プログラミング言語　第八回

担当：篠沢 佳久
櫻井　彰人

平成20年 6月16日

本日の内容

- 配列
  - 宣言
  - 代入
  - 要素の参照方法
  - コマンドライン引数（特殊な配列）
- 練習問題①～⑥

配列とは

配列とは
配列の宣言

配列の必要性

- x1=5, x2=4, x3=3, ・・・, x100=10
  100個の変数の合計値を求めたい
- 乱数を1000個生成し，変数に格納し，処理したい

「配列」を利用

今回は配列

- 配列とは、普通、一次元の表、二次元の表、三次元の表、、、、、のこと
- Ruby の場合は、ちょっと、違う
- 「列」だと思ってください。
  - 値の列、場所の列

博識の方へ:
Ruby の配列は、CやJavaの配列とは大きく異なります。
（Lisp を源流とする）リスト構造（の発展型）と考えてよい。
身近なところでは、Mathematica のリストとそっくりです

こんな具合です。

定数:
　["Perl", "Python", "Ruby", "Scheme"]
変数への代入:
　names = ["Perl", "Python", "Ruby", "Scheme"]
印字: (print は目的(?)にあいません)
　p ["Perl", "Python", "Ruby", "Scheme"]

```
irb(main):016:0> p ["Perl", "Python", "Ruby", "Scheme"]
["Perl", "Python", "Ruby", "Scheme"]
=> nil
irb(main):017:0> puts ["Perl", "Python", "Ruby", "Scheme"]
Perl
Python
Ruby
Scheme
=> nil
irb(main):018:0> print ["Perl", "Python", "Ruby", "Scheme"]
PerlPythonRubyScheme=> nil
irb(main):019:0>
```

配列の要素①

names = ["Perl", "Python", "Ruby", "Scheme"]

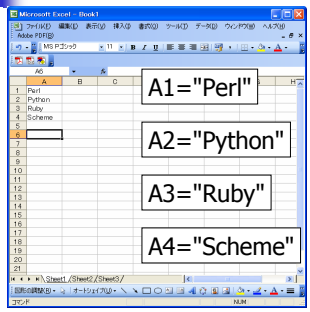
イメージ的には表計算のセル

要素番号は0から始まる

要素番号（インデックス）	配列名 names
0	"Perl"
1	"Python"
2	"Ruby"
3	"Scheme"

names[0]
names[1]
names[2]
names[3]

（参考）表計算ソフトのセル



配列の要素②

```
irb(main):004:0> names = ["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):005:0> p names[ 0 ]
"Perl"
=> nil
irb(main):006:0> p names[ 1 ]
"Python"
=> nil
irb(main):007:0> p names[ 2 ]
"Ruby"
=> nil
irb(main):008:0> p names[ 3 ]
"Scheme"
=> nil
irb(main):009:0> p names[ 4 ]
nil
=> nil
```

names[4]には値が代入されていない
→nil となる

配列の宣言

- names = ["Perl", "Python", "Ruby", "Scheme"]
- a = [0 , 2 , 4 , 6 , 8]
- 配列名 = [値1 , 値2 , ・・・ , 値n]

	a
0	0
1	2
2	4
3	6
4	8

配列の要素③

この配列のa.lengthの値は5

- a = [0 , 2 , 4 , 6 , 8]
- 配列名[要素番号]
- 配列の要素数
- 配列名.length

	a	
0	0	a[0]
1	2	a[1]
2	4	a[2]
3	6	a[3]
4	8	a[4]

```
irb(main):001:0> names = ["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):002:0> names.length
=> 4
```


配列の要素への代入①

配列名[要素番号] = 値

```
names = ["Perl", "Python", "Ruby", "Scheme"]
names[ 0 ] = "C"
names[ 3 ] = "Java"
```

要素番号（インデックス）	配列名 names	
0	"Perl"	"C"に置き換わる
1	"Python"	
2	"Ruby"	
3	"Scheme"	"Java"に置き換わる

配列の要素への代入②

```
irb(main):014:0> names = ["Perl", "Python", "Ruby",
   "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):015:0> names[ 0 ] = "C"
=> "C"
irb(main):016:0> names[ 3 ] = "Java"
=> "Java"
irb(main):017:0> p names
["C", "Python", "Ruby", "Java"]
=> nil
```


試してみよう

```
irb(main):001:0> a = [ 1,2,3,4,5]
=> [1, 2, 3, 4, 5]
irb(main):002:0> p( a )
[1, 2, 3, 4, 5]
=> nil
irb(main):003:0> print( a )
12345=> nil
irb(main):004:0> puts( a )
1
2
3
4
5
=> nil
irb(main):005:0> a[0]
=> 1
irb(main):006:0> a[1]
=> 2
```

```
irb(main):007:0> a.length
=> 5
irb(main):008:0> a[a.length-1]
=> 5
irb(main):009:0> a[a.length]
=> nil
irb(main):010:0> a[a.length+1]
=> nil
irb(main):011:0> a.length
=> 5
irb(main):012:0> a[7] = 77
=> 77
irb(main):013:0> p( a )
[1, 2, 3, 4, 5, nil, nil, 77]
=> nil
```

注目

Ruby の配列は柔軟

- すでに存在する要素に代入できる
  - これは当たり前
- まだ「ない要素」に代入すると、配列を拡張！して（当該要素を作って）くれる

注目!

```
irb(main):006:0> abc = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):007:0> abc[3] = "d"
=> "d"
irb(main):008:0> abc
=> ["a", "b", "c", "d"]
irb(main):009:0> abc[10] = "k"
=> "k"
irb(main):010:0> abc
=> ["a", "b", "c", "d", nil, nil, nil, nil, nil, "k"]
irb(main):011:0>
```


各要素に代入する（失敗編）

注目!

```
irb(main):062:0> primes[0]=2
NameError: undefined local variable or method `primes' for main:Object
        from (irb):62
        from :0
irb(main):063:0>
```

あれ？

いろいろややこしい事情があるのです。
Ruby では（Rubyに限らずどの言語でも）、未定義の変数が使われるとエラー
　Ruby では「使う」以外に現れると、「これから使うぞ！」という宣言と考える
Ruby では、左辺に現れる以外は、「使う」ことに相当
　従って、新しい名前を左辺に書くと、普通は、「これから使うぞ！」という宣言になる。
　（だから問題は発生しない）
　しかし、配列の要素として現れる（ primes[0])と「使う」ことになってしまう
　（ないものをいきなり使うことはできない。使おうとすればエラー！）

配列だということを教える①

- もちろん、教える相手は Ruby

配列名=Array.new(要素数)
要素数分の配列を用意する

```
abc = Array.new(5)
```

注目!

```
irb(main):073:0> abc = Array.new( 5 )
=> [nil, nil, nil, nil, nil]
irb(main):074:0>
```

注目!

```
irb(main):075:0> abc[3]=333
=> 333
irb(main):076:0> abc
=> [nil, nil, nil, 333, nil]
irb(main):077:0>
```


配列だということを教える②

```
abc = Array.new(5)
```

	abc
0	
1	
2	
3	
4	

配列名abc
要素数5個を用意する

abc[0]からabc[4]まで
値は入っていない（nil）

配列だということを教える③

配列名=[]

x=[]
x[0] = 3
x[1] = 5

xが配列であることを宣言

```
irb(main):008:0> x=[]
=> []
irb(main):009:0> x[ 0 ] = 3
=> 3
irb(main):010:0> x[ 1 ] = 5
=> 5
irb(main):011:0> p x
[3, 5]
=> nil
```

注目!

配列の宣言のまとめ

- 要素が分かっている場合
  - 配列名 = [値1, 値2, … , 値n]
- 要素数のみが決まっている場合
  - 配列名 = Array.new(要素数)
- 要素数が決まっていない場合
  - 配列名 = []

試してみよう 2

```
irb(main):001:0> a[3]
NameError: undefined local variable or method `a' for main:Object
        from (irb):1
        from :0
irb(main):002:0> x = a[2]
NameError: undefined local variable or method `a' for main:Object
        from (irb):2
        from :0
irb(main):003:0> a = Array.new(5)
=> [nil, nil, nil, nil, nil]
irb(main):004:0> a[10]
=> nil
irb(main):005:0> a
=> [nil, nil, nil, nil, nil]
```

配列名はa
要素数は5個と宣言

配列と繰り返し

配列の要素の参照方法

配列の要素の参照方法①

names = ["Perl", "Python", "Ruby", "Scheme"]

	names	
0	"Perl"	names[0]
1	"Python"	names[1]
2	"Ruby"	names[2]
3	"Scheme"	names[3]

names[0]～names[3]まで順番に参照するには？

繰り返しを利用する

配列の要素の参照方法②

- 配列の要素の参照方法には

1. 要素番号を用いて要素の値を取り出す方法
2. 要素の値を直接取り出す方法

- があります

要素番号を用いて一つずつ取り出す

注目!

```
names=["Perl", "Python", "Ruby", "Scheme"]
4.times { | i | print( "#{i} 番目は #{ names[i] }¥n" ) }
```

```
irb(main):033:0> names=["Perl", "Python", "Ruby",
"Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):034:0> 4.times { | i | print( "#{i} 番目は
#{names[i]}¥n" ) }
0 番目は Perl
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
irb(main):035:0>
```

i には0,1,2,3と代入される

names[0], names[1], names[2],names[3]と参照される

要素番号を用いて一つずつ取り出す(続)

```
names=["Perl", "Python", "Ruby", "Scheme"]
names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n" ) }
```

注目!

```
irb(main):035:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):036:0> names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n"
) }
0 番目は Perl
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
```

names.length=4

要素番号を用いて一つずつ取り出す(続々)

```
a=[1,3,5,7,9]
(0..a.length-1).each{ |i|
  print( a[ i ] , "¥n" )
}
```

i には0,1,2,3,4が代入される

a[0],a[1],a[2],a[3],a[4]と参照される

```
a=[1,3,5,7,9]
(2..4).each{ |i|
  print( a[ i ] , "¥n" )
}
```

i には2,3,4が代入される

a[2],a[3],a[4]と参照される

要素番号を用いて一つずつ取り出す

	a	
0	0	a[0]
1	2	a[1]
2	4	a[2]
3	6	a[3]
4	8	a[4]

この順番に要素を取り出したい

```
配列の長さ.times{ |i|
      配列[ i ]の処理
}
```

```
a.length.times{ |i|
      print( a[ i ] , "¥n" )
}
```

i は0,1,2,3,4 と代入されるため a[0], a[1], a[2], a[3], a[4] となる

要素を直接一つずつ取り出す

```
["Perl", "Python", "Ruby", "Scheme"].each { | lang |
   print( "I like ", lang, "¥n" )
}
```

注目!

```
irb(main):028:0> ["Perl", "Python", "Ruby", "Scheme"].each
irb(main):029:1* { | lang | print( "I like ", lang, "¥n" )
irb(main):030:1> }
I like Perl
I like Python
I like Ruby
I like Scheme
=> ["Perl", "Python", "Ruby", "Scheme"]
```

lang に "Perl" , "Python" , "Ruby" , "Scheme" と代入される

要素を直接一つずつ取り出す

```
[値1,値2,…,値n].each{ |i|
      print( i , "¥n" )
}
```

i に値1,値2,...値nが代入される

```
[1,3,5,7,9].each{ |i|
  print( i , "¥n" )
}
```

i に1,3,5,7,9が代入される

要素を直接一つずつ取り出す（続）

前ページと同じ

```
names=["Perl", "Python", "Ruby", "Scheme"]
names.each { | lang | print( "I like #{lang}¥n" ) }
```

```
irb(main):030:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):031:0> names.each { | lang | print( "I like #{lang}¥n" ) }
I like Perl
I like Python
I like Ruby
I like Scheme
=> ["Perl", "Python", "Ruby", "Scheme"]
```


要素を直接一つずつ取り出す（続）

```
配列名=[値1,値2,…,値n]
配列名.each{ |i|
   print( i , "¥n" )
}
```

i に値1,値2,…値nが代入される

```
a=[1,3,5,7,9]
a.each{ |i|
  print( i , "¥n" )
}
```

i に1,3,5,7,9が代入される

まとめ①

- 要素番号で要素の値を参照したい場合

```
a=[1,3,5,7,9]
```

```
a.length.times{ |i|
      print( a[ i ] , "¥n" )
}
```

```
(0..a.length-1).each{ |i|
   print( a[ i ] , "¥n" )
}
```


まとめ②

- 要素を直接参照したい場合

```
a=[1,3,5,7,9]
a.each{ |i|
  print( i , "¥n" )
}
```

```
[1,3,5,7,9].each{ |i|
   print( i , "¥n" )
}
```


試してみよう 3

```
irb(main):001:0> a = [11,12,13,14,15]
=> [11, 12, 13, 14, 15]
irb(main):002:0> a.length
=> 5
irb(main):003:0> a.each{ |x| print( "#{x} " ) }
11 12 13 14 15 => [11, 12, 13, 14, 15]
irb(main):004:0> (0..a.length-1).each{ |i| puts( "#{i}:
#{a[i]} " ) }
0: 11
1: 12
2: 13
3: 14
4: 15
⇒0..4
```

a.length-1 であることに注意

試してみよう 3

```
irb(main):005:0> (3..a.length+1).each{ |i| puts( "#{i}:
#{a[i]} " ) }
3: 14
4: 15
5:
6:
=> 3..6
irb(main):006:0> n=88; [2,3,5,7,11].each{ |p| puts(
"#{n} is divisible by #{p}") if n%p==0 }
88 is divisible by 2
88 is divisible by 11
=> [2, 3, 5, 7, 11]
```

a[a.length]

a[a.length+1]

a.length 以上の要素は存在しない

試してみよう 4

注目!

```
names=["Perl", "Python", "Ruby", "Scheme"]
names[0] = "Ada"
names.length.times { | i |
      print( "#{i} 番目は #{ names[i] }¥n" )
}
```

```
irb(main):059:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):060:0> names[0] = "Ada"
=> "Ada"
irb(main):061:0> names.length.times { | i |
print( "#{i} 番目は #{ names[i] }¥n"
 ) }
0 番目は Ada
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
```

試してみよう 5

```
irb(main):001:0> a = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):002:0> a[0] = "x"
=> "x"
irb(main):003:0> a
=> ["x", "b", "c"]
irb(main):004:0> a[7]="h"
=> "h"
irb(main):005:0> a
=> ["x", "b", "c", nil, nil, nil, nil, "h"]
```

38

試してみよう 5

	a
0	"x"
1	"b"
2	"c"
3	nil
4	nil
5	nil
6	nil
7	"h"

a=["x" , "b" , "c"]

a[7] ="h"

a[3] から a[6] は nil（値がない）

39

試してみよう 5

```
irb(main):006:0> a.each{ |x| x="0" }
=> ["x", "b", "c", nil, nil, nil, nil, "h"]
irb(main):007:0> a
=> ["x", "b", "c", nil, nil, nil, nil, "h"]
irb(main):008:0> (0..a.length-1).each{ |i| a[i]="0" }
=> 0..7
irb(main):009:0> a
=> ["0", "0", "0", "0", "0", "0", "0", "0"]
```

40

試してみよう 5

```
a = [ 1 , 2 , 3 , 4 , 5 ]
a.each{ |x|
   print( " 代入前 " , x , "¥n" )
   x="0"
   print( " 代入後 " , x , "¥n" )
}
p a
```

x にはa[0]～a[4]の値が代入されるだけで配列の要素を直接変更するわけではない

```
C:¥ruby>ruby sample.rb
 代入前 1
 代入後 0
 代入前 2
 代入後 0
 代入前 3
 代入後 0
 代入前 4
 代入後 0
 代入前 5
 代入後 0
[1, 2, 3, 4, 5]
```

41

試してみよう 5

```
a = [ 2 , 3 , 5 , 8 , 4 ]
(0..a.length-1).each{ |i|
   print( "代入前" , a[ i ] , "¥n"    )
   a[i]="0"
   print( "代入前" , a[ i ] , "¥n"    )
}
p a
```

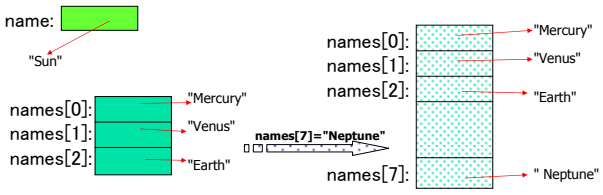
配列の要素を直接変更している

```
C:¥ruby>ruby sample.rb
代入前2
代入前0
代入前3
代入前0
代入前5
代入前0
代入前8
代入前0
代入前4
代入前0
["0", "0", "0", "0", "0"]
```

42

変数と配列の裏事情

ユーザ（皆さんです）が、変数や配列を使いたいというと、
コンピュータは「場所」を確保する必要あり



配列の大きさが変わると、Ruby は配列を作り直している。
古い場所はそのまま捨て置かれる。
すなわち、無駄が発生している。ユーザには見えないが。

配列要素には何が代入できるか

- 変数に代入できるものなら何でも代入できる
- 整数、浮動小数点数、文字列、配列！
- しかも、混在！できる
  - CやJavaでは「混在」はできない

```
irb(main):011:0> abc = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):012:0> abc[1] = 111
=> 111
irb(main):013:0> abc
=> ["a", 111, "c"]
irb(main):014:0> abc[3] = 3.33
=> 3.33
irb(main):015:0> abc
=> ["a", 111, "c", 3.33]
irb(main):016:0> abc[4] = [4,5,6]
=> [4, 5, 6]
irb(main):017:0> abc
=> ["a", 111, "c", 3.33, [4, 5, 6]]
```

注目!

試してみよう 6

前のスライドと同じです

```
irb(main):011:0> abc = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):012:0> abc[1] = 111
=> 111
irb(main):013:0> abc
=> ["a", 111, "c"]
irb(main):014:0> abc[3] = 3.33
=> 3.33
irb(main):015:0> abc
=> ["a", 111, "c", 3.33]
irb(main):016:0> abc[4] = [4,5,6]
=> [4, 5, 6]
irb(main):017:0> abc
=> ["a", 111, "c", 3.33, [4, 5, 6]]
```


配列の要素の参照例

配列の要素の参照例①

配列の要素の合計を求める

```
a=[4,2,1,6,7]

sum = 0
a.length.times{ |i|
    sum += a[ i ]
}
print( " sum = " , sum )
```

```
a=[4,2,1,6,7]

sum = 0
a.each{ |i|
        sum += i
}
print( " sum = " , sum , "¥n" )
```


配列の要素の参照例②

配列の要素の合計を求める

```
a=[4,2,1,6,7]

i = 0
sum = 0
while i < a.length do
   sum += a[ i ]
   i += 1
end
print( " sum = " , sum )
```

```
a=[4,2,1,6,7]

i = 0
sum = 0
loop{
        sum += a[ i ]
        i += 1
        break if i == a.length
}
print( " sum = " , sum )
```

while , loop を用いても同じ動作ができます

配列の要素の参照例③

どのように出力が異なるのでしょうか

```
a=[4,2,1,6,7]

a.length.times { |i|
    if i % 2 != 0 then
        print( a[ i ] , "¥n" )
    end
}
```

```
a=[4,2,1,6,7]

a.each { |i|
    if i % 2 != 0 then
        print( i , "¥n" )
    end
}
```


配列の要素の参照例④

```
x=[]
10.times{ |i|
        x[ i ] = rand( 100 )
}

sum = 0
x.length.times{ |i|
        sum += x[ i ]
}
p x
print( " sum = " , sum )
```

配列xを宣言

10個の乱数を生成
配列xに格納

合計値の計算

配列の要素の参照例⑤

配列の最後の要素から出力

```
a=[4,2,1,6,7]

a.length.times { |i|
    print( a[ a.length-1-i ], "¥n" )
}
```

a[4],a[3],a[2],a[1],a[0]
の順に出力される

```
a=[4,2,1,6,7]

(a.length-1).step(0,-1){ |i|
        print( a[ i ] , "¥n" )
}
```

step
iは4,3,2,1,0と代入される

配列の要素の参照例⑥

配列のコピー

```
a=[4,2,1,6,7]
x=[]

a.length.times { |i|
    x[ i ] =  a[ i ]
}
p x
```

配列xを宣言

x[i]にa[i]を代入

```
a=[4,2,1,6,7]
x=[]

a.length.times { |i|
    x[ i ] =  a[ i ]*a[ i ]
}
p x
```

配列xを宣言

x[i]にa[i]*a[i]を代入

注意: 配列の要素の参照例⑥

「配列のコピー」にはなりません！

```
a=[4,2,1,6,7]
x=a
p a
p x
a[0]=10
p a
p x
```

配列xにコピー?

配列aだけ変更

しかし、x も変っている！

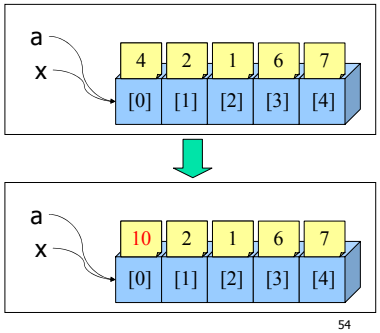
```
C:¥ruby>ruby sample.rb
[4, 2, 1, 6, 7]
[4, 2, 1, 6, 7]
[10, 2, 1, 6, 7]
[10, 2, 1, 6, 7]
```


注意': 配列の要素の参照例⑥

```
a=[4,2,1,6,7]
x=a
p a, x

a[0]=10

p a, x
```



配列の要素の参照例⑥'

前のページと同じです

```
a=[4,2,1,6,7]
x=Array.new(a.length)

a.length.times { |i|
    x[ i ] =  a[ i ]
}
p x
```

x[i]にa[i]を代入

```
a=[4,2,1,6,7]
x=Array.new(a.length)

a.length.times { |i|
    x[ i ] =  a[ i ]*a[ i ]
}
p x
```

x[i]にa[i]*a[i]を代入

配列xの要素数が分かっている場合

配列の要素の参照例⑦

配列の最後の要素に値を追加

```
a=[4,2,1,6,7]

a[ a.length ] = 8
p a
```

a[0]	a[1]	a[2]	a[3]	a[4]
4	2	1	6	7

↓

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
4	2	1	6	7	8

配列の要素の参照例⑧

配列の先頭の要素に値を追加

```
a=[4,2,1,6,7]

n=a.length
n.times{ |i|
    a[ n-i ] = a[ n-i-1 ]
}

a[ 0 ] = 5
p a
```

a[0]	a[1]	a[2]	a[3]	a[4]
4	2	1	6	7

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
4	4	2	1	6	7

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
5	4	2	1	6	7

配列の要素の参照例⑨

どういうプログラムでしょう？

```
e=[]

n = 20
fact = 1.0
e[ 0 ] = 1.0
(1..n).each{ |x|
    fact = fact * x.to_f
    e[ x ] = e[ x-1 ] + 1.0 / fact
}

e.length.times{ |i|
    printf( "%3d: %64.60f¥n" , i , e[ i ] )
}
```


配列の要素の参照例⑩

実行結果

```
C:¥ruby>ruby sample.rb
3 ***
3 ***
5 *****
2 **
3 ***
4 ****
2 **
1 *
7 *******
5 *****
```

```
a=[]

n=10
n.times{ |i|
    a[ i ] = rand( 10 )
}

a.length.times { |i|
    print( a[ i ] , " " , "*" * a[ i ] , "¥n" )
}
```

10個の乱数を生成

＊を a[i]個出力

コマンドライン引数

コマンドライン引数①

- Rubyプログラムにデータを渡すことができます
  - データを整数に限ると次のようにできます

実行例と実行結果例

```
H:¥ruby>ruby sample0302 2 3
 2 + 3 = 5
H:¥ruby>ruby Sample0302 8 2
 8 + 2 = 10
```

引数

コマンドライン引数②

- ruby プログラム 値1 値2 値3
  - 値1, 値2, ・・・ を引数と呼ぶ
- 値1は ARGV[0] に格納される
- 値2は ARGV[1] に格納される
- 値3は ARGV[2]に格納される
- ARGV[0], ARGV[1], ARGV[2] は文字列型配列

コマンドライン引数③

- ruby プログラム 値1 値2 値3 … 値n
- 値1は ARGV[0] に格納される
- 値2は ARGV[1] に格納される
- 値nは ARGV[n]に格納される

コマンドライン引数:プログラム

プログラム例 (sample0302.rb)

```
print( ARGV[0] + " + " + ARGV[1]+ " = " )
print( ARGV[0].to_i + ARGV[1].to_i )
```

または

```
print( "#{ARGV[0]} + #{ARGV[1]} = " )
print( ARGV[0].to_i + ARGV[1].to_i )
```

文字列型のため型変換(整数)が必要

実行例と実行結果例

```
G:¥Ruby>ruby sample0302.rb 3 5
3 + 5 = 8
G:¥Ruby>ruby sample0302.rb 100 200
100 + 200 = 300
```


浮動小数点数にすると

プログラム例 (sample0303.rb)

```
print( ARGV[0] + " + " + ARGV[1]+ " = " )
print( ARGV[0].to_f + ARGV[1].to_f )
```

または

```
print( "#{ARGV[0]} + #{ARGV[1]} = " )
print( ARGV[0].to_f + ARGV[1].to_f )
```

文字列型のため型変換(小数)が必要

実行例と実行結果例

```
G:¥Ruby>ruby sample0303.rb 3 5
3 + 5 = 8.0
G:¥Ruby>ruby sample0303.rb 3.0 5.0
3.0 + 5.0 = 8.0
G:¥Ruby>ruby sample0303.rb 100 200.0
100 + 200.0 = 300.0
```


3個にすると

プログラム例 (sample0304.rb)

```
print( ARGV[0] + " + " + ARGV[1] + " + " + ARGV[2] + " = " )
print( ARGV[0].to_f + ARGV[1].to_f + ARGV[2].to_f )
```

実行例と実行結果例

```
G:¥Ruby>ruby sample0304.rb 1.23 4.56 7.89
1.23 + 4.56 + 7.89 = 13.68
```

練習問題

配列に関する練習①～⑥

練習問題①

- 配列 a=[5,4,2,7,6] の要素の中で最小値，最大値を求めるプログラムを書きなさい

練習問題②

- 二つの配列a，bに対して a=[1,2,3,4,5]，b=[1,4,9,16,25] というようにtimes を用いて各要素に値を格納するプログラムを書きなさい．

練習問題③

- 二つの配列 a=[4,3,6,9,1] b=[1,9,5,2,3] をベクトルとした場合，二つのベクトルの和を配列 x に，二つの内積を変数 y に求めるプログラムを書きなさい

参考ですが…

```
irb(main):003:0> a=[4,3,6,9,1]
=> [4, 3, 6, 9, 1]
irb(main):004:0> b=[1,9,5,2,3]
=> [1, 9, 5, 2, 3]
irb(main):005:0> a+b
=> [4, 3, 6, 9, 1, 1, 9, 5, 2, 3]
```

「＋」演算子は二つの配列を結合

練習問題④

- キーボードから整数を入力し，順番に配列xに格納し，その結果を出力するプログラムを書きなさい．（キーボードからの入力はqを入力することで終了とする）

練習問題⑤

- 配列x=[3,4,9,6,2]の要素をtimesメソッドを用いて，逆順に並び変えるプログラムを作成しなさい
- 配列xの要素を直接入れ換えること

```
irb(main):001:0> x=[3,5,6,7]
=> [3, 5, 6, 7]
irb(main):002:0> x.reverse
=> [7, 6, 5, 3]
```

reverse というのがありますが使ってはいけません

補足: 入れ替え

- 変数値の入替え
  - 変数 a と変数 b に入っている値を入替えたい。どうすればいいか？
    - 当然、a=b; b=a ではだめです。どうして？

```
irb(main):007:0> a=1; b=10; puts( "a=#{a}, b=#{b}" )
a=1, b=10
=> nil
irb(main):008:0> a=b; b=a; puts( "a=#{a}, b=#{b}" )
a=10, b=10
=> nil
```


補足: 入れ替え その2

- 入替えには、作業領域があればよい

```
irb(main):009:0> a=1; b=10; puts( "a=#{a}, b=#{b}" )
a=1, b=10
=> nil
irb(main):010:0> w=a; a=b; b=w; puts( "a=#{a}, b=#{b}" )
a=10, b=1
=> nil
```

- 配列要素に対しても同様

```
irb(main):001:0> x=[ 3 , 5 ]
=> [3, 5]
irb(main):002:0> work=x[0]
=> 3
irb(main):003:0> x[0]=x[1]
=> 5
irb(main):004:0> x[1]=work
=> 3
irb(main):005:0> p x
[5, 3]
=> nil
```


練習問題⑥

- 配列 a 内に整数が記憶されているとする。この内容を、偶数と奇数にわけ、配列 b に入れよ。ただし、配列 b の前半に偶数、後半に奇数とせよ。ただし、
  - Q1: 順序に関しては、考慮しなくてもよい
    - 奇数の順序は逆になる、とすると問題は簡単になる。偶数は b の先頭から、奇数は、b の後ろから入れていけばよいから。
    - 次に入れるべき、b 内の位置を2個の変数で記憶すればよい
    - Ruby のマニュアルを調べて、push と unshift を用いると、偶数の順序は逆、奇数の順序は保存というプログラムが書ける
  - Q2: 偶数同士の順序、奇数同士の順序は保存せよ
    - いくつかの方法が考えられる
      - Q1-1 を行ってから、奇数の並ぶ順序を反転する
      - まずは a 内の偶数の個数を数える。b 内の奇数が来るべき先頭位置が分かる
      - まず、a 内の偶数だけ b に入れる。次に、奇数を入れていく
      - 配列 c を用意し、奇数はこちらに記憶していく。最後に b の後ろに c の要素を移す。

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
```

Q1の場合 `[4, 2, 6, 8, 2, 3, 3, 9, 7, 9, 5, 3, 5, 9, 5, 1, 1, 3]`

Q2の場合 `[4, 2, 6, 8, 2, 3, 1, 1, 5, 9, 5, 3, 5, 9, 7, 9, 3, 3]`

練習⑥ Q1のヒント

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
```

```
b = Array.new( a.length )
j = 0
k = a.length - 1
(0..a.length-1).each{ |i|
# ここで a の要素を b に移す
}
p( b )
```

```
b = Array.new( a.length )
j = 0
k = a.length - 1
a.length.times{ |i|
# ここで a の要素を b に移す
}
p( b )
```

出力結果

```
[4, 2, 6, 8, 2, 3, 3, 9, 7, 9, 5, 3, 5, 9, 5, 1, 1, 3]
```


練習⑥ Q2のヒント

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
```

```
a = [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3]
b = Array.new( a.length )
c = []

j = 0
a.length.times{ |i|
        # 偶数は配列bに，奇数は配列cに格納
}

c.length.times{ |i|
        # 配列bに配列cを追加
}
p( b )
```

出力結果

```
[4, 2, 6, 8, 2, 3, 1, 1, 5, 9, 5, 3, 5, 9, 7, 9, 3, 3]
```


練習問題

- 練習問題①から⑥を（できるだけ）（頑張って）行ないなさい。
- プログラムと実行結果をワープロに貼り付けて、keio.jp から提出して下さい。