Why I like Python, and quite like Ruby

By Tibs / Tony Ibbs (they / he)

Presented at CamPUG, virtually, Tuesday 3rd August 2021

Written in reStructuredText, converted to PDF using rst2pdf.

Slides and accompanying material at
https://github.com/tibs/why-I-quite-like-Ruby

What we shall cover

- Python
- Ruby

Part 1: Python

Finding Python

Some time around 1994, Python 1.0 or 1.1

Previously: BCPL, Fortran, C, Emacs Lisp, TeX/LaTeX, VMS DCL

(For reference, Java 1.0 is 1995, as is Ruby 0.95)

Significant indentation, and the aim to be readable

Python is "runnable pseudocode".

```
if something:
   do_other_thing()
```

Python emphasises *readability*, even if that makes writing code a bit harder.

Multi-paradigm

The language is object oriented. But the programs do not need to be.

High level datastructures built in

In languages like C, it is *so boring* to have to rewrite these again and again.

Dictionaries all the way down

Python is malleable

Metaprogramming is easier than one might expect.

And in many ways Python itself is malleable.

But, mostly, Python programmers *don't do this*

Values or methods?

I love the fact that you can start with a value:

```
class UsefulNumbers:
     random = 4
```

and later on realise that it should be a method:

```
import random
class UsefulNumbers
   @property
   def random(self):
      return random.random()
```

and the user need not care.

"Safe and sane"

Python programmers do not have a reputation for being wild and wacky in their code.

(Even though we could be if we wanted to.)

Batteries included

Certainly compared with the other languages I was used to.

The community and a gentle sense of humour

Not, by any means, unique to Python.

- `import ni`
- Guido's time machine
- https://github.com/DRMacIver/schroedinteger
- https://pyos.github.io/dg/
- https://docs.hylang.org/ (not actually a joke)

Docstrings - these came later

Docstrings were introduced in Python 1.2 in 1995.

They were "playtested" *in the existing Python implementation* before being formally adopted.

The Zen of Python

"The Zen of Python" is a joke by Tim Peters from 1999.

```
>>> import this
```

Too much other stuff to go into

- `f` strings
- `__repr__` versus `__str__`
- numbers with underlines in them (they make my life a lot easier)
- `mypy` typing

Lots of other things

Part 2: Ruby

Why did I learn Ruby?

In July 2019, the team I work on moved from Python/Django to Ruby/Rails.

Caveat: I learn Ruby and Rails at the same time

And Rails is at least as opinionated in how it uses Ruby as Django is in how it uses Python.

(Rails likes "magic" even more than Django does.)

Also, I've only been using Ruby a short time.

Matz

Matz is Yukihiro Matsumoto, the creator of Ruby.

https://en.wikipedia.org/wiki/Yukihiro_Matsumoto

There is a saying in the Ruby community: "Matz is nice so we are nice"

Ruby's inspirations

Finally, like Python, Ruby is solidly built on well-proven ideas from programming history. Just not the same ideas.

Smalltalk, Lisp, Perl, etc.

Origins: Ruby's lisp features

Matz in an email in 2006:

> Ruby is a language designed in the following steps:
>
> - take a simple lisp language (like one prior to CL).
> - remove macros, s-expression.
> - add simple object system (much simpler than CLOS).
> - add blocks, inspired by higher order functions.
> - add methods found in Smalltalk.
> - add functionality found in Perl (in OO way).

So, Ruby was a Lisp originally, in theory.

Let's call it MatzLisp from now on. ;-)

Why do I say I only "quite" like Ruby?

Ruby errs a *little* too much on the magic side for me.

But I love the fact that Ruby takes some very different approaches than Python, while fitting in the same programming "space".

Readability / writability

Ruby wants to make programming "a joy for programmers", so it wants code that is easy/fun to write, as well as easy to read.

Synonyms and extra methods

Ruby is much happier with "more than one way to do things", if it makes the programmer's life easier. For instance:

```
hash.each_key do |k|
   ...
end
```

as well as (the less colloquial)

```
hash.keys.each do |k|
   ...
end
```

Begin and end and things

Ruby doesn't have significant indentation, but its block delineation is nice.

```
begin
  ...
end
```

```
if choice
  ...
elsif some_other_choice
  ...
end
```

Line continuation

```
difference = minimum -
             maximum
```

and

```
allow(ledger).to receive(:record)
  .with(expense)
  .and_return(RecordResult.new(true, 417, nil))
```

I don't think I need to say any more...

Strongly object oriented, but easy to use...

What do we mean by "Object Oriented"?

1. *Encapsulation*
2. *Protection*
3. *Ad hoc polymorphism*
4. *Parametric polymorphism*
5. *Everything is an object*
6. *All you can do is send a message* (AYCDISAM)
7. *Specification inheritance*
8. *Implementation inheritance/reuse*
9. *Sum-of-product-of-function pattern*

"an a la carte menu" - Jonathan Rees on the meaning of Object-Oriented (2001)

Ruby still feels like a multi-paradigm language

This is a perfectly good Ruby program:

```
puts "Hello"
puts "====="
```

No self

This is for information, not because I'm keen on it. I *like* explicit `self`. But lots of people don't.

Object values

Ruby uses setter and getter methods for (almost) all value access, but it makes it so easy to create those that you don't really think about it.

Readonly values

```ruby
class Rectangle
  attr_reader :width, :height
  def initialize(width, height)
    @width = width
    @height = height
  end
end
```

```ruby
r = Rectangle.new(1,2)
r.width = 3
```

```
in `<main>': undefined method `width=' for
  #<Rectangle:0x00007fe9bc9520d8 @width=1, @height=2> (NoMethodError)
Did you mean?  width
```

Writable values

```ruby
class MutableRectangle
  attr_accessor :width, :height
  def initialize(width, height)
    @width = width
    @height = height
  end
end

m = MutableRectangle.new(1,2)
m.width = 3
m.width             # => 3
```

Doing it "by hand"

```ruby
class Example
  def value=(v)
    @value = v
  end
  def value
    @value
  end
end
```

```ruby
e = Example.new
e.value              # => nil
e.value = 3
e.value              # => 3
```

? and ! at the end of method names

Methods ending with ? should return a boolean, for instance

```
[].empty?    # => true
```

Methods ending with ! should do something permanent or potentially dangerous, and should generally be paired with an equivalent method that doesn't end with !.

For instance:

```
Enumerable#sort   # returns a new sorted object
Enumerable#sort!  # sorts in place, mutating the object
```

and, in Rails:

```
ActiveRecord::Base#save   # returns false if saving failed
ActiveRecord::Base#save!  # raises an exception
```

Symbols

What is a symbol?

According to Programming Ruby

> A Ruby symbol is an identifier corresponding to a string of characters, often a name.

Somewhat simplistically, it's a constant whose value is itself.

For instance:

```
:symbol
```

Messages from smalltalk

In Ruby, the documentation would have it that:

```
obj.thing
```

sends the message `thing` to the object `obj`, which will respond appropriately if it understands that message.

The `send` method makes this explicit:

```
obj.send(:thing)
```

`obj.send(:thing)` effectively calls `obj.thing`.

You can use `send` to call private methods.

```ruby
class Something
  # ...
private
  def reset
    # ...
  end
end
```

```ruby
s = Something.new
s.reset            # Ruby says you're trying to call a private method
s.send(:reset)     # Ruby calls the method for you
```

One can ask if an object understands a message:

```
s.responds_to?(:reset)  # => false, because it's private
3.responds_to?(:times)  # => true
```

And catch messages as they "go past":

```ruby
class Example
  def method_missing(name, *args, &block)
    if name == :random
      4
    else
      name.to_s
    end
  end
end
```

```ruby
e = Example.new
e.random               # => 4
e.aha                  # => "aha"
e.whatever             # => "whatever"
```

Ruby and monkey patching

Caveat

The Ruby Style Guide says:

> **No Needless Metaprogramming**
>
> Avoid needless metaprogramming.
>
> **No Monkey Patching**
>
> Do not mess around in core classes when writing libraries (do not monkey-patch them).

Old-style monkey patching

Very simple to do, quite nice to write, but rather too powerful for its own good.

This is quite nice - open the `String` class and add a method:

```
class String
  def prefix_with_hat
    "^#{self}"
  end
end
```

```
'abcd'.prefix_with_hat    # => '^abcd'
```

(this and the following section borrow from
https://6ftdan.com/allyourdev/2015/01/20/refinements-over-monkey-patching/)

But what if we change an existing method? It looks very similar:

```
class String
  def reverse
    self.prefix_with_hat
  end
end
```

```
'abcd'.reverse    # => '^abcd'
```

We have changed *all* uses of the `reverse` method, wherever they may be.

Refinements

Refinements give more control.

```
module HattyString
  refine String do
    def reverse
      self.prefix_with_hat
    end
  end
end
```

```ruby
class A
  using HattyString
  def a(str)
    str.reverse
  end
end

class B
  def a(str)
    str.reverse
  end
end
```

and now we've isolated the changes:

```ruby
A.new.a('abcd')   # => '^abcd'
B.new.a('abcd')   # => 'dcba'
```

Blocks

I think everyone is required to mention blocks when talking about Ruby.

Ruby blocks are (essentially) anonymous functions that can be passed to methods.

It's not really possible to have a nice syntax for this in Python, because of significant indentation. But that's OK, we don't have to have everything!

Who needs a `for` loop?

```ruby
(1..3).each do |index|
  puts index
end
```

prints out:

```
1
2
3
```

Aside on ranges

If that inclusive range feels wrong, Ruby has an alternative:

```
(1...3).each do |index|
  puts index
end
```

prints out:

```
1
2
```

Nice example from The Ruby Style Guide

```ruby
def with_io_error_handling
  yield
rescue IOError
  # handle IOError
end

with_io_error_handling do
  something_that_might_fail
end
```

This shows a nice use of blocks to wrap code in much the same way as we would use a context manager (and `with`) in Python.

Although that's bad style

Actually, it's generally bad style to use the `do .. end` notation for blocks that could easily (and perhaps more readably) fit on one line.

So our previous example should *actually* be written:

```
with_io_error_handling { something_that_might_fail }
```

using the in-line `{ .. }` notation.

And whilst I still dislike `{` and `}` as the *only* block delimiters, I must admit that this convention actually works quite well.

Lisp-1 or Lisp-2

At the start of https://bugs.ruby-lang.org/issues/15799#note-29 Matz says:

> Unlike JavaScript and Python (Lisp-1 like languages), Ruby is a Lisp-2 like language, in which methods and variable have separated namespaces. In Lisp-1 like languages, `f1 = function; f1()` calls function (single namespace).

So in Python we expect to be able to do:

```
fn = len
fn([1, 2, 3])                          # => 3
sorted(['abc', 'x', 'de'], key=len)  # => ['x', 'de', 'abc']
```

and

```
a = 3                                  # gives us 'a'
def a(): print('A')                    # overwrites 'a'
```

Ruby does not work like that, and passing around methods takes a little more work.

Bare callables

(IS THERE A PROPER NAME FOR THIS?)

In Python:

```
callable
```

just "sits there". You need to use the ( ) (call) operator to make something happen:

```
callable()
callable(1, 2, 3)
```

In Ruby:

```
callable
```

will call the method of that name (if there is one).

Of course, because Ruby allows a value and a method to have the same name, it does have to do a little guesswork in some contexts to decide which is needed.

Omitting (and)

On the other hand, since Ruby knows that a method is not a value, it is free to treat it differently. So the ( ) can be optional.

(There are stylistic guidelines, of course - see The Ruby Style Guide section DSL Method Calls)

```
method(1, 2, 3)       # OK
method 1 2 3          # often more colloquial
```

I think that this can often be *much more readable.*

Which leads to DSLs

A DSL is a Domain Specific Language.

Ruby is often said to be good for "creating" domain specific languages, but what I think that actually means is that, given blocks and the ability to elide ( ) when calling methods, one can end up with something that already looks like a DSL.

DSL example 1: bundle/gem files

Very nice configuration files that read naturally, but are actually Ruby code.

Somewhat randomly:

```
ruby "2.1.3"
gem "nokogiri", ">= 1.4.2"
git "https://github.com/rails/rails.git" do
  gem "activesupport"
  gem "actionpack"
end
group :development, :optional => true do
  gem "wimble"
  gem "womble"
end
```

DSL example 2: rspec

rspec is (effectively) a Ruby DSL, providing Behaviour Driven Development.

It gets close to being a Cucumber language in pure Ruby, and also provides Hamcrest-like abilities as well.

There's a rather good book called Effective Testing with RSpec 3

Here's a simple example from the front page of the rspec website:

```
require 'bowling'

Rspec.describe Bowling "#score" do
  context "with no strikes or spares" do
    it "sums the pin count for each roll" do
      bowling = Bowling.new
      20.times { bowling.hit(4) }
      expect(bowling.score).to eq 80
    end
  end
end
```

and if you run that (and `bowling` has been implemented) you might see:

```
/rspec --format doc

Bowling#score
  with no strikes or spares
    sums the pin count for each roll

Finished in 0.00137 seconds (files took 0.13421 seconds to load)
1 example, 0 failures
```

Here's another example, this time from page 68 of Effective Testing with RSpec 3:

```
it 'returns the expense id' do
  expense = { some: 'data' }

  allow(ledger).to receive(:record)
    .with(expense)
    .and_return(RecordResult.new(true, 417, nil))

  post '/expenses', JSON.generate(expense)

  parsed = JSON.parse(last_response.body)
  expect(parsed).to include('expense_id' => 417)
end
```

DSL Example 3: Sonic Pi

Sonic Pi is "a code-based music creation and performance tool".

From their web page, IDM Breakbeat:

```
define :play_bb do |n|
  sample :drum_heavy_kick
  sample :ambi_drone, rate: [0.25, 0.5, 0.125, 1].choose,
    amp: 0.25 if rand < 0.125
  sample :ambi_lunar_land, rate: [0.5, 0.125, 1, -1, -0.5].choose,
    amp: 0.25 if rand < 0.125
  sample :loop_amen, attack: 0, release: 0.05, start: 1 - (1.0 / n),
    rate: [1,1,1,1,1,1,-1].choose
  sleep sample_duration(:loop_amen) / n
end
loop {play_bb([1,2,4,8,16].choose)}
```

The community

As I said earlier, not unique to Python.

I've only attended one Ruby conference so far, Euruko 2021, which unfortunately had to be virtual. But all the evidence I've seen leads me to think that the Ruby community is just as friendly and helpful (although possibly slightly smaller outside Japan) as the Python community.

Why the Lucky Stiff (optional slide)

To a programmer of a certain age, Ruby's Why the Lucky Stiff was a very distinct presence on the scene. I'm not aware of anything quite like his work in any other programming language.

The book "Why's (poignant) guide to Ruby" is available online at http://poignant.guide/, and there is an interesting documentary about the person and the book at https://www.youtube.com/watch?v=64anPPVUw5U.

Python, Ruby and "unexpected consequences" (optional slide)

Because Python has significant indentation, it can't really (easily) have blocks in the Ruby style.

Because Ruby is a Lisp-2, it has to do some guesswork, sometimes, to decide whether to use a value or a method.

Because Ruby allows leaving off `()` when calling methods, which it can safely do because it is a Lisp-2, it also allows the creation of (apparent) DSLs, like `rspec` and the bundle/gem file format

Fin

Written in reStructuredText, converted to PDF using rst2pdf

Slides and accompanying material at
https://github.com/tibs/why-I-quite-like-Ruby

Where next?

My heart is with Python, and I'm currently paid to write in Ruby, so what language should I think about next?

Well, for various reasons (and despite some residual prejudice I have left over from the 1980s), it looks as if the obvious answer is Common Lisp.

```
CL-USER> (defun hello ()
           (format t "Hello, World!~%"))
HELLO
CL-USER> (hello)
Hello, World!
NIL
CL-USER>
```