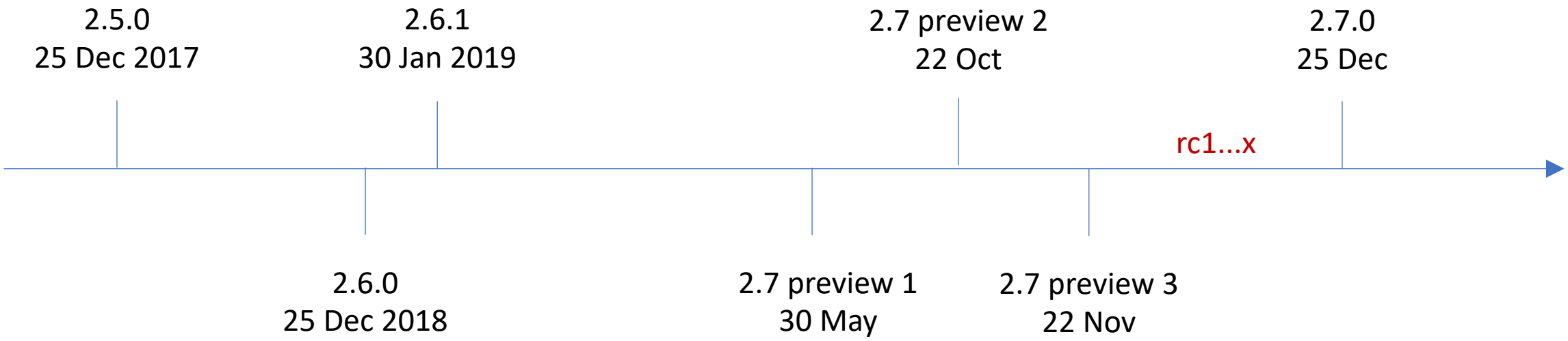
# A Quick Look at Ruby 2.7

Melbourne Ruby Meetup

# Ruby release timeline

- If you're not aware, Ruby minor releases have been consistently released on the 25th of December
- Each release begins with a series of previews, followed by RCs
- Patches continue to be released for previous minor versions



# Key features and improvements

- Compaction GC
  - GC.compact to reduce memory fragmentation
- REPL improvements
- Beginless range (experimental)
- JIT improvements
- ~~Method reference operator (.:)~~
- ~~Pipeline operator~~
- **Pattern matching (experimental)**
- **Separation of positional and keyword arguments**

# Code for this talk

- All of the snippets shown in this talk have been published on Github
- You can find the code at:
  - https://github.com/tristanpenman/ruby-2.7-examples
- Currently tested against ruby-2.7-preview3

# Pattern matching

# Pattern matching

- In a nutshell, pattern matching provides another way to control the flow of execution in a program and to destructure (aka. capture) input
  - You may have seen it in other languages such as Elixir, Rust, or even Haskell
- It is often implemented as a way to choose which variant of a function should be used, by attempting to match input against a series of patterns
  - e.g here is the factorial function, as it might be defined in Haskell vs Ruby:

```
factorial 0 = 1                          -- Patterns (on the left) will be tested, until a match
factorial n = n * factorial(n - 1)       -- is found. Then the code on the right will be executed.
```

```
def factorial(n)
  n.zero? ? 1 : n * factorial(n - 1)
end
```

# Pattern matching

pattern-matching/00-arrays.rb

- How could pattern matching work in Ruby?
  - Well, instead of selecting a variation of a function, Ruby's pattern matching is implemented as an extension to 'case' expressions
  - Allows for destructuring/capturing of input, partial matches, etc
- Here is a simple example that matches an array containing three elements, and *captures* their values as local variables:

```
def match_array(input)
  case input
  in [a, b, c]                                  # <-- pattern matching using the 'in' keyword
    "matching - #{a}, #{b}, #{c}"
  else
    "no match"
  end
end
```

# Matching an array with values

pattern-matching/01-array-with-values.rb

- We can also match an array where some (or all) elements have a fixed value:

```
def match_array_with_values(input)
  case input
  in [1, b, 3]
    puts "match - #{input}, second element is #{b}"
  else
    puts "no matches"
  end
end

# This will match
puts match_array_with_values([1, 2, 3])

# This will not match
puts match_array_with_values([3, 2, 1])

# This will also not match, because it contains four elements
puts match_array_with_values([1, 2, 3, 4])
```

# Matching a hash

pattern-matching/02-hashes.rb

- You can also pattern match with hashes
- Here's a more fleshed out example that shows how you can also pattern match hash values, with some values fixed:

```
def match_hash(input)
  case input
  in {op: :add, left: left, right: right}
    left + right
  in {op: :sub, left: left, right: right}
    left - right
  in {op: :abs, val: val}
    val.abs
  in {op: op}
    raise ArgumentError, "Unknown op: #{op}"
  end
end
```

```
# Matched
puts match_hash(op: :add, left: 1, right: 2)

# Notice that extra keys are ignored
puts match_hash(op: :abs, val: -1, extra: 2)

# Unmatched op
begin
  puts match_hash(op: :div, left: 1, right: 2)
rescue => e
  puts "Error: #{e}"
end
```

# Pinning variables: motivation

pattern-matching/03-pinning-motivation.rb

- To understand the importance of pinning, we'll start with an example, where we attempt to capture a repeated value (without pinning):

```
def match_without_pinning(input)
  case input
  in [a, a]   # We'd like to assert that the first and second elements are equal
    a
  else
    '_'
  end
end

# The fact that we see none of this output suggests that this doesn't even parse as valid Ruby
puts 'begin'
puts match_without_pinning([1, 1])
puts match_without_pinning([1, 2])
puts 'end'
```

# Pinning variables: solution

pattern-matching/04-pinning-solution.rb

- Instead, we have to prefix a repeated variable with a hat symbol (^) to tell Ruby that it should remain unchanged:

```
def with_pinning(input)
  case input
  in [ a, ^a ]   # We'd like to assert that the first and second elements are equal
    a
  else
    '_'
  end
end

# Output is '-'
puts with_pinning [ 0, 1 ]
puts with_pinning [ 1, 0 ]

# Output is '1'
puts with_pinning [ 1, 1 ]
```

# Mixing 'when' and 'in'

pattern-matching/05-mixing-when-and-in.rb

- It might seem like you can mix 'when' and 'in'... you can't

```
def match_mixed(test)
  case test
  when 'hello'
    'hello'
  in [a, b]
    "a: #{a}, b: #{b}"
  end
end

puts match_mixed('hello')
```

```
$ ruby 05-mixing-when-and-in.rb
05-mixing-when-and-in.rb:9: syntax error, unexpected `in', expecting `end'
  in [a, b]
05-mixing-when-and-in.rb:12: syntax error, unexpected `end', expecting end-of-input
```

# Matching multiple patterns

pattern-matching/06-multiple-patterns.rb

- We can match more than one pattern using a single 'in'

```
def match_multiple(input)
  case input
  in 0 | 1
    true
  else
    false
  end
end

# True
puts match_multiple(0)
puts match_multiple(1)

# False
puts match_multiple(2)
```

# Guard clauses

pattern-matching/07-guard-clauses.rb

- Allows patterns to be matched conditionally, using captured variables:

```
def match_with_guards(input)
  case input
  in {op: :abs, val: val} if val.positive?
    val
  in {op: :abs, val: val} if val.negative?
    -val
  end
end

# Matched
puts match_with_guards(op: :abs, val: 1)
puts match_with_guards(op: :abs, val: -1)

# Unmatched
puts match_with_guards(op: :abs, val: 0)
```

# No matching patterns

pattern-matching/08-no-matches.rb

- What happens when there are no matches?

```
def match(input)
  case input
  in 'ping'
    'pong'
  end
end

# Matched
puts match('ping')

begin
  # Does not return nil, as we might have expected; instead we find that
  # NoMatchingPatternError is raised
  puts match('pong')
rescue NoMatchingPatternError => e
  puts "Error: #{e}"
end
```

# Nesting and splats

pattern-matching/09-nesting-and-splats.rb

- A more advanced example that shows nesting and splat operators:

```
def match_nested(input)
  case input
  in [a, {b: [c, d, *e]}]
    "matched #{a}, #{c}, #{d}, #{e}"
  else
    'no match'
  end
end

# Match with array splat
puts match_nested([1, {b: [2, 3]}])

# Match with object splat
puts match_nested([1, {b: [2, 3, 4, 5]}])

# No match
puts match_nested([1, {c: [2, 3]}])
```

# Separation of positional and keyword arguments

# Separation of positional and keyword arguments

- Automatic conversion of *keyword arguments* and *positional arguments* is deprecated, and conversion will be removed in Ruby 3
- That's a bit of a mouthful – how will this change impact us?
- To understand all of this, we need to review Ruby argument handling...

# Background: Keyword arguments

- Ruby 2.0 gave us keyword arguments, with default values
  - This can aid readability, by allowing arguments to be re-ordered:

```
def purchase_items(item: nil, qty: 1)
  puts "purchasing #{qty} unit(s) of #{item} (but not really)"
end

# These are all equivalent
purchase_items(item: 'socks', qty: 1)
purchase_items(qty: 1, item: 'socks')
purchase_items(item: 'socks')

# And before Ruby 2.7, automatic conversion from a hash is also equivalent
purchase_items({item: 'socks', qty: 1})

# This works, but it is probably not what we want, and requires custom validation
purchase_items(qty: 3)
```

# Background: Required keyword arguments

- Not long after, *required* keyword arguments were introduced, as part of Ruby 2.1
  - Simply omit the default value
  - But language ergonomics dictated that assignment of required keyword arguments should be done automatically

```
def purchase_items(item:, qty: 1)
  puts "purchasing #{qty} unit(s) of #{item} (but not really)"
end

# These are all equivalent
purchase_items(item: 'socks', qty: 1)
purchase_items(qty: 1, item: 'socks')
purchase_items(item: 'socks')

# Not so happy... raises ArgumentError
purchase_items(qty: 1)
```

# Background: Ruby argument handling

- Putting this together, we see that Ruby methods support many different kinds of arguments:
  - Required positional arguments
  - Optional positional arguments
  - Variable positional arguments
  - Required keyword arguments
  - Optional keyword arguments
  - Arbitrary keyword arguments
  - Blocks

# Background: Ruby argument handling

- This example, adapted from from Marc-André Lafortune's blog on Ruby 2.0, shows how these can all be combined with a single method:
  - http://blog.marc-andre.ca/2013/02/23/ruby-2-by-example/

```ruby
class C
  def hi(needed,
         maybe = "42",
         *args,
         named1: 'hello',
         named2:,
         **options,
         &block)
  end
end

C.instance_method(:hi).parameters
```

```ruby
# => [ [:req, :needed],
#      [:opt, :maybe],
#      [:rest, :args],
#      [:keyreq, :named2],
#      [:key, :named],
#      [:keyrest, :options],
#      [:block, :block] ]
```

# Background: Ruby argument handling

- The problem is that Ruby is allowed to convert things that look like arbitrary keyword arguments into a positional argument
- At the same time, it is allowed to convert something that looks like a variable positional argument into keyword arguments
- There are very specific scenarios where this is problematic (and confusing), so Ruby 3 aims to better separate positional and keyword arguments
  - This change is the first step in that direction
- The Ruby 2.7 release notes go into much more detail about exactly what these scenarios are, but I'm going to illustrate two, to help explain the motivation for this change

# Separation of positional and keyword arguments

- Ruby 3 aims to simplify argument handling
  - Keyword arguments will be treated separately from positional arguments
  - Hence, automatic conversion is being deprecated

```
def do_the_thing_1(*pos, key: 1)
  puts "pos: #{pos}, key: #{key}"
end

# keyword argument provided unambiguously
do_the_thing_1(key: 2)

# Should the hash go into 'pos' or be treated as a keyword argument?
do_the_thing_1({key: 2})

# Say what you mean...
do_the_thing_1(**{key: 2})
```

# Separation of positional and keyword arguments

- These are the kinds of warnings we'll get when we run that code in Ruby 2.7:

```
2.7.0-preview3 :071 > do_the_thing_1(key: 2)
pos: [], key: 2
 => nil

2.7.0-preview3 :072 > do_the_thing_1({key: 2})
(irb):72: warning: The last argument is used as the keyword parameter
(irb):58: warning: for `do_the_thing_1' defined here
pos: [], key: 2
 => nil

2.7.0-preview3 :073 > do_the_thing_1(**{key: 2})
pos: []
key: 2
 => nil
```

- This is basically warning us that something that *looks* like a positional argument has instead been used to provide keyword arguments

# Separation of positional and keyword arguments

- Can it go the other way? i.e. Can something that looks like a keyword argument be used as a positional argument?
  - Spoiler: it can

```
def do_the_thing_2(pos, key: 1)
  puts "pos: #{pos}, key: #{key}"
end

# Looks like keyword argument is provided, but 'pos' needs to be assigned, and this
# can be done by converting the keyword arguments to a hash
do_the_thing_2(key: 2)

# Looks like a positional argument, 'pos' is assignable, keyword argument will be
# assigned it's default value
do_the_thing_2({key: 2})
```

# Separation of positional and keyword arguments

- Similar to the previous example, we're warned that something that looks like keyword argument is being converted into a hash, and being used to satisfy the last positional argument:

```
2.7.0-preview3 :080 > do_the_thing_2(key: 2)
(irb):79: warning: The keyword argument is passed as the last hash parameter
(irb):74: warning: for `do_the_thing_2' defined here
pos: {:key=>2}, key: 1
 => nil

2.7.0-preview3 :081 > do_the_thing_2({key: 2})
pos: {:key=>2}, key: 1
 => nil
```

# Conclusions

- To me, Ruby 2.7 feels like a nice step forward
  - Paves the way for removal of tech debt in Ruby 3
  - Adds nice language features
  - Less visible improvements like GC and JIT can benefit everyone

# Conclusions

- Pattern matching is a powerful language feature that I really *want* to use in production code as soon as it is considered stable
  - I started using Ruby about ten years ago, and like many, found it so much more joyful than other languages
  - Playing around with Pattern Matching over the past few weeks has brought back some of that same joy!

# Resources

- General
  - https://www.ruby-lang.org/en/news/2019/11/23/ruby-2-7-0-preview3-released/
- Pattern matching
  - https://medium.com/@baweaver/ruby-2-7-pattern-matching-first-impressions-cdb93c6246e6
- Separation of positional and keyword arguments
  - https://thoughtbot.com/blog/ruby-2-keyword-arguments
  - https://blog.saeloun.com/2019/10/07/ruby-2-7-keyword-arguments-redesign.html
- Compaction GC
  - https://stackify.com/how-does-ruby-garbage-collection-work-a-simple-tutorial/
  - https://youtu.be/H8iWLoarTZc

# Questions