

# Ruby 講義

## 第15回 夏学期総復習

Kuniaki IGARASHI/igaiga

2012.7.19 at 一橋大学

社会科学における情報技術とコンテンツ作成III  
(ニフティ株式会社寄附講義)

○ 剰余金の配当に関するお知らせ

○ ニフティ、「@nifty EMOBILE LTE 定額にねんプラン」の提供を開始

○ 「@nifty温泉」で「母の日 全国一斉！100のありがとう風呂」特設サイト公開

○ 「スマブレ！」のサービス停止について

○ ニフティとサンリオウェーブ、iOS向けアプリ「Hello Kitty Worl...」

○ 平成24年3月期 決算短信

○ 特別損失の計上に関するお知らせ

○ 「シユフモ」登録会員数150万人を突破、「2012年主婦の全国節電調査（冬季...」

ニフティとなら、きっとかなう。  
With Us, You Can.

# ニフティ株式会社

アット・ニフティ  
楽しいサービスがいっぱい



アクセスマップ  
大森から西新宿へ移転いたしました



@nifty Web募金  
東日本大震災復興支援  
募金受付中



- 2012年4月25日 IR [特別損失の計上に関するお知らせ](#)
- 2012年4月25日 IR [剰余金の配当に関するお知らせ](#)
- 2012年4月19日 IR [@nifty EMOBILE LTE 定額にねんプラン」の提供を開始](#)
- 2012年4月19日 IR [ニフティとサンリオウェーブ、iOS向けアプリ『Hello Kitty World』を台湾で提供開始](#)
- 2012年4月10日 お知らせ [「@nifty温泉」で「母の日 全国一斉！100のありがとう風呂」特設サイト公開](#)

# 提供

講師

# 五十嵐邦明

株式会社万葉

エンジニア



GARASHI

.9.25 at 高専カンファ

いがいが  
⑤

Teaching Assistant 演習 健二  
クックパッド株式会社 エンジニア



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。

總復習

shell編

# 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 ..
```

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**



總復習

Ruby編

# 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

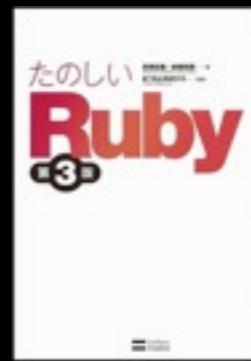
[1,2,3] : Arrayオブジェクト

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

**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

# 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**

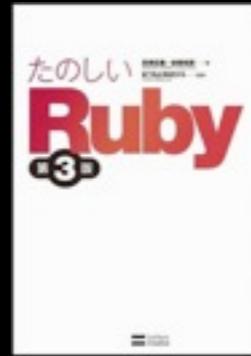
**/ : 割り算**

**% : 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。**

# 変数



教科書  
p.21

オブジェクトへのラベル・荷札

変数 = オブジェクト

変数にオブジェクトを代入する

**name = "igarashi"**

(変数nameに"igarashi"オブジェクトを代入)

**puts name**

**=> "igarashi"**

# 変数はオブジェクトを指す荷札

```
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？

変数

オブジェクト

`a = "abc"`

aはオブジェクト  
"abc"を示す変数

a

abc

`b = a`

bもaと同じ  
"abc"を示す変数

a

b

abc

`a.upcase!`

aの指すオブジェクトを大文字にする

a

b

ABC

`puts a`  
`puts b`

`a => "ABC"`

`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
```

さっちは→  
こうでした

```
a = "abc"
```

```
b = a
```

```
a.upcase!
```

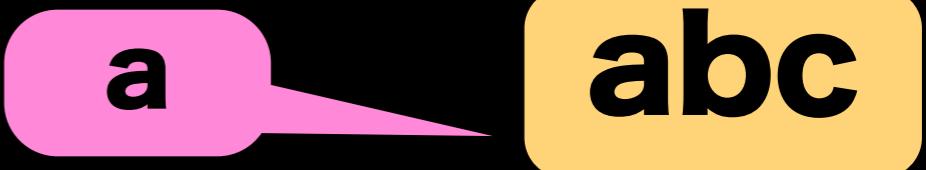
```
puts a
```

```
puts b
```

変数 オブジェクト

`a = "abc"`

aはオブジェクト  
"abc"を示す変数



`b = "abc"`

bは別のオブジェクト  
"abc"を指す



`a.upcase!`

aの指すオブジェクトを大文字にする



`puts a`

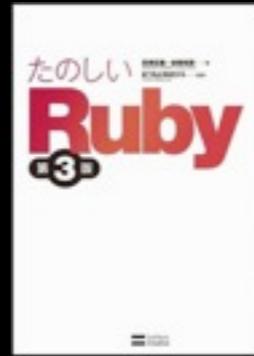
**a => "ABC"**

`puts b`

**b => "abc"**

# 条件判断

# 条件判断 if文



教科書  
p.25-27

`if` 条件

条件が成り立ったときに実行したい処理

`end`

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

- ▶ == → 等しいかどうか判定
- ▶ != → 等しくないかどうか判定(==の逆)

```
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、異なれば false を返します。  
比較は == です。= だと代入になるので注意。

# 条件判断 if - else - end

条件が不成立の時に実行したい処理と一緒に書くこともできます。

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

# 条件判断 if - elseif - else - end

if文の条件に加えてさらに条件を書くこともできます。

**if 条件1**

条件1が成立した時に実行したい処理

**elseif 条件2**

条件2が成立の時に実行したい処理

**else**

条件1も2も成立しなかった時に実行したい処理

**end**

elseif は2つ以上つなげることもできます。

# 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

# unless文

条件不成立を判定する文。if文の逆です。

**unless 条件**

条件不成立時に実行したい処理

**end**

以下の2つは同じ意味です。

**unless x == 1**

**puts "x は 1 ではない"**

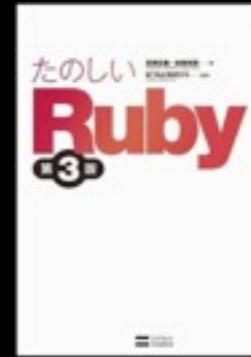
**end**

**if x != 1**

**puts "x は 1 ではない"**

**end**

# 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

その他  
大事なこと

# 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。

# 文字列中に変数を埋め込む

"#{変数名}" と書くと、  
文字列に変数の内容を埋め込むことができます。

```
name = "igarashi"  
puts "Author #{name}" #=> Author igarashi  
変数nameの中身が表示される
```

以下の2つのコードのtextは同じ文字列になります。

```
name = "igarashi"  
text = "Author #{name}"  ←→ text = "Author igarashi"  
                         同じ動作
```

# 2種類の文字列表現

Rubyには2種類の文字列の表現方法があります。

ダブルクオートとシングルクオートです。

前のページで説明した変数の中身を表示する場合はダブルクオートを使う必要があります。

ダブルクオート："#{変数名}" と書くと変数の中身を表示

```
name = "igarashi"
```

```
puts "Author #{name}" #=> Author igarashi
```

変数nameの中身が表示される

シングルクオート：'#{変数名}' と書くとそのまま表示

```
name = "igarashi"
```

```
puts 'Author #{name}' #=> Author #{name}
```

変数nameの中身が表示されず、そのまま出力される

# エラーメッセージ

正しくないプログラムを実行しようとすると、  
エラーメッセージが表示されます。

**helloruby.rb**

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

```
prin("Hi.") ←正しくない
```

```
$ ruby helloruby.rb
```

```
helloruby.rb:2:in `<main>': undefined  
method `prin' for main:Object  
(NoMethodError)
```

# 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

**helloruby.rb:2:in `<main>': undefined  
method `prin' for main:Object  
(NoMethodError)**

日本語訳

**helloruby.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  
**そんなメソッドなかとよエラー** が起きたよ

# 文字コードと マジックコメント



教科書  
p.16

日本語を含むコードを書くとき

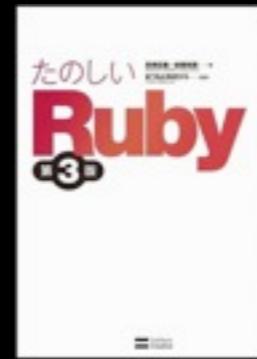
- ・ファイルをUTF-8で保存する。
- ・1行目にマジックコメントを書く  
(そのファイルの文字コードをRubyへ教えるため)  
例) # encoding: utf-8

# encoding: utf-8 ← マジックコメント

print("ちはやふるかみよもきかずたつたがわ\n")

print("からくれないにみづくくるとは\n")

# 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。

# Array

# 配列(Array)



教科書  
p.33~

## ほかのオブジェクトの入れもの

作り方の例：

```
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。

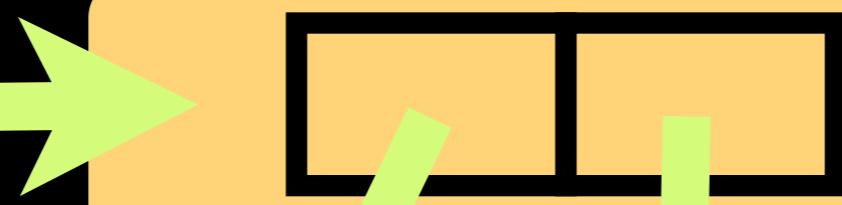
(概念図は教科書p.35 図2.2 を参照。)

# 配列(Array)概念図

```
names = ["五十嵐", "濱崎"]
```

変数

- names



五十嵐

濱崎

Array  
オブジェクト

String  
オブジェクト

# 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"

最初の要素は0番です。1始まりではないので注意です。

また、末尾からの番号でも読み込めます。(-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")
```

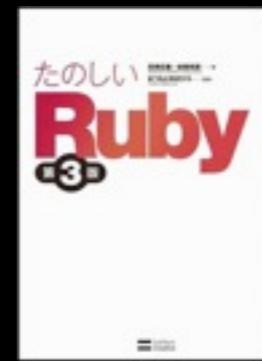
```
p names → ["五十嵐", "濱崎", "山田"]
```

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~

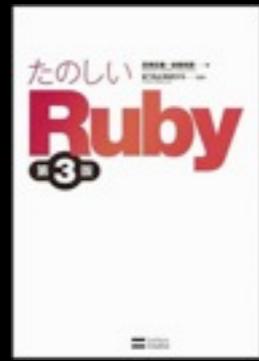
eachで中身を順番に処理する  
ものすごーーく大事！！！

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"
```

```
  "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)。

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

# ブロック

do ~ end で書かれる処理のかたまり

```
array.each do |i|
  puts i
end
```

```
array.sort_by do |i|
  -i
end
```

# 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



## ingredients→

揚げ物楽しい、豚肉安い、ソースも簡単

## description ↑

hmskpad

## author ↑

1枚くらい

材料 (1人分)

豚ロース

少々

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

データにラベルを付けると扱い易いです

1

2

3

4

包丁の峰で合体左上

# 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

豚ロース

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 = {**

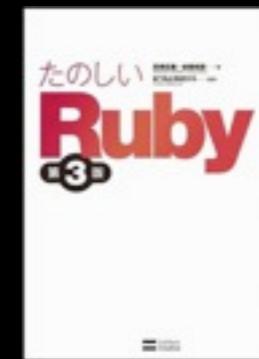
**:title => "オーソドックスなとんかつ",**

**: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



教科書  
p.40~

# ハッシュ(Hash)

キーと値の組を持つオブジェクトの入れ  
もの。なんでも入ります。

作り方の例：

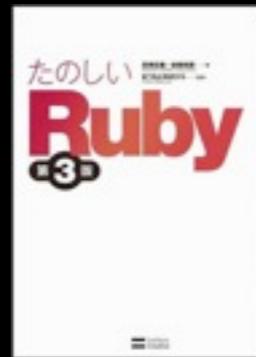
```
recipe = {  
  :title => "♥日向夏のジャム♥",  
  :author => "濱崎" }
```

↑ キー

↑ 値

{}で囲う キー => 値 区切りは , 空配列は {}  
ここでキーに使われている : 始まりのものは何？

# シンボル



教科書  
p.41～

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(古いデータを新データで上書き)

ハッシュオブジェクト内でキーは唯一のもの(ユニーク)

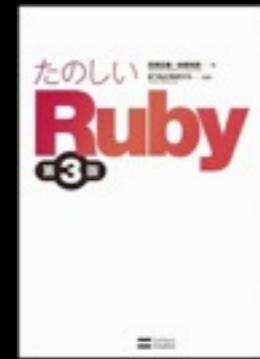
```
recipe = { :title => "♥日向夏のジャム♥",
           :author => "濱崎" }
```

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

追加後

```
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



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

# 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

- ・順番が保持されなくとも困らないもの

※Ruby 1.9 からはHashも順番を保持します。

- ・キーが重複しない場合に利用

# リファレンス マニュアルの検索

# 調べてみよう

## Arrayオブジェクトの全要素を逆順にするには？

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/1.9.3/>  
or 「サクサク Ruby リファレンス」でgoogle検索

The screenshot shows the Ruby 1.9.3 Reference Manual for the Array class. The search bar at the top contains the word "Array". The main content area displays the Array class with its methods listed:

- Array**  
<Enumerable  
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
- Array#&**  
self & other -> Array  
集合の積演算です。両方の配列に含まれる要素からなる新しい配列
- Array#\***  
self \* times -> Array  
配列の内容を times 回 繰り返した新しい配列を作成し返します。
- Array#+**  
self + other -> Array  
自身と other の内容を繋げた配列を生成して返します。
- Array#-**  
self - other -> Array  
自身から other の要素を取り除いた配列を生成して返します。

To the right of the search bar, there is a link to the Ruby 1.9.3 Reference Manual.

**Ruby 1.9.3 リファレンスマニュアル**

## オブジェクト指向スクリプト言語 Ruby リファレンスマニュアル

- Ruby オフィシャルサイト <http://www.ruby-lang.org/ja/>
- version 1.9 対応リファレンス
- 原著：まつもとゆきひろ
- 最新版URL: <http://www.ruby-lang.org/ja/documentation/>

# 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。

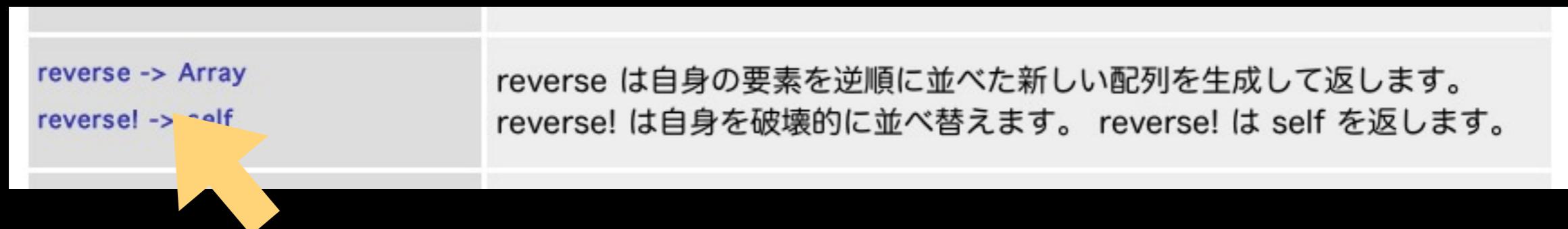
3. 右側にArrayのメソッド一覧などの説明が表示されます。

A screenshot of the detailed documentation for the 'Array' class. The title 'class Array' is prominently displayed. Below it, the inheritance chain is listed: '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' (An array class. An array can hold any Ruby object as an element. Generally, arrays are used as array literals). On the left side, there is a sidebar with the same search results for 'Array' as shown in the previous screenshot.

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

A screenshot of the Ruby 1.9.3 Reference Manual. The URL in the address bar is 'Ruby 1.9.3 リファレンスマニュアル > ライブラリ一覧 > 組み込みライブラリ > Arrayクラス > reverse'. The title of the page is 'instance method Array#reverse'. Below the title, it says 'reverse -> Array' and 'reverse! -> self'. The explanatory text reads: '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。'. At the bottom, there are two code snippets:

```
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を使えばできそうです。

# リファレンスマニュアル検索

## 5. シンプルなコード例を作ってirbで試してみる

```
$ irb
```

```
[3,2,1].reverse  
#=> [1, 2, 3]
```

これが使えそうです。

# 「逆引きRuby」ページもオススメ

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

- 配列
  - プログラムで配列を定義する
  - 配列要素をカンマ区切りで出力する
  - 配列の要素数を取得する
  - 配列に要素を追加する
  - 配列の先頭または末尾から要素を取りだす
  - 部分配列を取りだす
  - 配列を任意の値で埋める
  - 配列を空にする
  - 配列同士を結合する
  - 配列同士の和・積を取る
  - 複数の要素を変更する
  - 配列の配列をフラットな配列にする
  - 配列をソートする
  - 条件式を指定したソート
  - 配列を逆順にする
  - 指定した位置の要素を取り除く
  - 一致する要素を全て取り除く
  - 配列から重複した要素を取り除く
  - 配列から指定条件を満たす要素を取り除く
  - 配列から指定条件を満たす要素を抽出する
  - 配列中の要素を探す
  - 配列の配列を検索する
  - 配列の各要素にブロックを実行し配列を作成する
  - 配列の各要素に対して繰り返しブロックを実行する
  - 配列の要素をランダムに抽出する
  - 実行するたびに取り出される要素が異なります。

## 配列

配列を使うためにはArrayクラスを使います。

### プログラムで配列を定義する

プログラム内で配列を定義するには以下のように記述します。

```
fruits = ["apple", "orange", "lemon"]
scores = [55, 49, 100, 150, 0]
```

配列を参照する場合はArray#[]メソッドを使い、引数として配列の要素番号を指定します。要素番号は0から始まります。上の例ではfruits[0]は"apple"、scores[3]は150を返します。

以下のように配列をネスト（入れ子）にすることもできます。

```
fruits = [3, ["apple", 250], ["orange", 400], ["lemon", 300]]
p fruits[0] #=> 3
p fruits[1][1] #=> 250
p fruits[3][0] #=> "lemon"
```

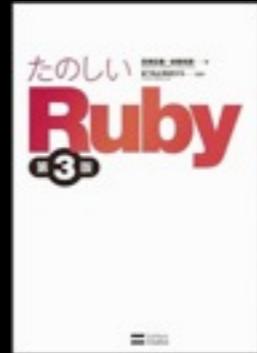
### 配列要素をカンマ区切りで出力する

Array#joinメソッドを使うと配列の要素を任意の文字列で区切った文字列を取得することができます。

```
p ["apple", "orange", "lemon"].join(',') #=> "apple,orange,lemon"
p ["apple", "orange", "lemon"].join('#') #=> "apple#orange#lemon"
p [55, 49, 100, 150, 0].join(',') #=> "55,49,100,150,0"
p [3, ["apple", 250], ["orange", 400], ["lemon", 300]].join(',') #=> "3,apple,250,orange,400,lemon,300"
```

応用  
ファイルから  
読み込み

# 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：ファイルから読み込んで全行表示

```
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

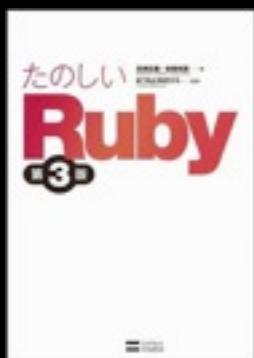
**while ... end** : 条件が偽(false, nil)になるまで繰り返し

# 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 text = file.gets  
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。

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。



while文は  
教科書 p.93

# 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)

# 演習問題

# 演習問題 1

0~29の数字を1行に1つずつ出力してください。  
その際に3の倍数のときは Fizz 、  
5の倍数のときは Buzz 、  
3の倍数かつ5の倍数のときは FizzBuzz を  
一緒に表示してください。

出力例：

0 FizzBuzz

1

2

3 Fizz

4

5 Buzz

...(略)...

15 FizzBuzz

...(略)...

参考：

0~29の数字を1行に1つずつ出力するコード

```
30.times do |i|  
  puts i  
end
```

# 演習問題 2

```
[{:title => "a", :price => 70},  
 {:title => "b", :price => 200},  
 {:title => "c", :price => 50}]
```

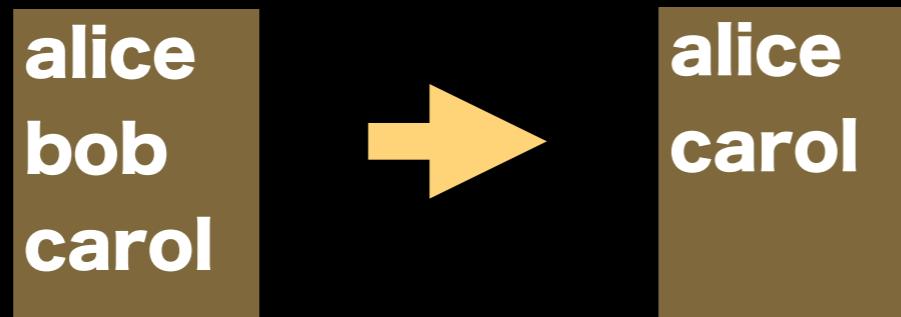
というオブジェクトから、以下のようなオブジェクトを作る  
コードを書いてください。

```
[{:title=>"a", :price=>70, :special=>"Low price!"},  
 {:title=>"b", :price=>200},  
 {:title=>"c", :price=>50, :special=>"Low price!"}]
```

# 演習問題【上級】S1

あるテキストファイルから、aという文字列を含む行だけを抽出した別のテキストファイルを作ってください。

例：



ヒント："sentence" という文字列が書かれたテキストファイルを作るのは以下のコードになります。

```
out_filename = 'out.txt'  
out_file = File.open(out_filename, 'w:UTF-8')  
out_file.puts "sentence"  
out_file.close
```

# 演習問題【上級】 S2

日本の都道府県で、男性と女性の人口差が最も大きいのはどこか、データから解析してください。

ヒント：データは以下にあります。

<http://www.e-stat.go.jp/SG1/estat>List.do>

男女別人口及び世帯の種類(2区分)別世帯数 の CSV

ヒント：文字例をカンマで区切ってArrayにするには

`String#split` を使います。

`"a,b,c".split(",") #=> ["a","b","c"]`

**演習問題**

**解答**

# 演習問題 1 解答

0~29の数字を1行に1つずつ出力してください。  
その際に3の倍数のときは Fizz 、  
5の倍数のときは Buzz 、  
3の倍数かつ5の倍数のときは FizzBuzz を  
一緒に表示してください。

出力例：

0 FizzBuzz

1

2

3 Fizz

4

5 Buzz

...(略)...

15 FizzBuzz

...(略)...

```
30.times do |i|
  if i % (3*5) == 0
    puts "#{i} FizzBuzz"
  elsif i % 3 == 0
    puts "#{i} Fizz"
  elsif i % 5 == 0
    puts "#{i} Buzz"
  else
    puts i
  end
end
```

# 演習問題 2 解答

```
[{:title => "a", :price => 70},  
 {:title => "b", :price => 200},  
 {:title => "c", :price => 50}]
```

というオブジェクトから、以下のようなオブジェクトを作るコードを書いてください。

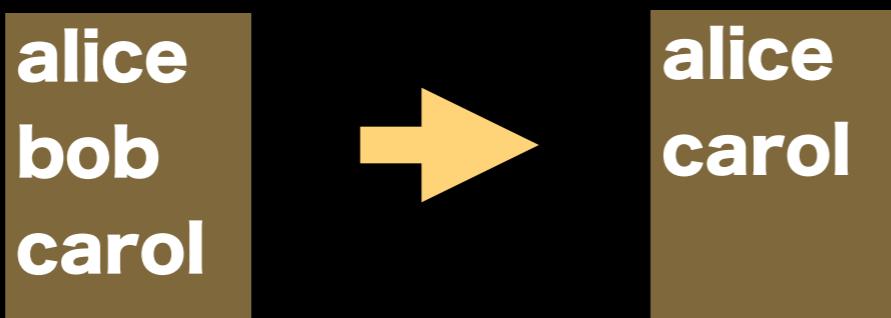
```
[{:title=>"a", :price=>70, :special=>"Low price!"},  
 {:title=>"b", :price=>200},  
 {:title=>"c", :price=>50, :special=>"Low price!"}]
```

```
items = [{:title => "a", :price => 70},  
          {:title => "b", :price => 200},  
          {:title => "c", :price => 50}]  
  
items.each do |item|  
  item[:special] = 'Low price!' if item[:price] < 100  
end  
  
p items  
#=> [{:title=>"a", :price=>70, :special=>"Low price!"},  
#=> {:title=>"b", :price=>200},  
#=> {:title=>"c", :price=>50, :special=>"Low price!"}]
```

# 演習問題【上級】S1 解1

あるテキストファイルから、aという文字列を含む行だけを抽出した別のテキストファイルを作ってください。

例：



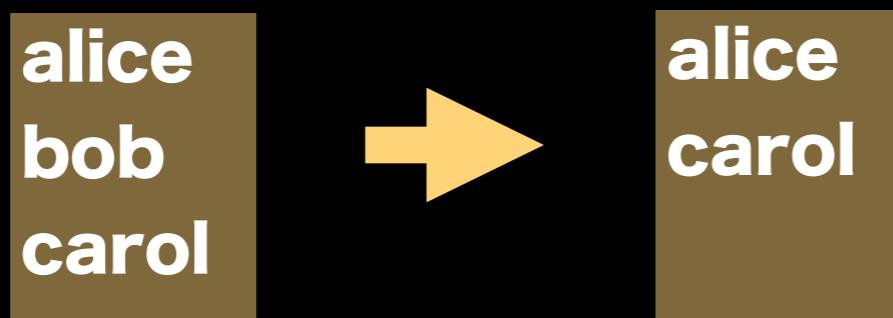
```
# ファイルから読み込み
included = [] # 出力用データを格納するArray
in_filename = 'in.txt'
File.open(in_filename, 'r:UTF-8') do |in_file|
  while text = in_file.gets
    included << text if text =~ /a/
  end
end
# ファイルへ書き込み
out_filename = 'out.txt'
File.open(out_filename, 'w:UTF-8') do |out_file|
  included.each do |i|
    out_file.puts i
  end
end
```

2つのファイルをopenします。  
出力側は"w:UTF-8"(書き込み)  
です。  
また、File.openにブロックを  
渡すとブロック完了時に自動で  
closeします

# 演習問題【上級】S1 解2

あるテキストファイルから、aという文字列を含む行だけを抽出した別のテキストファイルを作ってください。

例：



```
in_filename = 'in.txt'  
out_filename = 'out.txt'  
File.open(in_filename, 'r:UTF-8') do |in_file|  
  File.open(out_filename, 'w:UTF-8') do |out_file|  
    while text = in_file.gets  
      out_file.puts text if text =~ /a/  
    end  
  end  
end
```

in と out を同時に開く書き方です。結果格納用のArrayが不要になるのでスッキリです。

# 演習問題【上級】S2

日本の都道府県で、男性と女性の人口差が最も大きいのはどこか、データから解析してください。

ヒント：データは以下にあります。

<http://www.e-stat.go.jp/SG1/estat>List.do>

男女別人口及び世帯の種類(2区分)別世帯数 の CSV

ヒント：文字例をカンマで区切ってArrayにするには

`String#split` を使います。

`"a,b,c".split(",") #=> ["a","b","c"]`

私もまだ書いていないので、誰か書けたらコードを私まで送ってください。 : )

メソッド

# メソッドの定義、呼び出し

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

メソッド定義には **def** を使う

```
def hello  
  print "Hello, Ruby.\n"  
end
```

メソッド定義  
サンプル

```
hello()
```

呼び出し

メソッド呼び出しはメソッド名に()をつけてます。  
※この()は省略可能です。(曖昧にならない限り)

たのしい  
**Ruby**  
3

教科書  
p.120~152

クラス



# クラス

クラス：オブジェクトの種類を表すもの。型。種族。

[1,2,3]

["a","b"]

[]

"abc"

"ちはやふる"

""

**Array**オブジェクト

= **Array**クラスのオブジェクト

**String**オブジェクト

= **String**クラスのオブジェクト

"abc"や"ちはやふる"は別オブジェクトですが、同じ**String**クラスに属してるので、同じメソッド群が用意されています。

「○○クラスのオブジェクト」を

「○○クラスのインスタンス」と言うこともあります。インスタンスって？

# 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)

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

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

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"

では、クラスを実際に  
作ってみましょう。

# オーソドックスなとんかつ ←title

レシピID:1188379



## description ↑

揚げ物楽しい、豚肉安い、ソースも簡単

hmskpad

## author ↑

1枚くらい

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

マヨネーズ

100ccくらい

## ingredients →

Hashの例で出てきたレシピを  
今回はクラスを使って表現します。

# 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  
  #定義を書く  
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。

# classにメソッドをつくる

Recipeクラスはまだ何も仕事ができないので、  
メソッドを実装して仕事ができるように育てます。

最初に、タイトルを取得できるようにtitleメソッドを作ります。

```
class Recipe  
  def title  
    "cheese cake"  
  end
```

```
end
```

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

メソッドが返す値は、最後の文の実行結果です。

タイトルを返せるようになりました。

しかし決まったタイトルしか返せないので、次はタイトルを設定できるようにしましょう。

# Recipeクラスにタイトルを設定できるようにしてみる

次は、タイトルを取得できるようにtitleメソッドを作ります。

※注：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.title
#=> 6:in `title': undefined local variable or
method `recipe_title' for #<Recipe:
0x10796d3d8> (NameError)
```

title= メソッドを実装してタイトルを設定できるようにしてみました。 recipe\_title 変数へ代入することができました。

しかし、titleメソッドで recipe\_title変数の中身を返すようにしましたが、 recipe\_title変数がないエラーがでました。なんで？？

# エラーの理由は 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
```



ここでは上記の  
recipe\_title変数  
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。

このような変数を  
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。

では、スコープの広い  
変数を作るにはどうす  
ればいいでしょうか？

# 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)
```

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。

# こんなコードになりました

整理すると、こんなコードになりました。

Recipeクラスはレシピのタイトルを格納したり取り出せたりします。

```
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"
```

これでも機能としては足りる  
のですが、コードを少し読み  
やすく改良します。

# コードをちょっと良くします！

@recipe\_titleという変数の名前を変えましょう。

レシピクラスの中なのでレシピなのは当たり前。

ただの @title という名前に変更します。

```
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"
```

```
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"
```



同じ動作

# コードをちょっと良くします2

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。

# レシピクラスができました

ほかの要素 :author, :description, :ingredients も追加します。

```
# -*- coding: utf-8 -*-
class Recipe
  attr_accessor :title, :author, :description, :ingredients
end

recipe = Recipe.new
recipe.title = "cheese cake"
recipe.author = "igarashi"
recipe.description = "やばい"
recipe.ingredients = "（略）"
p recipe.title #=> "cheese cake"
p recipe.author #=> "igarashi"
p recipe.description #=> "やばい"
p recipe.ingredients #=> "（略）"
```

title や author などを格納したり取り出したりできます。

# 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、単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、単語を **\_** 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）

# initialize メソッド

initializeという名前のメソッドを作つておくと、  
インスタンスを作る際(newメソッドが呼ばれた際)に  
自動で実行されます。

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

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

次のページでどんな時に使うか見てみましょう。

# initialize メソッド

newメソッドに引数を渡すと、initializeメソッドで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
```

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

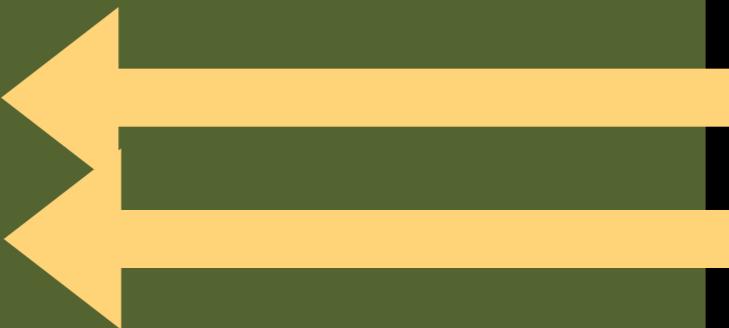
# インスタンスメソッド

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_in
    "COOKPAD"
  end
end

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

クラスに対して呼ぶ。  
newしないで呼ぶ。

# インスタンスマソッドと クラスメソッド

インスタンスマ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
  attr_accessor :title, :author
  def self.published_in
    "COOKPAD"
  end
end
```

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

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

クラスメソッドは  
クラスに対して呼ぶ。  
インスタンスマソッドは  
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_in
  @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。  
(あんこ量はインスタンス変数なので)

# 記法

インスタンスマソッドとクラスマソッドは  
次のような記法で表すこともあります。

クラス名#インスタンスマソッド名  
クラス名.クラスマソッド名

```
class Recipe
  def self.published_in
    "COOKPAD"
  end
  def title
    @title
  end
end
```

**Recipe#title**  
**Recipe.published\_in**

# 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
```

```
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

たとえばBookクラスとMagazineクラス  
(雑誌)を作るとします。

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

```
class Magazie
  attr_accessor :title, :price, :number
end
```

別々に定義を書いてもいいのですが、共通項  
もたくさんあります。

# 継承

そんなときは、継承を使うと便利です。

## 継承を使った書き方

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

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

## 継承を使わない書き方

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

```
class Magazine  
  attr_accessor :title, :price, :number  
end
```

←→  
同じ動作

class Magazine < Book と書くことで、Bookクラスを継承した Magazineクラスを定義できます。 MagazineクラスはBookクラスの性質を受け継ぎます。何を受け継ぐかを次のページで解説します。

# 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、親クラスの全てのインスタンス変数、メソッド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"
```

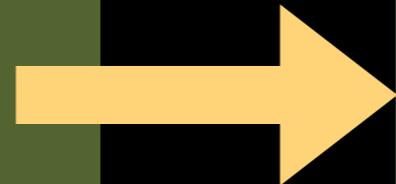
# Module

メソッドを共同利用する仕組み

# Module

クラスでモジュールをincludeすると、moduleに定義してあるメソッドをクラスへ追加することができます。

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



```
class Alice
  include Greeting
end

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

モジュールを定義します。  
モジュールにはメソッドを  
定義します。

include したAliceクラスに  
はhelloメソッドが追加され  
ます。

# 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)

# Module

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

```
class Alice
  include Greeting
end
```



```
class Alice
  def hello
    puts "Hello!"
  end
end
```

# Module

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

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

```
class Alice
  include Greeting
end

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

```
class Bob
  include Greeting
end
```

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

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

# includeとrequireの違い

```
class Sample  
  include Greeting  
end
```

```
require "filename"
```

**include** は Sample クラスに Greeting モジュールで定義されているメソッドを追加する命令

**require** は他のファイル(.rb)で定義されているメソッドやクラス、モジュールを読めるようにする命令

# Gem

Hegarty from  
my work  
With love from

Audrie



# 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

- ▶ spreadsheet
- ▶ Excelファイルを読み書きする
  
- ▶ twitter
- ▶ twitterへのアクセスを簡単にしてくれる

ほかにも数万種類のライブラリが公開されています

## gemを紹介、管理しているサイト

**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

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。
```

# 演習問題

# 演習問題

1. 以下のように使えるPhotoクラスを実装してください。

```
p Photo.info #=> "Photo class"  
photo = Photo.new  
photo.from = "igarashi"  
p photo.from #=> "igarashi"
```

2. 上の1で作ったPhotoクラスを継承したDigitalPhoto  
クラスを作ってください。以下のように使えるとします。

```
photo = DigitalPhoto.new  
photo.from = "hamasaki"  
photo.type = "jpeg"  
p photo.from #=> hamasaki  
p photo.type #=> jpeg
```

# 演習問題解答

1.以下のように使えるPhoto クラスを実装してください。

```
p Photo.info #=> "Photo class"  
photo = Photo.new  
photo.from = "igarashi"  
p photo.from #=> "igarashi"
```

解答例

```
class Photo  
attr_accessor :from  
def self.info  
"Photo class"  
end  
end
```

```
p Photo.info #=> "Photo class"  
photo = Photo.new  
photo.from = "igarashi"  
p photo.from #=> "igarashi"
```

# 演習問題解答

2. 上の1で作ったPhotoクラスを継承したDigitalPhotoクラスを作ってください。以下のように使えるとします。

```
photo = DigitalPhoto.new  
photo.from = "hamasaki"  
photo.type = "jpeg"  
p photo.from #=> hamasaki  
p photo.type #=> jpeg
```

## 解答例

```
class Photo  
attr_accessor :from  
def self.info  
  "Photo class"  
end  
  
class DigitalPhoto < Photo  
attr_accessor :type  
end  
  
photo = DigitalPhoto.new  
photo.from = "hamasaki"  
photo.type = "jpeg"  
p photo.from #=> hamasaki  
p photo.type #=> jpeg
```

# 総復習 Rails編

Ruby編が長くなつたので略。

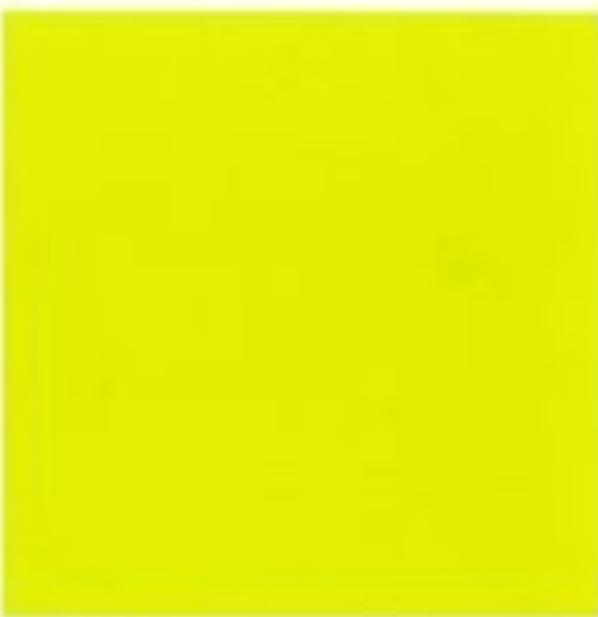
第13回と第14回の資料を読んでください。

- ▶ <https://speakerdeck.com/u/igaiga/p/ruby-and-rails-13>
- ▶ <https://speakerdeck.com/u/igaiga/p/ruby-and-rails-14>

プログラムで  
できること

# 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



誰でも簡単!売買システムから選ぶだけ »

シストレ  
初心者でも  
安心!!

アルゴトレード365には、厳選された297種類の売買システムが搭載されており、トラックレコード(実際に売買システムを稼動させた取引結果の履歴)をもとに各種ランキングデータが公開されていますので、お好みの売買システムを選ぶだけで、あとはパソコンをOFFにしていても**24時間**、ルールに従って自動売買が稼動しつづけます。

### ¥ 利益ランキング

» 一定期間の利益率が高い順に表示します。

### % 勝率ランキング

» 一定期間の勝率が高い順に表示します。

### ★ 登録ランキング

» 登録者(お気に入り登録)が多い順に表示します。

### 売買システムランキング

期間: 1年間

順位	売買システム	損益グラフ	平均損益率	白銀先頭	お気に入り	成績
1	波動ブレイカー USD/CHF 60分足		29.63 %	STOP	★	
2	波動ブレイカー AUD/JPY 60分足		19.68 %	START	★	
3	ムーヴマン EUR/USD 60分足		19.59 %	START	★	
4	波動ブレイcker EUR/JPY 60分足		18.86 %	START	★	
5	スパイクショット AUD/USD 8分足		15.95 %	START	★	



## 年間利益ランキング

シミュレート期間  
2011年5月1日～2012年5月24日



第1位 » ムーヴマン

CHF/JPY 60分足 損益曲線 損益(pips)

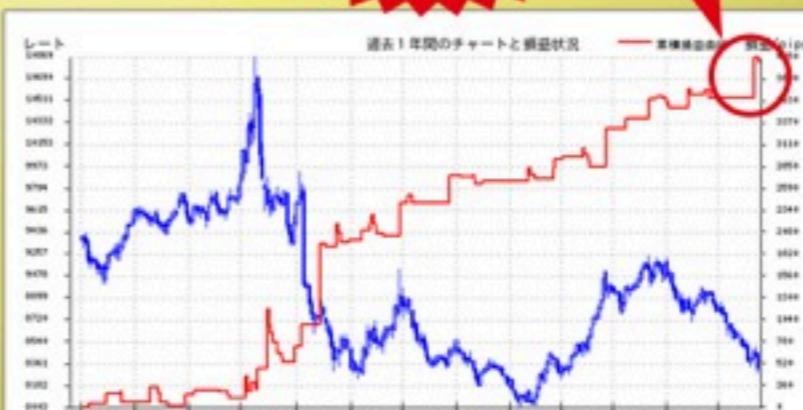
期間損益 +4,097 pips

総トレード数 71回

勝率 49.30%

最大ドローダウン -632p

期間中  
(40.97円)  
4,097 の利益!  
pips



タイプ 順張り

指標 直近の高値安値と過去N期間の値幅の平均

エントリールール

直近最安値+当日の値幅の平均値のX倍を上回ったら、  
買い。  
直近最高値-当日の値幅の平均値のX倍を下回ったら、  
終値で売り。

イグジットルール

売りのエントリールールを満たした場合、転売。  
買いのエントリールールを満たした場合、買い戻し。

ドテン 有り

\*各売買システムに表示されている「平均損益率」、「勝率」等の数値は、過去の結果に基づいて算出したものであり、将来の結果を保証するものではございません。また、売買システムリリース以前の数値は、過去のくりっく365の為替レートを使用して売買システムを稼動させた場合のシミュレーション結果です。

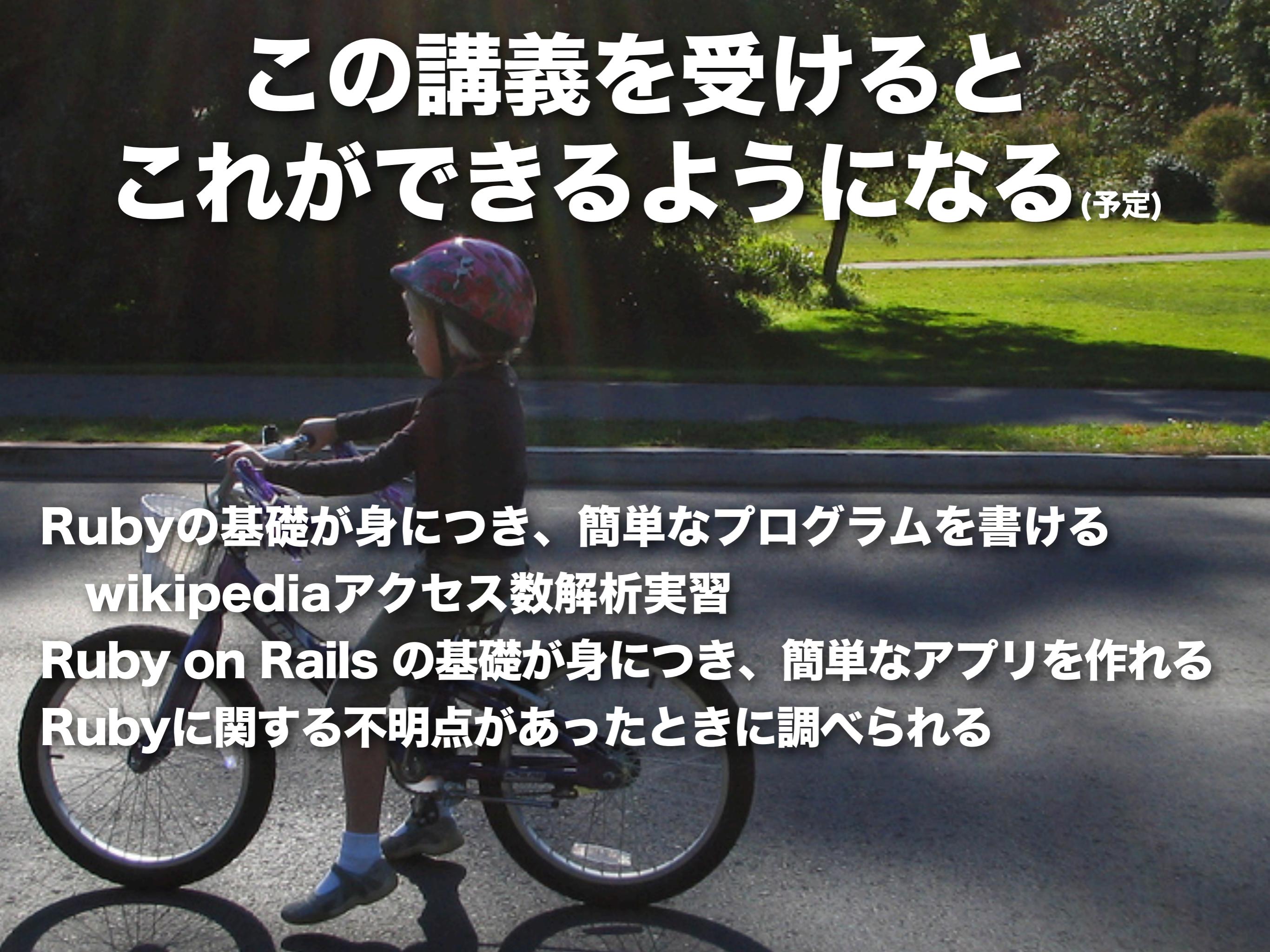
この講義を受けると  
これができるようになる(予定)

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 や CSV の  
データを見たら  
Rubyを思い出してください

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

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



# 講義資料置き場

講義資料置き場をつくりました。  
過去の資料がDLできます。

<https://github.com/hitotsubashi-ruby/lecture2012>

or

<http://bit.ly/ruby-lecture>

# 雑談・質問用facebookグループ

facebookグループを作りました

<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
- ・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