# Debugging Cheatsheet

If you're banging your head against the wall, some tips:

1. Find the top line that looks familiar. Put a debugger on that line. Print out everything that shows up on the line.
2. Google it
3. If there are lines of code that are really long, break them down into parts.
4. Trace back: "What method calls check\_valid\_move?? Where does cats get assigned? Let's put a debugger there." Keep working "backwards" towards the source.
5. Comment out lines of code until the error changes.
6. Go back to when you had it working and see what changed.
7. Reference the [learn how to debug](https://asherkingabramson.com/blog/productivity/learn-to-debug) blog post.

# Breaking Confusing Methods Into Parts

Consider this line of code:

exp(b, n) = b \* exp(b, n - 1)

It's math, so it's sort of readable. But exp, b and n are still terrible names for variables. They're too short.

Copy this line of code into a new text file. Starting with the first code that runs on the line, name each part of the line as descriptively as you can on a separate line. Then try walking through what's happening.

Start by rewriting it as:

exponent(base, power) = base \* exponent(base, power - 1)

Then, power - 1 runs first. So rewrite as:

smaller\_exponent = power - 1

exponent(base, power) = base \* exponent(base, smaller\_exponent)

Then:

smaller\_exponent = power - 1

base\_to\_the\_smaller\_exponent = exponent(base, smaller\_exponent)

exponent(base, power) = base \* base\_to\_the\_smaller\_exponent

Resulting in:

exponent(base, power) = base \* base\_to\_the\_smaller\_exponent

See how much clearer that last line is? Your code should read like Hemingway.

### Two important takeaways

1. We named the variables as descriptively as possible.
2. We broke complicated lines down into parts.

Hold yourself to these standards almost every time you write code.

### A Counterargument

You might ask: doesn't naming variables slow me down?

There are four major reasons that long variable names are much faster for you in the long run.

1. Over the coming weeks, you're about to get a lot better at typing quickly.
2. Your text editor (i.e., VS Code, Atom, Sublime Text) will autocomplete long variable names.
3. It's easier for someone else to read your code.
4. You reduce the amount of data in your [working memory](https://en.wikipedia.org/wiki/Working_memory), because you don't have to remember what abbrevations stand for what concepts. Less data in your working memory gives you more space to tackle new problems.

## Short Exercise

The method below estimates how much money your child will have when they retire, based on your age.

def return\_number(a)

((a / 2)\*\*2) \* (65 - (a / 2))

end

Sample Inputs and Outputs:

* An input of 30 should output 11250
* An input of 50 should output 25000

You know the algorithm involves these steps in some order:

* Calculate how much money they'll have by multiplying by the number of years they have left until retirement.
* Estimate your child's current bank account by squaring their age.
* Estimate your child's age by dividing your age by two.
* Calculate how many years your child has left until they retire at age 65

The method is written so poorly, it may as well be the script for Home Alone 4. There are one-letter variables and there's too much happening on one line.

Rewrite the method by identifying the first operations that run (HINT: find the innermost parentheses). Then copy these operations onto separate lines (like the "exponent" example above) and use descriptive variable names.

DO NOT try to write the algorithm from scratch. That is NOT the point of the exercise.

Finally, rewrite the name of the method itself. Methods that have a side-effect (modify state) should be named with verbs, e.g. #activate\_account or #reveal\_card. Methods that return a useful value should be named with nouns describing their output, e.g. #full\_name or #days\_since\_last\_login.

Test that the method still returns the right output by running the code in pry.

Then check with a TA and see if the naming makes sense to them. The code should almost look like english once you're done with it.

# Common Exceptions

There are several exceptions that frequently come up. They can be mysterious at first. Exceptions are trying to tell you what went wrong, so being able to understand them is important. Here are the most common.

## Goals

* Know these most common exceptions and what they signify.

## NameError

A NameError exception is thrown when you try to use a variable or method that hasn't been defined.

class NumbersClass

def answer\_number

42

end

def loneliest\_number

1

end

def numbers

[favorite\_nmber, loneliest\_number]

end

end

[3] pry(main)> nc = NumbersClass.new

=> #<NumbersClass:0x007ff9dc12c608>

[4] pry(main)> nc.numbers

NameError: undefined local variable or method `favorite\_nmber' for #<NumbersClass:0x007ff9dc12c608>

from: /Users/ruggeri/test.rb:11:in `numbers'

from: (pry):4:in `\_\_pry\_\_'

Here we've mistyped the method name; the error tells us the name favorite\_nmber hasn't been defined. Ruby looks for either a local variable or a method. It tells you that it was looking for the method favorite\_nmber defined on the NumbersClass.

### Uninitialized Constant

Here's a variation on your standard NameError:

[3] pry(main)> UnloadedClass.new

NameError: uninitialized constant UnloadedClass

from: (pry):3:in `\_\_pry\_\_'

"Uninitialized constant" means that Ruby is trying to find a class (or other constant), but didn't find it. This could be because of a class name typo, or often because we forgot to require the Ruby file that loads the class.

You'll get this variation if the name starts with an upper-case letter, since in Ruby variables/methods start with lower case letters, and classes start with upper case letters.

### NoMethodError

This is similar to NameError; in fact it's a subclass. This is thrown when it's clear the user wanted to call a method (didn't try to look up a variable) that doesn't exist:

[1] pry(main)> "my string".my\_fantasy\_method

NoMethodError: undefined method `my\_fantasy\_method' for "my string":String

from: (pry):1:in `\_\_pry\_\_'

Again, Ruby tells us the method name it looked up, and the object.

A particularly common error occurs when a variable is nil when we don't expect this

[5] pry(main)> my\_array[0] # what if my\_array == nil?

NoMethodError: undefined method `[]' for nil:NilClass

from: (pry):2:in `\_\_pry\_\_'

nil is an instance of the class NilClass, which doesn't have the method we want.

## ArgumentError: wrong number of arguments

If we don't give a method the right number of arguments, we will get an exception thrown at us:

[12] pry(main)> [1, 2, 3].shuffle("unwanted argument")

ArgumentError: wrong number of arguments (1 for 0)

from: (pry):9:in `shuffle'

Here we give the shuffle method an argument when it doesn't take one. Ruby throws an ArgumentError exception back at us; it tells us that we passed the wrong number of arguments. It even says that we passed 1 argument when 0 were expected.

## TypeError

A TypeError may be thrown if you pass the wrong type of thing to a method. For instance, numbers can only add other numbers:

[8] pry(main)> 2 + ""

TypeError: String can't be coerced into Integer

from: (pry):8:in `+'

Here, we try to add a String to a number (Integer is the standard integer class). The method + works by first trying to turn its argument into a Integer, then adding it. A String cannot be turned into a Integer (coerced), so the method complains.

This error normally occurs when you call a method with the wrong types of things. For instance:

> [] + ""

> [] + 2

> "" + 2

> "" + []

None of these operations make sense; all of them will throw a TypeError.

## LoadError

Load errors are common:

[2] pry(main)> require 'primes.rb'

LoadError: cannot load such file -- primes.rb

There are two very common causes. Sometimes you are trying to load a file that is provided by a gem, but you haven't installed the gem yet.

Another common cause is that you are trying to load another source file in your project, but you forgot the initial './'. Relative includes are used to include files that are inside your project, you write them like this:

[2] pry(main)> require './primes.rb'

Of course, the file can be be in a subdirectory:

[2] pry(main)> require './path/to/source/file/primes.rb'

## SyntaxError

Writing ungrammatical Ruby code will net you a SyntaxError. This tells you that Ruby didn't understand your code.

There are lots of sources of syntax errors. The most common are forgetting to close quotes, parentheses, or do-end blocks.

Consider a source file that fails to close a do block:

[1, 2, 3].each do |num|

puts num

# end should go here

When you load this source file, you'll get:

[1] pry(main)> require './test.rb'

SyntaxError: /Users/ruggeri/test.rb:3: syntax error, unexpected $end, expecting keyword\_end

from: /Users/ruggeri/.rvm/rubies/ruby-1.9.3-p194/lib/ruby/site\_ruby/1.9.1/rubygems/custom\_require.rb:36:in `require'

from: /Users/ruggeri/.rvm/rubies/ruby-1.9.3-p194/lib/ruby/site\_ruby/1.9.1/rubygems/custom\_require.rb:36:in `require'

from: (pry):1:in `\_\_pry\_\_'

Here $end means the end of the source file. Here Ruby is telling you that it didn't expect the end of the file ($end) before the keyword end (keyword\_end in the error message).

You can get a similar message with too many ends:

[1, 2, 3].each do |num|

puts num

end

end # one too many

[2] pry(main)> load './test.rb'

SyntaxError: /Users/ruggeri/test.rb:4: syntax error, unexpected keyword\_end, expecting $end

end # one too many

^

from: (pry):2:in `load'

This just reverses the prior message; we hit the keyword end when we were expecting the end of the file; that is, when we weren't expecting one.