Ruby and Python by Example

20 Aug 2013

After exploring Ruby briefly in a programming languages class last fall, I've finally gotten back around to playing with the language more. Most of my recent experience is in Python, so I've had a lot of fun comparing the two languages and their idioms. This list is a personal reference comparing how the same tasks can be solved in both languages.

In the rest of the post, *Ruby snippets are on the left* and *Python snippets on the right*. In the snippets, *lines of code that correspond between the two languages are aligned*, when possible. These examples were tested with Ruby 2.0.0 and Python 2.7.3.

```
# Ruby snippets on the left
puts "Hello, World!"
```

```
# Python snippets on the right
print "Hello, World!"
```

How to Explore

To run a REPL from the command line:

```
irb
```

```
python
```

I always like to see *what* I can do with various objects:

```
"foo".methods.sort
String.instance_methods.sort
("foo".methods - Object.instance_methods).sort
```

```
dir("foo")
```

To load a file (of function/class definitions, for example) to play with in the REPL:

```
irb -I . -r file.rb
```

```
python -i file.py
```

Printing and String Interpolation

Printing values is useful for confirming expectations and debugging. Ruby's .inspect is similar to Python's repr() in that they both return the string form of the object they are called with. Ruby's "p" function calls .inspect on its arguments.

```
a = "test"
puts a
# test
puts a.inspect
# "test"
p a # equivalent to above
# "test"
```

```
a = "test"
print a
# test
print repr(a)
# 'test'
```

String interpolation/formatting is used to put expression or variable values into a string:

```
puts "value of a: %s" % a
# value of a: test
puts "value of a: #{a}"
# value of a: test
```

```
print "value of a: %s" % a
# value of a: test
print "value of a: {0}".format(a)
# value of a: test
```

Nothing and Truthiness

Testing nothing (represented by nil in Ruby and None in Python):

```
my_var.nil?
```

```
my_var is None
```

True/false values are lowercase in Ruby and capitalized in Python. The following snippets contain the values that evaluate as false in each language:

```
false
nil
```

```
False
None
0
0.0
""
[]
```

Boolean Expressions

The potential gotcha in this category is that Ruby's `and` and `or` operators have very low precedence and thus are generally reserved for control flow. For boolean expressions in Ruby, use && and ||.

```
!false # true

# these short-circuit:
false && true # false
true || false # true
```

```
not False # True

# these short-circuit:
False and True # False
True or False # True
```

Arrays/Lists

An ordered, integer-indexed collection is called an `Array` in Ruby and a `list` in Python.

Instantiation

To make arrays/lists:

```
a = []
b = [1,2,3]
c = Array.new
d = Array[1,2,3]
e = (1..5).to_a
```

```
a = []
b = [1,2,3]
```

```
c = list()
d = list([1,2,3])
e = list(range(5))
```

Operations

Some simple operations available on arrays:

```
a = [1,2]; b = [3,3]
a.include?(1) # true
a + b # [1,2,3,3]

# makes shallow copies:
[a] * 2 # [[1,2],[1,2]]

a[0] # 1
a.first # 1
a[-1] # 2
a.last # 2
a.size # 2
a.length # 2
a.count # 2
b.index(3) # 0
b.count(3) # 2
```

```
a = [1,2]; b = [3,3]
1 in a # True
a + b # [1,2,3,3]

# makes shallow copies:
[a] * 2 # [[1,2],[1,2]]

a[0] # 1

a[-1] # 2

len(a) # 2


b.index(3) # 0
b.count(3) # 2
```

Slicing

Slicing in Ruby is typically done with Range objects, which have inclusive and exclusive forms relative to the end-value as follows: 0..2 contains [0,1,2] and 0...2 contains [0,1]. Basic slicing:

```
vals = [a,b,c,d]
vals[1..2] # [b,c]
vals[1...3] # [b,c]
vals[-3..-2] # [b,c]
vals[-3...-1] # [b,c]

# alternate form:
# array[start_index, length]
vals[1,2] # [b,c]
vals[-3,2] # [b,c]
```

```
vals = [a,b,c,d]
vals[1:3] # [b,c]

vals[-3:-1] # [b,c]
```

Slicing to/from the end of list:

```
vals = [a,b,c,d]
vals[2...vals.size] # [c,d]
vals[2,vals.size] # [c,d]
vals.last(vals.size - 2) # [c,d]
vals[0...2] # [a,b]
vals[0,2] # [a,b]
vals.first(2) # [a,b]
```

```
vals = [a,b,c,d]
vals[2:] # [c,d]


vals[:2] # [a,b]
```

Note: For list comprehensions and iterating, see Blocks.

Hashes/Dicts

A mapping from keys to values is called a `Hash` in Ruby and a `dict`` in Python.

Instantiation

To make hashes/dicts:

```
a = {}
b = {'x'=>1, 2=>2, Fixnum=>3}
c = Hash.new
d = Hash[[['x',1],[2,2]]]
e = Hash.new { |hash,key| hash[key] = [] }
# Blocks! Getting ahead of myself...
```

```
a = {}
b = {'x':1, 2:2, int:3}
c = dict()
d = dict([['x',1],[2,2]])
e = defaultdict(list) # from collections module
```

Operations

Some simple operations available on hashes:

```
a = {'x'=>2}
a['y'] = 5
a.has_key?('y') # true
a.key?('y') # true
a.delete('y') # 5
!a.include?('y') # true
a.keys # ['x']
a.values # [2]
a.to_a # [['x',2]]
```

```
a = {'x':2}
a['y'] = 5
'y' in a # True

del a['y']
'y' not in a # True
a.keys() # ['x']
a.values() # [2]
a.items() # [('x',2)]
```

Symbols

A unique piece of Ruby that you quickly run into is symbols. Symbols are kind of like immutable strings indicated by a prepended colon (e.g. :vehicle) that have performance advantages over regular, mutable strings in Ruby. Symbols are typically used for naming things like hash keys and for referencing variables, method names, etc.

Some symbol operations:

```
:foo.to_s # "foo"
"foo".intern # :foo
"foo".to_sym # :foo
:foo == :foo # true
:foo.object_id == :foo.object_id # true
```

Control Flow

Everything is pretty much the same here except Ruby gives more options (an obvious trend at this point):

If-block:

```
if true
  puts "Gets here"
elsif false
  puts "Doesn't get here"
else
  puts "Definitely not getting here"
end
```

```
if True:
  print "Gets here"
elif False:
  print "Doesn't get here"
else:
  print "Definitely not getting here"
```

Inline if:

```
if answer then "right" else "wrong" end
(answer) ? "right" : "wrong"
puts "You're right!" if answer
puts "You're right!" unless !answer
```

```
"right" if answer else "wrong"

print "You're right!" if answer
```

For-loop (in Ruby, `for` is just sugar for calling `.each` on the given Enumerable):

```
for i in 0..2
```

```
  puts i
end
(0..2).each { |i| puts i }
# Blocks! Coming soon now...
```

```
for i in range(3):
  print i
```

While-loop:

```
while true
  # do stuff
end
until false
  # do stuff
end
```

```
while True:
  # do stuff
```

There's a lot more that could be said here, about break/next, case, and/or, redo and retry, exception-handling, and more.

Methods/Functions

A named chunk of code is a method in Ruby and a function in Python:

```
def myfunc
  puts "Hello!"
end
```

```
def myfunc():
  print "Hello!"
```

The return keyword is optional in Ruby; if it's omitted then the value of the final expression in the method is returned:

```
def myfunc
  1 + 1
end
myfunc # 2
```

```
def myfunc2
  return "foo"
  "nope"
end
myfunc # "foo"
```

```
def myfunc():
  1 + 1

myfunc # None

def myfunc2():
  return "foo"

myfunc # "foo"
```

Ruby methods and Python functions can accommodate optional parameters:

```
def myfunc(x, y=2)
  x + y
end
myfunc(2) # 4
myfunc(2,3) # 5
```

```
def myfunc(x, y=2):
  return x + y

myfunc(2) # 4
myfunc(2,3) # 5
```

Both languages can collect extra arguments into an array:

```
def myfunc(x, y=2, *args)
  args
end
myfunc(1, 2, 3, 4, 5) # [3,4,5]
```

```
def myfunc(x, y=2, *args):
  return args

myfunc(1, 2, 3, 4, 5) # (3,4,5)
```

Ruby allows key-value pairs as arguments and both languages can collect extra keyword arguments:

```
def myfunc(req, optional=2, *args,
           foo: 'bar', **options)
  p args
  p foo
  p options
end
myfunc(1, 2, 3, hey:'there', hi:3)
# [3]
# "bar"
# {:hey=>"there", :hi=>3}
```

```
def myfunc(req, optional=2, *args, **kwargs):
  print args
  print kwargs

myfunc(1,2,3, hey='there', hi=3)
# (3,)
# {'hi': 3, 'hey': 'there'}
```