Ruby Explorations II

Mark Keane...CSI...UCD

De Basics

Part II: Very, very small programs...

- A: defining programs over several files
- B: looking at defining classes
- C: some methods...if, blocks, iterators
- D: simple I/O and some reflections

What you should now know

Using ***irb*** and ***ruby***

How variables are assigned values

How methods are defined in files

Lots of Ruby kernel methods

```
+
=
==
obj.instance_of?(class)
ios.puts(obj, ...)
puts(obj, ...)
p(obj, ...)
str.length
obj.class
obj.to_s
string.to_i(base=10)
Float.induced_from(obj)
str.insert(index, other_str)
def
```

Part A:

defining programs over several files...

A Suite of Methods I

most programs will involve a suite of classes and methods over many files

methods will invoke other methods to achieve overall task

let's look at the one-file case

```
def get_name
   puts "What is your surname?:"
   name = gets
   print_thanks
   if check_name_ok?(name)
  then print_new_name(name) end
end

def print_thanks
 puts "Thanks for that."
end

def check_name_ok?(nameo)
 if nameo.length > 10
     then error("way too long a name")
 else true end
end

def print_new_name(namer)
 newname = namer + "babygi"
 puts "Your new name is:"
 puts newname
end

def error(sp_message)
 puts "\n**ERROR**:#{sp_message}.\n"
end

get_name
```

name1.rb

seq of calls

BW:
~ make sure double quotes are correct
~ spaces and <cr> as shown

\n is newline

method call


```
def get_name
   puts "What is your surname?:"
   name = gets
   print_thanks
   if check_name_ok?(name)
  then print_new_name(name) end
end

def print_thanks
 puts "Thanks for that."
end

def check_name_ok?(nameo)
 if nameo.length > 10
     then error("way too long a name")
 else true end
end

def print_new_name(namer)
 newname = namer + "babygi"
 puts "Your new name is:"
 puts newname
end

def error(sp_message)
 puts "\n**ERROR**:#{sp_message}.\n"
end

get_name
```


gets
gets user input

simple conditional

printing variable values

file name

name1.rb

Running name1.rb

user added

why ?

```
$ ruby name1.rb
What is your surname?:
marko
Thanks for that.
Your new name is:
marko
babygi
$ ruby name1_with_p.rb
What is your surname?:
marko
Thanks for that.
Your new name is:
"marko\nbabygi"
$
```

use ***p*** to check

```
>> word1 = "gee\n"
=> "gee\n"
>> word2 = "whizz"
=> "whizz"
>> new = word1 + word2
=> "gee\nwhizz"
>> new
=> "gee\nwhizz"
>> puts new
gee
whizz
=> nil
>> word1.chomp
=> "gee"
>> new = word1.chomp +
word2
=> "geewhizz"
>> puts new
geewhizz
=> nil
```

A Suite of Methods II

most programs will involve a suite of classes and methods over many files

so, lets break these up

and deal with the problems that arise...

A Suite of Files

```
require_relative 'thanks'
require_relative 'error'

def get_name
   puts "What is your surname?:"
   name = gets.chomp
   print_thanks
   if check_name_ok?(name)
  then print_new_name(name) end
end

def check_name_ok?(nameo)
 if nameo.length > 10
     then error("name too long")
 else true end
end

def print_new_name(namer)
 newname = namer + "babygi"
 puts "Your new name is:"
 puts newname
end
```

name2.rb

```
def print_thanks
 puts "Thanks for that."
end
```

thanks.rb

```
def error(sp_message)
 puts "\n**ERROR**:#{sp_message}.\n"
end
```

error.rb

Comments on many files...

evaluation of prog is lazy, as needed, so missing file only throws error when the method is explicitly called

nb, if you have same method name in two files, it will be overwritten

classes will help

```
$ ruby name2.rb
ruby name2.rb
name2.rb:27: warning: method
redefined; discarding old
print_thanks
What is your surname?:
markeee
Thanks for that.
Your new name is:
markeebabygi
$
```

This is not OOP !

What we have seen nicely shows how a suite of methods over many files might be built up

it also shows how methods can invoke other methods in other files...

but, this is just bog-standard functional programming rather than OOP-proper; *you will never do this!*

OOP works with objects and their methods

Part B:

looking at defining classes...

My First Class I

In OOP everything is an object; typically an instance of a class

A class is, like, the general type of thing we are going to deal with

An instance, is one specific version of that class, with its specific details filled in

Classes also have their associated methods

object

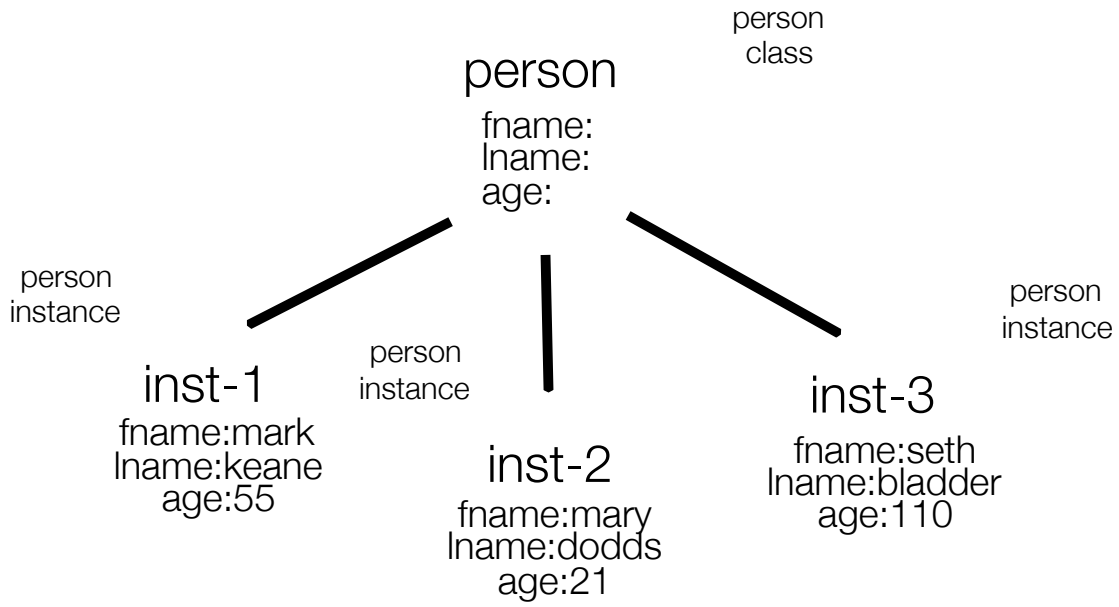
method

"string".length

instance of String Class

method of String Class

My First Class II



My First Class III

the class usually has attributes (virtual & actual) to assigned...

the class has methods (e.g., that assign values to attributes, that do things to instances of that class)

Ruby handles these requirements neatly

Classes I

Person class is defined

one instance created

four methods to give its instance-variables values

nb, instance variables

```
class Person
  def fname
     @fname
  end

  def give_fname(name)
     @fname = name
  end

  def lname
     @lname
  end

  def give_lname(name)
     @lname = name
  end
end

inst1 = Person.new
inst1.give_fname("mark")
inst1.give_lname("keane")

p inst1
```

instance variable

people.rb

```
$ ruby people.rb
#<Person:0x27d80 @lname="keane",@fname="mark">
$
```

Classes II

Person class is defined

three instances created

fname and ***lname*** are used to create and access its attributes and assign, change instance-var values

```
class Person
   attr_accessor :fname, :lname
end

inst1 = Person.new
inst1.fname = “mark”
inst1.lname = “keane”

inst2 = Person.new
inst2.fname = “mary”
inst2.lname = “dodds”

inst3 = Person.new
inst3.fname = “seth”
inst3.lname = “bladder”

p inst3
puts inst3
```

class def creates **new**

we will see its **Person::new**

people1.rb

```
$ ruby people1.rb
#<Person:0x27cb8 @lname="bladder", @fname=“seth">
#<Person:0x27cb8>
$
```

Classes IIa

Person class is defined

three instances created

fname and ***lname*** are still available for access and assignment;but we have neater instance creation

```
class Person
   attr_accessor :fname, :lname
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
end


inst1 = Person.new(“mark”,”keane”)
inst2 = Person.new(“mary”,“dodds”)
inst3 = Person.new(“seth”,“bladder”)

p inst1
```

people1a.rb

```
$ ruby people1a.rb
#<Person:0x27cb8 @lname="keane", @fname="mark">
$
```

Classes IIa

Person class is defined

three instances created

fname and ***lname*** are still available for access and assignment;but we have neater instance creation

```
class Person
   attr_accessor :fname, :lname
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
end


inst1 = Person.new(“mark”,”keane”)
inst2 = Person.new(“mary”,“dodds”)
inst3 = Person.new(“seth”,“bladder”)

p inst1
```

initialize is special

initalize throws weird..

people1a.rb

```
$ ruby people1a.rb
#<Person:0x27cb8 @lname="keane", @fname="mark">
$
```

Classes IIb: attr_assessor?

```
class Person
   attr_accessor :name1, :name2
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
end

inst1 = Person.new(“mark”,”keane”)
p inst1
inst1.name1 = “blibbidy”
p inst1
```

people1a.rb*

```
$ ruby people1a*.rb
#<Person:0x007fb35994b268 @fname="mark", @lname="keane">
#<Person:0x007fb35994b268 @fname="mark", @lname="keane", @name1="blibbidy"
```

Classes IIIa

Person class is defined

we create a print method for the Person Class

can use ***to_s***, as a special case for Person objects, like ***5.to_s***

```
class Person
   attr_accessor :fname, :lname
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end
end

inst2 = Person.new("mark","koo")
p inst2
p inst2.to_s
puts inst2
```

people2.rb

```
$ ruby people2.rb
#<Person:0x27cb8 @lname="koo", @fname="mark">
"hi mark, koo"
hi mark, koo
$
```


Classes IIIa

Person class is defined

we create a print method for the Person Class

can use ***to_s***, as a special case for Person objects, like ***5.to_s***

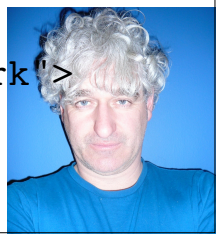
```
class Person
   attr_accessor :fname, :lname
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end
end

inst2 = Person.new("mark","koo")
p inst2
p inst2.to_s
puts inst2
```

people2.rb


```
$ ruby people2.rb
#<Person:0x27cb8 @lname="keane", @fname="mark">
"hi mark, koo"
hi mark, koo
$
```

what the hell is this !



Classes IIIa

Person class is defined

we create a print method for the Person Class

can use ***to_s***, as a special case for Person objects, like ***5.to_s***

```
class Person
   attr_accessor :fname, :lname
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end
end
inst2 = Person.new("mark","koo")
p inst2
p inst2.to_s
puts inst2
```

why not use **print_person** rather than **to_s**

people2.rb

Classes IVa

Person class has *our* method to create instances (what ***new*** +args does)

make_person sort of does what **new** gives you for free; its a *class method*

note use of ***self***

```
class Person
   attr_accessor :fname, :lname, :age
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end
   def self.make_person(fnme, lnme, age)
     new_person = Person.new(fnme,lnme)
     new_person.age = age
     new_person
   end
   def age_person
     @age = @age + 1
   end
end

mk = Person.make_person("mark","keane",53)
p mk
puts mk
p mk.age_person
```

will cause bugs

people3.rb

```
$ ruby people3.rb
#<Person:0x007fcfa39f9050 @fname="mark", @lname="keane", @age=53>
hi mark, keane
```

Classes IVa

Person class has *our* method to create instances (what ***new*** +args does)

make_person sort of does what **new** gives you for free

note use of ***self***

```
class Person
   attr_accessor :fname, :lname, :age
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end

   def self.make_person(fnme, lnme, age)
     new_person = Person.new(fnme,lnme)
     new_person.age = age
     new_person
   end

   def age_person
     @age = @age + 1
   end
end

mk = Person.make_person("mark","keane",53)
p mk
puts mk
mk.age_person
```

self makes it a *class method*

this is an *instance method*

people3.rb

```
$ ruby people3.rb
#<Person:0x007fcfa39f9050 @fname="mark", @lname="keane", @age=53>
hi mark, keane
54
```

Classes IVb

nb, other use of ***self***

method returns last thing evaluated

```
class Person
   attr_accessor :fname, :lname, :age
   def initialize(fname, lname)
     @fname = fname
     @lname = lname
   end
   def to_s
     "hi " + @fname + ", " + @lname
   end

   def self.make_person(fnme, lnme, age)
     new_person = Person.new(fnme,lnme)
     new_person.age = age
     new_person
   end

   def age_person
     self.age = self.age + 1
     @age
   end
end
mk = Person.make_person("mark","keane",53)
p mk
puts mk
p mk.age_person
```

using **self** like a variable

people3.rb

```
$ ruby people3.rb
#<Person:0x007f861a0f2880 @fname="mark", @lname="keane", @age=55>
hi mark, keane
56
```

Teaser

What sort of objects are **true**, **false** and **nil** ?

```
Last login: Wed Aug 31 09:17:38 on console
markkean% rib

>> "marko".class

=> String

>> 5.class
=> Fixnum

>> true.class
=> TrueClass

>> false.class
=> FalseClass

>> nil.class
=> NilClass

>> String.class
=> Class
```

Part C:

more methods...control, blocks and iterations

simple conditionals

Control Ia

if is the conditional

elsif and ***end*** are a great source of bugs

will cause bugs

best use explicit statements, nb **==**

what do these do:

p marko.is_he_old
puts marko.is_he_old

```
class Male
  attr_accessor :name, :age
  def initialize(nme, age)
    @name = nme
    @age = age
  end

  def is_he_old
    age_of_pers = @age
    if age_of_pers > 21
      then puts “yea, an oldie”
    elsif age_of_pers == 21
      then puts “middle_aged...”
    else puts “just a child !”
    end
  end

end
marko = Male.new(“marko”, 55)
marko.is_he_old
```

test it with different inputs

oldie.rb

Control Ib

if is the conditional

elsif and ***end*** are a great source of bugs

will cause bugs

best use explicit statements, nb **==**

turning ***is_he_old*** into a predicate ***is_he_old?***

```
class Male
  attr_accessor :name, :age

  def initialize(nme, age)
    @name = nme
    @age = age
  end

  def is_he_old?
    age_of_pers = @age
    if age_of_pers > 21
       then puts "yea, an oldie"
            true
    elsif age_of_pers == 21
       then puts "middle_aged..."
            false
    else puts "just a child !"
         false
    end
  end

end
marko = Male.new(“marko”, 55)
marko.is_he_old?
p marko.is_he_old?
```

oldie1.rb

what would happen if you gave it “aaah”

blocks... I said...bb...blocks

Block is MethodWithNoName

blocks are one of the most thrilling aspects of Ruby

they allow you to swap new bodies into a method

like Lisp's anon-fns

block is within the squiggly brackets

```
def add_two(no)
  no += 2
end

p add_two(5) => 7

def add_two_b
   yield(10)
end

p add_two_b {|no| no + 2} => 12
```

defblock.rb

Block is MethodWithNoName

blocks are one of the most thrilling aspects of Ruby

they allow you to swap new bodies into a method

like Lisp's anon-fns

block is within the squiggly brackets

body of the method

placeholder for body

```
def add_two(no)
  no += 2
end

p add_two(5) => 7

def add_two_b
  yield(10)
end

p add_two_b {|no| no + 2} => 12
```

defblock.rb

put this block of code into the yield line binding the value to the local variable *no*

Passing water & blocks...

blocks are one of the most thrilling aspects of Ruby

just as you can invoke a method with a value that gets bound to its args

so, too you can invoke a method with a block that is used as its body

note the iterative nature of **upto**

```
>> 3.times{ print "ruby !"}
ruby !ruby !ruby !
=> 3

>> 1.upto(9){|x| print x}
 123456789
=> 1

>> 1.upto(9){|x| p x.to_s + "wagawaga"}
 "1wagawaga"
"2wagawaga"
"3wagawaga"
"4wagawaga"
"5wagawaga"
"6wagawaga"
"7wagawaga"
"8wagawaga"
"9wagawaga"
=> 1
```

Blocks shine in iteration

blocks really come into their own when you use iterative constructs

rem, often the essence of computation is breaking a problem down into small tasks done over and over again...

imagine, I have a book of tickets in one hand and a block that punches holes in the other hand

imagine, I have a simple command that says apply that block to every ticket; that command is **each**

Blocks & ***each*** I

blocks are often used with iterators, like **each**

so, **each** invokes the block-code on each item in the list/array

very useful

```
listy = [“a_”, “b_”, “c_”]
def sheepify(an_array)
  an_array.each {|item| p item + “sheep”}
end

p sheepify(listy)
```

sheep.rb

```
$ ruby sheep.rb
“a_sheep”
“b_sheep”
“c_sheep”
["a_", "b_", "c_"]
$
```

what would happen if you dropped the **p**

Blocks & ***each*** lla

blocks are often used with iterators, like **each**

when the block is more than one line, ***do...end*** is often used instead of brackets

does **object_id** change if you give it "**b**_" twice

Er… Symbols?

```
listy =["a_", "b_", "c_"]

def sheepify_with_do(an_array)
  an_array.each do |item|
     new_sheep = item + "sheep"
     uniq_no = new_sheep.object_id
     puts "#{new_sheep} is no #{uniq_no}"
    end
end

sheepify_with_do(listy)
```

sheepdo.rb

```
$ ruby sheepdo.rb
a_sheep is no 56493
b_sheep is no 56890
c_sheep is no 50933
$
```

Blocks & ***each*** lla

blocks are often used with iterators, like **each**

when the block is more than one line, **do...end** is often used instead of brackets

```
listy =["a_", "b_", "c_"]
def sheepify_with_do(an_array)
  an_array.each do |item|   { acts-like
     new_sheep = item + "sheep"
     uniq_no = new_sheep.object_id
     puts "#{new_sheep} is no #{uniq_no}"
    end                    } acts-like
end
      do...end match
sheepify_with_do(listy)
```

sheepdo.rb

```
$ ruby sheepdo.rb
a_sheep is no 56493
b_sheep is no 56890
c_sheep is no 50933
$
```

Blocks & ***each*** llb

blocks are often used with iterators, like **each**

when the block is more than one line, **do...end** is often used instead

```
listy =[“a_”, “b_”, “c_”]
def sheepify_with_do(an_array)
  an_array.each do |item|
     new_sheep = item + “sheep”
     uniq_no = new_sheep.object_id
     puts “#{new_sheep} is no #{uniq_no}”
  end
end

sheepify_with_do(listy)
```

sheepdo.rb

will cause bugs

```
$ ruby sheepdo.rb
a_sheep is no 56493
b_sheep is no 56890
c_sheep is no 50933
$
```

Our forgotten ***ends***...

most people remember **def...end**

but, you will forget to put in **if...then...end**

and, you will forget to put in **do...end**

sometimes you will forget **{...}**

So, please...please... use indentation

will cause bugs

Other Blocks

Anything in curly brackets or between **do...end** is a block

A block of code to be bandied around

```
def block_eg
  puts “this is the first message”
  yield
  puts “this is the middle message”
  yield
  puts “this is the last\n\n”
end

block_eg {puts “-------CUT HERE-------”}

def block_with_args
  puts “First we say this”
  yield(“CUT”, “HERE”)
  puts “this is the bit we cut out”
  yield(“CUT”, “AGAIN HERE”)
  puts “this is the last bit”
end

block_with_args {|a, b| puts “---#{a} #{b}---”}
```

the block

the block with args

block.rb

More Iteration...

we can also pass blocks to **times**

```
int.times {|i| block}
=> int
```

interesting to see what the value of ***i*** is (see **count**)

```
def five_it(item)
  5.times do |count|
     p item + count.to_s
   end
end

five_it(“boo”)

def print_several(no, item)
  no.times {p item}
end

print_several(10, “gpp”)
```

times.rb

```
$ ruby times.rb
“boo0”
“boo1”
“boo2”
“boo3”
“boo4”
$
```

Part D:

simple I/O and some reflection...

Simple I/O

```
filo = File.open("info.txt", "r")
filo.each_line {|line| puts line + " tag"}
```

```
this is a text file
we will read from it
and we will feel good....
like making fire....
```

info.txt

reader.rb

reading from a file

we call the **open** method from the **File** Class

we give it the file-name/ path and use **each_line**, which closes file too

"r" for read, ***"w"*** for write

nb **each_line** is using a block

```
$ ruby reader.rb
this is a text file
 tag
we will read from it
 tag
and we will feel good....
 tag
like making fire.... tag
$
```

Aside:
Methods & Their Syntax

Differing Method Syntax I

```
"string".length
"string".include?("in")
```

normal dot syntax
obj.method(arg...)

instance methods
invoked via an object
(i.e., instance of class)
with or without
additional parameters

```
class String
  def length
    body-that-finds_length
  end

  def include?(str)
    body-that-matches str
  end
end
```

Differing Method Syntax II

```
a + b          a.+(b)
a = 1          a.=(1)
a == 2         a.==(2)
```

disguised
normal dot syntax
obj.method(arg...)

instance methods with syntactic sugar that is easier to read for us humans

```
class Math
  def +(no)
    body-adds-stuff
  end
end
```

Differing Method Syntax III

```
puts “foo”
puts(“foo”)
```

```
$stdout.puts(“foo”)
```

still
normal dot syntax
obj.method(arg...)

instance methods where parentheses are not required and have implicit objects...but we use the parentheses in this house

```
so...still defined
the usual way...
```

Differing Method Syntax IV

```
Person.new
Person.new("migg", "ure")
File.open("into.txt", "r")
```

still dot syntax
but object is a class
Class.method(arg...)

class methods where we use the class as the object to the method

```
class Person
  def self.new
    creates-instance
  end
end
```

OR

```
class Person
  def self.new(a,b)
    creates-instance
  end
end
```

One Last Mystery...

```
Float(5)
Float 5
```

what the flip !

Well...

Float is a Kernel function that attempts to convert its arguments; 3 others exist Array, String and Integer

Kernel is a mixin for the Object class at the top of the Ruby Object hierarchy

But, don't worry your pretty head about it...

By way of explication...

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

So...

```
>> Float(5)
=> 5.0

>> Kernel::Float(5)
=> 5.0

>> String "foo"
=> "foo"

>> Kernel::String 5
=> "5"

>> Array(1..5)
=> [1, 2, 3, 4, 5]

>> Kernel::Array(1..6)
=> [1, 2, 3, 4, 5, 6]
```

The End