プログラミング言語　第十二回

担当：篠沢 佳久
櫻井 彰人

平成20年7月14日

本日の内容

- 講義のまとめ（本資料）
  - Rubyにおける式の記述
  - 入出力
  - 条件式
  - 繰り返し①～④
  - 一次元配列
  - 二重ループ
  - 二次元配列
  - 関数
- まとめの問題

評価の付け方

- 平常点
- レポート4回
  - 本日の問題は一回分のレポートとして評価

Rubyにおける式の記述

算術演算子，変数
代入式
データの型，型変換

データの型

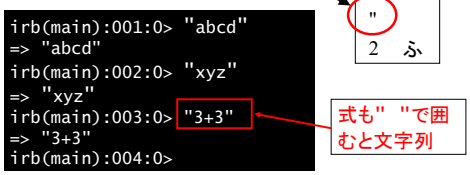
- データ
  - コンピュータの演算・操作の対象
  - 文字列、小数点のある数、小数点のない数
- データの型
  - そのデータに適用が許される演算・操作の集合
- 小数点のない数: 小数点のない数による加減乗除
- 小数点のある数：小数点のある数による加減乗除
  - 小数点のない数に小数点のある数を足そうとすると（それはできない）、前者を小数点のある数に変換して、足し算をする
  - この変換を「型変換」という

数値には型がある

- 「3」は「整数」
- 「3.0」は「小数」
- 「3/2」は「整数を整数で割り算」
  → 結果は「整数」
- 「3.0/2」は「小数を整数で割り算」
  → 結果は「小数」
- 「3.0/2.0」は「小数を小数で割り算」
  → 結果は「小数」

文字列型

- 文字列を使用する場合は" "（ダブルクォート）で囲む



整数型の算術演算子

演算子	用途	例	演算結果
+	加算	3+2	5
-	減算	4-2	2
*	乗算	2*2	4
/	除算	4/2	2
%	剰余	5%2	1
**	べき	5**3	125

浮動少数点数型の算術演算子

演算子	用途	例	演算結果
+	加算	3.1+2.2	5.3
-	減算	4.2-2.1	2.1
*	乗算	2.1*2.1	4.41
/	除算	4.2/2.1	2.0
%	剰余	5.0%2.1	0.8
**	べき	2.1**0.5	1.44913

比較演算子

演算子	用途	例	演算結果
==	等	3==2	false
>	大	4 > 2	true
<	小	4 < 2	false
>=	大or等	4>=2	true
<=	小or等	4<=2	false
!=	非等	3 != 2	true

論理演算子

演算子	用途	例	演算結果
!	否定	! (3==2)	true
&&	かつ	2==2 && 4>2	true
||	または	2==3 || 4>2	true
not	否定	not 3==2	true
and	かつ	2==2 and 4>2	true
or	または	2==3 or 4>2	true

代入演算子

演算子	用途	例	演算結果
=	代入	x = 4.3	x ← 4.3
+=	加算後代入	x +=3.1	x ← x + 3.1
-=	減算後代入	x -=3	x ← x - 3
*=	乗算後代入	x *=3	x ← x * 3
/=	除算後代入	x /=3	x ← x / 3
%=	剰余の代入	x %=3	x ← x % 3
**=	冪の代入	x **=3	x ← x ** 3

数学用関数①

- 平方根
  - Math.sqrt(2)
- 三角関数
  - Math.sin(Math::PI)
  - Math.cos(Math::PI)
  - Math.tan(Math::PI)

ラジアン

π
Math::PI

数学用関数②

- 自然対数
  - Math.log(Math::E)
- 常用対数
  - Math.log10(100)
- 指数
  - Math.exp(2)

e
Math::E

その他の数学関数の一例
http://www.ruby-lang.org/ja/man/?cmd=view;name=Math

変数とは

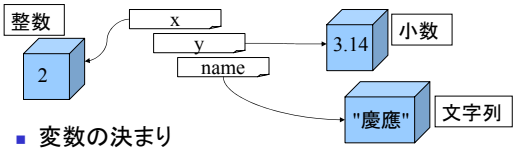
- 今日覚える重要なことに「変数」があります。
- 変数は、中学から代数で慣れ親しんだ変数とそっくりな概念です。そっくりですが、随分違いもあります。よく注意してください。
- コンピュータにおける変数とは、まず第一に、データを一時的に記憶しておく場所です。
- そして、場所を区別するために名前（識別子）をつけます。
- Rubyの変数の型は記憶しているデータの型で決まります（重要！）
  - Ruby の変数は、単に、場所の名前と思えばよい

変数①

- x = 2
  - 変数x は整数型で値は2
- y = 3.14
  - 変数yは小数型で値は3.14
- name = "慶應"
  - 変数nameは文字列型で値は"慶應"

変数②

- コンピュータにデータを記憶させる機能のこと
  - 変数: 名前を作って、データが入った箱を区別するイメージ



- 変数の決まり
  - 変数に「名前」（識別子）をつける
  - 変数に値を代入する、とは、割当てる or 割付けること

変数への値の代入

- 「識別子 = 値」で変数に値を代入
- 数学のイコールとは意味が違うことに注意

```
irb(main):005:0> x = 100
=> 100
irb(main):005:0> y = 100
=> 100
irb(main):006:0> x = "Nice !"
=> "Nice !"
irb(main):007:0> x = 3.14
=> 3.14
irb(main):008:0>
```

代入式①

- x = 2
- y = 10
- (x + y) / 2
  => 6
- x * y
  => 20
- Math.sqrt(y / x)
  => 2.23606797749979

x , y を変数
x = 2 を代入式と呼ぶ

代入式②

- x = 2
- y = 10

- a = (x + y) / 2
- b = x * y
- c = y / x
- d = Math.sqrt(c)

整数型への変換

- 整数に変換
  3.1415.to_i

  "3".to_i

  "3".to_i + 5

整数へ変換
値.to_i

小数型への変換

- 小数に変換
  3.to_f

  "3.1415".to_f

  "3".to_f

  "3.1415".to_f * 2.5

小数へ変換
値.to_f

文字列型への変換

- 文字列に変換
  3.to_s

  3.1415.to_s

  3.to_s +"5"

  3.1415.to_s * 2

文字列へ変換
値.to_s

変数の型変換①

- 変数においても型変換が可能

x=3 整数型
x.to_f 小数へ変換
x.to_s 文字列へ変換

x=3.1415 小数型
x.to_i 整数へ変換
x.to_s 文字列へ変換

変数の型変換②

- 文字列型の変数においても型変換が可能

x="3.14" 文字列型
x.to_f 小数へ変換
x.to_i 整数へ変換

x="abcd" 文字列型
x.to_f 小数へ変換
x.to_i 整数へ変換

if then else end①

- 「if 論理式 then 式1 else 式2 end」という式がある
- 論理式がtrueならば式1を実行，falseならば式2を実行

```
if a > 0 then
        y = 3
else
        y = -3
end
```

a>0 ならば y=3

違う場合は y=-3

if then else end②

```
x = -10
if x < 0 then
        -x
else
        x
end
```

x<0 ならば -x ← xは-10なのでこちらの式が実行される

そうでなければ x

入出力

標準入力
標準出力

キーボードからの入力

- line = gets.chop
- line = gets.chomp

- gets
  - キーボードから文字列を読み込む
  - この場合，改行文字が文字列の最後に含む
- chop（chomp）
  - 最後の一文字を削除する（chompの場合，改行の場合のみ削除）
- line には読み込まれた文字列が代入される
- 文字列のため，数字に「to_i」「to_f」を用いて数値に変換する

標準入力①

- キーボードからの入力
- gets

入力

```
irb(main):018:0> gets
34
=> "34¥n"
```

① getsと打つ
② キーボードから入力

```
irb(main):019:0> gets
abcd
=> "abcd¥n"
```

入力した値は文字列として処理される
最後に改行「¥n」が入る

標準入力②

```
irb(main):024:0> a=gets
3.1415
=> "3.1415¥n"
irb(main):025:0> p a
"3.1415¥n"
=> nil
```

変数a に入力した値を代入
変数a は文字列型
最後に改行文字が入る

```
irb(main):028:0> x=gets
abcd
=> "abcd¥n"
irb(main):029:0> p x
"abcd¥n"
=> nil
```

変数x に入力した値を代入
変数x は文字列型
最後に改行文字が入る

標準入力③

```
irb(main):041:0> x=gets
3.1415
=> "3.1415¥n"
irb(main):042:0> x.chomp
=> "3.1415"
irb(main):043:0> x.chomp.to_f
=> 3.1415
irb(main):044:0> x.chomp.to_i
=> 3
```

chop で最後の一文字（改行）を削除

文字列型を小数に変換

文字列型を整数に変換

出力式

- print(変数)
- print("コメント")
  - コメント（文字列）を表示する場合は" "で囲む
- print("コメント" , 変数)
- print(変数1 , 変数2 , ・・・ , 変数n)
  - 変数，コメントを一行で表示したい場合は，「,」で区切る

改行

- 「¥n」
  - 改行文字
  - 一行改行される

```
irb(main):021:0> x=2;y=3
=> 3
irb(main):022:0> print( "x=" , x , " y=" , y )
x=2 y=3=> nil
irb(main):023:0> print( "x=" , x , "¥n y=" , y )
x=2
 y=3=> nil
```

改行される

その他の出力①

- puts(変数)
- puts("コメント")
- puts("コメント" , 変数)

その他の出力②

- p というのもあります

```
irb(main):031:0> x=Math::PI
=> 3.14159265358979
irb(main):032:0> p x
3.14159265358979
=> nil
```

```
irb(main):035:0> x="abcd"
=> "abcd"
irb(main):036:0> p x
"abcd"
=> nil
```

文字列と数値を区別して表示

出力フォーマット①

- より凝って出力したい場合には、
  - printf("%+d", 1)

  のように、printf を用います
- " … " の部分がフォーマット(書式)です。
- 詳細は、

  http://www.ruby-lang.org/ja/man/
  ⇒「目次」中の「付録」中の sprintfフォーマット

出力フォーマット②

- %d
- printf("%d" , x)
  - xを10進整数で表示する
- %f
- printf("%f" , x)
  - xを10進浮動小数点数で表示する

出力フォーマット③

- %x
- printf("%x" , x)
  - xを16進整数で表示する
- %s
- printf("%s" , x)
  - xを文字列で表示する

出力フォーマット④

- 桁数の指定
- printf("%5d" , x)
  - xを5桁の整数で表示する
- printf("%5.2f" , x)
  - xを5桁の小数，小数点以下を2桁で表示する

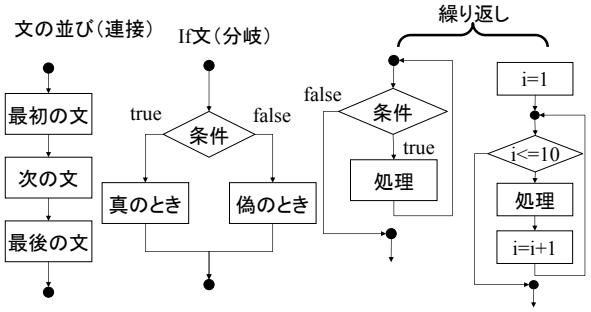
出力フォーマット⑤

- 桁数の指定
- printf("%-5d" , x)
  - xを左詰で5桁の整数で表示する
- printf("%-5.2f" , x)
  - xを左詰めで5桁の小数，小数点以下を2桁で表示する

条件式

制御構造
条件式

制御構造



条件式（if式）

プログラムを書いている際には

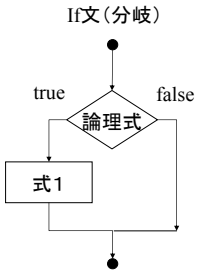
- ある条件式が成立した場合には，処理Aを行ない，成立しなかった場合には処理Bを行なう

という要求が発生する

if式①

if 論理式 then 式1 end

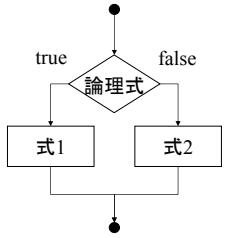
if 論理式 then
　式1 # 論理式がtrueのときの値
end



if式②

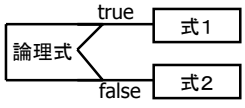
if 論理式 then 式1 else 式2 end

if 論理式 then
　式1 # 論理式がtrueのときの値
else
　式2 # 論理式がfalseのときの値
end

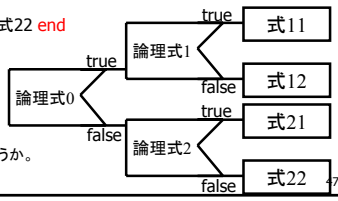


if式③

if 論理式 then 式1 else 式2 end

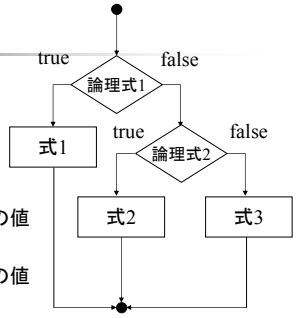


if 論理式0 then
　if 論理式1 then 式11 else 式12 end
else
　if 論理式2 then 式21 else 式22 end
end



正式にはPAD図といいます
ここでは、分岐図とでもいいましょうか。

if式④



if 論理式1 then
　式1 # 論理式1がtrueのときの値
elsif 論理式2 then
　式2 # 論理式2がtrueのときの値
else
　式3 # 論理式1,2がfalseのときの値
end

繰り返し①

無限の繰り返し
if修飾子

無限の繰り返し①

```
loop{
        式
}
```

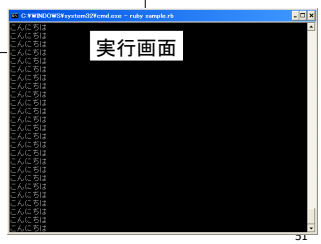
式が永久に実行される
停止するために break を用いる

```
loop{
        式
        break 条件式
}
次の式
```

条件式を満たした場合のみ停止する（loopブロックの次の式を実行する）

無限の繰り返し②

```
loop{
  print( "こんにちは¥n" )
}
```

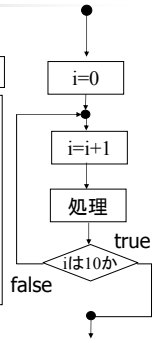


無限に「こんにちは」と表示される
停止するにはCtrlキーを押しながらc

無限の繰り返し③

10回で「こんにちは」の表示をやめるには？

```
i = 0
loop{
  i = i+1
  print( i , "回目のこんにちは¥n" )
  break  if i == 10
}
```



無限の繰り返し④

- loop{ … }
- 上記「 …」を無限に繰り返す。無限個のコピーを作ると考えてもよい。ただし、いきなり作るのではなく、必要があったら作るのですが。
- しかし、いずれにせよ、無限に作られるのは困る。
- 途中で止めなければ意味がない。
- 途中で止める道具（これも式だが、まったく式らしくない）が break です。

```
i = 0
loop{
  print( "やっほ～ " )
  if i>=10 then break end
  puts( "      Yee-ha! " )
  i = i+1
}
```

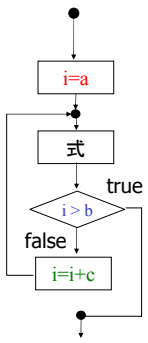
```
i = 0
loop{
  print( "やっほ～ " )
  break if i>=10
  puts( "      Yee-ha! " )
  i = i+1
}
```

これを if 修飾子という
式 if 論理式
が一般形

無限の繰り返し⑤

a から b まで c ずつ加算しながら繰り返し処理を行なう

```
i = a
loop{
        式
        break if  i > b
        i += c
}
```



if修飾子①

```
if i==0 then
   break
end
```

```
break if i==0
```

```
if a > b then
   print( " aはbよりも大きい" )
end
```

```
print( " aはbよりも大きい" ) if a>b
```


if修飾子②

```
if a !=b  then
   a = a * 2
   b = b + 5
end
```

```
a=a*2 ; b = b+ 5  if a != b
```


繰り返し②

回数の決まった繰り返し
（times, each）

times①

- 同じ処理をn回繰り返したい
- n.times

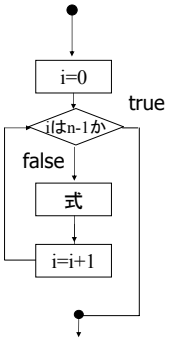
```
n.times {
    式
}
```

式をn回繰り返す

times②

```
n.times{ |i|
   式
}
```

n回式を繰り返す
iには自動的に0からn-1が代入される



times③

```
10.times { |i|
  print( i , "回目のこんにちは¥n" )
}
```

```
C:¥ruby>ruby sample.rb
0回目のこんにちは
1回目のこんにちは
2回目のこんにちは
3回目のこんにちは
4回目のこんにちは
5回目のこんにちは
6回目のこんにちは
7回目のこんにちは
8回目のこんにちは
9回目のこんにちは
```

変数iには0から代入されていく

変数iには9まで代入されていく

each①

```
n.times{ |i|
  式
}
```

n回式を繰り返す
iには自動的に0からn-1が代入される

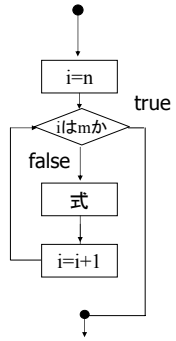
```
(n..m).each{ |i|
  式
}
```

m-n回式を繰り返す
iには自動的にnからmが代入される

each②

```
(n..m).times{ |i|
  式
}
```

i には自動的にnからmが代入される
その結果、式は m-n回繰り返される



each③

10のべき乗を表示するプログラム

```
(0..9).each{ |i|
  print( 10 ** i , "¥n" )
}
```

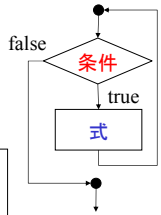
```
10.times{ |i|
  print( 10 ** i , "¥n" )
}
```


繰り返し③

whileループ

whileによる繰り返し

```
while 条件 do
      式
end
```



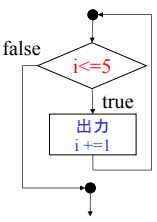
条件がtrueである場合は式を実行し，false の場合は式を実行しない

すなわち条件がtrueである限りは式を繰り返し実行する

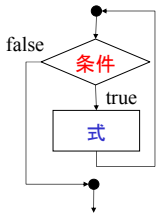
while による繰り返し

```
i = 1
while i <= 5 do
  puts( i.to_s + "回目です。" )
  i += 1
end
```

```
1回目です。
2回目です。
3回目です。
4回目です。
5回目です。
```



while による繰り返しにおける注意



条件がtrueである限り式は実行され続ける．そのため，条件がfalseとなり終了するように条件を設定しなければならない

times（each）と while の違い

- 同じである。ただし、コンセプトには結構な違いがある
  - times, each: 各回には、制御変数の値の違いしかない
  - while, loop: 各回には、制御変数以外の変数で違いがある。または、制御変数を自分で作ってあげないといけない

例: times と while の違い①

"繰り返し回数"が分っている場合

```
n =10
n.times{ |i|
  puts( i )
}
```

⇔

制御変数

```
i=0
while i<10 do
        puts( i )
        i = i + 1
end
```

制御変数を自分で指定しなければならない

例: times と while の違い②

```
i = 2
while i < 100000 do
   puts( i )
   i = i * 2
end
```

"停止条件"が分っているが"繰り返し回数"は分らない

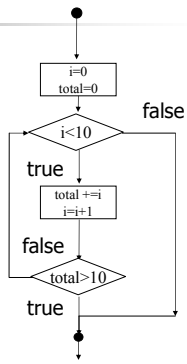
times では書きづらい

break文①

- 繰り返し（ループ）の中で使用
- そのbreakが所属するループを1つ抜ける

break文②

```
i = 0
total = 0
while i < 10 do
   total += i
   i += 1
   break if total > 10
end
```



next①

- nextはもっとも内側のループの次の繰り返しにジャンプします。
- while であれば、「継続条件」の判定の直前が再開場所です。

next②

```
(0..5).each { |i|
  if ( i==1 || i==4 ) then
     next
  end
  puts( "iは #{i}" )
}
```

「または」です

出力結果

```
iは 0
iは 2
iは 3
iは 5
```

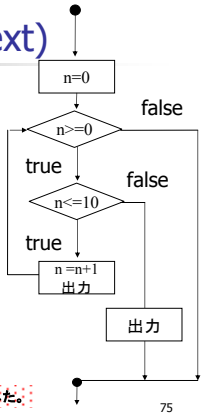

プログラム例(break,next)

ループの先頭であるwhileに戻る

```
n = 0
while n>=0 do
  if n<=10 then
    print( n, " ")
    n += 1
    next
  else
    print( "変数 n は10を越えました。" )
    break
  end
end
```

ループwhileを抜ける

```
0 1 2 3 4 5 6 7 8 9 10 変数 n は10を越えました。
```



繰り返し④

その他の繰り返し
(upto, downto, step, for)

downto, upto, and step

- 例で学ぼう

```
7.downto(3) { |i| print( i," " ) }
```
7 6 5 4 3

```
3.upto(7) { |i| print( i," " ) }
```
3 4 5 6 7

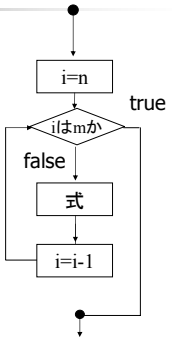
```
2.step(12,3) { |i| print( i," " ) }
```
2 5 8 11

```
12.step(2,-3) { |i| print( i," " ) }
```
12 9 6 3

downto①

```
n.downto(m){ |i|
      式
}
```

n-m回式を繰り返す(n>m)
iには自動的にnからmが代入される



downto②

```
total = 0
10.downto(0){ |i|
  total += i
  break if total > 20
}
```

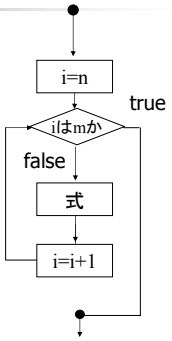
while で書いた場合

```
i = 10
total = 0
while i >= 0 do
  total += i
  break if total > 20
  i -= 1
end
```


upto①

```
n.upto(m){ |i|
    式
}
```

m-n回式を繰り返す（n<m）
iには自動的にnからmが代入される



upto②

```
n.upto(m){ |i|
    式
}
```

⇔

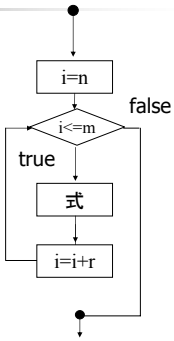
```
(n..m).each { |i|
    式
}
```

基本的には each と同じ

step①

```
n.step(m,r){ |i|
    式
}
```

nからmまで刻み幅はrで繰り返す
i には n, n+r, n+2*r ,... が入る



step②

```
0.step(10,2){ |x|
  print( x , "¥n" )
}
```

```
C:¥ruby>ruby sample.rb
0
2
4
6
8
10
```

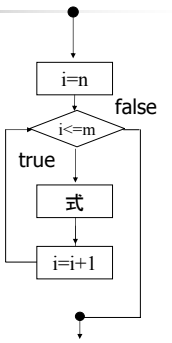
```
1.step(10,2){ |x|
  print( x , "¥n" )
}
```

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


forループ①

```
for i in n..m do
  式
end
```

```
(n..m).each{ |i|
  式
}
```



forループ②

```
for x in 1..10 do
  puts( 2**x )
end
```

```
C:¥ruby>ruby sample.rb
2
4
8
16
32
64
128
256
512
1024
```


一次元配列

配列の宣言
配列の要素の参照

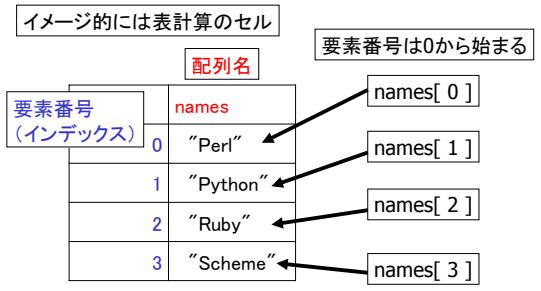
配列の宣言

- 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①

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



配列の要素②

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

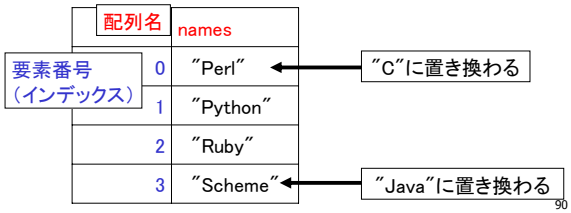
	a	
0	0	a[0]
1	2	a[1]
2	4	a[2]
3	6	a[3]
4	8	a[4]

```
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"
```



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
```


配列の宣言①

- 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③

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[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

配列の宣言④

```
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）

配列の宣言⑤

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!

配列の要素の参照方法

- 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①

	a
0	0
1	2
2	4
3	6
4	8

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

要素番号を用いて一つずつ取り出す②

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

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

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


要素番号を用いて一つずつ取り出す③

```
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

要素を直接一つずつ取り出す①

```
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

要素を直接一つずつ取り出す②

```
[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③

- 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" )
}
```


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

二重ループ

二重ループ①

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

```
(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

②のループによって，xは0から9まで変わる

二重ループの必要性

```
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 = 9
(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

二重ループ②

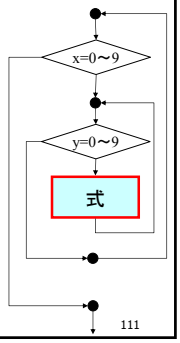
eachを用いた場合

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

timesを用いた場合

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

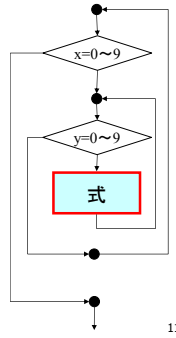
外側と内側の制御変数は異なる名前にする
（この場合は，x と y）



二重ループ③

while を用いた場合

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



二次元配列

二次元配列の宣言
二次元配列の要素の参照

二次元配列の宣言①

3×3の行列 a

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

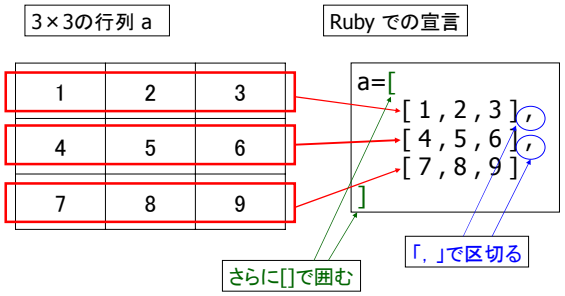
表の場合

1	2	3
4	5	6
7	8	9

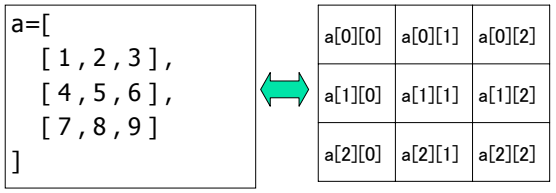
Ruby での宣言

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

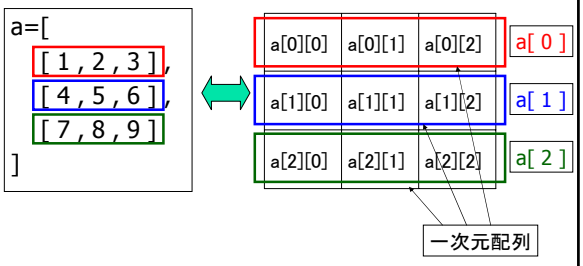
二次元配列の宣言②



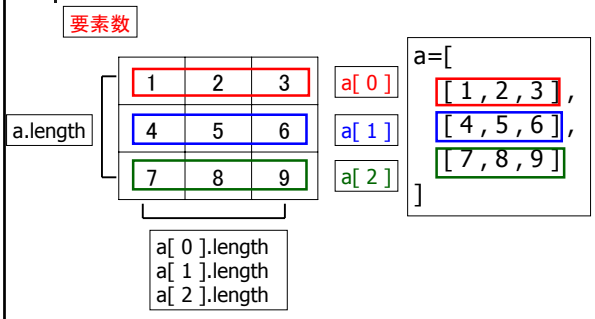
二次元配列の要素の参照①



二次元配列の要素の参照②



二次元配列の要素の参照③

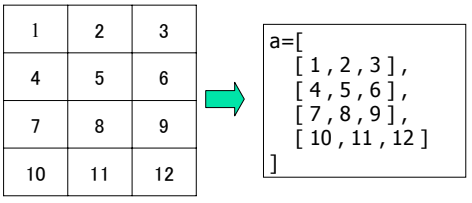


二次元配列の宣言①

- 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

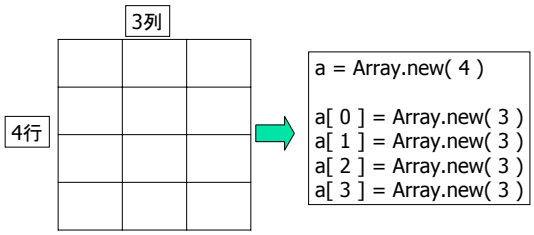
二次元配列の宣言②

- 要素の値が分かっている場合



二次元配列の宣言③

- 要素数のみ分かっている場合



二次元配列の宣言④

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]]
```


二次元配列の要素の参照①

要素番号（インデックス）を用いての参照

```
a=[
  [ 1 , 2 , 3 ] ,
  [ 4 , 5 , 6 ] ,
  [ 7 , 8 , 9 ] ,
  [ 10 , 11 , 12 ]
]

4.times{ |i|
  3.times{ |j|
    print( " a[ " , i , " ][ " , j , " ] = " ,
        a[ i ][ j ] , "¥n" )
  }
}
```

i=0, j=0～2

```
C:¥Ruby>ruby sample.rb
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
```

i　j

二次元配列の要素の参照②

```
a=[
  [ 1 , 2 , 3 ] ,
  [ 4 , 5 , 6 ] ,
  [ 7 , 8 , 9 ] ,
  [ 10 , 11 , 12 ]
]
a.length.times{ |i|
  a[ i ].length.times{ |j|
    print( " a[ " , i , " ][ " , j , " ] = " ,
        a[ i ][ j ] , "¥n" )
  }
}
```

a.lengthの値は4

```
C:¥Ruby>ruby sample.rb
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
```

a[0].length, a[1].length, a[2].length, a[3].length は全て3

二次元配列の要素の参照③

each を用いての参照

```
a=[
  [ 1 , 2 , 3 ] ,
  [ 4 , 5 , 6 ] ,
  [ 7 , 8 , 9 ] ,
  [ 10 , 11 , 12 ]
]

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

```
C:¥Ruby>ruby sample.rb
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
```


二次元配列の要素の参照④

要素番号（インデックス）ではなく直接、二次元配列の要素を参照するには？

```
a=[
  [ 1 , 2 , 3 ] ,
  [ 4 , 5 , 6 ] ,
  [ 7 , 8 , 9 ] ,
  [ 10 , 11 , 12 ]
]

a.each{ |i|
  p i
}
```

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

a[0]　a[1]　a[2]　a[3]

変数 i には順番に a[0] , a[1] , a[2] , a[3] が代入される

関数

関数とサブルーチン
関数の定義

関数の定義

```
def 関数名( 仮引数1, 仮引数2, ・・・,仮引数n )
    実行したい処理
    return 返す値
end
```

```
def max( x, y )
  if x >= y then
    return x
  else
    return y
  end
end
```

```
def average( x, y )
   z = (x+y)/2
   return z
end
```


関数の呼び出し

関数名(実引数1, 実引数2, ・・・, 実引数n)

```
i = 3; j = 5
x = max( i, j )
puts( "#{x} is the maximum of #{i} and #{j}" )
```

```
a = 9; b = 3
x = average( a, b )
puts( "#{x} is the average of #{a} and #{b}" )
```


関数の定義の例

- 2数の最大値を値とする関数 max(x,y) を定義してみる

```
def max( x, y )
  if  x >= y then
    return x
  else
    return y
  end
end

i = 3; j = 5
x = max( i, j )
puts( "#{x} is the maximum of #{i} and #{j}" )
```

仮引数

実引数

```
5 is the maximum of 3 and 5
```


引数の受け渡し

```
i = 3; j = 5
x = max( i, j )
```

i=3, j=5 として関数に渡される

必ず呼び出す関数の仮引数と一致する順番に実引数を記述する

```
def max( x, y )
  if  x >= y then
    return x
  else
    return y
  end
end
```

x=3, y=5 として処理する

関数の戻り値

```
def max( x, y )
  if  x >= y then
    return x
  else
    return y
  end
end
```

```
i = 3; j = 5
x = max( i, j )
```

関数の戻り値は5なのでxには5が代入される

x=3, y=5 なので y の値を返す

return 変数名
return 値
関数中において,値を戻す

サブルーチンと関数

- 違いは、
  - 関数: 戻り値あり
  - サブルーチン: 戻り値がない
- その結果、使い方が違う。
  - 関数: 式の中
  - サブルーチン: 一つの文として
- 本講義では区別なしに用いる

戻り値のない関数

- 「戻り値のない関数」は関数とは言い難いが、大目にみてください。
  - サブルーチンとか副プログラムと言うのが正しい。
- 仕事だけする。例えば、印字のみ。

```
def sayHello( n )
  n.times{ |i|
    i.times{
      print( " " )
    }
    puts( "Hello!" )
  }
end

sayHello( 4 )
```

```
Hello!
 Hello!
  Hello!
   Hello!
=> 4
```


引数も戻り値もない関数

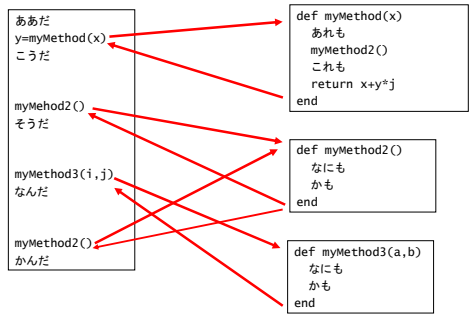
```
def max()
  x = 3
  y = 5
  if  x >= y then
    z = x
  else
    z = y
  end
  puts( "#{z} is the maximum of #{x} and #{y}" )
end

max()
```

引数がない

```
C:¥Ruby>ruby sample.rb
5 is the maximum of 3 and 5
```


関数のまとめ



おわりに

- FDアンケートに回答して下さい
- 楽しい夏休みを…
  - その前に期末試験頑張って下さい