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

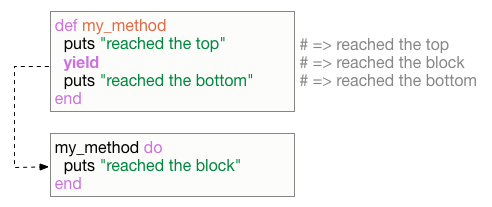
Arguments can be set to default values using the assignment operator, e.g. ‘account=100’ will assign a default value of 100 to the argument ‘account’, if no argument for ‘account’ is passed.

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.



Predicate metods are those which end in a question mark, e.g. ‘*integer*.odd?’, ‘*array*.include?(1)’ or ‘*string*.starts\_with?(‘a’)’and by convention these methods will either return true or false.

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’.

Ruby also has the ability for parallel assignment, which is where multipul assignment operations happen in parrallel. Parallel assignment allows both numbers which would normally be changed during sequential assignment to be changed and assigned simulatinously, and allows for methods to return multipul values if required. To perform parallel assignment, comma separate values, ‘i1, i2 = i2, i1+i2’ or ‘return a, b, c’ with ‘v1, v2 = method1’ to receive the mutlipul values.

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.

Multiline comments can be made using ‘=begin’ and ‘=end’ on the line before and after the comment section.

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. Symbols are not variables, so nothing can be assigned to them, but since they are immutable they should be used for names which will not be changed.

Arrays

An array in ruby is an object which stores a collection of any number of other objects, can be of multipul levels, and is defined between two square brackets. Arrays can be initialized using Array.new(size, default value), Array.new(size) {expression for default value}, or simply a = []. Objects can be called from arrays using array[index] or if multipul level array[index][index2]…etc, and indexing starts at 0. ends of arrays can be called using negative indexes, which are considered relative, so -1 pulls the last index of the array.

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”, …}

Unlike in arrays, where the key value is an indice (array[indice]), keys and values in hashes can be any object, however if they are symbols they will be immutable (not be able to be changed once created). To make a hash key a symbol, use :”<string>” => “<value>”

Values in hashes can be called via indexing: hash[key] will return the value associated with the key. For example calling: Hash[key2] will equal ‘value2’

One of the main uses of a hash is to refactor a conditional statement. This is done by setting the keys to the optional values of the conditional and the values as the appropriate responses, then calling the hash with the variable set as the key.

Since hashes can contain arrays and other objects, when the value object is called, any method which can be used on it normally can still be used.

When iterating over a hash, the element block contains both a key and a value pair ‘|key, value|’ this allows you to select either or both for the iteration.

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’. Break term can also be called in shorthand in conjunction with a boolean ‘break if <boolean-true>’ 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*) 🡨 range should be array of values, of which i is each sucsessive e.g. (1..4)

*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* 🡨 while true looks will only break when break is written

*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 triggered 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.

my\_lambda = lambda {|name| puts "Hello, #{name}!"}

Classes

Class is the type of object which can create object instances of itself. The behaviour of a class/instance is encapsulated within its notation with any implementation details being hidden, leaving only the methods and attributes require to use the object exposed on the public interface. An example of a class is Integer, String, or the main program called ‘Object’. Methods can be written to classes by defining them in the class notation, these methods can then be used by the class instances using ‘instance\_object.method\_name’. Classes are defined as:

class ClassName

def initialize(arg,..)

@atributes = arg

…

end

def attribute=(<input\_from\_instance>)

@instance\_variable = <input\_from\_instance>

end

def attribute(<input\_from\_instance>)

@instance\_variable = <input\_from\_instance>

end

def method

<method\_procedure>

return <self + @instance\_variable> -example

end

end

If when opening up a class of ClassName it doesn’t exisit, the Class class will automatically make a new class for it. Using the initalise method will allow instance atributes to be created from the arguments of the ‘Class.new(arg, arg ,etc)’ method.

Attributes for instances can be created using @atribute\_name, and defining them using either of the two methods above. Due to the different setter methods, one will expect the input\_from\_instance to come from ‘instance.attribute\_name = <value>’ and the other ‘instance.attribute(<input>)’.

Methods for instance can be defined using the method above. When defining methods inside a class, the class will not know what the instance is initially. To retrieve the instance and apply procedures to it, use the ‘self’ notation.

Method can also be written for the class itself, by defining the method name with ‘self.<method>’. This will allow design for access to particular class variables and see details created on the whole class, such as average values etc.

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

object = ClassName.new(arg)

Each class should have one purpose or job, sometimes referred to as its responsibility. Cohesion is making sure that a class is designed with a single, well-focused purpose. A class has high cohesion when everything inside of it relates to that purpose, without anything extraneous. Perfection is achieved when there is nothing left to take away.

Ruby also has a class called Class, which using the ‘string = Class.new’ syntax will create a new class of the string name specified.

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.

Classes are good for defining a program ‘Domain’, which represents a the program situation as best a possible.

Class Constants & Variables

Classes have various different constants/variables which can be defined and accessed in different areas and different ways:

Global variables - These are defined anywhere in the script with the ‘$<variable>’ notation. These variables will be accessible by any code in the script, so are generally not used since they can pose security issues.

Local variables - These are defined locally inside classes or methods, they will only be available in their immediate scope to the class definition or method. Generally avoid in classes since the scope is so limited.

Instance variables - These variables are defined with the ‘@<variable>’ notation. Instance variables can be created/modified anywhere in the class definition, however generally they are given a value/default value in the initializer, since if only defined in a method they may not have a value and return a NilClass error. Instance variables will be unique to each instance created of the class and therefore only have the scope of the specific instance they belong to. They can be accessed on the public interface of the class instance using variable encapsultation via manual definition of a method or use of ‘attr\_reader’.

Class variables - These variables are defined with the ‘@@<variable>’ notation. Class variables can also be created/modified anywhere in the class defintion. Class variables can be accessed by any instance of the class, allowing for automatic variable sharing across the class hierarchy. However, class variables are set when the code for the class is run e.g. initally, therefore if there are mutlipul assignments to one class variable in different super/sub classes, the last one in the defintions will be be the value of the class variable. Class variables can be accessed on the public interface of the class using variable encapsultation via manual definition of a method or use of ‘attr\_reader’.

Class constants - Class constants are defined in the class using the full capitals notation, e.g ‘CLASSCONSTANT’. These constants can be accessed by calling throughout both the class definition and instances of the class, they can also be acceed on the public interface of the class by callling ‘ClassName::CONSTANT’.

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

For example variables defined inside methods/classes cannot be read outside of them, this is due to the variables ‘Scope’ which is defined as local. It is also not possible to read variables defined outside of a method/class inside a method/class, this is because methods/classes/modules are scope gates.

To jump scope gates and allow local variables into the definitions of methods, classes, and modules use blocks during definition:

* Classes Class.new { block }
* Modules Module.new { block }
* Methods new\_method(<method-name>) { block }

Blocks act as a kind of one way scope gate where variables defined in them are part of a new scope, but they can still access local vairables in the scope they are created in. Varibales called in blocks can also be modified, however this can be stopped by defining the variables to be unmodified in the block element section with a semicolon before: { |n; variable 1, variable 2| <procedure> } If a block is to repeat, such as in a each method, the variables inside will reset as a new scope is defined each loop.

Since in Ruby methods and variables can be called without an explicit reciever and paraentheses, if there are two named the same in the same scope of code, the variable will shadow the method. To then call the method either use self.method or method().

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

Object Oriented Design

Refactoring

Refactoring is taking already written code and simplifying it to make it easier to read, and execute smoother and quicker. A common method of refactoring is to put sections of code into methods, this is called abstraction. However, it is important that a method only does one thing and is named correctly, this is called ‘The Single Responsibility Principle’. Giving each method only one thing to do also allows for quick debugging and modification during development.

Debugging

It is common to have bugs in a program, so it is important to follow a consistent debugging procedure to fix them. In its simplest form debugging should be as follows:

1. Tighten the loop - Find line of code the bug is coming from
2. Get visibility - Use p to inspect attributes and code being used on line which bug occurs
3. Fix bug - Fix anything found
4. Repeat - Repeat until no more bugs found

Following the debugging tree is very important when trying to find the line of code the bug has come from. The tree will show a stack trace of where the program errored and which line of code each time. Take time to follow each error line upwards understanding the what the code is doing each time, until you reach the final line where the error occurred. On this line something will be wrong so think through logically and try to see what mistakes there are.

Regex

Many methods in Ruby can take a regular expression (regex) as an argument in order to match text. In Ruby regexes are expressed between two forward slashes ‘/*<regex>*/’ can be used in many ways, some examples are:

*<string>* =~ /*<regex>*/ returns first index in string which matches regex, nil otherwise.

*<string>*.match(/*<regex>*/) returns matched letters if matched, nil otherwise

=~ regexes can be used with either a if conditional with the regex != nil to confirm if a string matches a regex, or !!(*<string>* =~ *<regex>*) to convert the match output to a boolean. Regexes can be either simply a string to match such as matching ‘like’ in ‘I like cats’ or a set or pattern of charater classes, contained in ‘[<charater-class>]’ to match variants of more specific strings. The first set of charater classes are ranges, which match any charaters in the given range:

* [a-z] matches charaters between a-z, no caps
* [B-H] matches charaters between B and H, only caps
* [0-9] matches charaters between 0 and 9
* [^3-6] exludes matching charaters between 3-6

Charater classes can be combined by lining them up in the square brackets, such as [a-zA-Z]/ will match charaters between a and z both lowercase and uppercase.

There are useful shorthands for many types of charater classes in Ruby, these are escaped (backslashed) of the regex in order to give them to regex algorthim as parameters:

* \w [0-9a-zA-Z\_] numbers letters and underscore
* \d [0-9] digits
* \s whitespace tabs/spaces/newlines
* \W negative [0-9a-zA-Z\_] matches anything not in \w
* \D any non digit
* \S any non space
* \b any word boundary

Another charater is the ‘.’, this will match everything but new lines. To match the literal ‘.’ it has to be escaped i.e. ‘\.’.

Variables can be inserted into a regex using ‘#{<variable>}’, if variables contain metacharaters which don’t want to be interpreted as so, use ‘#{Regexp.escape(<variable>)}. Multipul variables in either string or array form can be added using ‘#{Regexp.union(<variable>)}’, which will make a union regex object of all items to then match any of them, similar to [0-9] matching all digits, items will be escaped.

Regexes can be given options using the ‘(a|b)’ syntax, allowing either a or b to be captured.

Modifiers are used on charater classes to match more than one charater, they are great for matching patterns since set numbers of charaters can be selected for matching:

* + one or more e.g. /\d+/ matches to one or more digits
* \* zero or more
* ? zero or one
* {4} exactly three charaters in a row e.g /a{4}/
* {3,5} between 3 and 5 charaters
* {1,} one or more

Modifiers can be made exact using the following before or after a charater or charater class:

* ^ start of line
* $ end of line
* \A start of string
* \Z end of string

For example using /^*<regex>*{4}$/ which will only match if the charater set in the string is exactly 4 long and at the start and end of a line (i.e. the whole line).

Captures allow for parts of a string to be matched and stored more specifically than simply matching and storing the whole result. To capture part of a regex put the expression in parentheses ‘/(*<captured-regex>*)/’, the capture will then be expressed in the match data object array. If more than one capture group is specified e.g. /(*<group>*)(*<group>*)/ the regex will search for the match of the whole expression and then store each capture group seperately, in acceding numbers of the matched-data-group array. The matched data object can be assigned to a variable and manipulated in many ways.

To make the capture clearer, it can be assigned a name/key, to do this inside the capture parentheses use ‘?<*<capture-name>*>’, this will then store whatever is captured by the symbol <capture-name> in the matched data object. To then call back the data captured use ‘*<matched-data-object>*[:*<capture-name>*]’, similar to how hashes are called.

Example of calling a matched group:

m = “John 31”.match(/\w+ (\d+)/

m[1] 🡺 31

Example of calling a named match:

m = “John 31”.match(/\w+ (?<age>\d+)/

m[:age] 🡺 31

Groups can also be set to non-capture, this allows a group to be used during the match process, however it will then be excluded from the matched-data-object on return. To set a non-capture group use (?:*<regex>*).

Look ahead and look behind allows the regex to match a pattern on a string depending on what is or isn’t before or after it. Firstly positive look ahead or behind will match a pattern if the trailing or leading pattern is also matched:

* Positive lookahead / (*<pattern>*)(?=*<pattern-after>*)/
* Positive lookbehind /(?<=*<pattern-before>*) (*<pattern>)*/

Then negative look ahead or behind does the opposite, matching a pattern if the trailing or leading pattern is not matched:

* Negative lookahead /(*<pattern>*)(?!*<pattern-after>*)/
* Negative lookbehind /(*<pattern>)*(?<!*<pattern-before>*)/

Ruby regular expression, as everything in Ruby, is part of the Regexp object class. This means they can be created similar to instances of other classes:

regex = Regexp.new(“a”) 🡺 /a/

regex = %r{ [a-zA-Z] }

Regexp instances can also be made by simply assinging a variables to the regex, like you would a variable to a string. This means complex regular expression can be made and used in code multiple times without cluttering.

There are several parameters which can be passed onto the regex engine after the expression has been defined:

* i make the regex case insensitive
* m dot will match a newline (\n)
* x ignore whitespace

For example to match one charater of letters but being case insensitive:

regex = /[a-z]/i

Complex regular expression can be split onto multiple lines to allow for easier reading, for example:

LOG\_FORMAT = %r{

(\d{2}:\d{2}) # Time

\s(\w+) # Event type

\s(.\*) # Message

}x

A common regular expression is validating an email address, to return true if matched:

def matchemail(email)

email.match?(/\A[\w.+-]+@[\w-]+\.\w+\z/)

end

Rakefiles

Rakefiles are similar to unix makefies and windows batch files however run in Ruby. A rakefile will run a set of default tasks, or very specific ones depending on how they are called. Here is an example rakefile:

*task default: %w[test]*

*task :test do*

*ruby "test/unittest.rb"*

*end*

This rake file will run the task ‘test’ by default, which in turn runs the file ./test/unittest.rb.

A special feature of rakefiles is dependencies. Dependencies will only run given the previous task has completed, these can be added using the rocket symbol:

*task default: %w[test]*

*task :test do*

*ruby "test/unittest.rb"*

*end*

*task :raking => [:test] do*

*puts "Should put this after :test"*

*end*

Rakefiles are generally called via rake and set up in a Ruby project folder like so:

-Project-Folder

-Rakefile

-lib

sample\_ruby\_file.rb

-test

sample\_ruby\_test.rb

another\_test.rb

Opening and Saving to a File

Using the File class its possible to open/create a file to save data into in a script. To do this the file must first be opened:

<filename> = File.open(“<computerfilename.ext>”, “w”)

File.open(filename, argument) {|file| expression}

Then written as many times as required to using either print, puts, or p:

<filename>.puts <string>

Then closed to save:

<filename>.close

file does not need closing if opened using block method

This will write the the text data to the file on the computer as the “w” command was specified, various others can be used to do different things:

* "r" Read-only, starts at beginning of file (default mode).
* "r+" Read-write, starts at beginning of file.
* "w" Write-only, truncates existing file to zero length or creates a new file for writing.
* "w+" Read-write, truncates existing file to zero length or creates a new file for reading and writing.
* "a" Write-only, each write call appends data at end of file. Creates a new file for writing if file does not exist.
* "a+" Read-write, each write call appends data at end of file. Creates a new file for reading and writing if file does not exist.

When reading from csv files using the ‘r’ arugment, use readline with parallel assignment for each value seperated by commas ‘file.readlines.each{|line| name, age, cohort = line.chomp.split(',')}’.

Loading Arguments from the Commandline

If the program is loaded from the commandline with arguments, they will be automattical inserted into the special ARGV array in the program. The arguments will be added into the ARGV array in the order which they are writted in after the program name.

When supplying arguments to a ruby program, if there is a gets statement in script it will automatically attempt to get the data from the list of files supplied, unless there are no file supplied. Therefore, to allow the gets command to read specifically from the input stream use ‘STDIN.get’ instead.

Continuous Integration

Continious integration is a critical part of the professional developers stack - it's an automated process that will run a suite of tests any time code is submitted to a version control repo (such as GitHub) or when someone submits suggested code changes to that repo through a pull request.

CI is generally used alongside an agile software development workflow. An organization will compile list of tasks that comprise a product roadmap. These tasks are then distributed amongst software engineering team members for delivery. Using CI enables these software development tasks to be developed independently and in parallel amongst the assigned developers. Once one of theses tasks is complete, a developer will introduce that new work to the CI system to be integrated with the rest of the project.

Recursion

Recursion is a method which calls itself until a given condition is met.

Test Driven Development

Test Driven Development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed.

There are three rules of TDD:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

By following the rules, tests are written before any functional code and development of large projects is done in small functional pieces. The tests provide documentation on the public interface of the object, meaning if at any point it become nessecarry to modify the implemenaton of object behaviour, the tests will prove that the object doesn’t get broken in development.

So from the rules there are three phases of TDD:

1. Red - Write test which fails
2. Green - Write only enough code to pass test
3. Refactor - Refactor written code to be in its simplest form

TDD & BDD

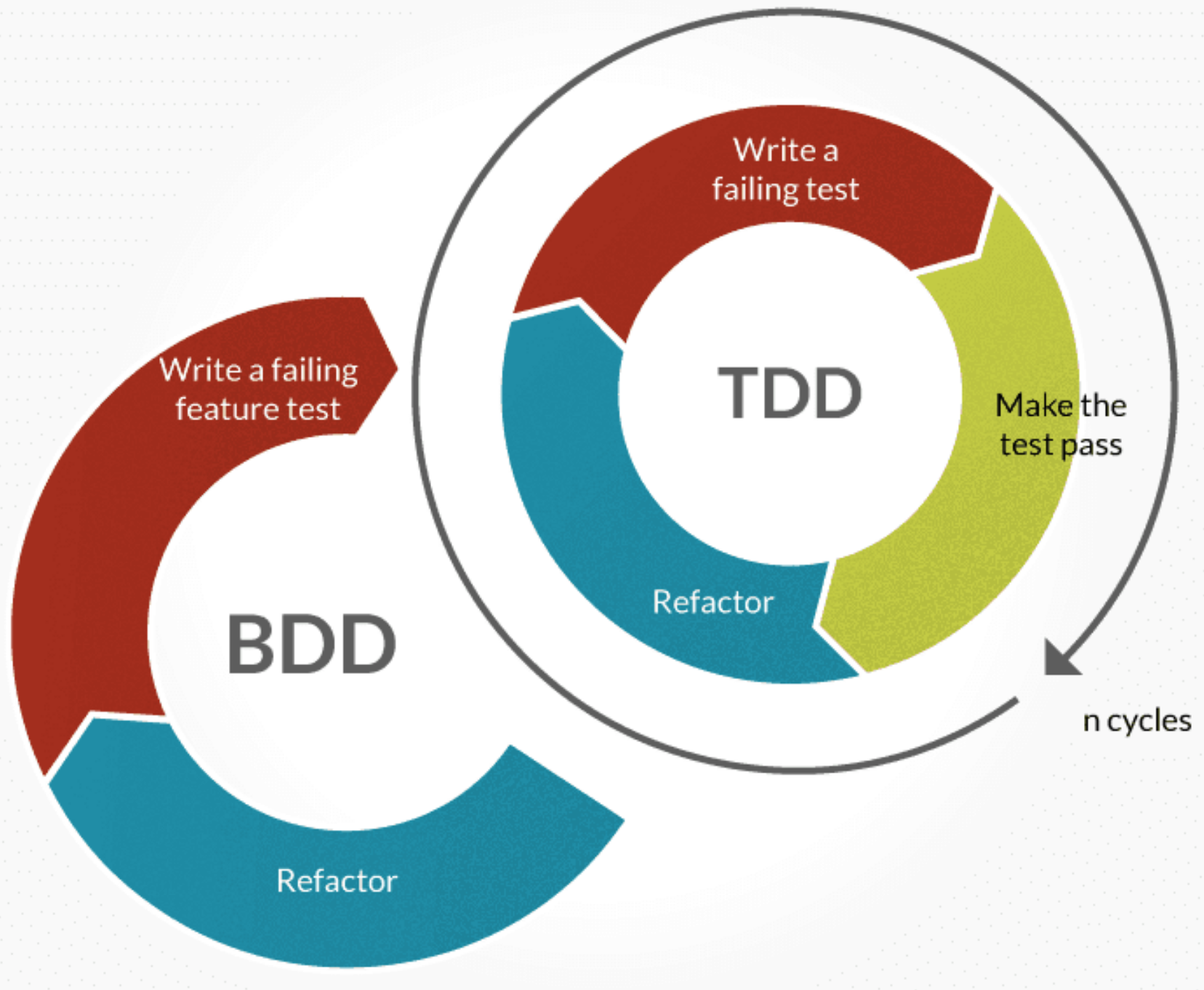
TDD - Test Driven Development

* Written to check the implementation of functionaility
* Inside out
* What will each function do
* As code evolves tests can fail

BDD - Behaviour Driven Development

* Written to check the behaviour of a system from a end users perspective
* Outside in
* What will happened to a system under a certain condition

TDD and BDD can be used together to create systems, by first creating a feature test expect certain behaviour from the system, then creating unit test for each function to create that desired behaviour of the feature test. Once the feature test passed, move onto the next and the cycle continues.



Rspec

RSpec is a common method of TDD. RSpec calls a script and expects a specific answer to be outputted, the specifics of the answer can be as detailed as wanted and depends on how the RSpec script has been written.

Test for what you are expecting to happen, or what you don’t want to happen, by performing the task and seeing what happends. Test for the behaviour on public interface, ie messages between objects and methods returning. Finding the behaviour to test means asking two questions:

* Who is the user - whoever is calling using the method (other class or user)
* What is the code doing for the user - What should the interface return

Therefore, write tests as if you are the user and do not test the mechanisms or state of the system directly, i.e. no calling instance variables in tests. This allows for quick TDD of what the class does without having to think about implementation behind it and get too specific with the test (which might cause it to be a pain to change in the future).

RSpec scripts are put in the project ‘spec’ folder and require ‘\_spec.rb’ at the end to identify as a spec file, and the project scripts in the ‘lib’ folder and will have to be explicitly required in the spec file using ‘require ‘<script>’’. RSpec is then called by simply calling ‘rspec’ on the command line. To initalise RSpec, use ‘rspec –init’ which will create:

* .rspec 🡨 Contains default arguments which run with rspec, --colour --require spec\_helper
* spec/rspec\_helper.rb 🡨 add details

To run specific rspec scripts, add the path to the script as an argument to rspec.

A typical RSpec template script looks like:

*require ‘script’*

*describe "Solution" do* 🡨 Name for RSpec script

*it "should test for something" do* 🡨 Describe what script does

*expect(script(<argument>)).to eq <output>* 🡨 Add test cases

*end*

*end*

With some values added:

*require ‘../lib/greeter.rb*

*describe 'Greeter' do*

*it 'greets Rico' do*

*expect(greet('Rico')).to eq 'Hello, Rico, how are you today?'*

*end*

*end*

Test cases follow:

* Given 🡨 Arguments
* When 🡨 Arguments passed into script
* Then 🡨 Expected output

eq is an example of an RSpec matcher, which matches the expected result with the one outputed by the script. There are many different matchers in RSpec.

In TDD the specification scripts are written before the script, so that when run they fail. A script is then written to match the specification written and pass the RSpec test. Once the script has passed, the script is then refactored so code is more appealing and better written.

Disable rspec tests by adding an ‘x’ before the ‘it’ statement e.g. ‘xit’

**RSpec doubles**

Tests should be written as ‘unit tests’ which mean each test tests and the test is isolated from all other code. Therefore, tests should only test one behaviour and should not rely on other external code or objects, so it might be nesseceary to mock other objects using RSpec doubles.

Doubles are objects which stand in for external object in a test. This allows for consistencey on the public interface of object being tested and the RSpec test will not rely on the stability of an external object to test the functionality of the tested object, so if the other objects break, the test will still pass for the unchanged object being tested.

**RSpec Matchers**

expect{ subject.method }.to change{ subject.attribute}.by val

expect { i = 2 }.not\_to change { i }.from( 1 )

expect { i = 2 }.to change { i }.from( 1 ).to( 2 )

expect { i += 1 }.to change { i }.by ( 1 )

expect { i += 1 }.to change { i }.by\_at\_least ( 1 )

expect { i += 1 }.to change { i }.by\_at\_most ( 1 )

expect(subject.get\_ids).to be\_an\_instance\_of(Array)

expect(subject.get\_ids).to match\_array(expected\_array) 🡨 checks contents of array are some no matter of order

**Raising Exceptions**

An exception is a special kind of object which represents some kind of expectional condition (error), indicating something has gone wrong. Raising an exception if something unexcepted is done is called a Guard Clause. Generally the Ruby program will crash if an exception is raised, however error handlers ( {block} ) can be used to expect for an exception and perform an action if an exception occurs, this is called a Rescue Clause.

There are many different types of exception object which can be raised, but the default it RuntimeError, which can be raised and given a message using ‘raise(“<message>”). Different error classes can be raised if they are specified as an arugment, e.g ‘raise(ArgumentError, “Argument is not numeric!”).

To handle exceptions, enclose the block of code which will raise an error in handlers (begin -> end), then including a ‘rescue’ clause, will catch the error and continue the program from the resuce clause instead of exiting the program. Using an ‘ensure’ clause will mean any code after the ensure block will always be run, no matter if an error is raised or not.

*begin*

*raise “This is an error!”*

*rescue*

*puts “Program saved!”*

*end*

Rescue clauses can be given an argument to look out for a specific error, e.g. ‘rescue RuntimeError’ or ‘rescue SynatxError’. This will only rescue the error if it is a specific type.

Using an ‘ensure’ clause will mean any code after the ensure block will always be run, no matter if an error is raised or not.

*begin*

*raise “This is an error!”*

*ensure*

*puts “I always run!”*

*end*

Errors messages can be passed through to the rescued code block using the rocket symbol, e.g. ‘rescue RuntimeError => e’ with e as the error object. This allows for printing out or logging or error messages. However, beaware that improper error messsges can provide critial system information for attackers, since they can reveal backtraces, database dumps and error codes. It is recommended that production applications should not use puts e.backtrack.inspect unless being stored directly in a log not viewable by the end user.

Guard Clauses

Guard clauses are a one line way of protecting the running of some code unless a condition is met. Generally they are put at the start of a method to ensure that arguments/parameters meet a criteria, so that the following code will run without giving exceptions or other problems. Guard clauses can be used to return values or error codes, whatever will be most useful to the user invoking the method.

*def my\_method(variable)*

*return nil unless variable == 'great'*

*# do something great*

*end*

Class Design - Setter Methods

If checking input from a setter method meets some restrictions and the object instance variables can be set at both initialization and after, delegate the setter methods in initialization, calling ‘self.getter\_method’ in initialization so even then the restrictions are enforced.

String Formatting

It is possible to format strings in ruby using the sprintf format. sprintf uses format specifiers in the string, escaped by %, then fills in the string using an appended array, e.g.

*time = 5*

*message = "Processing of the data has finished in %d seconds" % [time]*

*output => "Processing of the data has finished in 5 seconds"*

There are many different format specifiers which can be used, all can be here:[*https://alvinalexander.com/programming/printf-format-cheat-sheet*](https://alvinalexander.com/programming/printf-format-cheat-sheet)

* %c Charater
* %d decimal number
* %f float
* %i integer
* %s string

The integer format can be controlled by adding numbers before the format specifier:

* %3d gives minimum integer width of 3, right justified - ‘ f’
* %-3s gives minimum integer width of 3, left justified - ‘fe ‘
* %03d fills blank space up to 3 with zeros, right justified - ‘021’
* %+5d gives minimum 5 with a plus sign, right justified - ‘ +3’
* %.2d 2 decimal places - ’10.20’
* %6.2d 6 wide but 2 decimal places - ‘ 1.21‘

A good reason to use this type of string formatting in Ruby is to add leading or trailing zeros, or use a map function to fill a string, e.g

*array = [1,2,3]*

*"Value 1: %03d, Value 2: %03d, Value 3: %03d" % array.map{|n| n\*\*n }*

*output => "Value 1: 001, Value 2: 004, Value 3: 027"*