# Why I like Python, and quite like Ruby

## Notes for the slides

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

> A short explanation of particular reasons I like Python, followed by a discussion of some of the things that also make me like Ruby, with comparison between the two languages.

## What we shall cover

We're going to start with how I found Python, and some of the things I like about it.

Then we'll talk about why I learnt Ruby, and some the of the things I like about *it*.

## Part 1: Python



The Python logo is a trademark of the Python Software Foundation

Some of the material here is borrowed from my own An amble through some of Python's history, which talks about the history of Python (and its features) from the early days until Python 3.6.

## Finding Python

I don't remember exactly when I first found Python, but my first dated Python script is from 1994, and that would fit, as it is when Python 1.0 and 1.1 were released.

My background had been with BCPL, Fortran and C (and also Emacs Lisp and TeX/LaTeX and VMS DCL).

Work had tasked me with looking for a programming language we could embed into our object oriented database. At that time, the obvious choices were things like Tcl and various small lisps or schemes. But I also found Python, and fell in love with it, in large part because it was what I already wrote down when writing pseudo-code (with the exception of the colons - more on them later too). It also had much of the feel I had liked from using VMS DCL (Digital Command Language - a relatively sophisticated Basic descendant, with close integration with the operating system).

As it happened, we didn't choose any of my found languages (we ended up with a simplified Pascal clone from Byte magazine), and I think at the time we would have thought Python was too much work to embed.

(For reference, Java 1.0 is 1995, as is Ruby 0.95. Perl 1.0 was released in 1987)

## Significant indentation, and the aim to be readable

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

I'd already seen the use of significant indentation in the occam programming language, some years before, although I'd never used that language.

When writing pseudocode for myself, of course, I didn't see any need for `begin` and `end` (or `${` and `}`, or, most horrible of all `{` and `}`), but naturally just indented the code and left it at that.

So when I say Python, that aspect of it just seemed natural.

Also the decision, in general, to be relatively unadventurous in syntax made it easy to read and easy to learn (remember Python was also a somewhat smaller language in the 1990s).

As someone coined, Python is "runnable pseudocode".

(and I like the colons)

Of course, when I read further I discovered that Guido van Rossum had deliberately decided to emphasise *readability* as being valuable, even if that added a little more burden when writing code. The reasoning being that programs are communication, to other human beings as much as computers, and so we should do as much to make that communcation easy and pleasant as we can.

Being readable may also be important to the original programmer when they come back to the code six months later.

## Multi-paradigm

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

(Discuss)

## High level datastructures built in

*So boring* to have to rewrite these again and again in C or whatever.

Arrays, dictionaries (maps or hashes), other sorts of collection.

(although proper sets not until 2.3 in 2003)

## Dictionaries all the way down

OK, that's not strictly true, but dictionaries are definitely a fundamental concept in Python, and many things either used to be dictionaries or act almost as if they are still dictionaries.

And the dictionaries are efficient enough that they're a good choice for many tasks.

## Python is malleable

Metaprogramming in Python was always easier than many other languages, and it has been improved and simplified over several decades, to make the common cases easier to do and (especially) easier to understand.

I really love the fact that Python is so malleable - you can get at and alter the behaviour of Python to a remarkable extent, using Python itself.

I also love that, mostly, Python programmers *don't do this* - we're a relatively conservative lot.

## Values or methods?

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

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

and later on realise that this actually needs to be a method, and change it without the user needing to change their own code:

```python
import random

class UsefulNumbers

    @property
    def random(self):
        return random.random()
```

I also love the fact that this is now much easier to do (using `@property`) than it used to be in earlier versions of Python.

## "Safe and sane"

Python programmers do not have a reputation for being wild and wacky in their code - I think this is a good thing!

## Batteries included

Really - compare with the other languages I was used to.

(although it should be said that this is true for Ruby as well)

## The community and a gentle sense of humour

Not, by any means, unique to Python.

As one example, when a new (and more sophisitcated) form of `import` was being tried out (in Python 1.3), the command to enable it was `import ni`. `ni` obviously stands for "new import", but was also a reference to "*the Knights who say Ni*" from the film "Monty Python and the Holy Grail".

There was also a long running joke about Guido's time machine, which enabled him to go back in time and implement a new language feature that people had just asked for (in real life, not realising that it had been there all along).

See An amble through some of Python's history for some other examples.

Also:

- https://github.com/DRMacIver/schroedinteger

    schroedinteger, from David MacIver (of course): "A schroedinteger behaves in as many ways as possible as if it were a real integer. However it's very indecisive and hasn't necessarily decided which integer it is.

    "You create it in a superposition of values. After that, every time you ask a question about its value, it determines a range of possible answers, picks one at random, and updates its knowledge about the range of values it could possibly have.""

- https://pyos.github.io/dg/

    dg: an alternative syntax for Python 3

    "Haskell's syntax but none of its type system"

- https://docs.hylang.org/

    not actually a joke, but a lisp syntax for Python, which can interoperate with "normal" Python

## Docstrings - these came later

Docstrings were introduced in Python 1.2 in 1995.

I remember they were first proposed (I think based on Emacs Lisp).

One of the lovely things about Python is that it was possible to play test them very easily, by just putting a string in the appropriate place (a string by itself just "sits there", so that's OK), and then writing some Python code to introspect the relevant module, class or method, and retrieve the string.

I personally much prefer this approach to that of the "magic comment", probably mostly because it feels natural that the docstring should be accessible via the AST (Abstract Syntax Tree, the parsed representation of the program).

## The Zen of Python

"The Zen of Python" is a joke by Tim Peters from 1999. It has been incorporated into the Python library as an easter egg:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This is a good joke because it is also true (well, mostly).

It can also be used as an interesting way to compare other programming languages to Python.

(I am way too much amused by how the above gets syntax highlighted, but it's also too big to fit on a slide, so the audience won't see that)

## Too much other stuff to go into

Like f strings, and __repr__ versus __str__, and numbers with underlines in them (makes my life a lot easier), and mypy typing, and all sorts of other things.

But I think they're smaller things than the above, in some difficult-to-measure sense.

Side note on the mpypy static typing work:

I like the fact that it is inline, and not consigned to other files (as, for instance, is the case with C++). The notation isn't perfect, but as with many things in Python, is a reasonable compromise between several conflicting factors.

# Part 2: Ruby

The Ruby Logo is Copyright (c) 2006, Yukihiro Matsumoto

Not everything I say may be exactly true, either because I have a misunderstanding about how Ruby works, or have misremembered soemthing, or because I'm oversimplifying for the purpose of this talk.

## Why did I learn Ruby?

It's not an obvious language to learn if you already know Python.

However, in July 2019, our team at work moved from working on a Python/Django project to working on projects written using Ruby/Rails.

## Caveat: I use Rails

I've learnt Ruby in the Rails context, so my views on the language itself may be as skewed as the views of a Python programmer who learnt the language to use Django. In particular, Rails likes "magic" even more than Django does.

Also, remember I've only been using Ruby for a short while, and have not seen its history "in action", whereas for Python I remember the evolution of the language and its environment.

## 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

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

The main influences are normally given as Smalltalk, Lisp and Perl.

*Not* Python. Matz knew Python well (I remember seeing him on `comp.lang.python` back in the day, and he obviously had a good knowledge of Python), so this is a conscious choice.

## Origins: Ruby's lisp features

In an email message back in 2006, Matz explained why Ruby has some lisp features

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. ;-)

("CL" is Common Lisp, and "CLOS" is the Common Lisp Object System)

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

Because I don't really like some of the stylistic choices - it errs a little too much on the magic side for me (this is *very* much a matter of choice!).

But there's a lot of stuff I *do* like, and more importantly, to me, I love the fact that Ruby takes some very different approaches than Python, in some cases producing what feels like much the same result (for instance, how values are defined) and in some cases shows paths that Python could not take, but that are still valuable approaches to explore (blocks, optional `()` on calling methods, and so on).

Note: this is meant to be a talk about the things I like in both languages, so don't expect me to look for things that I'm not keen on or think could be done better. There is no perfect programming language, and moreover different programming languages suit different programmers.

# Readability / writability

Python strongly errs toward being readable, even if that makes it slightly harder to write.

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 more likely to add synonyms for things - much less interested in "only one way". Instead, wants to give the predictable way (and thus easier to write). For instance:

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

as well as (the less colloquial)

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

# Begin and end and things

Ruby doesn't have significant indentation, but it does have decent sane block delineation (unlike, for instance, C-derived languages).

In particular, the *end* of a block is always indicated by `end`:

```ruby
begin
    ...
end
```

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

and so on.

(and yes, `elsif` takes a bit of getting used to for a Python programmer)

Indentation in Ruby is conventionally two spaces.

## Line continuation

```ruby
difference = minimum -
             maximum
```

If the punctuation on a line indicates an expression is not finished, it continues to the next line. I used to love this in BCPL.

And

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

I think this is a lot more readable than if the `.` separators/operators had to be at the end of each line.

I say on the slide "I don't think I need to say any more...", because I think Ruby has thought a lot about how to make this work well, and I don't think it hurts the appearance of the language at all.

What it *does* do is make the grammar more complex.

## Strongly object oriented, but easy to use...

I shall explain over the next few slides

## What do we mean by "Object Oriented"?

1. *Encapsulation* - the ability to syntactically hide the implementation of a type. E.g. in C or Pascal you always know whether something is a struct or an array, but in CLU and Java you can hide the difference.

2. *Protection* - the inability of the client of a type to detect its implementation. This guarantees that a behavior-preserving change to an implementation will not break its clients, and also makes sure that things like passwords don't leak out.

3. *Ad hoc polymorphism* - functions and data structures with parameters that can take on values of many different types.

4. *Parametric polymorphism* - functions and data structures that parameterize over arbitrary values (e.g. list of anything). ML and Lisp both have this. Java doesn't quite because of its non-Object types.

5. *Everything is an object* - all values are objects. True in Smalltalk (?) but not in Java (because of int and friends).

6. *All you can do is send a message* (AYCDISAM) = Actors model - there is no direct manipulation of objects, only communication with (or invocation of) them. The presence of fields in Java violates this.

7. *Specification inheritance* = subtyping - there are distinct types known to the language with the property that a value of one type is as good as a value of another for the purposes of type correctness. (E.g. Java interface inheritance.)

8. *Implementation inheritance/reuse* - having written one pile of code, a similar pile (e.g. a superset) can be generated in a controlled manner, i.e. the code doesn't have to be copied and edited. A limited and peculiar kind of abstraction. (E.g. Java class inheritance.)

9. *Sum-of-product-of-function pattern* - objects are (in effect) restricted to be functions that take as first argument a distinguished method key argument that is drawn from a finite set of simple names.

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

He has Java as {1,2,3,7,8,9}, and Lisp as {3,4,5,7}

Simula-67 was {1,3,7,9} and he says "many people take this as a definition of OO".

By my (quick and maybe wrong) reckoning,

- Python is {3,4,5,7,8,9}, while

- Ruby is {3,4,5,**6**,7,8,9}

... readers may be inerested in working this out for themselves.

Incidentally, while never formally part of the definition of OO, many people (particularly in the early years) would also include Garbage Collection.

The wikipedia page on Object-oriented programming regards Ruby as a "pure" OO language, whereas Python is designed mainly as OO, with some procedural elements.

## Ruby still feels like a multi-paradigm language

While everything is an object, and modules and classes are the only constructs to create objects, Ruby does actually allow you to write simple linear scripts (with no mention of `module` or `class`, or even the need to define a method).

And methods can (apparently) be declared at the top level.

So this is a perfectly good Ruby program:

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

I like that Ruby goes out of its way to make this possible, because it makes life better for the programmer.

(It's actually doing things with methods and classes and modules for you, but it's not making you do it yourself if you don't want to.)

## No self

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

Like many mainstream OO languages, it is not necessary to say `self` in almost all cases.

The Ruby Style Guide says "Avoid `self` where not required."

(The Ruby Style Guide is rather wonderful - I recommend it.)

Use of `@` to indicate equivalent of `self.` for values *inside* methods of the same class. But seems to be only when necessary, otherwise just use the accessor methods.

# 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.

- Python: assume an `a.x` is a value, but can add plumbing to make it be a method call.

- Ruby: `a.x` is always a setter/getter method call. *But* there's syntax to set that up with one line without needing to write methods unless you need to.

# Readonly values

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

```
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
```

To do this in Python, we'd need to use `@property`.

# 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
```

To do this in Python, we'd simply set the values as `self.width` and `self.height` in our `__init__` method.

## 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
```

Obviously this simple case doesn't need explicit methods (we should use the `attr` variants instead, as above).

In Python, we would again use `@property`.

## ? and ! at the end of method names

The Ruby Style Guide refers to these as "Predicate Methods Suffix" and "Dangerous Methods Suffix". "Surprising" might also be a good term instead of "Dangerous".

Methods ending with `?` should return a boolean, for instance:

```ruby
[].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 easier to check for
- `ActiveRecord::Base#save!` raises an exception

The second form suggests that we don't expect the "save" to fail.

The style guide also suggests that it's generally a good idea to implement the "safe" method (`sort`) as a wrapper around the "dangerous" or "surprising" method (so `sort` should presumably take a copy and then `sort!` it).

I rather like these - I think it's a fairly natural usage, and very readable.

The use of `?` and `!` at the end of a method name may be taken from Scheme, which uses `?` for predicates (`even?`) and `!` for mutating functions ()``set!``). Common Lisp, in contrast, uses a trailing `p` for predicates (so `evenp`).

We'll also see `=` at the end of method names in the section on object values and getters and setters.

# 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
```

As you might expect, symbols are "interned" - that is, there is only a single copy of each symbol.

Ruby uses symbols a lot, and is good at converting symbols to their string representation when necessary (`:symbol` becomes `symbol`)

So why doesn't Python have symbols, if they're so useful?

My suspicion is that they're a little bit hard to understand when you first come across them (I know I found them a bit hard to distinguish from the concept of strings), and so that didn't fit the idea of simplicity that (especially early) Python was striving for.

They're very much a part of lisps, though, so it was probably inevitable that Ruby would have such a useful thing.

On the whole, I like having symbols available. In Python we have to use a string in many places instead of a symbol, and then worry about guaranteeing that it is the same string. Also, Python doesn't guarantee to intern all strings (although nowadays I believe most constant strings are likely to be interned in CPython).

# 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 knows that message (in the normal OO manner).

The `send` method makes this explicit:

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

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

It can even be used to call a `private` method.

For instance, given:

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

it's not possible to do:

```
s = Something.new
s.reset
```

(Ruby will tell you you're trying to call a private method), but it *is* possible to do:

```
s = Something.new
s.send(:reset)
```

(Although The Ruby Style Guide does suggest you should think carefully about whether `public_send` would be better, as it honours the `private` visibility.)

One can ask if an object understands a message:

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

It's also quite easy to catch messages as they "go past" and decide what to do with them, using `method_missing` method:

```
class Example
  def method_missing(name, *args, &block)
    if name == :random
      puts "4"
    else
      puts "#{name}"
    end
  end
end
```

The `method_missing` method is documented as:

> A callback invoked by the interpreter if `respond_to?` is called and does not find a method.

Given the above:

```
irb(main):028:0> e = Example.new
=> #<Example:0x00007f807c975dc0>
irb(main):029:0> e.random
4
=> nil
irb(main):030:0> e.aha
aha
=> nil
irb(main**:031:0> e.whatever
whatever
=> nil
```

**Note** I've been naughty with this class, because I didn't define a `respond_to_missing?` method so that a caller could ask what messages the object *does* respond to. Because of that:

```
e.respond_to?(:random)   => false
```

which is misleading.

I do rather like the message passing idea, and the underlying support for it (even if Ruby doesn't make one talk that way all the time (there's still "calling a method").

I also rather like the `responds_to?` and `method_missing` mechanisms.

Finally, I appreciate the fact that The Ruby Style Guide suggests not using any of this in most cases - it does, however, explain why, and suggest alternatives.

**Note** I believe it *is* important to use a programming languages own terms for its concepts. In this case it shows up relative subtelties in the way the language is mean to work and be used. I've always had a particulare dislike for the sort of C or C++ programmer who insists on discussing Python method calling in C or C++ terms only, zeroing in on pointer management, and refusing to use Python's own terms, often citing "but that's what the low level implementation does" (perhaps true in CPython, perhaps not in other variants). There is normally a (good) reason for the terminology a programming language uses to talk about itself.

## Ruby and monkey patching

It certainly used to be that Ruby had a reputation in the Python world as glorying in (what Python people saw as) the over use of monkey patching - reaching back into a class definition and changing it at run time.

And there's *some* justice to this, except that "monkey patching" in Ruby isn't the same thing as in Python, because both the philosophy and the technology are different.

For a start, since Ruby thinks about sending messages to objects, it seems quite reasonable to intercept a message, either one that would normally not correspond to a method, or one that would be specified by the class or one of its super classes (or interfaces - I haven't mentioned interfaces before, but they're another thing that Ruby has - DO I NEED TO CHECK WHAT I MEAN HERE / EXPLAIN A BIT MORE?).

But secondly, the *mechanism* for monkey patching is not the same (DOUBLE CHECK THIS - I'M WRITING FROM MEMORY). In Ruby changing the behaviour of an object at run-time also inserts a "shim" layer around that object - the monkey patching is kept much more hygenic (and introspectable?) than in Python.

In some respects, Python can only monkey patch by doing very low level manipulations, wherea Ruby has proper support for it.

That doesn't mean one should go wild with this - it can still be a practical problem - but Ruby has definitely thought more about what should be possible to do cleanly here.

## 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

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

Basic monkey patching is very simple to do, quite nice to write, but rather too powerful for its own good.

For instance, we can "open" the String class and add a useful (missing) method:

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

and now `'abcd'.prefix_with_hat` will give us `'^abcd'`.

But imagine instead we decide to change an existing method:

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

As expected, `'abcd'.reverse` now gives us `'^abcd'` as well. But *all* usages of the `reverse` method are affected, including those where we didn't intend the effect - we've replaced the original method.

And yes, we could save the original definition of the method, and put it back again later (making sure we allow for exceptions and other unexpected flows of control), but that's all rather a pain.

## Refinements

Refinements give more control.

We can instead refine the `String` class inside a module:

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

and use that in a localised manner:

```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'
```

Which is actually rather nice.

## 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!

## Blocks 1: 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:

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

prints out:

```
1
2
```

Why is it that way round (`..` being inclusive and `...` being exclusive)?

Presumably because these operators (which also have more complicated / subtler uses than we've shown) are taken from Perl.

It may or may not be relevant that `1 .. 3` in Pascal is inclusive.

## 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.

It also shows the `begin ... rescue ... end` mechanism that is equivalent to Python's `try ... except`.

## 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])
```

or even pass `fn` as an argument to a callable, without needing to do anything special. On the other hand:

```
a = 3
def a(): print('A')
```

does not give us two different things called `a`

In Ruby, those are not the case, and doing the equivalent things takes a little more work (although only a little). And this has just about never arisen in my Ruby career so far - perhaps because a programming style that uses blocks leads to a different sort of code.

## Bare callables

(IS THERE A PROPER NAME FOR THIS?)

In Python:

```
callable
```

just "sits there" (well, except in the REPL, where it will report what it is)

You need to use the `()` (call) operators (!) to make something happen:

```
callable()
```

and to call with arguments you need to put those arguments inside the `()`:

```
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, because (is this a because? close enough for this talk) Ruby knows that a method is not a value, it is free to treat it differently. And that means, in partcular, that the `()` in a method call are optional.

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

So instead of:

```
method(1, 2, 3)
```

it's quite possible (and often colloquial) to do:

```
method 1 2 3
```

It is worth saying 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
```

(Of course, since they are Ruby code, they could become programs - there's good reason to not allow configuration files to be Turing complete - but in practice people don't seem to abuse this.)

# 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:

```ruby
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
```

You quickly stop seeing the `do` at the end of the introductory lines, but they are, of course, starting blocks, and `desribe`, `context` and `it` are actually methods.

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

```ruby
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
```

Notes:

1. `{ some: 'data' }` is the more colloquial way of writing the hash `{ 'some' => 'data' }`, as described in The Ruby Style Guide.

2. The ability to start lines like `.with(expense)` with the dot, instead of requiring it at the end of the preceding line, seems to me to make this much more readable.

3. `post` does what it sounds like it does

4. `last_response` is a method that returns the last response receive in the session.

## DSL Example 3: Sonic Pi

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

From their web page, IDM Breakbeat:

```ruby
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)}
```

By now, you should be able to see that this is Ruby code, but you don't need to know that to use Sonic Pi.

## 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.

(and, for what it's worth, I also found that Write the Docs conferences are lovely - nothing to do with Python or Ruby!)

## 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.

(Although Lobster, a statically typed language with a Python-like syntax, seems to be doing something interesting here.)

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.

# Addenda: More on Lisp-1 versus Lisp-2

- Lisp-1 vs Lisp-2 - a nice simple overview by hornbeck, 2009

- Technical Issues of Separation in Function Cells and Value Cells by Richard P. Gabriel and Kent M. Pitman, 2001, actually introduces the concepts, giving history and implications (this article is also available on Kent Pitman's site).

  This article uses subscripts for the numbers, $Lisp_1$ and $Lisp_2$, which I think is clearer as it doesn't look like language version numbers. As the articles says:

  - $Lisp_1$ has a single namespace that serves a dual role as the function namespace and value namespace; that is, its function namespace and value namespace are not distinct. In $Lisp_1$, the functional position of a form and the argument positions of forms are evaluated according to the same rules. Scheme and ... are $Lisp_1$ dialects.

  - $Lisp_2$ has distinct function and value namespaces. In $Lisp_2$, the rules for evaluation in the functional position of a form are distinct from those for evaluation in the argument positions of the form. Common Lisp is a $Lisp_2$ dialect.

- Xah Lee has a nice piece from 2008 explaining why not to use the terms Lisp-1 and Lisp-2 (the page starts with an overview of the terms), suggesting that:

  - "lisp-2" should be called multi-value-name languages.

  - "lisp-1" should be called single-value-name languages.

- Xah Lee also has an article Ruby Creator Matz: How Emacs changed my life - it's an annotated transcript of the slides from a talk by Matz.

# Some other links

- About Ruby at https://www.ruby-lang.org/

- Programming Ruby ("The Pick-axe Book", also available as a printed book)

- Why did Ruby creator chose to use the concept of Symbols?

- 23 years of Ruby (podcast interview with Matz from 2016, with a transcript)

- https://en.wikipedia.org/wiki/Ruby_(programming_language) quotes Matz from 1999:

  I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had the smell of a toy language (it still has). The object-oriented language seemed very promising. I knew Python then. But I didn't like it, because I didn't think it was a true object-oriented language – OO features appeared to be add-on to the language. As a language maniac and OO fan for 15 years, I really wanted a genuine object-oriented, easy-to-use scripting language. I looked for but couldn't find one. So I decided to make it.

- My own An amble through some of Python's history which also describes how I came to Python, references various examples of Python humour, and covers a lot of other stuff.