【Ruby】基礎

目次

環境編

■用語

▶ ※ rbenv も rvm もRubyのインストールやバージョン管理を行うツールである。

■環境構築（全般）

▶ バージョンを確認

■環境構築（rbenvを利用）

Windows (WSLでUbuntuを導入)

▶ ☆ 0．WSL 2でUbuntu環境を構築

▶ ☆ 1．基本的なライブラリ群をインストール（推奨？）

▶ ☆ 2．Rubyのインストール

macOS

▶ ☆ 1．Rubyのインストール

■環境構築（rvmを利用）

たぶんOSによらない

▶ ☆ 1．Rubyのインストール

【Ruby】基礎

目次

- 環境編
- 基礎文法編
- オブジェクト編
- クラス編
- モジュール編
- ライブラリ編
- 応用編

環境編

■用語

▶ ※ rbenv も rvm もRubyのインストールやバージョン管理を行うツールである。

■環境構築（全般）

▶ バージョンを確認　　$ ruby -v

■環境構築（rbenvを利用）

Windows (WSLでUbuntuを導入)

▶ ☆ 0．WSL 2でUbuntu環境を構築

▶ ☆ 1．基本的なライブラリ群をインストール（推奨？）

▶ ☆ 2．Rubyのインストール

macOS

▶ ☆ 1．Rubyのインストール

■環境構築（rvmを利用）

たぶんOSによらない

▶ ☆ 1．Rubyのインストール

基礎文法編

■初歩的注意

- ▶ ※ すべての値（リテラル）はオブジェクト（つまりメソッドを従えている）。
- ▶ ※ クラスやモジュールもオブジェクトである。
- ▶ ※ トップレベルは `main` という名の `Object` クラスのインスタンスになっている。
- ▶ ※ コード文が続くことが明らかな場合にかぎり自由に改行できる。
  また、`\` によってコード文の途中で改行することもできる。
- ▶ ※ セミコロン `;` によってコード文を改行せずに複数の命令ができる。
- ▶ ※ 暗黙の型変換はされない！
- ▶ ※ 「文のように見えるが実は式である」ような要素が多い。
- ▶ ☆ 演算子の優先順位
- ▶ ※ ガベージコレクション (GC) が常に働いている。

■用語

- ▶ 擬似変数
- ▶ トップレベル
- ▶ 特異メソッド
- ▶ 組み込み定数

■基礎

- ▶ コメントのしかた
- ▶ 変数を定義
- ▶ 複数の変数に同時代入
- ▶ 変数がnilの時だけ代入
- ▶ 大域脱出
- ▶ グローバル変数
- ▶ 文字列をコードとして実行
- ▶ 文字列をOSコマンドとして実行
- ▶ Ruby のバージョン番号

■標準入出力・標準エラー入出力

- ▶ ｺﾏﾝﾄﾞﾗｲﾝでの引数
- ▶ １行の文字列を入力

基礎文法編

■初歩的注意

- ▶ ※ すべての値（リテラル）はオブジェクト（つまりメソッドを従えている）。
- ▶ ※ クラスやモジュールもオブジェクトである。
- ▶ ※ トップレベルは `main` という名の `Object` クラスのインスタンスになっている。
- ▶ ※ コード文が続くことが明らかな場合にかぎり自由に改行できる。
  また、`\` によってコード文の途中で改行することもできる。
- ▶ ※ セミコロン `;` によってコード文を改行せずに複数の命令ができる。
- ▶ ※ 暗黙の型変換はされない！
- ▶ ※ 「文のように見えるが実は式である」ような要素が多い。
- ▶ ☆ 演算子の優先順位
- ▶ ※ ガベージコレクション (GC) が常に働いている。

■用語

- ▶ 擬似変数　特殊な変数。値の代入はできない。 `nil` , `true` など。
- ▶ トップレベル　クラス定義やモジュール定義に囲まれていない一番外側
- ▶ 特異メソッド　特定のオブにだけ紐づくメソ
- ▶ 組み込み定数　Object クラスに初めから定義されている定数

■基礎

- ▶ コメントのしかた　# で行末まで、あるいは =begin =end で囲めば改行可能。
- ▶ 変数を定義　hoge = 値
- ▶ 複数の変数に同時代入　hoge, ... = 値1, ...　か　hoge, ... = [値1, ...]　※交換も可能に！
- ▶ 変数がnilの時だけ代入　hoge ||= 値
- ▶ 大域脱出　catch :*tag* { ‥　throw :*tag*, 返り値　‥ }　※返り値は省略可
- ▶ グローバル変数　$hoge = 値　※あまり使うべきでない
- ▶ 文字列をコードとして実行　eval('文字列')
- ▶ 文字列をOSコマンドとして実行　`文字列`　%x!文字列!　※ ﾊﾞｯｸｸｵｰﾄﾘﾃﾗﾙ と %記法
- ▶ Ruby のバージョン番号　RUBY_VERSION

■標準入出力・標準エラー入出力

- ▶ ｺﾏﾝﾄﾞﾗｲﾝでの引数　ARGV[**0**]　※ $ ruby ○○.rb **第0引数**
- ▶ １行の文字列を入力　変数 = gets.comp

- ▶ 改行せず出力
- ▶ 改行して出力
- ▶ 改行して標エラに出力
- ▶ デバックに便利な出力
- ▶ きれいに出力

■条件分岐

- ▶ 条件分岐
- ▶ 単純なif文の略記
- ▶ 否定のif
- ▶ 単純な否定if文の略記
- ▶ 真偽値以外も評価
- ▶ 真偽値に変換
- ▶ 比較演算子
- ▶ ※ 論理演算子を使った式について評価が終了されるタイミングは、**式全体の真偽が確定した瞬間**である。
- ▶ ※ 論理演算子を使った式の返り値は、**最後に評価した式の値**である。
- ▶ 論理演算子
- ▶ ２股分岐の略記
- ▶ switch 文
- ▶ ※ `==` は値が等価かを判定する（オブのID番号が同じかを判定するのではない）。なお、人間にとって自然な等価判定がされる（たとえば `1 == 1.0` は真になる）。

■繰り返し処理

- ▶ *n*回処理を繰り返す
- ▶ *n*から*m*まで繰り返す
- ▶ stepして〃
- ▶ while文
- ▶ do-while文
- ▶ 無限ループ
- ▶ 中断し次へ・脱出
- ▶ 中断しその回を再び

- ▶ 改行せず出力　print 式 #　※: `nil`
- ▶ 改行して出力　puts 式 #　※: `nil`
- ▶ 改行して標エラに出力　warn 式 #　※: `nil`
- ▶ デバックに便利な出力　p 式 #　※: `式` のオブジェクト
- ▶ きれいに出力　pp 式 #

■条件分岐

- ▶ 条件分岐　if　**elsif**　else　**end**
- ▶ 単純なif文の略記　条件 and 1行の処理　か　1行の処理 if 条件
- ▶ 否定のif　unless　else　end　※unlessは偽のときだけ実行。
- ▶ 単純な否定if文の略記　条件 or 1行の処理　か　1行の処理 unless 条件
- ▶ 真偽値以外も評価　false nil 以外はすべて true と評価
- ▶ 真偽値に変換　!!式
- ▶ 比較演算子　<　<=　>　>=　==　!=
- ▶ ※ 論理演算子を使った式について評価が終了されるタイミングは、**式全体の真偽が確定した瞬間**である。
- ▶ ※ 論理演算子を使った式の返り値は、**最後に評価した式の値**である。
- ▶ 論理演算子　&&　||　！　!( )　※ `not` `and` `or` も使えるが評価優先順位低い
- ▶ ２股分岐の略記　条件 = ? 真での値 : 偽での値
- ▶ switch 文　case 式↲　when 値↲　処理↲　else↲　処理↲　end
- ▶ ※ `==` は値が等価かを判定する（オブのID番号が同じかを判定するのではない）。なお、人間にとって自然な等価判定がされる（たとえば `1 == 1.0` は真になる）。

■繰り返し処理

- ▶ *n*回処理を繰り返す　*n*.times { |n| 処理 }　※ nには0～*n*-1が入る
- ▶ *n*から*m*まで繰り返す　*n*.upto(*m*) { |n| 処理 }　※*m*含む　※*n*>*m*なら　*n*.downto(*m*) ‥
- ▶ stepして〃　*n*.step(*m*, *step*)　※*m*含みうる　※当然*step*は負も可
- ▶ while文　while 条件↲　処理↲　end　unless 逆の条件↲　‥
- ▶ do-while文　begin↲　処理↲　end while 条件
- ▶ 無限ループ　loop { 処理 }　while true↲　処理↲　end
- ▶ 中断し次へ・脱出　**next**　・　break
- ▶ 中断しその回を再び　redo

■メソッド

- ▶ メソ(引数なし)を定義
- ▶ メソ(引数あり)を定義
- ▶ デフォルト値を設定
- ▶ 仮引数を必ず書かせる
- ▶ 想定外のｷｰ引を受付け
- ▶ 可変長引数を設定
- ▶ 返り値を返す
- ▶ アーリーリターン
- ▶ ☆ 引数の順番の制限
- ▶ ※ `hoge?` など `?` で終わる名前のメソは慣習として真偽値を返す。
- ▶ ※ `hoge!` など `!` で終わる名前のメソは慣習として、使用する際は注意が必要なもの。
- ▶ ※ オーバーロード（引数のデータ型や個数の違うメソを複数定義する）の考え方はない。
- ▶ ※ 全く同じ機能のメソに名前が複数ついていることがよくある（**エイリアスメソッド**）。
- ▶ ｴｲﾘｱｽメソを定義
- ▶ メソを削除
- ▶ ※ 各種演算子の挙動は、内部的にはメソッドとして定義されている。
- ▶ ☆ 挙動を独自に再定義できる演算子

- ▶ メソの呼び出し
- ▶ ｷｰﾜｰﾄﾞ引数つきの〃
- ▶ 配列の全要素を渡す

■例外処理

- ▶ ☆ 例外処理を書く際の心構え
- ▶ ☆ 例外クラスの継承関係（一部抜粋）
- ▶ わざと例外を投げる
- ▶ 例外処理の始まりと終わり
- ▶ ある例外の発生時に限り処理
- ▶ 例外の発生時に限り処理
- ▶ 例外オブのエラーメッセージ

■メソッド

- ▶ メソ(引数なし)を定義　def hoge_hoge↲　処理↲　end
- ▶ メソ(引数あり)を定義　def hoge(p1, p2)
- ▶ デフォルト値を設定　def hoge(p1, p2 = 80, p3 = Time.now)　※前でもOK
- ▶ 仮引数を必ず書かせる　def hoge(p1, p2: 80, p3: Time.now, p4:)　※ｷｰﾜｰﾄﾞ引数という
- ▶ 想定外のｷｰ引を受付け　def hoge(p1, p2: 80, **p3)　※ `**p3` の部分。
- ▶ 可変長引数を設定　def hoge(p1, p2. *p3)
- ▶ 返り値を返す　最後に評価された式が返り値になる
- ▶ アーリーリターン　return 返り値 if ～
- ▶ ☆ 引数の順番の制限
- ▶ ※ `hoge?` など `?` で終わる名前のメソは慣習として真偽値を返す。
- ▶ ※ `hoge!` など `!` で終わる名前のメソは慣習として、使用する際は注意が必要なもの。
- ▶ ※ オーバーロード（引数のデータ型や個数の違うメソを複数定義する）の考え方はない。
- ▶ ※ 全く同じ機能のメソに名前が複数ついていることがよくある（**エイリアスメソッド**）。
- ▶ ｴｲﾘｱｽメソを定義　alias hoge2 hoge1　※ hoge1はすでに定義されている必要
- ▶ メソを削除　undef hoge　※おそらくクラス内で行うことが推奨される
- ▶ ※ 各種演算子の挙動は、内部的にはメソッドとして定義されている。
- ▶ ☆ 挙動を独自に再定義できる演算子

- ▶ メソの呼び出し　hoge()　**hoge**　**hoge(*arg1*, ...)**　hoge *arg1*, ...
- ▶ ｷｰﾜｰﾄﾞ引数つきの〃　hoge(*arg1*, p2: *arg2*, p4: *arg4*)　※ハッシュで一気に渡せる☞
- ▶ 配列の全要素を渡す　hoge(..., *配列, ...)

■例外処理

- ▶ ☆ 例外処理を書く際の心構え
- ▶ ☆ 例外クラスの継承関係（一部抜粋）
- ▶ わざと例外を投げる　raise *message*　raise *ExcepClass*, *message*
raise *ExcepClass*.new(*message*)
- ▶ 例外処理の始まりと終わり　begin↲　‥↲　**end**
- ▶ ある例外の発生時に限り処理　rescue *ExcepClass1*, ...↲　rescue *ExcepClass1*, ... => e↲
- ▶ 例外の発生時に限り処理　rescue↲　rescue => e↲
- ▶ 例外オブのエラーメッセージ　e.message

▶ 例外オブのスタックトレース

▶ 例外発生時に再度やり直す

▶ 捕捉した例外を再び起こす

▶ 正常終了時に限り処理

▶ 異常正常によらず最後に処理

▶ ※ `ensure` 節では絶対に `return` を使うな。

▶ ※ 例外処理の戻り値は `begin` 節で最後に評価された式。

▶ １行の例外処理記法

▶ ☆ 例外処理だけのメソッドにおける略記法

■テスティング

▶ Minitest ラ

▶ テストクラスを定義

▶ ※ テストクラスにつける名前は `Test` で終わるか始まることが多い。また、テストファイルの名前は `hoge_test.rb` のようにテストクラスと合わせるのが普通。

▶ テストメソッドを定義

▶ 式が期待通りの値か確認

▶ 式が真であることを確認

▶ 式が偽であることを確認

■よくやる間違い

- **end** を忘れる

オブジェクト編

■オブジェクトの分類

▶ ミュータブルなオブ

■主なデータ型（リテラルの形式）

- 数値 `123`
- 文字列 `"Hello"`
- 真偽値 `true` `false`
- 配列 `[1, 2, 3]`

▶ 例外オブのスタックトレース e.backtrace

▶ 例外発生時に再度やり直す (rescue節の中で) retry ※ `begin` 節の頭から再開する

▶ 捕捉した例外を再び起こす (resuce節の中で) raise

▶ 正常終了時に限り処理 else↲

▶ 異常正常によらず最後に処理 ensure↲

▶ ※ `ensure` 節では絶対に `return` を使うな。

▶ ※ 例外処理の戻り値は `begin` 節で最後に評価された式。

▶ １行の例外処理記法 例外が発生しそうな処理 rescue 例外発生の場合の返り値

▶ ☆ 例外処理だけのメソッドにおける略記法

■テスティング

▶ Minitest ラ require 'minitest/autorun'

▶ テストクラスを定義 class HogeTest < Minitest::Test↲ ‥↲ end

▶ ※ テストクラスにつける名前は `Test` で終わるか始まることが多い。また、テストファイルの名前は `hoge_test.rb` のようにテストクラスと合わせるのが普通。

▶ テストメソッドを定義 def test_hoge↲ ‥↲ end ※必ず `test_` で始まる名に

▶ 式が期待通りの値か確認 assert_equal 期待される値, 式

▶ 式が真であることを確認 assert 式

▶ 式が偽であることを確認 refute 式

■よくやる間違い

- **end** を忘れる

オブジェクト編

■オブジェクトの分類

▶ ミュータブルなオブ 破壊的な変更が適用できるオブ

■主なデータ型（リテラルの形式）

- 数値 `123`
- 文字列 `"Hello"`
- 真偽値 `true` `false`
- 配列 `[1, 2, 3]`

- ハッシュ `{'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee'}`
- 正規表現 `/\d+-\d+/`

■オブジェクト一般

▶ ※ オブジェクトはみな、Objectクラスの子孫クラスのインスタンスである。よって、以下で紹介しているのはObjectクラスが持つメソや機能である。…といいつつ、その親クラスであるBasicObjectクラスのそれが含まれているかもしれないが、さほど問題ではないので許してほしい。

▶ オブに紐づくメソ呼出

▶ メソ名の文字列等から〃

▶ nilかもオブにメソ呼出

▶ オブのID

▶ 同じ(IDの)オブか

▶ オブのクラス

▶ 〃が*class*か

▶ 〃の親クラス

▶ 〃は*class*の子孫か

▶ 〃は*module*注入済か

▶ 文字列に

▶ nilかどうか

▶ 有するメソの一覧

▶ あるメソを有するか

▶ 破壊的変更を禁止

▶ ２つのオブがﾊｯｼｭのキーとして等しいか

▶ あるオブにだけメソ（特異メソ）を追加

▶ あるオブにだけモ注入

■文字列

▶ 文字列を作成

▶ 特殊な文字を表現

▶ 複数行にわたる文字列

- ハッシュ `{'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee'}`
- 正規表現 `/\d+-\d+/`

■オブジェクト一般

▶ ※ オブジェクトはみな、Objectクラスの子孫クラスのインスタンスである。よって、以下で紹介しているのはObjectクラスが持つメソや機能である。…といいつつ、その親クラスであるBasicObjectクラスのそれが含まれているかもしれないが、さほど問題ではないので許してほしい。

▶ オブに紐づくメソ呼出 *obj*.hoge() **.hoge** **.hoge(*arg1*, ...)** .hoge *arg1*, ...

▶ メソ名の文字列等から〃 *obj*.send('hoge', *arg1*, *arg2*, ...) ※ `:hoge` (シンボル) でも可

▶ nilかもオブにメソ呼出 *obj*&.hoge ※*obj*が `nil` なら `nil` を返す

▶ オブのID *obj*.object_id

▶ 同じ(IDの)オブか *obj1*.equal?(*obj2*)

▶ オブのクラス *obj*.class

▶ 〃が*class*か *obj*.instance_of?(*class*) ※: 真偽値

▶ 〃の親クラス obj.superclass

▶ 〃は*class*の子孫か *obj*.is_a?(*class*) ※㋮ `kind_of?` ※自クラスでも `true`

▶ 〃は*module*注入済か *obj*.is_a?(*module*) ※ 〃

▶ 文字列に *obj*.to_s

▶ nilかどうか *obj*.nil?

▶ 有するメソの一覧 *obj*.methods ※: シンボルの配列

▶ あるメソを有するか *obj*.respond_to?(:関数名)

▶ 破壊的変更を禁止 *obj*.freeze [*obj*.freeze, ...].freeze ※ミュオブにだけやればOK

▶ ２つのオブがﾊｯｼｭのキーとして等しいか *obj1*.eql?(*obj2*) *obj1*.hash == *obj2*.hash

▶ あるオブにだけメソ（特異メソ）を追加
・def obj※[1].メソ名※[2]↲ ‥↲ end ※[1]数値とシンNG
・class << obj※[1]↲ def メソ名※[2]↲ ‥ ※[2]既存OK

▶ あるオブにだけモ注入 obj.intend(モ) obj.extend(モ)

■文字列

▶ 文字列を作成 'やあ' "やあ" %q!やあ! %Q!やあ! %!やあ!

▶ 特殊な文字を表現 \n \r \t ※ `" "` **で囲んでないと無効。**

▶ 複数行にわたる文字列 <<HOGE HOGE で上下から挟んでコード上で改行

▶ 複数行文字列へメソ	'apple'.prepend(<<HOGE) ↲ lemon ↲ banana ↲ HOGE <<HOGE.upcase ↲ Hello, ↲ Good-bye. ↲ HOGE
▶ 文字列中にクオート	" "内なら ' \"　　' '内なら \' "　　%q! !や%Q! !内なら ' "
▶ 文字列の結合	s1 + s2 + ...　　s << *str* #!
▶ 変数展開	"Hello, #{name}"　※ `" "` か `%Q! !` か `%! !` で囲む必要 sprintf('%sは%10dです', 変数1, 変数2)
▶ 文字列の反復	*str* * *n*
▶ 途中の文字	s[*n*]
▶ 途中の複数の文字	s[*n*..*m*]　s[*n*...*m*]　※連続する文字しか取得できないね
▶ TEXTJOIN	[80, 90, 40].join
▶ 数値への変換	*str*.to_i　*str*.to_f
▶ シンへの変換	*str*.to_sym　※㋮ `intern`
▶ 有理数・複素数へ変換	*str*.to_r　・　*str*.to_c
▶ すべて大・小文字に	.upcase　・　.downcase
▶ 文字の配列に	.chars　.each_char.to_a

■数値

▶ カンマ書きたい！	数値の途中に `_` はつけられるのでそれで我慢　例) `2_000_000`
▶ 算術演算子	+　-　*　/　%　**
▶ 算術代入演算子	=　+=　-=　*=　/=　**=
▶ 整数÷整数で端数も	数字なら一方に `.0` をつける　変数なら一方に `.to_f` をつける
▶ 文字列への変換	数値.to_s
▶ 2・8・16進数に変換	数値.to_s(2)　・　.to_s(8)　・　.to_s(16)
▶ 2・8・16進数で表現	0b・0・0x を先頭につけて数字を連ねればよい
▶ m×10^n で表現	1.23e-4
▶ 有理数を表現	2 / 3r　2r / 3　※両者は同じ値になる
▶ 複素数を表現	0.3 - 0.5i
▶ 奇数・偶数かどうか	数値.odd?　・　数値.even?
▶ 商と余り	数値1.divmod(数値2)　※数値1÷数値2　※: [商, 余り]

■配列

▶ 配列を定義	a = [80, 90, 40]

- ▶ 空の配列
- ▶ 均一な値の配列
- ▶ 数列の配列をつくる
- ▶ ☆ 文字列の配列を簡潔につくる（%記法）
- ▶ Splitで配列つくる
- ▶ 要素が1文字の配列
- ▶ １つの要素を参照
- ▶ 複数の要素を参照(?)
- ▶ 最初・最後の複数要素
- ▶ 要素数
- ▶ 要素を追加
- ▶ 末尾に追加
- ▶ 連結
- ▶ 要素を削除
- ▶ 集合演算
- ▶ ランダムに１つ取得
- ▶ シャッフル

- ▶ 単純な繰り返し処理
- ▶ 添え字も取得して〃
- ▶ ※ `do` ～ `end` をブロックという。代わりに `{` ～ `}` でも表せる。また、改行をしないこともできる。
- ▶ ※ ブロックの `| |` の部分に書かれる変数のことをブロック引数という。ブロック内でブロック変数を使わないときは `|` ごと省略することもできる。
- ▶ 条件にあう要素のみに
- ▶ 条件にあう要素を削除
- ▶ 条件にあう最初の要素
- ▶ 要素に一律の変換処理
- ▶ 畳み込み演算
- ▶ COUNTIF的な
- ▶ 全要素が条件にあうか

- ▶ 空の配列　[]　Array.new
- ▶ 均一な値の配列　Array.new(*len*) { ミュオブ }　Array.new(*len*, イミオブ)
- ▶ 数列の配列をつくる　Array.new(*len*) { |n| 一般項 }
- ▶ ☆ 文字列の配列を簡潔につくる（%記法）
- ▶ Splitで配列つくる　*str*.split(*delimiter*)　※ *delimiter* は文字列でも正オでもOK
- ▶ 要素が1文字の配列　*str*.cahrs
- ▶ １つの要素を参照　a[*n*]　※*n*が正で過大なら `nil` 、負で過小ならエラー
- ▶ 複数の要素を参照(?)　a[*n..m*]　a[*n...m*]　a[*n*, *cnt*]　※ `= 値` や `= 配列` の挙動に注意
- ▶ 最初・最後の複数要素　a.first(*cnt*)　・　a.last(*cnt*)　※ *cnt*を省略すると1つに。
- ▶ 要素数　a.size　か　a.length
- ▶ 要素を追加　a[*n*] = 値　※ *n*≥最大添え字+2 ならnilで埋め合わされる
- ▶ 末尾に追加　a << 値 #!　a.push(値1, ...) #!
- ▶ 連結　a1 + a2　[*a1, *a2]　（か　a1.concat(a2) #!　）
- ▶ 要素を削除　a.delete_at(*n*) #!　a.delete(値) #!　※: 削除要素; `nil`
- ▶ 集合演算　a1 | a2　a1 & a2　a1 - a2
- ▶ ランダムに１つ取得　a.sample
- ▶ シャッフル　a.shuffle

- ▶ 単純な繰り返し処理　a.each do |変数1, ...|↲　処理↲　end
- ▶ 添え字も取得して〃　a.each_with_index do |(i1, ...), idx|↲　‥　※1つなら `|i, idx|`
- ▶ ※ `do` ～ `end` をブロックという。代わりに `{` ～ `}` でも表せる。また、改行をしないこともできる。
- ▶ ※ ブロックの `| |` の部分に書かれる変数のことをブロック引数という。ブロック内でブロック変数を使わないときは `|` ごと省略することもできる。
- ▶ 条件にあう要素のみに　a.select { |i| 条件 }　※㋐ `find_all`
- ▶ 条件にあう要素を削除　a.delete_if { |i| 条件 }　#!　a.reject { |i|　〃　}　#!
- ▶ 条件にあう最初の要素　a.find { |i| 条件 }　※㋐ `detect`
- ▶ 要素に一律の変換処理　a.map { |i| 処理 }　※㋐ `collect`
- ▶ 畳み込み演算　a.inject(初項) { |result, i| 処理 }　※㋐ `reduce`
- ▶ COUNTIF的な　a.count { |i| 条件 }　※ 条件が `i==値` なら `a.count(値)` でOK
- ▶ 全要素が条件にあうか　a.all? { |i| 条件 }

- ▶ 条件にあう要素あるか
- ▶ ※ こういった、ブロックを使うメソのことをブロック付きメソッドと呼ぶことがある。
- ▶ 添え字も得てブロメソ
- ▶ ブロメソの省略記法

■範囲（Range）

- ▶ 範囲を定義
- ▶ ある値を含むか
- ▶ 離散化し配列に変換
- ▶ 離散化し繰り返し処理

■ハッシュ

- ▶ ﾊｯｼｭを定義
- ▶ 空のﾊｯｼｭ
- ▶ キーがシンなら略記
- ▶ 初期値を設定して定義
- ▶ 配列をﾊｯｼｭに
- ▶ 要素を参照
- ▶ 要素数
- ▶ 要素を追加、上書き
- ▶ 要素を削除
- ▶ 単純な繰り返し処理
- ▶ キー・値の配列
- ▶ あるキーが存在するか
- ▶ 連結
- ▶ 配列にする
- ▶ ※ 最後の引数がハッシュであればハッシュリテラルの `{ }` を省略できる。

■シンボル

- ▶ シンを表現
- ▶ シンの配列
- ▶ 文字列への変換
- ▶ ※ シンボルは文字列より処理速度が段違いに速い！

- ▶ 条件にあう要素あるか　a.any? { |i| 条件 }
- ▶ ※ こういった、ブロックを使うメソのことをブロック付きメソッドと呼ぶことがある。
- ▶ 添え字も得てブロメソ　a.ブロメソ.with_index { |(i1, ...), idx| 処理 }　※1つなら `|i, idx|`
- ▶ ブロメソの省略記法　a.ブロメソ(&:メソ)

■範囲（Range）

- ▶ 範囲を定義　rng = *start*..*end*　*start*...*stop*　※前者は閉区間, 後者は右開
- ▶ ある値を含むか　rng.include?(値)
- ▶ 離散化し配列に変換　rng.to_a
- ▶ 離散化し繰り返し処理　rng.each { ‥ }　rng.step(*n*) { ‥ }

■ハッシュ

- ▶ ﾊｯｼｭを定義　h = { :abc => 2, :def => true }　※ `'abc' =>` も可　※~~ブロック~~
- ▶ 空のﾊｯｼｭ　{}
- ▶ キーがシンなら略記　h = { abc: 2, def: true }
- ▶ 初期値を設定して定義　h = Hash.new(ミュオブの初期値)　.new { イミオブの初期値 }
- ▶ 配列をﾊｯｼｭに　(a = [[*key1*, *value1*], [*key2*, *value2*], ...] に対して)　a.to_h
- ▶ 要素を参照　h[*key*]　※: *value*; `nil` or 初期値
- ▶ 要素数　a.size　か　a.lengthy
- ▶ 要素を追加、上書き　h[*key*] = *value*
- ▶ 要素を削除　a.delete(***key***) #!　※: 削除要素の値; `nil`
- ▶ 単純な繰り返し処理　h.each do |key, i|
- ▶ キー・値の配列　h.keys　・　h.values
- ▶ あるキーが存在するか　h.has_key?(*key*)　※㋮ `key?` `include?` `member?`
- ▶ 連結　{**h1, **h2}
- ▶ 配列にする　h.to_a　※: [[*key1*, *value1*], [*key2*, *value2*], ...]
- ▶ ※ 最後の引数がハッシュであればハッシュリテラルの `{ }` を省略できる。

■シンボル

- ▶ シンを表現　:apple　:'apple'　:"apple"　%s!apple!
- ▶ シンの配列　%i!apple lemon!　%I!apple lemon!
- ▶ 文字列への変換　*sym*.to_s　※㋮ `id2name`
- ▶ ※ シンボルは文字列より処理速度が段違いに速い！

▶ ※ 同じシンボルは全く同じオブジェクトである。

■正規表現オブジェクト

▶ 正オを表現

▶ 大小文字を区別せぬ〃

▶ .が改行にも合う〃

▶ 空白改行を無視させ〃

▶ ※ 上記のオプションは複数同時に使うこともできる。

▶ *str*に合う箇所あるか

▶ もっと高速に〃

▶ ※ case 文に使うこともできる。その場合は `=~` とか不要。

▶ 〃ないか

▶ 名つきｷｬﾌﾟﾁｬを変数に

▶ MatchDataオブ

▶ ﾊﾟﾀｰﾝ全体が合う箇所

▶ *n*番目のｷｬﾌﾟﾁｬが〃

▶ 1~n番目の〃の配列

▶ 名前つきｷｬﾌﾟﾁｬが〃

▶ 置換

▶ 場合分けして置換

▶ ※ 同じシンボルは全く同じオブジェクトである。

■正規表現オブジェクト

▶ 正オを表現 */pattern/* %r!*pattern*! Regexp.new('*pattern*')

▶ 大小文字を区別せぬ〃 */pat*/i %r!*pat*!i Regexp.new('*pat*', Rexexp::IGNORECASE)

▶ .が改行にも合う〃 */pat*/m %r!*pat*!m Regexp.new('*pat*', Rexexp::MULTILINE)

▶ 空白改行を無視させ〃 */pat*/x %r!*pat*!x Regexp.new('*pat*', Rexexp::EXTENDED)

▶ ※ 上記のオプションは複数同時に使うこともできる。

▶ *str*に合う箇所あるか *str* =~ *regexp* *regexp* =~ *str* ※: 開始位置; `nil`

▶ もっと高速に〃 *regexp*.match?(*str*) *str*.match?(*regexp*) ※: 真偽値

▶ ※ case 文に使うこともできる。その場合は `=~` とか不要。

▶ 〃ないか *str* !~ *regexp* *regexp* !~ *str* ※: `true` ; `false`

▶ 名つきｷｬﾌﾟﾁｬを変数に rexexp =~ *str* ※逆は不可 ※*regexp*は変数に入っててはダメ

▶ MatchDataオブ **m** = *regexp*.match(*str*) *str*.match(*regexp*)

▶ ﾊﾟﾀｰﾝ全体が合う箇所 **m**[0] *str*[*regexp*] ※後者の代 `slice` 。以降の `[]` も同じ

▶ *n*番目のｷｬﾌﾟﾁｬが〃 **m**[*n*] *str*[*regexp*, *n*] ※*n*は負でも可

▶ 1~n番目の〃の配列 *str*.scan(*regexp*) ※ `()` がネストされてると多次配列に

▶ 名前つきｷｬﾌﾟﾁｬが〃 **m**[:*name*] **m**['*name*'] *str*[*regexp*, :*name*] *str*[*regexp*, '*name*']

▶ 置換 *str*.gsub(*regexpOld*, *new*※) .gsub!(〃) #! ※ `\n` `\k<name>` ｱﾘ

▶ 場合分けして置換 str.gsub(*reOld*, *hashAboutRule*) .gsub(*reOld*) { |matched| ‥ }

クラス編

■クラスの用語とその役割

▶ インスタンスメソッド

▶ クラスメソッド

▶ 定数

▶ インスタンス変数

▶ ｸﾗｽｲﾝｽﾀﾝｽ変数

▶ クラス変数

▶ ☆ どこからアクセス可能か

クラス編

■クラスの用語とその役割

▶ インスタンスメソッド ｲﾝｽﾀﾝｽ化してできたオブをレシーバにして呼び出せるメソ

▶ クラスメソッド クラスをレシーバにして呼び出せるメソ

▶ 定数 値を変更しない変数

▶ インスタンス変数 ｲﾝｽﾀﾝｽ化してできたオブごとに管理される変数

▶ ｸﾗｽｲﾝｽﾀﾝｽ変数 クラスごとに管理される変数

▶ クラス変数 クラスの家系ごとに管理される変数

▶ ☆ どこからアクセス可能か

■クラスの定義

- ▶ クラスを定義
- ▶ インスタンスメソッド
- ▶ コンストラクタ
- ▶ 定数
- ▶ 〃への再代入を禁止
- ▶ 〃の破壊的変更を禁止
- ▶ ｸﾗｽ外から〃参照を禁止
- ▶ インスタンス変数
- ▶ 〃を読み書き可能に
- ▶ クラスメソッド
- ▶ ｲﾝｽﾀﾒｿ内で〃呼ぶ
- ▶ ｸﾗｽ定義直下で〃呼ぶ
- ▶ ※ `def hoge(self, …)` などと `self` をメソに渡すとエラーになる。
- ▶ ※ `self.hoge` などと前につけてメソを呼び出してもエラーにはならないが、つけないことも多い。ただし、**セッターメソッド `hoge=` を呼び出すのなら必須**（ `self.hoge =` ）。反対に、**privateメソッドにはつけてはならない。**
- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数
- ▶ 〃を読み書き可能に
- ▶ クラス変数
- ▶ 〃を読み書き可能に

- ▶ ※ クラス定義は別のクラス定義のなかに書ける（ネストできる）。
- ▶ ※ クラス定義はモジュール定義のなかに書ける（ゆえに**名前空間をつくれる**）。
- ▶ 別ｸﾗ・モ内の別ｸﾗ参照
- ▶ ﾄｯﾌﾟﾚﾍﾞﾙの別ｸﾗ参照
- ▶ ☆ あるメソを自身は実装していないが子クラスには実装を要請する
- ▶ ※ クラスは同一のソースファイル上で再オープンして編集できる。

公開レベル

■クラスの定義

- ▶ クラスを定義　class FooFoo↲　‥↲　end
- ▶ インスタンスメソッド　def hoge_hoge(p1, …)↲　‥↲　end
- ▶ コンストラクタ　def initialize(p1, …)↲　‥↲　end
- ▶ 定数　HOGE_HOGE = 値
- ▶ 〃への再代入を禁止　‥↲　freeze　※やる人はまずいない
- ▶ 〃の破壊的変更を禁止　HOGE = 'abc'.freeze　HOGE = ['abc'.freeze, …].freeze　(cf.)
- ▶ ｸﾗｽ外から〃参照を禁止　private_constant :HOGE, :FUGA, …　※やる人は少ない
- ▶ インスタンス変数　ｲﾝｽﾀﾒｿ内で　@hoge　※未初期化で参照すれば `nil`
- ▶ 〃を読み書き可能に　attr_accessor :hoge, :fuga, …
  ※読・書のみ可能にしたいなら `attr_reader` ・ `attr_writer` に
- ▶ クラスメソッド
  ・def self.hoge↲　‥↲　end
  ・class << self↲　def hoge↲　‥↲　end↲　end
- ▶ ｲﾝｽﾀﾒｿ内で〃呼ぶ　Foo.hoge　か　self.class.hoge　※ 前者だと継承時困るかもね
- ▶ ｸﾗｽ定義直下で〃呼ぶ　self.hoge　※ ただし `self.hoge` が定義されたより下で可能
- ▶ ※ `def hoge(self, …)` などと `self` をメソに渡すとエラーになる。
- ▶ ※ `self.hoge` などと前につけてメソを呼び出してもエラーにはならないが、つけないことも多い。ただし、**セッターメソッド `hoge=` を呼び出すのなら必須**（ `self.hoge =` ）。反対に、**privateメソッドにはつけてはならない。**
- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数　クラス定義直下またはクラスメソ内で　@hoge
- ▶ 〃を読み書き可能に　class << self↲　attr_accessor :hoge, …　※適宜 _reader _writer に
- ▶ クラス変数　@@hoge　※変更すると家系全体のそれも変更してしまう！
- ▶ 〃を読み書き可能に　attr_accessorなどは使えないので自分でアクセサメソを定義する

- ▶ ※ クラス定義は別のクラス定義のなかに書ける（ネストできる）。
- ▶ ※ クラス定義はモジュール定義のなかに書ける（ゆえに**名前空間をつくれる**）。
- ▶ 別ｸﾗ・モ内の別ｸﾗ参照　外部クラス::内部クラス　・　モ::クラス
- ▶ ﾄｯﾌﾟﾚﾍﾞﾙの別ｸﾗ参照　::クラス　※ `::` 不要な場合も、明示したいときにはアリ。
- ▶ ☆ あるメソを自身は実装していないが子クラスには実装を要請する
- ▶ ※ クラスは同一のソースファイル上で再オープンして編集できる。

公開レベル

- ※ メソッドには public、private、protected の３つの公開レベルがある。
- publicにすると
- privateにすると
- protectedにすると
- ※ privateメソも protectedメソも継承される！（子クラスでも呼び出せる）
- publicメソを定義
- privateメソを定義
- protectedメソを定義
- 後から公開レベル変更
- ※ クラス自体にたいして公開レベルを設定することはできない。定義されたクラスはすべてpubilcになる。

継承

- ※ 継承すれば is-a の関係になる。
- ※ Rubyは単一継承。（ただし、ミックスインという多重継承に似た機能がある）
- ※ 作成したクラスはデフォルトでは Object クラスを(直接)継承している。
- クラスを継承
- ※ メソッドのオーバーライド（上書き）は可能。
- ｲﾝｽﾀﾝｽﾒソ内で親の同名のそれを呼ぶ
- ※ クラスメソッドも継承される。
- ※ 意図しないオーバーライドやインスタンス変数の上書きを防ぐために、親クラスの実装を把握しておくべきである。

モジュール

- ｲﾝｽﾀﾝｽﾒソとして注入
- クラス定義より優先で ｲﾝｽﾀﾝｽﾒソとして注入
- クラスメソとして注入

■クラスの操作

- クラスメソッド
- 定数
- 定数への再代入を禁止

- ※ メソッドには public、private、protected の３つの公開レベルがある。
- publicにすると　クラスの外部からでも自由に呼び出せる
- privateにすると　レシーバ.メソ ではなく メソ と書くことで呼び出せる
- protectedにすると　クラス外部から呼べないが、中だと レシーバ.メソ で呼べる
- ※ privateメソも protectedメソも継承される！（子クラスでも呼び出せる）
- publicメソを定義　デフォではpubilicメソ　※ public↲ def hoge↲ ‥ でもOK
- privateメソを定義　private↲ def hoge↲ ‥↲ end↲ def fuga↲ ‥
- protectedメソを定義　protected↲ 〃
- 後から公開レベル変更　private :hoge, :fuga　など
- ※ クラス自体にたいして公開レベルを設定することはできない。定義されたクラスはすべてpubilcになる。

継承

- ※ 継承すれば is-a の関係になる。
- ※ Rubyは単一継承。（ただし、ミックスインという多重継承に似た機能がある）
- ※ 作成したクラスはデフォルトでは Object クラスを(直接)継承している。
- クラスを継承　class Foo < Bar↲ ‥　※この場合Barが親。
- ※ メソッドのオーバーライド（上書き）は可能。
- ｲﾝｽﾀﾝｽﾒソ内で親の同名のそれを呼ぶ　super(*arg1*, ...)　※引数同じなら `super` だけでOK
- ※ クラスメソッドも継承される。
- ※ 意図しないオーバーライドやインスタンス変数の上書きを防ぐために、親クラスの実装を把握しておくべきである。

モジュール

- ｲﾝｽﾀﾝｽﾒソとして注入　include Piyo
- クラス定義より優先で ｲﾝｽﾀﾝｽﾒソとして注入　prepend Piyo
※ クラスで定義したｲﾝｽﾀﾝｽﾒソより先にモのメソが呼ばれる
- クラスメソとして注入　extend Piyo

■クラスの操作

- クラスメソッド　Foo.hoge
- 定数　Foo::HOGE　※実は値の変更ができてしまう！(警告は出る)
- 定数への再代入を禁止　Foo.freeze　※やる人はまずいない

- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数
- ▶ クラス変数
- ▶ あるモが注入済か
- ▶ 注入済のモ
- ▶ メソ探索の経路を確認
- ▶ クラス定義より優先で
  ｲﾝｽﾀﾒソとしてモ注入

- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数　Foo.hoge　Foo.hoge =
- ▶ クラス変数　Foo.hoge　Foo.hoge =
- ▶ あるモが注入済か　Foo.include?(Piyo)
- ▶ 注入済のモ　Foo.included_modules　※: モ名の配列
- ▶ メソ探索の経路を確認　Foo.ancestors
- ▶ クラス定義より優先で
  ｲﾝｽﾀﾒソとしてモ注入　Foo.prepend Piyo #

モジュール編

■モジュールの用途

- モジュールの注入（ include と extend ）
- モジュールを利用した名前空間の作成
- 関数や定数を提供するモジュールの作成
- 状態を保持するモジュールの作成
- 上記の組み合わせ

■モジュールの定義

- ▶ ※ モジュールは、is-a の関係にないが、ある機能を複数のクラスにまたがって実装させたいときに使う。
- ▶ モジュールを定義
- ▶ ※ モジュールは、そのインスタンスを作成できない。
- ▶ ※ モジュールは、継承できない。
- ▶ ※ モジュールで設定したメソッドの公開レベルは、注入されても保持される。
- ▶ ※ インスタンス変数の値を更新したいなら、`@hoge =` と直接かきかえようとするのではなく、`self.hoge =` とセッターメソッドを経由するのがよい。
- ▶ ※ 「モジュールオブジェクトの特異メソ」（モジュール関数）を定義することも可能。
- ▶ モ内のメソを**モ関数**
  としても流用したい
- ▶ ※ ｸﾗｽｲﾝｽﾀﾝｽ変数を設定することで、状態を保持するモジュールを作成できる。
- ▶ ※ クラス変数を設定することは避けるべきなんじゃないかな。
- ▶ 別のモを注入

モジュール編

■モジュールの用途

- モジュールの注入（ include と extend ）
- モジュールを利用した名前空間の作成
- 関数や定数を提供するモジュールの作成
- 状態を保持するモジュールの作成
- 上記の組み合わせ

■モジュールの定義

- ▶ ※ モジュールは、is-a の関係にないが、ある機能を複数のクラスにまたがって実装させたいときに使う。
- ▶ モジュールを定義　module PiyoPiyo↲　‥↲　end
- ▶ ※ モジュールは、そのインスタンスを作成できない。
- ▶ ※ モジュールは、継承できない。
- ▶ ※ モジュールで設定したメソッドの公開レベルは、注入されても保持される。
- ▶ ※ インスタンス変数の値を更新したいなら、`@hoge =` と直接かきかえようとするのではなく、`self.hoge =` とセッターメソッドを経由するのがよい。
- ▶ ※ 「モジュールオブジェクトの特異メソ」（モジュール関数）を定義することも可能。
- ▶ モ内のメソを**モ関数**
  としても流用したい　・module_function :hoge, :fuga, …　※hogeら定義後に書け
  ・module_function↲ def hoge↲ ‥↲ end↲ def fuga↲ ‥
- ▶ ※ ｸﾗｽｲﾝｽﾀﾝｽ変数を設定することで、状態を保持するモジュールを作成できる。
- ▶ ※ クラス変数を設定することは避けるべきなんじゃないかな。
- ▶ 別のモを注入　include モ

▶ ※ モジュールは同一のソースファイル上で再オープンして編集できる。

■組み込みモジュール

- ▶ Enumerable モ
- ▶ Comparable モ
- ▶ Kernel モ
- ▶ Math モ

■モジュールの操作

- ▶ モジュール関数
- ▶ 定数
- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数

▶ ※ モジュールは同一のソースファイル上で再オープンして編集できる。

■組み込みモジュール

- ▶ Enumerable モ　繰り返し処理を可能にする
- ▶ Comparable モ　比較演算を可能にする
- ▶ Kernel モ　便利メソッドを提供
- ▶ Math モ　数学の計算用のメソや定数を提供

■モジュールの操作

- ▶ モジュール関数　Piyo.hoge
- ▶ 定数　Piyo::HOGE
- ▶ ｸﾗｽｲﾝｽﾀﾝｽ変数　Piyo.hoge

ライブラリ編

■ライブラリ

- ▶ ☆ ライブラリの分類
- ▶ ※ 組み込みライブラリ以外のライブラリは、利用するにあたり読み込む必要がある。
- ▶ ラを読み込む(初)
- ▶ 自作ラを読み込む(初)
- ▶ ラを読み込む(再,初)
- ▶ 自作ラを読み込む(再,初)

■gem

- ▶ ※ gem とは、Ruby における外部ライブラリのことである。
- ▶ ※ gem に関する情報は、RubyGems.org にアクセスすると確認できる。
- ▶ インストール時間を短縮する設定に(gemドキュをインストールしない)

Bundlerでgemを管理する場合に限らない(？)

- ▶ gemをインストール
- ▶ バージョン指定してgemをインストール

ライブラリ編

■ライブラリ

- ▶ ☆ ライブラリの分類
- ▶ ※ 組み込みライブラリ以外のライブラリは、利用するにあたり読み込む必要がある。
- ▶ ラを読み込む(初)　require 'ラ名'
- ▶ 自作ラを読み込む(初)　require '**Rubyを実行しているディ**に対するラの相対パス'　か　require '絶対パス'　※ ←↑↓ はどれも `.rb` を省略可能　require_relative '**実行ファが存在するディ**に対するラの相対パス'
- ▶ ラを読み込む(再,初)　load 'ラ'
- ▶ 自作ラを読み込む(再,初)　load '**Rubyを実行しているディ**に対するラの相対パス'　か　load '絶対パス'

■gem

- ▶ ※ gem とは、Ruby における外部ライブラリのことである。
- ▶ ※ gem に関する情報は、RubyGems.org にアクセスすると確認できる。
- ▶ インストール時間を短縮する設定に(gemドキュをインストールしない)　$ echo "gem: --no-document" >> ~/.gemrc　※実行は1度きりでいい

Bundlerでgemを管理する場合に限らない(？)

- ▶ gemをインストール　$ gem install *gemName*
- ▶ バージョン指定してgemをインストール　$ gem install *gemName* -v *ver*

- ▶ インストール済の全gemの一覧
- ▶ インストール済のgemをキーワード検索
- ▶ インストール済のgemを使いコマンド実行
- ▶ インストール済のgemをアンインストール

Bundlerでgemを管理する場合に限り

- ▶ ※ Bundlerを使うことで、複数のgemの依存関係を保ちながらgemの管理ができる。
- ▶ Bundlerをインストール
- ▶ 現在の作業ディ内に Gemfile を作成
- ▶ ☆ Gemfile を編集
- ▶ Gemfile に沿ってgemをインストール
- ▶ インストール済の全gemの一覧
- ▶ インストール済のgemを使いコマンド実行
- ▶ インストール済のgemのバージョンを確認
- ▶ インストール済のgemのバージョンを更新
- ▶ ※ 以上の `bundle` コマンドは `$ bundle _ver_ ～` とすることでBundlerのバージョンを指定して実行できる。

応用編

■時間

- ▶ 特定の日時のTimeオブ
- ▶ Date, DatieTimeｸﾗｽを読み込む
- ▶ 特定の日付のDateオブ
- ▶ 特定の日時のDateTimeオブ
- ▶ ※ Timeクラスのほうが `require` なしで使える点と、サマータイムやうるう秒を扱える点で優位である。また、TimeクラスとDateTimeクラスとではタイムゾーンが異なる。

■システム

- ▶ 環境変数の値を取得

■ファイル・ディレクトリの操作

ファイル

- ▶ インストール済の全gemの一覧　$ gem list
- ▶ インストール済のgemをキーワード検索　$ gem list キーワード
- ▶ インストール済のgemを使いコマンド実行　$ *command*※　※ `gemName ～`
- ▶ インストール済のgemをアンインストール　$ gem uninstall *gemName*

Bundlerでgemを管理する場合に限り

- ▶ ※ Bundlerを使うことで、複数のgemの依存関係を保ちながらgemの管理ができる。
- ▶ Bundlerをインストール　$ gem install bundler　※Ruby 2.5～ 不要
- ▶ 現在の作業ディ内に Gemfile を作成　$ bundle init
- ▶ ☆ Gemfile を編集
- ▶ Gemfile に沿ってgemをインストール　$ bundle install
- ▶ インストール済の全gemの一覧　$ bundle list
- ▶ インストール済のgemを使いコマンド実行　$ bundle exec *command*※　※ `gemName ～`
- ▶ インストール済のgemのバージョンを確認　$ bundle exec *gemName* -v
- ▶ インストール済のgemのバージョンを更新　$ bundle update *gemName*
- ▶ ※ 以上の `bundle` コマンドは `$ bundle _ver_ ～` とすることでBundlerのバージョンを指定して実行できる。

応用編

■時間

- ▶ 特定の日時のTimeオブ　Time.new(*y*, *month*, *d*, *h*, *minute*, *s*)
- ▶ Date, DatieTimeｸﾗｽを読み込む　require 'date'
- ▶ 特定の日付のDateオブ　Date.new(*y*, *m*, *d*)
- ▶ 特定の日時のDateTimeオブ　DateTime.new(*y*, *month*, *d*, *h*, *minute*, *s*)
- ▶ ※ Timeクラスのほうが `require` なしで使える点と、サマータイムやうるう秒を扱える点で優位である。また、TimeクラスとDateTimeクラスとではタイムゾーンが異なる。

■システム

- ▶ 環境変数の値を取得　ENV['環境変数']

■ファイル・ディレクトリの操作

ファイル

▶ ファが作業ディにあるか

ディレクトリ

▶ ディが作業ディにあるか

▶ ファが作業ディにあるか File.exists?('*path*')

ディレクトリ

▶ ディが作業ディにあるか Dir.exists?('*path*')