Ruby Debugging Cheatsheet & Tips

A comprehensive guide to debugging Ruby code using built-in tools, Pry, and Byebug. This cheat sheet covers essential commands, techniques, and best practices to efficiently identify and resolve issues in your Ruby applications.

Basic Debugging Techniques

Using `puts` and `p`

`puts` - Prints a string to the console, often used for simple debugging.

Example:

```
x = 5
puts "=" * 10
puts "Value of x: #{x}" # Output: Value of x: 5
puts "=" * 10
```

`p` - Prints a more detailed representation of an object, including its class.

Example:

```
arr = [1, 2, 3]
p arr # Output: [1, 2, 3]
```

`pp` - Pretty prints objects for better readability (requires `require 'pp'`).

Example:

```
require 'pp'
hash = {a: 1, b: {c: 2, d: 3}}
pp hash
```

Backtraces

Understanding backtraces is crucial for pinpointing the source of errors. Ruby provides detailed information about the call stack when an exception occurs.

Example:

```
def a
  b
end

def b
  raise 'Boom!'
end

a # This will generate a backtrace
```

Analyzing a Backtrace:

- The topmost line indicates the exception type and message.
- Subsequent lines show the call stack, with the most recent call at the top.
- Each line includes the file name, line number, and method name.
- Don't be afraid to generate exceptions to debug code

Logging

Using Ruby's built-in `Logger` class can help track program execution and variable states.

Example:

```
require 'logger'

logger = Logger.new(STDOUT)
logger.level = Logger::DEBUG # Set log level (DEBUG, INFO, WARN, ERROR, FATAL)

x = 10
logger.debug "Value of x: #{x}"
```

Log Levels:

- `DEBUG`: Detailed information, useful for debugging.
- `INFO`: General information about the application's operation.
- `WARN`: Potentially harmful situations.
- `ERROR`: Error events that might still allow the application to continue running.
- `FATAL`: Severe errors that cause the application to terminate.

Inspecting Methods with `method`

The `method(:method_name)` syntax allows you to obtain a `Method` object, enabling introspection and advanced debugging techniques.

Accessing Method Objects:

```
str = "hello"
method_object = str.method(:upcase)
puts method_object.call # => "HELLO"
```

Retrieving Method Source Location:

```
method_object = String.method(:new)
puts method_object.source_location # => ["string.rb", 42]
```

This returns an array containing the file path and line number where the method is defined.

Handling Methods Defined in C:

For methods implemented in C, `source_location` will return `nil`.

```
method_object = 1.method(:+) # Example of a C implemented method
puts method_object.source_location # => nil
```

Method Caller

`caller` method:

Returns an array of strings representing the call stack. Each string describes a single method call, including the file name, line number, and method name.

Basic usage:

```
def my_method
  caller
end

my_method # => ["/path/to/file.rb:2:in `my_method'", ...]
```

`caller(n)`:

Returns only the `n` most recent calls. Useful for limiting the output when the call stack is very deep.

Using `method` with Pry for Debugging:

Within a Pry session, `method` can be used to quickly inspect methods.

```
require 'pry'

def my_method(arg1, arg2)
  binding.pry
  arg1 + arg2
end

my_method(5, 3)
```

Inside Pry:

```
method(:my_method).source_location # => ["/path/to/your/file.rb", 3]
```

Inspecting Method Parameters:

The `parameters` method provides information about a method's arguments.

```
def some_method(a, b = 1, *c, d: 2)
  # method body
end

method(:some_method).parameters # => [[:req, :a], [:opt, :b], [:rest, :c], [:keyreq, :d]]
```

Stop when you need

Stop based on some conditions

```
user = some_method
debugger if user.name == 'John'
```

or

```
user = some_method
debugger if $some_variable

# and set it in some place
$some_variable = true
```

Hash/JSON

If you need to print Hash or JSON in a nice way

```
Rails.logger.debug(JSON.pretty_generate(
params.permit!.to_h))
```

You can also use "pp" method or gems like awesome_print.

keyword argum

Limiting the output:

```
def method_a
  method_b
end

def method_b
  caller(1)
end

method_a # => ["/path/to/file.rb:5:in `method_b'"]
```

`caller_locations`:

Returns an array of `Thread::Backtrace::Location` objects, providing more structured information than strings.

Using `caller_locations`:

```
def my_method
  caller_locations
end

loc = my_method.first
loc.path   # => "/path/to/file.rb"
loc.lineno # => 2
loc.label  # => "my_method"
```

Filtering the call stack:

You can use `grep` or other array methods to filter the `caller` output to find specific method calls or files.

Filtering example:

```
def my_method
  caller.grep(/my_gem/)
end
```

Using `caller` for debugging:

Insert `puts caller` or `puts caller_locations` at strategic points to trace the execution path of your code.

Be aware of performance:

Avoid using `caller` in production code due to its performance overhead. It's primarily a debugging tool.

Debugging with Debug gem

Getting Started with Debug Gem

To start using the Debug gem, first add it to your Gemfile:

```
gem 'debug'
```

Then run `bundle install` to install it.

Require the Debug gem in your application with:

```
require 'debug'
```

To initiate a debugging session, insert the following line into your code where you want to start debugging:

```
debugger
```

Run your Ruby script. Execution will pause at the `debugger` line, and you'll enter the debug console.

Ensure you're running your Ruby application with `bundle exec` if launching with Bundler to include the gems in the environment.

Use `debugger(do: "...")` to execute a command and continue execution after hitting a breakpoint.

```
debugger(do: "info locals")
```

This will print local variables and then continue the program.

Use `debugger(pre: "...")` to execute a command upon hitting a breakpoint, before entering the console.

```
debugger(pre: "info locals")
```

This will print local variables and then open the console.

These options help automate common debugging tasks and reduce manual steps.

Navigation Commands

`next` - Execute the next line of code.

Moves to the next line within the same context, stepping over method calls.

`step` - Step into the method.

Executes the next line of code, stepping into any methods on the line.

`continue` or `c` - Resume program execution.

This will continue running the program until the next breakpoint.

`finish` - Execute until the current method returns.

This is useful for quickly skipping over long methods.

`break <line>` - Set a breakpoint at a specific line.

Example: `break 42` will pause execution when line 42 is reached.

Inspection Commands

`list` - Display the code around the current line.

Useful to see the surrounding context.

`p` or `print <expression>` - Evaluate and print an expression.

Example: `p user.name` to check the current name of the user object.

`display <expression>` - Automatically show the value of an expression every time the debugger pauses.

`info <subcommand>` - Show information about the program state.

For example, `info variables` lists all local variables and their values.

`frame` - Display the current call stack frame.

You can also use `frame up` and `frame down` to navigate the stack.

Breakpoints Management

`break <line>` - Set a breakpoint at a given line number.

Example: `break 15` sets a breakpoint at line 15.

`break if <condition>` - Conditional breakpoint.

Stops execution when the specified condition is true. Example: `break if x > 5`.

`delete <breakpoint_number>` - Remove a specific breakpoint.

Use `delete 1` to remove the first breakpoint.

`enable <breakpoint_number>` - Enable a disabled breakpoint.

Example: `enable 2`.

`disable <breakpoint_number>` - Temporarily disable a breakpoint without removing it.

Utilize `disable 3` to deactivate the third breakpoint.

Additional Commands

`quit` or `exit` - Terminate the debugging session.

`irb` - Open an interactive Ruby shell within the current context.

`history` - Display previous commands entered in the session.

`trace` - Print a trace of function calls on each line.

Activate tracing with specific options as needed.

`eval <expression>` - Evaluate Ruby code in the current context.

Example: `eval 'puts Hello, world!'`.

Debugging with Pry

Pry Basics

Pry is a powerful alternative to `irb` that provides enhanced debugging capabilities.

Installation:

```
gem install pry
```

To start a Pry session, insert `binding.pry` into your code.

Example:

```
require 'pry'

def my_method(arg)
  binding.pry # Execution pauses here
  puts arg
end

my_method('Hello, Pry!')
```

Common Pry Commands

`ls`	List variables and methods in the current scope.
`cd <object>`	Change the current context to the given object.
`whereami`	Show the current location in the code.
`show-source <method>`	Display the source code of a method.
`exit` or `Ctrl+D`	Exit the Pry session.
`help`	Display help information.

Advanced Pry Features

Pry supports command aliases, allowing you to create shortcuts for frequently used commands.

Example:

```
Pry.config.alias_command 'w', 'whereami'
```

Now you can use `w` instead of `whereami`.

Pry integrates well with other debugging tools like `pry-byebug` for step-by-step execution.

Debugging with Byebug

Byebug Basics

Byebug is a powerful debugger for Ruby that allows you to step through code, set breakpoints, and inspect variables.

Installation:

```
gem install byebug
```

To start debugging, insert `byebug` into your code where you want to pause execution.

Example:

```ruby
require 'byebug'

def my_method(arg)
  byebug # Execution pauses here
  puts arg
end

my_method('Hello, Byebug!')
```

Common Byebug Commands

`next` or `n`	Execute the next line of code.
`step` or `s`	Step into a method call.
`continue` or `c`	Continue execution until the next breakpoint or the end of the program.
`break <location>` or `b <location>`	Set a breakpoint at the specified location (e.g., `5`, `my_file.rb:10`).
`info`	Display information about the current state.
`display <expression>`	Automatically display the value of an expression each time the debugger stops.
`p <expression>`	Print the value of an expression.
`quit` or `q`	Exit the debugger.

Conditional Breakpoints

Byebug allows you to set breakpoints that are only triggered when a certain condition is met.

Example:

```ruby
break 10 if x > 5 # Break at line 10
only if x is greater than 5
```

Debugging Best Practices

General Tips

1. **Understand the Problem:** Before diving into debugging, make sure you fully understand the problem you're trying to solve. Reproduce the issue and identify the steps that lead to it.
2. **Write Tests:** Tests can help you isolate and reproduce bugs. Write unit tests to verify the behavior of individual components and integration tests to ensure that different parts of your application work together correctly.
3. **Use Version Control:** Regularly commit your code to version control. This allows you to easily revert to previous versions and compare changes to identify the source of bugs.

Debugging Workflow

1. **Start with Logging:** Add logging statements to track the flow of execution and the values of important variables.
2. **Use a Debugger:** When logging isn't enough, use a debugger like Byebug or Pry to step through the code and inspect the state of the application.
3. **Isolate the Issue:** Try to narrow down the source of the bug by commenting out code or simplifying the problem.
4. **Read Error Messages:** Pay close attention to error messages and backtraces. They often provide valuable clues about the cause of the problem.

Advanced Debugging Techniques

1. **Remote Debugging:** Debug code running on a remote server by connecting to the server with a debugger.
2. **Profiling:** Use profiling tools to identify performance bottlenecks in your code.
3. **Memory Analysis:** Analyze memory usage to detect and fix memory leaks.

Profiling and Performance Gems

`ruby-prof` A fast, accurate Ruby profiler, providing detailed performance reports for Ruby code.	Offers call stack, flat, and graph profiles to pinpoint bottlenecks.
`stackprof` A sampling call-stack profiler for Ruby, designed for speed and low overhead.	Captures stack samples to identify frequently called methods, helping optimize performance-critical sections.
`memory_profiler` A gem to profile memory usage in Ruby apps, identifying memory leaks and allocations.	Provides insights into object allocation, retention, and garbage collection behavior, crucial for memory optimization.
`benchmark` A standard library module for benchmarking Ruby code snippets.	Allows timing of code execution, comparing performance of different approaches, and identifying performance regressions.
`bullet` A gem to help increase your Rails application's performance by reducing the number of queries it makes.	Alerts you to N+1 queries, unused eager loading, and suggests solutions.
`rack-mini-profiler` A middleware that displays speed badge for every html page, showing overall load time.	Provides detailed information about request performance, including SQL queries, view rendering, and more, directly in the browser.
`derailed_benchmarks` A series of things you can use to benchmark different parts of your Rails or Ruby app.	Includes tools to measure memory usage, object allocations, and garbage collection performance.
`wrapped_print` My own gem to print values of objects to the console, without typing "puts" or "logger.debug".	Prints value of the object without modifying it. `user = find_user.wp` (this `.wp` will print the result of find_user method)