プログラミング言語　第九回

担当：篠沢 佳久
櫻井 彰人

平成23年 6月 27日

本日の内容

- 一次元配列の復習
- 二次元配列
- 多重ループ
  - ネスト（入れ子）構造
- 練習問題①～③

ネスト（入れ子）

- ネストする: 入れ子にすること

箱根の十二卵（田中一幸氏作）　左端は実際の鶏卵のLL玉ほど。右端は13番目のヒヨコ

http://dadandmam.whitesnow.jp/sub3-011.htm

配列の復習

一次元配列と繰り返し

配列の宣言

- 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 = []

配列の宣言②

要素が分かっている場合

a=[4, 6, 7, 9, 10]

	a	
0	4	a[0]
1	6	a[1]
2	7	a[2]
3	9	a[3]
4	10	a[4]

配列の宣言②

要素数が分かっている場合	要素数が決まっていない場合
a= Array.new(5)	a= []

```
a= Array.new(5)

a[0]=4
a[1]=6
a[2]=7
a[3]=9
a[4]=10
```

```
a= []

a[0]=4
a[1]=6
a[2]=7
a[3]=9
a[4]=10
```


配列の要素の参照方法①

- 要素番号で要素の値を参照したい場合

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

配列名.length
配列の要素数

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

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

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

```
C:¥Ruby>ruby sample.rb
1
3
5
7
9
```


配列の要素の参照方法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" )
}
```

```
C:¥Ruby>ruby sample.rb
1
3
5
7
9
```


一次元配列のプログラム例

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5 , 100 , 50 ]
```

配列name　文字列型

	name
0	A
1	B
2	C
3	D
4	E

配列 test　整数型

	test
0	85
1	60
2	5
3	100
4	50

配列の要素への代入

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

name[ 3 ] = "d"
test[ 3 ] = 90

p name
p test
```

```
C:¥Ruby>ruby sample.rb
["A", "B", "C", "d", "E"]
[85, 60, 5, 90, 50]
```


最後の要素への追加①

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

name[ name.length ] = "F"
test[ test.length ] = 70

p name
p test
```

```
C:¥Ruby>ruby sample.rb
["A", "B", "C", "D", "E", "F"]
[85, 60, 5, 100, 50, 70]
```

最後の要素への追加②

name[name.length] = "F"

test[test.length] = 70

配列name 文字列型

	name
0	A
1	B
2	C
3	D
4	E
5	F

配列 test 整数型

	test
0	85
1	60
2	5
3	100
4	50
5	70

平均点を求める①

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
test.length.times{ |i|
   sum += test[ i ]
}
print( "平均点 --> " , sum / test.length )
```

times を用いた方法

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


平均点を求める②

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
(0..test.length-1).each{ |i|
   sum += test[ i ]
}
print( "平均点 --> " , sum / test.length )
```

each を用いた方法

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


平均点を求める③

```
name = [ "A" , "B" , "C" , "D" , "E" ]
test = [ 85 , 60 , 5, 100 , 50 ]

sum = 0
test.each{ |i|
   sum += i
}
print( "平均点 --> " , sum / test.length )
```

each を用いた方法
配列の要素を直接参照

前頁との違いに注意して下さい

```
C:¥ruby>ruby sample.rb
平均点 --> 60
```


ローカル変数①

```
10.times{
   a = 10
}
print( a , "¥n" )
```

ローカル変数
ブロック*の範囲内でしか利用できない

```
C:¥Ruby>ruby sample.rb
sample.rb:4: undefined local variable or method `a' for main:Object
(NameError)
```

*Rubyには別の意味のブロックもあります

ブロック①

後半にでてくる二重ループの場合

ブロック

```
test.length.times{ |i|

   if average > test[ i ] then
       print( name[ i ] , ": " , test[ i ] , "点¥n" )
   end

}
```

ブロック

ブロック②

ブロック

```
(0..9).each{ |x|
  (0..9).each{ |y|
      z = x*x + y*y
      print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
  }
}
```

ブロック

ローカル変数②

```
a = 0
10.times{
  a = 10
}
print( a , "¥n" )
```

ローカル変数
ブロックの前で宣言しておく

```
C:¥Ruby>ruby sample.rb
10
```


グローバル変数

```
10.times{
  $a = 10
}
print( $a , "¥n" )
```

グローバル変数
変数名の前に「$」をつける
プログラムのどこからでも参照できる

```
C:¥Ruby>ruby sample.rb
10
```


二次元配列

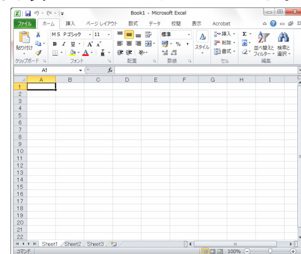
二次元配列の宣言
要素の参照，代入

2次元配列

- 表が使えると、随分便利です。
- スプレッドシートを思い起こしてください
  - スプレッドシートって何ですか？

二次元の配列＝二次元の表（行列）

- 表といえば、二次元かな。
- 表計算ソフトも2次元だしな。



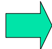
二次元配列の宣言①

3×3の行列 a

$$a=\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix}$$

表の場合

1	2	3
4	5	6
7	8	9



Ruby での宣言

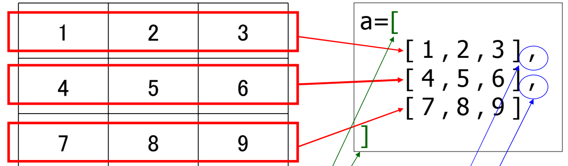
```
a=[
    [ 1 , 2 , 3 ] ,
    [ 4 , 5 , 6 ] ,
    [ 7 , 8 , 9 ]
]
```


二次元配列の宣言②

3×3の行列 a

Ruby での宣言

1	2	3
4	5	6
7	8	9



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


二次元配列の宣言③

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

p a
```

```
C:¥Ruby>ruby sample.rb
[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

一次元配列

二次元配列は一次元配列の要素を一次元配列としているとみなせる

二次元配列の宣言④

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

p a
```

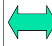
一次元配列

一次元配列

二次元配列は一次元配列の要素を一次元配列としているとみなせる

二次元配列の要素の参照①

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



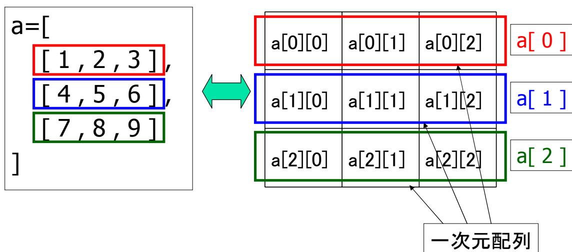
a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

二次元配列の要素の参照②

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

p a[ 0 ][ 0 ]
p a[ 0 ][ 1 ]
p a[ 0 ][ 2 ]

p a[ 1 ][ 0 ]
p a[ 1 ][ 1 ]
p a[ 1 ][ 2 ]

p a[ 2 ][ 0 ]
p a[ 2 ][ 1 ]
p a[ 2 ][ 2 ]
```

```
C:¥Ruby>ruby sample.rb
1
2
3
4
5
6
7
8
9
```

二次元配列の要素の参照③



二次元配列の要素の参照④

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

p a[ 0 ]
p a[ 1 ]
p a[ 2 ]
```

```
C:¥Ruby>ruby sample.rb
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

a[0] → [1, 2, 3]
a[1] → [4, 5, 6]
a[2] → [7, 8, 9]

二次元配列の要素の参照⑤

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

p a.length
p a[ 0 ].length
p a[ 1 ].length
p a[ 2 ].length
```

```
C:¥Ruby>ruby sample.rb
3
3
3
3
```

配列a の要素数

a[0], a[1], a[2] の要素数

二次元配列の要素の参照⑤'

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

p a.length
p a[ 0 ].length
p a[ 1 ].length
p a[ 2 ].length
```

```
C:¥Ruby>ruby sample.rb
3
1
2
3
```

配列a の要素数

a[0], a[1], a[2] の要素数

二次元配列の要素の参照⑤''

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

p a[ 0 ][ 0 ]
p a[ 0 ][ 1 ]
p a[ 0 ][ 2 ]

p a[ 1 ][ 0 ]
p a[ 1 ][ 1 ]
p a[ 1 ][ 2 ]

p a[ 2 ][ 0 ]
p a[ 2 ][ 1 ]
p a[ 2 ][ 2 ]
```

```
C:¥Ruby>ruby sample.rb
1
nil
nil
4
5
nil
7
8
9
```


二次元配列の要素の参照⑤'''

```
a=[
  [ 1 , 2 , 3 , 4],
  [ 5 , 6 , 7 ],
  [ 8 , 9 ],
  [ 10 ]
]

p a.length
p a[ 0 ].length
p a[ 1 ].length
p a[ 2 ].length
p a[ 3 ].length
```

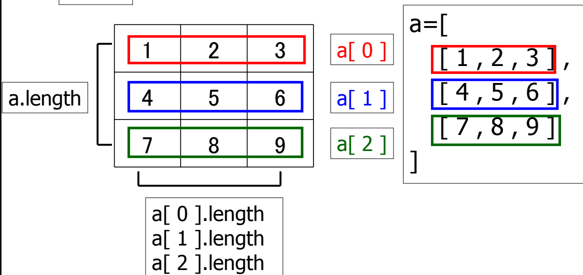
```
C:¥Ruby>ruby sample.rb
4
4
3
2
1
```

配列a の要素数

a[0], a[1], a[2], a[3] の要素数

二次元配列の要素の参照⑥

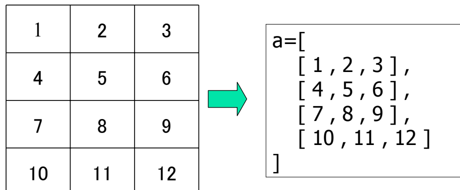
要素数



a[0].length
a[1].length
a[2].length

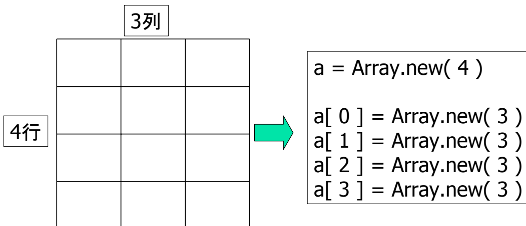
二次元配列の宣言①

- 要素の値が分かっている場合



二次元配列の宣言②

- 要素数のみ分かっている場合



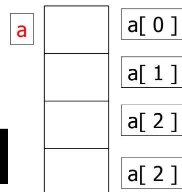

```
a = Array.new( 4 )

a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )
a[ 3 ] = Array.new( 3 )
```


二次元配列の宣言②'

要素数4の配列a を作成

```
a = Array.new( 4 )

p a
```

```
C:¥Ruby>ruby sample.rb
[nil, nil, nil, nil]
```

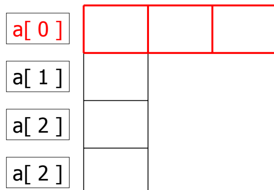


値が入っていないので「nil」となる

二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )

p a
```

配列a[0]に要素が3の配列を作成



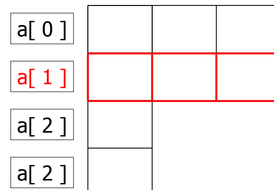

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], nil, nil, nil]
```

a[0] のみ3個の要素を持つ配列

二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )

p a
```

配列a[1]に要素が3の配列を作成



a[0] , a[1]
3個の要素を持つ配列

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], nil, nil]
```

二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )

p  a
```

配列a[2]に要素が3の配列を作成

a[0]
a[1]
a[2]
a[2]

a[0] , a[1] , a[2]
3個の要素を持つ配列

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], [nil, nil, nil], nil]
```


二次元配列の宣言②'

```
a = Array.new( 4 )
a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )
a[ 3 ] = Array.new( 3 )
p  a
```

配列a[3]に要素が3の配列を作成

a[0]
a[1]
a[2]
a[2]

```
C:¥Ruby>ruby sample.rb
[[nil, nil, nil], [nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]
```


二次元配列の要素への代入

```
a = Array.new( 4 )

a[ 0 ] = Array.new( 3 )
a[ 1 ] = Array.new( 3 )
a[ 2 ] = Array.new( 3 )
a[ 3 ] = Array.new( 3 )

a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```


二次元配列の宣言③

aは配列と宣言

```
a = []

a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] = []
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ] = []
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ] = []
a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

a[0], a[1], a[2]
, a[3]が配列である
ことを宣言

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```


二次元配列の宣言③'

```
a =[]

p  a
```

配列a を作成

```
C:¥Ruby>ruby sample.rb
[]
```

```
a =[]
a[ 0 ]=[]

p  a
```

配列a[0] を作成

```
C:¥Ruby>ruby sample.rb
[[]]
```


二次元配列の宣言③"

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

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3]]
```

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

a[ 1 ] =[]

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], []]
```

二次元配列の宣言③'''

```
a = []

a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] = []
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6]]
```


二次元配列の宣言③''''

```
a = []

a[ 0 ] = []
a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ] = []
a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ] = []
p a

a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

p a
```

```
C:¥Ruby>ruby sample.rb
[[1, 2, 3], [4, 5, 6], []]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```


二次元配列の宣言④

配列の宣言をしない場合

```
a = []

a[ 0 ][ 0 ] = 1
a[ 0 ][ 1 ] = 2
a[ 0 ][ 2 ] = 3

a[ 1 ][ 0 ] = 4
a[ 1 ][ 1 ] = 5
a[ 1 ][ 2 ] = 6
```

```
a[ 2 ][ 0 ] = 7
a[ 2 ][ 1 ] = 8
a[ 2 ][ 2 ] = 9

a[ 3 ][ 0 ] = 10
a[ 3 ][ 1 ] = 11
a[ 3 ][ 2 ] = 12

p a
```

a[0], a[1], a[2]が配列だと宣言していない

```
C:¥Ruby>ruby sample.rb
sample.rb:3: undefined method `[]=' for nil:NilClass (NoMethodError)
```


二次元配列の宣言のまとめ

- 要素の値が分かっている場合
- 要素数のみ分かっている場合
- 要素の値，要素数も分からない場合

二次元配列のまとめ①

- 一次元配列の一次元配列
  - Ruby では、子供の一次元配列の長さは異なってよい。Java でも同様。Cではダメ。

points[0]　points[1]　points[2]　points[3]

```
irb(main):001:0> points = [[70,60,83],[43,49,76],
irb(main):002:1*           [59,79,43],[67,74,83]]
=> [[70, 60, 83], [43, 49, 76], [59, 79, 43], [67, 74, 83]]
```

points[1][0]　points[1][1]　points[1][2]

points.length は 4, points[0].length は 3

二次元配列のまとめ②

- 一次元配列の一次元配列だから

```
irb(main):001:0> points = [[1, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
=> [[1, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
irb(main):002:0> points[0]
=> [1, 70, 60, 83]
irb(main):003:0> points[0][0] = 999
=> 999
irb(main):004:0> points
=> [[999, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
```

```
irb(main):004:0> points
=> [[999, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
irb(main):005:0> p = points[0]
=> [999, 70, 60, 83]
irb(main):006:0> p[0] = 99
=> 99
irb(main):007:0> p
=> [99, 70, 60, 83]
irb(main):008:0> points
=> [[99, 70, 60, 83], [2, 43, 49, 76], [3, 59, 79, 43], [4, 67, 74, 83]]
```

p と points[0] が同じものとなる

注意!　p の要素を変えたら points[0] の要素が変わった！

二重ループ

一重ループ

$y = x^2$

```
10.times{ |x|
        y = x*x
        print( x , ": " , y , "¥n" )
}
```

```
(0..9).each{ |x|
        y = x*x
        print( x , ": " , y , "¥n" )
}
```

```
C:¥ruby>ruby sample.rb
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
```


二重ループの必要性①

$z = x^2 + y^2$

$0 \le x < 10, 0 \le y < 10$ の範囲で値を求めるには？

x=0 の時，yの値を0から9まで変えて z を求める

```
x = 0
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```


x=1 の時，yの値を0から9まで変えて z を求める

```
x = 1
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

以下同様にx=9 まで同じことを繰り返し z を求める

```
x = 9
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )

}
```


二重ループの必要性②

```
x = 0
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

```
x = 1
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```

xの値も0から9まで一つずつ増やしていけばよい

```
x = 9
(0..9).each{ |y|
        z = x*x + y*y
        print( x , " " , y , ": " , z , "¥n" )
}
```


二重ループ

①のループによって，xは0から9まで変わる

①のループ

②のループ

```
(0..9).each{ |x|
    (0..9).each{ |y|
        z = x*x + y*y
        print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
    }
}
```

②のループによって，yは0から9まで変わる

二重ループの出力結果①

①のループ中 x=0 として
②のループの処理を行なう

```
C:¥ruby>ruby sample.rb
 x = 0 y = 0: z= 0
 x = 0 y = 1: z= 1
 x = 0 y = 2: z= 4
 x = 0 y = 3: z= 9
 x = 0 y = 4: z= 16
 x = 0 y = 5: z= 25
 x = 0 y = 6: z= 36
 x = 0 y = 7: z= 49
 x = 0 y = 8: z= 64
 x = 0 y = 9: z= 81
```

①のループ中 x=1 として
②のループの処理を行なう

```
x = 1 y = 0: z= 1
x = 1 y = 1: z= 2
x = 1 y = 2: z= 5
x = 1 y = 3: z= 10
x = 1 y = 4: z= 17
x = 1 y = 5: z= 26
x = 1 y = 6: z= 37
x = 1 y = 7: z= 50
x = 1 y = 8: z= 65
x = 1 y = 9: z= 82
```


二重ループの出力結果②

①のループ中 x=9 として
②のループの処理を行ない終了する

```
x = 9 y = 0: z= 81
x = 9 y = 1: z= 82
x = 9 y = 2: z= 85
x = 9 y = 3: z= 90
x = 9 y = 4: z= 97
x = 9 y = 5: z= 106
x = 9 y = 6: z= 117
x = 9 y = 7: z= 130
x = 9 y = 8: z= 145
x = 9 y = 9: z= 162
```

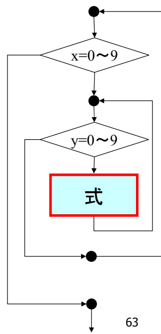

二重ループのまとめ①

eachを用いた場合

```
(0..9).each{ |x|
    (0..9).each{ |y|
        式
    }
}
```

timesを用いた場合

```
10.times{ |x|
    10.times{ |y|
        式
    }
}
```

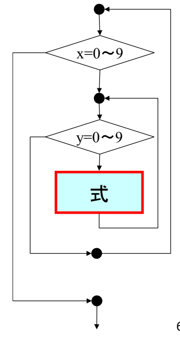


外側と内側の制御変数は異なる名前にする
（この場合は，x と y）

二重ループのまとめ②

while を用いた場合

```
x = 0
while x < 10 do
        y = 0
        while y < 10 do
                式
                y += 1
        end
        x +=1
end
```




```
(0..9).each{ |x|
    (0..9).each{ |y|
        z = x*x + y*y
        print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
    }
}
```

times を用いて書いた場合

```
10.times{ |x|
        10.times{ |y|
                z = x*x + y*y
                print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
        }
}
```



```
(0..9).each{ |x|
    (0..9).each{ |y|
        z = x*x + y*y
        print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
    }
}
```

while 文で書いた場合

```
x = 0
while x < 10 do
        y= 0
        while y < 10 do
                z = x * x + y * y
                print( " x = " , x , " y = " , y , ": z= " , z , "¥n" )
                y += 1
        end
        x += 1
end
```

二重ループの例

二重ループの例①

九九の表の表示プログラム

```
(1..9).each{ |x|
  (1..9).each{ |y|
      printf( " %d×%d=%2d" , x , y , x * y  )
  }
  print( "¥n" )
}
```

%2d
整数を二桁で表示

x=1

```
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```

x=1,2,...9と変わっていく

x=2

```
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```

・・・

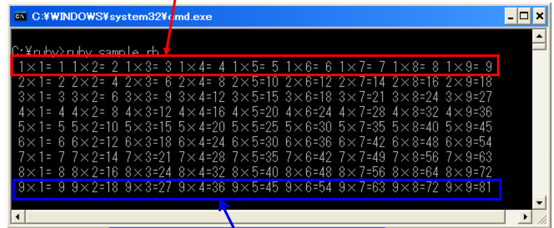
x=9

```
(1..9).each{ |y|
    printf( " %d×%d=%2d" , x , y , x * y  )
}
print( "¥n" )
```


二重ループの例①

前頁の実行画面

xを1とし，yを1から9まで変える



xを9とし，yを1から9まで変える

二重ループの例②

対角行列の表示プログラム

```
(1..9).each{ |x|
        (1..9).each{ |y|
            if x == y then
                  print( "1 " )
            else
                  print( "0 " )
            end
        }
        print( "¥n" )
}
```

xとyの値が同じ→"1 "
異なる場合は→"0 "

9個出力したら改行

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```


二重ループの例②の出力結果

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```

x=1 の時
x=2 の時
x=3 の時
x=4 の時
x=5 の時
x=6 の時
x=7の時
x=8の時
x=9の時

二重ループの例②の出力結果

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
```

x=y=1の場合
x=y=2の場合
x=y=3の場合
x=y=4の場合
x=y=5の場合
x=y=6の場合
x=y=7の場合
x=y=8の場合
x=y=9の場合

二重ループの例③

```
(1..9).each{ |x|
   (1..9).each{ |y|
      if x == (10-y) then
            print( "1 " )
      else
            print( "0 " )
      end
   }
   print( "¥n" )
}
```

xと(10-y)の値が同じ→"1 "
異なる場合は→"0 "

```
C:¥ruby>ruby sample.rb
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```


二重ループの例③の出力結果

```
C:¥ruby>ruby sample.rb
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

x=1 の時
x=2 の時
x=3 の時
x=4 の時
x=5 の時
x=6 の時
x=7の時
x=8の時
x=9の時

二重ループの例③の出力結果

```
C:¥ruby>ruby sample.rb
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

x=1,y=9
x=2,y=8
x=3,y=7
x=4,y=6
x=5,y=5
x=6,y=4
x=7,y=3
x=8,y=2
x=9,y=1

二重ループの例③'

```
(1..9).each{ |x|
   (1..9).each{ |y|
      if x == y or x == (10-y) then
            print( "1 " )
      else
            print( "0 " )
      end
   }
   print( "¥n" )
}
```

xとyの値が同じ，もしくはxと(10-y)の値が同じ→"1 "
異なる場合は→"0 "

```
C:¥ruby>ruby sample.rb
1 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 1 0
0 0 1 0 0 0 1 0 0
0 0 0 1 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0
0 0 1 0 0 0 1 0 0
0 1 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 1
```


二重ループの例④

```
(1..9).each { |i|
  (1..i).each { |j|
          print( j )
  }
  print ( "¥n" )
}
```

j は 1から i まで変わる

```
C:¥ruby>ruby sample.rb
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

二重ループの例④の出力結果

```
C:¥ruby>ruby sample.rb
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

i=1の時, j=1
i=2の時, j=1～2
i=3の時, j=1～3
⋮
i=8の時, j=1～8
i=9の時, j=1～9

二重ループの例⑤

```
(1..9).each { |i|
  (1..(10-i)).each{ |j|
    print( j )
  }
  print ( "¥n" )
}
```

j は 1から 10-i まで変わる

```
C:¥ruby>ruby sample.rb
123456789
12345678
1234567
123456
12345
1234
123
12
1
```


二重ループの例⑤の出力結果

```
C:¥ruby>ruby sample.rb
123456789
12345678
1234567
123456
12345
1234
123
12
1
```

i=1の時, j=1～9
i=2の時, j=1～8
i=3の時, j=1～7
⋮
i=8の時, j=1～2
i=9の時, j=1

二重ループの例⑥

```
11.times { |i|
  d = Math.sqrt( 100 - i*i ).to_i
  (1..d).each{
     print( " " )
  }
  ((d+1)..10).each{
      print( "*" )
  }
  print( "¥n" )
}
```

d回は" "(空白)を表示

10-d回は"*"を表示

```
C:¥WINDOWS¥system32¥c...
C:¥ruby>ruby sample.rb
          *
          *
          *
          *
         **
         **
        ***
       ****
      ******
**********
```

二重ループの例⑥

```
11.times { |i|
  d = Math.sqrt( 100 - i*i ).to_i
  print( d , "¥n" )
}
```

d の値はどう変わっていっているでしょうか

```
C:¥ruby>ruby sample.rb
10
9
9
9
9
8
8
7
6
4
0
```


二重ループの例⑥の出力結果

```
C:¥WINDOWS¥system32¥c...
C:¥ruby>ruby sample.rb
          *
          *
          *
          *
         **
         **
        ***
       ****
      ******
**********
```

d回" ", 10-d回"*"を表示

dの値: 10, 9, 9, 9, 9, 8, 8, 7, 6, 4, 0

10個 " ", 0個 "*"
9個 " ", 1個 "*"
8個 " ", 2個 "*"
7個 " ", 3個 "*"
6個 " ", 4個 "*"
4個 " ", 6個 "*"
0個 " ", 10個 "*"

二重ループの例⑦
（どうしてこのような出力になるのでしょうか）

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  (1..d).each{
    print( " " )
  }
  ((d+1)..20).each{
    print( "*" )
  }
  print( "¥n" )
}
```

d回は" "（空白）を表示

20-d回は"*"を表示

```
C:¥ruby>ruby sample.rb
                    *
                    *
                    *
                   **
                  ***
                  ***
                ******
               *******
            ***********
********************
C:¥ruby>
```


二重ループの例⑦
（ヒント：dの値はどう変わっていくでしょうか）

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  print( d , "¥n" )
}
```

```
C:¥ruby>ruby sample.rb
20
19
19
19
18
17
16
14
12
8
0
```


練習問題

練習問題①～③

練習問題①

- x, y ともに0から10までの整数とする．この場合，

① xとyの和が10となる組み合わせ

② x^2とy^2の和が100となる組み合わせ

を二重ループを用いて、それぞれ求めなさい

練習問題②

- 下記のプログラムにおいて，配列aには1から100の乱数が格納されます．
- j=1～8において，(a[j-1]+a[j]+a[j+1])/3

を求めなさい（移動平均）

```
a = []
10.times{ |i|
        a[ i ] = rand(100)+1
        print( a[ i ] , " " )
}
print( "¥n" )
```

```
C:¥Ruby>ruby sample.rb
21 91 34 17 61 22 7 86 93 93
21 91 34 -> 48
91 34 17 -> 47
34 17 61 -> 37
17 61 22 -> 33
61 22 7 -> 30
22 7 86 -> 38
7 86 93 -> 62
86 93 93 > 90
```


練習問題③

- 二次元配列 a, b の和と差を印字するプログラムを二重ループを用いて書きなさい

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

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

```
C:¥Ruby>ruby sample.rb
10 10 10
10 10 10
10 10 10

-8 -6 -4
-2 0 2
4 6 8
```

練習問題

- 練習問題①から③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。