Why Crystal Is My Next Language

Posted on June 6, 2018

Note: I've made various corrections and amendments based on various feedback I have received. Thanks everyone! :)

I have been a heavy user and lover of Python since 2011. At that time, when a good friend suggested I ditch Perl (eeek) and try Python instead, an entirely new world opened up to me. A world where readability counted above everything else and an explicit style ruled.

After around 7 years of using Python, I'm just as passionate about it now as I was then. However, as time goes on, one looks for new adventures and challenges. The time has come for me to try another language!

Python Challenges

Let me start by noting some of the challenges I've faced with Python:

- **Packaging**: This is one area where most interpreted languages share challenges. Tools such as FPM (https://github.com/jordansissel/fpm) make it really easy to ship an installable artifact that includes an entire virtualenv, but this still lacks the elegance of a single binary.
- **Static Typing**: As someone who started with C++ and absolutely adored it, I do miss the type safety that I was used to from C++. This goes hand in hand with compile-time checks that really helped me ensure my code was of reasonable quality before even being executed.
- **Speed**: Once again, a challenge shared by most interpreted languages. Python is fast enough for many tasks, but falls far behind compiled languages.
- **Verbosity**: We only got f-strings in Python 3.6, which was truly a big relief. However, we still have extremely verbose `self` syntax in classes and constructors are littered with `self.var = var`, which may be partly addressed with data classes in Python 3.7 (https://www.python.org/dev/peps/pep-0557/).
- **Implicit Private Class Members**: When I say private, I mean private damn it! As a former C++ guy, I found Python's underscore-prefix style for private attributes and methods a little … hacky? :')

Further to this, I’m not sure that I really love the direction that Python is taking in a few areas, particularly around async and typing.

- **Coroutines**: Although highly welcome, the new async functionality in Python feels very user-hostile and difficult to grasp. Existing code requires a good amount of work before it is made non-blocking too. I think this situation will improve in time as more libraries become available and as I understand and use the new libraries more though.
- **Type Annotations (and mypy)**: Honestly, type annotations are welcome … if they actually did anything in CPython. The new idea of using type annotations as part of various constructs (e.g. data classes) seems pointless without mainstream support in the main CPython distribution. In the meantime, mypy is not mainstream just yet but shows great promise as a type validator for Python in the future, particularly with the `--strict` flag enabled.

I should note that I’m still a massive fan and advocate of Python and think it’s still one of the best interpreted languages available today; particularly when you take into account its wonderful ecosystem and maturity.

What I’m Looking For

My starting point is really Python and Ruby. I’ve used Ruby from time to time where it was needed and really love it too. Ruby solves several problems that Python has (proper private/protected attributes, less verbose syntax .etc) but still suffers from performance problems and lacks static typing.

As such, I started looking for a new language with the following features:

- Similar syntax to Python and Ruby
- Single-binary distribution
- Compiled, statically typed and fast
- Object-oriented (oh classes, how I love you…)

Candidates

The following languages were ruled out:

- **Go**: No keyword arguments, no exceptions, no classes, no generics and awful naming styles all led to me saying no to Go (although perhaps this simplicity is what attracts many to it). I have actually spent quite some time learning and coding in Go and found it frustrating at best. A language like C++ had made many

advancements after C and offered us far greater flexibility yet it feels as though Go is taking us back to the days of C.
- **Elixir**: A fascinating functional language, but lack of OO features and the fact that a single binary distribution is not the target of this language is a bit of a bummer for my use case. However, various folks in my team use Elixir as their primary language for all new projects and have found it great in use. Elixir has a rich and proven legacy and should definitely be considered if a functional language is what you're after.
- **Rust**: This is an interesting language that I have spent some time attempting to learn. Really, I just feel that Rust is not aimed at my use case; it is a rather complex language that just doesn't seem to click with me and many others too.
- **Julia**: This language is really targeted more for scientific computing as opposed my use case. It also lacks the OO abilities that I'm after.
- **Pony**: A very fascinating language that seems to borrow a lot from Python, however it also borrows some things I dislike (e.g. underscore prefixed variables, lack of symmetry .etc). I generally didn't feel that Pony aligned with the way I think nor did it have the same traction as other languages making it rather primitive currently.

Some languages that I'm really interested in and hope to examine further in the future are:

- **Nim**: Nim was originally the front-runner as my next language and one which I hope to spend more time on in the future.
- **Swift**: Another popular object-oriented language that definitely deserves attention beyond development of iOS and Mac apps.

But ultimately, I decided to commit to learning **Crystal**!

The reasons are as follows:

- Crystal feels immediately familiar as it mostly follows Ruby's syntax
- It compiles into a fast, single executable
- The entire standard library is written in Crystal which makes it very easy to read when required
- It offers a full object-oriented approach similar to Ruby (which includes real protected and private members)
- Crystal uses static typing but also provides unions (ability to define a variable that can be of multiple types)

- It offers the ability to develop DSLs similar to Ruby (which is something I've always been interested in)
- Bindings to C libraries are fully native and written in Crystal (similar to ctypes in Python, only better)

Caveats

Crystal is a very young language that still hasn't hit 1.0. It often introduces breaking changes in releases and has limited libraries.

However, I plan to only use this language in my personal projects and was willing to become an early adopter due to the fact I feel that the language has enough promise to be worth using.

Experiences

Standard Library

The entire standard library is extremely easy to read and is something I reference all the time. The library also seems moderately extensive and is a great base to work with.

Here's an example of the addition of arrays:

```
def +(other : Array(U)) forall U
  new_size = size + other.size
  Array(T | U).build(new_size) do |buffer|
    buffer.copy_from(@buffer, size)
    (buffer + size).copy_from(other.to_unsafe, other.size)
    new_size
  end
end
```

And here's the function that obtains the extension of a file:

```
  def self.extname(filename) : String
    filename.check_no_null_byte

    dot_index = filename.rindex('.')

    if (dot_index && dot_index != filename.size - 1 &&
        dot_index - 1 > (filename.rindex(SEPARATOR) || 0))
      filename[dot_index, filename.size - dot_index]
    else
      ""
    end
  end
```

If you choose to try out Crystal, ensure you keep its source right by your side; it’s incredibly valuable and useful.

Binding to C Libraries

It’s amazing how easy this is!

Here’s an example of a binding to various functions that obtain user information from a Unix system:

```
lib LibC
  struct Passwd
    pw_name   : LibC::Char*
    pw_passwd : LibC::Char*
    pw_uid    : LibC::UInt
    pw_gid    : LibC::UInt
    pw_change : LibC::Long
    pw_class  : LibC::Char*
    pw_gecos  : LibC::Char*
    pw_dir    : LibC::Char*
    pw_shell  : LibC::Char*
    pw_expire : LibC::Long
  end

  fun getpwuid(uid : LibC::UInt) : Passwd*
  fun getpwnam(name : LibC::Char*) : Passwd*
  fun getpwent : Passwd*
  fun setpwent
  fun endpwent
end
```

Exception Handling

Similar exception handling is provided to both Ruby and Python:

```
    def self.mount?(path)
      begin
        stat_path = File.lstat(path)
      rescue Errno
        # It doesn't exist so not a mount point
        return false
      end
      ...
    end
```

Writing your own exceptions is trivial; simply inherit from the `Exception` class.

Import System & Namespaces

This was a bit of an adjustment coming from Python, but really brought me back to C++ days as Ruby follows a similar method to C++.

C++ namespaces are equivalent to Ruby/Crystal modules which you can define yourself. Requiring any library will import all items that it defines, so it's always ideal to ensure that your entire library is contained within a module to avoid namespace pollution.

Initially I was a bit concerned about this, but I find it liberating being able to easily build up a module from any number of files. However, I will admit that it makes finding where things came from more of a challenge.

```
require "yaml"

# In this example, the yaml library provides a module called YAML
# which contains a function named parse which we are calling below
data = YAML.parse(File.read("./foo.yml"))
puts data
```

Classes

One of my very favourite things about Crystal is how it handles assignment of instance variables:

```
class Person
  def initialize(@name : String, @age : Int = 0)
  end
  ...
end
```

This creates a constructor that will automatically assign the provided parameters to instance variables. The equivalent code in Python would be:

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.age = age
```

Although it's a personal thing, I also really like the symmetry of the `end` statements and the two space indentation in Ruby/Crystal. I feel that it ultimately makes the code more beautiful and elegant to read.

And of course, we have proper protected and private members and abstract classes too; both features I missed from my C++ days.

Documentation

I absolutely love Crystal's documentation. It is so inviting and enjoyable to read. However, as with any new language, it is possibly not as comprehensive as it could be.

The main two pieces of documentation provided are:

- Crystal Docs (https://crystal-lang.org/docs/): Offers a very enjoyable walkthrough of most features offered by the language. Be sure to hit the little **A** icon on the top of the screen to adjust your font, font size and theme (nice touch). I recommend starting here.
- Crystal API Reference (https://crystal-lang.org/api/): Details all modules offered and their respective classes and functions.

Another incredibly valuable resource is the Crystal chatroom on Gitter (https://gitter.im/crystal-lang/crystal). Everyone in the channel is very welcoming and helpful. They have been a great source of information for me on my journey thus far.

Performance

Although it's too early for me to really determine performance gains, it's always fun to do a Fibonacci test :)

Ruby / Crystal

```
def fib(n)
  if n <= 1
    1
  else
    fib(n - 1) + fib(n - 2)
  end
end

puts fib(42)
```

Python

```python
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

print(fib(42))
```

C

```c
#include <stdio.h>

int fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

int main()
{
    printf("%d\n", fib(42));
    return 0;
}
```

Compiled with `-O3` for best performance.

C++

```cpp
#include <iostream>

using namespace std;

int fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

int main()
{
    cout << fib(42) << endl;
    return 0;
}
```

Compiled with `-O3` for best performance.

Go

```go
package main

import "fmt"

func fib(n int) int {
    if n <= 1 {
        return 1
    } else {
        return fib(n - 1) + fib(n - 2)
    }
}

func main() {
    fmt.Println(fib(42))
}
```

Results

```
Runtime          Time (sec)
---------------- ----------
C 4.2.1               0.747
Crystal 0.24.2        0.751
C++ 4.2.1             0.930
Go 1.10.2             1.615
PyPy3 6.0.0          12.578
Ruby 2.5.1           37.944
CPython 3.6.5       128.172
```

Conclusion

Although it’s early days for both me and the language itself, I’m very optimistic and hopeful that Crystal will soon be the choice for many in production. I think that the language will be a natural progression for Python and Ruby users alike.

Be on the lookout for more posts about Crystal in the near future, including tips and tricks that I come across.