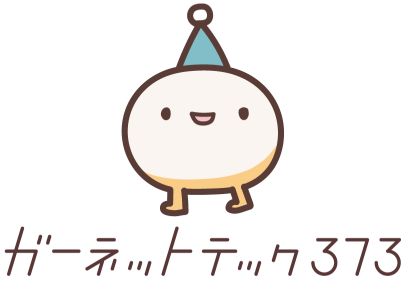
Rubyを学ぶポイントとコツ

五十嵐邦明 / igaiga

2025/03/05 Ruby Association Rubyセミナー Online

自己紹介

- 五十嵐邦明(igaiga) ガーネットテック373株式会社 代表取締役
- フリーランスのRailsエンジニア
- プログラミングスクール「フィヨルドブートキャンプ」顧問
- https://x.com/igaiga555
- 著書
  - ゼロからわかる Ruby超入門 ⬅️ 今日たくさん引用します
  - Railsの練習帳 ⬅️ 今日たくさん引用します
  - Railsの教科書
  - パーフェクトRuby on Rails［増補改訂版］
  - RubyとRailsの学習ガイド ほか
- 謝辞: 応援してくれている妻、子(1歳)、家族に感謝します



今日の方針

- 想定聴講者: これから新しくRubyで開発をされる方々
- 網羅的な説明ではなく、要所を選んで説明します
  - 体系立てた説明は「ゼロからわかるRuby超入門」をどうぞ（宣伝）
- 基礎的な内容で構成しています
- ちょっと難しめの話も混ぜています
  - 難しいところはパスして、しばらくしてからまた読んでもらえると嬉しいです
- 基本的にはRubyの範囲でお話をします
  - Railsの話も関係ある話題では軽く説明します
- 最後に今後の学び方の話もします

エラーメッセージの読み方

- エラーが出ると悲しい気持ちになるかもしれません
- でも、エラーメッセージは**Rubyから私たちへの思いやりあるヒント**です
- ここではエラーメッセージの読み方を解説します

エラーメッセージの読み方

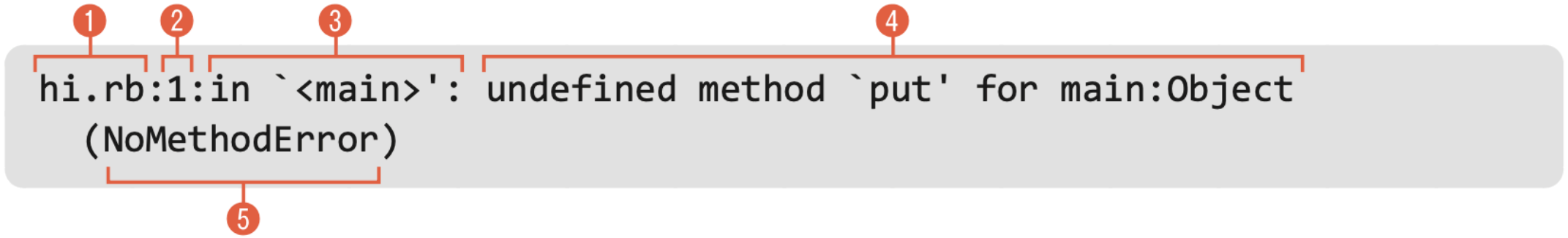
例: puts メソッドを間違えてputと書いてしまったためエラーになった

ソースコード

```
put "hi"
```

エラーメッセージ

```
$ ruby hi.rb
hi.rb:1:in '<main>': undefined method 'put' for main
 (NoMethodError)

put "hi"
^^^
Did you mean?  puts
               putc
```



- (1) `hi.rb` : 実行したファイル名
- (2) `1:` プログラムの**何行目**でエラーが起きたか(🌟重要🌟)
- (3) 今回は得られる情報が少ないので省略
- (4) `undefined method 'put'` : 訳「定義されていないメソッド'put'」
  - 「putメソッドを実行したいが、定義(用意)されていないので困りました」
- (5) `NoMethodError` : エラー(例外)の名前
  - メソッド(ここではput)が用意されていないときに起こるエラー
- Ruby超入門 Page.67 「エラーメッセージの読み方」より

```
put "hi"
^^^
Did you mean?  puts
               putc
```

- `^^^` の部分(ここではput)でエラーが起きていることを示している
- `Did you mean? puts putc`
  - `Did you mean?` は「もしかして？」の意味
  - 「もしかして puts や putc なのではありませんか？」
- まとめ: 「hi.rb の 1 行目に書かれた put メソッドは定義されていません。もしかして、puts や putc ではないですか?」
- 正しくは `puts "hi"` なので、put を puts と書き直せばエラーを修正できる

小数を正確に計算するときはRationalやBigDecimalをつかう

```
p 0.1+0.1+0.1 #=> 0.30000000000000004
```

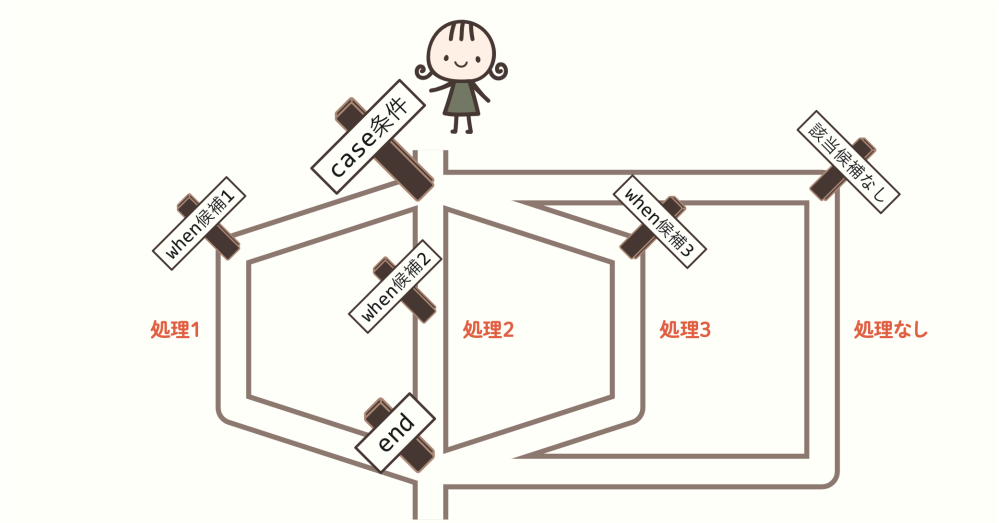
- たとえば⬆️のような小数を計算するコードは、結果が不正確になります
- `0.1` はFloatオブジェクト
  - Floatオブジェクトは厳密に正確な値を保持できないことがあるのが原因
- 正確に小数を扱うときはRationalオブジェクトやBigDecimalオブジェクトをつかう
- 数字の末尾にrをつけて書くとRationalオブジェクトをつくることができます。
- BigDecimalオブジェクトをつかうときは `require "bigdecimal"` が必要です。

```
p 0.1r+0.1r+0.1r #=> 3/10
p (0.1r+0.1r+0.1r).to_f #=> 0.3

require "bigdecimal"
p BigDecimal("0.1")+BigDecimal("0.1")+BigDecimal("0.1") #=> 0.3e0
```

case whenで多様なN者択一のコードを書く

- case when構文はN個の選択肢の中から1つを選んで実行します
- ifは**二者択一**、case whenは**N者択一**
- **「ifは2つに分岐、case whenは3つ以上に分岐」**と考えても良いでしょう
- いろいろな書き方ができます（このあと説明します）

case when の概念図 Ruby超入門 91ページより



when で条件判定する例

- caseの後ろに変数を置かずにwhenで条件判定する書き方

```
x = 2
y = 2
case
when x == 1
  puts 1
when y == 2
  puts 2
else
  puts "others"
end
#=> 2
```

caseの後ろに変数を置いてwhenでマッチする例

- caseの後ろに変数を置き、代入されたオブジェクトをwhenで判定させる書き方

```
x = 2
case x
when 1
  puts 1
when 2
  puts 2
else
  puts "others"
end
#=> 2
```

whenにRangeオブジェクトを書く例

- whenにRangeオブジェクトを書く例
  - 「こんな感じで動いて欲しい」動作になっているのではないでしょうか
- whenでの条件判定は===で行われています

```
x = 100
case x
when 1..20
  puts "1..20"
when (21..) # ()をつけないとSyntaxErrorエラー
  puts "21.."
else
  puts "others"
end
#=> 21..
```

whenにクラスを書く例

- whenにクラスを書く例
- オブジェクトのクラス（または親クラス）にマッチするか判定します
- (ただし、後述するダックタイピングで判定する方がRubyらしいです)

```
x = "foo"
case x
when String
  puts "String"
when Integer
  puts "Integer"
end
#=> String
```

elsifのかわりにcase whenをつかうとN者択一を表現できる

- if elsif構文が出てきたときは、case when構文で書き換え可能です
- N者択一を表現したいのであれば、case when構文をつかった方が直感的です
- 次のページのコードはif elsifをcase whenで書き換えた例です

```ruby
# 悪い例(N者択一ならばif elsifよりもcase whenが良い)
x = 2; y = 2
if x == 1
  puts "x == 1"
elsif y == 2
  puts "y == 2"
else
  puts "others"
end #=> y == 2
```

```ruby
# 良い例
x = 2; y = 2
case
when x == 1
  puts "x == 1"
when y == 2
  puts "y == 2"
else
  puts "others"
end #=> y == 2
```

selfを省略できるケースでは省略することが一般的

- selfを省略できるとき、ほとんどのケースで省略することが一般的です
- selfを省略できる例

```
class Foo
  def foo
    "foo!"
  end
  def bar
    # ⬇️ fooメソッドを呼び出すとき、 self.foo と書くよりも、selfを省略して foo と書く方が一般的
    foo
  end
end
puts Foo.new.bar #=> "foo!"
```

- selfを省略できないケースは数が少ないので、こちらを覚える方が簡単です
    - 次のページから説明していきます

selfを省略できないケース「ローカル変数代入と見分けられない」

- `foo = 1` は `foo=` メソッドの呼び出しではなく、ローカル変数 `foo` への代入
- `foo=` メソッドを呼び出したいときは `self.foo = 2` と書きます
- attr_writerやRailsのモデルクラスのattributesへの代入(後述)も同様の注意が必要

```
class Foo
  def foo=(arg)
    @foo = arg
  end
  def set_foo
    foo = 1 # ローカル変数fooへ代入
    self.foo = 2 # foo=メソッドの呼び出し
  end
end
```

selfを省略できないケース「ローカル変数代入と見分けられない2」

- 「Railsのモデルでのattributes(カラム)への代入」でも同じ状況になります
- usersテーブルにカラムnameがあると、Userモデルクラスに `name=` が追加される
- `name = "igaiga"`
  - `name=` メソッドの呼び出しではなく、ローカル変数 `name` への代入
  - `name=` メソッドを呼び出したいときは `self.name = "igaiga"`

```ruby
class User < ApplicationRecord
  # usersテーブルにカラムnameがある
  def set_name
    name = "igaiga" # ローカル変数nameへ代入
    self.name = "igaiga" # name=メソッドの呼び出し(attributesのnameへ代入)
  end
end
```

メソッドの引数が2つ以上のときはキーワード引数が便利

- メソッドに引数が2つ以上あるとき
  - 呼び出し側で**引数の指定順を間違える可能性**があります
  - キーワード引数をつかうと、呼び出し側で**好きな順序で**引数を書けます

```
# 問題が起こる例
def order(item, size)
  puts "#{item}を#{size}でください"
end

# 呼び出し側で引数の順序を間違える可能性がある
order "カフェラテ", "スモール"
# order "スモール", "カフェラテ" # こう書いてしまうと間違い
```

メソッドの引数が2つ以上のときはキーワード引数が便利

キーワード引数で書き直したコード例

- キーワード引数をつかうと、呼び出し側で好きな順序で引数を書けます

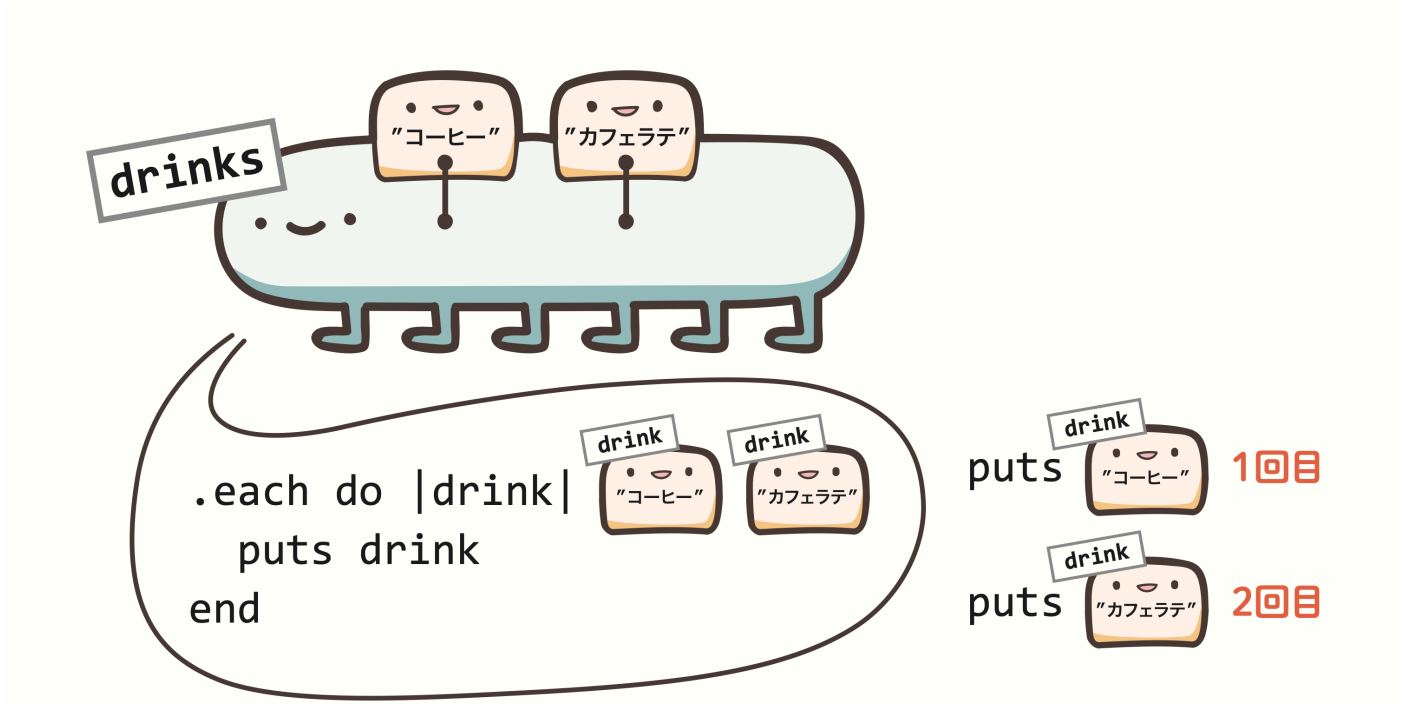
```
# 良い例
def order(item:, size:)
  puts "#{item}を#{size}でください"
end

# 呼び出し側で好きな順序で引数を書ける
order item: "カフェラテ", size: "スモール"
# order size: "スモール", item: "カフェラテ" # こう書いても良い
```

each と map のつかいわけ

- 「配列のeachメソッドで書いたコード、よく考えたらmapで書けた」
  - という経験があるかもしれません
- eachメソッドとmapメソッドは、各要素にブロックの処理をする点が似ています
- eachメソッドは **「全要素を繰り返し処理」** するメソッドです
- mapメソッドは **「全要素を変換処理して新しい配列を得る」** メソッドです
- mapメソッドでは**元の配列と変換後の配列の要素数が同じ**になります
- eachメソッドの結果、**元の配列と同じ要素数の配列**を結果として得たとき
  - mapメソッドで書けないかを検討してみるといいかもしれません

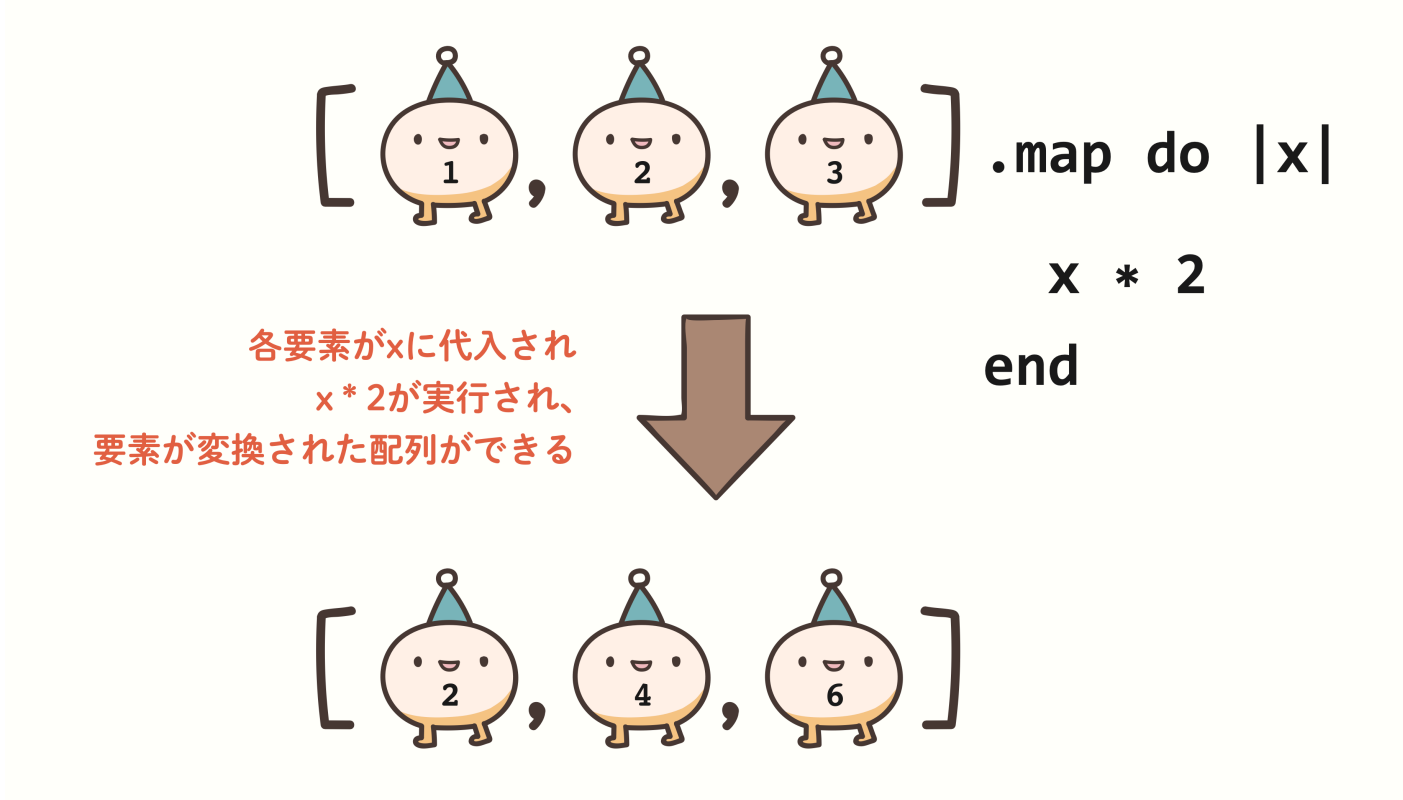
eachメソッド概念図 （Ruby超入門114ページより）

- eachメソッドは全要素を繰り返し処理するメソッド



mapメソッド概念図（Ruby超入門 140ページ より）

- mapメソッドは全要素を変換した新しい配列を得るメソッド
- mapメソッドでは**元の配列と変換後の配列の要素数が同じ**になる



名前重要

- Rubyではクラス名、メソッド名、変数名などの名前付けを重要視することが多い
- 理由の1つ
  - **型を書かないのでクラス、変数、メソッドの名前が主たる情報を書く場所**
- 名前付けの例を次のページで1つだけ説明します
- 名前付けについてより詳しく学びたい人へお勧め記事
  - WEB+DB PRESS vol.110 「名前付け大全」
  - WEB+DB PRESS 総集編 で読めます

名前付けの例: 配列には複数形、その要素には単数形

```
drinks = ["コーヒー", "カフェラテ", "モカ"] #  「複数の飲み物」が代入されるので複数形
drinks.each do |drink| # ブロック変数drinkは「1つの飲み物」が代入されるから単数形
  puts drink
end
```

- 配列はコーヒーとカフェラテのように **「複数の飲み物」** が代入されるので **複数形**
- ブロック変数drinkは **「1つの飲み物」** が代入されるから **単数形**
  - drink は変数なので、プログラムの動作だけならば名前はxでもなんでも良い
  - 読みやすさのため、代入されるものが飲み物なのでdrinkと命名
- 複数形: 配列やハッシュを代入する変数、配列やハッシュを返すメソッドなど
- 単数形: 単数のものを代入する変数、単数のものを返すメソッド、繰り返し中のブロック変数など

Rubyのコードでつかう記号

- 次のページから説明していきます
  - !!
  - ||=
  - &.
  - &:method
  - ナンバーパラメータ `_1`, `_2`, ...
- その他の参考ページ
  - Railsの練習帳「Rubyの引数でつかわれる記法」
    - *, **, &, ... などの説明
  - Rubyリファレンスマニュアル「リテラル」
    - 記号も含めてRubyの文法でつかわれる各種記法が載っています

!!

- `!!foo` と書くと、fooが指すオブジェクトをtrueまたはfalseへ変換
- notの意味の `!` を2回重ねたもので、notのnotで2回反転します
- 結果、次のような変換が行われます
  - false ➡️ false
  - nil ➡️ false
  - true ➡️ true
  - それ以外のオブジェクト ➡️ true
- nilとfalseはfalseへ、それ以外はtrueへ変換されます
  - (補足) **Rubyにおいてifなどの判定で偽になるものはfalseとnilの2つだけ**
- 末尾が?で終わるメソッドでtrueまたはfalseを返したいときによくつかわれます

||=

- `foo ||= 1` は `foo = foo || 1` と同じです
  - `foo += 1` が `foo = foo + 1` なのと同じ考え方で覚えられます
- 次のような意図でよくつかいます
  - fooに何か代入されていたらそのまま
  - fooに何も代入されていなければ1を代入
- 「変数fooを初期化する」と考えることもできます
- 注意: fooに既に `false` または `nil` が意味を持って代入されているとき
  - `foo ||= 1` はfooが `false` または `nil` のとき、fooへ `1` が代入されてしまう

&.

- レシーバがnilのときにはメソッド呼び出しをせず、nilを返すメソッド呼び出し
  - レシーバ: `x.method` と書いたときのxが指すオブジェクトのこと
- 例: `foo&.upcase` (upcaseメソッド: 文字列を大文字へ変換)
  - fooが(nil以外の例えば)"abc"のとき: upcaseメソッドが実行され"ABC"を返す
  - fooがnilのとき: nilを返す(upcaseメソッドは実行されない)
- nilのときにNoMethodErrorにさせずにメソッド呼び出しができます
- 通称ぼっち演算子と呼ばれています
  - 人がひとりで体育座りをしているように見える様から

&:method 記法

```
p ["abc", "123"].map{|text| text.reverse } #=> ["cba", "321"]
```

- ⬆️ 配列の各要素をreverseメソッドで逆順にするコード
- 各要素に対してメソッドを呼び出すだけのブロックは短く書く記法があります
- 次のコードは前のコードと同じ結果になります

```
p ["abc", "123"].map(&:reverse) #=> ["cba", "321"]
```

- ブロックで呼び出したいメソッド名の先頭に:をつけてシンボルにします
- さらにその前に&をつけて引数で渡します
- 暗記するのがかんたんです
    - 動きを知りたい方はこちら: Railsの練習帳 Rubyの引数でつかわれる記法

ナンバーパラメータ

- ナンバーパラメータは、ブロック変数を省略して書ける記法です
  - ブロック引数を書かずに `_1` をその替わりとしてつかうことができます
  - 複数のブロック引数を取るメソッドでは、`_1`, `_2`, `_3` … がつかえます。

```
p ["a", "b"].map{ _1.upcase } #=> ["A", "B"]
# 次のコードと同じ
# p ["a", "b"].map{|x| x.upcase } #=> ["A", "B"]
```

- Ruby3.4からは `_1` のようにつかえる `it` が導入されました
  - ナンバーパラメータが1つだけのときに読みやすく書けます

```
p ["a", "b"].map{ it.upcase } #=> ["A", "B"]
# 次のコードと同じ
# p ["a", "b"].map{ _1.upcase } #=> ["A", "B"]
```

クラスメソッドとインスタンスメソッドのインスタンス変数は別

- **インスタンス変数の持ち主はself**になります
- クラスメソッドとインスタンスメソッドでのインスタンス変数は**別物**になります
  - それぞれのメソッドでのselfが異なるため(詳しい説明: Ruby超入門 Page.270)
- クラスが持ち主のインスタンス変数をクラスインスタンス変数とも呼びます(後述)

```
class Drink
  def name # インスタンスメソッドname
    @name = "カフェラテ" # （インスタンスメソッドの）インスタンス変数@nameへ代入
  end
  def self.name # クラスメソッドname
    @name  # （クラスメソッドの）インスタンス変数@nameを返す
  end
end
drink = Drink.new
drink.name # インスタンスメソッドnameを呼び出し
p Drink.name #=> nil # クラスメソッドnameを呼び出すとnil
```

クラスインスタンス変数

- **インスタンス変数の持ち主はself**になります
- クラスメソッドのselfはそのクラスになります
- クラス直下のselfはそのクラスになります
- クラスが持ち主のインスタンス変数をクラスインスタンス変数とも呼びます

```
class Drink
  @name = "カフェラテ" # クラスインスタンス変数
  # p self #=> Drink
  def self.name # クラスメソッドname
    # p self #=> Drink
    @name # クラスインスタンス変数
  end
end
p Drink.name #=> "カフェラテ"
```

クラスを判定するよりも、ダックタイピングで判定する

- オブジェクトがある種類のときだけ処理をするケースを考えます
- オブジェクトに対してつかうメソッドを呼び出しできるか判定するのがお勧め
  - ダックタイピング的な判定
- オブジェクトのクラスを判定することは、Rubyらしくないのでお勧めしません
- respond_to?メソッドでレシーバがメソッドを呼び出し可能か判定します
  - 引数にメソッド名のシンボルを渡します

```
def foo(x)
  # 良い例
  p x.upcase if x.respond_to?(:upcase) #=> "ABC"
  # 悪い例
  p x.upcase if x.is_a?(String) #=> "ABC"
end

foo "abc"
```

ブロックとローカル変数の仕様

ブロックとは

- `do` から `end` までの処理のかたまり(複数行で書くときによくつかわれる)
- `{` から `}` までの処理のかたまり(1行で書くときによくつかわれる)
- メソッドへ渡してつかう

ブロック中で代入したローカル変数

- ブロックの中でだけつかうことができます

```
1.times do
  x = 1
  p x #=> 1
end
p x #=> NameError: undefined local variable or method 'x' # ブロック外ではつかえない
```

ブロックの外(前)で代入したローカル変数

- ブロックの中でも、ブロックが終わった後でもつかうことができます
- ブロックの中でのローカル変数変更はブロック後も残ります

```
x = 1
1.times do
  p x #=> 1
  x = 2
  p x #=> 2
end
p x #=> 2
```

```
result = [] # ブロックの外でローカル変数へ代入しておく
[1,2,3].each do |i|
  result << i * 2 # ブロックの中で結果をつくる
end
p result #=> [2, 4, 6] # ブロックが終わった後にもつかえる
# 前述のmapで置換可能ケースなので実用上はmapの方が読みやすい [1,2,3].map{|i| i * 2 }
```

メソッド探索順を調べる Module#ancestorsメソッド

- Module#ancestorsメソッドでメソッド探索順を調べられます
  - 自分クラス、親クラス、prepend, include, extendされたモジュールなどへ探索
- メソッド探索順とメソッド呼び出しの流れ
  - 呼び出されたメソッドを探索順に探し、最初にみつけたメソッドを実行
  - 呼び出されたメソッド中でsuperが呼ばれると次順のメソッドを実行(後述)
  - 呼び出されたメソッドの終端まで実行したら終わり

```
class A
end
class B < A
end
p B.ancestors #=> [B, A, Object, Kernel, BasicObject]
```

メソッド探索順で次の順序である同名メソッドを呼び出す super

- `super`: 探索次順の同名メソッドを呼び出し
- `super` は現在のメソッドへ渡ってきた**引数を**探索次順メソッドへ**自動で渡す**
  - `super(x)` のように引数を明示して渡すことも可能
  - 引数を渡したくないときは `super()` と書く
- superで呼び出したメソッドが終わったらsuperを呼んだところへ戻る
  - superの戻り値は探索次順メソッドの戻り値
- 次のページでサンプルコードを出して説明します

super のサンプルコード

```ruby
class A
  def foo(a, b)
    "#{a}, #{b}"
  end
end

class B < A
  def foo(x, y)
    super # A#fooメソッドを引数x,yを渡して呼び出し
  end
end

p B.ancestors #=> [B, A, Object, Kernel, BasicObject] # メソッド探索順
p B.new.foo(1,2) #=> "1,2"
```

initializeメソッドではsuperを忘れずに

- initializeメソッドはどのクラスでも同じ名前になります
- initializeメソッド中でsuperを呼ばないと、親クラス群のinitializeメソッドが呼ばれなくなります
  - 初期化が不十分になることがあります
- initializeメソッドを実装するとき
  - 原則としてsuperを呼び出すのが良いでしょう
- initializeメソッドを実装しないとき
  - 探索次順のinitializeメソッドが呼ばれるので気にせず大丈夫

initializeメソッドではsuperを忘れずに サンプルコード

```ruby
class A
  def initialize(x, y)
    @x = x
    @y = y
  end
end

class B < A
  def initialize(x,y)
    # ... 自分のクラスの初期化処理(親クラスより前に実行)
    super # 書き忘れると親クラスAのinitializeが実行されない
    # ... 自分のクラスの初期化処理(親クラスより後に実行)
  end
  def x
    @x
  end
end

p B.new(1,2).x #=> 1
```

リファレンスマニュアルの調べ方

- Rubyリファレンスマニュアル(通称「るりま」)
- Rubyの各機能について辞書のように調べられます
- 例: Arrayクラスのメソッド一覧 「組み込みライブラリ Builtin libraries」 - 「Array」
- わからない機能を調べるほかに、全体を一読するのもお勧め
  - Array, Hash, String, Enumerableはよくつかうので読んでおくとRuby力up
- Rubyリファレンスマニュアルを全文検索できるるりまサーチもあります
- クラスとメソッドをマニュアルなどで記述するときの記法
  - インスタンスメソッド: クラス名#メソッド名 (# 記号を間に入れて書く)
    - 例: Array#map, String#upcase
  - クラスメソッド: クラス名.メソッド名 (. 記号を間に入れて書く)
    - 例: Array.new, String.new

Rubyのライブラリ

- Rubyのライブラリには3種類あります
- Rubyと一緒にインストールされる「組み込みライブラリ」、「標準添付ライブラリ」
  - **「組み込みライブラリ」** : requireせずにつかえるクラス群
    - 例: Array、Hash、String
  - **「標準添付ライブラリ」** : requireしてつかうクラス群
    - 例: JSON、YAML、OpenSSL
- Rubyとは別にインストールする **「Gem」**

Default gemsとBundled gems

- 別の分類で、Default gemsとBundled gemsという仕組みもあります
- **Default gems**
  - Rubyと一緒にインストールされるライブラリをGemで更新する仕組み
  - RubyGemsをつかって各ライブラリ単体でバージョンアップ可能
  - 標準添付ライブラリの多くと一部の組み込みライブラリはこの仕組みで提供
  - Bundlerで特別扱いされるため、Gemfileに書かなくてもつかえる
- **Bundled gems**
  - Rubyと一緒にインストールされるGem
  - Rubyの開発と共にテストされていて、対象Rubyバージョンで動作確認済み
  - Bundlerで特別扱いされず(前ページの一般Gemと同じ扱い)、Gemfileへ要追記
- 参考: 徹底解説！default gemsとbundled gemsのすべて

IRBとbinding.irb

- IRB: 対話的にRubyコードを実行できる環境
  - Default gemとしてRubyに最初からインストールされています
- `binding.irb`
  - 書いた場所で一時停止して入力したRubyコードを実行可能
  - 追加インストール不要で実行可能、requireも不要
  - 多くのケースで便利にデバッグできます
  - より多くの機能でデバッグしたいときはdebug gemをつかいます(後述)
  - デバッグのほか、つかい方を知りたいオブジェクトを調べるのも便利です
    - `methods` メソッド: そのオブジェクトで呼び出せるメソッド一覧
    - `instance_variables` メソッド: オブジェクトが持つ全インスタンス変数

debug gemとbinding.break

- Bundled gemとしてRubyに最初からインストールされています
  - Gemfileをつかっているときは `gem "debug"` を追記
    - Railsでは次のようにdevelopment環境とtest環境へ追加
    - `gem "debug", group: [:development, :test]`
- `require "debug"` したあとで `binding.break` を書きます
  - 書いた場所で一時停止してデバッグコンソールが起動
  - Rubyコードが実行可能なほか、さまざまなコマンドがつかえます(後述)
- `binding.break` にかわりに `binding.b` や `debugger` とも書けます

binding.break

- binding.breakのサンプルコード

debug_sample.rb

```ruby
require "debug"

class Foo
  def bar(arg)
    "hi"
    binding.break
  end
end

Foo.new.bar(555)
```

- `binding.break` のところで一時停止してデバッグコンソールが起動

```
$ ruby debug_sample.rb
[1, 10] in debug_sample.rb
     1| require "debug"
     2|
     3| class Foo
     4|   def bar(arg)
     5|     "hi"
=>   6|     binding.break
     7|   end
     8| end
     9|
    10| Foo.new.bar(555)
=>#0    Foo#bar(arg=555) at debug_sample.rb:6
  #1    <main> at debug_sample.rb:10
(rdbg)
```

- 一時停止したときのメソッド呼び出し履歴が表示
- メソッド呼び出し時に渡された引数もあわせて表示されます(便利)

デバッグコンソールのつかい方 その1 基本コマンド

- Rubyコードを実行可能(binding.irbと同じ)
- next or n
  - 1行ずつステップ実行
- continue or c
  - 一時停止を解除してプログラムを再開
- help or ?
  - 利用できるコマンドを表示

デバッグコンソールのつかい方 その2 infoコマンド

- infoコマンド(iと省略可能): 変数一覧とそれらの情報を表示
- i l
    - ローカル変数と代入されたオブジェクトを表示
- i i
    - selfが持つインスタンス変数と代入されたオブジェクトを表示
- i i <object>
    - objectが持つインスタンス変数と代入されたオブジェクトを表示(便利)
- i c
    - アクセス可能な定数と代入されたオブジェクトを表示(トップレベルを除く)

デバッグコンソールのつかい方 その3 lsコマンド

- lsコマンド: infoコマンド同様に情報を表示する機能
- ls
  - 現在のスコープでつかえる次の情報を表示
    - メソッド、定数、ローカル変数、インスタンス変数
- ls <object>
  - objectのスコープでつかえるメソッド、インスタンス変数を表示

デバッグコンソールのつかい方 その4 backtraceコマンド

- backtraceコマンド(btと省略可能): メソッド呼び出し履歴(バックトレース)を表示
- bt
  - メソッド呼び出し履歴(バックトレース)を表示
- bt <num>
  - 直近num件だけ表示
- bt /regexp/
  - メソッド名やファイルパスを正規表現 /regexp/ でフィルターして表示

デバッグコンソールのつかい方 その4 backtraceコマンド表示例

Railsの一覧表示画面(indexアクション)を表示したときのbtコマンドの表示結果

```
=>#0    BooksController#index at ~/work/rails707_ruby322_books_app/app/controllers/books_controller.rb:6
  #1    ActionController::BasicImplicitRender#send_action(method="index", args=[]) at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actionpack-7.0.7/lib/action_controller/metal/basic_implicit_render.rb:6
  #2    AbstractController::Base#process_action at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actionpack-7.0.7/lib/abstract_controller/base.rb:215
  #3    ActionController::Rendering#process_action at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actionpack-7.0.7/lib/action_controller/metal/rendering.rb:165
  #4    block in process_action at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actionpack-7.0.7/lib/abstract_controller/callbacks.rb:234
  #5    block in run_callbacks at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/activesupport-7.0.7/lib/active_support/callbacks.rb:118
  #6    ActionText::Rendering::ClassMethods#with_renderer(renderer=#<BooksController:0x0000000000f820>) at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actiontext-7.0.7/lib/action_text/rendering.rb:20
  #7    block {|controller=#<BooksController:0x0000000000f820>, action=#<Proc:0x00000001096b05d0 /Users/igaiga/....|} in <class:Engine> (4 levels) at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/actiontext-7.0.7/lib/action_text/engine.rb:69
  #8    [C] BasicObject#instance_exec at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/activesupport-7.0.7/lib/active_support/callbacks.rb:127
  #9    block in run_callbacks at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/activesupport-7.0.7/lib/active_support/callbacks.rb:127
  #10   ActiveSupport::Callbacks#run_callbacks(kind=:process_action) at ~/.rbenv/versions/3.2.2/lib/ruby/gems/3.2.0/gems/activesupport-7.0.7/lib/active_support/callbacks.rb:138
```

- バックトレース表示する別のメソッド
  - Kernel.#callerメソッド
  - Kernel.#caller_locationsメソッド
- debug gemのbtコマンドはメソッド呼び出し時に渡した引数の内容も表示(便利)

デバッグコンソールのつかい方 その5 trace コマンド

- trace コマンド: 以降の処理で指定したイベントが発生したときにその情報を表示
- trace call
  - メソッド呼び出しとreturnのときにその情報を表示
  - returnのときは、その戻り値も表示
  - trace call /regexp/ のように正規表現で絞ると便利
    - /regexp/ はtrace callのほか、全てのtraceコマンドでつかえます
- trace exception
  - 例外が投げられたときにその情報を表示
  - これをセットしておくとrescueされた例外にも気づけます(便利)

デバッグコンソールのつかい方 その6 binding.break do:

- binding.break do "コマンド"
  - do以降のコマンドを実行してプログラム実行を再開
  - デバッグコンソールのコマンド入力待ちにならないため、高速に試行できる
- 例: コード中の指定区間で投げられた例外を表示
  - `binding.break do: "trace exception"`
  - `binding.break do: "trace off exception"` でコードを囲む

```ruby
require "debug"
binding.break do: "trace exception"
# ...Rubyコード...（この間に投げられた例外が表示される）
raise RuntimeError.new("foo")
binding.break do: "trace off exception"
#=> DEBUGGER (trace/exception) #th:1 #depth:1  #<RuntimeError: foo> at debug.rb:4
```

デバッグコンソールのつかい方 その7 catch, watch

- 一時停止してデバッグコンソールを起動するコマンドです
- catch <ExceptonName>
  - ExceptonName例外が投げられたときに一時停止
- watch <@instance_variable_name>
  - (selfが持っている)指定されたインスタンス変数が変更されたときに一時停止

デバッグコンソールのつかい方 その8 コマンドの調べ方

- helpコマンド: コマンド一覧と説明を表示
- 次のページも参考になります
    - debug gem GitHub
    - debug gem GitHub コマンド一覧
    - ruby/debug cheatsheet
    - Railsの練習帳 debug gem
- debug gem以外のデバッグのノウハウは次のページにまとめています
    - Railsの練習帳 デバッグに便利な道具

Gemのソースコードも変更して実行できる

- 自分で書いたコードだけでなく、Gemのソースコードも変更して実行できます
- `binding.irb` や `binding.break` を書いて実行を一時停止できる
- Method#source_locationやbundle showコマンド(後述)でGemのパスを調べられる
- 変更したいファイルをエディタで開いて編集して保存
- Gemをインストール時のコードに戻すコマンド
  - `gem pristine gem名` コマンド
  - `bundle pristine gem名` コマンド
  - `bundle pristine` コマンド: Gemfileに書かれた全てのgemを元に戻す

Method#source_location メソッド

- Method#source_location: メソッドが定義されているソースファイルのパスを返す
- 取得方法
  - methodメソッドにシンボルでメソッド名を渡してMethodオブジェクトを取得
  - Methodオブジェクトに対してsource_locationメソッドを呼ぶ
    - ソースファイルのパスと行数を表示
    - pメソッドでMethodオブジェクトを表示しても同等の情報を得られる
- source_locationメソッドでnilが返るときはRubyで定義されていないメソッド
  - たとえばC言語で定義されているメソッドはnilを返す

```
require "csv"
CSV.method(:read) #=> Methodオブジェクト
CSV.method(:read).source_location #=> ["~/.rbenv/versions/3.4.1/lib/ruby/gems/3.4.0/gems/csv-3.3.2/lib/csv.rb", 1922]
```

Methodオブジェクトの便利なメソッド

- Methodオブジェクトには他にも便利なメソッドが用意されています
- Method#owner: メソッドが定義されているクラスまたはモジュールを返す
- Method#original_name: aliasがつかわれているときにalias先のメソッド名を返す
- Method#super_method: superで呼び出されるメソッドオブジェクトを返す
- 定数の定義元を調べるModule#const_source_locationもあります
    - source_locationメソッドの定数版

```
class Foo
end
p Module.const_source_location(:Foo) #=> ["example.rb", 1]
```

bundle show, gem whichコマンド

- Gem名からそのGemがインストールされているパスを調べるコマンドです
- BundlerとRubyGemsとそれぞれにコマンドが用意されています
- $ bundle show gem名
- $ gem which gem名
- RailsアプリではBundlerのコマンドである bundle show をつかうのがお勧めです
- $ bundle open gem名
  - gemのコード(bundle showが返すパス)をエディタで開く
  - 環境変数EDITORやBUNDLER_EDITORを設定しておく必要があります

そのほか話したかったこと

- Frozen String Literalとマジックコメント
- JSON.parseメソッド symbolize_namesオプション
- パターンマッチ case in構文
- Railsの練習帳 「Rubyの知識」 のページに書いていますのでぜひどうぞ

Rubyの学び方 Web編

- Railsの練習帳(Rubyのことも書いています)
  - Rubyの知識
  - Rubyの引数でつかわれる記法
  - デバッグに便利な道具
  - debug gem
- この資料の前のページで説明
  - Rubyリファレンスマニュアル(通称: るりま)
  - るりまサーチ

Rubyの学び方 書籍編

- ゼロからわかるRuby超入門
  - 入門書。今日の話の中でもいろいろ引用しました。Ruby3.2対応。
- Ruby コードレシピ集
  - 「〇〇したい」からそれに対応するコードを調べられるスタイルの本です。Ruby3.3対応。
  - 私もこのスタイルの昔の本でRubyを学びました

Railsの学び方 書籍編、Web編

- Railsの教科書
  - Railsの基礎を図解を見ながら手早く理解できる書籍
- はじめてつくるWebアプリケーション
  - Webアプリの仕組みやRailsの基礎知識などをやさしく解説する書籍
- Railsガイド
  - Railsコアチームが執筆、更新しているRailsを説明するWeb記事
- Railsガイド Railsをはじめよう
  - Railsの基礎知識と各種機能を解説する(最近執筆された)Web記事
- Railsの練習帳
  - Railsの学習をサポートするWeb記事。今日もたびたび引用。

Rubyの学び方 上級編

- メタプログラミングRuby
  - メタプロ(プログラミングコードを記述するコード)のわかりやすい解説本
  - メソッド探索順やクラスメソッドの実現方法など、Rubyの技術を幅広く説明
- 研鑽Rubyプログラミング
  - 深く幅広いRuby技術の長所短所とつかいどころを説明する中上級者向け書籍
- Rubyのウラガワ WEB+DB PRESS Vol.110〜122連載
  - RubyVMの仕組みについて解説する雑誌連載記事
- なるほどUnixプロセス
  - プロセスやシグナルといったUnix基礎技術をRubyコードで解説する本
- Webで使えるmrubyシステムプログラミング入門
  - プロセス、ソケット、ネットワークなどの基礎技術をわかりやすく解説
  - strace, gdb, valgrind, readelf, perfなどの解析ツールも解説

Rubyの学び方 コミュニティ

- 地域Rubyコミュニティ地域.rb 日本Rubyの会
  - 全国各地のRubyコミュニティ紹介
  - 活動内容例: 雑談を交えてもくもくとコードを書いたり、テーマを決めて勉強会、発表したい人が発表するなど
  - オンライン開催も現地開催もあります
- Regional.rb and the Tokyo Metropolis
  - 東京近郊のRubyコミュニティ紹介 東京Ruby会議12講演
- RubyEventCheck 日本Rubyの会
  - Ruby関連イベントページへのリンク集
- ruby-jp Slack
  - Rubyistが多く集うSlack

Rubyの学び方 わからないことを減らす作業

- 今日のセミナーや本などで学ぶことはわかることを増やす作業
  - 一方で、わからないことを減らしていく作業も効果的
  - わからないことを減らす作業
    - 「たくさんの知識から優先度高く得るべきものを見つける」作業
  - 練習問題を解く、コードレビュー(次のページで説明)を受けるなど
- 言語処理100本ノック
  - 練習問題がたくさん用意されているWeb資料

Rubyの学び方 コードレビュー

- 自分が書いたコードにレビューを受けると効率良くわからないところを探せます
- フィヨルドブートキャンプ
  - 有料プログラミングスクール(私もお手伝いしているので宣伝込みです)
  - メンター(現役Railsプログラマー)たちからのコードレビューを受けられる
  - 質問をしたり、ペアプロをしたり
  - カリキュラム内容
  - 個人だけではなく法人契約も可能(たとえば新人研修の一環での利用など)
- コードレビューで学ぶ Ruby on Rails 第二版
  - Railsで中級者を目指す方々向けの本
- 伝わるコードレビュー
  - コードレビューする側の技術として、4月に新刊が出るそうです

Railsの練習帳 スポンサー募集 (宣伝)

- Railsの練習帳は無料で読める形で公開しています
- **読者からではなく、エンジニアを応援する企業からお金をいただく形を模索中**
- Rails学習者の中にはこれからRailsエンジニアを目指す人たちがいます
  - 仕事をやめて学んでいる方など、金銭状況が厳しい方もいます
- Railsエンジニアを応援する会社さんの支援を受けて、もっと書きたい！
- 模索中なので、スポンサーの組み方からご相談させてください
- スポンサー特典例: 社名広告掲載、五十嵐による定期勉強会の開催、など
- スポンサー方法例: 金銭提供、五十嵐の業務時間の一部を執筆に充てる、など
- Zennで個人スポンサーになってくれたみなさまに感謝します
- iCARE様、コインチェック様、ペアプロ業務を通じた応援に感謝します

お仕事募集(宣伝)

- Railsの業務委託の仕事(いわゆる技術顧問業)を月に何日かリモートで承っています
- 今日のような単発の講義や長期の育成も承ります
- 業務内容の例（「こんなこともできる？」とご相談いただけましたら）
- 著書 パーフェクトRails や Railsの練習帳 などをつかった読書会、講義
- ペアプロ屋、二人三脚での開発技術相談、実装
- 入社後の研修育成体制を構築して、採用できるエンジニア範囲を増やす
- コードの健康診断とレポート
- レガシーコード改善実装、RubyとRailsのバージョンアップ実装
- ガーネットテック373株式会社
    - 仕事内容詳細ページ、問い合わせページ ⬅️ こちらから気軽にご相談ください