Ruby Basic

An introductory two day course on Ruby programming

- by Peter Marklund (http://marklunds.com)

License

Hosted at: https://github.com/peter/ruby_basic

Course Introduction

Meet the Teacher

- A web developer specialized in Ruby on Rails
- Blogs at http://marklunds.com
- Can be seen at the Stockholm Ruby User Group meetups as well as the Nordic Ruby conference

Course Objective

To be comfortable with the **fundamentals of the language** and have the confidence to continue exploring the language on your own. To have a good grasp of commonly used language features and **features that make Ruby different** from other languages.

Disclaimer

This course does *not* offer comprehensive coverage of the Ruby language. Please refer to other resources for that. I recommend the book "The Ruby Programming Language" by David Flanagan and Yukihiro Matsumoto.

Prerequisites

- Object oriented programming experience
- A computer (laptop) with Ruby 1.9 and an editor/IDE installed

Schedule

Time	Day 1	Day 2
9:00-9:50	Introduction Structure and Execution	Classes and Objects
10:00-10:50	Datatypes Control Structures	Modules Methods and Operators
11:00-12:00	Exercises	Exercises
12:00-13:00	Lunch	Lunch
13:00-13:50	Variables and Constants	Blocks and Proc Objects Reflection and Meta Programming
14:00-14:50	Boolean Expressions Assignment	Standard Library RubyGems and Bundler
15:00-17:00	Exercises	Exercises

Course Material

- Slides
- Code examples
- Exercises

Presentation of Participants

- Company and role
- Programming experience
- Expectations on the course

Course Evaluation

At the end of the course I will ask you to fill out an evaluation

Chapter 1

Ruby Introduction

Meet Ruby

- Generic, interpreted, reflective, garbage collected
- Optimized for programmer productivity and happiness
- Concise but readable and clean syntax
- Pure OO - everything is an object (including primitives, classes, and nil)
- Supports procedural and functional programming styles
- Strong Dynamic typing / Duck Typing
- Open Source (GPL or "The Ruby License")

Ruby Timeline

- 1995 - Ruby was released in Japan
- 2000 - First english book published
- 2004 - Ruby on Rails was released
- 2010 - Ruby 1.9.2 released
- 2011 - Several alternative implementations of Ruby such as JRuby and Rubinius are available

Kent Beck on Ruby

"I always thought Smalltalk would beat Java. I just didn't know it would be called Ruby when it did"

Ruby and Happiness

Bill Venners: "So basically Ruby helps me enjoy my life by helping me get my job done more quickly and with more fun?"

Yukihiro Matsumoto: "It helps me do that. I'm not sure Ruby works for you, but I hope so"

Chapter 2

Structure and Execution

Line Breaks and Semicolons

- Statements are separated by line breaks
- You can put several statements on one line if you separate them by semicolon
- After an operator, comma, or dot in a method invocation you can have a line break and Ruby will know that the statement continues on the next line
- You can have line breaks in strings

Execution and I18n

- Single pass interpreted language (no compilation)
- Everything, including the definition of classes, happens at runtime, and can be changed at runtime
- No main method
- Source code is ASCII by default but can be configured to be utf-8 by using a comment at the top of the file
- Ruby can be executed from the command line with the `ruby` command or interactively with the `irb` command

Loading Ruby Code

Source code: load_path.rb

- You use the require method to load Ruby code from another file
- The require method is also used to load code from the standard library and third party libraries (RubyGems)
- By convention Ruby code lives in a lib directory. For Ruby to find the file to load, the lib directory needs to be in the Ruby load path ($: / $LOAD_PATH)
- When invoking ruby on the command line you can supply load paths with the -I option

Executing External Programs

Source code: external_programs.rb

- Use the system method to execute an external program on your system. The system command returns true for a zero exit status. Process id and exit status are available in the global variable $?.

- Use backquotes (`command`) or %x[command] to capture the output from an external program

- For more advanced process management with control over stdout, stdin, and stderr, see the Open3 module in the standard library

Identifier Names

```
MyClass, MyModule

MY_CONSTANT = 3.14

my_method

local_variable = 3.14

@instance_variable

@@class_variable

$global_variable
```

Method Name suffixes

```
dangerous_method!

query_method?

setter_method=
```

Chapter 3

Datatypes

Numeric

Source code: numeric.rb

- All numbers in Ruby are objects with the base class Numeric
- Most numbers that you come across will be objects of the Fixnum or Float subclasses
- The BigDecimal class in the std lib is more appropriate than Float for financial calculations as it avoids rounding errors
- The standard library also has the classes Complex and Rational

String

Source code: string.rb

- Text in Ruby is represented by mutable objects of the String class
- Strings are encoded sequences of characters with UTF-8 and multi-byte support
- There are double and single quoted string literals
- Common string operators for concatenation and random access: `+`, `<<`, `[]`
- Strings can interpolate any expression with #{}
- Some useful String instance methods: `size`, `each_char`, `each_line`, `empty?`, `gsub`, `strip`, `upcase`

Array

Source code: array.rb

- Arrays are resizable, untyped, and mutable sequences of objects that are accessed by index (zero based)
- Array and String share several operators: `+`, `<<`, `[ ]`
- The Array class mixes in the Enumerable module and this gives you access to methods like each, select, map, and inject etc.

Hash

Source code: hash.rb

- Hashes are associative arrays with key-value pairs
- Keys can be any objects but must be unique according to the eql? method and have a hash method that returns a Fixnum. Keys are often symbols.
- In Ruby you can use hashes as named method arguments
- Hash implements the Enumerable module

Symbol

Source code: symbol.rb

- Symbols in Ruby are immutable strings
- Symbol literals are a string or identifier prefixed by a colon: `:my_symbol`
- Symbols are typically used as hash keys. They are also used to represent method names
- Symbols have better performance characteristics than strings - less memory usage and faster comparisons
- You can make a string object immutable by invoking `freeze` on it
- You can convert between strings and symbols with to_s and to_sym

Range

Source code: range.rb

- A range object represents all values between a start and end value, such as `1..100`

- When created with two dots a range includes the last value. With three dots it doesn't.

- Some useful Range methods: `each`, `step`, `include?`, `member?`

- Floating point ranges are continuous whereas alphanumeric ranges are discrete

- You can create a discrete range between arbitrary end point objects as long as they implement <=> and the succ methods

Regexp

Source code: regexp.rb

- Regular expression literals are delimited with slashes: `/ . . . /`
- To match a regular expression against a string you typically use the `=~` operator or `String#match`
- You can do regular expression replacements in strings using `String#gsub`
- The syntax and semantics for regular expressions is similar to other languages. You can use character classes, grouping, capturing, repetition, anchors, and statement modifiers (ignore case, multiline etc.)

Struct

Source code: struct.rb

- *"A Struct is a convenient way to bundle a number of attributes together, using accessor methods, without having to write an explicit class"*
- *`Struct.new` creates a class with the attribute accessors that you specify*
- *`OpenStruct.new` creates an object that can respond to any attribute accessors*

Chapter 4

Control Structures

Conditionals

Source code: conditionals.rb

- Create if statements with the keywords: `if`, `elsif`, `else`, `end`
- If statements return the last evaluated expression
- The ? operator (conditional/ternary operator) is a nice alternative to if statements for one-liners
- For complex branching Ruby offers powerful `case` statements based on the === (triple equals / case equality) operator

Loops

Source code: loops.rb

- Enumerable collections are usually iterated in Ruby with the `each` method
- If you prefer, you can use `for..in` loops instead of `each`
- Similar to other languages, Ruby also supports `while` and `until` loops

Exceptions

Source code: exceptions.rb

- Exceptions are thrown with the `raise` keyword and caught with the `rescue` keyword

- Use an `ensure` block for code that should always execute. Use the `retry` keyword to re-execute your code.

- Ruby has a pre-defined class hierarchy with `Exception` as the base class

- If you are raising exceptions with specific semantics in your application or API, consider defining a custom exception class

Chapter 5

Variables and Constants

Variable Scope

Source code: variable_scope.rb

- Define global variables (if you must) with the $ prefix
- Define class variables with the @@ prefix
- Define instance variables with the @ prefix
- Names of local variables have no prefix. You define local variables by assigning them a value.

Constants

Source code: constants.rb

- Constant names start with an uppercase letter
- To define a constant, you assign a value to it
- Constants are typically defined inside classes or modules
- To access a constant inside a module or class you use the :: (double colon) syntax
- Ruby will issue a warning if you change the value of a constant

Globals

A selection of useful global variables and constants:

```
ARGV (command line args)
ENV (environment variables)
RUBY_VERSION
STDIN ($stdin)
STDOUT ($stdout)
$$ ($PROCESS_ID)
$? (exit status of process)
$: ($LOAD_PATH)
```

Chapter 6

Boolean Expressions

Truth

Source code: truth.rb

- The only values in Ruby that are false in a boolean sense are:
  `false, nil`
- Unlike in some languages, zero (0) and the empty string (“”) evaluate to true in if statements

nil

- The absence of an object is in Ruby represented by the nil keyword.
- The nil keyword holds an object that is accessible everywhere and it is the only instance of NilClass < Object < BasicObject
- In boolean expressions, `nil` is false

Equality

Source code: equality.rb

- Every Ruby object is assigned a unique object_id by the interpreter
- The Object#equal? method returns true only if two objects have the same object_id, i.e. if they are the same object in memory
- The == operator should be used to determine if two objects hold the same contents

Boolean Operators and Assignments

Source code: boolean_operators.rb

- Boolean expressions appear in if statements and other control structures

- Boolean AND is represented by the && operator and boolean OR by the || operator

- Use extra parentheses for grouping if this helps clarify the expression to the reader

- It is common in Ruby to mix assignments with boolean expressions

Chapter 7

Assignment

Assignment Idioms

```
a, b = b, a
```

```
a = 1; b = 1
```

```
a = b = 1
```

```
a += 1
```

```
a, b = [1, 2]
```

```
a = b || c
```

```
a ||= b
```

Chapter 8

Classes and Objects

Defining a Class and Instantiating an Object

Source code: person.rb

- The class keyword defines a class
- The def keyword defines a method
- The initialize method is the constructor
- Instance variables have the @ prefix
- Objects are created by invoking the new method on the class
- Methods are invoked with the object.method notation. Method parentheses can be omitted when invoking a method

Class Inheritance

Source code: programmer.rb

- To inherit from another class we use class MyClass < BaseClass
- The keyword super invokes a method in the base class
- Most classes in Ruby inherit from the built in Object class. The Object class has the Kernel module mixed in and inherits from BasicObject. BasicObject sits at the top of the class hierarchy and represents a bare bones (blank slate) object with a minimal set of methods.

Getter and Setter Methods

Source code: getter_setter.rb

- Instance variables are always private and need to be accessed through getter/setter methods

- For the set method, Ruby allows you to use an equal sign name suffix and invoke the method with normal variable assignment syntax: `object.my_attribute = value`

attr_accessor

Source code: attr_accessor.rb

The `attr_accessor` macro offers a concise way to generate the getter and setter methods for the instance variables (attributes) of a class.

The Current Object

Source code: current_object.rb

- Wherever Ruby code is executing there is always the concept of a current object. The current object is always returned by the `self` keyword.

- A method in Ruby is always invoked on an object. If you don't specify which object you are invoking a method on then it is invoked on `self`.

Duck Typing

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Duck Typing Explained

Source code: duck_typing.rb

- In Ruby, you don't specify the class or interface of your variables or method arguments. It's not the class of an object that determines its type, but rather what methods it responds to.

- Library examples of Duck Typing are Enumerable (each method), Comparable (<=> method), the << (append) operator (Array, String, IO, etc.), the call method for Rack applications.

- Duck typing allows for flexibility and polymorphic substitution. This can help you in your design, testing, and performance tuning.

Duck Typing and Protocols

In Duck Typing, how do we define the type of the object, i.e. its interface or protocol? A protocol can be defined in terms of a set of methods that an object responds to. Each method is defined by its name, its argument signature, its return value, its contract (what it requires and what it promises) and its semantics (including any side effects). The protocol is specified in documentation and should follow conventions in the Ruby community. The Rack web server protocol is a good example of a formalized Ruby protocol.

Chapter 9

Modules

Mixins

Source code: mixins.rb

- The module keyword defines a module
- Modules are a collection of methods, constants, and class variables
- By using the include keyword you can “mix in” one or more module in a class. This is Ruby’s answer to multiple inheritance.
- Unlike classes, you cannot instantiate a module

Mixins and The Inheritance Chain

Source code: inheritance_chain.rb

- When you mix in a module in a class the module gets added right above the class in the inheritance chain, i.e. between the class and the superclass

- When you are doing method overrides, you can use `super` to walk up the inheritance chain

- If multiple modules implement the same method, then the module included last will take precedence. This property is used by Ruby on Rails to allow overriding of framework methods.

Namespaces

Source code: namespaces.rb

- The second use case for modules in Ruby is namespacing
- If you define a class inside a module it becomes scoped in that modules "namespace"
- Modules and classes can be nested
- Use double colon to access a class inside a namespace: `MyLibrary::Payment::Account`

Modules vs Classes

- Modules model characteristics or properties of entities or things. Modules can't be instantiated. Module names tend to be adjectives (Comparable, Enumerable etc.). A class can mix in several modules.

- Classes model entities or things. Class names tend to be nouns. A class can only have one super class (Enumeration, Item etc.)

Chapter 10

Methods and Operators

Defining and Invoking Methods

- Methods are identified by their name only. There is no overloading on argument signatures.
- You can have class methods and instance methods
- Methods can be public, protected, or private
- The last evaluated expression in a method is the return value
- When invoking a method argument parentheses are optional
- Methods always have a receiver. The implicit receiver is self.
- You can optionally pass a code block to a method

Method Arguments

- Arguments can have default values
- A method can have a variable number of arguments
- A Hash argument can be used to emulate named arguments

Variable Number of Arguments

Source code: variable_arguments.rb

You can have one method argument, usually the last one, prefixed with a star (*). That argument will then be assigned an array of values containing zero or more arguments passed

Method Visibility

Source code: method_visibility.rb

- You set method visibility with the keywords `public`, `protected`, `private`
- Methods are public by default
- Private methods always have `self` as the impicit receiver. You cannot have an explicit receiver when invoking a private method.
- Protected methods can be invoked from another object, but only if that object has a superclass in common with the receiver

Singleton Methods

Source code: singleton_methods.rb

- You can define a method on an individual object. In Ruby we call such an object a singleton method and it is only available for that one object.

- Singleton methods live in an anonymous class of the object called the Eigenclass (a.k.a. Metaclass, Singleton class)

Class Methods

Source code: class_methods.rb

- You can define class methods by prefixing the method name with self: `def self.method_name`
- Class methods are invoked just like instance methods, but on the class object: `MyClass.class_method`
- Class methods are inherited and can be overridden in subclasses.
- Class methods are implemented as singleton methods on the class object

Method Lookup

When you invoke a method, Ruby goes looking for it:

1. Check for a singleton method with a matching name

2. Search the class for an instance method with a matching name

3. Search instance methods of modules included in the class. Modules included later are searched first.

4. Move up the inheritance hierarchy and repeat steps 2 and 3

5. If no method is found, invoke method_missing instead

Method Operators

Source code: method_operators.rb

- Many operators in Ruby are implemented as methods and can be overridden

- `2 + 2` is equivalent to `2.+(2)`, i.e. we are invoking the plus method on the object 2 and passing along the argument 2

- Here are some examples of operators that can be useful to override in your classes: `<, <=, ==, =>, >, <=>, !=, <<, +, -, /, *`

The Comparable Module

Source code: comparable.rb

- If you have a class that needs sorting then you should define the <=> method (the comparison method). The return value should be -1, 0, or 1.

- If you include (mix in) the Comparable module from the standard library, you get the following comparison methods added: `<`, `<=`, `==`, `=>`, `>`

The Enumerable Module

Source code: enumerable.rb

"The Enumerable mixin provides collection classes with several traversal and searching methods, and with the ability to sort. The class must provide a method each, which yields successive members of the collection. If Enumerable#max, min, or sort is used, the objects in the collection must also implement a meaningful <=> operator, as these methods rely on an ordering between members of the collection."

- from http://www.ruby-doc.org/core/classes/Enumerable.html

The Variable/Method Ambiguity Gotcha

Source code: variable_method.rb

Since local variables and methods have the same naming scheme and since methods can be invoked without parentheses, the two can sometimes get mixed up:

```
foobar = 5 # Assigns local variable

self.foobar = 5 # Method invocation

foobar # Local variable if it exists, otherwise method
```

Methods and Parentheses Gotcha

Source code: method_parentheses.rb

- Leaving out parentheses when invoking methods can sometimes help readability but sometimes it can lead to ambiguities
- When invoking a method, never put space between the method name and the opening argument parenthesis
- When defining a method, by convention there is no space between the method name and the opening parenthesis for the parameter list

Chapter 11

Blocks and Proc Objects

Blocks

Source code: blocks.rb

- Code blocks can optionally be passed as the last argument to a method. From within the method the block is invoked with the keyword `yield`
- Similar to an anonymous method, a code block can accept arguments
- A code block is more than the code to be executed, it also contains the variable bindings in the scope where it's defined (it's a closure)

Proc Objects

Source code: blocks.rb

- A Proc object is an object representation of a code block that can be assigned to variables, passed around as arguments etc.
- You create a proc object with `Proc.new` and you invoke it with the `call` method
- You can convert back and forth between proc objects and blocks with the `&` operator

Lambdas

Source code: blocks.rb

- You can also create Proc objects with the `lambda` keyword

- Lambdas differ from proc objects in how they deal with return statements and are stricter about the number of arguments you pass to them. Lambdas are closer to methods in behavior

- As of Ruby 1.9, there is an alternative syntax for creating lambdas:

```
->(args) {...}
```

Symbol Shortcut for Code Blocks

Source code: blocks.rb

Ruby provides syntactic sugar for the common use case where you want to create a code block that takes a single argument and invokes a method on it. You can create such a code block by prefixing a symbol with `&` (ampersand):

```
%w(cat dog).map(&:upcase)
```

Chapter 12

Reflection and Meta Programming

Introspection

Source code: introspection.rb

- Ruby offers a variety of methods for introspecting the class and module hierarchy of an object: `class`, `superclass`, `ancestors`, `instance_of?`, `is_a?`

- You can fetch the list of class and instance methods from a class object by invoking `methods` or `instance_methods`. You can also check if an object implements a certain method with `respond_to?`

Dynamic Method Dispatch

Source code: method_missing_and_send.rb

- If you invoke a non-existent method on an object a NoMethodError will be raised. However, if you add a method called `method_missing` to your class you can intercept any invocations of non-existent methods and do something clever instead.

- You can use `method_missing` for dynamic method names, proxies and delegation etc.

- You can use the `send` method to invoke a method with a name determined at runtime. The `send` method allows you to invoke private methods.

Open Classes and Method Aliases

Source code: open_classes.rb

- In Ruby all class definitions are dynamic and classes can be re-opened and added to or changed at runtime

- You can use alias_method to create synonyms for method names (like `Array#size` and `Array#length`)

- You can add methods to classes in the standard library. This approach has been used successfully by the ActiveSupport library from Ruby on Rails.

Hooks

Source code: hooks.rb

In Ruby you can hook into certain events in the lifecycle of classes and modules. When a class is subclassed, its `inherited` method is invoked. When a module is mixed in to a class, its `included` method is invoked. When a class cannot be loaded, `const_missing` is invoked. The Ruby on Rails framework uses `const_missing` to load classes automatically as they are used.

Evaluating Code

Source code: eval.rb

- The `eval` method takes a string that will be interpreted as code and executed on the fly

- The `eval` method can also optionally take a Binding object with the variable context to evaluate the code in. To get the Binding object of the current scope you invoke the `binding` method.

- The `class_eval` method executes in the context of a class definition. The `module_eval` method is synonymous.

- The `instance_eval` method executes code "inside an object" by setting self (the current object). This is a common approach to creating DSLs (Domain Specific Languages).

Defining Methods Dynamically

Source code: define_method.rb

- As an alternative to class_eval, you can use `define_method` to define methods with dynamic names.

- You pass a block to `define_method` that will be called (with or without arguments) when the method is invoked

Chapter 13

Standard Library

Overview of the Standard Library

DRb (distributed object system)

net/ftp, net/http, net/imap, net/pop, net/smtp, net/telnet, open-uri, openssl

Base64, Digest (HMAC, SHA256, and SHA384)

optparse (option parsing, see the Trollop library for an alternative)

Pathname, FileUtils

Logger

CSV

REXML (pure Ruby XML library, Nokogiri is a good alternative)

YAML (popular serialization alternative to XML, JSON)

Thread (see also Fiber in Ruby core)

Timeout

...

For a complete list, see: http://www.ruby-doc.org/stdlib

Files

Source code: files.rb

- The File class has useful class methods for getting information about an existing file.
- The File class mixes in the IO module that has methods like read, readlines, and open for reading and writing files
- You can use the Tempfile class to create temporary files
- The Dir class can create directories and list their files
- FileUtils provides Unix style file operations like cp, mv, rm, mkdir etc.

Unit Testing

Source code: unit_testing.rb

- Testing and particularly unit testing is very wide spread and valued in the Ruby community
- Unit tests typically test at the class or method level
- Test::Unit is the most common testing library
- The Ruby 1.9 standard library also comes bundled with the MiniTest library. Minitest has a richer set of features than Test::Unit and supports a specification style (BDD) syntax in addition to the Test::Unit syntax.
- RSpec and Cucumber are powerful testing libraries that have popularized the specification and natural language approach to writing tests.

Documenting your Code

Source code: rdoc.rb

- The RDoc library uses source code comments in order to generate API documentation in HTML format

- RDoc comments typically precede class, module, and method definitions and use a simple markup syntax. A method comment typically contains a description, a parameter list, and usage examples.

- Yard is an alternative library that offers RDoc compatibility and adds @param style tags and ability to document parameter types etc.

- TomDoc is a lightweight documentation library from GitHub that emphasizes simplicity and human readable syntax.

Rake

Source code: rakefile.rb

- The Rake library allows you to define tasks that can be executed from the command line

- A task is a named block of Ruby code that may depend on one or more other tasks to be executed before it can execute. Tasks are defined in a file called Rakefile.

- Like the C build tool Make, Rake was invented to manage build dependencies. It can also be used simply as a convenient way to access utility scripts from the command line. Thor is an alternative library for writing command line scripts.

- Invoking rake -T on the command line gives you a list of all tasks and their descriptions

Chapter 14

RubyGems and Bundler

About RubyGems

- RubyGems is a packaging system for Ruby libraries
- Similar to a Linux packaging system like apt-get, RubyGems provides versioning, dependency management, and automated installation
- When you create a RubyGem from your library it gets bundled into a single .gem file and uploaded to the RubyGems.org repository
- A RubyGem must contain a .gemspec file with meta information about the library, its version and dependencies, etc.
- Examples of RubyGem commands are install, uninstall, list, build, push, and unpack. Invoke `gem help commands` on the command line for more.

Bundler

Source code: bundler.rb

- Bundler is a library that helps you manage the RubyGem library dependencies of your application

- The library dependencies are specified in a file called Gemfile. For each gem dependency you should specify a version such as “3.0.0” or “~> 1.4.2”

- Use `bundle install` and `bundle update` on the command line to make sure your application has the set of libraries that it needs

Creating RubyGems with Bundler

- The Bundler library has support for creating and publishing RubyGems

- In order to create a new RubyGem, invoke `bundle gem gem_name` on the command line. That command will create an empty RubyGem with a .gemspec file and rake tasks for building and releasing your gem.