Build a mini Ruby debugger in under 300 lines

Stan Lo

Build a mini Ruby debugger in under ~~300~~ 200 lines

Stan Lo

shopify

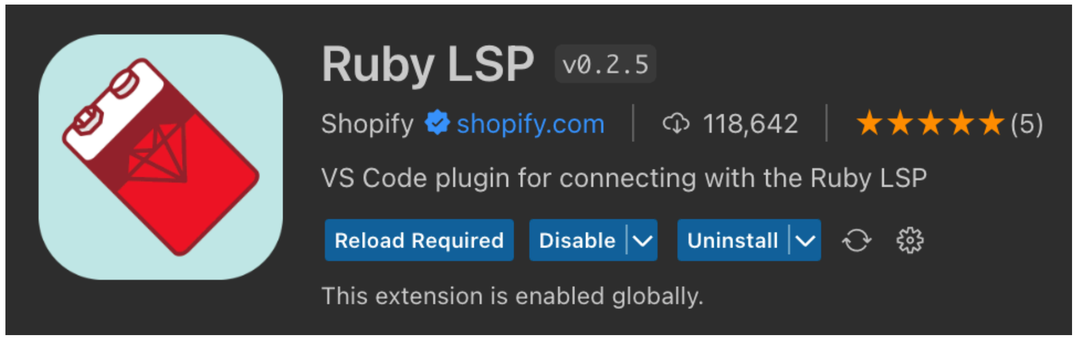
About me

About me

- 🇹🇼 → 🇬🇧

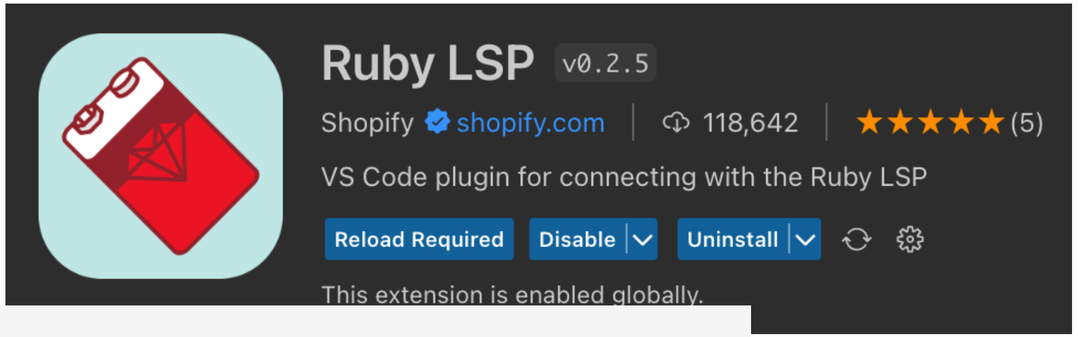
About me

- 🇹🇼 → 🇬🇧
- Ruby Developer Experience Team



About me

- 🇹🇼 → 🇬🇧
- Ruby Developer Experience Team 

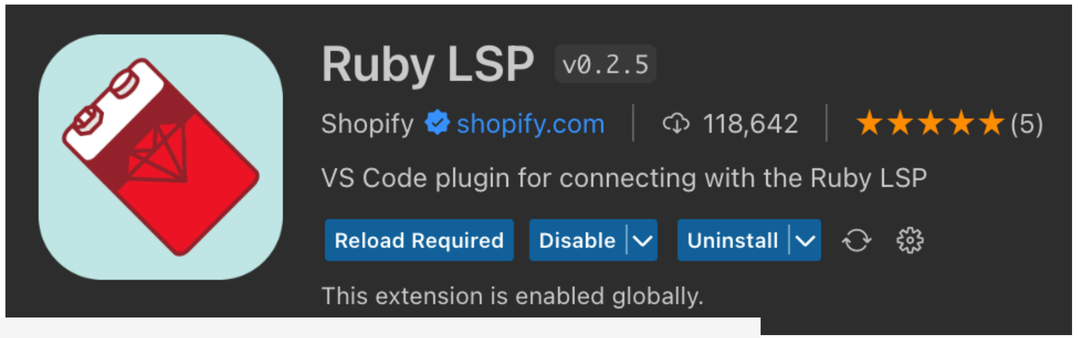


EN

Code indexing: How language servers understand our code

About me

- 🇹🇼 → 🇬🇧
- Ruby Developer Experience Team 

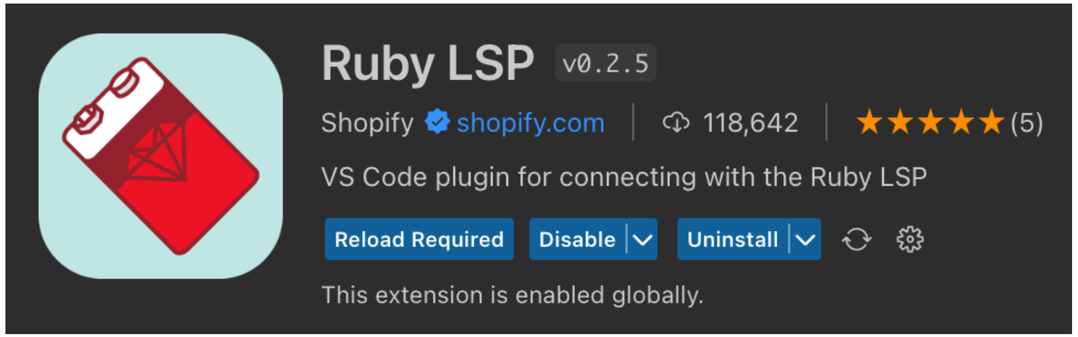


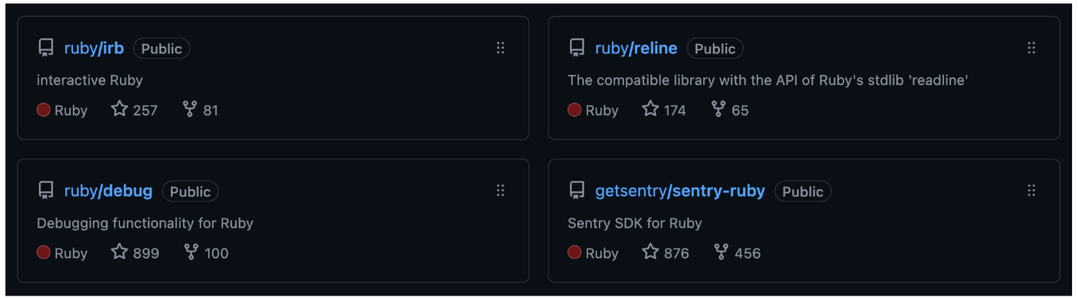
EN

Gradual typing for Ruby: comparing RBS and RBI/Sorbet

About me

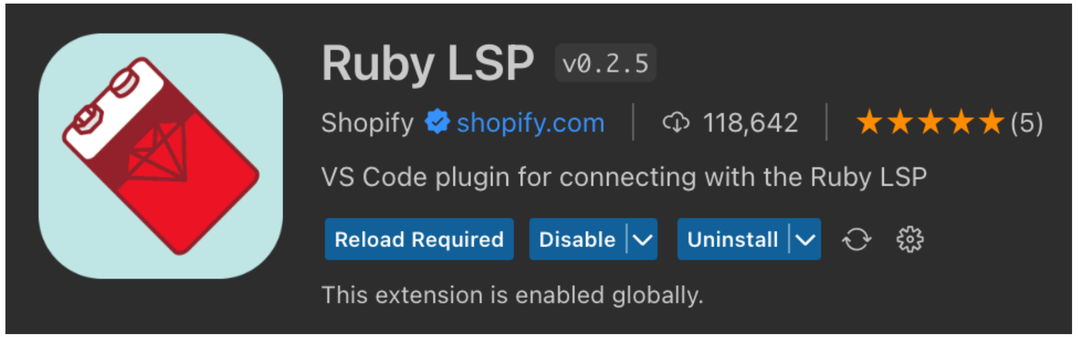
- 🇹🇼 → 🇬🇧
- Ruby Developer Experience Team
- Maintainer of **IRB**, **Reline**, and **sentry-ruby**

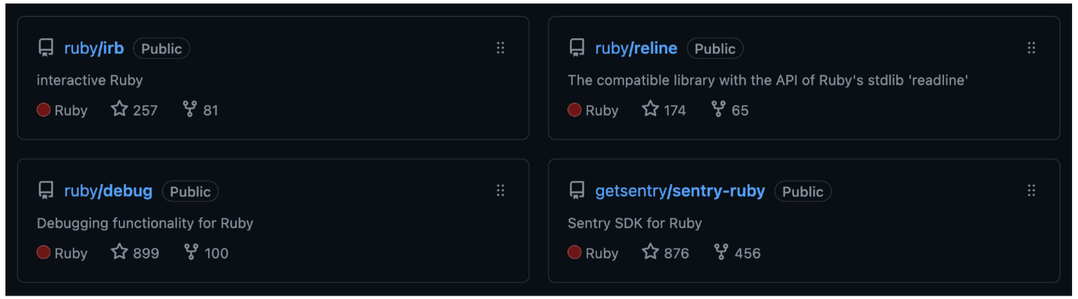




About me

- 🇹🇼 → 🇬🇧
- Ruby Developer Experience Team
- Maintainer of **IRB**, **Reline**, and **sentry-ruby**
- Links:
  - GitHub: @st0012 Twitter: @_st0012
  - ruby.social: @st0012
  - https://st0012.dev





ruby/debug

The best investment for your productivity

2022-09-09

shopify

Why?

Why?

- Familiarise yourself with debugging tools

Why?

- Familiarise yourself with debugging tools
- Grasp debugger implementation for informed usage decisions

Why?

- Familiarise yourself with debugging tools
- Grasp debugger implementation for informed usage decisions
- Learn to build custom debugging tools

Overview

- 5 steps with
  - Demo
  - Implementation overview
- A complementary GitHub repository

0. Key components

- **Binding** objects and **Kernel#binding**
- **TracePoint** class
- **Reline** library

Binding & Kernel.binding

- Objects of class **Binding** encapsulate the execution context at some particular place in the code and retain this context for future use.

- If you get a **Binding** object, likely through **Kernel#binding**, you can access the context it captures.

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

```ruby
class Foo
  def initialize
    @name = "foo"
  end

  def get_binding(x)
    binding
  end
end

puts binding.eval("self") #=> main

b = Foo.new.get_binding("bar")
puts b.eval("self") #=> #<Foo:0x00007f9b1a0b0e60>
puts b.eval("@name") #=> "foo"
puts b.eval("x") #=> "bar"
puts b.source_location.to_s #=> ["examples/binding.rb", 7]
```

TracePoint & line event

- **TracePoint** allows you to trace different type of Ruby execution events.
  - E.g. **:raise**, **:call**, **:line**...etc.
- The **:line** event is triggered by line executions, which has access to:
  - Line number
  - File path
  - **Binding**

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```
examples/trace_point.rb:9 is being executed. Locals: []
examples/trace_point.rb:2 is being executed. Locals: [:word]
Hello RubyKaigi!
```

```ruby
def greeting(word)
  puts "Hello #{word}!"
end

TracePoint.trace(:line) do |tp|
  puts "#{tp.path}:#{tp.lineno} is being executed. Locals: #{tp.binding.local_variables}"
end

greeting("RubyKaigi")
```

```
examples/trace_point.rb:9 is being executed. Locals: []
examples/trace_point.rb:2 is being executed. Locals: [:word]
Hello RubyKaigi!
```

TracePoint

- Convenient for debugging, but bad for performance
- Most TracePoint events cancel out YJIT's optimisation
- **Don't use it in production**

Reline

- Powers essential Ruby tools: **IRB** and **ruby/debug**
- Feature-rich with capabilities like:
  - Autocompletion
  - Multi-line input
  - Input history
- (Note: We won't be utilising these advanced features in this talk)

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Reline - Example

```ruby
require "reline"

puts 'This is echo program by Reline.'

while line = Reline.readline("echo> ")
  case line.chomp
  when 'exit'
    exit 0
  else
    puts "=> #{line}"
  end
end
```

Steps

1. Breakpoint and REPL (**binding.debug**)
2. Step-in (**step**)
3. Step-over (**next**)
4. Breakpoint commands (**break** and **delete**)
5. Debugger executable (**exe/debug**)

```ruby
def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

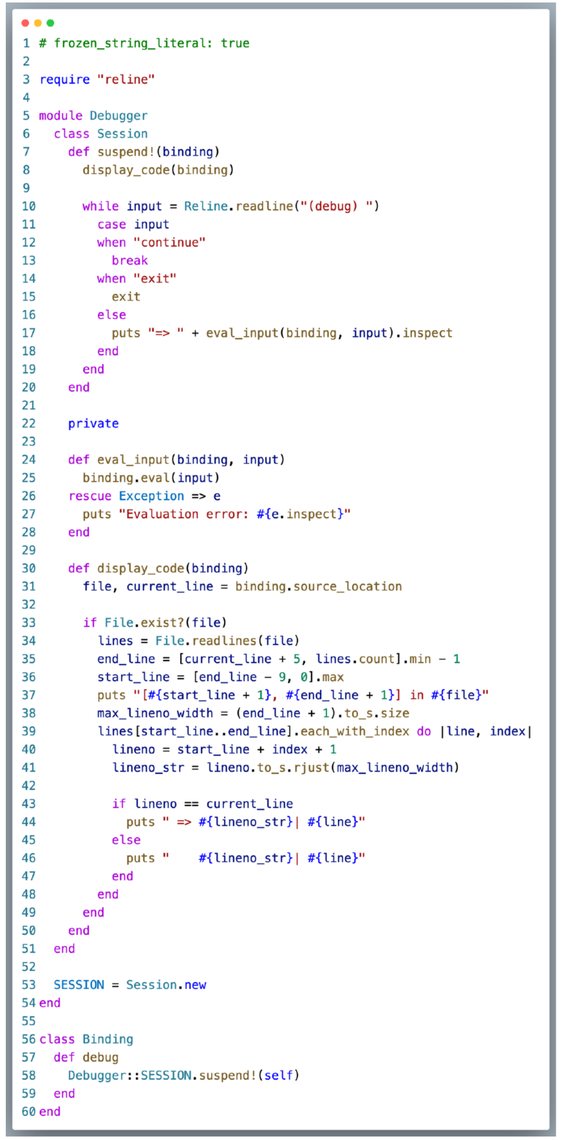
1. Breakpoint and REPL

- Stop program execution with **binding.debug**
- Open a REPL (Read-eval-print loop)
- Supports commands to **continue** or **exit** the program

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    binding.debug
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

```
> b
```

```
> b
```



```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```


```ruby
  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

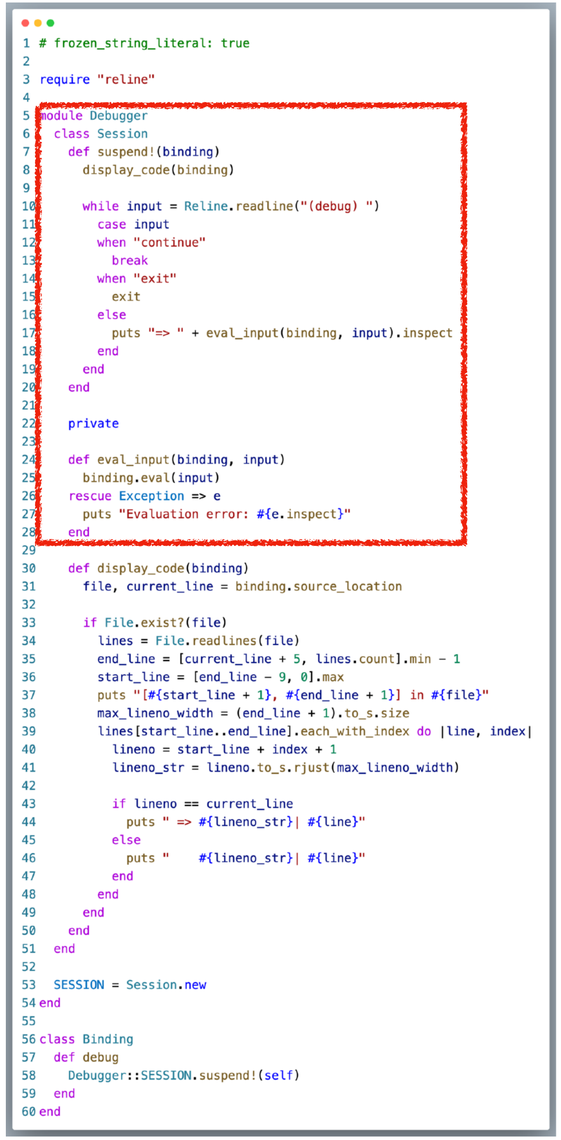
```ruby
module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end
```



```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
def display_code(binding)
  file, current_line = binding.source_location

  if File.exist?(file)
    lines = File.readlines(file)
    end_line = [current_line + 5, lines.count].min - 1
    start_line = [end_line - 9, 0].max
    puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
    max_lineno_width = (end_line + 1).to_s.size
    lines[start_line..end_line].each_with_index do |line, index|
      lineno = start_line + index + 1
      lineno_str = lineno.to_s.rjust(max_lineno_width)

      if lineno == current_line
        puts " => #{lineno_str}| #{line}"
      else
        puts "    #{lineno_str}| #{line}"
      end
    end
  end
end
```

```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

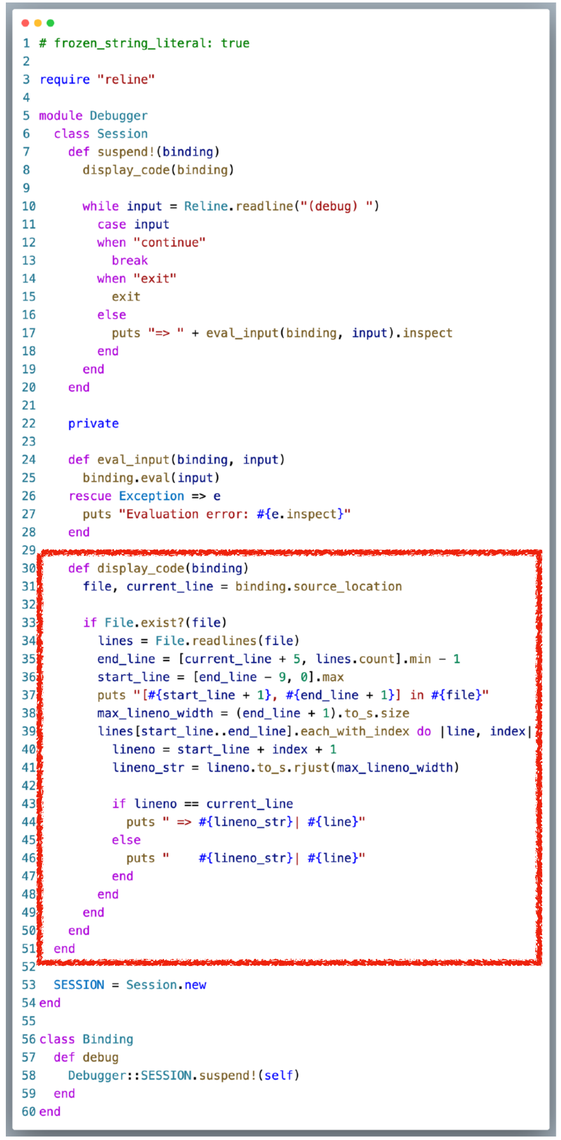
```ruby
def display_code(binding)
  file, current_line = binding.source_location

  if File.exist?(file)
    lines = File.readlines(file)
    end_line = [current_line + 5, lines.count].min - 1
    start_line = [end_line - 9, 0].max
    puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
    max_lineno_width = (end_line + 1).to_s.size
    lines[start_line..end_line].each_with_index do |line, index|
      lineno = start_line + index + 1
      lineno_str = lineno.to_s.rjust(max_lineno_width)

      if lineno == current_line
        puts " => #{lineno_str}| #{line}"
      else
        puts "    #{lineno_str}| #{line}"
      end
    end
  end
end
```

```
[3, 12] in app.rb
     3| def fib(num)
     4|   if num < 2
     5|     num
     6|   else
 =>  7|     binding.debug
     8|     fib(num-1) + fib(num-2)
     9|   end
    10| end
    11|
    12| a = fib(6)
(debug)
```

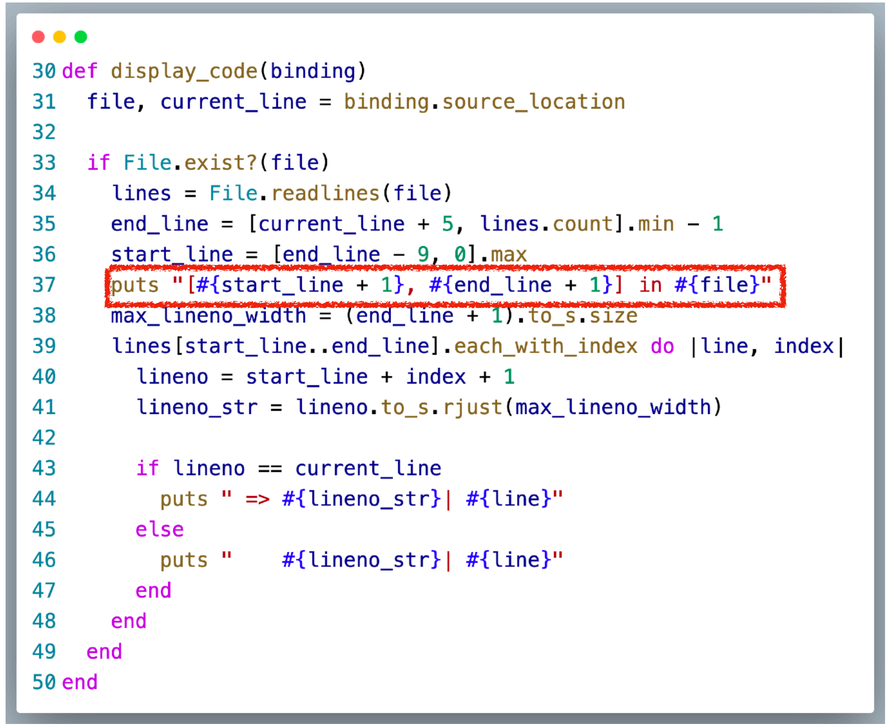
```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
def display_code(binding)
  file, current_line = binding.source_location

  if File.exist?(file)
    lines = File.readlines(file)
    end_line = [current_line + 5, lines.count].min - 1
    start_line = [end_line - 9, 0].max
    puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
    max_lineno_width = (end_line + 1).to_s.size
    lines[start_line..end_line].each_with_index do |line, index|
      lineno = start_line + index + 1
      lineno_str = lineno.to_s.rjust(max_lineno_width)

      if lineno == current_line
        puts " => #{lineno_str}| #{line}"
      else
        puts "    #{lineno_str}| #{line}"
      end
    end
  end
end
```

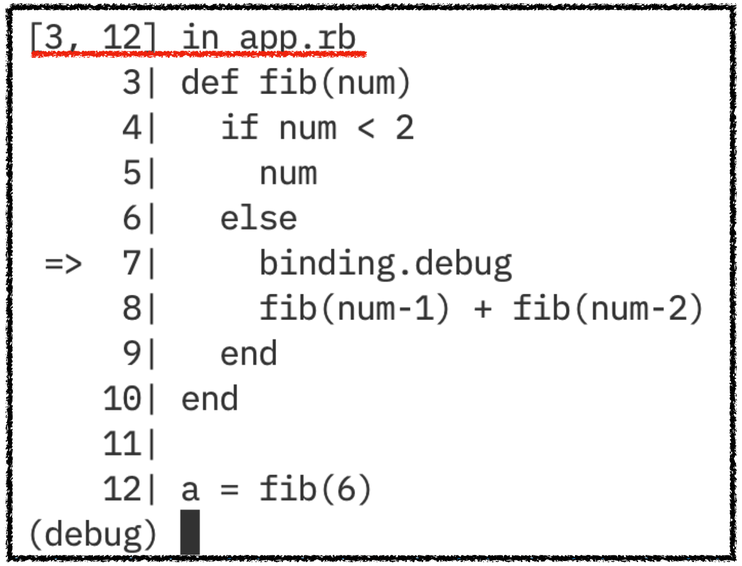
```
[3, 12] in app.rb
     3| def fib(num)
     4|   if num < 2
     5|     num
     6|   else
 =>  7|     binding.debug
     8|     fib(num-1) + fib(num-2)
     9|   end
    10| end
    11|
    12| a = fib(6)
(debug)
```



```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```



```ruby
def display_code(binding)
  file, current_line = binding.source_location

  if File.exist?(file)
    lines = File.readlines(file)
    end_line = [current_line + 5, lines.count].min - 1
    start_line = [end_line - 9, 0].max
    puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
    max_lineno_width = (end_line + 1).to_s.size
    lines[start_line..end_line].each_with_index do |line, index|
      lineno = start_line + index + 1
      lineno_str = lineno.to_s.rjust(max_lineno_width)

      if lineno == current_line
        puts " => #{lineno_str}| #{line}"
      else
        puts "    #{lineno_str}| #{line}"
      end
    end
  end
end
```



```
[3, 12] in app.rb
     3| def fib(num)
     4|   if num < 2
     5|     num
     6|   else
 =>  7|     binding.debug
     8|     fib(num-1) + fib(num-2)
     9|   end
    10| end
    11|
    12| a = fib(6)
(debug)
```

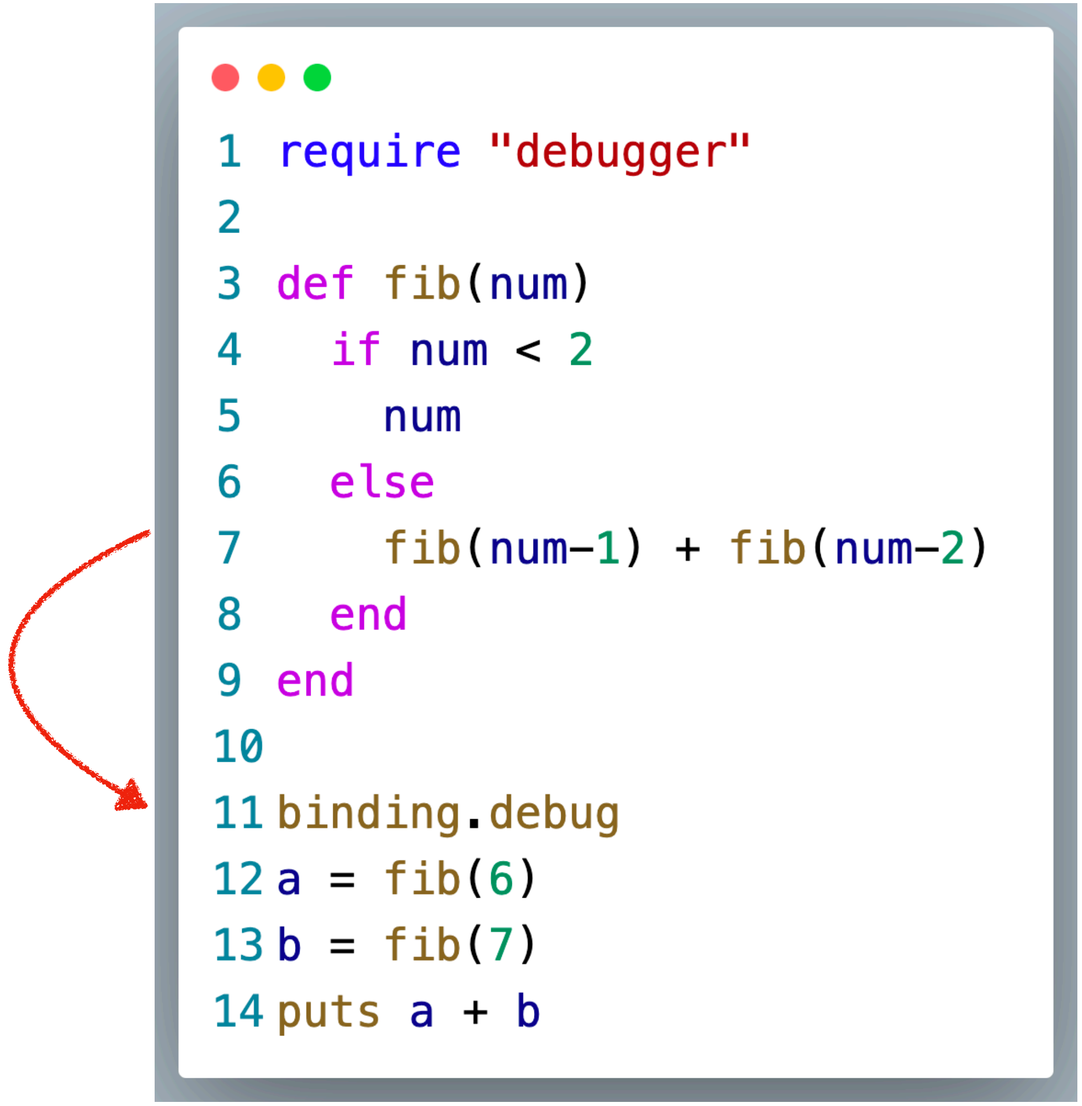
```ruby
# frozen_string_literal: true

require "reline"

module Debugger
  class Session
    def suspend!(binding)
      display_code(binding)

      while input = Reline.readline("(debug) ")
        case input
        when "continue"
          break
        when "exit"
          exit
        else
          puts "=> " + eval_input(binding, input).inspect
        end
      end
    end

    private

    def eval_input(binding, input)
      binding.eval(input)
    rescue Exception => e
      puts "Evaluation error: #{e.inspect}"
    end

    def display_code(binding)
      file, current_line = binding.source_location

      if File.exist?(file)
        lines = File.readlines(file)
        end_line = [current_line + 5, lines.count].min - 1
        start_line = [end_line - 9, 0].max
        puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
        max_lineno_width = (end_line + 1).to_s.size
        lines[start_line..end_line].each_with_index do |line, index|
          lineno = start_line + index + 1
          lineno_str = lineno.to_s.rjust(max_lineno_width)

          if lineno == current_line
            puts " => #{lineno_str}| #{line}"
          else
            puts "    #{lineno_str}| #{line}"
          end
        end
      end
    end
  end

  SESSION = Session.new
end

class Binding
  def debug
    Debugger::SESSION.suspend!(self)
  end
end
```

```ruby
def display_code(binding)
  file, current_line = binding.source_location

  if File.exist?(file)
    lines = File.readlines(file)
    end_line = [current_line + 5, lines.count].min - 1
    start_line = [end_line - 9, 0].max
    puts "[#{start_line + 1}, #{end_line + 1}] in #{file}"
    max_lineno_width = (end_line + 1).to_s.size
    lines[start_line..end_line].each_with_index do |line, index|
      lineno = start_line + index + 1
      lineno_str = lineno.to_s.rjust(max_lineno_width)

      if lineno == current_line
        puts " => #{lineno_str}| #{line}"
      else
        puts "    #{lineno_str}| #{line}"
      end
    end
  end
end
```

```
[3, 12] in app.rb
     3| def fib(num)
     4|   if num < 2
     5|     num
     6|   else
=>   7|     binding.debug
     8|     fib(num-1) + fib(num-2)
     9|   end
    10| end
    11|
    12| a = fib(6)
(debug)
```

2. Step-in

- Once received **step**, the debugger steps to the next program execution
- Allows us to move deeper into the program


1 require "debugger"
2
3 def fib(num)
4 if num < 2
5 num
6 else
7 fib(num-1) + fib(num-2)
8 end
9 end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b

>

>

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```
28 def step_in
29   TracePoint.trace(:line) do |tp|
30     # There are some internal files we don't want to step into
31     next if internal_path?(File.expand_path(tp.path))
32
33     # Disable the TracePoint after we hit the next execution
34     tp.disable
35     suspend!(tp.binding)
36   end
37 end
```

```
1  require "debugger"
2
3  def fib(num)
4    if num < 2
5      num
6    else
7      fib(num-1) + fib(num-2)
8    end
9  end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

binding.debug
a = fib(6)
b = fib(7)
puts a + b
```

```
28 def step_in
29   TracePoint.trace(:line) do |tp|
30     # There are some internal files we don't want to step into
31     next if internal_path?(File.expand_path(tp.path))
32
33     # Disable the TracePoint after we hit the next execution
34     tp.disable
35     suspend!(tp.binding)
36   end
37 end
```

```
1  require "debugger"
2
3  def fib(num)
4    if num < 2
5      num
6    else
7      fib(num-1) + fib(num-2)
8    end
9  end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

❌ Debugger execution

❌ Standard/default libraries

❌ Ruby internal

✅ Their program’s next execution

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
# 1. Check if the path is inside the debugger itself
# 2. Check if the path is inside Ruby's standard library
# 3. Check if the path is inside Ruby's internal files
# 4. Check if the path is inside Reline
def internal_path?(path)
  path.start_with?(__dir__) || path.start_with?(RbConfig::CONFIG["rubylibdir"]) ||
  path.match?(/<internal:/) || path.start_with?(RELINE_PATH)
end
```

```ruby
28 def step_in
29   TracePoint.trace(:line) do |tp|
30     # There are some internal files we don't want to step into
31     next if internal_path?(File.expand_path(tp.path))
32
33     # Disable the TracePoint after we hit the next execution
34     tp.disable
35     suspend!(tp.binding)
36   end
37 end
```

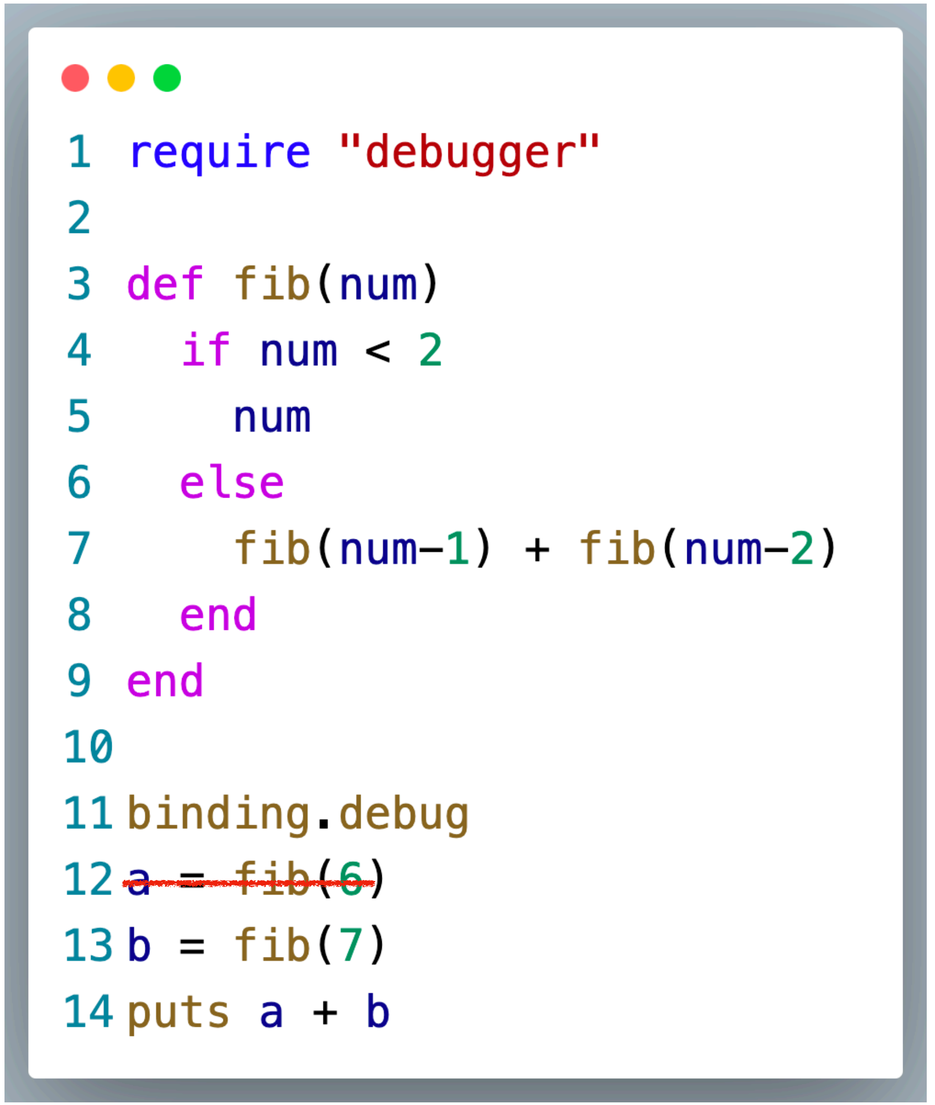
```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

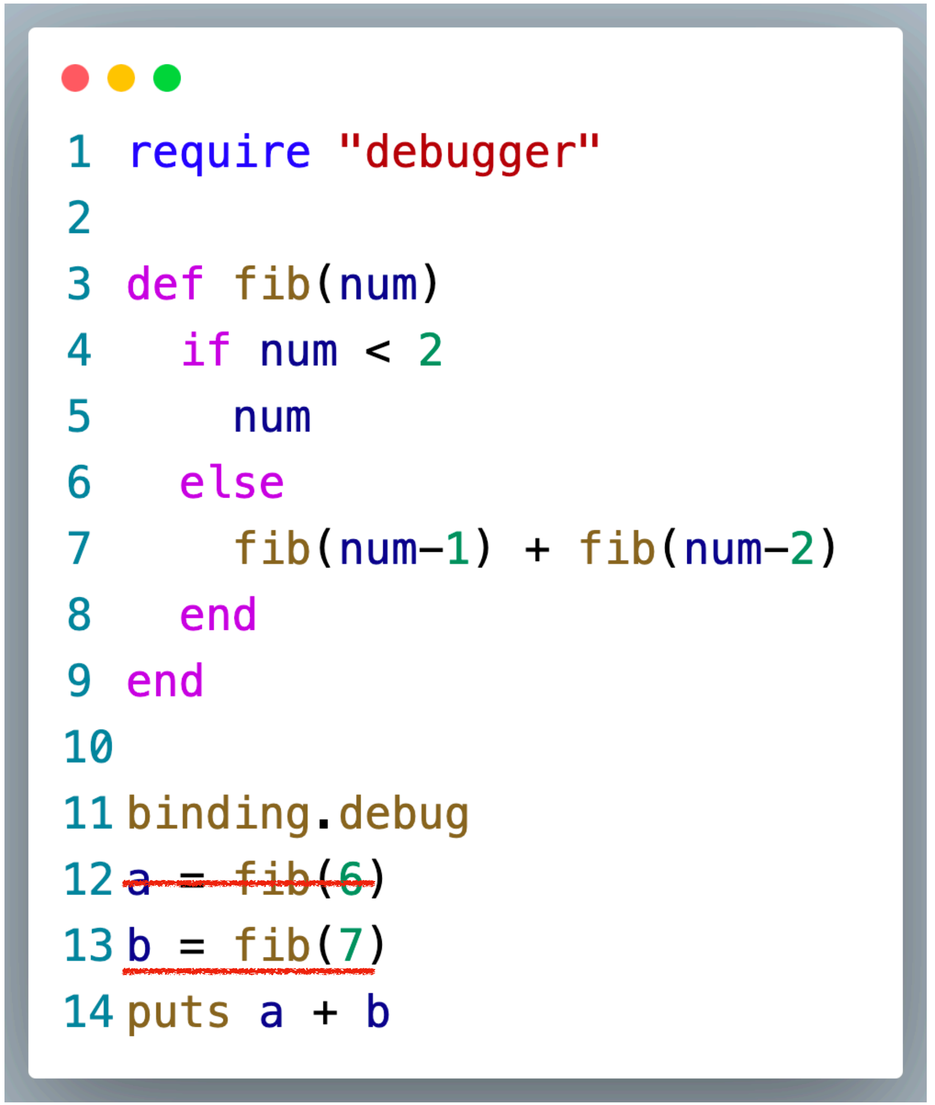
```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

3. Step-over

- Once received **next**, the debugger steps to the next line
- Skip detail executions

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

binding.debug
a = fib(6)
b = fib(7)
puts a + b
```


1 require "debugger"
2
3 def fib(num)
4   if num < 2
5   num
6   else
7   fib(num-1) + fib(num-2)
8   end
9 end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b


1 require "debugger"
2
3 def fib(num)
4   if num < 2
5   num
6   else
7   fib(num-1) + fib(num-2)
8   end
9 end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b

```
> b
```

```
> b
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "next"
    step_over
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "next"
    step_over
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "next"
    step_over
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
while input = Reline.readline("(debug) ")
  case input
  when "step"
    step_in
    break
  when "next"
    step_over
    break
  when "continue"
    break
  when "exit"
    exit
  else
    puts "=> " + eval_input(binding, input).inspect
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_in
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))

    # Disable the TracePoint after we hit the next execution
    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  puts "Current depth: #{current_depth}"
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    line = File.readlines(tp.path)[tp.lineno - 1]
    puts "Line #{tp.lineno} (depth: #{depth}): #{line}"
    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  puts "Current depth: #{current_depth}"
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    line = File.readlines(tp.path)[tp.lineno - 1]
    puts "Line #{tp.lineno} (depth: #{depth}): #{line}"
    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  puts "Current depth: #{current_depth}"
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    line = File.readlines(tp.path)[tp.lineno - 1]
    puts "Line #{tp.lineno} (depth: #{depth}): #{line}"
    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```
     5|     num
     6|   else
     7|     fib(num-1) + fib(num-2)
     8|   end
     9| end
    10|
    11| binding.debug
 => 12| a = fib(6)
    13| b = fib(7)
    14| puts a + b
(debug) next
Current depth: 1
Line 4 (depth: 2):   if num < 2
Line 7 (depth: 2):     fib(num-1) + fib(num-2)
Line 4 (depth: 3):   if num < 2
Line 7 (depth: 3):     fib(num-1) + fib(num-2)
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 7 (depth: 6):     fib(num-1) + fib(num-2)
Line 4 (depth: 7):   if num < 2
Line 5 (depth: 7):     num
Line 4 (depth: 7):   if num < 2
Line 5 (depth: 7):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 3):   if num < 2
Line 7 (depth: 3):     fib(num-1) + fib(num-2)
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 13 (depth: 1): b = fib(7)
[5, 14] in app.rb
     5|     num
     6|   else
     7|     fib(num-1) + fib(num-2)
     8|   end
     9| end
    10|
    11| binding.debug
    12| a = fib(6)
 => 13| b = fib(7)
    14| puts a + b
```

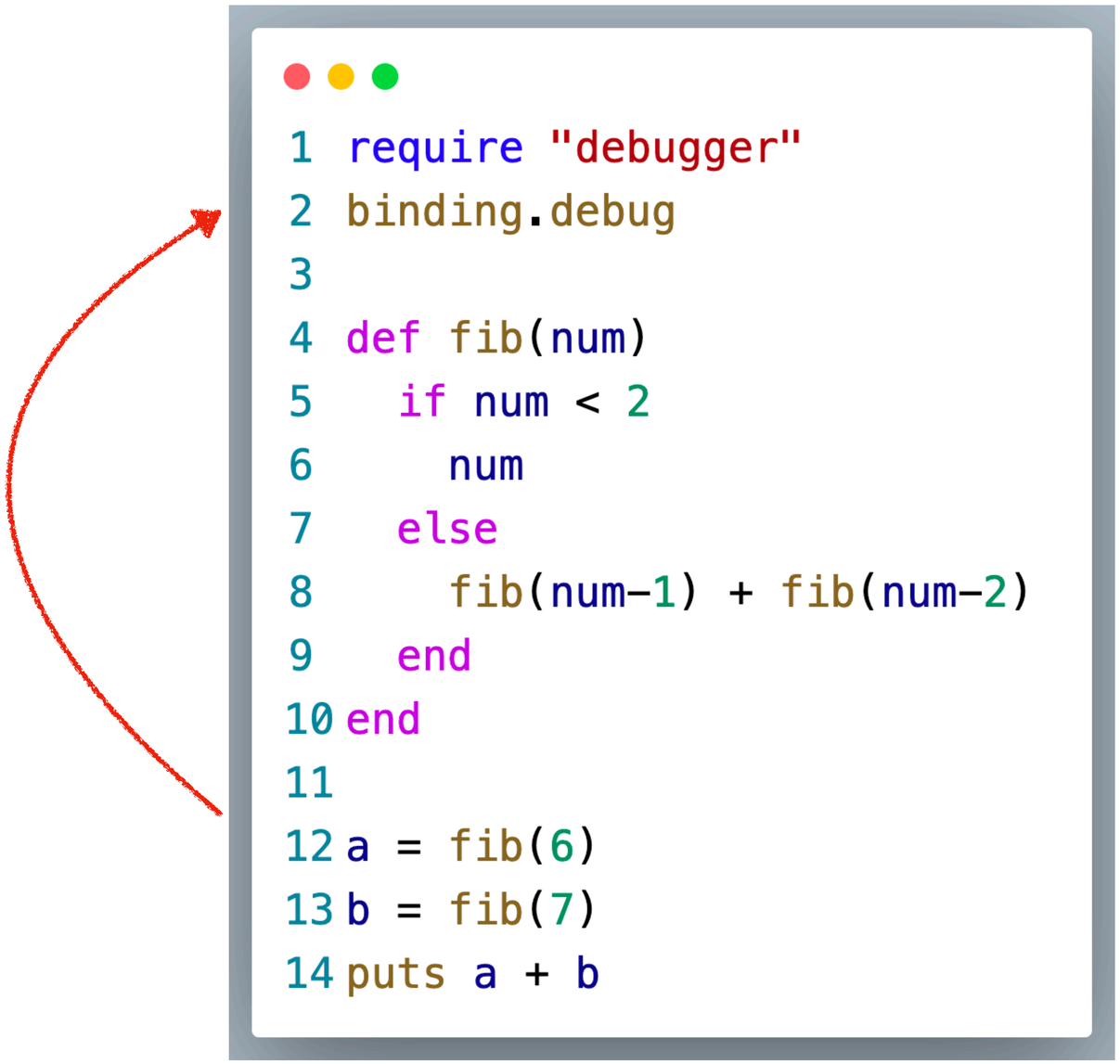
```ruby
def step_over
  # ignore call frames from the debugger itself
  current_depth = caller.length - 2

  puts "Current depth: #{current_depth}"
  TracePoint.trace(:line) do |tp|
    # There are some internal files we don't want to step into
    next if internal_path?(File.expand_path(tp.path))
    depth = caller.length

    line = File.readlines(tp.path)[tp.lineno - 1]
    puts "Line #{tp.lineno} (depth: #{depth}): #{line}"
    next if current_depth < depth

    tp.disable
    suspend!(tp.binding)
  end
end
```

```
(debug) next
Current depth: 1
Line 4 (depth: 2):   if num < 2
Line 7 (depth: 2):     fib(num-1) + fib(num-2)
Line 4 (depth: 3):   if num < 2
Line 7 (depth: 3):     fib(num-1) + fib(num-2)
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 7 (depth: 6):     fib(num-1) + fib(num-2)
Line 4 (depth: 7):   if num < 2
Line 5 (depth: 7):     num
Line 4 (depth: 7):   if num < 2
Line 5 (depth: 7):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 3):   if num < 2
Line 7 (depth: 3):     fib(num-1) + fib(num-2)
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 7 (depth: 5):     fib(num-1) + fib(num-2)
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 6):   if num < 2
Line 5 (depth: 6):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 4):   if num < 2
Line 7 (depth: 4):     fib(num-1) + fib(num-2)
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 4 (depth: 5):   if num < 2
Line 5 (depth: 5):     num
Line 13 (depth: 1): b = fib(7)
[5, 14] in app.rb
     5|     num
     6|   else
     7|     fib(num-1) + fib(num-2)
     8|   end
     9| end
    10|
```

4. Breakpoint commands

- Add/remove breakpoints without modifying the program
- Greatly increase the range of movement (e.g. debug gems without bundle open)

- **break <file>:<num>** and **break <num>** to add breakpoints
- **break** to list breakpoints
- **delete <id>** to delete breakpoints



```ruby
require "debugger"
binding.debug

def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

```
> b
```

```
> b
```

```ruby
module Debugger
  class Session
    def initialize
      @breakpoints = []
    end
```

```ruby
def add_breakpoint(file, line, **options)
  bp = LineBreakpoint.new(file, line, **options)
  @breakpoints << bp
  puts "Breakpoint added: #{bp.location}"
  bp.enable
end
```

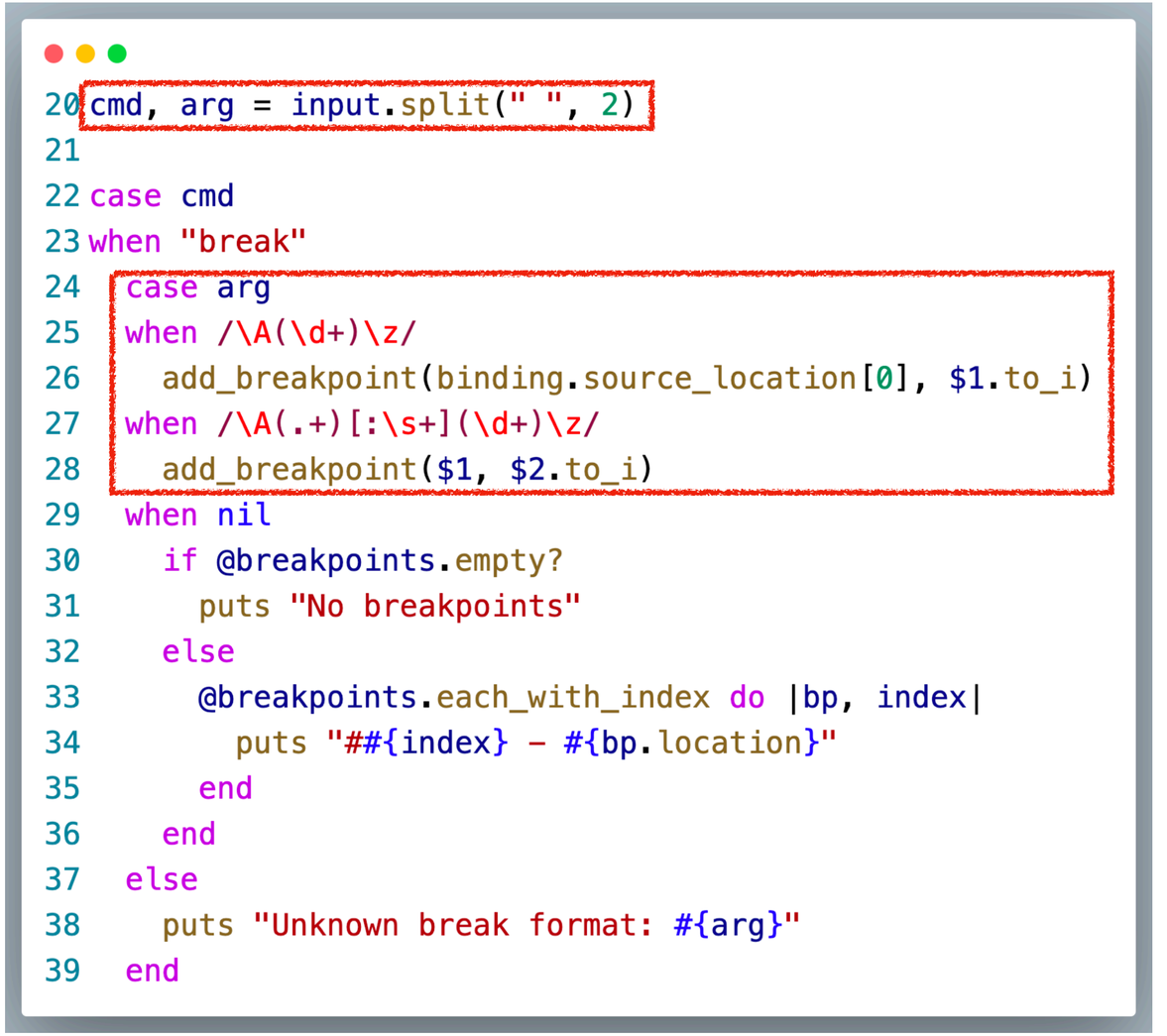
```
cmd, arg = input.split(" ", 2)

case cmd
when "break"
  case arg
  when /\A(\d+)\z/
    add_breakpoint(binding.source_location[0], $1.to_i)
  when /\A(.+)[:\s+](\d+)\z/
    add_breakpoint($1, $2.to_i)
  when nil
    if @breakpoints.empty?
      puts "No breakpoints"
    else
      @breakpoints.each_with_index do |bp, index|
        puts "##{index} - #{bp.location}"
      end
    end
  else
    puts "Unknown break format: #{arg}"
  end
```

Split input

```
20 cmd, arg = input.split(" ", 2)
21
22 case cmd
23 when "break"
24   case arg
25   when /\A(\d+)\z/
26     add_breakpoint(binding.source_location[0], $1.to_i)
27   when /\A(.+)[:\s+](\d+)\z/
28     add_breakpoint($1, $2.to_i)
29   when nil
30     if @breakpoints.empty?
31       puts "No breakpoints"
32     else
33       @breakpoints.each_with_index do |bp, index|
34         puts "##{index} - #{bp.location}"
35       end
36     end
37   else
38     puts "Unknown break format: #{arg}"
39   end
```

Split input

Add breakpoints

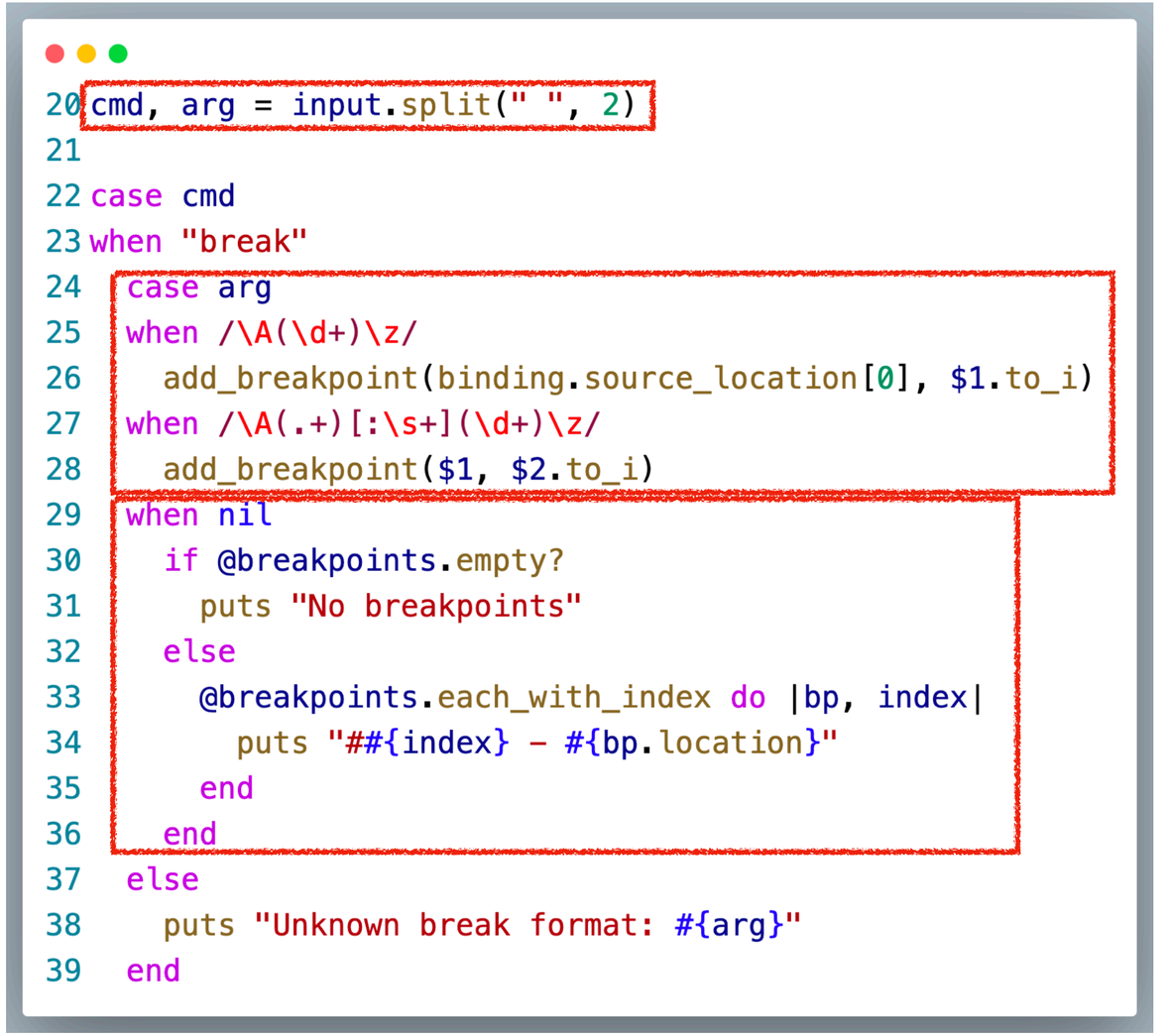


```ruby
cmd, arg = input.split(" ", 2)

case cmd
when "break"
  case arg
  when /\A(\d+)\z/
    add_breakpoint(binding.source_location[0], $1.to_i)
  when /\A(.+)[:\s+](\d+)\z/
    add_breakpoint($1, $2.to_i)
  when nil
    if @breakpoints.empty?
      puts "No breakpoints"
    else
      @breakpoints.each_with_index do |bp, index|
        puts "##{index} - #{bp.location}"
      end
    end
  else
    puts "Unknown break format: #{arg}"
  end
```

Split input

Add breakpoints

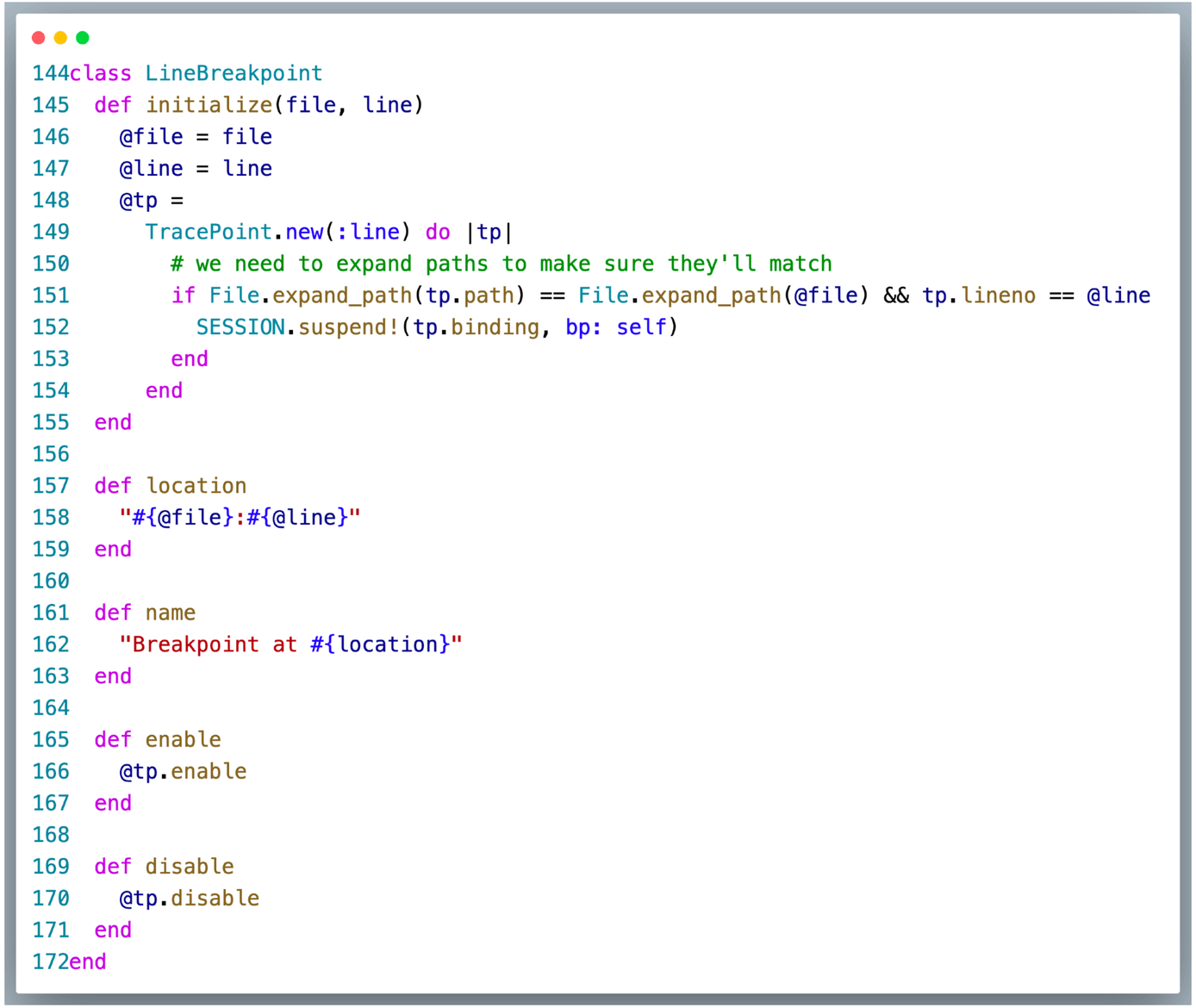
List breakpoints



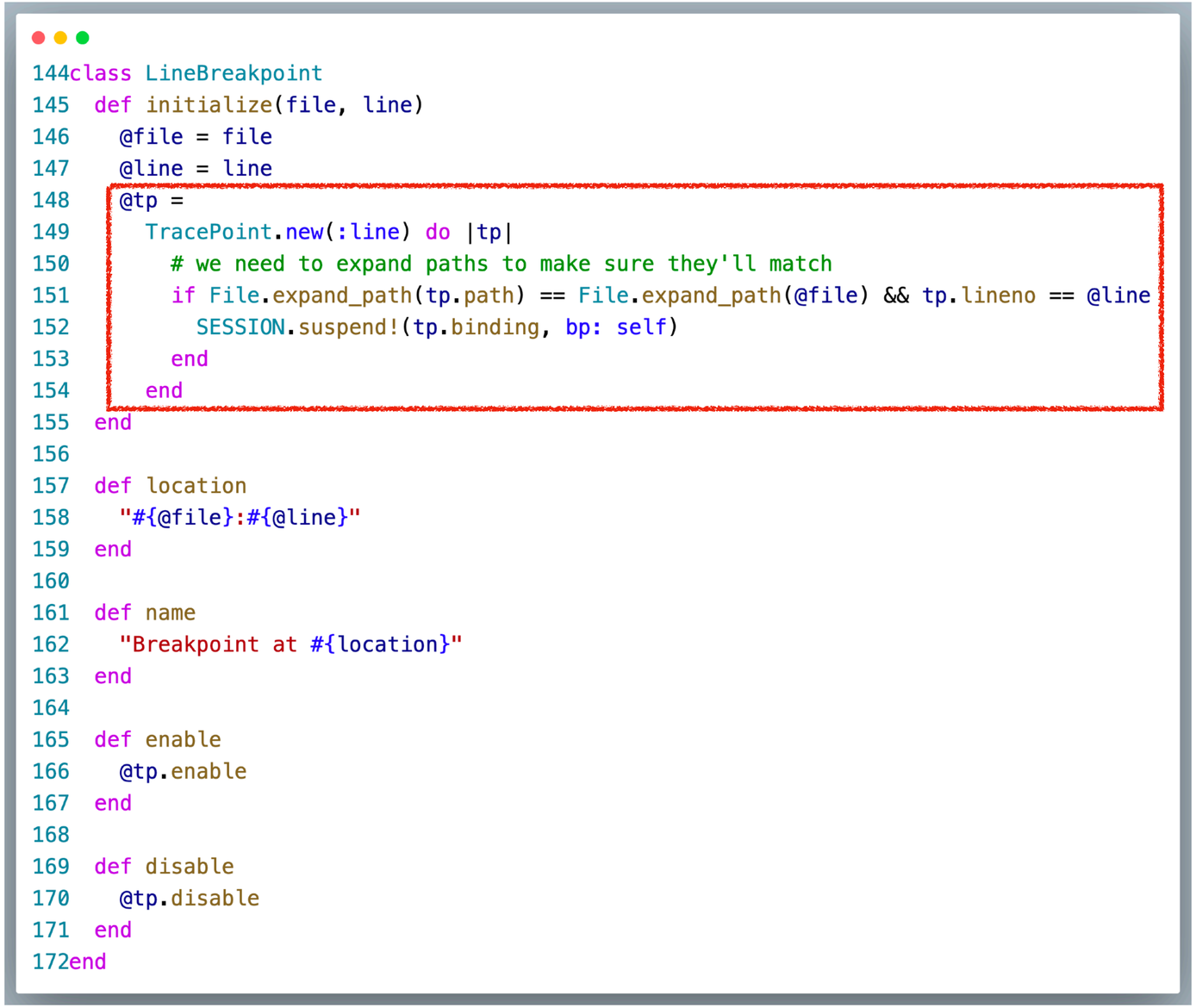
```
20 cmd, arg = input.split(" ", 2)
21
22 case cmd
23 when "break"
24   case arg
25   when /\A(\d+)\z/
26     add_breakpoint(binding.source_location[0], $1.to_i)
27   when /\A(.+)[:\s+](\d+)\z/
28     add_breakpoint($1, $2.to_i)
29   when nil
30     if @breakpoints.empty?
31       puts "No breakpoints"
32     else
33       @breakpoints.each_with_index do |bp, index|
34         puts "##{index} - #{bp.location}"
35       end
36     end
37   else
38     puts "Unknown break format: #{arg}"
39   end
```

Delete breakpoints

```ruby
when "delete"
  index = arg.to_i

  if bp = @breakpoints.delete_at(index)
    bp.disable
    puts "Breakpoint ##{index} (#{bp.location}) has been deleted"
  else
    puts "Breakpoint ##{index} not found"
  end
```



```ruby
class LineBreakpoint
  def initialize(file, line)
    @file = file
    @line = line
    @tp =
      TracePoint.new(:line) do |tp|
        # we need to expand paths to make sure they'll match
        if File.expand_path(tp.path) == File.expand_path(@file) && tp.lineno == @line
          SESSION.suspend!(tp.binding, bp: self)
        end
      end
  end

  def location
    "#{@file}:#{@line}"
  end

  def name
    "Breakpoint at #{location}"
  end

  def enable
    @tp.enable
  end

  def disable
    @tp.disable
  end
end
```



```ruby
class LineBreakpoint
  def initialize(file, line)
    @file = file
    @line = line
    @tp =
      TracePoint.new(:line) do |tp|
        # we need to expand paths to make sure they'll match
        if File.expand_path(tp.path) == File.expand_path(@file) && tp.lineno == @line
          SESSION.suspend!(tp.binding, bp: self)
        end
      end
  end

  def location
    "#{@file}:#{@line}"
  end

  def name
    "Breakpoint at #{location}"
  end

  def enable
    @tp.enable
  end

  def disable
    @tp.disable
  end
end
```

```ruby
class LineBreakpoint
  def initialize(file, line)
    @file = file
    @line = line
    @tp =
      TracePoint.new(:line) do |tp|
        # we need to expand paths to make sure they'll match
        if File.expand_path(tp.path) == File.expand_path(@file) && tp.lineno == @line
          SESSION.suspend!(tp.binding, bp: self)
        end
      end
  end

  def location
    "#{@file}:#{@line}"
  end

  def name
    "Breakpoint at #{location}"
  end

  def enable
    @tp.enable
  end

  def disable
    @tp.disable
  end
end
```

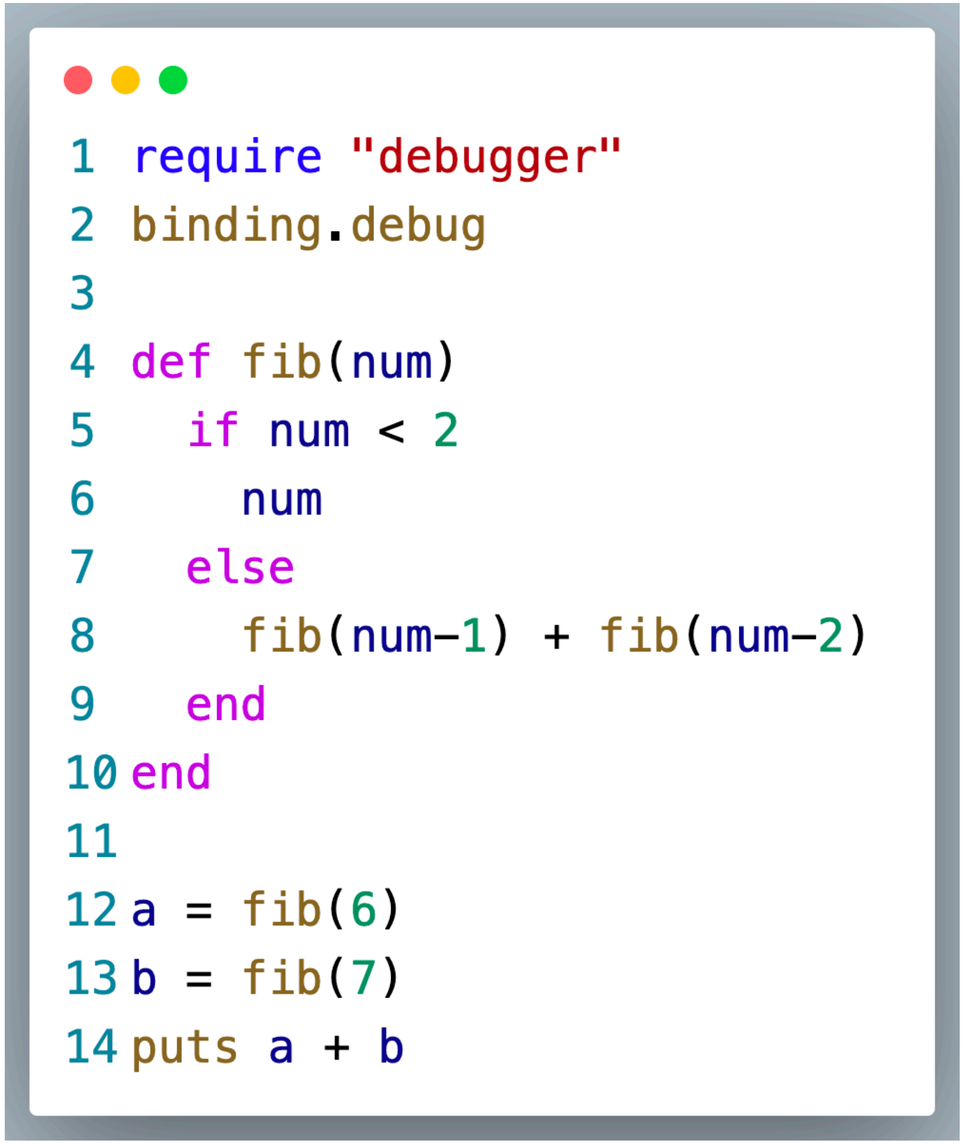
```ruby
class LineBreakpoint
  def initialize(file, line)
    @file = file
    @line = line
    @tp =
      TracePoint.new(:line) do |tp|
        # we need to expand paths to make sure they'll match
        if File.expand_path(tp.path) == File.expand_path(@file) && tp.lineno == @line
          SESSION.suspend!(tp.binding, bp: self)
        end
      end
  end

  def location
    "#{@file}:#{@line}"
  end

  def name
    "Breakpoint at #{location}"
  end

  def enable
    @tp.enable
  end

  def disable
    @tp.disable
  end
end
```

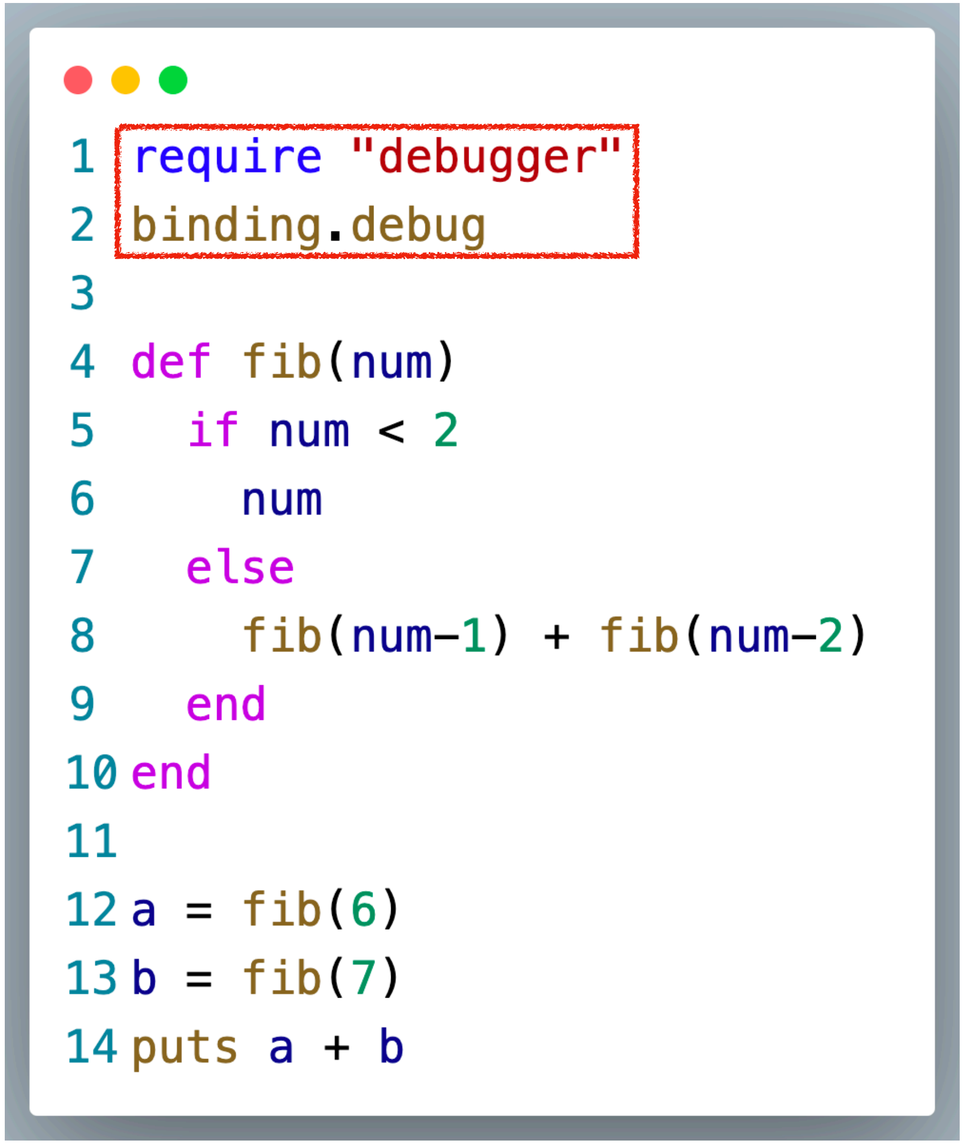
How does ruby/debug avoid this?

- Collects ISeq objects from ObjectSpace
- Locates the ISeq object of the breakpoint location
- Uses heuristic to check which line is best to stop the program
- Activates TracePoint on that ISeq object’s specific line

How does ruby/debug avoid this?

- Collects ISeq objects from ObjectSpace
- Locates the ISeq object of the breakpoint location
- Uses heuristic to check which line is best to stop the program
- Activates TracePoint on that ISeq object’s specific line
- Reduces runtime overhead with more sophisticated breakpoint activation


require "debugger"
binding.debug
def fib(num)
if num < 2
num
else
fib(num-1) + fib(num-2)
end
end
a = fib(6)
b = fib(7)
puts a + b


require "debugger"
binding.debug
def fib(num)
if num < 2
num
else
fib(num-1) + fib(num-2)
end
end
a = fib(6)
b = fib(7)
puts a + b

5. Debugger executable

- Debug without requiring the debugger
- **$ exe/debug app.rb** runs **app.rb** with debugger required
- Stop at the beginning of the program to receive further instructions (e.g. breakpoint commands)

```ruby
def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

```
> b
```

```
> b
```

$ exe/debug app.rb

exe/debug

$ exe/debug app.rb

```ruby
#!/usr/bin/env ruby

program, *_ = ARGV

Kernel.exec({ "RUBYOPT" => "-Ilib -rdebugger" }, "ruby", program)
```

exe/debug

$ exe/debug app.rb

```ruby
#!/usr/bin/env ruby

program, *_ = ARGV

Kernel.exec({ "RUBYOPT" => "-Ilib -rdebugger" }, "ruby", program)
```

exec([env,] command... [,options])

Replaces the current process by running the given external *command*, which can take one of the following forms:

exe/debug

$ exe/debug app.rb

```ruby
#!/usr/bin/env ruby

program, *_ = ARGV

Kernel.exec({ "RUBYOPT" => "-Ilib -rdebugger" }, "ruby", program)
```

exec([env,] command... [,options])

Replaces the current process by running the given external *command*, which can take one of the following forms:

$ RUBYOPT="-Ilib -rdebugger" ruby app.rb

exe/debug

$ exe/debug app.rb

```ruby
#!/usr/bin/env ruby

program, *_ = ARGV

Kernel.exec({ "RUBYOPT" => "-Ilib -rdebugger" }, "ruby", program)
```

exec([env,] command... [,options])

Replaces the current process by running the given external *command*, which can take one of the following forms:

$ RUBYOPT="-Ilib -rdebugger" ruby app.rb

lib/debugger.rb

```ruby
187if ENV["RUBYOPT"] && ENV["RUBYOPT"].split.include?("-rdebugger")
188  Debugger::SESSION.add_breakpoint($0, 1, once: true)
189end
```

The Result - 189 lines

```
> b
```

The Result - 189 lines

```
> b
```

Recap

```ruby
def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

Reline

Binding

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    binding.debug
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

Recap

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    binding.debug
    fib(num-1) + fib(num-2)
  end
end

a = fib(6)
b = fib(7)
puts a + b
```

TracePoint

Step-in

Step-over

```ruby
require "debugger"

def fib(num)
  if num < 2
    num
  else
    fib(num-1) + fib(num-2)
  end
end

binding.debug
a = fib(6)
b = fib(7)
puts a + b
```

Recap

```
1  require "debugger"
2
3  def fib(num)
4    if num < 2
5      num
6    else
7      fib(num-1) + fib(num-2)
8    end
9  end
10
11 binding.debug
12 a = fib(6)
13 b = fib(7)
14 puts a + b
```

Breakpoint commands

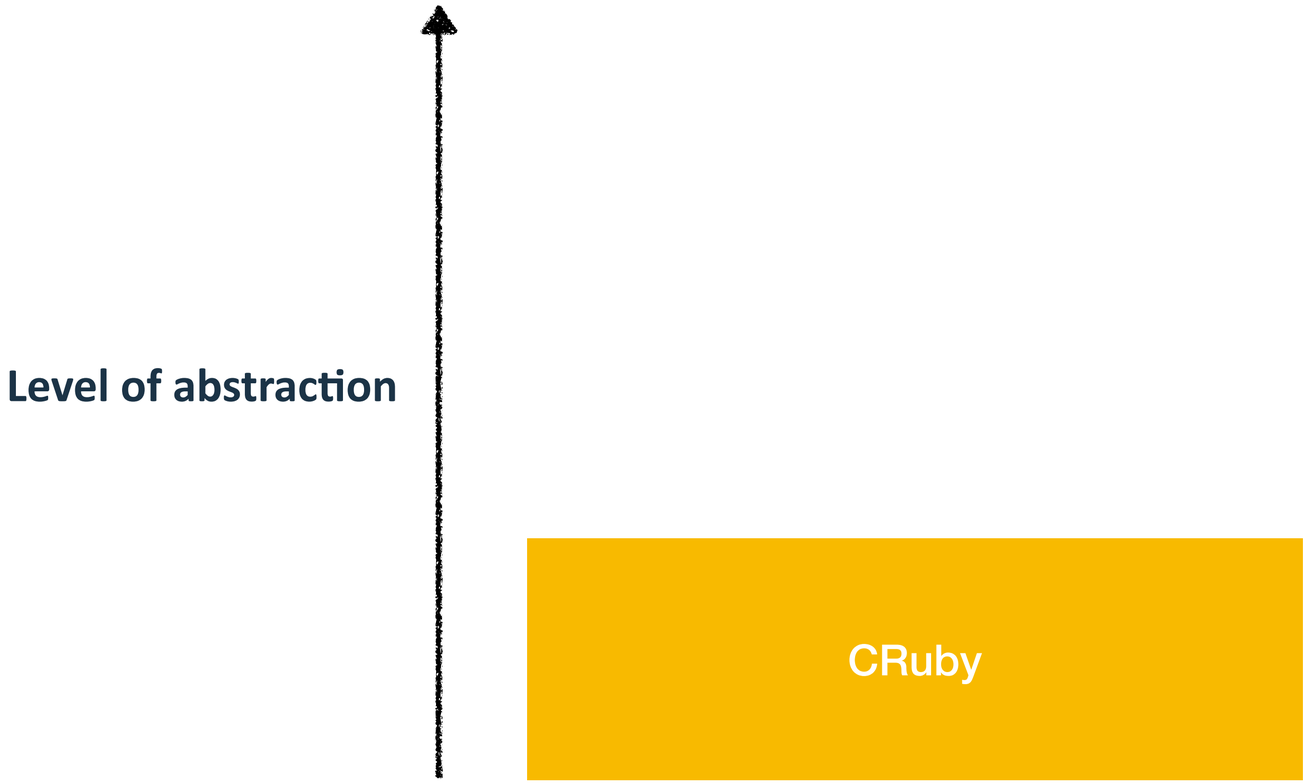
$ exe/debug

```
1  def fib(num)
2    if num < 2
3      num
4    else
5      fib(num-1) + fib(num-2)
6    end
7  end
8
9  a = fib(6)
10 b = fib(7)
11 puts a + b
```

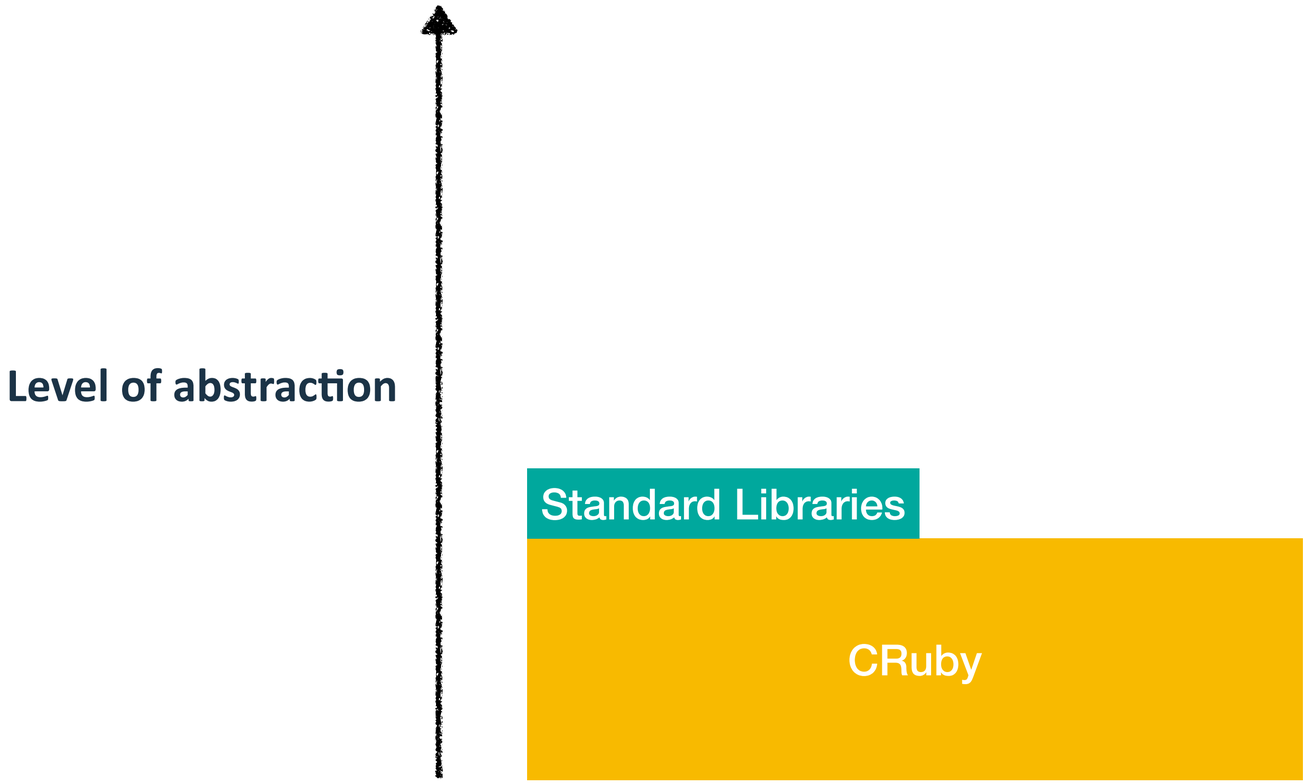
How to choose debugging tools?

Puts? Debuggers?

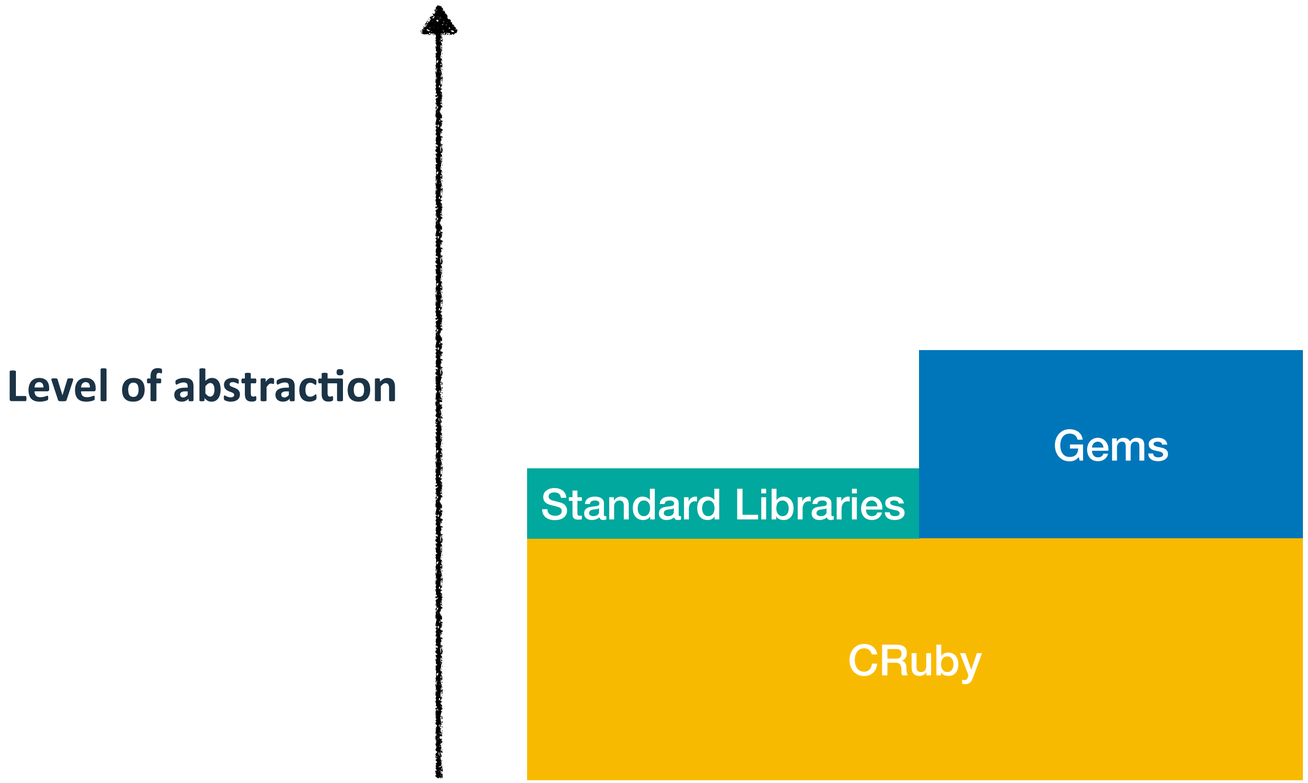
Puts? Debuggers?



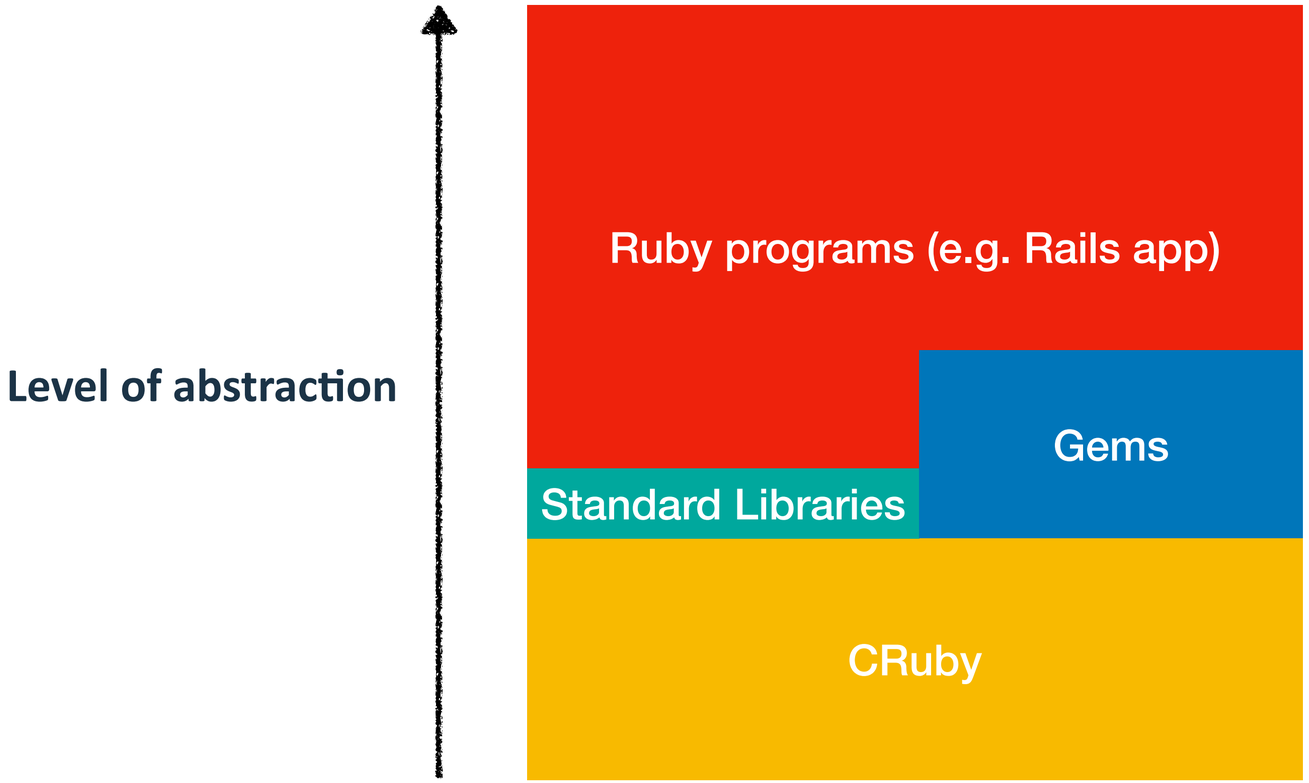
Puts? Debuggers?



Puts? Debuggers?



Puts? Debuggers?



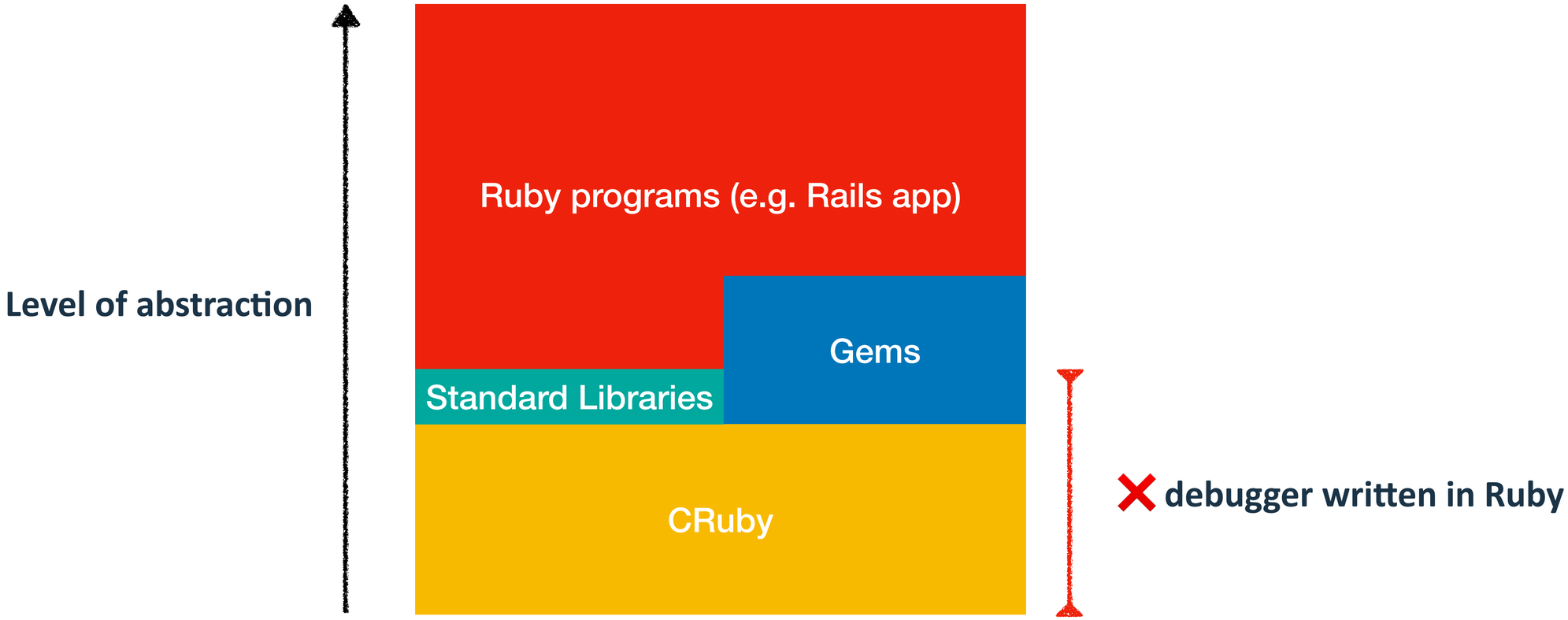
What CAN’T the debugger debug?

- Itself
- Reline
- Standard libraries
- CRuby (e.g. TracePoint, Binding)

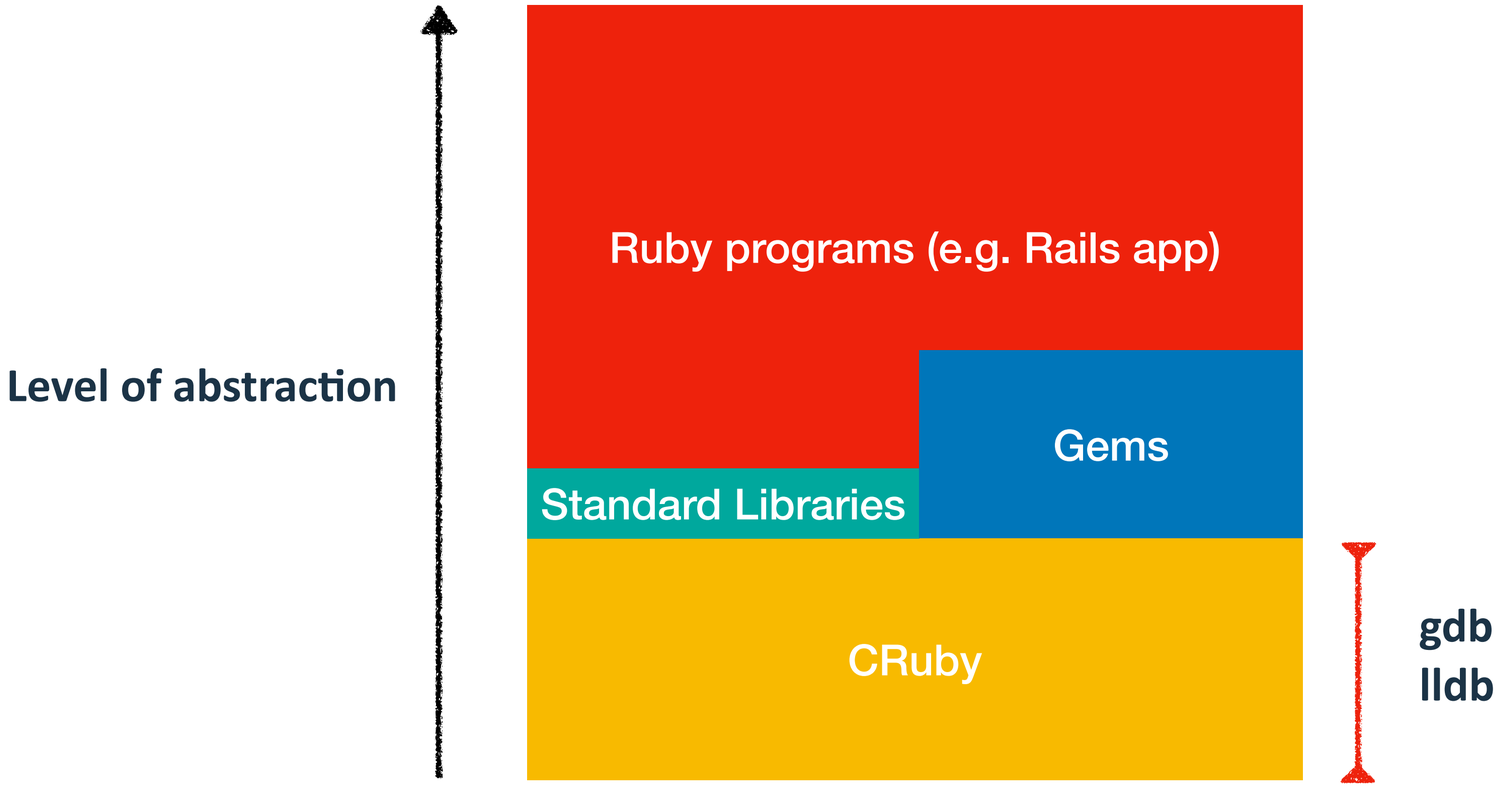
Side-effects

- TracePoint
- Tracks and retains data
- Stop/resume threads

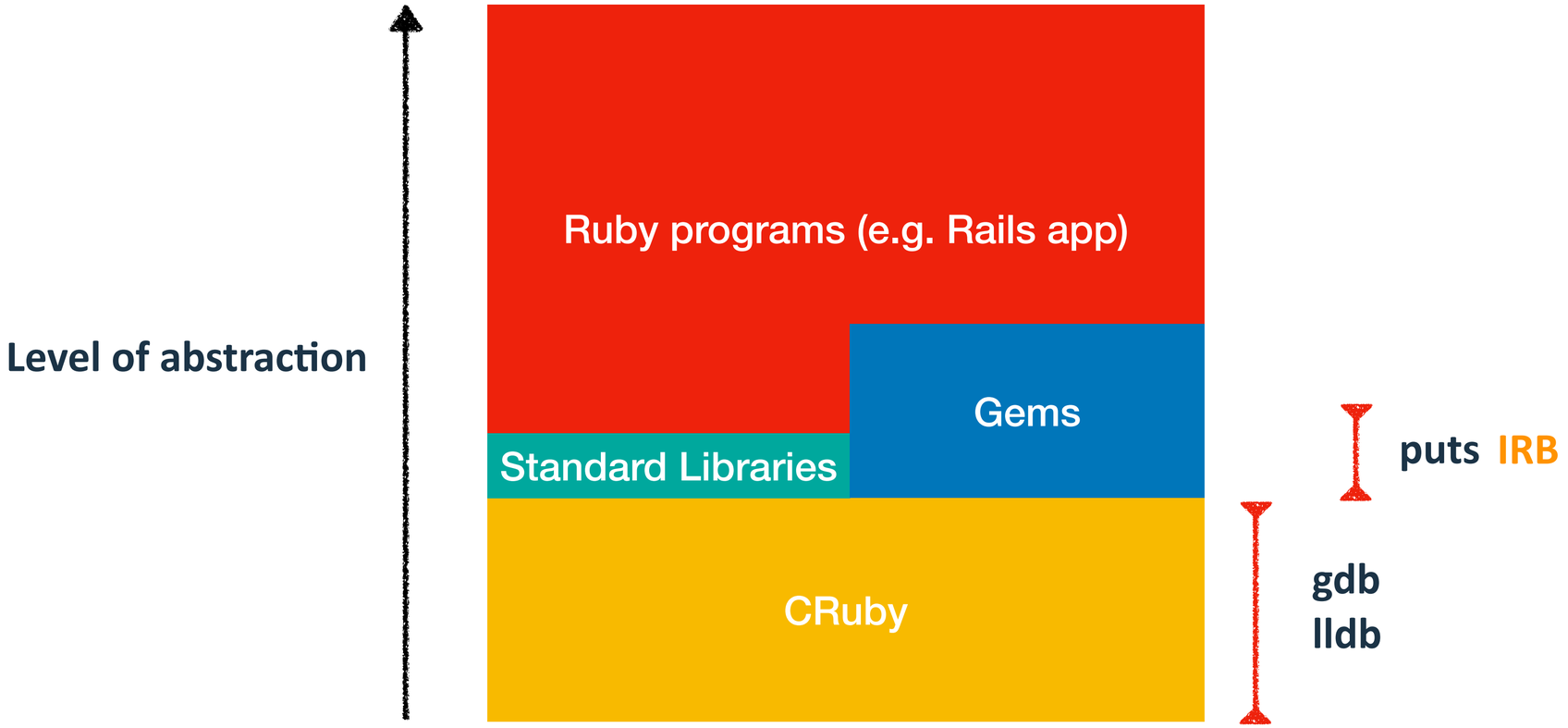
Puts? Debuggers?



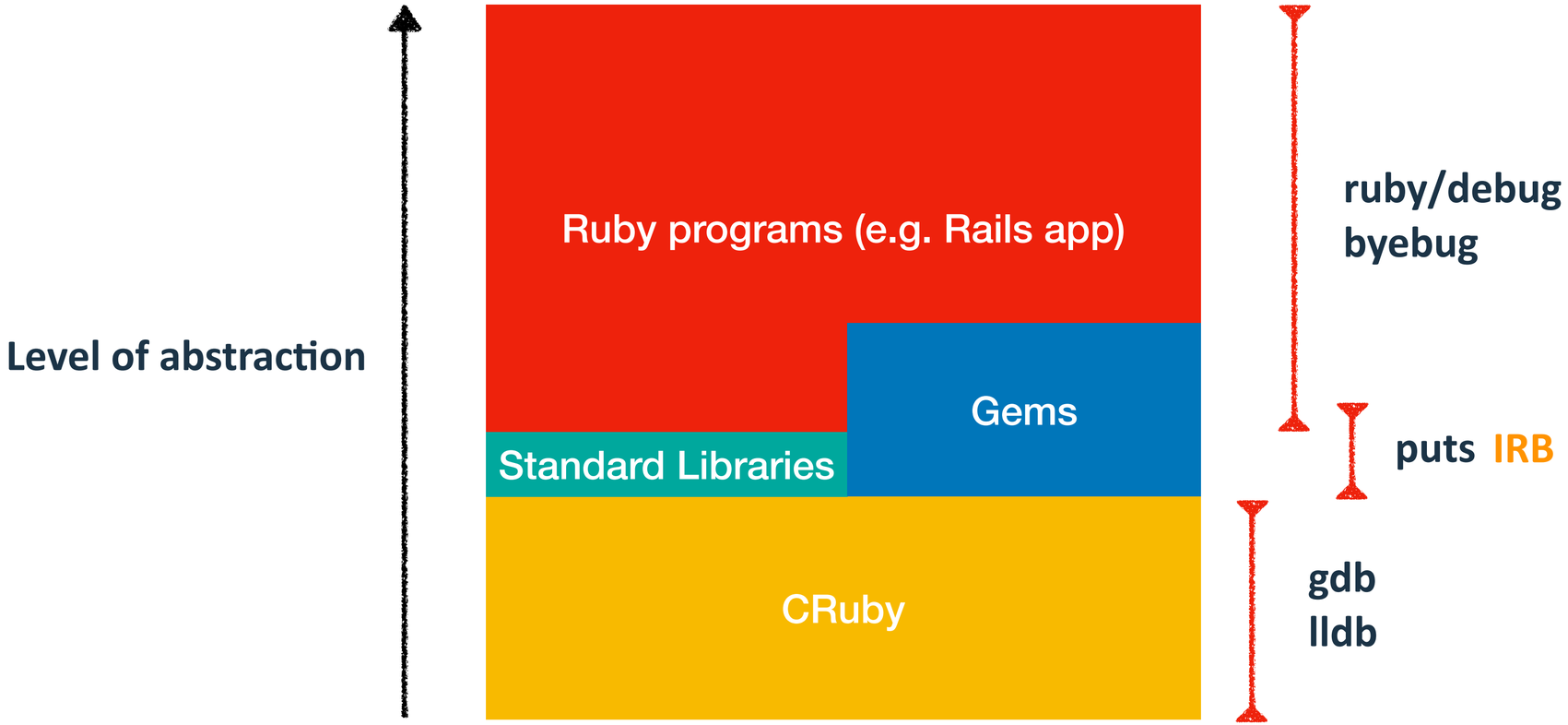
Puts? Debuggers?



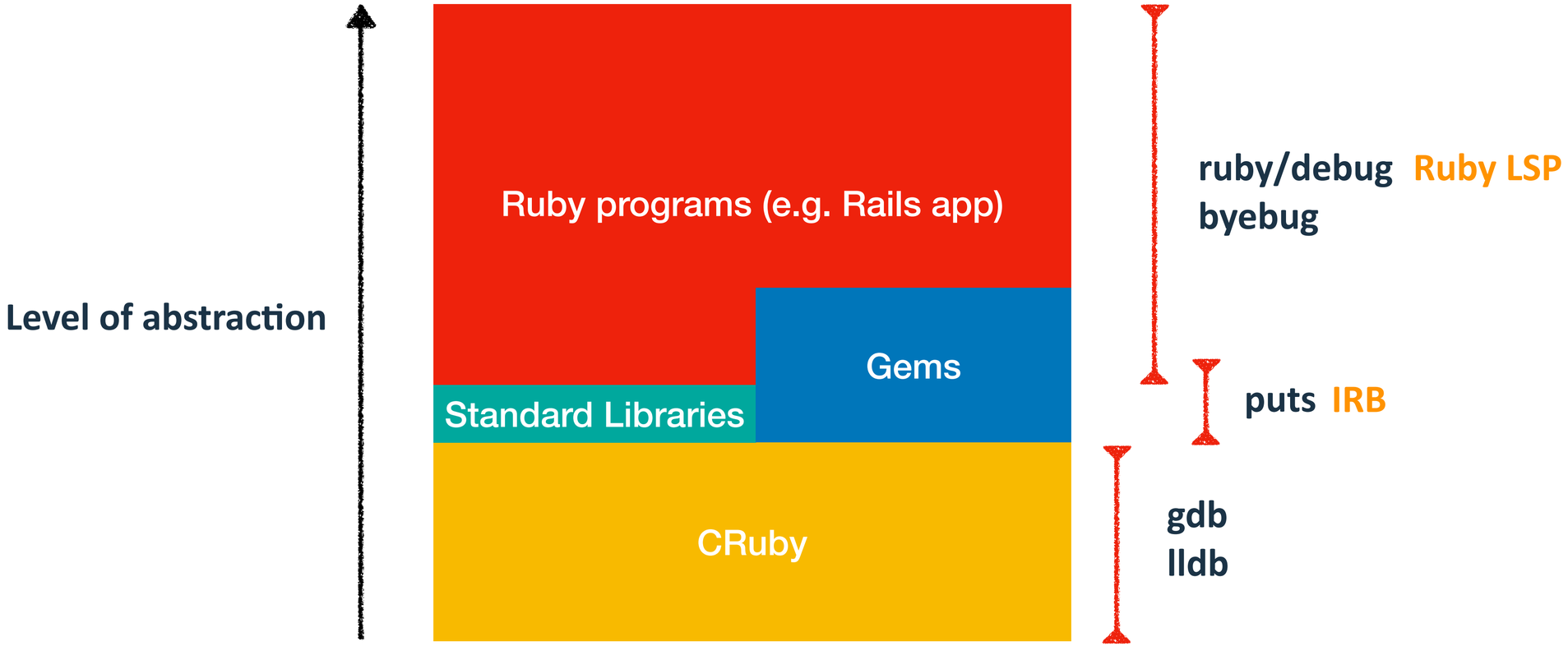
Puts? Debuggers?



Puts? Debuggers?



Puts? Debuggers?



GitHub Repo

https://github.com/st0012/mini-debugger

Next Steps

- Make **step** and **next** accept a **<n>** argument
  - e.g. **step 2** does 2 steps
- Implement **catch** command
  - Breakpoints triggered when an exception is raised
- Implement **finish** command
  - Finish the current frame

Thanks for listening