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

2021

教科書

目次

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が、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、筆で作り上げたと思います。

Ruby はプログラムを作るための道具です。プログラムを作ることで、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。

さあ、プログラミングをはじめよう！

3. 学習の手引き

プログラミングをはじめる前に、用語について確認しましょう！

ジュニアドクター鳥海塾の授業では、様々な記号や文字、もしくはそれらを組み合わせたものが出てきます。例えば、《%》や《”》、《C-1》、《C-2》などです。

少し難しいように見えますが、安心してください！
覚えようとしなくても、教科書を読みながら学ぶことが出来ます。

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！
- わからなかったら、先生やメンターさんに質問しよう！

4. 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**を作ろう。

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。

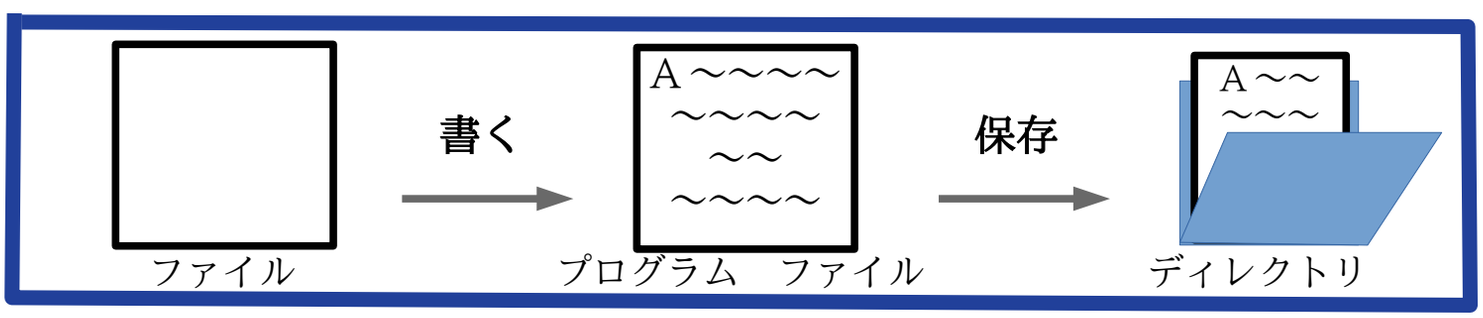
そのため、ファイルを整理しておくための場所が必要です。

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、「~（チルダ）」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。



5. プログラムを作ろう！

5.1. プログラミングの基礎

5.1.1. 作成から実行まで

○プログラムファイルを作る

1. C-l で **Emacs**（イーマックス）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/…Ruby 用のディレクトリの中にファイルを作ること。
.rb…Ruby のプログラムのファイルであること。

○プログラムを実行する

1. C-2 で **terminal**（ターミナル）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！

やってみよう

- 試しにプログラム（sample-hello.rb）を動かしてみよう！

5.1.2. print（プリント）

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(”よろしくお願いします。\n”)
```

※#!/usr/bin/env ruby…ruby のあるディレクトリを指定している。

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

\n（バックスラッシュエヌ） …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)

5.1.3. puts（プットエス）

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(”よろしくお願いします。”)
```

注意!!
#!/usr/bin/env ruby は./*ファイル名*での実行、# -*- coding: utf-8 -*- は文字化けの防止に必要なので、この 2 行を必ず初めに書きましょう。

まとめ

- ファイルの作り方を確認しましょう。
- print，puts はそれぞれ何をするかわかりましたか？

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！

5.2. 入出力処理

5.2.1. 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…ダブルクォーテーション”　　”や シングルクォーテーション’　　’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。

5.2.2. 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、イコール= 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！

5.2.3. gets（ゲットエス）

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（エンター）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**.chomp**（チョンプ）をつけます。また、入力した文字を数値に変えるときは、**.to_i**（トゥーアイ）または**.to_f**（トゥーエフ）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

➢ # （シャープ） 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！

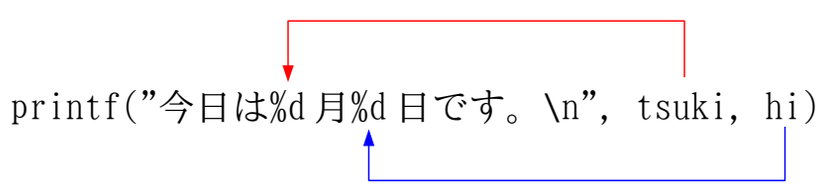
5.2.4. printf（プリントエフ）

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(小数点第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。

まとめ

- getsとchomp, to_i, to_fの関係をおさえましょう。
- 変数のルールについて理解できましたか？
- printfの使い方はわかりましたか？

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)
```

5.3. 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。

5.3.1. 算術演算

数値計算は、「+」や「-」といった**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
%	剰余
**	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。

〇剰余(%)

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÷2＝1」の場合は「0」、余りのある「9÷6＝1…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　→　0　　9%6　→　3

〇冪乗(**)

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「2×2×2」、数学だと「2^3」の答「8」が出力されます。

例）2**3　→　8

5.3.2. 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

5.3.3. 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」
!	否定
not	否定

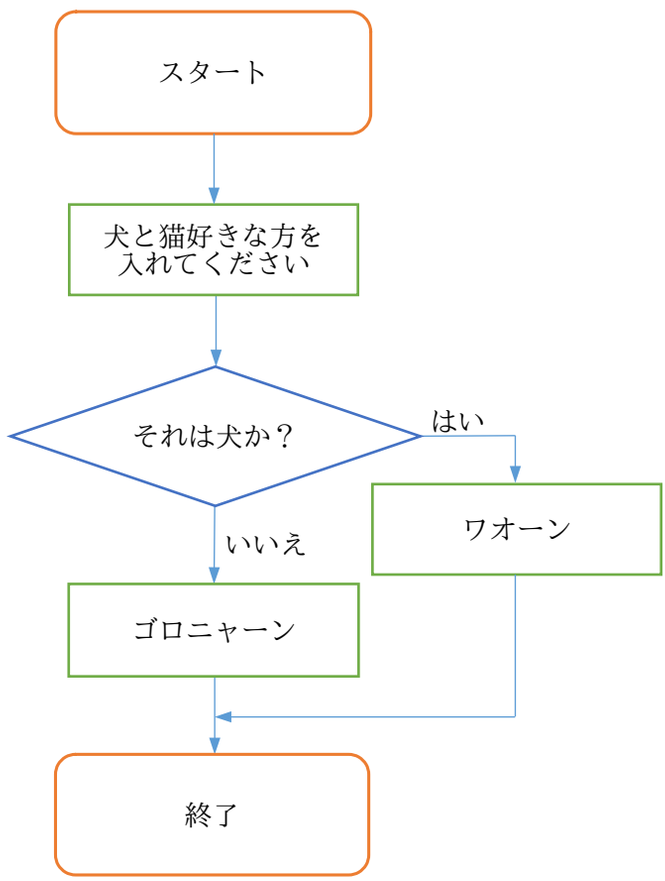
等式や不等式が成り立つことを **true**(トゥルー)、成り立たないことを **false**(フォルス)で表します。

5.4. 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を書いて処理を終了させます。

5.4.1. if（イフ）

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。

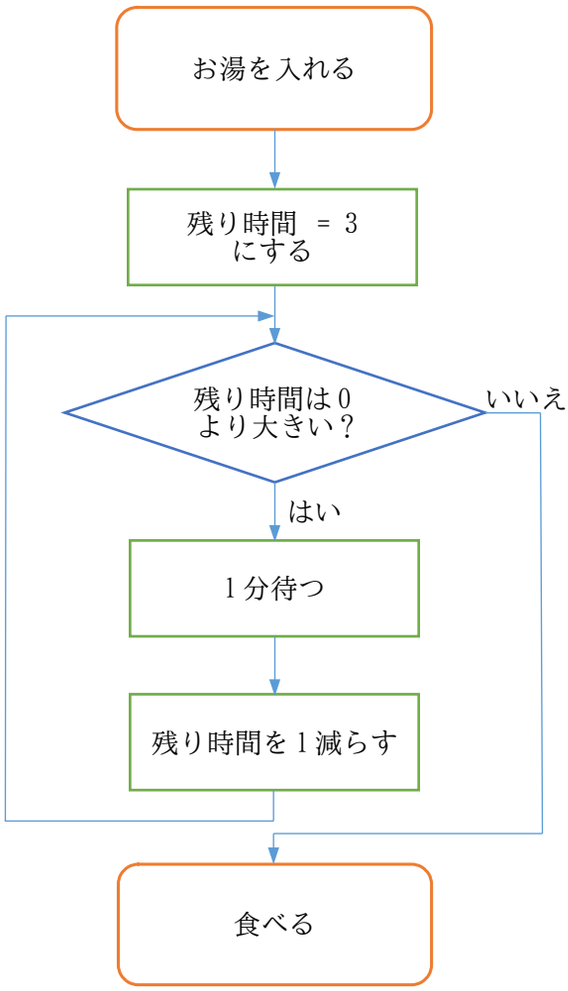


dobutsu.rb

```
#!/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”                    #入力が犬の場合
  print ”ワオーン”                 #条件を満たしたときの処理
else                               #その他の場合
  print ”ゴロニャーン”             #条件に当てはまらない処理
end
```

5.4.2. while（ホワイル）

while は繰り返しを作るときに使います。「条件○○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                #nokori が 0 より大きいなら処理をする
  sleep(60)                     #60 秒プログラムを止める（P24 参照）
  nokori = nokori - 1           #nokori を 1 減少させる
end

print ”いただきまーす！”        #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！

5.4.3. break(ブレイク), next(ネクスト), redo(リドゥー)

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。

まとめ

- 選択肢の入れ間違いがないようにしましょう。
- 演算の仕方や、if と while の違いはわかりましたか？

利用例（mondai.rb）

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

puts ”問題！”
puts ”今の元号はなんでしょうか？”
puts ”1：昭和”
puts ”2：平成”
puts ”3：令和”

while true                          #対応する end までの処理が無限に続く
  print ”答を入力：”
  kotae = gets.chomp.to_i           #解答を入力
  if kotae == 1
    puts ”ブブー！不正解...”
    break                           #条件 1 を満たしループ終了
  elsif kotae == 2
    puts ”ブブー！不正解...”
    break                           #条件 2 を満たしループ終了
  elsif kotae == 3
    puts ”ピンポーン！正解!!”
    break                           #条件 3 を満たしループ終了
  else
    puts ”その選択肢はないよ！”
    redo                            #”答を入力：”まで戻る
  end
end
```

ポイント

➢ elsif(エルスイフ)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！

5.5. 配列

5.5.1. 配列とは

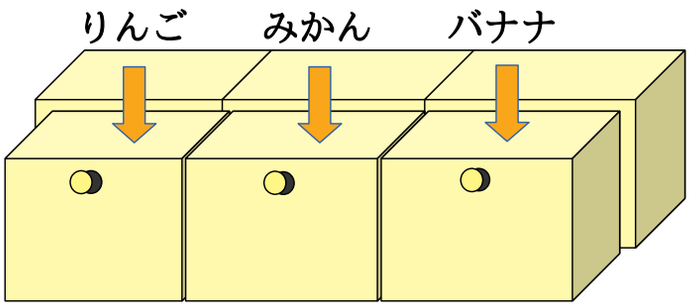
複数の値をひとまとめにできるのが**配列**で、下の例のように表現します。

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

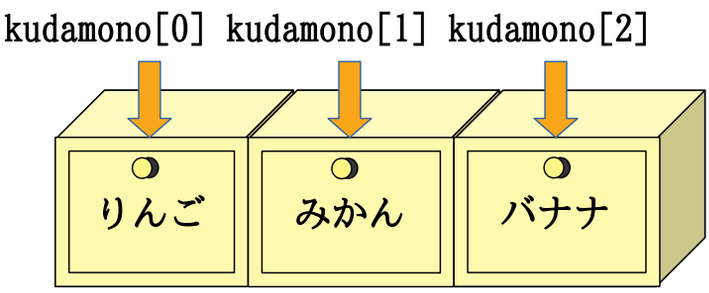
注意!!
丸いカッコ()ではなく、角ばったカッコ[]を使いましょう。

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**(そえじ)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

- 下記のプログラムを実行してみよう！

```
#!/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
```

5.5.2. 配列への値の格納

あらかじめ空の配列を作ることで、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おき、

```
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
```

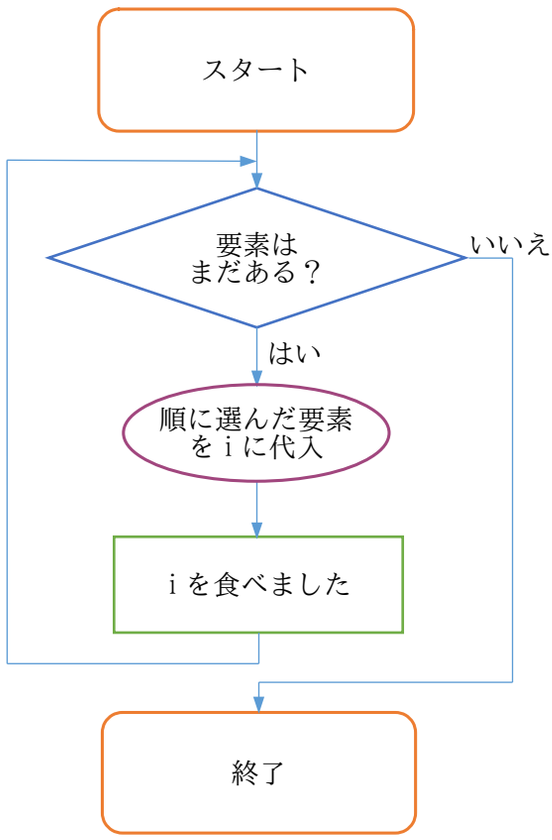
のように、<<を使って末尾に追加していく方が便利に格納できます。

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
```

5.5.3. 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、**each**（イーチ）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！

5.5.4. 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！

5.5.5. 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（レングス）

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
```

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

○push（プッシュ）

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（シフト）

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
```

○sort（ソート）

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
```

○reverse（リバース）

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。

まとめ

- 配列の書き方をしっかり覚えましょう。
- ループをうまく配列に利用できましたか？
- 配列の操作方法はわかりましたか？

5.6. 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**(ひきすう)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」がいらない場合もあります。

5.6.1. def(デフ)

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、

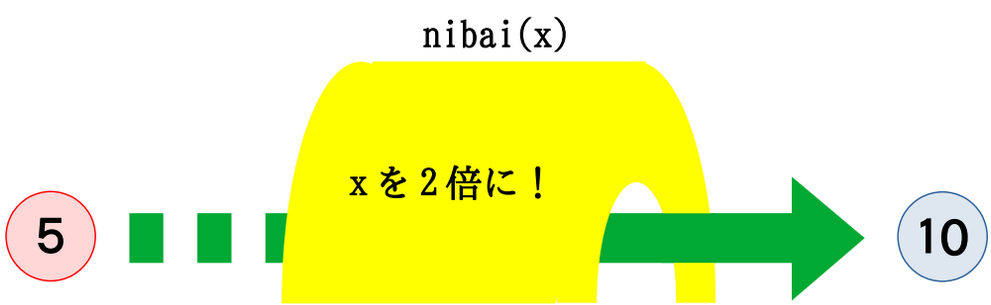
```
def nibai(x)
  x*2
end
```

と書きます。

こうしておいて y=nibai(5)とすると、5×2 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**(かりひきすう)といい、そのメソッドがもら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 nibai(5)のように 1 つの引数(ひきすう)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**(じつひきすう)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。

5.6.2. 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(ひきすう)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
```

5.6.3. 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！

5.6.4. return（リターン）

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))
```

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

5.6.5. 乱数の利用

janken.rb

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

def janken()                            #引数がいらない場合()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
```

まとめ

- メソッドについて理解できましたか？
- これまでの学習を振り返り、しっかりと身につけましょう。

発展

- 今まで習ったものを使って、自由にプログラミングしてみよう！

6. プログラムをもっと楽しくしてみよう！

6.1. 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、**rand**(ランド)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。

6.1.1. 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、**srand**(エスランド)()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
```

6.1.2. 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coin.sampleで書けます。

6.2. 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**sleep**(スリープ)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!!”
```

6.3. 時刻

Time.now(タイム　ナウ)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
```

6.4. 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**spawn**(スポーン)と**system**(システム)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。

6.4.1. 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**Process.kill**(プロセス　キル)に渡して終了させます。

○画像表示

spawn ”display -geometry +*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
```

6.4.2. 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 *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
```

○アスキーアート

system ”banner *moji*”で、簡単にアスキーアートを作ることができます。日本語に非対応のため、使用する文字は半角の英数字か記号にしましょう。

banner.rb

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

system ”banner a!”        #「a!」の部分が#で表現
```

7. おまけ

7.1. Emacs（イーマックス）

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。

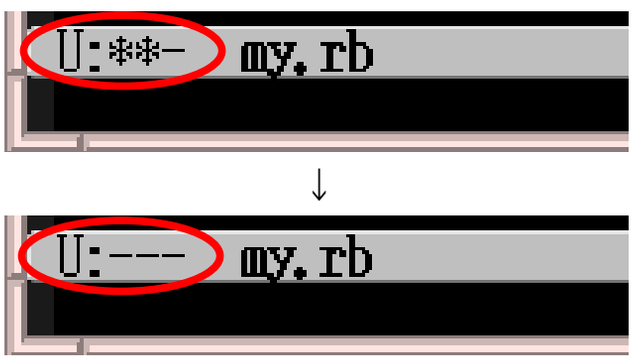
例）C-f は Ctrl キーを押しながら、f のキーを押す。

○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

7.2. irb（アイアールビー）

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。

7.3. コマンド集

Emacs(イーマックス)へ移動 C-1	terminal(ターミナル)へ移動 C-2	console(コンソール)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。

7.4. 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

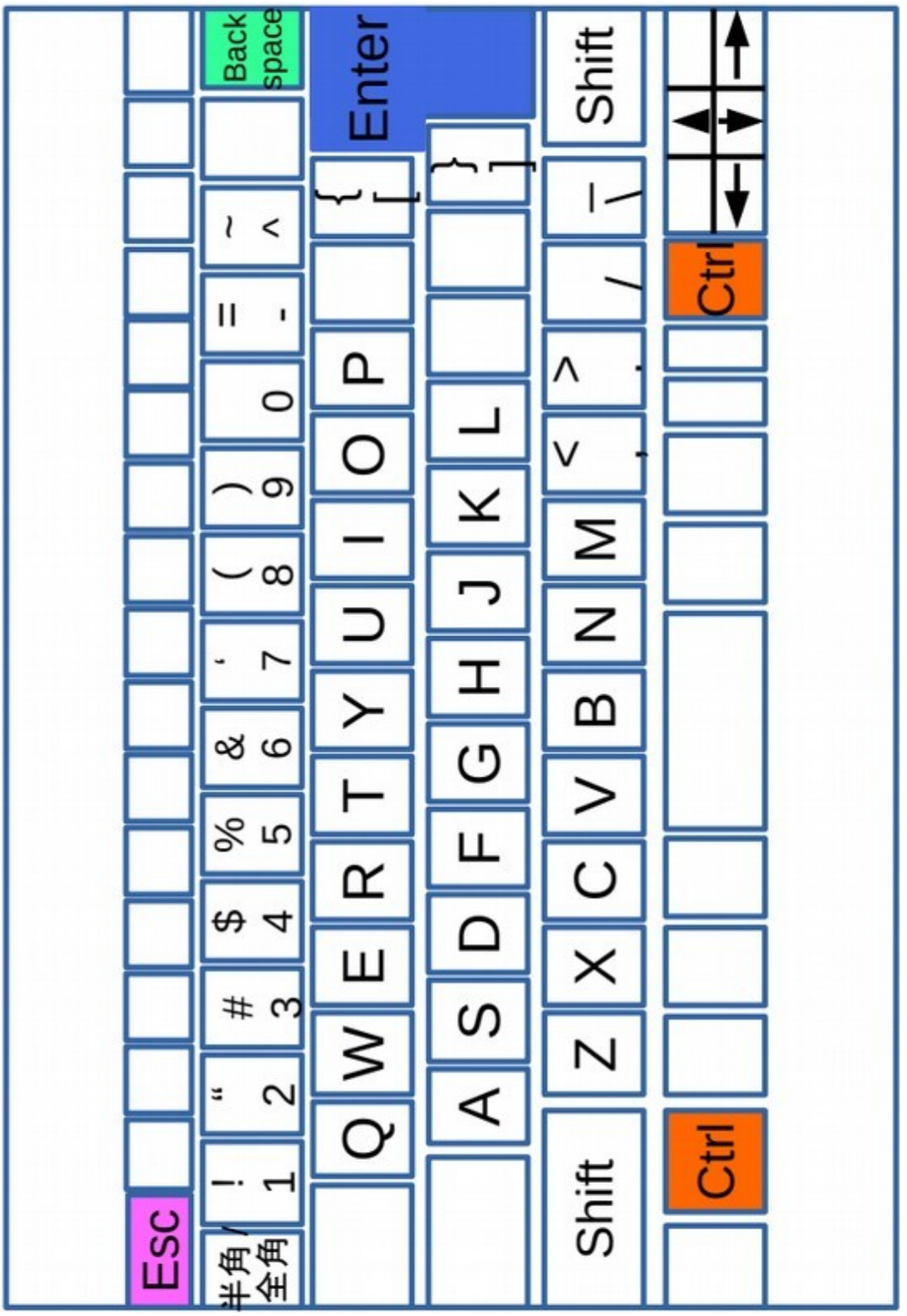
```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

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した！”
```

8. 参考資料

8.1. アルファベット表

A a	B b	C c .chomp, case	D d
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		

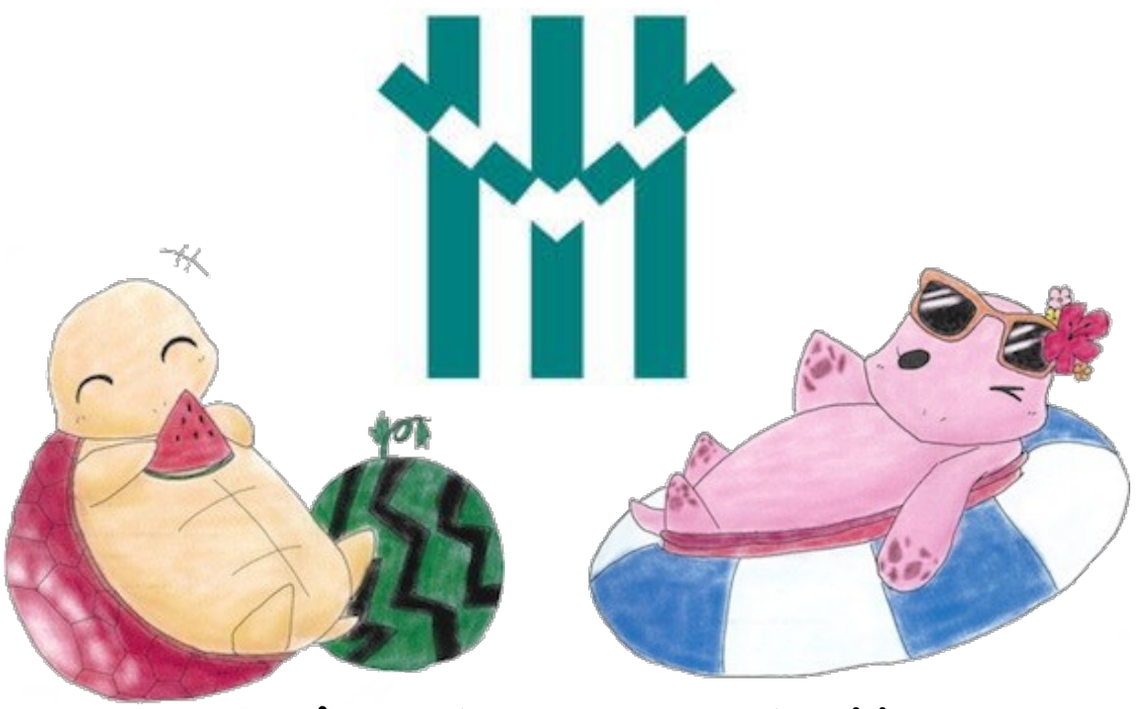
8.2. キーボード表


Esc
半角/全角
! 1
" 2
3
$ 4
% 5
& 6
' 7
(8
) 9
0
= -
~ ^
Back space
Q W E R T Y U I O P
{ [
}]
Enter
A S D F G H J K L
Shift
Z X C V B N M
< ,
> .
? /
_ \
Shift
Ctrl
Ctrl

ジュニアドクター鳥海塾 2021 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

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/



Let’s enjoy programming!!

Name