

プログラミングで  
考える力、伝える力を身につけよう！



ジュニアドクター

鳥海塾

Junior-Doctor Chokai Academy

教科書

2024



# 目次

1. みんなへのメッセージ.....	3
2. Hello! プログラミングの世界へ.....	4
3. ファイルを作ろう!.....	5
4. プログラムを作ろう!.....	6
4.1. プログラミングの基礎.....	6
4.1.1. 作成から実行まで.....	6
4.1.2. print.....	7
4.1.3. puts.....	7
4.2. 入出力処理.....	7
4.2.1. 値.....	7
4.2.2. 変数.....	8
4.2.3. gets.....	8
4.2.4. printf.....	9
4.3. 演算.....	11
4.3.1. 算術演算.....	11
4.3.2. 代入演算子.....	12
4.3.3. 論理演算子.....	12
4.4. 制御構造.....	13
4.4.1. if.....	13
4.4.2. while.....	14
4.4.3. break, next, redo.....	15
4.5. 配列.....	16
4.5.1. 配列への値の格納.....	17
4.5.2. for・each.....	18
4.5.3. 配列のループ.....	19
4.5.4. 配列の操作.....	20
4.6. メソッド定義.....	22
4.6.1. def.....	22
4.6.2. メソッドの参照.....	23

4.6.3. 返却値.....	23
4.6.4. return.....	24
4.6.5. 乱数の利用.....	24
4.7. CSV.....	25
4.7.1. CSV ファイルの構造.....	25
4.7.2. CSV ライブラリ.....	25
5. プログラムをもっと楽しくしてみよう！.....	28
5.1. 乱数.....	28
5.1.1. 数値の場合.....	28
5.1.2. 文字列の場合.....	28
5.2. 一時停止.....	29
5.3. 時刻.....	29
5.4. 外部コマンドの起動.....	29
5.4.1. spawn.....	29
5.4.2. system.....	30
6. おまけ.....	31
6.1. Emacs.....	31
6.2. irb.....	32
6.3. コマンド集.....	33
6.4. プログラム例.....	34
7. 参考資料.....	35
7.1. アルファベット表.....	35
7.2. キーボード表.....	36

# 1. みんなへのメッセージ

私達は、PC（パーソナルコンピューター）やゲーム機、スマートフォンに囲まれた社会環境で育ち、このようなデジタル機器を使いこなしています。例えば、ゲームやチャット、そしてインターネットでの情報収集などです。

今やこのような作業は多くの人が当たり前のように行っています。

しかし、ここには大きな問題があります。多くの人は、創造力を発揮することなくデジタル機器を使えています。それは、ゲームやアニメ、シミュレーションに触れて満足しているだけで、自分自身の作品を作り出しているわけではありません。

「ジュニアドクター鳥海塾」では、プログラミングの基礎や地域社会の情報技術について学習していきます。プログラミングを通して、あなたの**考えていることを形にする**ことができます。あなた自身のゲームや物語、社会に貢献できるプログラムを作ることができるのです。

この「ジュニアドクター鳥海塾」の学びでは、理科や数学、情報分野といった科学技術を応用し、創造的に考え、それを伝える技を学ぶことができます。これは社会に出るためのすばらしいスタートです！

作品を作ったら、ぜひ周りのみんなに紹介しましょう！

User name
Password

## 2. Hello! プログラミングの世界へ

### ようこそ、プログラミングの世界へ。

ここでは、プログラミングとはなにか、Rubyとはなにかについて簡単に紹介します。プログラミングとは、コンピュータに「**これこれを、こうやって動かしてほしい!**」と伝えることです。そして、プログラミングによって作られるものがプログラムです。

Rubyはプログラムを作るためのプログラミング言語です。周りにあるものをのぞいてみてください。どんなものでも最初からそこにあるわけではありません。誰かそれを作った人がいます。そして、それを作るためには材料や道具が必要です。

図工の時間を思い出してみましょう。皆さんが何か作るときや絵を描くときは、ノリやハサミ、絵の具、筆で作りましたと思います。

Rubyはプログラムを作るための道具です。プログラムをすることで、ゲームを作ることができます。なんでもしてくれるコンピューターやお絵かきソフトだって作れるかもしれません。なんでもできます。

次の3つの手順を参考に学習を進めよう。

1. さらっとおまけを見ながらPCの使い方をマスター
2. どんどんプログラミング!
3. わからないところは、すかさず前のページを見る

この3つを繰り返して、プログラミングをマスターしよう!

プログラミングは実際に自分でプログラムを作成することで身につきます。

#### ポイント

- あとで見返してもわかるよう、教科書に書き込もう!
- 困ったら、34ページからのおまけを見るといいよ!
- わからなかったら、先生やメンターさんに質問しよう!

### 3. ファイルを作ろう！

プログラミングをする前に、まずは**ファイル**を作りましょう。

例えば、絵を書くときには、紙が必要です。

プログラミングも同じで、プログラムを書くためにも、専用の紙が必要になります。

そのプログラミングの紙のことを「**ファイル**」といいます。

また、多くのファイルを作ったとき、ファイルが散らかっていると、目的のファイルを探すのが大変になります。

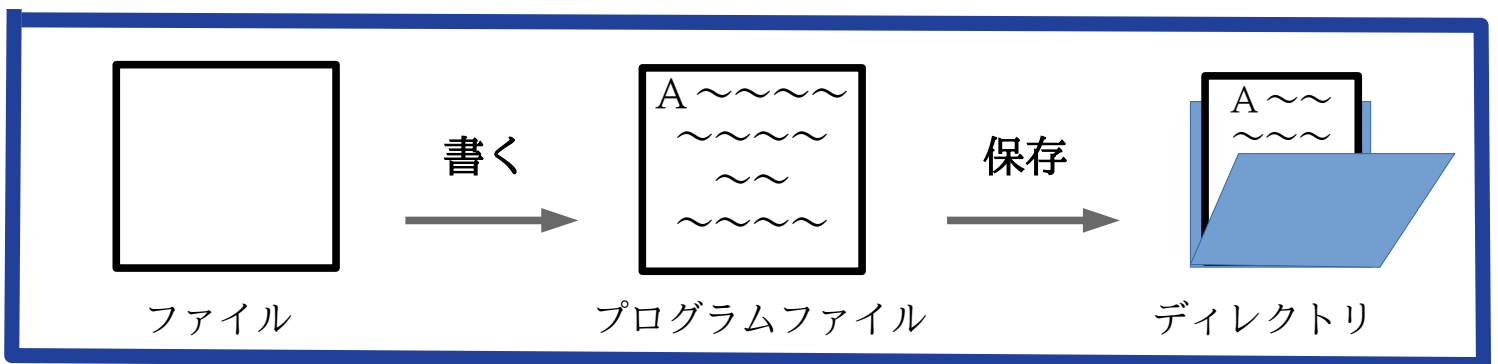
そのため、ファイルを整理しておくための場所が必要です。

ファイルを整理しておく場所のことを「**ディレクトリ**」といいます。

ディレクトリ…ファイルをしまっておく場所

ファイル …プログラムを書く紙

皆さんが、パソコンを起動して最初にいる場所のことを「**ホームディレクトリ**」といい、「<sup>チルダ</sup>~」という記号で表します。今回はその「**ホームディレクトリ**」の下にある「Ruby」というディレクトリを使います。



## 4. プログラムを作ろう！

### 4.1. プログラミングの基礎

まずは、プログラムの実行と編集をしてみましょう。以下で使うキーの説明は、6.1(34ページ)と6.3(36ページ)にあります。

#### 4.1.1. 作成から実行まで

##### ○プログラムファイルを作る

1. C-1 (コントロール Ctrl キーを押しながら 1 を押す)で イーマックス Emacs を選び、  
C-x C-f でファイルを開く。

2. ファイル名を決める。(ここでは hello.rb)

```
Find file: ~/Ruby/hello.rb
```

3. プログラムを 1 行ずつ書いていく。

4. 書き終わったら C-x C-s で必ず保存する。

※~/Ruby/ファイル名.rb … Ruby用のディレクトリの中にファイルを作る。  
.rb … Rubyのプログラムのファイルである。

##### ○プログラムを実行する

1. C-2 で ターミナル terminal を選ぶ。

2. Ruby のディレクトリに入る。

```
% cd Ruby
```

3. 作ったものがあることを確認する。

```
% ls
```

4. Ruby プログラムを実行する。

```
% ruby hello.rb
```

※コンピュータのプログラムは、上から順に実行されます。

##### ポイント

➤ 突然落ちることがあるので、こまめに保存しよう！

➤ `chmod +x hello.rb` とすると、`./hello.rb` で直接起動できるよ！



## 4.1.2. <sup>プリント</sup> print

print は()内の文字列を出力することができます。

### konnichiha.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

print("こんにちは!")
print("JD 鳥海塾の○○○○です。 \n")
print("よろしくお願ひします。 \n")
```

※#!/usr/bin/env ruby … このファイルを実行する方法を指定している。

# -\*- coding: utf-8 -\*- … 文字コードを UTF-8 に指定している。

バックスラッシュエヌ

\n …改行文字と呼ばれる次の行に移動するもの(環境で¥n の場合も)。

## 4.1.3. <sup>プットエス</sup> puts

puts は print とほぼ同じですが、最後に改行が追加されます。

### konnichiha2.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

puts("こんにちは!")
puts("JD 鳥海塾の○○○○です。 ")
puts("よろしくお願ひします。 ")
```

## ポイント

- ファイルの作り方は何度も練習して覚えよう！
- print と puts の違いを確認しよう！

## 発展

- 好きな言葉が出るようにプログラムを作ってみよう！

## 4.2. 入出力処理

プログラム上では、同じ文字でも違う働きを持つことがあります。

### 4.2.1. 値

処理の対象となるデータを**値**といい、文字列と数値を区別して扱います。Ruby のようなプログラムは、値を出し入れすることで動きます。

- 文字列…<sup>ダブルクォーテーション</sup>” や <sup>シングルクォーテーション</sup>’ で囲まれた、文字が並んだ値。  
例) ”hello”, ’1 + 2’
- 数値…足し算や引き算など、計算することのできる値。  
例) 123 + 123 → 246  
※”123” + ”123”と文字列にすると、計算されるのではなく、”123123”のように文字が繋がります。

プログラムでは、これらの値をしまうための箱が必要になります。

## 4.2.2. 変数

値に付ける名前を**変数**といい、好きな名前を付けることができます。

x = 1

y = x + 2

のように、<sup>イコール</sup>= で結んで値を**代入**します。後で利用することを**参照**といいます。

### 注意!!

変数名において、使える文字は英数字とアンダースコア(\_)のみで、最初の文字は英語の小文字にします。

### 具体例

○良い例	×悪い例
programming	programming!
junior_doctor	junior-doctor
chokaiAcademy	chokai academy
first	1st

### ポイント

- 変数を表す名前をつけて、わかりやすくしよう!
- ハイフン(-)や空白を使わないように気をつけよう!

## 4.2.3. <sup>ゲットエス</sup>gets

人間がキーボードに打ち込んだ値は、getsによって取り出せます。入力したときのEnterは、必ず改行文字として文字列の末尾につきます。これを切り取るには、<sup>ドットチョンプ</sup>.chompをつけます。また、入力した文字を数値に変えるときは、<sup>ドットトゥーアイ</sup>.to\_i または <sup>ドットトゥーエフ</sup>.to\_f を使います。

- `.chomp` をつけると改行文字(`\n`)を切り取ります。
- `.to_i` をつけると文字列から整数に変換します。
- `.to_f` をつけると文字列から小数に変換します。

### nyuryoku.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

print("数字を入れる\n")
nenrei = gets.to_i      #キーボードの入力を整数に変換して変数に代入
```


### ポイント

- <sup>シャープ</sup> `#` の後ろには、内容の説明といったプログラムとしては実行されない **コメント** を書くことができるよ！

### 4.2.4. <sup>プリントエフ</sup> printf

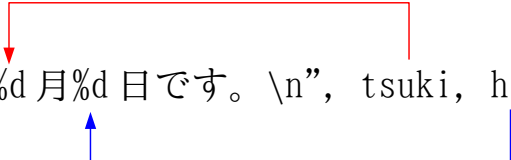
プログラムの結果の出力では、数値を再び文字列に戻す必要があります。  
`printf` を使うと、指定したフォーマットで出力することができます。

```
printf("私は%d歳です。 \n", nenrei)
```



カンマ(,)で区切って複数指定することもできます。

```
printf("今日は%d月%d日です。 \n", tsuki, hi)
```



**%d**…対応する値を整数の文字列に置き換えます。

**%f**…対応する値を小数(小数第6位まで)の文字列に置き換えます。

**%s**…対応する値を文字列に置き換えます。

## nenrei.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

print("数字を入れる\n")
nenrei = gets.to_i
printf("%d歳です。 \n", nenrei)
```

## 桁数の指定

printfでは、%とd, fなどの間に数字を入れると、桁数を指定できます。

### ○プラスだと右詰め

printf("今年は%6d年です。 \n", toshi) → 今年は 2021年です。

### ○マイナスだと左詰め

printf("今年は%-6d年です。 \n", toshi) → 今年は2021 年です。

また、%fは、小数点以下の桁数も指定できます。

printf("円周率は%5.3fです。 \n", pi) → 円周率は3.141です。

## ポイント

- getsとchomp, to\_i, to\_fの関係をおさえよう！
- 変数の名前のルールに気をつけよう！

## やってみよう

- 自分の年齢を変数に代入して自己紹介しよう！

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

nenrei = △△

print("×××××学校□年○○○○です。")
printf("△△歳です。", nenrei)
```

## 4.3. 演算

Ruby 上の数値計算や条件式は、算数や数学とほぼ同じ書き方で記述します。

### 4.3.1. 算術演算

数値計算は、「+」や「-」といった<sup>えんざんし</sup>演算子を用いて行います。

算数や数学では、掛け算の記号に「×」、割り算の記号に「÷」を使いますが、Ruby では、掛け算は「\*」、割り算は「/」を使います。

演算子	意味
+	足し算
-	引き算
*	掛け算
/	割り算
%	<sup>じょうよ</sup> 剰余
**	<sup>べきじょう</sup> 冪乗

演算子には、

```
「**」 > 「*」 「/」 「%」 > 「+」 「-」
```

といった優先順位があります。

ただし、()があれば、先に()内が計算されます。また、丸カッコ()のみを使用し、内側のカッコほど優先順位が高くなります。

#### ○剰余(%)

剰余は、「余り」「余分」「残り」といった意味があり、割り算したときの余りを出してくれます。余りのない「 $2 \div 2 = 1$ 」の場合は「0」、余りのある「 $9 \div 6 = 1 \dots 3$ 」の場合は「3」と表示されます。このように、剰余は「余り」を答として出力します。

例)  $2\%2 \rightarrow 0$        $9\%6 \rightarrow 3$

#### ○冪乗(\*\*)

冪乗は、ある数を何回かけるかというものです。例えば2を3回かけたい、つまり、算数だと「 $2 \times 2 \times 2$ 」、数学だと「 $2^3$ 」の答「8」が出力されます。

例)  $2^{**}3 \rightarrow 8$

### 4.3.2. 代入演算子

変数への代入を伴う演算子を**代入演算子**といいます。

代入演算子	意味		
=	通常代入	x=5	xが5になる
+=	足し算代入	x+=5	xが5増える
*=	掛け算代入	x*=5	xが5倍になる
-=	引き算代入	x-=5	xが5減る
/=	割り算代入	x/=5	xが1/5になる
%=	剰余代入	x%=5	x=x%5と同じ
**=	冪乗代入	x**=5	x=x**5と同じ

x=10のときに、代入演算でどのように変化するかを以下に示します。

代入演算	計算後のxの値
x += 1	11
x *= 2	20
x -= 1	9
x /= 2	5
x %= 3	1
x **= 2	100

### 4.3.3. 論理演算子

制御構造の判定条件式などに利用する演算子を**論理演算子**といいます。ここでは、よく使う**比較演算子**を以下に示します。

比較演算子	意味
==	左側と右側が等しいかどうか
<	左側が右側より小さいかどうか
<=	左側が右側以下かどうか
>	左側が右側より大きいかどうか
>=	左側が右側以上かどうか
&&	「かつ」 → 「○○であり△△である」
	「または」 → 「○○か△△である」
!, not	否定

等式や不等式が成り立つときは トゥルー **true**、成り立たないときは フォルス **false** という値が結果として返されます。true, false は**真偽値**といいます。

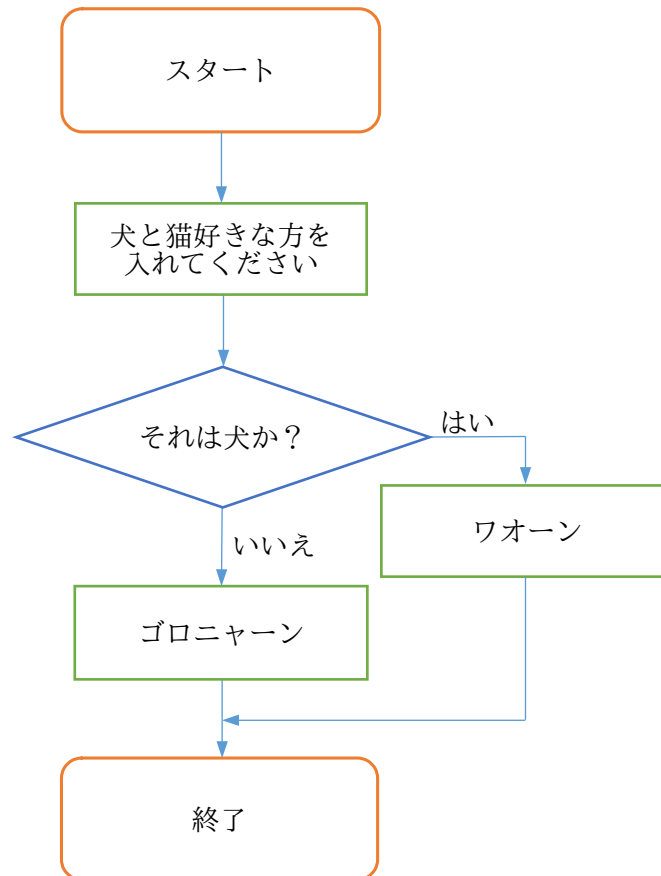
例えば、 $5 > 3$  は true、 $5 == 3$  は false になります。

## 4.4. 制御構造

プログラムは、通常1行目から順番に、一方通行で実行されて終了します。この流れを変えるものが**制御構造**です。プログラムの一部を実行させなかったり、同じ処理を何回も繰り返したりすることができます。演算子を用いて条件を表し、最後に end を書いて制御構造の終わりを示します。

### 4.4.1. if

if は分かれ道を作るときに使います。「もしも○○であるならば△△という処理を行う」といったプログラムを実行することができます。if のように、場合分けして実行結果を変化させる処理のことを**条件分岐**といいます。



#### dobutsu.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

print "犬または猫を入れてください\n"
suki = gets.chomp

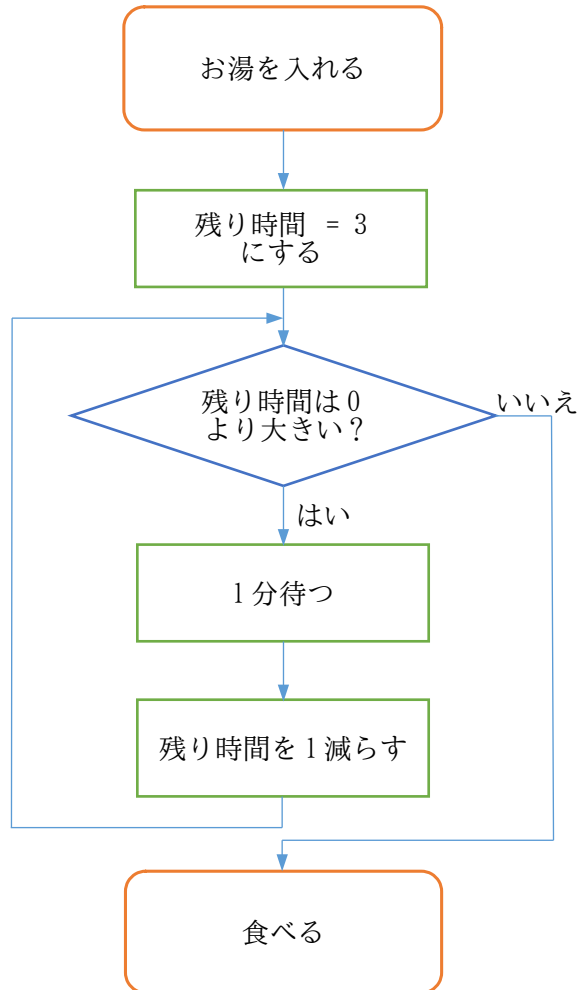
if suki == "犬"
  print "ワオーン"
else
  print "ゴロニャーン"
end
```

#入力が犬の場合  
#条件を満たしたときの処理  
#その他の場合  
#条件に当てはまらない処理

```
end
```

#### 4.4.2. <sup>ホワイル</sup>while

while は繰り返しを作るときに使います。「条件〇〇が正しいとき、△△を繰り返す処理を行う」といったプログラムを実行することができます。while のように、何度も繰り返す処理のことを**ループ**といいます。



#### ramen.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

print "お湯を入れました\n"
nokori = 3

while nokori > 0
  sleep(60)
  nokori = nokori - 1
end

print "いただきます!"
```

#nokori が0より大きいなら処理をする  
#60秒プログラムを止める (P24参照)  
#nokoriを1減少させる

#ループから抜けたときの処理

※while では、条件を true にすると無限ループになります。



## やってみよう

- while のプログラムを、数字や言葉を変えて実行してみよう！

### 4.4.3. break, next, redo

ループを止めたり、飛ばしたり、戻したりする際は、以下のものを使います。

- break…処理を中止して、ループを終わらせる。
- next…処理をスキップして、次のループに移る。
- redo…条件を判断せず、処理を最初からやり直す。

#### 利用例 (mondai.rb)

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

puts "問題！"
puts "今の元号はなんですか？"
puts "1：昭和"
puts "2：平成"
puts "3：令和"

while true
  print "答を入力："
  kotae = gets.chomp.to_i
  if kotae == 1
    puts "ブブー！不正解..."
    break
  elsif kotae == 2
    puts "ブブー！不正解..."
    break
  elsif kotae == 3
    puts "ピンポン！正解!!"
    break
  else
    puts "その選択肢はないよ！"
    redo
  end
end
```

## ポイント

- elsif と puts を入れれば、何個でも選択肢を加えることができるよ！
- 制御構造の最後には、忘れずに end を入れよう！

## 4.5. 配列

複数の値をひとまとめにできるのが**配列**で、下の例のように表現します。

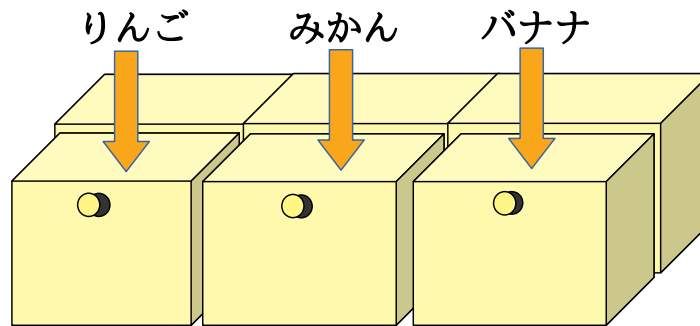
例) kudamono = ["りんご", "みかん", "バナナ"] nedan = [130, 80, 100]

### 注意!!

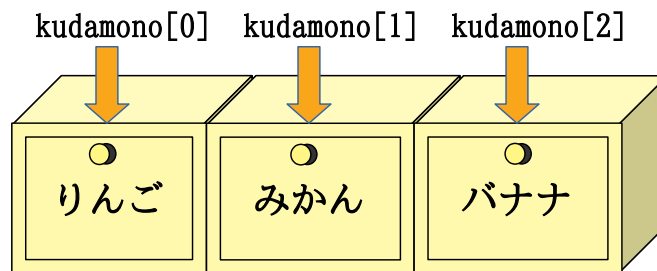
丸いカッコ()ではなく、角ばったカッコ[]を使いましょう。

配列は「引き出し」のようなイメージで考えてみましょう。

kudamono = ["りんご", "みかん", "バナナ"]を引き出しで表すと、下の図のようになり、一つひとつの値を**要素**といいます。



配列の中の一つを表すときは変数[x]という書き方をします。kudamono[0]やnedan[1]のように、0から数えて何番目かを[]内に**添字**として表現することで、特定の位置に値を入れたり、保持された値を参照したりできます。



### 注意!!

配列の要素の位置を指定するときは、1からではなく0から数えます。

## やってみよう

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kudamono = ["りんご", "みかん", "バナナ"]
puts "好きな数字を入れて、値をとり出そう"
x = gets.to_i
puts kudamono[x]
```

- 配列の要素をチームメンバーに変更して、プログラムを書いてみよう！

## kudamono.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kudamono = ["りんご", "みかん", "バナナ"]      #要素を格納
nedan = [130, 80, 100]                        #対応する要素を格納

printf("%s の値段は%d 円です。 \n", kudamono[0], nedan[0])
printf("%s の値段は%d 円です。 \n", kudamono[1], nedan[1])
printf("%s の値段は%d 円です。 \n", kudamono[2], nedan[2])
#指定した添字に対応する kudamono と nedan をそれぞれ表示
```

### 4.5.1. 配列への値の格納

あらかじめ空の配列を作ること、後から要素を追加していくことができます。x = []のように、角カッコ[]の中に何も書かないで、

```
x[0] = 1
x[1] = 2
x[2] = 3
```

と追記して値を格納していきます。

ただ、[添字] = 値と直接指定するのではなく、

```
x << 1
x << 2
x << 3
```

のように、<<を使って末尾に追加していく方が便利に格納できます。

## kyoka.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

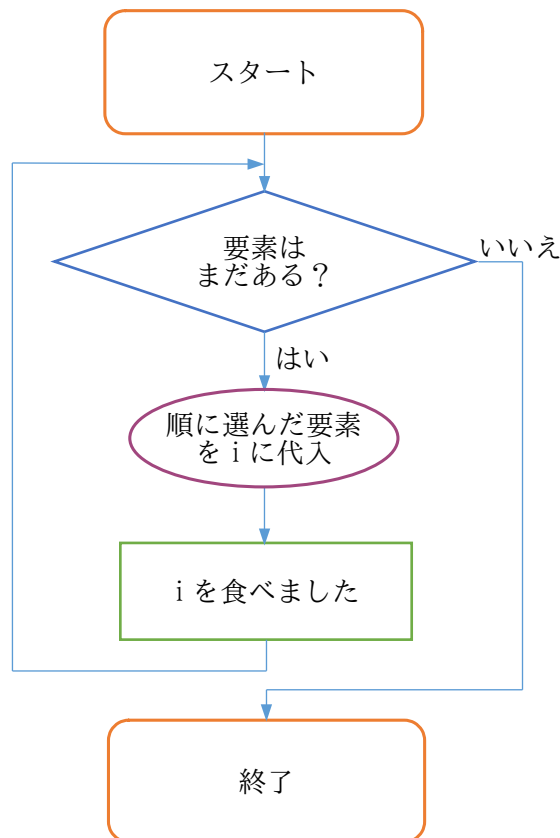
kyoka = []      #空の配列を用意

kyoka << "国語" #kyoka = ["国語"]
kyoka << "数学" #kyoka = ["国語", "数学"]
kyoka << "社会" #kyoka = ["国語", "数学", "社会"]
kyoka << "理科" #kyoka = ["国語", "数学", "社会", "理科"]
kyoka << "英語" #kyoka = ["国語", "数学", "社会", "理科", "英語"]

printf("一時間目は%s です。 \n", kyoka[0])
printf("二時間目は%s です。 \n", kyoka[1])
printf("三時間目は%s です。 \n", kyoka[2])
printf("四時間目は%s です。 \n", kyoka[3])
printf("五時間目は%s です。 \n", kyoka[4])
#指定した添字に対応する kyoka をそれぞれ表示
```

## 4.5.2. for・each

ここで、よく使う制御構造として、whileの仲間のforについて説明していきます。forは配列や範囲などの複数の値を、順に変数へ代入する処理を行うときに使います。「○○の中の××から順に△△という処理を行う」といったプログラムを実行することができます。



### syokuzi.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

for i in ["りんご", "みかん", "バナナ"]#りんごからバナナまで i に代入
  printf("%s を食べました!\n", i)      #要素を順に取り出して出力
end
```

for i in ["りんご", "みかん", "バナナ"]のところは、<sup>イーチ</sup>eachを使って

```
["りんご", "みかん", "バナナ"].each do |i|
```

のように書けば、forと同じ動きになります。

### ポイント

- ループでは、変数 i (整数の integer や添字の index などの頭文字) をよく使うことを覚えておこう！

### 4.5.3. 配列のループ

配列では、要素を1つずつ取り出すことも可能ですが、ループを使って、より実用的にプログラムを作ることができます。大きく分けて、添字を変化させる while と、要素を順次代入する for・each のパターンがあります。

#### ○while の場合

##### nedan\_w.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

nedan = [50, 100, 150, 200]
i = 0 #i の初期値を 0 に設定

while i < nedan.length #i が nedan の要素の数未満の間 (P21 参照)
  printf("%d つ目の値段は%d 円です。\\n", i+1, nedan[i])
  i += 1 #i を 1 増やして次の要素に移動
end
```

#### ○for の場合

##### nedan\_f.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

nedan = [50, 100, 150, 200]
i = 1 #i の初期値を 1 に設定

for n in nedan #nedan の要素を先頭から順次 n に代入
  printf("%d つ目の値段は%d 円です。\\n", i, n)
  i += 1 #i を 1 増やす
end
```

#### ○each の場合

##### nedan\_e.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

nedan = [50, 100, 150, 200]
i = 1 #i の初期値を 1 に設定

nedan.each do |n| #nedan の要素を先頭から順次 n に代入
  printf("%d つ目の値段は%d 円です。\\n", i, n)
  i += 1 #i を 1 増やす
end
```

#### ポイント

- 配列の要素を区切るときは読点(,)ではなくカンマ(,)を使うよ！
- while と for・each をうまく使い分けよう！

## 4.5.4. 配列の操作

ここでは、配列の要素を操作するいくつかの方法を紹介します。

### ○length<sup>レングス</sup>

配列の長さ（要素の数）を返します。

#### nagasa.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

eigo = ["a", "b", "c"]
puts eigo.length          #3 と表示
```

※<sup>サイズ</sup>**size**でも同じように、配列の長さを返すことができます。

### ○push<sup>プッシュ</sup>

配列の末尾に要素を追加します。

#### tsuika.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kazu = [1, 2, 3]
printf("要素が増えて%sになりました。\\n", kazu.push(4))
#kazuに4の入ったkazu[3]が追加され、[1, 2, 3, 4]となって表示
```

### ○shift<sup>シフト</sup>

配列の先頭の要素を取り除き、その値を返します。

#### sento.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kazu = [1, 2, 3]
printf("%dを取り除いて%sになりました。\\n", kazu.shift, kazu)
#kazu.shiftが1を返し、kazuが[2, 3]となって表示
```

## ソート Osort

配列の要素を小さい順に並べ換えた結果を返します。文字列を要素に含む場合は、辞書順に並べ換えた結果を返します。

### narabekae.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kazu = [2, 34, 41, 11, 15]
tango = ["じゅにあ", "どくたー", "ちょうかい", "じゅく"]
printf("%s というふうに数の小さい順になりました。\\n", kazu.sort)
#[2, 11, 15, 34, 41]となって表示
printf("%s というふうに辞書順になりました。\\n", tango.sort)
#[ "じゅく", "じゅにあ", "ちょうかい", "どくたー" ] となって表示
```

## リバース Oreverse

配列の要素の順序を逆にした配列を返します。

### hantai.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kazu = [2, 34, 41, 11, 15]
tango = ["じゅにあ", "どくたー", "ちょうかい", "じゅく"]
printf("%s というふうに逆になりました。\\n", kazu.reverse)
#[15, 11, 41, 34, 2] となって表示
printf("%s というふうに逆になりました。\\n", tango.reverse)
#[ "じゅく", "ちょうかい", "どくたー", "じゅにあ" ] となって表示
```

※sort と reverse を組み合わせると、大きい順や辞書の逆順が得られます。

sort, reverse では元の並びが変わりませんが、!をつけ sort!, reverse! のようにすると直接操作できます。これを**破壊的操作**といいます。

## ポイント

- 配列の書き方はしっかり覚えよう！
- 値を格納するときは、先に配列を用意しよう！
- 配列を操作したら、要素の順序に気をつけよう！

## 4.6. メソッド定義

決められた計算や処理に名前をつけて、いつでも何度でも呼び出すことができます。この決められた計算や処理のことを**メソッド**といいます。今までプログラムで使用してきた、printf, gets, to\_i もメソッドの仲間です。

Ruby でメソッドを呼び出すときは、

何に対して. どうする(必要な情報)

という形にし、「どうする」の部分がメソッド名になります。ここで「必要な情報」のことを**引数**<sup>ひきすう</sup>といいます。また、メソッドによっては、「何に対して.」と「必要な情報」がいない場合もあります。

### 4.6.1. def

メソッドは、def を使って定義することができます。

「x を 2 倍する」という計算に名前をつけて利用してみましょう。これは数学では  $2x$  と書きます。Ruby で "nibai" という名前で計算を表すには、

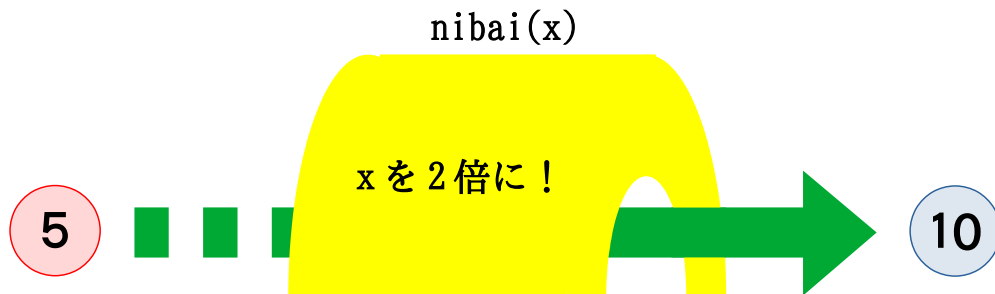
```
def nibai(x)
  x*2
end
```

と書きます。

こうしておいて  $y = \text{nibai}(5)$  とすると、 $5 \times 2$  が計算されて  $y = 10$  になります。

def メソッド名のカッコ内に書くものを**仮引数**<sup>かりひきすう</sup>といい、そのメソッドがもってくる値が自動的に代入されます。次の行から end までの間に、させたい処理や計算を書きます。

上の例の nibai は、「メソッド nibai を呼ぶときには値を 1 つつけてね」という意味です。そのため、別の場所でメソッドを呼びたいときは  $\text{nibai}(5)$  のように 1 つの**引数**<sup>ひきすう</sup>が必要です。メソッドを呼ぶ時に与える値のことを**実引数**<sup>じつひきすう</sup>あるいはたんに引数といいます。



メソッドは、不思議なトンネルをくぐって、変身するイメージです。



## 4.6.2. メソッドの参照

メソッドは、定義しただけでは動きません。プログラムの別の箇所で参照しなければ、何もせずに終わってしまいます。Rubyでは、メソッド名(引数)と書くだけでよいので、`nibai(3)`のようにしてメソッドを呼び出せます。

引数を  $y$  とすると、`nibai(y)` の部分が定義したメソッドの呼び出しになります。まず、直前の行で読み込んだ数値を  $y$  に入れて、`nibai` メソッドに渡します。そして、仮引数  $x$  に入力した値を代入し、計算結果を返します。

### nibai.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

def nibai(x)                #メソッド名と仮引数を指定
  x*2                       #定義本体
end                          #nibai を抜ける

puts "数値を入れてください。2倍します。"
y = gets.to_i              #入力を y に代入
printf("%d\n", nibai(y))  #nibai に渡して結果を返す
```

## 4.6.3. 返却値

複数の文がメソッド定義されている場合、最後に実行した文の値がメソッドの実行結果として返されます。これを**返却値**といいます。

### keisan.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

def keisan(x)
  a = x*2
  b = 1
  c = a + b                #keisan の実行結果
end

puts "数値を入れてください。計算した結果を表示します。"
y = gets.to_i
printf("%d\n", keisan(y))  #x×2+1 を出力
```

## ポイント

- メソッド名も変数名や配列名のように、わかりやすい名前にしよう！
- 引数を2つ以上利用するときは、仮引数をカンマ(,)で区切ろう！

#### 4.6.4. <sup>リターン</sup> return

メソッドの途中で、すぐに呼び主へ制御を返すこともできます。

##### shikakkei.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

def shikakkei(tate, yoko)      #tate と yoko 2つの引数をとる
  if tate < 0 || yoko < 0    #仮引数のどちらかが負(マイナス)の場合
    return nil              #nilを返す
  end
  return tate*yoko          #「縦×横」の結果を返す
end

puts "長方形または正方形の面積を計算します。"
print "縦の長さを入力："
tate = gets.to_i
print "横の長さを入力："
yoko = gets.to_i
printf("この図形の面積は%dです。\\n", shikakkei(tate, yoko))
```

※<sup>ニル</sup>nilは「無」「空」「ゼロ」という意味で、値がない状態を表します。

#### 4.6.5. 乱数の利用

乱数(詳細は31ページに)とメソッドを組み合わせると、次のようなプログラムを作ることができます。

##### janken.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

def janken()                  #引数がいない場合()内は空
  te = ["グー", "チョキ", "パー"]
  te[rand(te.length)]        #乱数を発生 (P27 参照)
end

puts "ジャ〜ンケ〜ン"
sleep(1)                      #1秒間停止
printf("%s!\\n", janken)     #グー, チョキ, パーのいずれかを出力
```

## 4.7. CSV

CSV(Comma Separated Value)形式とは、あらゆる種類のソフトウェアで読み書きできる「表形式」で表現できるデータのことです。

出席番号	名前	学年	住所
------	----	----	----

表の中の1つの行にまとまったデータのことを**レコード**といい、その中身一つ一つを**フィールド**といいます。

### 4.7.1. CSV ファイルの構造

フィールドをカンマ(,)で区切り、1レコードが1行にまとめられています。また、以下のようなルールがあります。

- 数値はそのまま入力
- 文字列はそのままか”(ダブルクォート)で囲う
- 文字列に空白が入るときは必ず”(ダブルクォート)で囲う

123 (数値)

Hello (文字列)

”456” (文字列)

”Hello, World!” (文字列)

#### 例

番号	名前	住所	コメント
002	公益太郎	酒田市飯森山	Hello, everyone!

この表を CSV 形式に直すと、次のようになります。

```
番号,名前,住所,コメント  
002,公益太郎,酒田市飯森山,”Hello, everyone!”
```

※効率的に入力するには Calc がよい→37P 参照

### 4.7.2. CSV ライブラリ

CSV ライブラリは、Ruby で CSV ファイルを便利に使うための道具のようなものです。ファイルデータの読み込み、書き込み、繰り返し処理など、様々な機能があります。使うときは、プログラムの最初の方に `require 'csv'` と書く必要があります。

## ○CSV.read

CSV ファイルからデータを読み込むときに使うメソッドです。

`data=CSV.read("ファイル名.csv")` とすると、`data` という名前の変数に各行が配列化されたものが更に配列化されたものが代入されます。

### 例 shonai.csv

```
建物名, 住所, 電話番号
羽黒山 五重塔, 山形県鶴岡市羽黒町手向, 000-000-000
加茂水族館, 山形県鶴岡市今泉字大久保, 999-999-999
山居倉庫, 山形県酒田市山居町, 123-123-123
```

この CSV ファイルを `CSV.read` で読み込むと次のような配列になります。

```
[["建物名", "住所", "電話番号"],
 ["羽黒山 五重塔", "山形県鶴岡市羽黒町手向", "000-000-000"],
 ["加茂水族館", "山形県鶴岡市今泉字大久保", "999-999-999"],
 ["山居倉庫", "山形県酒田市山居町", "123-123-123"]]
```

また、`data=CSV.read("ファイル名.csv", headers:true)` とすると、CSV ファイルの 1 行目が表の見出しのように扱われ、データを読み出しやすくなります。

### kankou.rb

```
1:#!/usr/bin/env ruby
2:# -*- coding: utf-8 -*-
3:require 'csv' #CSV ライブラリ読込
4:data = CSV.read("shonai.csv", headers:true) #CSV ファイル読込
5:
6:data.each do |row| # rowには1行ずつ値が入り繰り返される
7:   printf("%sの詳細情報は？\n", row["建物名"])
8:   printf("住所 => %s l n 電話番号 => %s \n",
9:         row["住所"], row["電話番号"])
10:   print("-----\n")
11:end #仕切り
```

※ `headers:true` は `:headers => true` と書いても良い。

## ○CSV.open

CSV ファイルにデータを書き込むときは、`CSV.open` メソッドを使います。

`CSV.open(ファイル, モード) do |変数|` と書いたあとに変数にデータを代入すると、そのデータが CSV ファイルに書き込まれます。

モードの部分に次のようなオプションをつける必要があります。

- "r"  
読み込みモードで開く。
- "w"  
書き込み(新規作成)モードで開く。既存のファイルは0バイトにクリアされる。
- "a"  
書き込みモードで開く。ファイルが既に存在する場合は、ファイルの終わりから開始し、なければ新規に作成する。

### kakikomi.rb

```
#!/usr/bin/env ruby
# coding: utf-8
require 'csv'
CSV.open("output.csv", "w") do |csv|
  # csv 変数に値を追加、w は書き込みモード
  csv << ["a", "b", "c"]      # << で1行ぶんの配列を指定する
  csv << [1, 2, 3]
  csv << ["りんご", "みかん", "バナナ"]
end
```

このプログラムを実行すると、次の CSV ファイルが作成されます。

#### output.csv

```
a, b, c
1, 2, 3
りんご, みかん, バナナ
```

### ポイント

- 数値と文字列の違いに気をつけよう！
- ファイルの読み込みや書き込みには、CSV ライブラリを使うよ！
- 使うときはプログラムの最初のほうに require 'csv' と必ず書こう！

## 4.8. 正規表現

P26 の kanou.rb 6 行目の次にデータの絞り込み機能を追加してみましよう。

データから文字・文字列を検索するときにそれらのパターンを指定する一般的な表現方法を**正規表現**といいます。調べたい文字・文字列が正規表現によってすぐに検索することができます。

例えば、i (小文字) が含まれる文字・文字列の場合以下のように検索されます。

○ : i, programming, Application (小文字の i が含まれるので検索される。)

× : I, Image (大文字の I なので検索されない。)

### moji.csv

```
moji
I
Programing
Application
I
Image
```

### kensaku1.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

require 'csv'

data = CSV.read("moji.csv", headers: true)
data.each do |row|
  if /i/ =~ row["moij"]
    puts row["moji"]      #/i/が正規表現
  end
end
end
```

## ポイント

- 正規表現：/(スラッシュ) で囲みます。
- =~ : 正規表現と文字列のマッチを行います。

### 4.8.1. 正規表現のメタ文字

文字・文字列以外に正規表現内で特殊な働きをする文字（メタ文字）で検索することができます。

例えば

- 同じ文字が1回以上続くものを取り出したい (はああい → はあ+い)
- 数字のみのデータを取り出したい (2024年 → \d+年)

というようにメタ文字を組み合わせることで複雑なデータを検索することができます。

※+と\dについては次の「主なメタ表一覧」を見て、マッチする文字列は何なのか、どのような例があるのか確認してみましょう。

## 主なメタ文字一覧

メタ文字	マッチする文字列
.	なんでもいい1文字（改行を含まない）
*	直前の文字0回以上の繰り返し
+	直前の文字1回以上の繰り返し
?	直前の文字0回もしくは1回の繰り返し
文字列1   文字列2	どちらかにマッチすれば良い
\	直後のメタ文字を文字としてマッチさせる
\s	空白文字（スペース、タブ、改行など）
\S	空白文字以外
\d	0～9の数字
\D	0～9の数字以外

## 主なメタ文字のマッチ例一覧

メタ文字	正規表現	マッチする	マッチしない
.	/o.k/	book	back
*	/.*ok/	nippon	japan
+	/-+/	o-----i	oi
?	/ts?u/	tsu, tu	su
文字列1   文字列2	/cat dog/	cat, dog	fox
\	/\+/	10+1	10-1
\s	/e\s a/	We are	Weare
\S	/a\S u/	aisatu	aisa tu
\d	/\d sai/	15sai	juugosai
\D	/\D sai/	Juugosai	15sai

## 4.8.2. 例題プログラム

kensaku.csv

```
name, gakko, gakunen, kozukai
公益太郎, 三川八中, 2, 2000
飯森花子, 余目百中, 1, 4000
鶴岡一人, タキタロウ小, 4, 200
鶴岡二子, タキタロウ小, 2, 300
遊佐梅花, 丸池小, 6, 900
```

## kensaku2.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

require 'csv'

data = CSV.read("kensaku.csv", headers: true)
puts "学校で探す:1"
puts "金額で探す:2"
print "どれにしますか:"
sel = gets.to_i

if sel == 1
  print "探したい学校名を入れてください:"
  ptn = Regexp.new(gets.chomp)
  #gets.chomp で入力された文字列を Regexp.new により正規表現に直す
  data.each do |row|
    if ptn =~ row["gakko"]
      printf("%s さんは%s%s 年です\n", row["name"], row["gakko"],
            row["gakunen"])
    end
  end
elsif sel == 2
  print "絞り込みしたい金額を入れてください:"
  gaku = gets.to_i
  data.each do |row|
    if gaku < row["kazukai"].to_i
      printf("%s さんは%d円もらっています\n", row["name"],
            row["kozukai"].to_i)
    end
  end
  #csv から読んできたデータは全部文字列
  #数字で比較するときは.to_i や.to_f を使う
else
  printf("%d 番はありません。さようなら。 \n", sel)
end
```

### まとめ

- 主なメタ文字を覚えましょう。
- 正規表現について理解できましたか？

### 発展

プログラムの正規表現や CSV データを変更してみよう！

新たに正規表現を使用したプログラムを作成してみよう！



## 5. プログラムをもっと楽しくしてみよう！

### 5.1. 乱数

数字の中からランダムに発生させる数を**乱数**といい、<sup>ランド</sup>**rand**を使って取り出します。数値だけでなく、自分の作った配列から取り出すこともできます。お祭りの屋台にある、おみくじ屋さんからくじを1枚とるイメージです。

#### 5.1.1. 数値の場合

変数名 = rand(自然数)のようにして取り出すことができ、0から指定した自然数未満の乱数が発生します。また、<sup>エスランド</sup>**srand**()によって、発生する乱数の種を初期化できます。srand(10)のように特定の数を指定すると、それに応じて毎回同じ乱数を発生するのでプログラムを直すときに便利です。

※自然数…個数や順番を表すような普段使っている数。プラスの整数。  
種…乱数が発生するときの、最初に設定される値。  
初期化…実行する度に、違う乱数を発生させるために行う処理。

#### saikoro.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

srand()           #乱数の種を初期化
saikoro = rand(6) + 1 #1を足すことで7未満の整数を発生
printf("振ったサイコロの目は%dでした。\\n", saikoro)
#saikoroが1から6までの内ランダムで出力
```

#### 5.1.2. 文字列の場合

取り出したい要素を含む配列と、乱数を発生させる変数を作り、**配列名** [変数名]のようにして出力します。ここで、rand()内の数字には、配列に並んでいる要素の数を入れます。

#### coin.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

coin = ["表", "裏"]           #2個なので coin.length は 2
hyori = rand(coin.length)     #よって 0 か 1 のどちらかになる
printf("投げたコインは%sでした。\\n", coin[hyori])
#coin[hyori]が表か裏のいずれかで出力
```

#### 発展

- Rubyの場合、配列の中からランダムに取り出す処理は coin.sample と書けます。randを使わず coin[hyori]の部分を書けます。

## 5.2. 一時停止

プログラムは実行したらすぐに表示されます。逆にプログラムを一時停止することもできます。プログラムに動きをつけたいときに使ってみましょう。一時停止させたいときは<sup>スリープ</sup>**sleep**を使い、`sleep(秒数)`のようにします。

### rocket.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

puts "ロケット打ち上げまで..."
sleep(1)          #1 秒間停止 (以下同様)
puts "3"
sleep(1)
puts "2"
sleep(1)
puts "1"
sleep(1)
puts "発射!!"
```

## 5.3. 時刻

<sup>タイム</sup>**Time.now**<sup>ナウ</sup>を使うと、現在の時刻を受け取ることができます。秒数を数えるときは、`.to_i`メソッドを利用しましょう。

### stopwatch.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

puts "ストップウォッチスタート！ (Enter を押すと止まるよ) "
kaishi = Time.now.to_i          #現在の時刻を代入
teishi = gets.chomp            #入力完了まで時間経過
syuryo = Time.now.to_i         #Enter を押した時刻を代入
jikan = syuryo - kaishi        #要した秒数を代入
printf("タイムは%d秒です。 \n", jikan) #要した秒数が出力
```

## 5.4. 外部コマンドの起動

Ruby プログラムの中から、外部コマンドを起動させたい場合があります。そこで使えるのが<sup>スポン</sup>**spawn**と<sup>システム</sup>**system**です。これらによって、terminal でコマンドを実行したときと同じ結果が得られます。

### 5.4.1. spawn

プログラムの終了を待ち合わせずに、外部コマンドを起動します。起動したプログラムの ID が返るのでこれを保存しておいて、外部コマンドを止めたいときに<sup>プロセス</sup>**Process.kill**<sup>キル</sup>に渡して終了させます。

## ○画像表示

`spawn "display ディスプレイ -geometry ジオメトリ +x+y ファイル名"`では、画像の表示ができます。  
 $x$ には画面左から、 $y$ には画面上からのドットの位置を指定します。

### gazo.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

pid = spawn "display -geometry +10+20 image.jpg" #image.jpgの場合
#しばらく別の処理をする (sleep など)
Process.kill(:INT, pid) #:INT は信号の一種
```

## 5.4.2. system

プログラムの終了を待ち合わせて、外部コマンドを起動します。プログラムが途中で止まらないように、気をつける必要があります。

## ○Web サイトへのリンク

`system "firefox ファイアーフォックス URL"`を使うと、ブラウザを起動して  $URL$  に飛ばせます。

### firefox.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

system "firefox https://www.koeki-prj.org/jd/home/" #JDWeb を表示
```

## ○外部ファイルの実行

`system "ruby ファイル名"`によって、他のプログラムを呼び出せます。

### ruby.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

system "ruby sample-hello.rb" #sample-hello.rb を実行
```

## 6. おまけ

### 6.1. <sup>イーマックス</sup> Emacs

Emacs には便利なコマンドがあり、入力する際はコントロールキー (CONTROL や Ctrl, CTL) を使います。

C-<文字> : Ctrl キーを押しながら、文字キーを押す。

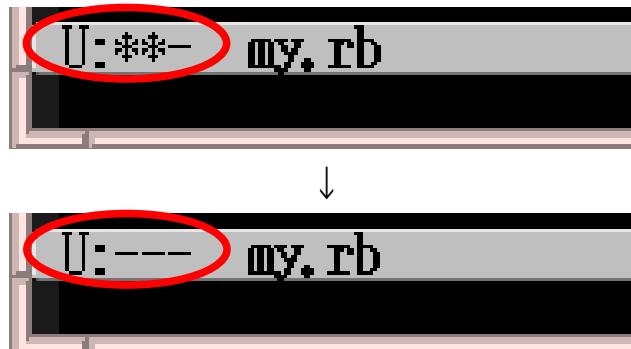
例) C-f は Ctrl キーを押しながら、f のキーを押す。

#### ○ファイルを作ろう！

1. C-x C-f の順番で入力する。
2. ファイル名を入力して、Enter を押す。

#### ○ファイルを保存しよう！

1. C-x C-s の順番で入力する。
2. Emacs の左下にある U:\*\*- が U:--- に変わったら保存完了！



※\*\*印のまま Emacs を終了してしまうと、編集した成果が台無しになるので注意しましょう。

- 取り消し (UNDO)  
もし文章を変えたあとで「あ、間違った！」と思ったら…  
→ **C-x u** で変更の一つ前に戻る。
- もし Emacs が反応しなくなったら…  
キーを押していてわけがわからなくなったら…  
→ とりあえず **C-g** を押してみる。

困ったら **C-g**

## 6.2. <sup>ファイアールビー</sup> irb

irb(Interactive Ruby)とは、対話的にプログラムを実行していくもので、即座に結果を返してくれます。簡単な計算はもちろんのこと、コードを確認するためにも使えます。

### 手順

1. C-2 で terminal を選ぶ。
2. 「irb」と入力して、Enter キーを押す。

```
% irb
irb(main):001:0>
```

3. 終了したいときはC-dを押す。

### 利用例

#### ○計算してみよう！

```
irb(main):001:0> 1 + 1
=> 2
irb(main):002:0> x = 1
=> 1
irb(main):003:0> y = 2
=> 2
irb(main):004:0> x + y
=> 3
```

#### ○コードを確認してみよう！

```
irb(main):005:0> puts "Hello!"
Hello!
=> nil
irb(main):006:0> printf("%d+%d=%d です。 \n", x, y, x + y)
1+2=3 です。
=> nil
```

=> の後ろには、実行したコードの結果として返された値が表示されます。

☆前に実行した処理は、↑（上のカーソルキー）で再度読み出せるよ。

## 6.3. コマンド集

イマックス

Emacsへ移動

C-1

ターミナル

terminalへ移動

C-2

コンソール

consoleへ移動

C-3

### Emacs 編

C-x C-f	ファイルを作成する/開く
C-x C-s	ファイルの保存
C-x C-w	ファイル名を変えて保存（名前を間違えたとき便利）
C-x u	1つ前の状態に戻す
C-x 1	画面の2分割を元に戻す
C-x 2	画面を上下に2分割
C-x 3	画面を左右に2分割
C-x h	すべて選択（C-x h Tabと続けて押すとインデントを直せる）
C-SPC	ポイントのマーク
C-w	マークした領域を切り取り
M-w	マークした領域をコピー
C-y	コピーした領域を貼り付け
C-g	コマンドの中止（困ったらとりあえず押してみよう！）

※SPC…スペースキー ポイント…文字の入る点滅した箇所  
 マークした領域…C-SPCした位置から移動したポイントまでの場所  
 M-<文字>…ESCキーを押し、手をすぐ離してから文字キーを押す

### terminal 編

cd ディレクトリ名	ディレクトリに移動
cd	ホームディレクトリに移動
mkdir ディレクトリ名	ディレクトリを作成
ls	今のディレクトリのファイルを表示
rm ファイル名	ファイルを削除
mv ファイル名(複数OK) ディレクトリ名	ファイルをディレクトリに移動する
mv 旧ファイル名 新ファイル名	ファイル名を変える
ruby ファイル名	ファイルを実行
C-c	プログラムの停止

### console 編

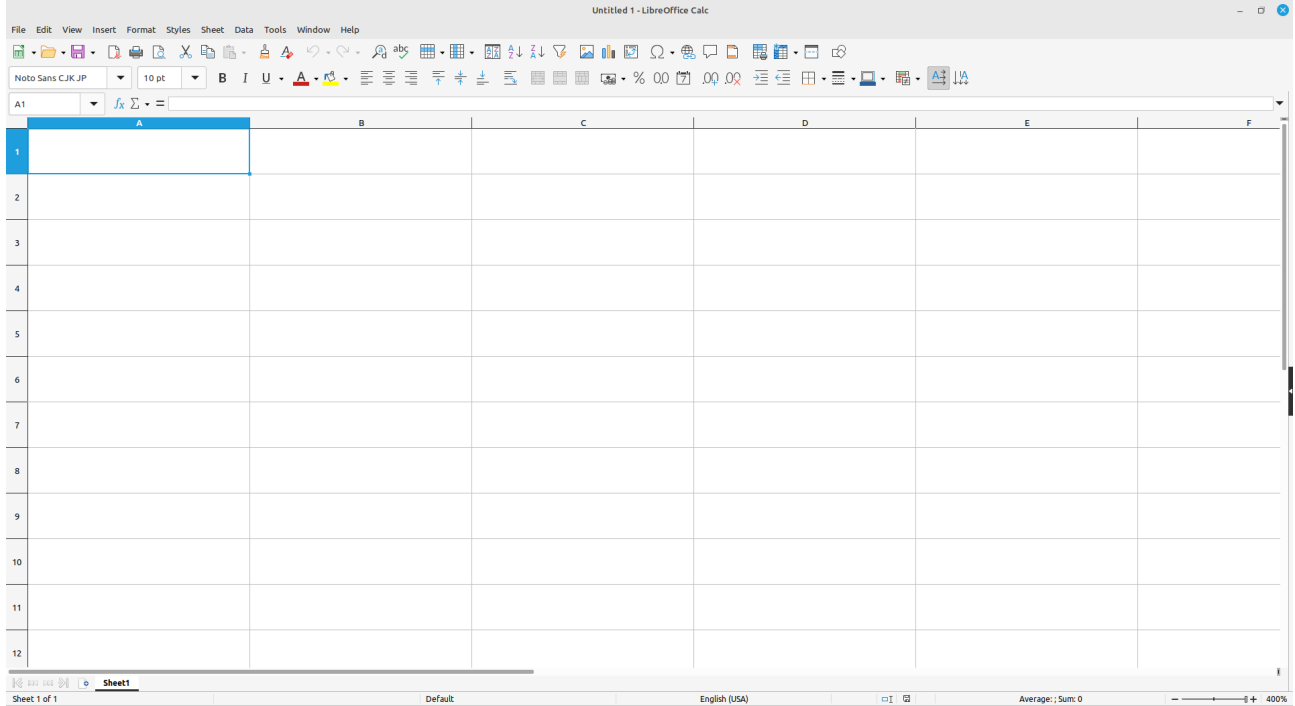
exit	ログアウト
shutdown -p now	電源を落とす

☆Emacs や terminal では、Tab でファイル名を補完できるよ。ファイル名を途中まで入力して、Tab キーを押すと続きのファイル名を自動出力するよ。

## 6.4. Calc を用いた CSV ファイルの作成

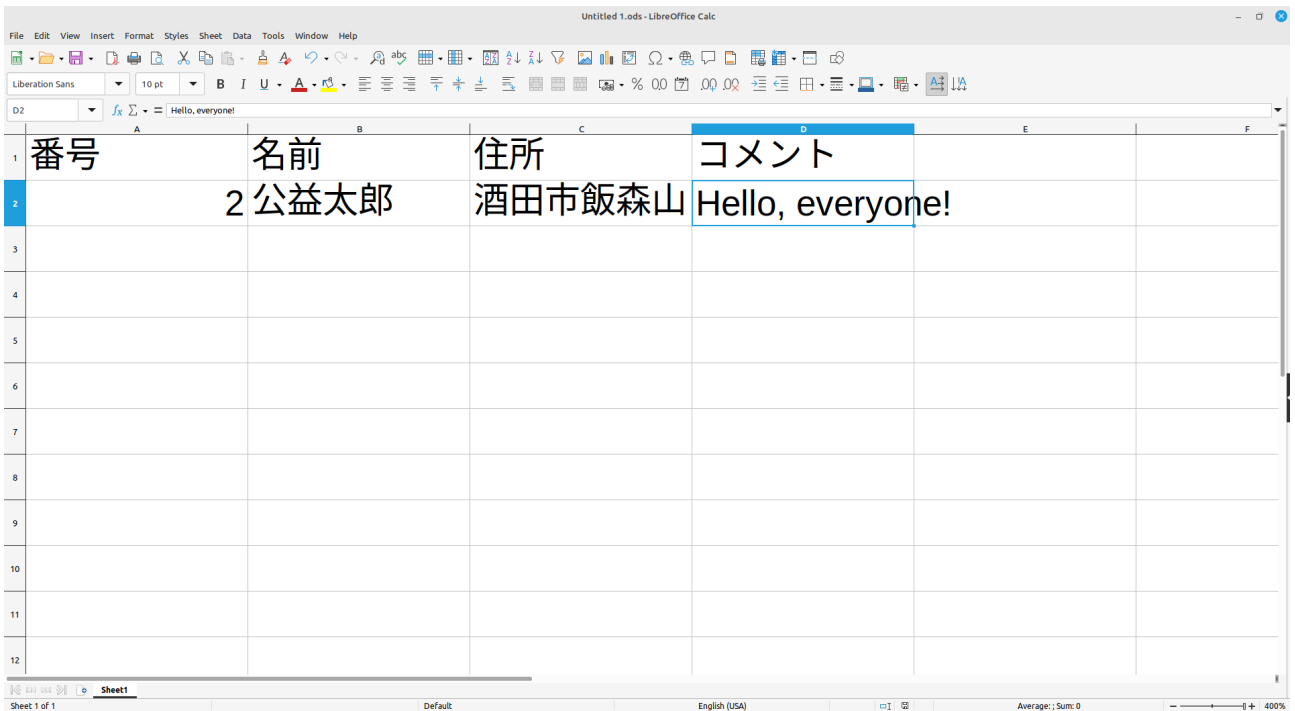
csv ファイルを効率的に作るために LibreOffice の表計算ソフトである Calc を使ってみましょう。

### 起動画面



1 のフィールドにはヘッダー名、2 以上のフィールドにはデータを入力します。4.7.1 の csv ファイルを参考に入力していきます。

### 入力後



保存するには

- file → Save as...を選択
- Name にファイル名を入力し、保存する場所を選択
- 下の方にある ODF Spreadsheet(.ods)を Text CSV(.csv)に変更
- Text CSV形式をクリック → OK をクリック

## 6.5. プログラム例

### kujibiki.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

kuji = ["1等", "2等", "3等", "4等", "ハズレ"]

puts "何がでるかな～?"
sleep(1)
puts "... "
sleep(1)
srand()
nani = rand(5)
printf("あなたは%sでした!\n", kuji[nani])
```

### reji.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

puts "スーパーのレジだよ。"
puts "買ったものの値段を入れてね。(100円→100と入力しよう)"
gokei = 0

while true
  puts "値段は? (終わりたい時はqを押してね)"
  nedan = gets.chomp

  if nedan == "q"
    break
  end

  gokei += nedan.to_i
  printf("今の小計は%d円だよ。 \n", gokei)
end

printf("合計%d円です。 \n", gokei)
puts "お買い上げありがとうございます!"
```

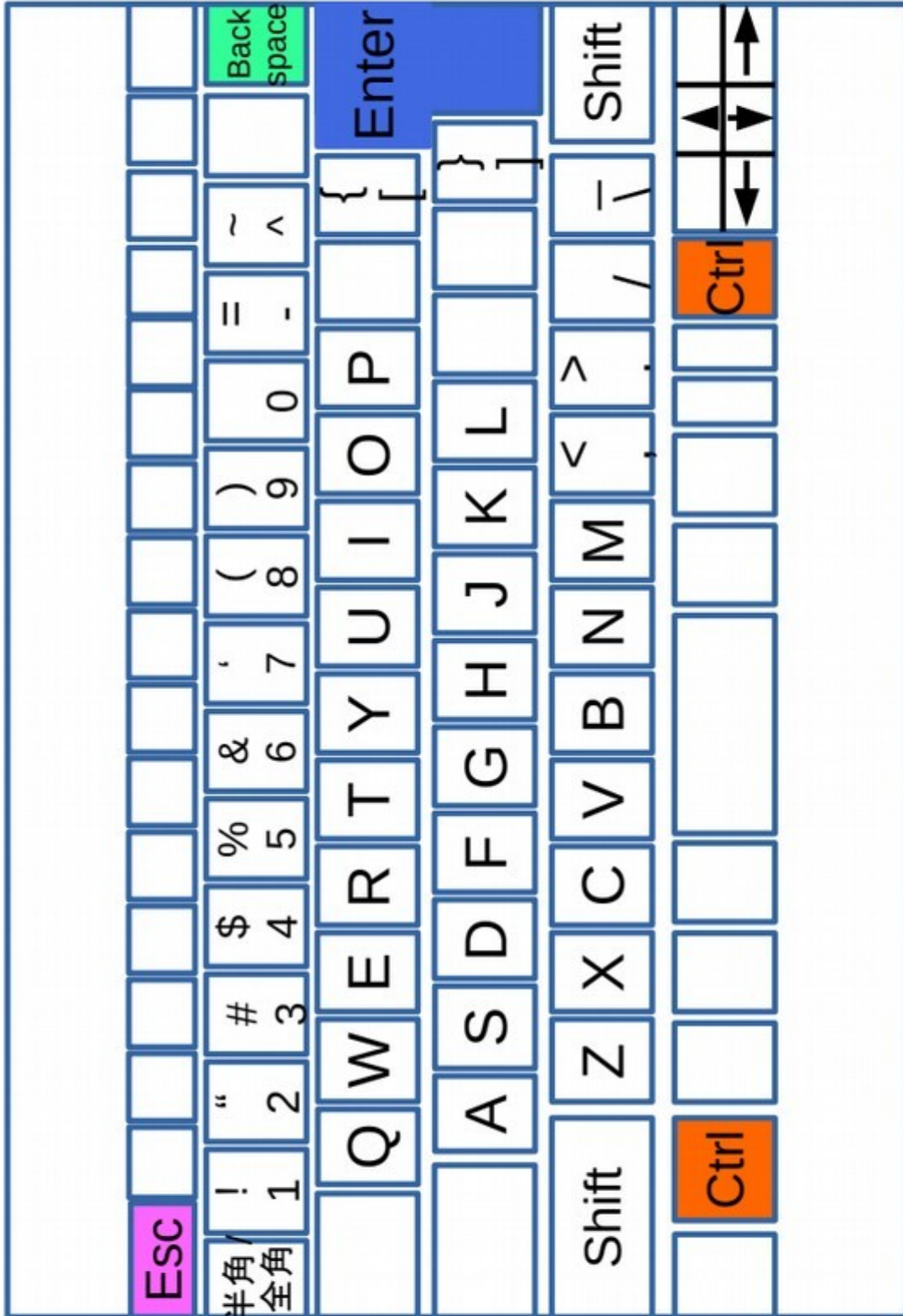


## 7. 参考資料

### 7.1. アルファベット表

A a	B b	C c .chomp, case	D d def
E e elsif, else, end	F f	G g gets	H h
I i if	J j	K k	L l ls
M m	N n \n	O o	P p print, puts, printf
Q q	R r ruby, rand	S s sleep	T t .to_i, .to_f
U u	V v	W w while, when	X x
Y y	Z z		

## 7.2. キーボード表



## ジュニアドクター鳥海塾 教科書

2021年8月21日 初版 第1刷発行  
2021年9月16日 第2版 第1刷発行  
2022年1月8日 第3版 第1刷発行  
2023年3月15日 第4版 第1刷発行  
2023年3月20日 第5版 第1刷発行

著者 東北公益文科大学 公益ジュニアドクターセンター  
発行所 東北公益文科大学 公益ジュニアドクターセンター  
〒998-8580  
山形県酒田市飯森山三丁目5番地の1  
電話 0234-41-1115  
<https://www.koeki-prj.org/jd/home/>

©2021-2024 東北公益文科大学 公益ジュニアドクターセンター

この事業は、国立研究開発法人科学技術振興機構(JST)による令和3年度「ジュニアドクター育成塾」事業に採択され実施しています。



Let's enjoy programming!!



Name