Python for Ruby Programmers

MIKE LEONE
LA RUBYCONF 2013
FEB 23 2013
This presentation:

1. Fast-paced
2. Python in 30 mins
3. It's easy!
About Me
Web  Mobile  Embedded
That's it.
Ruby: 2006
Python: 2010
New appreciation
Better Rubyist
What we'll cover
1. Example program
2. 13 Design similarities
3. 13 Differences
4. 5 Problem domains
5. “Feelings”
What we WON'T cover
1. Implementations
2. Performance (Good enough)
3. Parallelism (GIL)
4. Minor syntax (You'll get it)
What I'll try to convince you:
1. VERY similar!
2. You're already ready
3. Python = Better Rubyist
4. Hire a Pythonista
Zen of Python (abridged)
1. Explicit is better than implicit.
2. Flat is better than nested.
3. Readability counts.
4. There should be [only] one obvious way to do it.
5. Namespaces are a great idea -- let's do more of those!
Example:
Create a Person class
1. Name and age
2. Can greet you
3. Knows if it's Beiber
4. Greets Google directors
class Person
  GOOGLE_DIRECTORS = ["Larry Page", "Sergey Brin", "Eric Schmidt"]

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

  def say_name_and_age
    puts "My name is " + @name + " and my age is " + @age.to_s
  end

  def is_justin_beiber?
    @name == "Justin Beiber" && @age == 18
  end

  def greet_google_directors
    Person::GOOGLE_DIRECTORS.each do |name|
      puts "Hello " + name
    end
  end
end
class Person(object):
    GOOGLE_DIRECTORS = ["Larry Page", "Sergey Brin", "Eric Schmidt"]

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_name_and_age(self):
        print("My name is " + self.name + " and my age is " + str(self.age))

    def is_justin_beiber(self):
        return self.name == "Justin Beiber" and self.age == 18

    def greet_google_directors(self):
        for name in Person.GOOGLE_DIRECTORS:
            print("Hello " + name)
13 Similarities
1. Dynamically Typed
def print_object(some_sorta_object):
    print some_sorta_object.read()

print_object(my_file)

>>> "This is the file contents"
2. Everything is an Object
   (even functions!)
Python example:

5 + 3
```python
>>> 8
```

(5).__add__(3)
```python
>>> 8
```

(5).__str__()
```python
>>> "5"
```
3. Arrays
Ruby (array)

```ruby
things = [1, "hello", 2.5, 43]
things[1]
>> 'hello'
```
things = [1, "hello", 2.5, 43]
things[1]
>> 'hello'
Ruby (hash)

color_map = {
  red: "#FF0000",
  purple: "#800080"
}

color_map[:red]
>> "#FF0000"
color_map = {
    "red": "#FF0000",
    "purple": "#800080"}

color_map["red"]
>> "#FF0000"
5. No special line terminators (no semicolons!)
6. Strong functional programming paradigms
def triple(number):
    return number * 3

list = [1, 2, 3, 4]

map(triple, list)

>>> [3, 6, 9, 12]
def is_even(number):
    return number % 2 == 0

list = [1, 2, 3, 4]

filter(is_even, list)
>>> [2, 4]
Anonymous Functions
(lambda)

```
list = [1, 2, 3, 4]
map(lambda x: x * 3, list)
>>> [3, 6, 9, 12]
```
7. Awesome function parameters
Optional / named arguments
```python
def greet(name, greeting="Hello"):
    print greeting + ", " + name

greet("Bob")
>>> Hello, Bob

greet("Bob", greeting="Bonjour")
>>> Bonjour, Bob

greet("Bob", "Bonjour")
>>> Bonjour, Bob
```
Argument Unpacking
def greet_everyone(*names):
    for name in names:
        print "Hello, " + name

names = ["Bob", "Steve", "Jim"]

greet_everyone(*names)
Hello, Bob
Hello, Steve
Hello, Jim
8. Raising Exceptions
inputted_age = 30

if inputted_age > 25
    raise ArgumentError, "must be under 26"
end

ArgumentError: must be under 26
(stack trace...)

Ruby
inputted_age = 30

if inputted_age > 25:
    raise ValueError("must be under 26")

(stack trace...)
ValueError: must be under 26
9. Handling Exceptions
Ruby (rescue)

begin
  23 / 0
  puts "all good"
rescue ZeroDivisionError
  puts "can't do that, bro"
end

>> can't do that, bro
try:
    23 / 0
    print "all good"
except ZeroDivisionError:
    print "can't do that, bro"

>> can't do that, bro
10. One-line conditionals
"equal" if 3 == 3
>> "equal"

2 == 3 ? "equal" : "not equal"
>> "not equal"
if 3 == 3: "hello"

>>> "hello"

"equal" if 2==3 else "not equal"

>> "not equal"
11. Nice interactive shells
$ python
Python 2.7.3 (default)
>>> 2 + 2
4

$ irb
irb(main):001:0> 2 + 2
=> 4
12. Strong object reflection features
steve = Person("Steve Ballmer", 54)

steve.__dict__
>>> {'age': 54, 'name': 'Steve Ballmer'}

steve.__class__
>>> <class 'person.Person'>

dir(steve) # all methods/attributes
['GOOGLE_DIRECTORS', '__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'greet', 'greet_google_directors', 'is_justin_beiber', 'name', 'say_name_and_age']
13. Radically simpler than C++/Java!
13 Differences
1. No blocks
Blocks are the killer feature of Ruby
1. Powerful metaprogramming
2. DSLs
3. Great frameworks (Rails)
Decorators are a powerful Python feature similar to blocks.
They let you inject or modify code in a function, like blocks. But not enough time.
2. First-class Functions
Functions as variables
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def process_numbers(a, b, func):
    return func(a, b)

process_numbers(20, 5, add)
>>> 25

process_numbers(20, 5, subtract)
>>> 15
Similar behavior is possible in Ruby, but you must wrap functions in procs.
3. One-line lambdas, by design
Can't do this in Python:

```ruby
[1, 2, 3, 4, 5].map do |num|
  if num % 3 == 0
    0
  elsif num % 4 == 0
    num * 2
  else
    num
  end
end
```
But do you really want to do that?

1. Harder to test
2. Harder to follow
3. Accidentally duplicate
Python pushes you to factor things out into granular methods.
def process_num(num):
    if num % 3 == 0:
        return 0
    elif num % 4 == 0:
        return num * 2
    else:
        return num

return map(process_num, [1,2])
Python even supports inner methods to this end.
def start_program():
    def process_num(num):
        if num % 3 == 0:
            return 0
        elif num % 4 == 0:
            return num * 2
        else:
            return num
    return map(process_num, [1, 2])
4. Python has "tuple," an immutable list.
[1, 2, 3]  # can change

(1, 2, 3)  # can't change

(Should be homogeneous, but not enforced)
5. Python has more fine-grained importing
require 'open-uri'
result = open("http://google.com")

You bring in everything from open-uri

With lots of requires, which one is responsible?
from urllib2 import urlopen
response = urlopen("http://google.com")

“Explicit is better than implicit.”

You can write from urllib2 import *, but not a best practice.
You can import any number of specific functions or variables from a python module(file).
6. Python enforces indentation
In practice, this is pretty awesome.
Doesn't cause headaches. Makes code style more consistent.
7. Python has more values that evaluate to False.
False

None (nil)

[] , (), {} 

" 

0
8. Instead of Enumerable, Python has built-in functions
filter()  
map()  
reduce()
Convenient, but not as powerful as Enumerable.
9. Python has simpler conditionals
No case statements
(use if... elif... else instead)

if n == 0:
    print "You typed zero.\n"
elif n == 1 or n == 9 or n == 4:
    print "n is a perfect square\n"
elif n == 2:
    print "n is an even number\n"
elif n == 3 or n == 5 or n == 7:
    print "n is a prime number\n"
unless !person.present? && !company.present?
  puts "do you even know what you're doing?"
else
  puts "and now we're really confused"
end
10. No automatic return values
Functions return None if they don't have an explicit “return” statement.

def add(a, b):
    a + b

add(2, 3)
>>> None
11. “self” as an argument to every class instance method
Pro: Explicit
No clashes with local variables or other imported objects

Con: More verbose
Class Person(object):
    def __init__(self, name):
        self.name = name

    def say_hi_to(self, name):
        print self.name + " says hi to " + name

Person("Steve").say_hi_to("Bob")
>>> "Steve says hi to Bob"
12. No powerful module mixins like Ruby.
Multiple inheritance only (yuck)
13. Ruby has stronger metaprogramming features
Python doesn't have

define_method
class_eval
method_missing
Can't reopen a class and extend it

Can't extend built-in types (maybe for the better?)
5 Problem Domains
1. Web Development
Ruby: Rails, Sinatra

Ruby community is dominated by web development
Python: Django, Zope, Flask, Pylons.

Web dev is just one piece of the gigantic python community.
2. Mobile Applications
Neither language is popular for mobile apps.

Interesting Ruby projects: Rhodes, RubyMotion.
3. Desktop Applications
Python has PyQt, wxPython, PyGTK.

Ruby has shoes, wxRuby, RubyGTK, but not popular or mature.
4. Scientific Programming
Python has numpy, scipy. Widely used.

Ruby doesn't have any mature scientific/numeric libraries.
5. Windows Deployment (if you must)
Ruby: RailsInstaller is great. Doesn't target deployment.

JRuby makes windows deployment much easier.
But Python on windows is generally easier.

Why? Because entire toolchain is well-supported.

Windows neglect is self-perpetuating.
The Python community is just bigger.
“Feelings”
It's harder to write code that pisses off other developers in Python.
1. Smaller grammar
2. Forced indentation
3. Fewer ways to handle conditionals
4. The source of imported functionality is obvious
5. Explicit class and method structure makes code easier to follow.
Ruby is more elegant, but in practice, Python is easier to read.
Rails still wins for web development.
Nothing as good as Rails will ever be implemented in Python.

(metaprogramming, reflection)
Need to hire a Rubyist? Hire a Pythonista instead.
1. Ruby developer market is out of control.

2. There are more Python developers.

3. Pythonistas can ramp up EASILY (hours/days, not weeks/months)
Wrap-up
1. VERY similar!
2. You're already ready
3. Python = Better Rubyist
4. Hire a Pythonista
THANK YOU!