Ruby – General

IRB

IRB stands for interactive Ruby, when run in the terminal it creates an interactive session using the Ruby interpreter. This type of interactive loop is called a REPL (Read Evaluate Print Loop), closing the loop using “exit” delete all variables/objects.

A set of code can be created in a text file, called a procedure, and then required (imported) into the REPL, this will automatically run the set of code line by line. The suffix for Ruby code files is “.rb”. Requiring files into IRB can be done using the “irb -r <filename>” command, once run the session will stay in the REPL keeping all objects created. Calling procedures normally using the command line, will exit once the code is executed.

When a main program function is started in Ruby, a set of useful objects are automatically created, for example, the object class of numbers is created. These objects have been set to know how they interact with other objects created and their useful properties.

Messages

Messages are the way to tell objects to return something (return value). An example of a message is one.integer?, this tell the one object to perform its predefined function (procedure) of integer? and return a value. As Ruby is object oriented, all messages in Ruby are predefined in the object class, for example even 1 + 1, as the Ruby interpreter simply changes this to 1.+(1). The set of messages defined for an object it called its interface, to see a full interface use the “.methods” method.

It is possible to send many methods to an object, one after another, this is called chaining. Each method will performed then the returned value used for the next method and so on.

Methods

Predefined function in Ruby made up of a set of expression which returns a value. They can be either standard such as ‘print’ or user defined via:

def *method\_name*(arguments)

expression

end

The method will either return the value of the last expression or the value of return… Some standard methods can be implemented directly to an object, defined in the objects interface. Using a return statement in a method or loop, will stop the procedure and return the value specified.brea

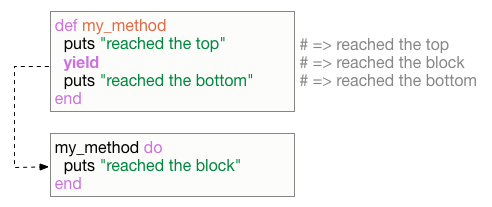
Arguments can be set to default values using the assignment operator, e.g. ‘account=100’ will assign a default value of 100 to the argument ‘account’, if no argument for ‘account’ is passed.

Splat arguments can be given at the end of the argument list. These are defined by ‘\*splat’ and any additional arguments given will be passed into method as an array in called ‘splat’ or other user defined value.

Depending on if the method has a yield section, when calling there can be the option add a block immediately after, like so:

*method\_name*(argument) {|item| expression}

The block will then be passed into the method at the yield, allowing for quick calling of common methods such as .each or .map, but allowing the function to perform different actions each time, depending on the block.



Objects

Almost everything in Ruby is an object. Each object has a specific object ID which can be seen using the method object.object\_id. names can be assigned to object ID using assignment operators:

* name = string (object) assigns name to object with value of string
* name ||= string assigns value to name if name isn’t already assigned

Objects are typically assigned a name using =, for example: ‘A = 19’ will assign ‘A’ to the object ID which equals 19. If another name is assigned to the same object, for example ‘B = A’, ‘B’ will also be assigned to the same object ID as ‘A’. Whatever an object contains is called the objects ‘state’.

The value of the object will only change if a mutating method is applied to the object name or object ID. Mutating methods generally have end with a ‘!’, however concatenation, indexing methods, and setter methods, e.g. ‘A << B’, ‘name[i] = 10’, and ‘person.name = value’ respectively are also mutating. Therefore, if two names link to the same object ID and one name has a mutating method applied, the value of both names will change. If the value is changed via other means, such as += or ‘a = a – b’ a new object ID will be created, and the name re-assigned to it.

Objects which can be mutated are called mutable objects, for example strings and arrays.

Objects which cannot be mutated are called immutable objects.

Constants, for example numbers and specified values, always start with a capital letter, but also tend to be all capitals for clear reading, for example “RUBY\_VERSION”.

Comments

Comments can be added into Ruby code by adding ‘#’. Anything after the hashtag on that line will not be read and is considered a comment.

Symbols

Symbols are immutable strings defined by ‘:string’. There can only be one symbol per name and looking up symbols is much faster than strings.

Arrays

The Array class creates instances which can store many other objects inside themselves, these can be 2D or Multidimensional. Arrays are created by:

Array.new(<size>) { |i| <input for each index>}

Arrays have a zero index, meaning they start from 0.

Call an array via:

array[index][level2 index][etc]

Push values into an array using either

array[index] = <value>

array[index].push(<value>)

Delete values from an array by:

array[index].delete\_at(<index>)

Arrays can be joined using the ‘+’ method, which will simply combine them together and return the result.

Arrays can also be iterated over using loops

Hashes

Hashes are like a dictionary of values, with each entry being having a name (key) and value. Hashes can be created in two ways:

new\_hash = Hash.new(*default\_value*)

new\_hash = { “key” => “value”, “key2” => “value”, …}

Keys and values can be any object, however if they are symbols they will be immutable (not be able to be changed once created).

Values in hashes can be called via indexing: hash[key] will return the value associated with the key.

Booleans

Booleans are statements which return true, false, or nil. Booleans have one expression on each side and if correct will return true, the common operators are:

* == equal to
* != not equal to
* .include? includes

Booleans can be combined and modified using:

* && Both must be true
* || Either must be true
* ! returns not the value

Control Flow

Control flow works on conditions which evaluate true and false statements, generally created by Booleans. Control flow will operate the block of code indented inside and must be ended with an ‘end’ statement. Common control flows are:

* if if true, run block
* elsif follows if statement and will run code if true and original doesn’t match
* else follows if statement and will run code if no other statement matches
* unless will run code unless the statement is true
* while will run code while the statement is true

An if statement can be run using the tertiary operator, saving code and time:

Boolean? (run if true) : (run if false)

Tertiary operators can be combined simply by adding more Booleans.

Loops

There are several loops which will continue either for a set number of iterations or until certain criteria are met. All will perform the expression indented then must end with an ‘end’. All loops can be stopped by the expression ‘break’ or the expression ‘return’. Loop iterations can also be skipped if ‘next if *boolean*’ condition is true.

**For** loops will pass in the iteration ‘i’ to the loop:

for *‘i’* in (*range*)

*expression*

end

**Loop** loops will continue until break is read.

Loop do

*expression*

*break*

end

**While** loops will continue while the Boolean is true

while *boolean*

*expression*

end

**Until** loops will continue until Boolean is true.

until *boolean*

*expression*

end

Ranges

Ranges will output an array of numbers or letters between the respective range.

* (1..10) will range from 1 to 10, including 10, in increments of 1
* (1…10) will range from 1 to 10, excluding 10, in increments of 1
* (“a”..”x”) will range from a to x, in increments of each letter
* (1..10).step(2) will range from 1 to 10, including 10, in increments of 2

Case Statements

Case statements match when an object is equal to the ‘when’ clause

case *object*

when *value*

*expression*

when *value*

*expression*

*else*

*expression*

end

Case statements come into particular use when an object such as a string can be many different known values.

Proc’s

Proc’s are blocks of code saved as objects, defined by:

name = Proc.new {|n| expression}

Procs can be called using ‘&name’ saving time rewriting certain bits of code. Procs do not check the number of arguments matches the required. If called in a method and a return is triggers the proc will return control back outside the method, meaning any remaining code is skipped.

Multiple procs can be passed into methods, whereas only one block can.

Lambdas

Lambda’s are very similar to process in both definition and calling. However, lambda’s check the number of arguments is correct, so if to many arguments are passed, it will return an argument error. Also if a return is triggered the lambda will return control flow back to the method, instead of skipping it.

Lambda are however still part of the Proc class.

Classes

Class is the type of object and defines the object attributes and what methods can be applied to it. An example of a class is Integer or String. Classes are defined as:

class ClassName

def initialize(arg,..)

@atributes = arg

…

end

end

object instances for the class (such as the string “Hello” or number “1”) can then be initialized using:

object = ClassName.new(arg)

Specific attributes from the class can be read without having to write a specific function to do so using the following readers in the class definition:

attr\_reader :attribute\_name allows attribute to be read by calling ‘object. attribute\_name’

attr\_writer :attribute\_name allows attribute to be written by calling ‘object. attribute\_name’

attr\_accessor :attribute\_name allows read and write of attribute by calling ‘object. attribute\_name’

When defining a class of objects methods can either be public or private. Public methods can be accessed by objects in the class, however private methods can only be accessed in the definition of the class. To define public or private, simply add a line with ‘public’ or ‘private’ before definitions.

Inheritance

Inheritance allows a new class to inherit methods and variables of an already created class. The class inherited from is called a super class and only one is allowed.

new class < defined class

class definitions

end

It is possible to overwrite an inherited method by explicitly defining a new one with the same name. Even if overwritten it is still possible to access inherited attributes using ‘super(var)’

Scope of Object

The scope of the object is where the object can be accessed. There are four different levels of scope:

* Global – object can be accessed anywhere in code (generally avoid for security) define by defining object outside of methods or defining object with $name
* Local – only accessible inside section of code defined in
* Class – only accessible in class defined in, defined by @@name in class
* Instance – only accessible in specific instance defined in, defined by @name in instance

Modules

It is possible to pull pre-existing modules or create your own. To pull a module use:

require “Module” in general coding, then module objects and methods can be used

include module in class definitions, then module objects and methods can be used

To use a modules methods or objects without pulling the whole thing use:

Module::OBJECT

To make a module:

module ModuleName

OBJECT = value

end

Mixin

Mixin is mixing a modules methods and objects into a class definition, such as:

include *module* allows class to use the module in its definition at instance level

extend *module* allows user to call module methods and objects at class definition level