**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.