

# Ruby 講義

## 第14回 夏学期総復習

五十嵐邦明

twitter : igaiga555

<http://www.facebook.com/kuniaki.igarashi>



2013.7.18 at 一橋大学  
ニフティ株式会社寄附講義  
社会科学における情報技術と  
コンテンツ作成III

# 五十嵐邦明 講師 株式会社万葉



twitter: igaiga555

<https://github.com/igaiga/>

<http://www.facebook.com/kuniaki.igarashi>

# 濱崎 健吾

Teaching Assistant  
fluxflex, inc(米国法人)



twitter: hmsk

<https://github.com/hmsk/>

<http://www.facebook.com/hamachang>

# 夏学期総復習

ここまでやったことで大事  
な部分をまとめました。

講義中に全部は説明できな  
いと思うのですが、復習用  
に使ってください。

# Rubyコード の実行方法

# 3つの世界

Ruby(irb)

Ruby語が通じる世界  
1行ずつコードを実行

Ruby(ファイル)

Ruby語が通じる世界  
ファイルにコードを書いて実行

Shell

OS語が通じる世界  
ターミナルの中

shell

# ターミナル

**shell**環境はターミナルから操作できます。

起動方法

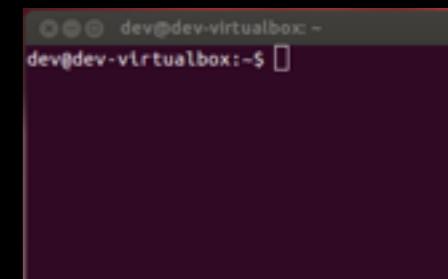
Mac : ターミナル

Win : "Command Prompt with Ruby and Rails"

Linux(Ubuntu) : 端末(※分からぬ場合は次のページ参照)

※ ↑ どれも今後はターミナルと呼びます。 (ターミナルの和訳が端末)

起動して以下のように入力してEnterを押してください。



# ターミナルの便利機能

ターミナルで使える便利な機能を紹介します。

↑キー：過去に入力したコマンドの履歴を  
遡って実行できます。

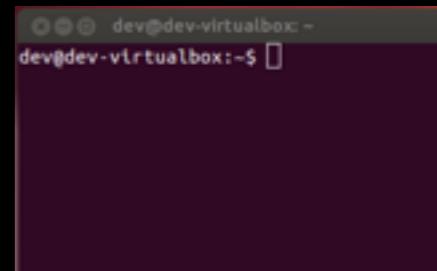
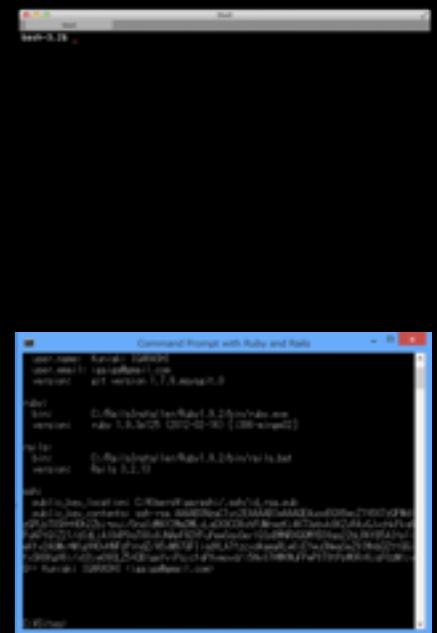
tabキー：途中まで入力した状態で押すと  
残りを補完してくれます。

たとえば、hello.rb というファイルを実行したいときに

\$ ruby he まで打って tab キーを押すと

\$ ruby hello.rb と補完されます。

補完候補が複数ある場合は候補を表示してくれます。



# よく使うshellコマンド

カレントフォルダ（現在のフォルダ）を移動

**\$ cd フォルダ名**

1つ上のフォルダへ移動

**\$ cd ..**

※ cd の後にピリオド2つ

カレントフォルダを表示

**\$ pwd**

カレントフォルダのファイルを表示

**\$ ls**

※コマンドは \$ 始まりで書いています。

この\$は始まりの印（プロンプトと言います）なので打たなくて大丈夫です。

# shellコマンド - フォルダ操作

フォルダーを作るコマンドです。

**\$ mkdir フォルダ名**

ちなみに消すのは rmdir コマンドです。

**\$ rmdir フォルダ名**

フォルダの中が空でないとrmdir では削除できません。

フォルダの中にファイルなどがあるのに消したい場合は rm -rf コマンドで削除できます。※削除したファイルは復元できないので注意して使ってください！！

★危険★

**\$ rm -rf フォルダ名**



# Rubyコード の実行

## irb

# irb (Interactive RuByの略)

irb はrubyコードを1行ずつ実行する環境です。  
ターミナルから以下のコマンドを実行します。

\$ irb

※先頭の \$ はターミナルであることを表すマークです。  
(\$と、その次のスペースは入力不要です。 irbと打てばOK。 )  
今後、ターミナルで打つコマンドは同じ書式  
(紫の背景色、 \$ マーク)で書きます。

こんな風に表示が出ればOKです。

2.0.0p0 :001 >

※irbを終了させるには exit と打ちます。

# Rubyコード の実行 ファイル編

# Rubyコードをファイルに記述して実行

## 1. エディタを起動します。

インストールしたエディタを起動します。

お勧めエディタ

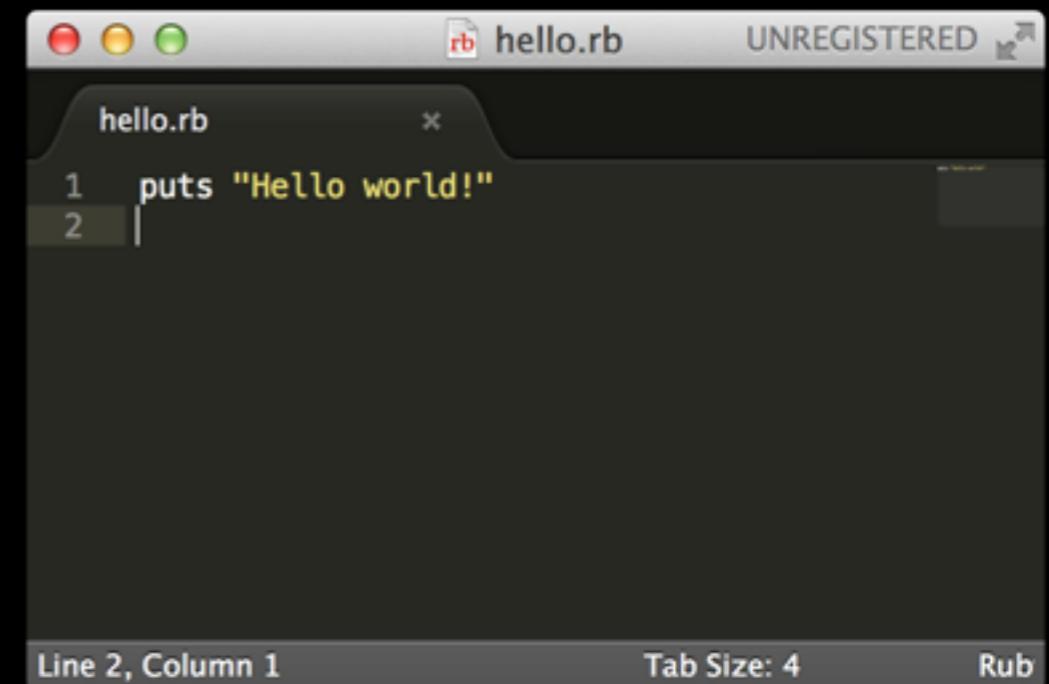
Win, Mac : SublimeText2

Linux(Ubuntu) : gedit

## 2. プログラムを入力します。

例：

**puts("Hello world!")**



A screenshot of the Sublime Text 2 interface. The window title is 'hello.rb'. The code editor contains the following text:

```
1 puts "Hello world!"
```

The status bar at the bottom shows 'Line 2, Column 1' and 'Tab Size: 4'. The tab bar at the top right shows 'UNREGISTERED'.

## 3. hello.rbという名前で保存します。

手順0. 作ったフォルダの下に保存してください。

※名前は .rb を付ければ、他の名前でもOKです。

# Rubyコードをファイルに記述して実行

4. ターミナルを起動します。

5. cd コマンドでhello.rb を保存したフォルダへ移動します。

\$ cd src

※pwdコマンドを使うと現在のフォルダを確認できます。 \$ pwd

6. hello.rb を実行します。

\$ ruby hello.rb

※ruby [ファイル名] で実行

Hello world! と表示されれば成功です。

成功例

\$ ruby hello.rb ↵enter  
Hello world!

※以下のエラーが出たときは lsコマンドでファイル有無を確認します。  
ruby: No such file or directory -- XXX.rb (LoadError)

総復習

Ruby 言語

# オブジェクト



教科書  
p.8

**2 : 整数(Fixnum)オブジェクト**

**3.14 : 小数 (浮動小数点数) オブジェクト  
(Floatオブジェクト)**

**"ちはやふる" : Stringオブジェクト**

**[1,2,3] : Arrayオブジェクト**

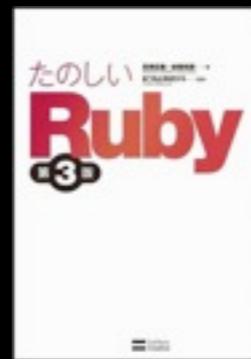
**{:alice => "A", :bob => "B"} : Hashオブジェクト**

`{alice : "A", bob : "B"} (←Ruby1.9以降はこう書くこともできます。)`

**nil : 無いことを表すオブジェクト**

**ほか、いろいろあります。**

# メソッドと引数



教科書  
p.9

`print("Hello, Ruby.\n")`

メソッド

引数

メソッド：手続き、命令

引数：メソッドに渡すデータ

`print`メソッドは画面に引数のデータを表示する命令

`print`メソッドに

`"Hello, Ruby.\n"` オブジェクトを  
引数として渡しています。

# 画面に表示するメソッド



よく使うので似た機能のメソッドが3つあります。

**print** : 表示(改行しない)

**puts** : 表示(改行する)

**p** : 調査(デバッグ)用

※用語解説 : デバッグ

バグ (不具合) を解消すること



# 変数

## オブジェクトへのラベル・荷札

`name = "igarashi"`

`name` という名前の変数に  
"igarashi" オブジェクトを代入しています。

変数は代入されたオブジェクトを書いたときと  
同じように振る舞います。  
以下の2つは同じ結果になります。

`puts name`

`puts "igarashi"`

igarashi

igarashi

実行結果

実行結果

# 名付け重要

変数名は分かりやすい名前にしよう

良い例

```
width = 20  
height = 3  
area =  
    width * height
```

悪い例

```
a1 = 20  
a2 = 3  
a3 =  
    a1 * a2
```

# オブジェクトをイメージする

```
a = "abc"
```

```
b = a
```

```
a.upcase!
```

```
puts a  
puts b
```

※`upcase!` は  
`String`オブジェクトを  
大文字にするメソッド

`a` は "ABC" になりますが、  
`b` はどうなるでしょう？

`a = "abc"`

`b = a`

`a.upcase!`

`puts a`  
`puts b`

aはオブジェクト”abc”を示す変数

bもaと同じ”abc”を示す変数

aの指すオブジェクトを大文字にする

`a → "ABC"`

`b → "ABC"`

変数

オブジェクト

a

abc

a  
b

abc

a  
b

ABC

# さっきっと似てるけどちょっと違うコード

```
a = "abc"
```

```
b = "abc"
```

```
a.upcase!
```

```
puts a
```

```
puts b
```

新しいコード

さっちは→  
こうでした

```
a = "abc"
```

```
b = a
```

```
a.upcase!
```

```
puts a
```

```
puts b
```

さっきのコード

`a = "abc"`

`b = "abc"`

`a.upcase!`

`puts a`

`puts b`

aはオブジェクト”abc”を示す変数

bは別のオブジェクト”abc”を指す

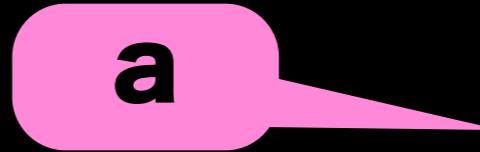
aの指すオブジェクトを大文字にする

`a → "ABC"`

`b → "abc"`

変数

オブジェクト



# 四則演算

**1 + 2**

**2 - 3**

**5 \* 10**

**100 / 4**

**12 % 5 (=2)**

**2\*\*32**

**10/3 (=3(3.333にならぬので注意))**

**10/0 (= ゼロ除算エラー)**

**+ : 足し算**

**- : 引き算**

**\* : 掛け算**

**/ : 割り算**

**% : 剰余(余り)**

**\*\* : 累乗**

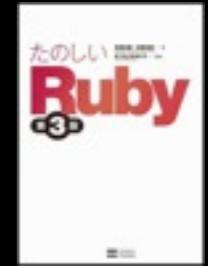
**logとかsinとかもあります。  
知りたい方はこちら。**

# エラーメッセージは お得な情報を教えてくれる

Rubyが教えてくれたエラーメッセージ  
**helloerror.rb:2:in `<main>': undefined  
method `prin' for main:Object  
(NoMethodError)**

日本語訳

**helloerror.rb** というファイルの **2** 行目で  
prinなんてメソッドはないので  
そんなメソッドないよエラー が起きたよ



教科書  
p.23

# コメント文

コードの中にある実行されない文  
コードの説明を書いたりします。

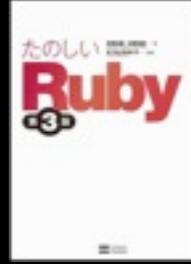
コメント文書式：# 以降はコメント文

```
# name = "igarashi"  
width = 2 # 文の途中からでもOK
```

↑の場合、以下の色塗りの部分がコメントになります。

```
# name = "igarashi"  
width = 2 # 文の途中からでもOK
```

# 条件判断：if 文



教科書  
p.25-27

**if 条件**

条件が成立したときに実行したい処理

**end**

※教科書には **then** が書いてありますが、省略可能です。

普通は省略します。私は書いたことないです。

条件には値が **true(真)** または **false (偽)** となる式を書くことが一般的

# 条件判断 == 演算子

```
x = 3 - 2
```

```
if x == 1
```

```
  puts "x is 1"
```

```
end
```

x が 1 と同じか判断し  
x が 1 の時に  
puts が実行されます。

== は左辺(x)と右辺(1)が同じかどうか調べて、  
同じならば true、異なる場合は false になります。

== が2個です。= が1つだと代入になってしまってるので注意。  
ちなみに、異なるかを判断する != もあります。  
ほかにも >, >=, <, <= なども使えます。

# インデント(字下げ)

```
if x == 1  
  puts "x is 1"
```

↑  
ind

例えばif文中など、こういう風に先頭にスペースを入れて書くことをインデントするといいます。

プログラムの実行には不要なのですが、

**絶対**に入れてください！

無いと人が読めないので・・・

ちなみにスペースの個数には流派がありますが、2個が主流のようです。

# 条件判断 if - else - end

**if 条件**

条件が成立した時に実行したい処理

**else**

条件が不成立の時に実行したい処理

**end**

条件が不成立の時に実行したい処理を書く  
こともできます。

# if 文は後ろにも書ける

条件成立時に実行したい処理 if 条件

```
if x == 1  
  puts "x is 1"  
end
```

左の文は以下のよう  
に1行で書くことも  
できます。

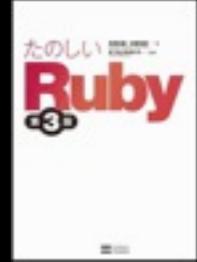
```
puts "x is 1" if x == 1
```

1行で書ける条件

- ・ 実行したい処理が1行だけのとき
- ・ else節を書かないとき

# 繰り返し

# 繰り返し : while文



教科書  
p.27

while 繰り返し続ける条件

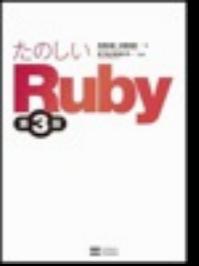
繰り返したい処理

end

1から10までの数を  
順番に表示するコード

```
i = 1
while i <= 10
    puts i
    i = i + 1
end
```

# 繰り返し : times文



教科書  
p.28

繰り返す回数.times do

繰り返したい処理

end

※doを  
忘れずに!

Ruby と5回表示する

5.times do  
  puts "Ruby"  
end

メソッド

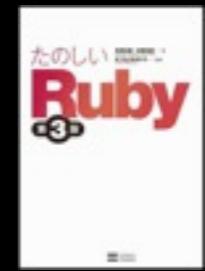
# メソッド

`puts` や `print` はメソッドだと前に説明しました。  
メソッドは「コード群を機能単位で集めたもの」と言えます。

`puts` や `print` は「表示する」ための機能を集めて提供しているメソッドです。これらはRubyがあらかじめ用意しているメソッドの中の1つです。

一方で、メソッドは自分で作ることもできます。これを「メソッドを定義する」と言います。次のページで定義方法を見ていきましょう。

# メソッドの定義、呼び出し



教科書  
p.28

メソッド定義には  
**def** を使います

```
def メソッド名
  メソッドで実行したい処理
end
```

定義

```
def hello
  puts "Hello"
end
```

呼び出し

```
hello()
```

hello()を呼ぶと、事前に定義していたhelloメソッドが呼ばれ実行されます。 (ここでは puts "Hello")  
hello()の()は省略可能です。 (曖昧にならない限り)

# 引数付きのメソッド

引数の仕組みを使って  
メソッドにデータを渡  
すことができます。

```
def メソッド名(引数)  
    メソッドで実行したい処理  
end
```

定義

```
def hello(word)  
    puts word  
end
```

呼び出し

```
hello("Hi")
```

どのように動作するか次のページで説明します

# 引数付きのメソッド

定義時にメソッド名に続いて(変数名)を書くと、呼び出し時に渡された引数をその変数に代入した状態でメソッドを実行できます。

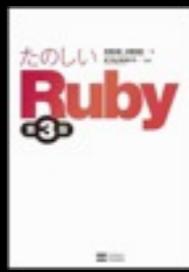
## 定義

```
def hello(word)
  puts word
end
```

## 呼び出し

```
hello("Hi")
```

`hello("Hi")`と呼び出すと、`hello` メソッドが呼ばれます。その際に `word = "Hi"` の代入が行われます。その後メソッドの中身が実行されます。`puts word` が実行され、`word` には "Hi" が代入されているので、動作としては "Hi" が表示されます。



# require

メソッド定義などを別のファイルに書き、読み込むことができます。

**hello.rb**

```
def hello
  puts "Hello"
end
```

**use\_hello.rb**

```
require "./hello"
hello()
```

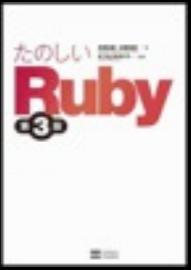
↑教科書は `require "hello"` になっていますが、Ruby1.9.2以降だとエラーになるので `"./hello"` としてください。 `./` は(shellで)今いるフォルダの意味です。

**実行**

上記2つのファイルを同じフォルダに置いて以下のコマンド

```
$ ruby use_hello.rb
```

# Array



# 配列(Array)

## ほかのオブジェクトの入れもの

例)

```
names = ["五十嵐", "濱崎"]  
numbers = [1,3,5]
```

[と]で囲い, で区切る。

文字列や数字ほか、どんなオブジェクトも入ります。

空っぽの配列は [] です。

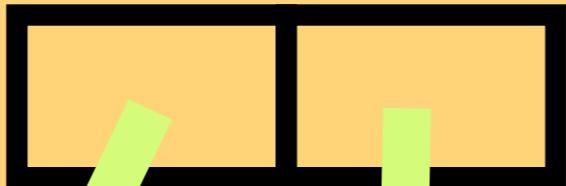
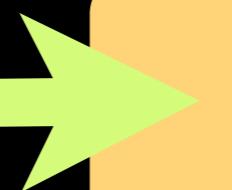
概念図を次のページに示します。

# 配列(Array)概念図

```
names = ["五十嵐", "濱崎"]
```

変数

● names



Array  
オブジェクト

五十嵐

濱崎

String  
オブジェクト

(教科書p.35 図2.2 も参照。)

# 配列から読み込む

番号(index)を指定して読み込み

`names = ["五十嵐", "濱崎"]`

`names[0] → "五十嵐"`

`names[1] → "濱崎"`

配列に[n]と書くと、n番目の要素を返します。

最初の要素は0番です。1始まりではないので注意です。

負数を指定すると末尾から読み込みます。(-1始まり)

`names[-1] → "濱崎"`

`names[-2] → "五十嵐"`

# 配列へ追加する

配列の末尾にオブジェクトを追加するには  
pushメソッドを使います。

```
names = ["五十嵐", "濱崎"]
```

```
names.push("山田")
```

```
p names → ["五十嵐", "濱崎", "山田"]
```

配列に入っている要素数を調べるには sizeメソッド

```
names = ["五十嵐", "濱崎", "山田"]
```

```
p names.size → 3
```

# 配列の繰り返し処理



教科書  
p.38

eachを使うと中身を順番に処理することができます。  
ものすごーーーく大事！！！

配列.each do |変数|

繰り返したい処理

end

# 配列の繰り返し処理



教科書  
p.38

```
names = ["五十嵐", "濱崎", "山田"]  
names.each do |name|  
  puts name  
end
```

"五十嵐"	実行結果
"濱崎"	
"山田"	

nameの両脇  
にある記号 |  
はパイプと読み  
ます。  
キーボードの右  
上の方にある  
(たぶん)。

変数nameの中身が1回目は"五十嵐"、2回目は "濱崎",  
3回目は"山田"となり、繰り返しputs文を実行

# Hash



検索

筍 塩麹 ボンゴレ あさり 新玉ねぎ もっと見る...

レシピをさがす

レシピをのせる

クックル

[«hmskpad のレシピ \(13品\)](#)

レシピID: 1188379

## オーソドックスなとんかつ



揚げ物楽しい、豚肉安い、ソースも簡単

 [hmskpad](#)

### 材料 (1人分)

豚ロース	1枚くらい
こしょう	少々
サラダ油	豚ロースが全て浸かるくらい

### ■ 衣

小麦粉 (薄力粉)	100gくらい
パン粉	100gくらい

# こういうページを表現したいときに

# オーソドックスなとんかつ ←title



## description ↑

揚げ物楽しい、豚肉安い、ソースも簡単

hmskpad

## author ↑

1枚くらい

少々

材料 (1人分)

豚ロース

こしょう

サラダ油

豚ロースが全て浸かるくらい

### ■ 衣

小麦粉 (薄力粉)

100gくらい

卵

1個弱

パン粉

100gくらい

### ■ ソース

ケチャップ

大さじ2

ウスターソース

100ccくらい

## ingredients →

データにラベルを付けると扱い易いです

1

2

3

4

# オーソドックスなとんかつ ←title



**description**

揚げ物楽しい、豚肉安い、ソースも簡単

hmskpad

**author**

豚ロース 1枚くらい

こしょう 少々

サラダ油 豚ロースが全て浸かるくらい

■ 衣

小麦粉（薄力粉） 100gくらい

卵 1個弱

パン粉 100gくらい

**ingredients**→

**recipe = {**

**:title => "オーソドックスなとんかつ",**

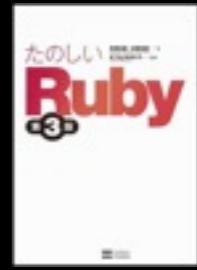
**:author => "hmskpad",**

**:description => "揚げ物楽しい、豚肉安い、ソースも簡単",**

**:ingredients => [省略] }**

Hashを使う  
とこんな感じ  
でまとめられ  
ます

# ハッシュ(Hash)



教科書  
p.40

キーと値の組を持てるオブジェクトの入れもの。  
キーをラベル的に使います。なんでも入ります。

```
recipe = {  
  :title => "♥日向夏のジャム♥",  
  :author => "濱崎" }
```

↑ キー              ↑ 値

{}で囲う キー => 値 区切りは , 空のハッシュは {}

ここでキーに使われている : 始まりのものは何でしょう？



# シンボル

ラベルとして使う文字列的なもの

`:title`

シンボルにするには先頭に`:`を付ける  
ハッシュのキーによく使います

文字列との変換もできます

シンボルへ `"foo".to_sym` → `:foo`

文字列へ `:foo.to_s` → `"foo"`

# ハッシュの別記法(Ruby1.9以降)

```
recipe = {  
  :title => "♥日向夏のジャム♥",  
  :author => "濱崎" }
```

これは、以下のようにも書けます。

```
recipe = {  
  title : "♥日向夏のジャム♥",  
  author : "濱崎" }  
※Ruby1.9以降
```

=> の代わりに : を置き、キーの先頭の : を外します。  
こちらの方が書きやすいので人気です。

# ハッシュから読み込む

```
recipe = { :title => "♥日向夏のジャム♥",
:author => "濱崎" }

p recipe[:title] → "♥日向夏のジャム♥"
p recipe[:author] → "濱崎"
```

ハッシュから値を読むときは、キーを指定します。ハッシュ名[キー]で読み込みます。

# ハッシュへ追加する

ハッシュ名[キー] = 格納したいオブジェクト

同じキーの要素は追加不可(上書き)

ハッシュオブジェクト内でキーは唯一のもの(ユニーク)

```
recipe = { :title => "♥日向夏のジャム♥",
:author => "濱崎" }
```

```
recipe[:url] = "http://cookpad.com/recipe/xxx"
```

# ↑ ここで追加

```
p recipe → { :title => "♥日向夏のジャム♥",
:author => "濱崎",
:url => "http://cookpad.com/recipe/xxx" }
```

# ハッシュの繰り返し処理



教科書  
p.42

Arrayと同じですが、変数を2個となります。

ハッシュ.each do |キーの変数, 値の変数|

繰り返したい処理

end

```
recipe = { :title => "♥日向夏のジャム♥", :author => "濱崎" }
recipe.each do |k, v|
  print k, " - ", v, "\n"
end
```

title - ♥日向夏のジャム♥  
author - 濱崎

実行結果

# ArrayとHashの使い分け

**Array** : 順番が決まっているいれもの

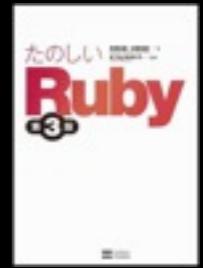
- ・並び順が重要なものの
- ・データを重複させたい場合にも使える

**Hash** : キー(名札)を付けられるいれもの

- ・キーが重複しない場合に利用
- ・順番が保持されなくとも困らないもの

※Ruby1.9 からはHashも順番を保持します。

# nil



教科書  
p.47

「ない」ことを表すオブジェクト  
例えば、ハッシュで存在しないキーを読もう  
とするとこの nil が返ってきます。

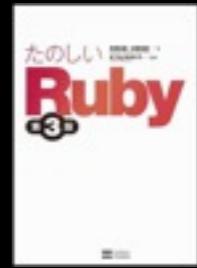
if 文などで条件判断をする場合、

nil は 偽（不成立）になります。

偽（不成立）になるのは false と nil の2つだけです。

それ以外の全ての値は真（成立）になります。

pp



教科書  
p.48

p より見易いデバッグ用メソッド

```
require "pp"
```

```
pp 見たい変数
```

ppには require文 が必要

ハッシュの中身を表示するときに便利です。

# テキストファイル操作

# ファイルから1行ずつ読み込み



教科書  
p.51-55

## サンプルコード

```
filename = "20120301-000000-ja.txt"
file = File.open(filename, "r:UTF-8")
while text = file.gets
  puts text
end
file.close
```

# ファイルを開く

```
filename = "20120301-000000-ja.txt"
file = File.open(filename, "r:UTF-8")
while text = file.gets
  puts text
end
file.close
```

"**r:UTF-8**"  
r は読み込みモード指定。  
readの意。  
**UTF-8**は文字コード指定。

File.open メソッドでファイルを開きます。  
(開く=データを読める状態にする)  
開いたFileオブジェクトをfileへ代入しておきます。

# ファイルから各行読み込み

```
filename = "20120301-000000-ja.txt"
file = File.open(filename, "r:UTF-8")
while text = file.gets
  puts text
end
file.close
```

**file.gets** : 1行読み込み

読み込めたらその行の内容を返す

読み込めなかつたらnilを返す

**while ... end** : 条件が偽(false, nil)になるまで繰り返し

# while文, file.getsメソッド

```
filename = "20120301.txt"
file = File.open(filename)
while text = file.gets
  puts text
end
file.close
```

while 文の条件は  
text = file.gets

になります。



while文は  
教科書 p.93

file.gets  
は1行読み込むメソッド  
読み込めたら真、  
ファイル終端で読み込めない  
と偽(nil)になります。

# ファイルから各行読み込み

```
filename = "20120301-000000-ja.txt"  
file = File.open(filename, "r", encoding: "UTF-8")  
while text = file.gets  
  puts text  
end
```

**while**: 条件成立中は繰り返し  
この場合、条件は  
**text = file.gets**  
になります。

**text** に1行目のデータが入って**end**まで処理

**while** の行へ戻り、

**text** に2行目のデータが入って**end**まで処理

... 全行繰り返し

終端で**file.gets**すると条件不成立になり繰り返し終了

は1行読み込むメソッド

ファイル終端で読み込めない

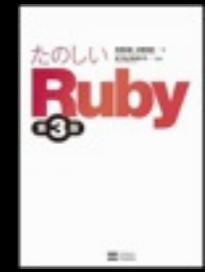
# ファイルを閉じる

```
filename = "20120301-000000-ja.txt"
file = File.open(filename, "r:UTF-8")
while text = file.gets
    puts text
end
file.close
```

**close メソッドでファイルを閉じます。**  
**(閉じる=もう使わないとRubyへ教えてあげる)**  
閉じないと、他のプログラムが同じファイルを開けない場合があります。

# その他のRuby文法

# next文



教科書  
p.102

```
while text = file.gets
```

```
... (A) ...
```

```
next
```

```
... (B) ...
```

```
end
```

繰り返し中にnext文を書くと、それ以降の処理を行わず、  
繰り返し先頭に戻します。

... (A) ... の部分の処理が行われてnextへくると、

... (B) ... の部分の処理は行われず、

繰り返し先頭へ戻ります。

教科書p.102の図6.2が分かりやすいです。

# unless文



教科書  
p.76

unless 条件

処理

end

条件が偽の時に処理を実行

if文と対になる文

if文とは逆に条件が偽の時に処理実行

以下の2つの文は同じ意味

`puts "hello" if x != 3`

`puts "hello" unless x == 3`

# 正規表現



教科書 p.44-46  
p.267-290

文字列がパターンに一致（マッチ）するか調べる道具

ものすごく便利で、強力で、奥が深いです。  
ここではごく基本的な説明だけを行います。

**文字列 =~ /正規表現パターン/**

`=~` → 正規表現マッチを行う演算子

左辺の文字列中に正規表現パターンが含まれるかを判定します。

パターンが英数字や漢字だけからなる場合、

単純に文字列にパターンが含まれるかどうかを判定します。

`"Ruby" =~ /Ruby/ #=> 0`

# マッチした場合はマッチ位置を返します

`"Ruby" =~ /Diamond/ #=> nil`

# マッチしない場合はnilを返します

# 正規表現



教科書 p.44-46  
p.267-290

ほかにも便利な検索の機能があります。

```
text =~ /^ja/
```

正規表現パターン中の <sup>^</sup> は行先頭の意味

/<sup>^</sup>ja/ は 対象文字列が ja から始まればマッチする  
パターンです。

```
"ja title count" =~ /^ja/ #=> 0
```

# 行先頭が ja から始まるのでマッチ

```
"xxxja" =~ /^ja/ #=> nil
```

# jaはあるが、行先頭ではないのでマッチしない

もっと知りたい人はこの辺のページを読んでみてください。

<http://doc.ruby-lang.org/ja/2.0.0/doc/spec=2fregexp.html>

<http://www.namaraii.com/rubytips/>

# CGI.unescape メソッド

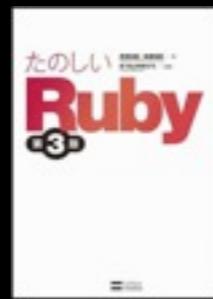
%XX%XXっていう謎の文字列を、読める形式に変換します。

```
require "cgi"  
CGI.unescape("%E3%81%84%E3%82%  
8D%E3%81%AF") #=> "いろは"
```

この変換をURLデコードと呼びます。

ブラウザのURL欄を見ていると時々出てくるので、出てきたら変換の実験してみても面白いと思います。

# 例外処理



教科書

p.153-168

**begin**

例外を発生させる可能性のある処理

**rescue Exception => 变数**

例外が起こった場合の処理

**end**

コード実行中、うまく処理できない場合などに例外を発生させるメソッドがあります。例外が発生した場合、**rescue** 節で例外を捕まえ、その際に実行する特殊処理を書くことができます。

もしも、例外を**rescue** で捕まえない場合は、プログラムはそこでエラー終了します。

# ブロック

do ~ end で書かれる処理のかたまり

桃色の部分がブロック

```
array.sort_by do |i|
```

```
i
```

```
end
```

以前出てきた each メソッドについてたのも実はブロックです。

```
array.each do |i|
```

```
  puts i
```

```
end
```

# メソッド+ブロック

`sort_by` や `each` はブロックを添えて呼び出します。  
ブロックを添えて呼び出すArrayのメソッドは  
大抵Arrayの全要素について繰り返し実行します。

```
array.sort_by do |i|
```

```
i
```

```
end
```

`sort_by` : 全要素をブロックの評価結果で並び替え

```
array.each do |i|
```

```
puts i
```

```
end
```

`each` : 全要素についてブロックを実行

# 破壊的メソッド

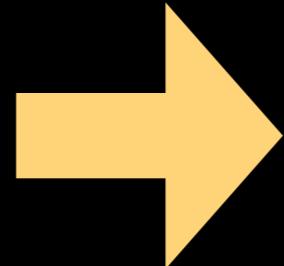
`upcase!` のようにオブジェクトの内容を  
変更するものを「破壊的メソッド」と言います

`"abc".upcase!`

実行前

abc

upcase!



実行後

ABC

オブジェクト  
が変身する

オブジェクト

オブジェクト

メソッド名末尾に `!` がついたら破壊的メソッド  
(ただし、破壊的メソッドでも `!` が付かないものもある)

# リファレンス マニュアル

Rubyのリファレンスマニュアルでメソッドを探せます。  
例えば以下のページでリファレンスマニュアルを検索できます。

<http://miyamae.github.com/rubydoc-ja/2.0.0/>  
or 「サクサク Ruby リファレンス」でgoogle検索



このサイトはRubyリファレンスマニュアル通称「るりま」を検索するためのページです。リファレンス検索するページは他にも「るりまサーチ」などがあります。

# リファレンスマニュアル検索



1. 画面左上の検索窓に「Array」と入力します。
2. 表示候補の最初の行「Array」をクリックします。

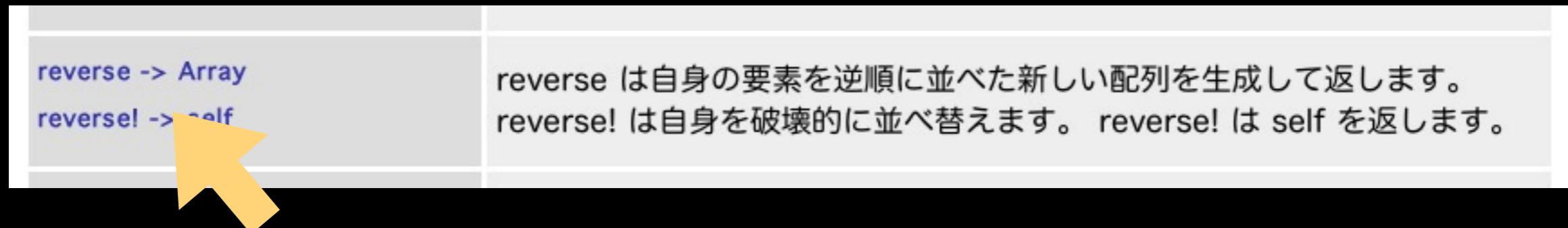
3. 右側にArrayのメソッド一覧などの説明が表示されます。

A screenshot of the detailed documentation for the 'Array' class in the Ruby 1.9.3 Reference Manual. The title 'class Array' is prominently displayed. Below it, the inheritance list is shown: 'クラスの継承リスト: Array < Enumerable < Object < Kernel < BasicObject'. A section titled '要約' (Summary) contains the text: '配列クラスです。配列は任意の Ruby オブジェクトを要素として持つことができます。一般的には配列は配列式を使って'.

下の方へ探していくと・・・

# リファレンスマニュアル検索

メソッド名と、そのメソッドの説明が並んでます。



4. 使えそうなメソッドのあたりをつけたらメソッド名をクリック

Ruby 1.9.3 リファレンスマニュアル > ライブラリー一覧 > 組み込みライブラリ > Arrayクラス > reverse

## instance method Array#reverse

reverse -> Array  
reverse! -> self

reverse は自身の要素を逆順に並べた新しい配列を生成して返します。 reverse! は自身を破壊的に並べ替えます。 reverse! は self を返します。

```
a = ["a", 2, true]
p a.reverse      #=> [true, 2, "a"]
p a              #=> ["a", 2, true] (変化なし)

a = ["a", 2, true]
p a.reverse!
p a              #=> [true, 2, "a"]
```

メソッド名

説明

サンプルコード

reverseを使えばできそうなので試してみます。

# プロも使う検索方法

Rubyのオブジェクトに直接聞く方法もあります。

```
$ irb
```

```
[] .methods
```

```
#=> ["zip", "pop", "find_index", "rassoc", "enum_slice", "map!",  
"minmax", "methods", "send", "shuffle!", "replace", ... ]
```

`methods` メソッドで使えるメソッド一覧が表示されます。英単語でなんとなく意味が分かるのでこれで見つけるときも多いです。

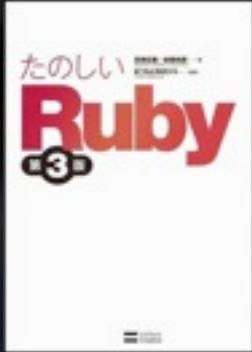
```
require 'pp'  
pp [].methods.sort
```

`pp` と `sort` を使うと見易いです。

★TODO

演習

- Array each



教科書  
p.120~152



# class

クラスとインスタンス

ローカル変数と  
インスタンス変数

`attr_accessor`

`initialize`メソッド

クラスメソッドと  
インスタンスメソッド

# クラスとインスタンス

**Array  
String**

**[1,2,3]  
"ちはやふる"**

クラス  
オブジェクトの  
種類を表すもの。  
型。 種族。 設計図。

インスタンス  
クラスから  
作られたもの。

# クラスとインスタンス



<http://www.flickr.com/photos/tonomura/2391806827/>

クラス  
設計図



[http://www.flickr.com/photos/\\_yoggy0/4406076358/](http://www.flickr.com/photos/_yoggy0/4406076358/)

インスタンス  
classから作られたもの

# インスタンスをつくる：newメソッド

**new**：クラスからインスタンスオブジェクトを作るメソッド  
(=たい焼きの型を使って、たい焼きを焼く)



たい焼きの型  
Taiyakiクラス

Taiyaki.new



Taiyaki.new



たい焼きの型(Taiyakiクラス)  
から、たい焼きインスタンスが  
1つできあがります。

newするとその回数だけ  
たい焼きインスタンスが  
できます。

**Array.new #=> []**

**String.new #=> ""**

実際のRubyのコードでは  
newメソッドを使うと、  
そのクラスの空っぽのオブジェクトができる

一方で、今まで書いていたように、  
インスタンスオブジェクトを  
直接作ることも可能です。

[1,2,3]

"ちはやふる"

# classのつくりかた

class(たい焼きの型)を自分で作る場合の書き方です。

```
class クラス名  
  クラスの定義  
end
```

クラス名は必ず大文字から始める。  
例：Array, String, Recipe, Book

例：

```
class Recipe  
  #ここにクラスの内容を書く  
end
```

# つかったclassを newしてみる

自作のclass(たい焼きの型)からnewしてインスタンス  
(たい焼き)をつくるてみます。

```
class Recipe
```

```
end
```

```
recipe = Recipe.new
```

小文字のrecipe は変数で、 インスタンスを代入しています。  
大文字始まりのRecipeはクラス名です。

# initialize メソッド

`initialize` という特別な名前のメソッドを作ると、  
インスタンスを作る際(`new`メソッドが呼ばれた際)に  
自動で実行するメソッドを作ることができます。

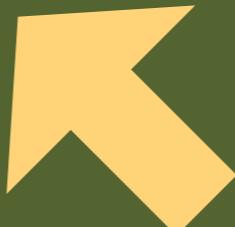
```
class Recipe
  attr_accessor :title, :author
  def initialize
    puts "initialized!"
  end
end
```

```
recipe = Recipe.new #=> "initialized!"
```

# initialize メソッド

newメソッドに引数を渡すと、initializeメソッドの引数として受け取ることができます。

```
class Recipe
  attr_accessor :title, :author
  def initialize(title, author)
    @title = title
    @author = author
  end
end
```



newメソッド呼び出し時に渡された引数が initializeメソッドに渡される。

```
recipe = Recipe.new("cheese cake", "igarashi")
p recipe.title #=> "cheese cake"
p recipe.author #=> "igarashi"
```

# 変数のスコープ（有効範囲）

変数には有効範囲があります。

```
class Recipe
  def title=(t)
    recipe_title = t  ※1
  end
  def title
    recipe_title
  end
end
```

recipe\_title のスコープ

ここで上記の recipe\_title 变数にアクセスできない

変数には有効範囲、生存期間があります。メソッド内で作成した変数は、そのメソッドの中だけが有効範囲です。

このような变数をローカル変数と呼びます。

※1 の行の変数 **recipe\_title** は、そのメソッドの中だけがスコープ（有効範囲）です。

# インスタンス変数

インスタンスオブジェクトが生存している間ずっと使える変数が  
インスタンス変数です。変数名の頭に @ をつけます。

```
class Recipe
  def title=(t)
    @recipe_title = t
  end
  def title
    @recipe_title
  end
end
```

```
recipe = Recipe.new
recipe.title = "cheese cake"
p recipe.title #=> "cheese cake"
```

@recipe\_title  
のスコープ

ここでも@recipe\_title変数  
にアクセスできる

# インスタンス変数の性質 1

```
class Recipe
  def title=(t)
    @recipe_title = t
  end
  def title
    @recipe_title
  end
end

recipe1 = Recipe.new
recipe1.title = "cheese cake"
recipe2 = Recipe.new
recipe2.title = "macaroon"

p recipe1.title #=> "cheese cake"
p recipe2.title #=> "macaroon"
```

インスタンスオブジェクト（たい焼き）ごとに  
インスタンス変数を  
別々に持っています。

# インスタンス変数の性質 2

※このコードには誤りがあります

```
class Recipe
  def title=(t)
    @recipe_title = t
  end
  def title
    @recipe_title
  end
end
```

```
recipe = Recipe.new
recipe.title = "cheese cake"
p recipe.recipe_title ✗
```

```
undefined method `recipe_title' for #<Recipe:
0x108650280 @recipe_title="cheese cake">
(NoMethodError)
```

実行結果

インスタンス変数は  
オブジェクトの外か  
ら直接アクセスでき  
ません。

アクセスする時は、  
そのオブジェクトの  
メソッドを通じてア  
クセスします。

# attr\_accessor

インスタンス変数を同名のメソッドで読み書きするコードはよく使うので、便利な書き方 attr\_accessor が用意されています。

```
class Recipe
  def title=(t)
    @title = t
  end
  def title
    @title
  end
end

recipe = Recipe.new
recipe.title = "cheese cake"
p recipe.title
#=> "cheese cake"
```

書き方：  
attr\_accessor インスタンス変数名のシンボル

```
class Recipe
  attr_accessor :title
end

recipe = Recipe.new
recipe.title = "cheese cake"
p recipe.title
#=> "cheese cake"
```

同じ動作

とても短くなりました。

# 命名規則

クラス名は大文字始まり、変数名は小文字のみ

クラス名の例： **Array, String, Recipe, Book**

2単語以上を組み合わせるときは、单語境界文字を大文字にします

例： **RecipeSite**

ちなみに、このタイプの規則を **Camel Case** といいます。（らくだ）

変数名の例： **array, string, recipe, book**

2単語以上を組み合わせるときは、单語を **\_** でつなぎます

例： **recipe\_site**

ちなみに、このタイプの規則を **Snake Case** といいます。（へび）

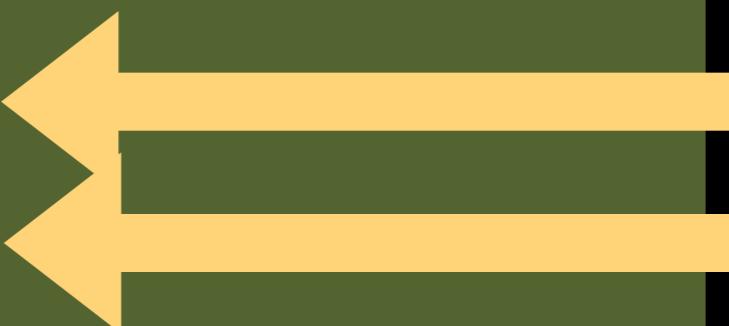
# インスタンスメソッド

ここまで見てきたような、クラスの中で普通に定義したメソッドをインスタンスメソッドといいます。インスタンスに対して呼ぶことができるメソッドです。

```
class Recipe
  attr_accessor :title, :author

  def title_and_author
    @title + " - " + @author
  end
end
```

```
recipe = Recipe.new
recipe.title = "cheese cake"
recipe.author = "igarashi"
p recipe.title_and_author #=> "cheese cake - igarashi"
```



`attr_accessor` で作られるメソッドもインスタンスメソッドです。

`title_and_author` は  
インスタンスメソッド

インスタンスメソッドは  
インスタンス(たい焼き)に対して呼ぶことができます。

# クラスメソッド

もう1種類、クラスメソッドというのも存在します。クラスメソッドはクラス（たい焼きの型）に対して呼び出します。  
self.メソッド名で定義します。

```
class クラス名
  def self.メソッド名
  end
end
```

```
class Recipe
  attr_accessor :title, :author
  def self.published_by
    "COOKPAD"
  end
end
```

```
p Recipe.published_by #=> "COOKPAD"
```

クラスに対して呼ぶ。  
newしないで呼ぶ。

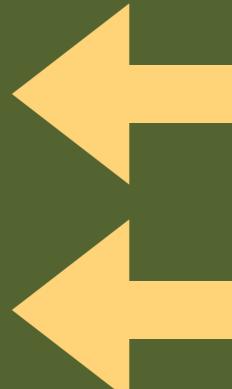
# インスタンスメソッド、クラスメソッド

インスタンスメソッドはインスタンス（たい焼き）に対して呼び、  
クラスメソッドはクラス（たい焼きの型）に対して呼びます。

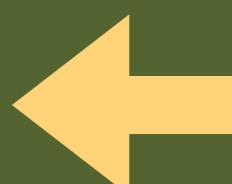
```
class Recipe
  def title
    @title
  end
  def title=(t)
    @title = t
  end
  def self.published_by
    "COOKPAD"
  end
end
```

```
p Recipe.published_by #=> "COOKPAD"
recipe = Recipe.new
recipe.title = "cheese cake"
```

```
recipe = Recipe.new
p recipe.published_by #=> "COOKPAD" X
Recipe.title = "cheese cake" X
```



インスタンスメソッド



インスタンスメソッド

self. がついているのでクラスメソッド

クラスメソッド

(この文ではpublished\_by)  
はクラスに対して呼ぶ。

インスタンスメソッド

(この文ではtitle)  
はインスタンスに対して呼ぶ。

逆は呼べない

# インスタンス変数にアクセスできる のはインスタンスマソッドだけ

インスタンス変数にアクセスできるのはインスタンスマソッドだけです。  
クラスメソッドの中ではインスタンス変数にアクセスできません。

```
class Recipe
  attr_accessor :title, :author
```

```
def title_and_author
  @title + " - " + @author
end
```

```
def self.published_by
  @title + " - " + @author X
end
end
```

インスタンスマソッドなので  
インスタンス変数にアクセス可

クラスメソッドなので  
インスタンス変数にアクセス不可

たい焼きの型に、「あんこ多い？」って聞い  
ても答えられないのと同じです。たぶん。  
(あんこはインスタンス(たい焼き)のもの)

# オブジェクト指向



クックパッド

食べ過ぎリセット

## お知らせオブジェクト

1番人気のレシピが今だけ無料で見放題▼

ログイン



料理名・食材名



目的・用途

レシピ検索

父の日 夏 夏バテ

注目ワード



レシピをのせる

## \*【簡単】チーズケーキオブジェクト

レシピを保存

単純作業ばかりの簡単レアチーズケーキ！

母からのお墨付きをもらいました♪

※材料、倍量に変更しました。

※ゼラチン多め

kanan店長

材料 (18cm底の抜ける丸型)

この部分の表示が  
意図通り出ない・・・

☆水きりヨーグルト	200g
☆砂糖	90g
☆レモン汁(ポッカレモン)	大2
☆蜂蜜	10g
☆牛乳	200ml
★ゼラチン	16g

このデータを管理す  
る役割はRecipeクラス  
が受け持ってるから、そ  
こに何かバグがあるのか  
も？各オブジェクトに役割と責任を持たせる設計指針を  
「オブジェクト指向」といいます。かして、袋の中に入  
れ揉んでなじませ  
る。・ゼラチンをふやか  
す。

# レシピオブジェクトの役割と責任

## レシピオブジェクト

```
class Recipe
```

```
def title          公開ゾーン  
  @title  
end  
def title=(t)  
  @title = t  
  @updated_at = Time.now  
end
```

```
...               非公開ゾーン
```

```
  @title = "cheese cake"  
  @author = "igarashi"
```

```
...  
end
```

データへのアクセスは所定  
のメソッドを通じてお願  
いします。

→ 公開ゾーンと  
非公開ゾーンを分ける

データを外から勝手に書き換えるとレシピオブジェクトは責任を果たせない  
= データ(変数)は外部からはアクセスできなくすべき(非公開ゾーンに置くべき)  
= Rubyのclassは最初からそうなるように設計されています

# public, private

## メソッドの公開・非公開を制御できます。

```
class AccessTest
```

```
public
```

```
#これ以降に書いたメソッドはpublic
```

```
def show
```

```
  puts "public method"
```

```
end
```

```
#ここに書いたメソッドもpublic
```

```
private
```

```
#これ以降に書いたメソッドはprivate
```

```
def secret
```

```
  puts "private method"
```

```
end
```

```
end
```

```
obj = AccessTest.new
```

```
obj.show #=> "public method"
```

```
obj.secret
```

## public

以降のメソッドを公開メソッドにします。(または、何も書かないと publicになります)

## private

以降のメソッドを非公開メソッドにします。非公開メソッドは、オブジェクト内部からは呼び出せますが、外部からは呼び出せません。

※protectedというのもあるのですが、滅多に使わないので省略

# オブジェクトの内部と外部

```
class AccessTest
  def show
    secret #ここは内部
  end

  private
  def secret
    puts "private method"
  end
end
```

```
obj = AccessTest.new
obj.secret #ここは外部 X
```

内部

内部

そのオブジェクトクラスのメソッド  
の中。そのオブジェクトクラスの  
class～endの間。

外部

それ以外

または、

secret のようにメソッド名だけで呼べ  
るところが内部、

obj.secret のように  
オブジェクト.メソッド名で  
呼ぶところが外部です。

継承



# 継承

既に定義されているクラスを拡張して新しいクラスを作ることを継承といいます。

(既にあるたい焼きの型を利用して、少し違う新しいたい焼きの型をつくるようなものです。)

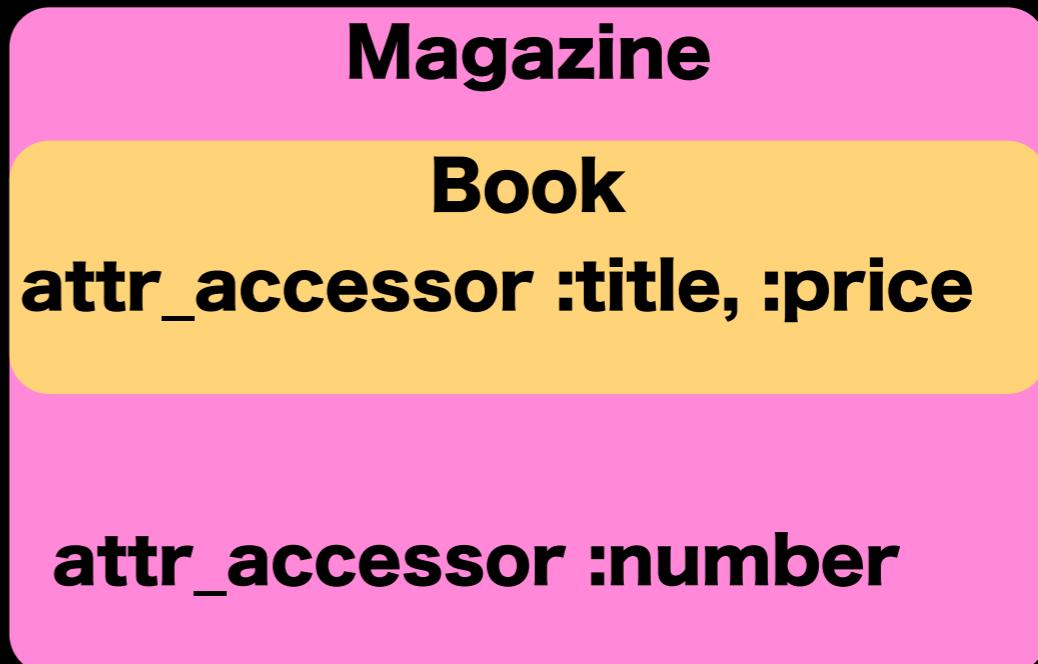
# 継承

## 継承を使った書き方

```
class Book  
  attr_accessor :title, :price  
end
```

```
class Magazine < Book  
  attr_accessor :number  
end
```

## 図解



Bookクラスを継承したMagazineクラスは、Bookクラスの持ち物を受け継ぎます。この例の場合は、Bookクラスの attr\_accessor :title, :price をMagazineクラスでも使えます。加えて、Magazineクラスにある attr\_accessor :number も利用できます。

# 継承

## 継承する場合の書式

```
class クラス名 < スーパークラス名  
  クラスの定義  
end
```

スーパークラスとは、継承元の  
クラス(親クラス)です。

継承したクラスは、親クラスの全てのメソッド、インスタンス変数などを受け継ぎます。

```
class Book  
  attr_accessor :title, :price  
end
```

```
class Magazine < Book  
  attr_accessor :number  
end
```

例えばBookクラスを継承した  
Magazineクラスは、titleとpriceを  
受け継いでいます。例えば、↓のような  
コードを書くことができます。

```
magazine = Magazine.new  
magazine.title = "CanCam"  
p magazine.title #=> "CanCam"
```

# Module

メソッドを共同利用する仕組み

# Module

複数のクラスで同じメソッドを利用したいときにmoduleを使うと重複なく書けるので便利です。

```
module Greeting
  def hello
    puts "Hello!"
  end
end
```

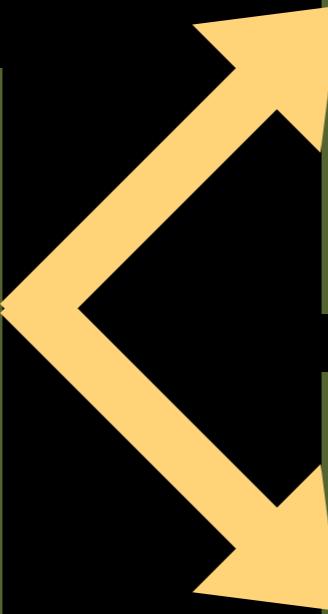
もしもhelloメソッドで表示する"Hello!"を"こんにちは"に変えたい場合はこのモジュールだけ変更すれば良い

```
class Alice
  include Greeting
end

alice = Alice.new
alice.hello #=> "Hello!"
```

```
class Bob
  include Greeting
end
```

```
bob = Bob.new
bob.hello #=> "Hello!"
```



# Moduleの文法

```
module モジュール名  
#メソッド定義  
end
```

```
class Sample  
include モジュール名  
end
```

module 定義

include(読み込み)

# 目次

Gem

Hugs from  
my work  
with love,

Hudrik

# Gem

Rubyのライブラリ管理システム

# Gem

ライブラリ：  
他のプログラムから読み込んで利用するためのプログラム

たくさんの便利なライブラリがネット上に公開されています。それらをインストールするなど、管理してくれる仕組みがGemです。

# Gemの例

- Excelファイルを読み書きする
  - twitterなど有名なWebサービスへのアクセスを簡単にしてくれる
  - 画像をリサイズしたりエフェクトをかける
- などなど数万種類のライブラリが公開されています

**710,437,847  
downloads**

of 40,705 gems cut since July 2009

Welcome to your community RubyGem h

Find your gems easier, publish them faster, and have

do

## LEARN

### Install RubyGems 1.8.24

Ruby's premier packaging system

### Browse the Guides

In depth explanations, tutorials, and references

### Gem Specification

Your gem's interface to the world

## SHARE

`gem update --system`

Update to the latest RubyGems version

`gem build foo.gemspec`

Build your gem

`gem push foo-1.0.0.gem`

Deploy your gem instantly

gemを紹介、管理しているサイト  
RubyGems.org is the Ruby community's gem hosting service. Instantly publish your gems and install them. Use the API to interact and find out more information about available gems. Become a contributor and enhance the site with your own changes.

<http://rubygems.org>

# Gemの使い方

## gemパッケージのインストール

1. shell からgemコマンドを使ってインストール

```
$ gem install gem名
```

2. コードで require "gem名"

```
require "gem名"  
# ここでgem を使うコードを書きます。
```

# Gemの使い方

インストール済のgemパッケージ一覧を表示

shell からgem list コマンドで一覧表示

```
$ gem list
```

# spreadsheet

Excelファイルを操作するgem

```
$ gem install spreadsheet
```

# Twitter

A Ruby interface to the Twitter API.

This project is maintained by [sferik](#)

 Download ZIP

 Download TAR

 View On GitHub

## The Twitter Ruby Gem

gem version 4.8.1

build passing

dependencies up-to-date



 DONATE [pledgie.com](#)  
\$300.00 Raised!

A Ruby interface to the Twitter API.

## Installation

```
gem install twitter
```

**\$ gem install twitter**

To ensure the code you're installing has been signed, download my GPG public key and verify the signature. To do this, you need to add my public key as a trusted certificate (you only need to do this once):

# Sinatra

Put this in  
your pipe

[README](#)  
[DOCUMENTATION](#)  
[CONTRIBUTE](#)  
[CODE](#)  
[CREW](#)  
[ABOUT](#)

```
require 'sinatra'  
get '/hi' do  
  "Hello World!"  
end
```

Sinatra gem Web アプリフレームワーク

\$ gem install sinatra

プログラムで  
できること



**NATE SILVER**  
ネイト・シルバー：人種は投票に影響を及ぼすか？

Michael Cosentino  
@cosentino

Follow

For the Nate-haters, here's the 538 prediction and actual results side by side pic.twitter.com/jbny4pRX

538 prediction      actual



7 Nov 12

Reply Retweet Favorite

予測結果比較

ネイト・シルバー (Nate Silver) 氏 (34歳) はニューヨーク在住の統計専門家であり、ニューヨークタイムズの人気ブログ「[FiveThirtyEight ~NateSilver's Political Calculus \(538 ~ネイト・シルバーの政治的微分積分\)](#)」を執筆するプロガーです。11月6日以降、アメリカの主要メディアで彼の名前を見ない日がないほどの人気が続いている。なぜでしょうか？

『大統領選でニューヨークタイムズのネイト・シルバーの数理モデル予測が全50州で的中—政治専門家はもはや不要?』 TechCrunch japan (11/8/2012)



Email Address

Subscribe

# Want to take on Wall Street? Quantopian's algorithmic trading platform now accepts outside data sets



BY MICHAEL CARNEY  
ON APRIL 2, 2013



Quantopian has built an online playground for quantitative traders, making it easier than ever for millions of everyday

TICKER

LATEST

“The NSA is wiring the world; they want to own internet.”

— Thomas Drake on PRISM and the NSA's desire to gather data

about 16 hours ago



“The future city is not going to be a congestion-free environment. That same prediction was made that cars would free cities from the congestion of horses on the street. You have to build the sewer system to accommodate the breaks during the Super Bowl; it won't be as pretty as we're envisioning.”

— Center for Internet and Society fellow Bryant Walker Smith discussing driverless cars with the New York Times

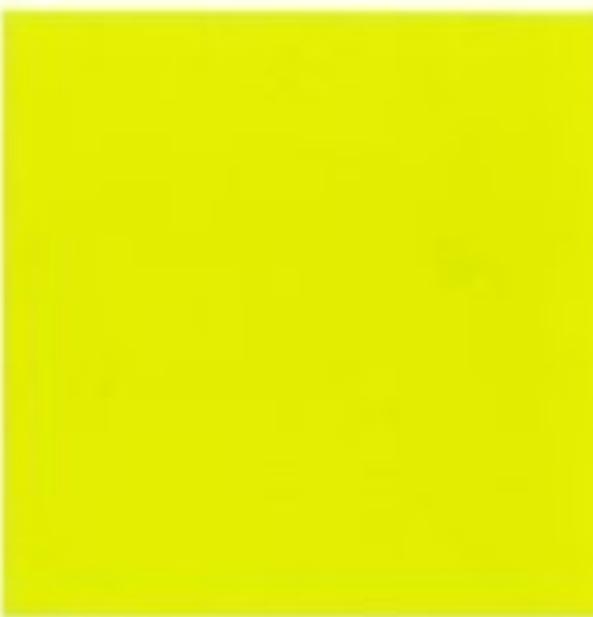
about 16 hours ago



130

# 科学する麻雀

## とつけき東北



「数理の力」が  
あなたの麻雀を変える！

裏スジは危険ではない／同じ打ちは無意味だ  
ベタオリには法則がある／「読み」など必要ない

この講義を受けると  
これができるようになる(予定)

Rubyの基礎が身につき、簡単なプログラムを書ける

wikipediaアクセス数解析実習

Ruby on Rails の基礎が身につき、簡単なアプリを作れる

Rubyに関する不明点があったときに調べられる

この講義を受けたら  
これができるようになった(希望)

Rubyの基礎が身につき、簡単なプログラムを書ける  
wikipediaアクセス数解析実習

Ruby on Rails の基礎が身につき、簡単なアプリを作れる  
Rubyに関する不明点があったときに調べられる

Excel のファイルを見たら  
Rubyを思い出してください

spreadsheet gem などで  
解析できなか、  
自動化できなか考えてください

twitterなど  
web上のデータを収集したい  
と思ったときは、  
gemを探すと良いです。

「Rubyは開発者を  
幸せにする言語。

もっと多くの開発者を  
Rubyで幸せにしたい。」

まつもとゆきひろ

みんなの工具箱の中に  
Rubyを詰めて  
そして  
いつか役だって  
もらえたなら幸いです



# 參考資料

# 書式

Rubyコード

`puts "abc"`

実行結果

"abc"

実行結果

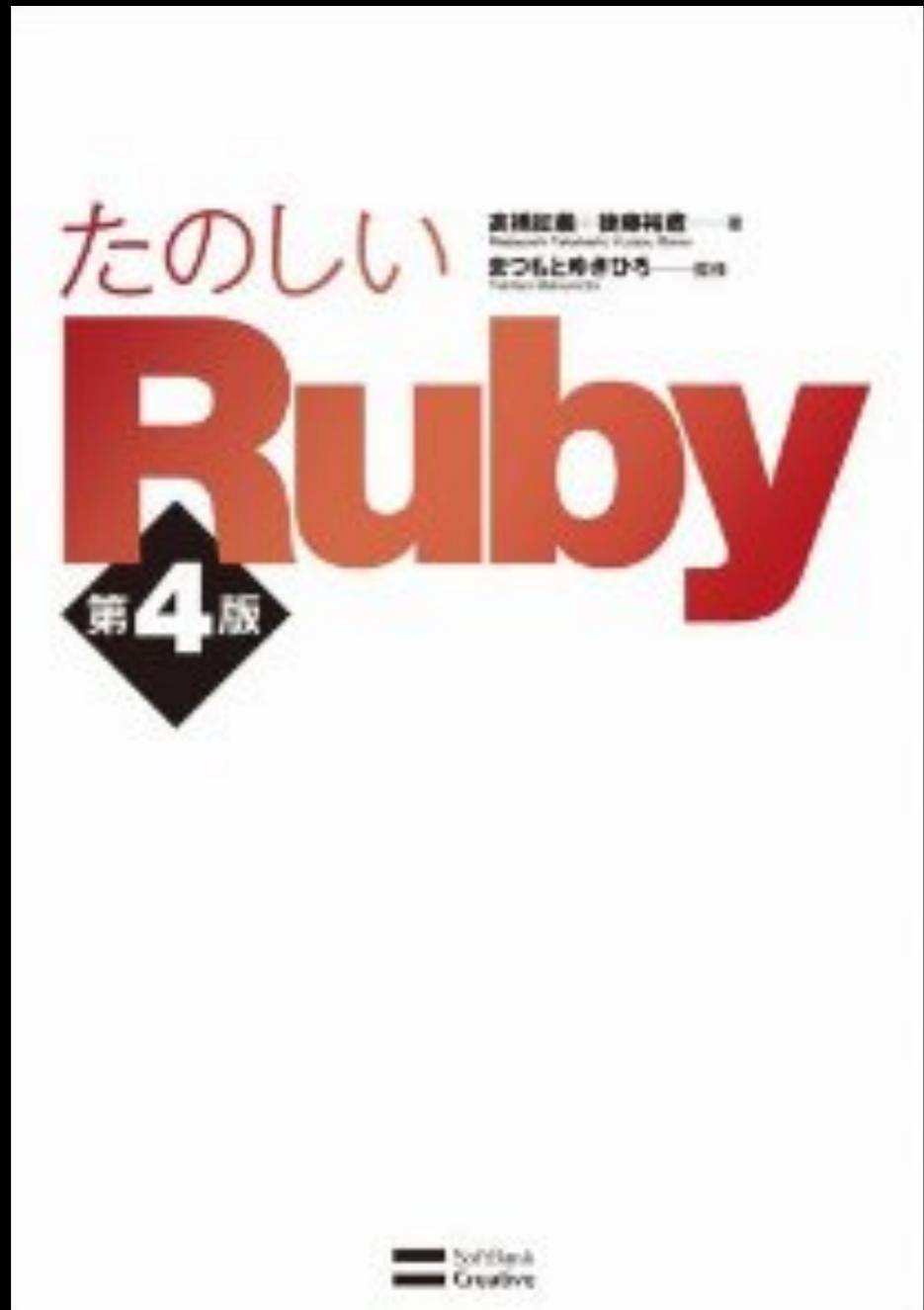
文中では `#=>` で書きます

`p 1+2 #=> 3`

shellコマンド

`$ ls`

# 教科書：たのしいRuby



<http://www.amazon.co.jp/dp/4797372273/>



お買い求めは  
大学生協または  
ジュンク堂池袋店で

# 課題チェック用の付箋の書き方

上の方に学籍番号と名前を書いてください

学籍番号

名前

※この辺に講師陣がクリアした課題番号を書いていきます。

# 講義資料置き場

過去の資料がDLできます。

<https://github.com/igaiga/hitotsubashi-ruby-2013>

# 雑談・質問用facebookグループ

<https://www.facebook.com/groups/hitotsubashi.rb>

- 加入/非加入は自由です
- 加入/非加入は成績に関係しません
- 参加者一覧は公開されます
- 参加者はスタッフ(講師・TA)と昨年、今年の受講者です
- 書き込みは参加者のみ見えます
- 希望者はアクセスして参加申請してください
- 雑談、質問、議論など何でも気にせずどうぞ~
- 質問に答えられる人は答えてあげてください
- 講師陣もお答えします
- 入ったら軽く自己紹介おねがいします