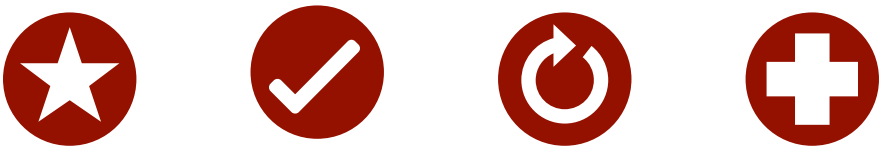
お試し版

技術書典6
(2019年春)

Ruby

のエラーメッセージが読み解けるようになる本

カウプラン機関極東支部



Kauplan Press

お試し版

▼ Rubyのエラーメッセージ：

undefined method `foo'

「未定義のメソッド `foo' 」
とだけ言われてもイミフ…

▼ この本での説明：

メソッド foo() が呼ばれたけど
見つからない、定義されてない

そういうことか!!

▼ 読者の声：

（Rubyのエラーメッセージが）

読める、読めるぞ!!

32歳 男性、職業：軍人

はじめに

この本は Ruby のエラーメッセージをひたすら解説した本です。また「初心者がエラーメッセージを理解できないなら、そのようなものを出すほうが悪い」ことを訴える本でもあります。

♣ この本の目的

この本の目的は、Ruby の初心者がエラーメッセージを読めるようになることです。具体的には次の 2 つです。

- エラーメッセージを見て、エラーの内容が理解できること
- エラーメッセージをヒントにして、エラーの原因を探れること

初心者の人は、エラーが出ると手も足も出なくなります。この本を読めば、そういった初心者の人でもエラーメッセージにひるまなくなります。

♣ なぜこの本を書いたのか

この本を書いた動機は、**エラーメッセージが分からないせいでプログラミングに挫折する人が多い**からです。そのような人の助けになるために、この本は存在します。

「エラーメッセージが分からない」とはどういうことでしょうか。いろんな意見があるでしょうが、要は次の 2 点だと思います。

エラーメッセージの内容が理解できない。

Ruby に限らず、**ソフトウェアが出すエラーメッセージは簡潔すぎる**ので、初心者にとっては内容がよく分かりません。

たとえば「`NoMethodError: undefined method ‘foo’`」という簡潔なエラーメッセージが出ても、初心者は「未定義のメソッド？ どういうこと？」となるばかりで、「未定義のメソッドが呼ばれた」ことだとは思いません。「`undefined method ‘foo’`」（未定義のメソッド）と「`method ‘foo()’ called but not defined`」（メソッド `foo()` が呼ばれたけど定義されてない）では分かりやすさに大きな違いがあります。残念ながら

Ruby のエラーメッセージは前者なので、初心者が理解できないのは当然なのです。

エラーメッセージを見てもエラーの原因が分からない。

仮にエラーメッセージの内容が分かったとしても、そこからエラーの原因を突き止めるのは、初心者には大変困難です。

たとえば「undefined method」というエラーメッセージが出たとしても、考えられる原因は 1 つではありません。メソッド名の間違いなのか、レシーバオブジェクトが nil だからなのか、「::」を「:」と間違えたことなのか等々、さまざまな可能性があります。これらを初心者に見分けろといっても、無理でしょう。

♣ この本の内容

そこでこの本では、次のような内容になっています。

- エラーメッセージを、初心者に分かる言葉で説明します。たとえば「undefined method `foo'」というエラーメッセージを、「未定義のメソッド `foo'」ではなく「メソッド foo() が呼ばれたけどそんなメソッドはなかった」と説明しています。これによって、**初心者でもエラーの内容が分かるようになります**。
- エラーの種類ごとに、考えられる原因を複数紹介しています。たとえば「undefined method `foo'」というエラーメッセージなら、foo() が未定義定義のケース、レシーバオブジェクトが nil のケース、「::」を「:」と間違ってるケース等々、考えられる原因をいくつも紹介しています。それだけでなく、エラーメッセージのどこに着目すべきかを説明しています。そのため、エラーメッセージを手がかりにして**初心者でもエラーの原因を探れるようになります**。

♣ 「エラーメッセージぐらい誰でも読めるだろ！」という勘違い

よく、中級者や上級者が初心者に対して「なんであいつらはエラーメッセージを読もうとしないのか。簡単な英語なんだからそれくらい読めよ！」と愚痴をこぼしたりバカにすることがあります。そういう人に遭遇するたび、本当にムカム

カします（~~全滅しないかな~~）。せっかくの機会なのではっきり言いますが、**そんなだからあなた達は初心者に教えるのに向いてないんです。**

初心者がエラーメッセージを読まないのではありません。**読んでもらえないようなエラーメッセージを出してるほうが悪い**のです。読まれないメッセージや伝わらないメッセージを出しておいて、「なんで読んでくれないかなー？」と言ってるほうがおかしいのです。そんな愚痴をこぼしたり初心者を非難する暇があったら、読んでもらえる工夫をすべきだし、初心者への配慮ができない自分たちの資質不足を反省すべきです。

とはいっても、ここまで広く普及した Ruby のエラーメッセージを今から初心者向けに大きく変えるのは難しいでしょう。技術的には可能だとしても、Matz や開発チームの理解を得る必要があるので、政治的に大変困難です。

それよりも、エラーメッセージの読み方や原因の探り方を初心者向けに解説するほうが、手軽かつ現実的な手段です。そしてそれがこの本なのです。

♣ 対象読者

この本は、Ruby の初心者を直接の対象としています。ここでいう「初心者」とは、Ruby の入門書を読んでいる途中の人、または入門書を一通り読み終えた（けどプログラムはまだうまく書けない）人を指しています[*1]。このレベルの人たちはエラーメッセージが読めないし、読めたとしても原因を探ることができないので、まさにこの本が役立つでしょう。

またこの本は、新人教育を担当する人も間接的な対象としています。まだプログラミングがおぼつかない新人に対し、教育担当者が「エラーメッセージぐらい読めよ」「この程度の英語も分からないのか」と心ない言葉を放つことがあります。そのような教育担当者に対し、新人がエラーメッセージを読めないのは英語のせいではないこと、また読んで理解できるようなエラーメッセージになってないことこそが問題だと気づいてもらうことも、本書の狙いです。

♣ この本に載ってないこと

この本は、Ruby におけるエラーメッセージの意味とその原因の探し方を説明した本です。次の内容については取り扱いません。

[*1] 本来なら、入門書を読み終えた人は初心者ではなく「初級者」と呼ぶべきですが、ここでは初級者も「初心者」に含めることにします。

- Ruby の文法の説明
- Ruby のプログラムをうまく書く方法
- Ruby でエラーを出さないような方法
- Ruby on Rails に関すること

これらについては、他の書籍をあたってください。

♣ ご購入される方への注意

あらかじめ告知しておきますが、**この本の内容は将来的に無料で公開される可能性があります**。というのも弊機関は、この本を基にして Ruby のエラーメッセージを初心者向けに改造するようなライブラリを作成・公開する予定だからです（公開時期は未定です）。その場合は、この本の内容が形を変えてそのライブラリに含まれます。

よって、今この本を購入したとして、もし同等の内容が無料で見られるようになったときに「騙された！ 返金しろ！」と言われても困るので、ここで事前に告知しておきます：**この本の内容は、将来的に無料で公開される可能性があります**（大事なことなので 2 回言いました）。

以上のことをご理解のうえご購入ください。

♣ 動作検証環境

この本の内容は、macOS Mojave 上で主に Ruby 2.5 を使って動作検証をしています。Ruby のバージョンによっては、あるいは OS が Windows や Linux の場合はエラーメッセージが一部異なることがありますが、この本の内容は十分通用するはずです。

♣ サポートサイト

次のサイトで、この本の正誤表を公開しています。

- https://kauplan.org/books/errmsg/

またご質問やご意見があれば、Twitter(@_kauplan) で受け付けます。

♣ 謝辞

今回は原稿の一部を@yagitch_tech 氏にレビューいただきました。あざす！

目次

《第 1 章》

エラーメッセージの基礎

この章では主に「エラーメッセージの構造」と「エラーに対する知識」について説明します。前者はエラーメッセージを読み解くために必要な知識であり、後者はエラーの原因を突き止めるのに必要な知識です。

どちらも大切な知識なので、初心者の人はしっかり理解してください。

【この章の内容】

1.1　エラーメッセージの構造

Ruby のエラーメッセージには、構造があります。

具体例で説明しましょう。たとえば次の例では、変数名が間違っているせいでエラーになっています。

▼ファイル「ex-error1.rb」

```
message = "Hello"
puts messege   # ← 変数名が間違っている
```

▼実行結果

```
$ ruby ex-error1.rb
Traceback (most recent call last):
ex-error1.rb:2:in `<main>': undefined local variable or method `me▸
ssege' for main:Object (NameError)
Did you mean?  message
```

この実行結果を詳しく見てみましょう。

- 最初の「$ ruby ex-error1.rb」は、ファイル「ex-error1.rb」を ruby コマンドで実行していることを表します。これはエラーメッセージの一部ではありません。
- 次の「Traceback」（トレースバック）とは、関数やメソッドがどこから呼び出されているかを表すデータのことです。また「(most recent call last):」は、直訳すると「いちばん最近の呼び出しが最後」であり、これは関数やメソッド呼び出しにおいて「古い呼び出しほど上、新しい呼び出しほど下に表示している」という意味です。トレースバックについては次の節で説明します。
- 次の「ex-error1.rb:2:」とは、ファイル「ex-error1.rb」の 2 行目でエラーが起こったことを表します。また「in ‘<main>’:」は、エラーが発生した関数やメソッドの名前を表示します。「<main>」というのは、関数やメソッドの中ではなかったことを意味します。
- そのあとの「undefined local variable or method ‘messege’ for

main:Object」がエラーメッセージの本体であり、直訳すると「main オブジェクトにとって ‘messege‘は未定義のローカル変数またはメソッド」となります。分かりくいですが、これは「‘messege’ という名前のローカル変数もメソッドもメソッドも見つからなかった」という意味です。

- 「for main:Object」は、「Object クラスの main オブジェクトにとっては」という意味です。「:」から前がオブジェクトを表し、後ろがそのクラスを表しています。
- 行末の「(NameError)」は、エラーの種類を表します（詳しくいうと「例外クラス名」といいます）。この例では変数名を間違えているので「NameError」というエラー（例外クラス）ですが、たとえば文法に間違いがあれば「SyntaxError」、文字コードに関するエラーなら「EncodingError」になります。
- 最後の「Did you mean? message」は、変数名やメソッド名を間違えたときに出てくるヒントであり、「もしかして、message じゃない？」という意味です（エラーの種類によっては出ません）。今回は「message」を「messege」と間違えたのが原因なので、まさにヒントの通りでした。

このように、**Ruby のエラーメッセージには構造があります**。これをきちんと教えてくれる入門書は少ないし、教えてくれるような気の利いた先輩もそうはいないでしょう。初心者の人は、エラーメッセージの構造を理解してください。

ヒントが出るのは Ruby 2.3 から

「Did you mean? message」のようなヒント（または Suggestion）が出るのは、Ruby 2.3 からの機能です。それより古いバージョンの Ruby を使っている場合は、「did_you_mean」という Gem をインストールするとヒントが出るようになります。

1.2 トレースバック

前の節で「トレースバック」という用語が出ました。改めて説明すると、トレースバック (Traceback)[*1] とは関数やメソッドがどこから呼び出されているかを表すデータです。

次の例を見てください。ここでは f1() が f2() を呼び出し、f2() が f3() を呼び出し、そして f3() の中で変数名を間違えているためエラーが発生します。

▼ファイル「ex-error2.rb」

```
def f1()
  f2()
end

def f2()
  f3()
end

def f3()
  message = "Hello"
  puts messege  # ← 変数名を間違えているのでエラー
end

f1()
```

▼実行結果（Ruby 2.5 以降の場合）

```
$ ruby ex-error2.rb
Traceback (most recent call last):
        3: from ex-error2.rb:14:in `<main>'
        2: from ex-error2.rb:2:in `f1'
        1: from ex-error2.rb:6:in `f2'
ex-error2.rb:11:in `f3': undefined local variable or method `messe▸
ge' for main:Object (NameError)
Did you mean?  message
```

[*1] バックトレース (Backtrace) やスタックトレース (Stack trace) ともいいます。

「Traceback」の行の下に、インデントされた 3 行がありますね。これがトレースバックです。「(most recent call last):」は直訳すると「いちばん最近の呼び出しが最後」であり、分かりやすく言い直すと「古い呼び出しほど上、新しい呼び出しほど下」という意味です。

トレースバックの 3 行を説明します。

- 「from ex-error2.rb:14:in ‘<main>’」は、ファイル「ex-error2.rb」の 14 行目でメソッドを呼び出したことを表しています。
- 「2: from ex-error2.rb:2:in ‘f1’」は、ファイル「ex-error2.rb」の 2 行目、メソッド f1() の中で呼び出したことを表しています。
- 「1: from ex-error2.rb:6:in ‘f2’」は、ファイル「ex-error2.rb」の 6 行目、メソッド f2() の中で呼び出したことを表しています。

このように、トレースバックをたどれば関数やメソッドがどこから呼び出されているかが分かります。

トレースバックの行には、どのメソッドを呼び出したかは書かれていないことに注意してください。たとえば「1: from ex-error2.rb:6:in ‘f2’」は、f2()「を」呼び出したのではなく f2()「から」呼び出したことを表します。

最後の「ex-error2.rb:11:in ‘f3’」はインデントされていませんが、実はこれもトレースバックの一部です。これはファイル「ex-error2.rb」の 11 行目、メソッド f3() の中でエラーが発生したことを表しています。

エラーメッセージ「undefined local variable or method ‘messege’ for main:Object」は前の節と同じなので説明は省略します。

Traceback? Backtrace? Stacktrace?

これまで説明したように、エラーメッセージには「Traceback」と出ます。しかし Ruby の例外オブジェクトには、例外発生時のトレースバックを返す「Exception#backtrace()」というメソッドがあります。また他のプログラミング言語では、「スタックトレース」(Stacktrace) という用語もよく使われます。

用語が統一されていないのが気になりますが、どれも同じものを表すと思って大丈夫です。

1.3 Ruby 2.4 以前と 2.5 以降の違い

前の節での実行例は Ruby 2.5 での結果でした。実は Ruby 2.4 以前のバージョンでは、エラーメッセージの表示が違います。

▼実行結果（Ruby 2.4 以前の場合）

```
$ ruby --version
ruby 2.4.5p335 (2018-10-18 revision 65137) [x86_64-darwin18]

$ ruby ex-error2.rb
ex-error2.rb:11:in `f3': undefined local variable or method `messe▸
ge' for main:Object (NameError)
Did you mean?  message
        from ex-error2.rb:6:in `f2'
        from ex-error2.rb:2:in `f1'
        from ex-error2.rb:14:in `<main>'
```

これを見ると、Ruby 2.5 のとき（2 ページ前の実行結果）と比べて次のような違いが分かります。

- エラーメッセージが表示される位置が違う。
  - Ruby 2.4 では最初に表示される。
  - Ruby 2.5 では最後に表示される。
- トレースバックの順番が違う。
  - Ruby 2.4 では新しい呼び出しほど上、古い呼び出しほど下。
  - Ruby 2.5 では新しい呼び出しほど下、古い呼び出しほど上。
- トレースバックに深さを表す番号の有無が違う。
  - Ruby 2.4 ではトレースバックの各行に番号がつかない。
  - Ruby 2.5 ではトレースバックの各行に番号がつく。

このように、Ruby 2.4 以前と 2.5 以降ではエラーメッセージの表示が違います。とはいっても実質的には同じ内容なので、どちらも読めるようになっておきましょう。

なお Ruby 2.5 以降でも、標準エラー出力 (stderr) を変更すると Ruby 2.4 と

同じような表示になります。

▼実行結果（Ruby 2.5 で標準エラー出力を変更した場合）

```
$ ruby ex-error2.rb 2>&1 | cat
ex-error2.rb:11:in `f3': undefined local variable or method `messe▸
ge' for main:Object (NameError)
Did you mean?  message
        from ex-error2.rb:6:in `f2'
        from ex-error2.rb:2:in `f1'
        from ex-error2.rb:14:in `<main>'
```

なぜ Ruby 2.5 でエラーの表示順序を変えたのか？

Ruby 2.4 までは、エラーメッセージを表示してからトレースバックを表示していました。そのためトレースバックが長くなると、エラーメッセージを読むために上にスクロールする必要がありました。そして、Ruby on Rails など最近の開発ではトレースバックが数十行の長さになることが普通であり、エラーのたびに上にスクロールする必要がありました。

これがとても面倒だという声が強くなったため、Ruby 2.5 からはエラーの表示順序を変更し、上にスクロールしなくてもエラーメッセージが読めるようになったのです。詳しくは https://bugs.ruby-lang.org/issues/8661 を読んでください。

ここまではエラーメッセージの構造を説明しました。これでエラーメッセージを読めるようになるはずです。

しかしエラーメッセージを読めるようになっただけでは、エラーに対する理解は浅いままです。エラーについての理解を深めないと、エラーの原因を探れるようにはなりません。

エラーへの理解を深めるために、次の節ではエラーの種類について説明します。

1.4 構文エラーと実行時エラー

Ruby プログラムを実行するとき、内部の処理は大きく 3 段階に分かれます。

(1) 構文解析：プログラムの構文を解析する。
(2) コード生成：解析結果をもとにバイトコードを生成する。
(3) コード実行：バイトコードを仮想マシンで実行する。

そして (1) の構文解析で起こるエラーを「構文エラー」(Syntax Error)、(3) のコード実行で起こるエラーを「実行時エラー」(Runtime Error) といいます[*2]。

- 構文エラーは、プログラムの構文に間違いがあるときに発生します。たとえば「def はあるのに end がない」「文字列が閉じていない」などです。
- 実行時エラーは、プログラムの構文に間違いはないけど実行してみたら何らかの問題があったときに発生します。たとえば「変数やメソッドが見つからない」「引数の数が合っていない」「ファイルが開けない」などです。

言い換えると、**実行する前に分かるエラーが構文エラー、実行してみて初めて分かるエラーが実行時エラー**と言えます。

構文エラーと実行時エラーを入試に例える

上で説明した内容を入試に例えると、出願の書類に不備があって願書が受け付けてもらえない（入試を受ける前に弾かれる）のが構文エラー、願書は受け付けてもらえたけど入試当日に遅刻したり受験票を忘れたり会場を間違えるのが実行時エラーに相当します。

なお入試で不合格になるのは、エラーではありません。判定式の実行結果が false になっただけです。

両者の違いを、サンプルプログラムで見てみましょう。
次のプログラムは、初期化してない変数を使っているせいでエラーになります。

[*2] (2) のコード生成ではエラーは起こらないはずです。もし発生したら Ruby 本体のバグ。

▼ファイル「ex-err1.rb」：初期化していない変数を使っているのでエラー

```
puts "abc"   # ← 問題なく実行される
puts abc     # ← 初期化してない変数を使っているのでエラー
```

▼実行結果

```
$ ruby ex-err1.rb
abc            ← puts "abc" の実行結果
Traceback (most recent call last):
ex-err1.rb:2:in `<main>': undefined local variable or method `abc'▸
 for main:Object (NameError)
```

実行結果を見ると、最初に「abc」と出力されているので「puts "abc"」が問題なく実行されていることが分かります。エラーはそのあとに発生しています。

つまりこのプログラムは、(1) の構文解析は成功していて、(3) のコード実行でエラーが出ていることが分かります（もし (1) の構文解析でエラーになっていたら「puts "abc"」も実行されないはず）。よって、このエラー (NameError) は実行時エラーです。

別の例を見てみましょう。次のプログラムは文字列が閉じていないせいでエラーになります。1 行目は前のプログラムとまったく同じであり、2 行目が違うことに注意してください。

▼ファイル「ex-err2.rb」：文字列が閉じていないせいでエラー

```
puts "abc"   # ← 問題なく閉じている
puts "xyz    # ← 文字列が閉じていない
```

▼実行結果

```
$ ruby ex-err2.rb
ex-err2.rb:2: unterminated string meets end of file
puts "xyz
        ^
```

実行結果を見ると、「abc」という出力がないので、プログラム 1 行目の「puts "abc"」は実行されていないことが分かります。

プログラムの 1 行目ですら実行されずにエラーが出たということは、このプロ

グラムは (1) の構文解析でエラーが起こったので (3) のコード実行に進めなかったということです。そのため、このエラーは構文エラーであると分かります。

このように Ruby のエラーは、(1) の構文解析で起こる構文エラーと、(3) のコード実行で起こる実行時エラーの 2 つに大別できます。この 2 つの区別がついていないと、エラーの原因を正しく探ることができません。初心者の人にとっては少々難しい話だと思いますが、きちんと理解するようにしてください。

正規表現のエラーはどっちのエラー？

正規表現に間違いがあれば、正規表現リテラルなら構文エラーが発生します。なぜなら、正規表現の間違いはプログラムの構文解析時に見つかるからです。

▼ファイル「ex-err3.rb」：正規表現の丸カッコが対応がとれていない

```
regexp = /((\d\d)-(\d\d)/
```

▼実行結果

```
$ ruby ex-err3.rb
ex-err3.rb:1: end pattern with unmatched parenthesis: /((\d\d)▸
-(\d\d)/
```

しかし正規表現を表す文字列をコンパイルすると、RegexpError という実行時エラーが発生します。なぜなら、実行してみないと正規表現の間違いが分からないからです。

▼ファイル「ex-err4.rb」：正規表現の丸カッコが対応がとれていない

```
regexp = Regexp.compile('((\d\d)-(\d\d)')
```

▼実行結果

```
$ ruby ex-err4.rb
Traceback (most recent call last):
        2: from ex-err4.rb:1:in `<main>'
        1: from ex-err4.rb:1:in `compile'
ex-err4.rb:1:in `initialize': end pattern with unmatched paren▸
thesis: /((dd)-(dd)/ (RegexpError)
```

このように、同じような原因でも構文エラーになったり実行時エラーになったりします。この 2 つの例を理解できれば、構文エラーと実行時エラーの区別がついているといえるでしょう。

1.5　構文エラーと実行時エラーの違い

構文エラー (Syntax Error) と実行時エラー (Runtime Error) の違いをもっと説明してみます。

エラーの種類

構文エラーは、1 種類しかありません。構文間違いの原因は様々ですが、エラーメッセージが違うだけで、エラーの種類としてはどれも同じです。なのでエラーの原因を調べるには、エラーメッセージが頼りです。

実行時エラーは、たくさんの種類があります。たとえば変数が見つからないのは NameError、メソッドが見つからないのは NoMethodError、引数の数が合わないのは ArgumentError、ライブラリが読み込めないのは LoadError、などです。なのでエラーの種類を見るだけで、原因をある程度推測できます[*3]。

`begin...rescue` で捕捉できるか

構文エラーは、プログラムの式や文を実行する前に発生します。エラーを捕捉するための `begin...rescue` は文なので、これを使っても構文エラーは捕捉できません。

実行時エラーは、`begin...rescue` で捕捉できます。そのための機能なので当然ですね。

エラー報告数

1 つのプログラムに複数の構文エラーが見つかった場合、Ruby はそれらをすべて報告します。そのため、複数のエラーメッセージが表示されることがあります[*4]。

実行時エラーは、(`begin...rescue` で捕捉しなければ) 最初に発生したエラーによってプログラムが終了します。複数のエラーメッセージが表示されることは通常はありません[*5]。

[*3] ただし**ある程度**推測できるだけです。詳しくは次の節で説明します。

[*4] 具体例は 2.8 節「正規表現リテラル中にエスケープなしで「/」」(p.37) を見てください。

[*5] ただし `begin...rescue` でエラー処理しているときに別のエラーが発生したときは、実行時エラーでも複数のエラーメッセージが出ることがあります。

1.6　構文エラーをチェックする方法

Ruby では、「-c」オプションを使うと構文エラーを簡単にチェックできます。このオプションをつけると、プログラムの構文を解析するけど実行はしません。

2 つ前の節で、Ruby 内部の処理は大きく 3 段階に分かれると説明しました。

(1) 構文解析：プログラムの構文を解析する。
(2) コード生成：解析結果をもとにバイトコードを生成する。
(3) コード実行：バイトコードを仮想マシンで実行する。

「-c」オプションを使うと、(1) の構文解析はするけど (2) のコード生成と (3) のコード実行はしません。そのため、プログラムに構文エラーがあるかどうかを調べるのに「-c」オプションが使えます。

例を見てみましょう。前の節で、構文エラーのあるプログラムを紹介しました。これを「-c」オプションつきで実行してみます。

▼構文エラーのあるプログラムを「-c」つきで実行

```
$ cat ex-err2.rb
puts "abc"
puts "xyz

$ ruby -c ex-err2.rb
ex-err2.rb:2: unterminated string meets end of file
puts "xyz
         ^
```

構文エラーが表示されましたね。これは「-c」なしで実行したときと同じエラーメッセージです。

続いて、実行時エラーを起こすプログラムを「-c」オプションつきで実行してみます。

▼実行時エラーを起こすプログラムを「-c」つきで実行

```
$ cat ex-err1.rb
puts "abc"
```

```
puts abc    # ← 初期化してない変数を使っているのでエラー

$ ruby -c ex-err1.rb
Syntax OK
```

「Syntax OK」と出力されました。これはプログラムの構文にエラーがないことを意味します。また出力に「abc」がないので、「puts "abc"」が実行されていないことも分かります。

またこのプログラムは、(前の節で見たように)「-c」なしで実行するとエラーが出ます。しかし「-c」つきだと実行がされないので、当然ですが実行エラーも出ません。

このように、「-c」オプションを使うとプログラムに構文エラーがないかチェックできます。

構文エラーと実行時エラーの区別がつかないと非効率

初心者の人が陥りやすいのが、構文エラーが発生しているのに実行時エラーを疑ってしまうことです。

どういうことかというと、構文エラーが発生しているのに「クラスやメソッドは定義されているか？」「変数は初期化しているか？」「ファイルは存在しているか？」などを疑って調査してしまうのです。あるいは、構文エラーが発生しているのに一生懸命 print デバッグをしようとするのです。こうなると正しい原因を突き止められないし、突き止めたとしても時間がかかって非効率です。

こうなってしまうのは、構文エラーと実行時エラーの区別がついていないせいです。両者の違いがちゃんと区別できるようになりましょう。

ここまでは、エラーに対する理解を深めるために構文エラーと実行時エラーについて説明しました。しかしエラーに対する理解を深めても、エラーの原因を突き止めるのは簡単ではありません。

次の節では、エラーの原因を突き止めるのに役立つ考え方を説明します。

1.7　エラーが同じでも原因は様々

ここからはエラーの原因を突き止めるのに役立つ考え方を紹介しますが、その前にたとえ話をしましょう。

皆さん、「腰痛」って経験ありますか？（おっさんくさい話ですまんな）。腰痛になる人は大勢いますが、その原因は人によって様々です。

- 姿勢が悪く、猫背である。
- 椅子に座るとき、足を組むくせがある。
- 昔からろくに運動をしておらず、腹筋が弱い。
- 昔から筋トレをしていて、腹筋運動をやりすぎた。
- ショルダーバッグをいつも右肩に掛けている。
- 歩き方が悪く、靴底の減り方が左右で違う。
- 枕が高く、自分の首に合っていない。
- etc

何が言いたいかというと、腰痛という「症状」に対して数多くの「原因」が考えられるということです。つまり、**ある「症状」とその考えられる「原因」は一対多の関係**です。

もちろん原因は人によって違うので、数多くの原因がすべてその人に当てはまるわけではありません。しかし「腰痛」という症状が同じでも、ある人は姿勢が原因、ある人は座り方が原因、またある人は歩き方が原因、というように、いろんな原因があり得るというのがポイントです。

賢明な読者の皆さんならもうお分かりでしょう。このことはプログラミングでも当てはまります。つまり、エラーという「症状」にも数多くの「原因」が考えられるのです。

初心者の人は「エラーの原因が異なればエラーメッセージも異なる」と考えがちですが、あいにくそうではありません。エラーメッセージが同じでも、エラーの原因はまったく異なることがあります。この事実を初心者の人は受け入れてください。これがエラーの原因を探るための第一歩です。

たとえば次の例では、それぞれ違う原因で NoMethodError が発生します。

▼ファイル「ex-error3.rb」

```
puts Regexp.mew('^$')  # メソッド名を間違えたせいで NoMethodError
```

▼ファイル「ex-error4.rb」

```
puts Regexp('^$')      #  「.new」を忘れたせいで NoMethodError
```

▼ファイル「ex-error5.rb」

```
retrun /^$/        #  「return」のつづりを間違えたせいで NoMethodError
```

▼実行結果

```
$ ruby ex-error3.rb
Traceback (most recent call last):
ex-error3.rb:1:in `<main>': undefined method `mew' for Regexp:Clas▸
s (NoMethodError)
Did you mean?  new

$ ruby ex-error4.rb
Traceback (most recent call last):
ex-error4.rb:1:in `<main>': undefined method `Regexp' for main:Obj▸
ect (NoMethodError)

$ ruby ex-error5.rb
Traceback (most recent call last):
ex-error5.rb:1:in `<main>': undefined method `retrun' for main:Obj▸
ect (NoMethodError)
```

実行結果を見ると、どれも似たようなエラーメッセージだし、エラーの種類（例外クラス）も NoMethodError で同じです。しかしエラーの原因はどれも違います。そのため、**エラーメッセージを見ただけで原因を特定することはできません**（推測はできます）。

だからこそ、エラーメッセージを手がかりにエラーの原因を突き止められるようになることが、初心者には必要なのです。

1.8　エラーは必ずしも原因を教えてはくれない

前の節で、腰痛になる原因は様々であることを説明しました。もう一度見てみましょう。

- 姿勢が悪く、猫背である。
- 椅子に座るとき、足を組むくせがある。
- 昔からろくに運動をしておらず、腹筋が弱い。
- 昔から筋トレをしていて、腹筋運動をやりすぎた。
- ショルダーバッグをいつも右肩に掛けている。
- 歩き方が悪く、靴底の減り方が左右で違う。
- 枕が高く、自分の首に合っていない。
- etc

これを見ると分かるように、**腰痛の原因は必ずしも腰にあるわけではありません**。他の部位に原因があることのほうが多いのです。

つまり、こういうことです：

- 腰の痛みは、必ずしも腰が原因であることを意味しない。
- 痛みの「症状」は腰だけど、その「原因」は別の部位にあることが多い。

賢明な読者の皆さんはもうお気づきでしょう。同じことがプログラミングにも言えます。つまり、エラーメッセージは必ずしも原因を意味しませんし、エラーという「症状」が発生した箇所とは別の箇所に「原因」があることが多いです。

ここでは 1 点目を説明します（2 点目は次の節で説明します）。

初心者の方には残念なことですが、**エラーメッセージはエラーの原因を正確に教えてくれるわけではありません**。教えてくれることもありますが、そういうケースは初心者の人が期待するほど多くはありません。

例として、前の節のサンプルコードと実行結果を再掲します。

▼ファイル「ex-error4.rb」

```
puts Regexp('^$')       #  「.new()」を忘れたせいで NoMethodError
```

▼実行結果

```
$ ruby ex-error4.rb
Traceback (most recent call last):
ex-error4.rb:1:in `<main>': undefined method `Regexp' for main:Obj▸
ect (NoMethodError)
```

エラーメッセージは「undefined method」（メソッドが未定義）ですが、実際の原因は「.new()」をつけ忘れたことです。エラーメッセージと原因が大きく違うことが分かります。

もうひとつ、前の節のサンプルコードと実行結果を再掲します。

▼ファイル「ex-error5.rb」

```
retrun /^$/       # 「return」のつづりを間違えたせいで NoMethodError
```

▼実行結果

```
$ ruby ex-error5.rb
Traceback (most recent call last):
ex-error5.rb:1:in `<main>': undefined method `retrun' for main:Obj▸
ect (NoMethodError)
```

エラーメッセージは「undefined method」（メソッドが未定義）ですが、実際の原因は「return」のつづりを間違えたことです。これもエラーメッセージと原因が大きく違います。

このように、エラーメッセージはエラーの原因を正確に表すわけではありません。エラーメッセージは原因を探るための手がかりでしかない、という認識をしてください。

1.9　エラーの発生箇所と原因は別の箇所

続いて、2 点目を説明します。

前の節で、痛みの「症状」は腰だけどその「原因」は別の部位にあることが多い、と説明しました。そのため腰痛の原因を見つけようと思ったら、腰だけでなく足や背骨や頭など全身を診る必要があります。

プログラミングでも、**エラーという「症状」が発生した箇所とは別の箇所に「原因」がある**ことが多いです。そのためエラーが発生した箇所だけを調べるのではなく、他の箇所も調べないと原因は突き止められません。

具体例を見てみましょう。次のはエラーの原因となった箇所と発生箇所が同じである例です。

- 3 行目で定数名を間違って参照しています（エラーの原因）。
- そのせいで、同じ 3 行目でエラーになります（エラーの発生箇所）。

▼ファイル「ex-error6.rb」

```
MESSAGE = "Hello"
def hello
  puts MESSEGE    # 定数名を間違えてるせいでエラー
end
hello()
```

▼実行結果

```
$ ruby ex-error6.rb
Traceback (most recent call last):
        1: from ex-error6.rb:5:in `<main>'
ex-error6.rb:3:in `hello': uninitialized constant MESSEGE (NameErr▸
or)
Did you mean?  MESSAGE
```

実行結果に「ex-error6.rb:3」とあるので、3 行目でエラーが発生したことがわかります。これは定数名を間違えた（つまりエラーの原因となった）行と同じです。このように、エラーの発生箇所と原因が同じ場所だと探しやすいです。

次に、エラーの発生箇所とは違う場所に原因がある例をお見せします。

- 1 行目の定数を設定する箇所で、定数名を間違えています（エラーの原因）。
- しかしここではエラーは発生せず、3 行目の定数名を参照する箇所でエラーになります（エラーの発生箇所）。

▼ファイル「ex-error7.rb」

```
MESSEGE = "Hello"   # 定数名を間違えてる（エラーの原因）
def hello
  puts MESSAGE      # 未定義の定数を参照してるのでエラー（発生箇所）
end
hello()
```

▼実行結果

```
$ ruby ex-error7.rb
Traceback (most recent call last):
        1: from ex-error7.rb:5:in `<main>'
ex-error7.rb:3:in `hello': uninitialized constant MESSAGE (NameErr▸
or)
Did you mean?  MESSEGE
```

実行結果に「ex-error7.rb:3」とあるので、3 行目でエラーが発生したことがわかります。しかしエラーの原因となったのは、1 行目で定数名を間違えて設定したせいです。つまりエラーの原因は 1 行目なのに、発生箇所は 3 行目となり、一致しません。このようにエラーの原因と発生箇所が分かれると、原因を探すのが難しくなります。

プログラムの行数がこのくらいならエラーの原因を見つけるのはまだ簡単です。しかし行数がもっと多くなったり、定数が別のファイルで設定されていると、初心者はエラーの原因を見つけられなくなります。なぜなら、初心者はエラーの発生箇所あたりしか見ないからです。そうではなく、エラーが発生したのとは別の箇所に原因があることはちっとも珍しくないので、もっと広い目を持って原因を探りましょう。

1.10　ここまでのまとめ

- エラーメッセージには構造がある。
  構造を知ればエラーメッセージは読み解ける。
- トレースバックはメソッドの呼び出し関係を表す。
  Ruby 2.5 以降なら新しい呼び出しほど下、Ruby 2.4 以前なら上。
- エラーは構文エラーと実行時エラーに大別できる。
  構文エラーは「-c」オプションでチェックできる。
- エラーメッセージが同じでも、原因は様々。
  エラーメッセージだけでは原因を特定できない。
- エラーメッセージは、エラーの原因を正確に表すわけではない。
  エラーメッセージは原因を探すための手がかりでしかない。
- エラーの原因は、エラーの発生箇所とは別の箇所にあることが多い。
  エラーが発生した箇所だけでなくもっと広い範囲を調べること。

《第 2 章》

構文エラー (SyntaxError)

構文エラー (SyntaxError) とは、プログラムの構文に間違いがあった場合に発生するエラーです。初心者が最初に遭遇するのは、恐らくこのエラーではないでしょうか。

この章では、構文エラーが発生する代表的な原因を紹介します。なお以降では、構文エラーを「SyntaxError」と表記しています。

【この章の内容】

2.1　「end」が足りない

♣概要

Ruby では「end」がたくさん登場します。この「end」が足りなくて対応がとれてないとき、SyntaxError が発生します。

♣例

たとえば次の例では、一見すると 6 行目の「end」は 1 行目の「def」と対応しているように見えますが、実は 2 行目の「if」と対応しています。そのため 1 行目の「def」と対応する「end」がなく、結果として SyntaxError になります。

▼ファイル「ex-lessend.rb」

```
def hello(name=nil)
  if name.nil?
    puts "Hello, World!"
  else
  puts "Hello, #{name}!"
end  # ← ここの「end」は、「def」ではなく「if」に対応している

hello("World")
```

▼実行結果

```
$ ruby ex-lessend.rb
ex-lessend.rb:8: syntax error, unexpected end-of-input, expecting ▸
keyword_end
hello("World")
              ^
```

♣解説

エラーメッセージの「syntax error, unexpected end-of-input, expecting keyword_end」というのは、「構文エラー、予期せず入力が終わった、期待していたのは end キーワード」という意味です。ここでの「end-of-input」というのはプログラムの終わりのことなので、もう少し分かりやすい言葉にすると「end が来るはずなのに、プログラムの終わりに来ちゃった

よ」ということです。エラーメッセージのあとに「hello("World")」と「^」が表示されてますよね？ これはプログラムの終わりだからです。

というわけで、この場合は「end」が足りないので「end」を追加してください。

ただしどこに追加するかは自分で判断する必要があります。今回の場合、Ruby は「def と対応する end がない」と判断しますが、実際には「if に対応する end がない」ことが原因なので、「if」に対応する「end」を追加してください。

Ruby の「-wc」オプションを使おう

Ruby に「-wc」オプションをつけると、対応する「end」がなければ警告メッセージを出してくれます。

- 「-w」オプション…警告 (Warming)[*1] を表示する。
- 「-c」オプション…構文のチェック (Check) だけ行い、実行はしない。

たとえば上のコードを「-wc」つきで実行すると、次のようになります。

▼実行結果（「-wc」つき）

```
$ ruby  -wc ex-lessend.rb
ex-lessend.rb:6: warning: mismatched indentations at 'end' wit
h 'if' at 2
ex-lessend.rb:8: syntax error, unexpected end-of-input, expect
ing keyword_end
hello("World")
              ^
```

「warning:」（警告）が出ていますね。その前に「ex-lessend.rb:6:」とあるので、6 行目に警告があることが分かります。そして「mismatched indentations at 'end' with 'if' at 2」というのは、「(6 行目の）'end' は、2 行目の'if' とインデントが一致しない」という意味です。

つまり **Ruby に「-wc」をつけるだけで、「end」の対応関係が揃ってないことをインデントで判断してくれるのです！**

今回のように**エラーメッセージに「keyword_end」が含まれている場合は「end」の不整合が発生している可能性が高い**ので、「ruby -wc」でチェックしてみてください。

[*1] 警告 (Warning) とは、エラーほど深刻ではないけど良くないことがあるよ、という意味です。天気予報でいえば「注意報」に相当すると思えばいいです。

2.3　他の言語のキーワードを使っている

♣概要

間違って他の言語のキーワードを使ってしまうと、SyntaxError になることがあります。

♣例

たとえば次の例は、関数を定義するのに「function」を使っています。JavaScript や PHP ならこれでいいのですが、Ruby では「def」を使うので、「function」は間違いです。

▼ファイル「ex-function.rb」

```
function add(x, y)  #「def」ではなく「function」を使っている
  return x + y
end
```

▼実行結果

```
$ ruby ex-function.rb
ex-function.rb:3: syntax error, unexpected keyword_end, expecting ▸
end-of-input
```

♣解説

エラーメッセージの「syntax error, unexpected keyword_end, expecting end-of-input」を直訳すると、「構文エラー、予期していない end キーワード (があった)、期待してるのは入力の終わり」となります。ここで「end-of-input」はプログラムの終わりを意味するので、エラーメッセージを言い換えると「プログラムの終わりのはずが end が来ちゃったよ」となります。

ここで、エラーになっているのが「end」であって、「function」ではないことに注意してください。この理由を説明します。

- 「function add(x, y)」は、「add(x,y)」の実行結果を「function」というメソッドに渡して呼び出している、と見なされるので、SyntaxError にはなりません。

- 次の「`return x + y`」は当然 SyntaxError になりません。
- そして最後の「`end`」は、対応する「`def`」や「`if`」や「`do`」がないため、SyntaxError になります。

しかし本当の原因は「`end`」ではなく、「`def`」と書くべきところを「`function`」と書いてしまったためです。「`def`」を「`function`」と間違えたせいで、「`end`」の不整合が発生しているわけですね。エラーメッセージは正確な原因を教えてくれないし[*3]、エラーの原因は発生箇所とは離れている[*4]ことがよく分かります。

今回の場合であれば「`function`」を「`def`」に修正すればエラーは直ります。しかし「`function`」が間違いであることに気づくためには、Ruby の文法に慣れておかないと難しいです。Ruby に十分慣れていれば「`function` じゃなくて `def` だ」とすぐ気づきますが、Ruby より JavaScript や PHP に慣れている人にとって「`function add(x, y); x + y; end`」は正しい構文に見えるので、間違いになかなか気づきません。

対策としては、予約語を色つきで表示する IDE やエディタを使うことです。そうすれば「`function`」に色がつかないので、間違いに気づきやすくなります。

ruby -wc を使っても end の不整合を見つけられないことがある

今回はエラーメッセージに「`unexpected keyword_end`」とあるので、「`end`」の不整合があることはすぐに分かります。しかし「`ruby -wc`」を使って確かめても「`end`」のミスマッチが見つかりません。

```
$ ruby -wc ex-function.rb
ex-function.rb:3: syntax error, unexpected keyword_end, expect▸
ing end-of-input
```

なぜなら今回は、余分な（つまり何にもマッチしない）「`end`」はあるけど、ミスマッチした「`end`」はないからです。「`end`」の不整合はたいてい「`ruby -wc`」で見つかりますが、必ずしも万能ではないことは知っておきましょう。

[*3] 1.8 節「エラーは必ずしも原因を教えてはくれない」(p.16) 参照。
[*4] 1.9 節「エラーの発生箇所と原因は別の箇所」(p.18) 参照。

2.5　予約語をローカル変数名に使っている

♣概要

予約語 (Reserved word) とはプログラムの中で特別な意味を持つ単語のことであり、Ruby ならたとえば「def」「class」「if」「while」「end」「case」があります。

そして予約語をローカル変数名[*7]に使っていると、SyntaxError になります。

♣例

たとえば次の例では、予約語である「next」をローカル変数名として使っているため、SyntaxError になります。

▼ファイル「ex-kwdvar.rb」

```
def link(n, max)
  prev = n > 0   ? n - 1 : nil
  next = n < max ? n + 1 : nil
  return prev, next
end
```

▼実行結果

```
$ ruby ex-kwdvar.rb
ex-kwdvar.rb:3: syntax error, unexpected '=', expecting keyword_en▸
d
  next = n < max ? n + 1 : nil
       ^
ex-kwdvar.rb:4: void value expression
  return prev, next
                   ^
```

♣解説

エラーメッセージの「syntax error, unexpected '=', expecting keyword_end」を直訳すると「構文エラー、予期してない'='、期待してたのは

[*7] 引数もローカル変数の一種であることに注意してください。つまり引数名には予約語を使えません。

end」となりますが、これを読んでも原因は分かりません。原因は予約語である「next」を変数名に使ったことなので、修正するには変数名を別の名前に変更します。

しかし、そもそも何が予約語なのかを初心者が把握しているわけがありません。対策としては、まず「ruby 予約語」でインターネット検索して出てくる Ruby マニュアルのページ[*8]を読んで、どんな予約語があるかを把握してください。

先ほどの Ruby マニュアルによると、Ruby の予約語は次の通りです。いきなり覚えるのは難しいので、まずは「next って予約語だったような気がする…」とうろ覚えだけでもしておきましょう。

▼ Ruby の予約語

```
BEGIN    class    ensure   nil      self     when
END      def      false    not      super    while
alias    defined? for      or       then     yield
and      do       if       redo     true     __LINE__
begin    else     in       rescue   undef    __FILE__
break    elsif    module   retry    unless   __ENCODING__
case     end      next     return   until
```

また IDE や高機能エディタを使うと、予約語に色がつきます。なので間違って予約語をローカル変数名に使うと気づきやすくなります。

「@next」や「$next」や「obj.next」は OK

先頭に「@」がつくインスタンス変数や、先頭に「$」がつくグローバル変数名では、予約語を名前に使っても SyntaxError になりません。たとえば「@next」というインスタンス変数や、「$next」というグローバル変数を使っても、何の問題もありません。

また初心者の人にとっては意外だと思いますが、予約語をメソッド名に使ってもエラーになりません。たとえば「def next」や「obj.next」はエラーになりません。なぜなら「def」や「.」の直後はメソッド名だと分かるからです。

[*8] https://docs.ruby-lang.org/ja/latest/doc/spec=2flexical.html#reserved

2.11　条件演算子のまわりに空白がない

♣ 概要

多くのプログラミング言語では「条件 ? 真のときの値 : 偽のときの値」のような式を書けます。この「?　:」を条件演算子 (Conditional Operator) または三項演算子 (Ternary Operator) といいます。

Ruby で条件演算子を使うとき、「?」や「:」のまわりには空白を入れる必要があります。空白を忘れると、SyntaxError になることがあります。

♣ 例

たとえば次の例では、「?」の前に空白がないせいで SyntaxError になります。

▼ ファイル「ex-condop1.rb」

```
def greater(x, y)
  x > y? x : y
end
```

▼ 実行結果

```
$ ruby ex-condop1.rb
ex-condop1.rb:2: syntax error, unexpected tIDENTIFIER, expecting k▸
eyword_do or '{' or '('
  x > y? x : y
         ^
```

♣ 解説

エラーメッセージの「`syntax error, unexpected tIDENTIFIER`」は「予期してない識別子[*19]がある」という意味です（が、原因と大きく離れています）。

Ruby ではメソッド名の最後に「?」をつけられます（たとえば「`Object#nil?`」や「`Array#empty?`」などが有名ですね）。そのため「`y?`」はメソッド名だと解釈されてしまい、変数「`y`」と演算子「?」の 2 つだとは認識されません。

条件演算子として認識させるには、「?」の前に空白をいれてください。

[*19] 識別子については 2.7 節「文字列リテラルが途中で閉じている」(p.35) を参照のこと。

2.12　条件演算子の「?」が抜けている

♣概要

条件演算子（三項演算子ともいう）は「条件式 ？　真のときの値 ：　偽のときの値」と書きます。このとき「?」が抜けていると、SyntaxError になります。

♣例

たとえば次の例では、条件式のメソッドが「.empty?」だったため、条件演算子を書いたつもりが「?」が抜けてしまい、そのせいで SyntaxError になります。

▼ファイル「ex-condop2.rb」

```
def f(arr)
  arr.nil? || arr.empty? "" : "-".join(arr)
end
```

▼実行結果

```
$ ruby ex-condop2.rb
ex-condop2.rb:2: syntax error, unexpected tSTRING_BEG, expecting k▸
eyword_end
  arr.nil? || arr.empty? "" : "-".join(arr)
                         ^
```

♣解説

エラーメッセージの「syntax error, unexpected tSTRING_BEG」は「予期せず文字列が開始してる」という意味です（が、原因と大きく離れています）。

Ruby ではメソッド名の最後に「?」をつけられます。そのせいで「x.nil?」や「x.empty?」を条件式に書くと、条件演算子の「?」をついつい書き忘れてしまいます。またコードを見たときも「x.nil?」や「x.empty?」の「?」があるせいで、条件式の「?」が抜けていることに気づかないことがあります。

これを修正するには、条件式のあとに「?」を追加してください。また条件演算子を使うときは、メソッド名末尾の「?」と条件演算子の「?」とを混同しないよう、注意してください。

《第 3 章》

NoMethodError

NoMethodError とは、呼び出したメソッドが存在しない場合に起こるエラーです。これだけ聞くと簡単なように感じるかもしれませんが、呼び出したメソッドが存在しない原因は多岐に渡るので、初心者がエラーの原因を突き止めるのは難しいです。

この章では、NoMethodError が発生する原因をたくさん紹介します。初心者の人にはとても役に立つでしょう。

【この章の内容】

3.4　レシーバオブジェクトが nil である

♣概要

レシーバオブジェクトが nil であることに気づかずにメソッド呼び出しを行うと、NoMethodError が発生します。

♣例

たとえば次の例では、変数の値が「nil」であるせいで、「.length()」メソッドの呼び出しで NoMethodError が発生します。

▼ファイル「ex-null1.rb」

```
val = nil
puts val.length()   # nil に対してメソッドを呼び出している
```

▼実行結果

```
$ ruby ex-null1.rb
Traceback (most recent call last):
ex-null1.rb:2:in `<main>': undefined method `length' for nil:NilCl▸
ass (NoMethodError)
```

♣解説

エラーメッセージの「undefined method ‘length’ for nil:NilClass (NoMethodError)」というのは、「NilClass のオブジェクトである nil には length() というメソッドは定義されていない」という意味です。

ここで「NilClass」というのは、nil のクラスです。Ruby では nil もオブジェクトであり、そのクラスが NilClass です。確認してみましょう。

▼nil のクラスが NilClass であることを確認

```
$ irb
irb(main):001:0> nil.class
=> NilClass
```

よって今回の場合、エラーメッセージの「for nil:NilClass」が大きなヒント

になって、オブジェクトが nil であることが分かります。なので、「if val != nil」を追加するなどして、nil の場合はメソッド呼び出しをしないように修正してください。

エラーメッセージを上書き

ここで説明した「オブジェクトが nil である」というのは、NoMethodError が発生する（おそらく）いちばん多い原因です。しかし「メソッドが定義されていない」というエラーメッセージから、「オブジェクトが nil である」という原因に気づくのは、（たとえ「for nil:NilClass」というヒントがあっても）初心者には難しいです。

そのため、オブジェクトが nil の場合の NoMethodError では、nil であることを直接表すようなエラーメッセージが望ましいです。次のようにするとそれが実現できます。

▼nil のときのエラーメッセージをより分かりやすく

```
NoMethodError.class_eval do
  def message   ## Ruby >= 2.3
    return super unless nil == self.receiver
    "receiver object is nil (method: `#{self.name}')"
  end
end

nil.foo()
  #=> NoMethodError: receiver object is nil (method: `foo')
"str".foo()
  #=> NoMethodError: undefined method `foo' for "str":String
```

もしこれを読んでいる Ruby コアチームの人がいたら、ぜひご一考ください。

3.7 初期化してないローカル変数を参照している

♣概要

Ruby では、初期化していないローカル変数にアクセスできることがあります。そのようなときはローカル変数の値が nil になるので、メソッド呼び出しをすると NoMethodError になります。

♣例

たとえば次の例では、ローカル変数「x」が初期化されずに使われています。この場合は「x」の値は nil になるので、「x + 1」が NoMethodError になります。

▼ファイル「ex-uninitvar1.rb」

```
x = x + 1
puts x
```

▼実行結果

```
$ ruby ex-uninitvar1.rb
Traceback (most recent call last):
ex-uninitvar1.rb:1:in `<main>': undefined method `+' for nil:NilCl▸
ass (NoMethodError)
```

♣解説

Ruby では通常、初期化していないローカル変数を使うと NameError になります（詳しくは 4.2 節「ローカル変数を初期化せずに使っている」(p.77) を参照してください）。しかし明示的な初期化をしなくても、ローカル変数が自動的に「nil」に初期化される場合があり（詳細は後述）、今回がまさにそのようなケースです。

今回の場合、ローカル変数「x」は明示的には初期化されていませんが、自動的に nil で初期化されます。そのため「x + 1」は「nil + 1」となり、nil には「+」演算子が定義されてないので NoMethodError になります[*4]。

このような場合は、ローカル変数をきちんと初期化してください。

[*4] Ruby では演算子もメソッドとして扱われます。詳しくは 3.6 節「グローバル変数名を間違えてる」(p.58) を参照してください。

初期化していないローカル変数にアクセスできるとき

Ruby では、初期化していないローカル変数にアクセスできることがあります。次の例を見てください。

```
def f
  if false  # if false なので、
    x = 10  # この代入文は絶対に実行されない。
  end
  puts x    # なのに代入してない変数へのアクセスがエラーにならない
end

f()  #=> nil
```

この例では「x = 10」が実行されることはありません（「if false」があるため）。言い換えると、このローカル変数は初期化されません。しかしそのあとで「puts x」のようにアクセスしても、エラーになりません[*5]。

つまりローカル変数を初期化しなくても、「これはローカル変数である」と Ruby に認識させることができれば、アクセスしてもエラーにならないのです。そしてそのように認識させるのが代入文であり、実際に実行されたかどうかは関係ありません。

先の例で「x = x + 1」が NameError にならないのも、代入文のおかげで「x」がローカル変数であることを Ruby が認識できたからです。そのため「x」へアクセスしても NameError にならないのです（かわりに nil となるため NoMethodError が発生するのは、すでに説明した通りです）。

[*5] 他のプログラミング言語でもこうなるとは限りません。たとえば Python なら、初期化してないローカル変数へアクセスすると必ずエラーになります。

3.11　オブジェクト生成時に「.new」を忘れた

♣概要

オブジェクトを生成するときに、クラス名に「.new」をつけ忘れると NoMethodError になります。

♣例

たとえば次の例では、「Time.new()」とするはずが「Time()」としてしまったため、NoMethodError になります。

▼ファイル「ex-nonew.rb」

```
t = Time(2019, 4, 14, 11, 0, 0)  #  「.new」がない
p t
```

▼実行結果

```
$ ruby ex-nonew.rb
Traceback (most recent call last):
ex-nonew.rb:1:in `<main>': undefined method `Time' for main:Object
 (NoMethodError)
```

♣解説

Ruby では、大文字始まりの名前は定数名と見なされますが、引数があればメソッド名だと見なされます。そのため、「Time()」は定数名ではなくメソッド呼び出しだと解釈されます。しかしそのようなメソッドは見つからないので、NoMethodError が発生します。

またエラーメッセージが「undefined method ‘Time’ 」なので、クラス名のはずの「Time」がメソッドと見なされていることが分かります。これに気づけは、今回の原因にもすぐ気づくでしょう。

♣補足

実は Python のような言語だと、オブジェクトを生成するのが「Time()」だけで済み、「Time.new()」や「new Time()」のようにする必要がありません。これに慣れていると今回のエラーを起こしてしまいがちです。注意してください。

3.12　「::」を「:」と書いてしまった

♣概要

「::」を間違えて「:」と書くと、NoMethodError になることがあります。

♣例

たとえば「Process::Status」を間違えて「Process:Status」と書いてしまうと、NoMethodError になります。

▼ファイル「ex-scolon1.rb」

```
x = Process:Status   # 「::」を「:」にしている
p x
```

▼実行結果

```
$ ruby ex-scolon1.rb
Traceback (most recent call last):
ex-scolon1.rb:1:in `<main>': undefined method `Process' for main:O▸
bject (NoMethodError)
Did you mean?  proc
```

♣解説

Ruby では「:Status」は Symbol を表すため、「Process:Status」は「Process(:Status)」というメソッド呼び出しだと解釈されます。そして「Process()」というメソッドが見つからず、NoMethodError が発生します。

「Process:Status」を構文エラーにしてほしい

「メソッドが定義されていない」というエラーメッセージから、「::」を間違って「:」と書いていることに気づける人は、初心者にはなかなかいないでしょう。

できれば Ruby の仕様を変更して、メソッド名と引数の間に「(」も空白もない場合は構文エラーにしてくれると、「Process:Status」のような間違いはすぐ気づけるようになるので、初心者には分かりやすくなります。

《第 4 章》

NameError

NameError は、名前に関する例外です。ここで「名前」とは、変数名だったりクラス名だったりします。主には変数名やクラス名を間違えたときに発生しますが、別の原因で発生することもよくあります。

ここでは、NameError が発生する原因を紹介します。

【この章の内容】

4.2 ローカル変数を初期化せずに使っている

♣概要

ローカル変数を初期化せずに使うと、NameError が発生します。

♣例

たとえば次の例では、ローカル変数「x」や「y」を初期化せずに使っているため、NameError が発生します。

▼ファイル「ex-notinit1.rb」

```
puts x + y
```

▼実行結果

```
$ ruby ex-notinit1.rb
Traceback (most recent call last):
ex-notinit1.rb:1:in `<main>': undefined local variable or method `▸
x' for main:Object (NameError)
```

♣解説

エラーメッセージの「undefined local variable or method ‘x’」というのは、「‘x’ というローカル変数またはメソッドが定義されていない」という意味です。このような場合は、ローカル変数をちゃんと初期化してください。

♣補足

Ruby では、初期化していないローカル変数にアクセスしても NameError が発生しないことがあります。これについては 3.7 節「初期化してないローカル変数を参照している」(p.60) を参照してください。

▼初期化していないローカル変数にアクセスしてもエラーにならない例

```
if false     # ここが if false なので、
  v = "ok"   # ここは実行されない（つまり、v は初期化されない）
end
puts v       # けれどアクセスできる（結果は「nil」が出力される）
```

4.4　クラス名を間違えてる

♣概要

クラス名を間違えると、NameError が発生します。

♣例

たとえば次の例では、「Hello」クラスを使うつもりが「Hallo」クラスと書いてしまったため、NameError が発生します。

▼ファイル「ex-wrongcls1.rb」

```
class Hello
end
p Hallo.new()    # クラス名を間違えているのでエラー
```

▼実行結果

```
$ ruby ex-wrongcls1.rb
Traceback (most recent call last):
ex-wrongcls1.rb:3:in `<main>': uninitialized constant Hallo (NameE▸
rror)
Did you mean?  Hello
```

また次の例では、「Hello」クラスのつもりが間違えて「Hallo」クラスを定義したため、「Hello」クラスを使うと NameError になります。

▼ファイル「ex-wrongcls2.rb」

```
class Hallo      # クラス名を間違えて定義している
end
p Hello.new()    # そのせいで、ここでエラー
```

▼実行結果

```
$ ruby ex-wrongcls2.rb
Traceback (most recent call last):
ex-wrongcls2.rb:3:in `<main>': uninitialized constant Hello (NameE▸
rror)
Did you mean?  Hallo
```

♣ 解説

1 番目のエラーメッセージ「uninitialized constant Hallo」というのは、「Hallo という定数が初期化されていない」という意味です。「uninitialized **constant**」（初期化されてない**定数**）となってますが、原因はクラス名を間違っていることなので、注意してください。

2 番目のエラーメッセージも同じような意味です。エラーメッセージに「ex-wrongcls2.rb:3」とあるので、エラーが発生したのは 3 行目です。しかし本当の原因は、1 行目のクラス定義でクラス名を間違ったことです。初心者の人にとっては気づきにくいので注意です。また「Did you mean? Hallo」というヒントに間違ったクラス名が出ていることにも気づけるようになりましょう。

クラスと定数

クラス名の間違いが原因なのにエラーメッセージが「uninitialized **constant**」（初期化されてない**定数**）になっているのは、初心者には分かりにくいですよね。

実は Ruby では、たとえば「Hello クラスを定義する」というのは、「新しいクラスオブジェクトを作って、定数 Hello に設定する」という動作になります。

▼「クラスを定義する」＝「クラスオブジェクトを定数に設定する」

```
## これは
class Hello
  def hi; puts "Hi!"; end
end

## これとだいたい同じ
Hello = Class.new do
  def hi; puts "Hi!"; end
end
```

そのため、「クラス名を間違う」ことは「定数を間違う」ことと同じなので、クラス名を間違えたときのエラーメッセージが「uninitialized constant Xxxx」になるのです。

Ruby の仕組み上、このようなエラーメッセージになってしまうのは仕方ないのですが、初心者のことを思うと「unknown class ‘Xxxx’」のような分かりやすいエラーメッセージが出てほしいと思います。

4.5 予約語のつづりを間違えている

♣概要

予約語のつづりを間違えると、NameError が発生することがあります。

予約語（キーワードともいう）とはプログラム中で特別な意味を持つ単語のことで、具体的には「def」や「class」や「if」や「end」などです。

♣例

たとえば次の例では、「true」を間違えて「ture」とタイプミスしたため、初期化していないローカル変数と見なされ、NameError になります。

▼ファイル「ex-wrongkwd1.rb」

```
result = 1 + 1 == 2 ? ture : false
puts result
```

▼実行結果

```
$ ruby ex-wrongkwd1.rb
Traceback (most recent call last):
ex-wrongkwd1.rb:1:in `<main>': undefined local variable or method ▸
`ture' for main:Object (NameError)
Did you mean?  true
```

♣解説

エラーメッセージを見ると「undefined local variable or method ‘ture’」となっています。これは「ture」が予約語とは見なされず、未定義のローカル変数またはメソッドと見なされているためです。

このような場合は、タイプミスを修正してください。また「Did you mean? true」というヒントが出ているので、これも見逃さないようにしましょう。

♣補足

なお予約語を間違えた場合には、NameError ではなく NoMethodError が発生することもあります。詳しくは 3.9 節「予約語のつづりを間違えている」(p.63)を参照してください。

4.9　「::」を「:」と書いてしまった

♣ 概要

「::」を間違えて「:」と書くと、NameError になることがあります。

♣ 例

たとえば次の例では、「Process::Status」と書くべきところを間違えて「Process:Status」と書いたせいで、NameError が発生します。

▼ ファイル「ex-scolon2.rb」

```
puts Process:Status   # 正しくは「Process::Status」
```

▼ 実行結果

```
$ ruby ex-scolon2.rb
Traceback (most recent call last):
ex-scolon2.rb:1:in `<main>': uninitialized constant Status (NameEr▸
ror)
```

♣ 解説

例にあげた「puts Process:Status」の場合、「puts」がメソッド名で「Process:Status」がキーワード引数[*7]だと解釈されます。よって「Process」が引数の名前、「Status」が引数の値と見なされます。しかし「Status」という定数が見つからないので、NameError が発生します。

このエラーで難しいのは、「Status」という定数はないけど「Process::Status」はあるという点です。そのため「uninitialized constant Status」と言われたら、定数がうまく参照できないせいだと思い込んでも仕方なく、まさか「::」を「:」と間違えたことが原因とは思わないでしょう。注意してください。

♣ 補足

原因が同じでも、NameError ではなく NoMethodError が発生することがあります。詳しくは 3.12 節「「::」を「:」と書いてしまった」(p.69) を参照のこと。

[*7] キーワード引数とは、メソッド呼び出し時に引数の値だけでなく名前も指定する機能です。詳しくはインターネットで検索してください。

《第 5 章》

TypeError

TypeError は、データ型に関するエラーを表します。たとえば演算子の両辺が異なるデータ型だったり、期待したデータ型ではなかった場合に発生します。

この章では、TypeError が起こる原因について紹介します。

【この章の内容】

5.2　できない演算をしようとしている

♣概要

同じデータ型どうしでの演算でも、できる演算とできない演算があります。そしてできない演算をしようとすると、TypeError になることがあります。

♣例

次のように、2 つの Time オブジェクトを足すと TypeError になります。

▼ファイル「ex-addtime.rb」

```
t1 = Time.new(2019, 4, 14, 11, 0, 0)  # 2019-04-14 11:00:00
t2 = Time.new(2019, 4, 14, 17, 0, 0)  # 2019-04-14 17:00:00
puts t2 - t1   # 差分は秒数
puts t2 + t1   #=> TypeError
```

▼実行結果

```
$ ruby ex-addtime.rb
21600.0
Traceback (most recent call last):
        1: from ex-addtime.rb:4:in `<main>'
ex-addtime.rb:4:in `+': time + time? (TypeError)
```

♣解説

Time オブジェクトは日時を表します。ある日時から別の日時を引くと、差分となる日数や秒数が求まります。上の例では「21600.0」という出力がありますよね？ これは 2 つの日時の差が 21600 秒だったことを表しています。つまり日時どうしの引き算は意味があります。

しかし日時どうしを足しても、意味のある結果にはなりません。そのため、日時どうしを足すと TypeError になります。これは Ruby に限らず、他のプログラミング言語でも同様です。

前の節では、異なるデータ型を使った演算がエラーになると説明しました。だからといって同じデータ型同士ならエラーにならないかというと、そういうわけではありません。難しいですね。

《第 6 章》

ArgumentError

ArgumentError は、引数に関するエラーを表します。たとえば引数の数が合っていない、引数の値がおかしい、などのときに発生します。

この章では、ArgumenetError が起こる原因を紹介します。

【この章の内容】

6.4　nil を含む配列をソートした

♣概要

nil を含むような配列をソートすると、ArgumentError になります。

♣例

次の例では配列に nil が含まれているため、ソートすると ArgumentError になります。

▼ファイル「ex-argerr7.rb」

```
arr = [5, 8, nil, 3, 9]
arr.sort()
```

▼実行結果

```
$ ruby ex-argerr7.rb
Traceback (most recent call last):
        1: from ex-argerr7.rb:2:in `<main>'
ex-argerr7.rb:2:in `sort': comparison of Integer with nil failed (▸
ArgumentError)
```

♣解説

上のエラーメッセージを直訳すると「Integer と nil との比較が失敗」となります。分かりやすく言い直すと、「整数と nil との比較はできない」です。

配列をソートするには、要素ごとに値を比較する必要があります。また前の節で説明したように、nil と何かを比較することはできません（比較するとArgumentError になります）。そのため、nil を含んでいる配列をソートするとArgumentError が発生します。

これを修正するには、配列から nil を取り除いてからソートしてください。または、Array#sort_by() を使って nil の代替値を指定してもいいでしょう。

▼要素が nil なら別の値を使ってソートする

```
arr = [5, 8, nil, 3, 9]
p arr.sort_by {|x| x.nil? ? 99999 : x }   #=> [3, 5, 8, 9, nil]
```

《第 7 章》

その他のエラー

この章では、その他のエラーについて紹介します。すべてのエラーを紹介するのは無理なので、代表的なエラーを紹介します。

【この章の内容】

7.6 IndexError

♣概要

IndexError は、範囲外の添字を配列に指定すると発生するエラーです。

♣例

たとえば次のように配列に対してマイナスの添字を使ったとき、範囲外への代入は IndexError になります。

▼ファイル「ex-indexerr1.rb」

```
arr = [10, 20, 30]
arr[-1] = 888    # arr[arr.length-1] = 888 と同じ
arr[-3] = 777    # arr[arr.length-3] = 777 と同じ
p arr            #=> [777, 20, 888]
arr[-4] = 666    #=> IndexError
```

▼実行結果

```
$ ruby ex-indexerr1.rb
[777, 20, 888]
Traceback (most recent call last):
ex-indexerr1.rb:5:in `<main>': index -4 too small for array; minim▸
um: -3 (IndexError)
```

また次のように、配列から値を取り出すときに Array#[] ではなく Array#fetch() を使うと、添字が範囲外なら IndexError になります。

▼ファイル「ex-indexerr2.rb」

```
arr = ["A", "B"]
p arr.fetch(0)   #=> "A"
p arr.fetch(1)   #=> "B"
p arr.fetch(2)   #=> IndexError
```

▼実行結果

```
$ ruby ex-indexerr2.rb
"A"
```

```
"B"
Traceback (most recent call last):
        1: from ex-indexerr2.rb:4:in `<main>'
ex-indexerr2.rb:4:in `fetch': index 2 outside of array bounds: -2.▸
..2 (IndexError)
```

♣解説

1 番目のエラーメッセージ「index -4 too small for array; minimum: -3」を直訳すると、「添字の -4 は配列に対して小さすぎる; 最小は -3」という意味です。分かりやすいエラーメッセージですね。

2 番目のエラーメッセージ「index 2 outside of array bounds: -2...2」を直訳すると、「添字の 2 は配列の範囲外：(範囲は) -2...2」となります（ここでの「bound」は「境界」や「限界」や「領域」といった意味です)。

ここで、エラーメッセージの「-2...2」に注目してください。これが「-2..2」(ピリオドが 2 つ) だと「-2 から 2 まで」という意味ですが、「-2...2」(ピリオドが 3 つ) だと「-2 から 2 **の 1 つ手前**まで」という意味になります。

つまり、この配列の境界 (添字の範囲) は -2 から 1 までなので、「arr.fetch(2)」は IndexError となったのでした。

IndexError のかわりにデフォルト値

Array#fetch() の第 2 引数にデフォルト値を指定できます。そうすると、添字が範囲外のときはデフォルト値が返され、IndexError にはなりません。

▼ファイル「ex-indexerr3.rb」

```
arr = ["A", "B"]
#p arr.fetch(2)       #=> IndexError
p arr.fetch(2, "X")   #=> "X"
```

▼実行結果

```
$ ruby ex-indexerr3.rb
"X"
```

同様に、Hash#fetch() も第 2 引数にデフォルト値を指定できます。

7.12 UncaughtThrowError

♣概要

UncaughtThrowError は、throw したけど catch していないときに発生します。たいていは raise を間違って throw と書いてしまったことが原因です。

♣例

次の例は、raise と書くべきのを間違って throw と書いてしまったために、catch されずに UncaughtThrowError が発生します。

▼ファイル「ex-throw1.rb」

```
def f()
  throw "xxxx"    # raise するつもりが throw になってる
end
f()
```

▼実行結果

```
$ ruby ex-throw1.rb
Traceback (most recent call last):
        2: from ex-throw1.rb:4:in `<main>'
        1: from ex-throw1.rb:2:in `f'
ex-throw1.rb:2:in `throw': uncaught throw "xxxx" (UncaughtThrowErr▸
or)
```

♣解説

上のエラーメッセージは「throw "xxxx"が発生しているけど catch されていない」という意味です。

Ruby では、raise でエラーを発生させて rescue で捕捉しますが、それとよく似た機能として throw() と catch() というグローバル関数があります。これは入れ子になったループから脱出（いわゆる大域脱出）するときに使用します。詳しい説明は省略するので、マニュアルやインターネットで検索してください。

これが発生するのは、たいていは Java や PHP ユーザが間違えて throw を使ったことが原因なので、raise を使うよう修正してください。

Ruby のエラーメッセージが読み解けるようになる本

2019-04-14　ver 1.0 発行（技術書典 6）

著者　カウプラン機関極東支部
サポート　https://kauplan.org/books/errmsg/
Twitter　@__kauplan
発行所　Kauplan Press