

IRT Tutorial v.1.1.0

Copyright © 2010-2011 Domizio Demichelis

Project URL: <https://github.com/ddnexus/irt>

*This tutorial assumes you have read the section "**What is IRT?**" in the project homepage.*

Sometimes, there are things that are simple to do, but not so simple to explain... at least for me :-). This is the case of this tutorial: what takes a lot of text to be explained, will take just a few seconds to be executed when you learn the trick, so don't worry: IRT is really fast and easy to use despite the 23 pages of this tutorial.

You will learn how to work with IRT with a deadly simple class (just to avoid to get distracted by the code). Some steps could seem trivial or useless in this context, but they will be useful when you will have to deal with real word applications, specially Rails applications.

The first principle to learn with IRT is that you should start to use it as soon as you write your very first lines of code for any project. You can use IRT as you use irb or the Rails console... to try your code on the way. The only difference is that you have a lot more features and tools at your disposal, and if you use them properly, when you will have done with your code, you will have done with your tests without any added effort.

So let's start from the beginning...

Your first project with IRT

Create a dir named 'irt_tutorial' and create the 'irt_tutorial/person.rb' file with this content:

```
class Person
  attr_accessor :name, :surname
  def initialize(name, surname)
    @name = name
    @surname = surname
  end
end
```

Now cd to the 'irt_tutorial' dir and launch irt with the path of a new file:

```
$ cd irt_tutorial
$ irt test/person.irt
```

Answer 'y' (or <enter>) and IRT will create a new directory 'test' with the 'person.irt' file in it, then it will run the file.

Suggestion

Creating new files and dirs with IRT and running multiple files in nested dirs is very easy by design. Put just a few (consistent) tests in each file and create new files often. Many small nested files are more manageable than a few big files...

Note about new files

The new created files are implicitly run with the -i (--interactive-eol) flag by default. That flag instruct irt to open an interactive session at the end of file, so you will have the possibility to add your statements. As long as you rerun the file (with the 'rr' command) IRT will remember the flag, anyway, if the session ends and you have to re-launch it from the command line, you must pass the -e flag explicitly or no interactive session will be opened automatically.

You could also add an explicit 'irt' statement (without any argument) anywhere in your .irt file, in order to open an interactive session at the line of the statement. That is a directive that makes IRT switch from file mode to interactive mode. You can use it whenever you want to play with the code in the file at any given line.

Adding your first test

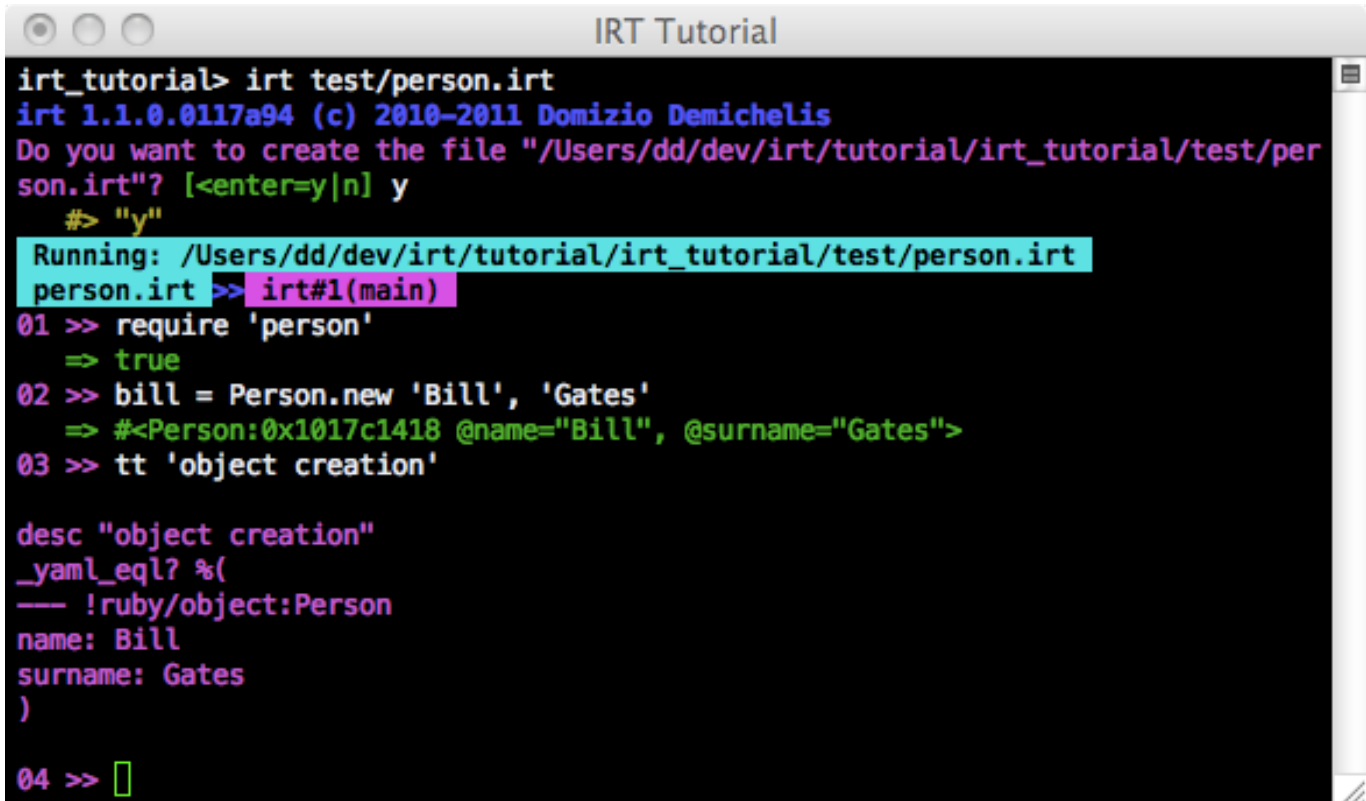
Use your console as a regular irb console and create an object with your Person class:

```
>> require 'person.rb'
>> bill = Person.new 'Bill', 'Gates'
```

The object 'bill' looks OK, so let's add a test ('add_test' or 'tt') checking for that value:

```
>> tt 'object creation'
```

Now you should see something like this:



```

IRT Tutorial
irt_tutorial> irt test/person.irt
irt 1.1.0.0117a94 (c) 2010-2011 Domizio Demichelis
Do you want to create the file "/Users/dd/dev/irt/tutorial/irt_tutorial/test/per
son.irt"? [<enter=y|n] y
  #> "y"
Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
person.irt >> irt#1(main)
01 >> require 'person'
    => true
02 >> bill = Person.new 'Bill', 'Gates'
    => #<Person:0x1017c1418 @name="Bill", @surname="Gates">
03 >> tt 'object creation'

desc "object creation"
_yaml_eql? %(
  — !ruby/object:Person
    name: Bill
    surname: Gates
  )

04 >> 

```

Warning: If you don't see any color in your console, you are probably on a Windoze system. Please read "ANSI Colors" in the README file in order to fix it.

As you see the 'tt' (or 'add_test') generated a few lines containing the description of the test, and the test itself, which may look a little weird: don't worry about that,

it is simply a directive with the serialization of the last value (the object 'bill' in our case) that tells IRT to test that value. You will copy and paste it into the 'person.irt' file... in just one step, but first, let's take a look around.

Type 'log' or just 'l' in the console:

The screenshot shows a terminal window titled "IRT Tutorial". At the top, it says "04 >> l". Below this, a "Virtual Log" is displayed with a pink header "irt#1(main)". The log contains the following code snippet:

```
1 require 'person'
2 bill = Person.new 'Bill', 'Gates'
3 desc "object creation"
3 _yaml_eql? %(
3   --- !ruby/object:Person
3   name: Bill
3   surname: Gates
3 )
```

Below the log, a prompt "person.irt >>" is followed by "irt#1(main)". At the bottom, it says "05 >> []".

Notice:

- The Virtual Log Tail shows you the latest lines of what is relevant to your testing and where it comes from (file or session), complete with its reference line no.
- The 'tt' command has been invoked at line 3, and has produced a few lines, which all refers to line #3).
- The last hunk - titled 'irt#1(main)' - is what you should add to the 'person.irt' file if you want to reproduce (and test) automatically what you just did in the interactive session.

Don't worry about all that line numbers in the way. You don't have to copy that from the log: the log is just a graphical representation of what has happened at any given time in your execution, that gives you an easy feedback at the touch of a 'l'. Depending on your OS, you have a few better ways to copy the last lines to the file.

Copy the session lines

The easiest way to copy all the session lines is using a combined copy-open command, that will fill the clipboard with the lines and will open an editor, possibly at the current evaled line. You can try the commands: 'cnano' (or 'cnn') or 'cvi' to copy the lines and open the file in place with nano or vi. You can eventually use 'cedit' (or 'ced') to copy the lines and open the file with your default GUI editor.

Just try them, and if they don't work yet (you will read the documentation to fix that) you can use other options to do the same.

You can use 'copy_lines' (or 'cl') and then manually switch to your preferred editor where you will paste the lines, or if that doesn't work yet, you can use 'print_lines' (or 'pl'), select and copy the lines from the console, then use 'nano', 'vi', 'edit' or manually switch to your preferred editor and paste the lines.

But let's suppose you are not lucky today, and that you have to print the lines, copy, switch and paste manually. Start with typing 'print_lines' (or 'pl'):

```
>> pl <select><copy> <switch><paste>
```



```

05 >> pl

require 'person'
bill = Person.new 'Bill', 'Gates'
desc "object creation"
_yaml_eql? %(
  — !ruby/object:Person
  name: Bill
  surname: Gates
)

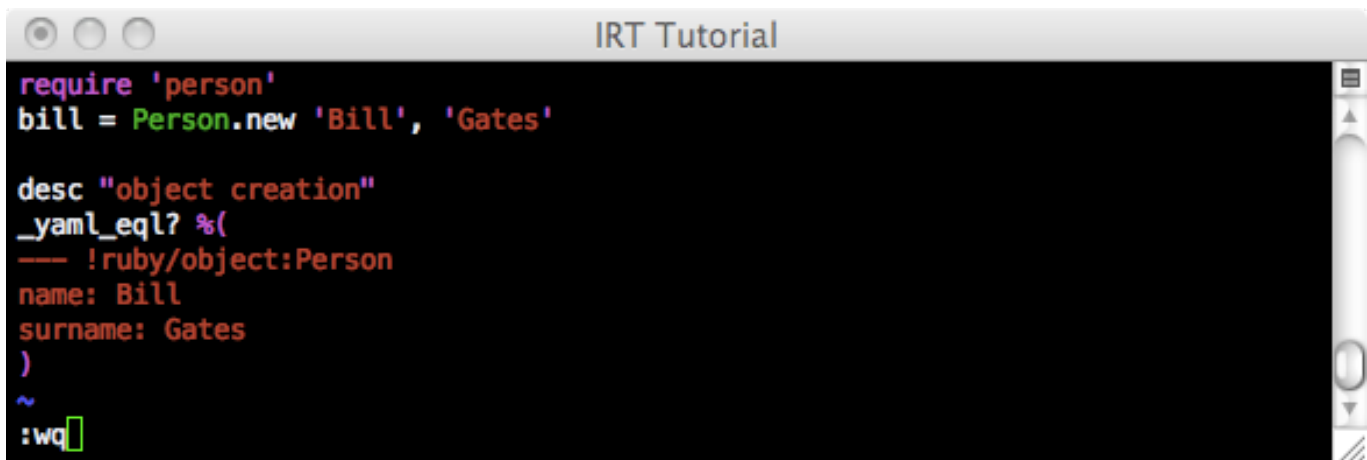
06 >> 

```

Notice:

The test lines outputted by a 'tt' or by a 'pl' command start and end with an empty line for easy select (just click on the first empty line and drag till the last empty line)

Now, your edited 'person.irt' file should look like this (below you can see the 'cvi' screen after the paste: we have just added a couple empty lines for better readability):

A screenshot of a terminal window with a dark background and light-colored text. The window title is "IRT Tutorial". The code displayed is as follows:

```
require 'person'
bill = Person.new 'Bill', 'Gates'

desc "object creation"
_yaml_eq!?(
  !ruby/object:Person
  name: Bill
  surname: Gates
)
~
:wq
```

The cursor is at the end of the last line, after the colon and 'wq'. The text is color-coded: 'require' is purple, 'Person' is green, 'Bill' and 'Gates' are orange, 'desc' is orange, and the rest is light gray.

Notice:

- The ':wq' in the last line is just the vi command for write and quit).

Save it, then close the inline editor to back to the session type 'rerun' or just 'rr'

Running your first test

The first test passed and printed a green line telling just that everything is ok. Now look around with 'log' (or 'l') again:

```

07 >> rr
The TEST is OK!

Rerunning: `irt_irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt `

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
1. OK! object creation
person.irt >> irt#1(main)
01 >> l

Virtual Log
person.irt
1 require 'person'
2 bill = Person.new 'Bill', 'Gates'
3
4 desc "object creation"
5 _yaml_eql? %(
6   — !ruby/object:Person
7   name: Bill
8   surname: Gates
9 )

person.irt >> irt#1(main)
02 >> 
  
```

Notice:

- The Virtual Log Tail now shows the executed lines from the file 'person_obj.rb', and no lines from the session (indeed we didn't write anything interactively this time).
- Now we are in an interactive session opened by the 'irt' directive at the end of the file (line #9)

Let's add a few more tests for a couple of accessors:

```

02 >> bill.name
    => "Bill"
03 >> tt 'name'

desc "name"
_eq1?( "Bill" )

04 >>
05 >> bill.surname
    => "Gates"
06 >> tt 'surname'

desc "surname"
_eq1?( "Gates" )

07 >> l

Virtual Log (tail)
person.irt
7  name: Bill
8  surname: Gates
9  )
irt#1(main)
2  bill.name
3  desc "name"
3  _eq1?( "Bill" )
4
5  bill.surname
6  desc "surname"
6  _eq1?( "Gates" )

person.irt >> irt#1(main)
08 >> 

```

Notice:

- We usually add an empty line after a test, so each test will look better separated from each other.
- If you want to see all the lines in the log and not just the tail that is shown now, you should use 'log <n>' to print <n> lines of the tail, or 'full_log' (or 'll') to print them all.

Now you can see the lines of the session. Just copy them with the method that has worked for you (as you did for the 'object creation' test), paste them into the file (leaving 'irt' at the end), save and 'rr'.

```

08 >> cvi

bill.name
desc "name"
_eq1?( "Bill" )

bill.surname
desc "surname"
_eq1?( "Gates" )

09 >> rr
The TEST is OK!

Rerunning: `irt_irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt `

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
1. OK! object creation
2. OK! name
3. OK! surname
person.irt >> irt#1(main)
01 >> 

```

We copied the lines with the 'cvi' command, pasted into vi, saved, and rerun the file. Cool! Now we have 3 test passed, and we can start to think how to break them :-).

So let's back coding. We have just realized that we need to add also a country field, so let's add it to our class:

```

class Person
  attr_accessor :name, :surname, :country
  def initialize(name, surname, country)
    @name = name
    @surname = surname
    @country = country
  end
end

```

You can use our preferred editor to edit the file, we do it in place by typing "vi 'person.rb'" in the console, edit the file, save it and close it (with :wq).

However you decided to change the file, that change is going to break our tests, so let's see what will happen and how easy will be fixing it. Type 'rr' in the console.

Dealing with exceptions

```

01 >> vi 'person.rb'
02 >> rr
  ALL 3 TESTs are OK!

Rerunning: `irt_irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt `

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
ArgumentError: wrong number of arguments (2 for 3)
    from /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2:in `initialize' [0]
    from /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2:in `new'
 [1]
    from /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2 [2]
person.irt >> irt#1(main)
01 >> l

Virtual Log
person.irt
1 require 'person'
2 bill = Person.new 'Bill', 'Gates'

person.irt >> irt#1(main)
02 >> 
  
```

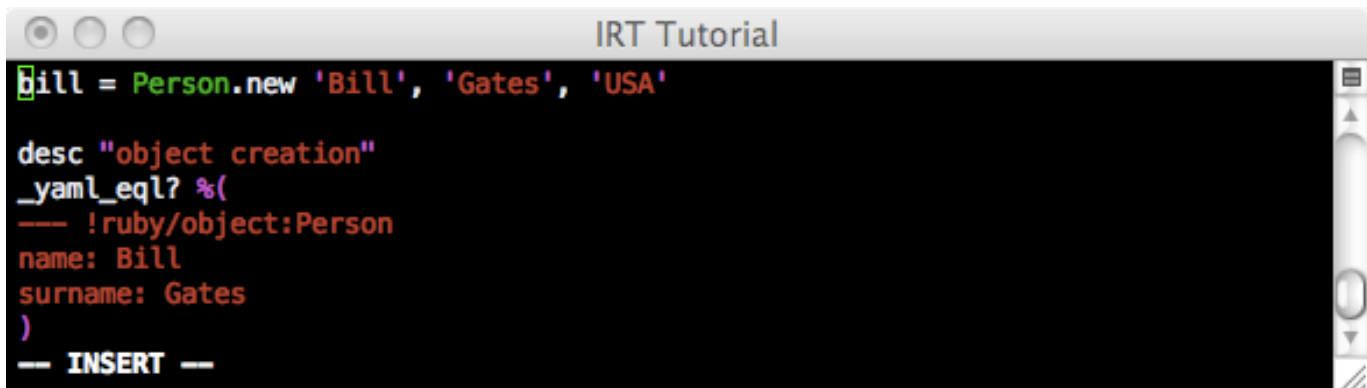
We got an error because the Person constructor now wants one more argument, and IRT just opened an interactive session at the line of the error: you can check it by looking at the log: it stops at line #2 of the file.

In a real world app you will probably have to check some value in the console in order to figure out what happened, but here we already know what went wrong, so we can skip that inspecting part.

You can open the file for in place editing at the line of the error by just typing 'nano' or 'vi' with no arguments, or if that is not installed, you can use 'edit' (or 'ed') or

simply manually switch to your editor.

We will use 'vi' again, which will open the 'test/person.irt' file in insert mode at the line #2.



```

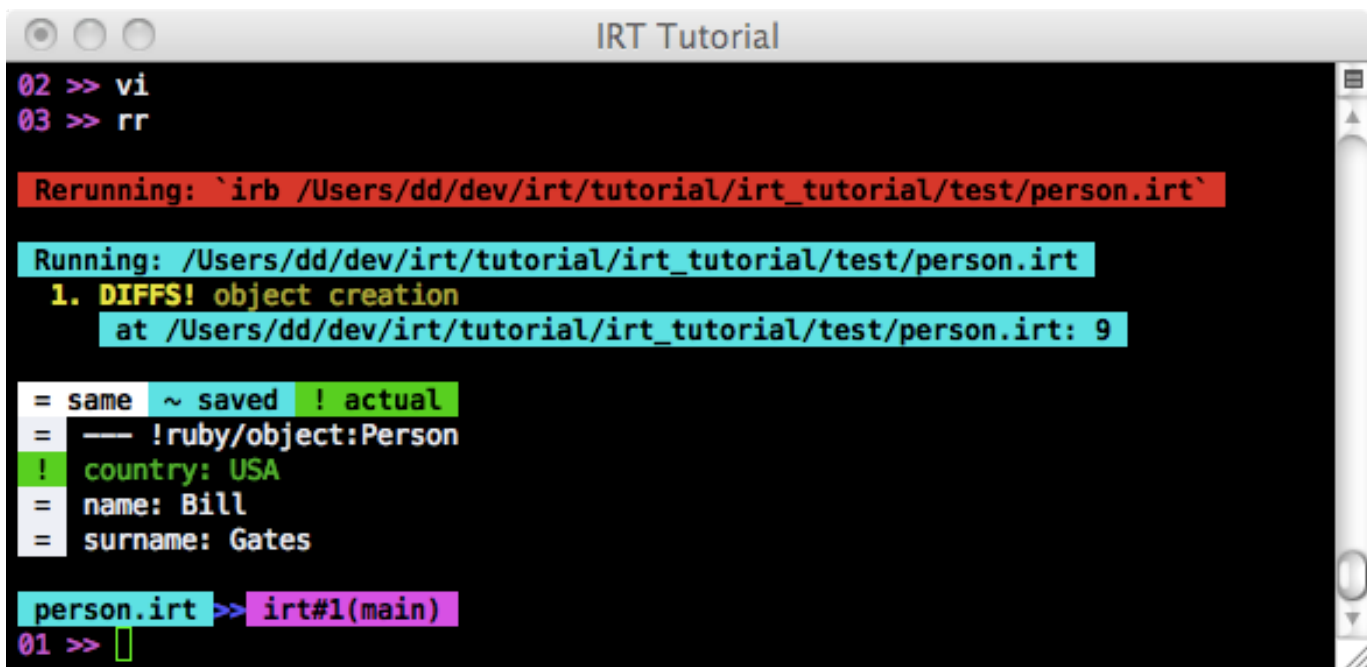
IRT Tutorial
bill = Person.new 'Bill', 'Gates', 'USA'

desc "object creation"
_yaml_eq1? %(
  --- !ruby/object:Person
  name: Bill
  surname: Gates
)
-- INSERT --

```

However you opened the file, just add 'USA' as the last parameter, save and 'rr'.

Dealing with failures



```

IRT Tutorial
02 >> vi
03 >> rr

Rerunning: `irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt`

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
1. DIFFS! object creation
   at /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt: 9

= same ~ saved ! actual
= --- !ruby/object:Person
! country: USA
= name: Bill
= surname: Gates

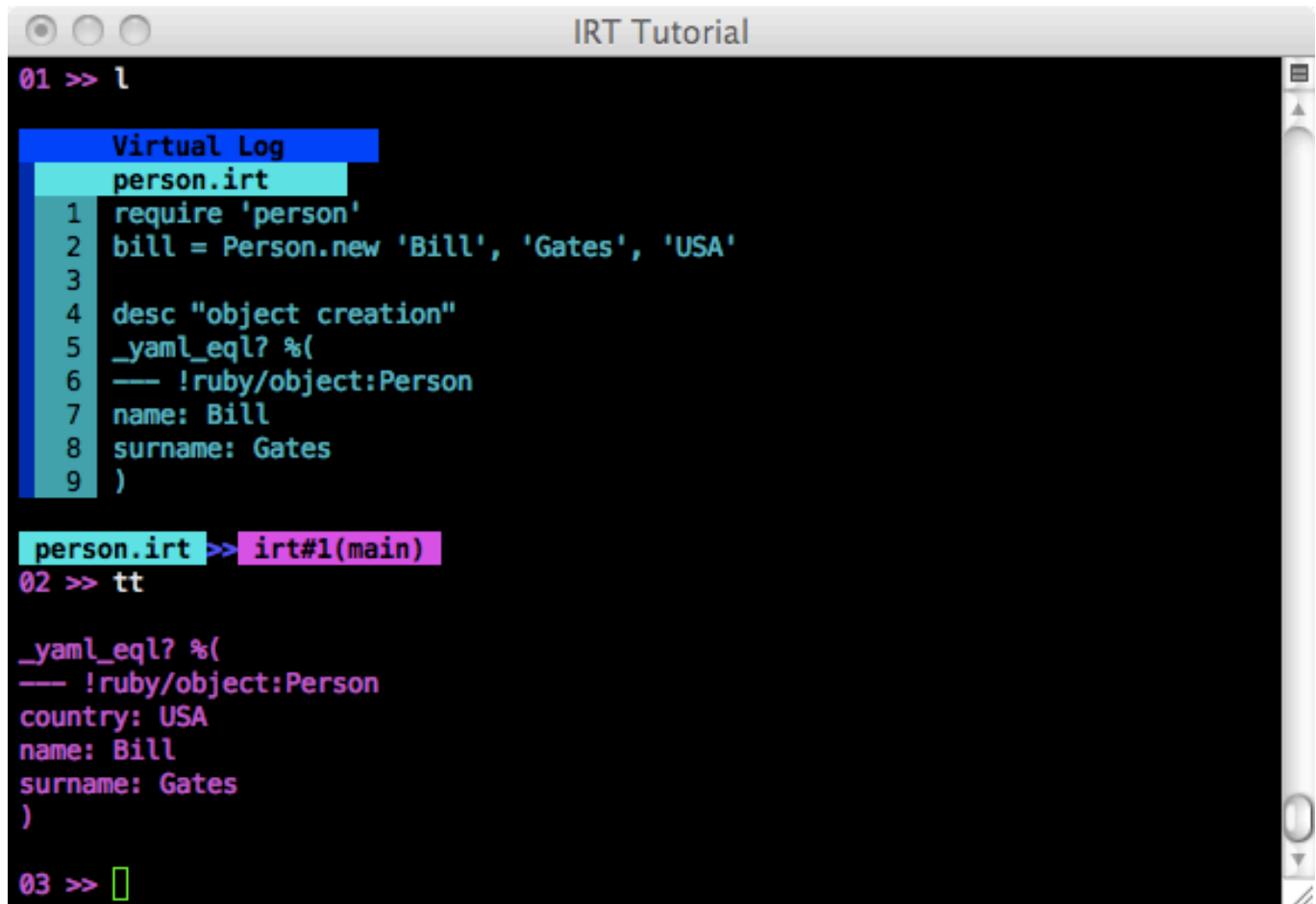
person.irt >> irt#1(main)
01 >> 

```

Well, now the error is gone, but the object has a different value from what we saved before, so the test failed and IRT opened an interactive session at the last eval'd line. Besides, IRT printed a nice diff report, that clearly highlight what is different between the saved and the actual value we got.

We can take a look the log again, just to remember what was the saved value (see the image below)

You might be tempted to manually add the missing value in the serialized string, after all it is very simple, but there is a safer way to do it (specially when you have more complex objects): just type 'tt' without any added description and you will get the test for the actual value we got now: you better substitute the old for the new.



```
IRT Tutorial
01 >> l
Virtual Log
person.irt
1 require 'person'
2 bill = Person.new 'Bill', 'Gates', 'USA'
3
4 desc "object creation"
5 _yaml_eql? %(
6 — !ruby/object:Person
7 name: Bill
8 surname: Gates
9 )
person.irt >> irt#1(main)
02 >> tt
_yaml_eql? %(
— !ruby/object:Person
country: USA
name: Bill
surname: Gates
)
03 >> []
```

You can do it with the copy-open command that best worked for you. We will do it with our preferred command 'cvi', paste and save, then go back to the session and type 'rr'.

```

03 >> cvi

_yaml_eq1? %(
  — !ruby/object:Person
  country: USA
  name: Bill
  surname: Gates
)

04 >> rr
1 TEST: 0 OKs, 1 DIFF, 0 ERRORS.

Rerunning: `irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt`

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
1. OK! object creation
2. OK! name
3. OK! surname
person.irt >> irt#1(main)
01 >> 

```

Now everything is ok again.

Note about in place edit

Notice: If you are not so used to use CLI text editors, please read the "Note about CLI text editors" in the README.

By looking at the examples till now, you might think that in place edit is not so useful after all. You could leave an editor window opened all the time with the file and just switch the windows back and forth.

Indeed in this context there is no real advantage, but in a real world application, you will soon have dozens of test files and you will probably run the full suite with just one command ('irt path/to/test/dir'), so when some exception is raised or some test fails, you would have to manually find the file among many others, open it and go to the right line.

With IRT you can do it with just one simple command: 'vi' or 'nn' and you will open the right file at the right line. Anyway, remember also that if you just want to look at what went wrong, you have always the 'log' (or 'l') command handy.

Inspecting Sessions

Sometimes you might want just to play with a single object, for example when you want to learn how it works. You don't need to test anything, you are just playing, so you can open a session INTO the object. Our 'bill' object is a deadly simple object, so you don't have so much to learn, but just pretend it is a real complex object, and type:

```
>> irt bill
```

```

01 >> irt bill
person.irt >> irt#1(main) >> irt#2(bill)
01 >> self
=> #<Person:0x1017cf608 @country="USA", @surname="Gates", @name="Bill">
02 >> name
=> "Bill"
03 >> @name
=> "Bill"
04 >> l

Virtual Log (tail)
person.irt
 9 surname: Gates
10 )
11
12 bill.name
13 desc "name"
14 _eq1?( "Bill" )
15
16 bill.surname
17 desc "surname"
18 _eq1?( "Gates" )

person.irt >> irt#1(main) >> irt#2(bill)
05 >> x
<< irt#2(bill)

person.irt >> irt#1(main)
02 >> 

```

Notice:

- The status bar is now showing one more black/white colored label (and prompt)
- Indeed we are in the 'irt#2(bill)' inspecting session, launched from the 'irt#1'

(main)' interactive session, that was launched from the 'person.irt' file.

- In this session `self == bill`, so you have access to all the object `bill` variables and methods.
- In inspect mode the log does not record any step of the session and you cannot add descriptions or tests.
- You can exit from the session by just typing 'exit' or 'x' or 'q'.
- You will be notified by a printed label and the status bar reflecting the new status.
- At exit you will be again in the session you start from.

Suggestion:

Opening an inspecting session is specially useful whenever you are in an interactive session preparing your tests, and you don't want to add anything to the log that is being recorded. An inspecting session is working in the isolating context of the inspected object.

Binding sessions

Let's pretend again that our `Person` class is a really useful class, used from many other objects and classes in our application. Suppose we want to know more about the internal of our code, or about what's going on inside some method, in our case... suppose we want to know more about the constructor.

We can open a binding session at any line in any file executed by IRT by just adding the directive 'irt binding', so open your `Person` class and add that line in the method, then rerun the file:

```
class Person
  attr_accessor :name, :surname, :country
  def initialize(name, surname, country)
    @name = name
    irt binding
    @surname = surname
    @country = country
  end
end
```

```

02 >> vi 'person.rb'
03 >> rr
ALL 3 TESTs are OK!

Rerunning: `irb /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt`

Running: /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt
person.irt >> irt#1(#<Person:0x1017fcb30>)
01 >> self
=> #<Person:0x1017fcb30 @name="Bill">
02 >> name
=> "Bill"
03 >> @name
=> "Bill"
04 >> self.name
=> "Bill"
05 >> surname
=> "Gates"
06 >> @surname
=> nil
07 >> self.surname
=> nil
08 >> y caller
-----
- ./person.rb:7:in `initialize'
- /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2:in `new'
- /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2:in `irb_binding'
- /opt/local/lib/ruby/1.8/irb/workspace.rb:52:in `irb_binding'
- /opt/local/lib/ruby/1.8/irb/workspace.rb:52
09 >> 

```

Notice:

- The status shows a yellow colored label and prompt, which represents the binding session.
- The self is the self at the binding line, so since we are in a constructor, self is an object of class Person.
- The @name has been set while the @surname (and @country) have not (it is after our irt call).
- You can see the callers from the method point of view.


```

09 >> l
Virtual Log
person.irt
1 require 'person'
2 bill = Person.new 'Bill', 'Gates', 'USA'

person.irt >> irt#1(#<Person:0x1017fcb30>)
10 >> tt
NotImplementedError: You cannot add a test in binding mode.
    from /opt/local/lib/ruby/gems/1.8/gems/irt-1.0.0.9dd8564/lib/irt/commands/test.rb:20:in `tt'
    from (irt#1):10:in `initialize'
    from ./person.rb:7:in `initialize'
    from /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2:in `new'
    from /Users/dd/dev/irt/tutorial/irt_tutorial/test/person.irt:2
11 >> vi
12 >> x
<< irt#1(#<Person:0x1017fcb30>)

1. OK! object creation
2. OK! name
3. OK! surname
person.irt >> irt#2(main)
01 >> 

```

Notice:

- In binding mode the log does not record any step of the session
- The commands usually available to an interactive session are not available (e.g. you cannot add descriptions or tests. Notice that the image shows the backtrace from an old version of IRT: now it's filtered and indexed.)
- In binding mode if you call 'vi', 'nano', 'nn' or 'edit' without any argument, IRT will open the file containing the 'irt binding' call with the cursor positioned right at that line (not for 'edit' though), so we can remove the 'irt binding' call from the 'person.rb' file right in the console.
- Type 'x' and the session will be closed, after which the file will continue to run till the 'irt' directive at the end...

Comparing objects

IRT can compare complex objects and print a nice and easy to check graphical diff report of the yaml dump of the 2 objects.

Here we create 'other_bill' Person object and compare it with 'bill'.

```

01 >> l
Virtual Log (tail)
person.irt
 9 surname: Gates
10 )
11
12 bill.name
13 desc "name"
14 _eq1?( "Bill" )
15
16 bill.surname
17 desc "surname"
18 _eq1?( "Gates" )

person.irt >> irt#l(main)
02 >> other_bill = Person.new 'Bill', 'Clinton', 'USA'
=> #<Person:0x101783820 @country="USA", @surname="Clinton", @name="Bill">
03 >> vdiff bill, other_bill

= same a b
= --- !ruby/object:Person
= country: USA
= name: Bill
a surname: Gates
b surname: Clinton

04 >> []

```

Checking the last value

When you invoke an IRT command, the last relevant value does not change (for example when you invoke 'vdiff' as you did before), besides the command is also ignored by the log.

You can check the last value by typing '_' (the variable set by irb to the last value), or you can type 'p', 'pp', 'ap', or 'y' without any argument to inspect it with that method:

```

04 >> _
=> #<Person:0x101783820 @country="USA", @surname="Clinton", @name="Bill">
05 >> p
#<Person:0x101783820 @country="USA", @surname="Clinton", @name="Bill">
06 >> pp
#<Person:0x101783820 @country="USA", @name="Bill", @surname="Clinton">
07 >> ap
#<Person:0x101783820 @country="USA", @surname="Clinton", @name="Bill">
08 >> y
— !ruby/object:Person
country: USA
name: Bill
surname: Clinton
09 >> wanted_typo
NameError: undefined local variable or method `wanted_typo' for main:Object
from (irt#1):9
10 >> l

Virtual Log (tail)
person.irt
10 )
11
12 bill.name
13 desc "name"
14 _eq1?( "Bill" )
15
16 bill.surname
17 desc "surname"
18 _eq1?( "Gates" )
irt#1(main)
2 other_bill = Person.new 'Bill', 'Clinton', 'USA'

person.irt >> irt#1(main)
11 >>

```

Notice:

- 'p', 'pp', 'ap' show the same output for a generic object Person. They offer different specialized formatting for different types of objects, so you can experiment with other types.
- The log ignores all the irt commands and all your typos, because they are not relevant for your code.

Contextual ri doc with autocompletion

In its basic form the 'ri' command can accept a string as the system ri command does (you can even omit the quotes).

```

01 >> ri reverse
----- Multiple choices:
1 ActiveSupport::Multibyte::Chars#reverse
2 Array#reverse
3 IPAddr#reverse
4 String#reverse

02 >> ri 4
----- String#reverse
str.reverse => new_str

Returns a new string with the characters from str in reverse order.

"stressed".reverse #=> "desserts"

03 >> 

```

Notice:

- unlike the system command when the search results in multiple choices, you can just type the index of the choice and get the doc you want with less typing.
- the shortcuts in the list will work until you use another command but 'ri'

The 'ri' command offers also a very useful contextual search, that will find the ri doc of the specific method used by the receiver.

Set 'my_str' to "any string", then start to search for a partial method with autocompletion (type TAB TAB).

```

03 >> my_str = "any string"
      => "any string"
04 >> ri my_str.eq
my_str.eql?    my_str.equal?
04 >> ri my_str.eq

```

complete the command with eql? and you will get the ri doc of the String#eql? method, because my_str.method(:eql?) is implemented by String.

```

04 >> ri my_str.eql?
----- String#eql?
str.eql?(other)  => true or false
-----
Two strings are equal if the have the same length and content.
05 >> 

```

If we use the 'ri' command on a different type of object that implement the same method, we will get the right doc (i.e. the array.method(:eql?) is implemented by Array, so we will get that doc):

```

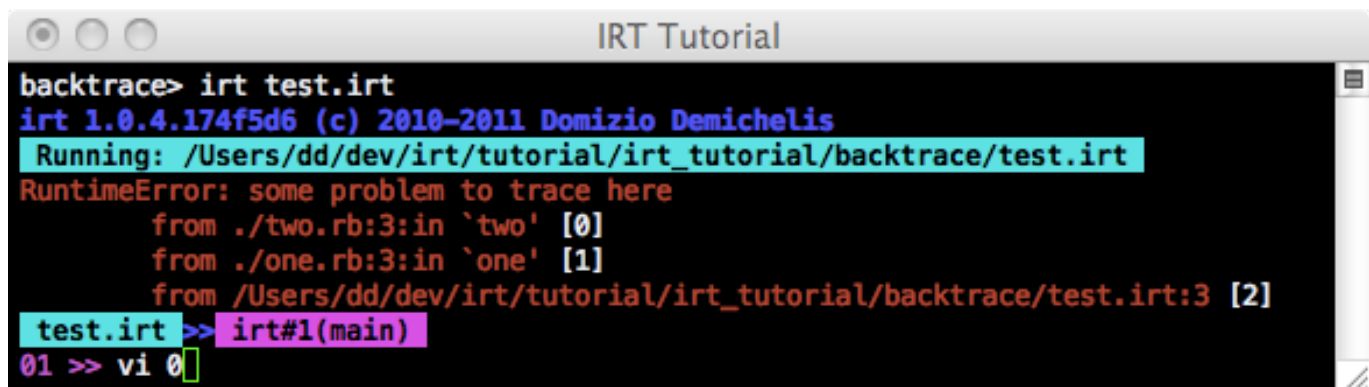
05 >> array = [1,2,3]
      => [1, 2, 3]
06 >> ri array.eql?
----- Array#eql?
array.eql?(other) -> true or false
-----
Returns true if array and other are the same object, or are both
arrays with the same content.
07 >> 

```

Opening backtraced files

When an error occurs, IRT shows you an indexed exception backtrace: as you can see each file:line in the backtrace has an index number (in brackets) that you can use to open that file at that line with your preferred in-place editor.

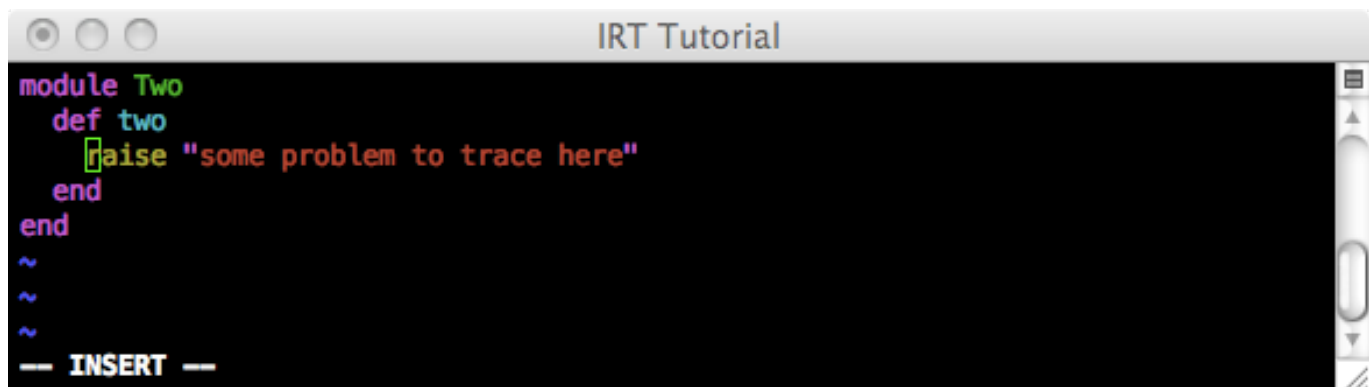
You have just to type '<editor> <index>' (<editor> is one of 'vi', 'nano' (or 'nn') 'edit' (or 'ed'), and <index> is the index number shown in the backtrace), and you will open it in insert mode. Very handy to inspect and fix in place, any code involved in the error.



```

IRT Tutorial
backtrace> irt test.irt
irt 1.0.4.174f5d6 (c) 2010-2011 Domizio Demichelis
Running: /Users/dd/dev/irt/tutorial/irt_tutorial/backtrace/test.irt
RuntimeError: some problem to trace here
    from ./two.rb:3:in `two' [0]
    from ./one.rb:3:in `one' [1]
    from /Users/dd/dev/irt/tutorial/irt_tutorial/backtrace/test.irt:3 [2]
test.irt >> irt#1(main)
01 >> vi 0
  
```

in this case 'vi 0' opens the file ./two.rb at line #3, as you can see below:



```

IRT Tutorial
module Two
  def two
    raise "some problem to trace here"
  end
end
~
~
~
-- INSERT --
  
```

Notice:

- In this case 'vi 2' would have the same effect of just 'vi'. (i.e. both open the .irt file at the error line)
- You can also copy a traceline from another source and paste it as the single editor argument in order to open it. (see the README for details)

Finalizing your suite

The irt executable can run single files or multiple files in a dir, so you can reorganize the .irt files in sub dirs and run the files in one single dir or in all dirs.

You can also organize the file contents a little better if you extract all the common parts (like loading libraries and adding helpers methods) and stick in one or more 'irt_helper.rb' files that will be loaded automatically when irt run (see the README "Note about IRT.autoload_helper_files").

Besides, you can extract reusable parts of your files (complete with code and tests) and use the "insert_file 'path/to/partial/file'" directive and use the inserted files as they were part of other files. (see the README "File insert/eval")

Just remove any irt invocation that might still be in your .irt files. You will run all your test with just one command ('irt path/to/test/dir') without opening any interactive session. Anyway, if anything will go wrong in the future, IRT will open an interactive session at the failing line, so you will always get your interactivity back when you really need it.

More...

Type 'irt_help' and experiment with the other features listed there.
Read also the README file, which contains a lot of informations not mentioned here.

Feedback, Support, Bugs Report, Requests, etc...

Drop me a line at: dd.nexus at gmail.com. I usually answer in less than 24 hours.