it reaches the class Object. If it still can't find a method, the very last place it looks is in the Kernel module, included in the class Object. And there, if it comes up short, it calls method_missingYou can override method_missing anywhere along that method lookup path, and tell Ruby what to do when it can't find a method.

http://www.thirdbit.net/articles/2007/08/01/10-things-you-should-know-about-method_missing/

```
class Thing
  def method_missing(m, *args, &block)
    puts "There's no method called #{m} here -- please try again."
    puts "parameters = #{args.inspect}"
  end
end
>> t = Thing.new
>> t.anything("ddd",3)
There's no method called anything here -- please try again.
parameters = ["ddd", 3]
=> nil
```

Enumerables

Ruby's Enumerable module has methods for all kinds of tasks that operate on a collection. Most likely if you can imagine a use for the #each method other than simply iterating, there is a good change a method exists to do what you had in mind. Arrays and hashes are Enumerable.

- Collection objects (like Array, Hash, etc.) "mixin" the Enumerable module
- The Enumerable module gives objects of collection classes additional collection-specific behaviors.
- The class requiring the Enumerable module must have an #each method because the additional collection-specific behaviors given by Enumerable are defined in terms of #each

Creating an Enumerable Collection

All you need to do to make your own class respond to all of the Enumerable methods is to define an #each method, since every Enumerable method can be implemented with each.

```
class MyCollection
     include Enumerable
     #lots of code

     def each
          #more code
     end
end
```

How do you know if something is Enumerable?

To see if you can call these methods, you can check if an instance responds to a particular method (preferred in code) or you can test the instance or class:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> a.respond_to? :any?
=> true
>> a.is_a? Enumerable
=> true
>> Array < Enumerable
=> true
```

Try this to see all of the classes that mixin Enumerable:

```
>> ObjectSpace.each_object(Class) { |cl| puts cl if cl < Enumerable}
```

Sources

This section is based on this excellent post:
http://vision-media.ca/resources/ruby/ruby-enumerable-method-examples

Please also see reference docs:
http://ruby-doc.org/core/classes/Enumerable.html

Our test subject for the following examples will be the following array:

```
vehicles = %w[ car truck boat plane helicopter bike ]
```

Standard Iteration With Each

Standard iteration is performed using the each method. This is typical in many languages. For instance in PHP this would be for each however in Ruby this is not built-in this is a method call on the vehicles array object. The sample code below simply outputs a list of our vehicles.

```
vehicles.each do |vehicle|
  puts vehicle
end
```

Modifying Every Member with Map

The map method allows us to modify each member in the same way and return a new collection with new members that were produced by the code inside our block.

```
>> vehicles.map { |v| v.upcase }
=> ["CAR", "TRUCK", "BOAT", "PLANE", "HELICOPTER", "BIKE"]
```

Searching Members With Grep

The grep method allows us to 'search' for members using a regular expression. Our first example below returns any member which contains an 'a'. The grep method also accepts a block, which is passed each matching value, 'collecting' the results returned and returning those as shown in the second example.

```
vehicles.grep /a/
# => ["car", "boat", "plane"]

vehicles.grep(/a/) { |v| v.upcase }
# => ["CAR", "BOAT", "PLANE"]
```

Evaluating All Members With all?

The all? method accepts a block and simply returns either true or false based on the evaluation within the block.

```
vehicles.all? { |v| v.length >= 3 }
# => true

vehicles.all? { |v| v.length < 2 }
# => false
```

Checking Evaluation For A Single Using Any?

The any? method compliments all? in the fact that when the block evaluates to true at any time, then true is returned.

```
vehicles.any? { |v| v.length == 3 }
# => true

vehicles.any? { |v| v.length > 10 }
# => false
```

Enumerable Methods with Complex Data

For our next examples we will be working with vehicles as well, however more complex data structures using Hashes.

```
irb
>> load 'vehicles.rb'
>> $vehicles
```

Collecting a List

The collect method is meant for this very task. Collect accepts a block whose values are collected into an array. This is commonly used in conjunction with the join method to create strings from complex data.

```
$vehicles.collect { |v| v[:name] }
# => ["Car", "Truck", "Boat", "Plane", "Helicopter", "Bike", "Sea Plane"]

$vehicles.collect { |v| v[:name] }.join ', '
# => "Car, Truck, Boat, Plane, Helicopter, Bike, Sea Plane"
```

Finding Members Using The Find Method

The find and find_all methods are the same although different in the obvious fact that one halts iteration after it finds a member, the other continues and finds the rest.
Consider the following examples, we are simply trying to find members that match names, have many wheels, or are ground or air based. The collect method is used to collect arrays of the names for demonstration display purposes, instead of displaying data from the #inspect method.

```
$vehicles.find { |v| v[:name] =~ /Plane/ }[:name]
# => "Plane"

$vehicles.find_all { |v| v[:name] =~ /Plane/ }.collect { |v| v[:name] }
# => ["Plane", "Sea Plane"]

$vehicles.find_all { |v| v[:wheels] > 0 }.collect { |v| v[:name] }
# => ["Car", "Truck", "Bike"]

$vehicles.find_all { |v| v[:classes].include? :ground }.collect { |v|
v[:name] }
# => ["Car", "Truck", "Bike", "Sea Plane"]
```

```
$vehicles.find_all { |v| v[:classes].include? :air }.collect { |v| v[:name] }
# => ["Plane", "Helicopter", "Sea Plane"]
```

Iterating With Storage Using Inject

When you are looking to collect values during iteration the inject method is the perfect one for the job. This method accepts a initialization parameter which is 0 and [] in the case below, this is then passed

```
$vehicles.inject(0) { |total_wheels, v| total_wheels += v[:wheels] }
# => 10

$vehicles.inject([]) { |classes, v| classes + v[:classes] }.uniq
# => [:ground, :water, :air]
```

Regular Expressions

Regular expressions allow matching and manipulation of textual data. They are abbreviated as regex or regexp, or alternatively, just patterns

Using Regular Expressions in Ruby

- Scan a string for multiple occurrences of a pattern.
- Replace part of a string with another string.
- Split a string based on a matching separator.

Regular expressions appear between two forward slashes: /match_me/
They are escaped with a backward slash (\).

Characters That Need to be Escaped
.|()[]{}+\^$*?

Match Method

Match is a method on both String and Regexp classes.

```
>> category = "power tools"
=> "power tools"

>> puts "on Sale" if category.match(/power tools/)
on Sale

>> puts "on Sale" if /power tools/.match(category)
on Sale
```

Match Operator =~

The match operator is like the match method, but returns the index of the match or nil. It is also defined for both String and Regexp classes.

```
>> category = "shoes"
=> "shoes"

>> puts "15 % off" if category =~ /shoes/
15 % off

>> puts "15 % off" if /shoes/ =~ category
15 % off

>> /pants/ =~ category
=> nil

>> /shoes/ =~ category
=> 0

>> category = "women's shoes”
>> /shoes/ =~ category
=> 8
```

Scan

To find multiple matches, the scan method will return an array.

```
>> numbers = "one two three"
=> "one two three"
>> numbers.scan(/\w+/)
=> ["one", "two", "three”]
Split with Regular Expressions
>> "one two\tthree".split(/\s/)
=> ["one", ”two", "three"]
```

gsub

Substitute a string for the matched pattern

```
”fred,mary,john".gsub(/fred/, “XXX”)
=> “XXX,mary,john”
```

gsub may also be called with a block

```
"one two\tthree".gsub(/(\w+)/) do |w|
   puts w
end

one
two
three
```

Title Case

Capitalize All Words of a Sentence:

```
>> full_name.gsub(/\b\w/){|s| s.upcase}
=> "Yukihiro Matsumoto"
```

[abc] A single character: a, b or c
[^abc] Any single character but a, b, or c
[a-z] Any single character in the range a-z
[a-zA-Z] Any single character in the range a-z or A-Z
^ Start of line
$ End of line
\A Start of string
\z End of string
. Any single character
\s Any whitespace character
\S Any non-whitespace character
\d Any digit
\D Any non-digit
\w Any word character (letter, number, underscore)
\W Any non-word character
\b Any word boundary character
(...) Capture everything enclosed
(a|b) a or b
a? Zero or one of a
a* Zero or more of a
a+ One or more of a
a{3} Exactly 3 of a
a{3,} 3 or more of a
a{3,6} Between 3 and 6 of a

Rspec Cheat Sheet

```
require "thing"

describe Thing do

 before do
   @thing = Thing.new
 end

  it "should check universal stability" do
    @thing.answer.should == 42
  end

 it "should enforce universal stability" do
    @thing.val = 3
    @thing.enforce_stability
    @thing.val.should == 42
  end

end
```

loads the file tested by the spec

describe [ClassName] do

- **describe** declares what is being tested and defines a context

- The beginning of each example is marked by "**it**" followed by a description of expected behavior in quotes

- the first lines of each spec set up the conditions required for the test to succeed
- if any line of the example fails to execute, the spec fails.
- **object.should** and
- **object.should_not** are used in rspec to compare actual to expected values (see the next page for syntax for the different types of comparisons)

Read more about RSpec

http://rspec.info/

http://www.pragprog.com/titles/achbd/the-rspec-book

```
target.should equal <value>
target.should not_equal <value>

target.should be_close <value>, <tolerance>
target.should_not be_close <value>, <tolerance>

target.should be <value>
target.should_not be <value>

target.should be_predicate [optional args]
target.should_not be_predicate [optional args]

target.should be < 6
target.should == 5
target.should_not == 'Samantha'

target.should match <regex>
target.should_not match <regex>

target.should be_an_instance_of <class>
target.should_not be_an_instance_of <class>

target.should be_a_kind_of <class>
target.should_not be_a_kind_of <class>

target.should respond_to <symbol>
target.should_not respond_to <symbol>

lambda {a_call}.should raise_error
lambda {a_call}.should raise_error(<exception> [, message])
lambda {a_call}.should_not raise_error
lambda {a_call}.should_not raise_error(<exception> [, message])

target.should include <object>
target.should_not include <object>

target.should have(<number>).things
target.should have_at_least(<number>).things
target.should have_at_most(<number>).things

target.should have(<number>).errors_on(:field)

lambda { thing.destroy }.should change(Thing, :count).by(-1)
```