Mathematica の基本操作（*Windows*）

- 起動： アイコンをダブル・クリック
- 計算実行： shift enter
- 計算が始まると ⇒ In[1] := が画面表示される

  角括弧

  式番号は自動的に更新される

  この後に式をタイプし， shift enter
- 終了： ファイルメニューのExit
- 計算続行： 下の余白をクリック

- 組込み関数： 関数名が大文字から始まる英文字の列

  InputForm, FullForm, Plus, Apply, List etc.

式

式の形式：　ヘッド[引数1，引数2，・・・，引数n]

角括弧

ヘッド，引数は他の式となりうる

複合式（；で区切る）
　式1；　式2；　・・・；　式n

FullForm　内部表現を見る
Apply　式のヘッドを替える

```
In[1] := FullForm[{a, b, c}]
Out[1] = List[a, b, c]

In[2] := FullForm[ a + b +c ]
Out[2] = Plus[a, b, c]

In[3] := Apply[Plus, {1, 2, 3}]
Out[3] = 6
```

記号

:=	右辺を登録．評価しない
=	割当て
= =	方程式の等号
%	得られた前の結果を入力に利用
%4	Out[4] の結果
?	オンライン・ヘルプ．関数，出力についての情報など
;	区切り．　; より前の結果は出力しない

{x, 0, 3}　変数の範囲　$0 \le x \le 3$

↳ 必要

```
In[4] := ?Log
  Log[z] gives the natural logarithm of z (logarithm to base e).
  Log[b, z] gives the logarithm to base b.

In[5] := ?*Exp*
```

Exp を含む文字列のリスト

四則演算

+, -, * (または, スペース), /　　加減乗除
^　　べき乗

In[6] := 8.1 + 2.5
Out[6] = 10.6

In[7] := 8.1 - 2.5
Out[7] = 5.6

In[8] := 8.1*2.5
Out[8] = 20.25

In[9] := 8.1/2.5
Out[9] = 3.24

In[10] := 8.1 2.5
Out[10] = 20.25

In[11] := 2.5^2
Out[11] = 6.25

In[12] := %/2
Out[12] = 3.125

In[13] := %9 2.5
Out[13] = 8.1

In[14] := 4 + 5; 7 4
Out[14] = 28

表示

［アトム（atomic expression）］

３種類

(1) シンボル(Symbol)： 最初英文字，その後に英数字

(2) 数字： 整数(Integer)，実数(Real)，複素数(Complex)，分数(Rational)

(3) 文字列(String)： 引用符（“”）で囲まれた文字，数字，スペース

アトムのFullForm はアトム自身

```
In[15] := {FullForm[darwin], FullForm[“read my lips”], FullForm[5]}
Out[15] = {darwin, “read my lips”, 5}
```

アトムの Head （0番目の部分）はアトムの型

```
In[16] := {Head[List], Head[“read my lips”], Head[5]}
Out[16] = {Symbol, String, Integer}
```

［非アトム（nonatomic expression）］

Part 式から非アトムを抽出

ReplacePart 非アトムを置き換える

```
In[17] := Part[{a, 7, c}, 1]
Out[17] = a
```

↳ リストの１番目の要素を抽出

```
In[18] := Part[a + b + c, 0]
Out[18] = Plus
```

↳ ヘッド（0番目）を抽出

```
In[19] := ReplacePart[{a, 7, c}, e, 2]
Out[19] = {a, e, c}
```

e をリスト2番目の要素と置き換える

パターン

パターン: _ を含む表現

_ (Blank), _ _ (BlankSequence), _ _ _ (BlankNullSequence)

パターンにはラベル（名前）をつけることができる.

name_, name_ _, name_ _ _, _h, _ _h, _ _ _h

_ _ 1つ以上の連続式, _ _ _ ゼロかそれ以上の連続式

- パターン・マッチ条件
- プログラム中のダミー変数

x^2 $\xleftarrow{\text{MatchQ}}$ _, x^_, x^_Integer, _Power, _^2, _Symbol^_Integer, _^_

```
In[20]:= MatchQ[{a, b, c}, {_, _}]
Out[20]=True
In[21]:= MatchQ[{  }, {_ _}]
Out[21]=False
In[22]:= MatchQ[x^2, _ _]
In[22]=True
```

```
In[23]:= MatchQ[{a, b, c}, {_}]
Out[23]=False
In[24]:= MatchQ[{  }, {_ _ _}]
Out[24]=True
In[25]:= MatchQ[x^2, _ _ _]
Out[25]=True
```

評価（演算の進め方）

1．書き換えルール

Set関数（=），　SetDelayed関数（:=）

{ 左辺：　パターン
{ 右辺：　書き換えテキスト

(1) パターンされた（_がついた）シンボルを右辺に代入
(2) ヘッドが評価．引数を左から右に評価
(3) 大域ルール・ベース(global rule base)に適用し，評価
(4) 書き換えルールがなくなるまで続ける

両関数の相違

Set関数（=）　右辺は大域ルール・ベースに置かれる前に評価
SetDelayed関数（:=）　　左辺，右辺のどちらも評価せず，大域ルール・ベースに置かれる

1．1　Set 関数(=)

- 表記法　　左辺＝右辺
- 値(リスト，数)にニックネームを与え，代用
  - 左辺：　名前(1文字で始まる)で始まる
  - 右辺：　式か括弧で囲まれた複合式．_は含まない

```
In[26] := a ={-1, 1}
Out[26] = {-1, 1}

In[27] := rand1 = Random[Integer, {1, 2}]
Out[27] = 2

In[28] := ?a
Global‘a
a ={-1, 1}

In[29] := ?rand1
Global‘rand1
rand1 = 2
```

```
In[30] := rand2 = ( b={-1, 1}; Random[real], b])
Out[30] = -0.230583

In[31] := ?b
Global‘b
b = {-1, 1}

In[32] := rand3 = {-1, 1}[[Random[Integer, {1, 2}]]];

In[33] := ?rand3
Global‘rand3
rand3 = -1

In[34] := rand3 = {-1, 1}[[Random[Integer, {1, 2}]]];

In[35] := ?rand3
Global‘rand3
rand3 = 1

In[36] := rand4 = {-1, 1}[[Random[Integer, {1, 2}]]]; Print[rand4];
                rand4 = rand4 + {-1, 1}[[Random[Integer, {1, 2}]]]
Out[36] = -1
Out[36] = -2
```

同一関数

結果は異なる

値はその都度更新

Printで出した中間値

更新値

１．２　SetDelayed関数（:=）

表記法　　左辺 := 右辺

　左辺：　名前 [name_,　‥]

　右辺：　式，複合式 [nameを含んでよい]

Name [arg1_, arg2_, ‥ , argn_] := (expr1; expr2; ‥ ; exprn)

ダミー変数

In[37] := f[x_] := x^2

In[38] := ?f

Global‘f

f[x_] := x^2

In[39] := f[8]

Out[39] = 64

In[40] := f[3]

Out[40] = 9

１．３　書き換えルールの制限の設置

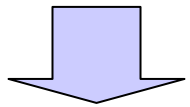
_h, _?, _/; は左辺のダミー変数に付属

１．４　書き換えルールの中の名前の局所化

Set, SetDelayed 関数の右辺を評価

⇒　全ての補助関数の書き換えが大域ルール・ベースに

⇒　他の箇所の名前とかち合えば問題



書き換えルールと補助関数を分離

⇨ 分離した書き換えルールで大域ルール・ベースに置かない
ルールの評価で使われるときだけ存在

⇒　Module関数

left-hand side :=

Module[{name1 = val1, name2, ・・}, right-hand side]

Module 計算の途中で使われる変数を関数の内部に局所化
始めの引数：リスト（局所的に使われる）
第２引数：式

$$\frac{x_1 + x_2 + \cdots + x_n}{(x_1 - x_2)(x_2 - x_3)\cdots(x_{n-1} - x_n)(x_n - x_1)}$$

```
In[41] := r = Table[x_i, {i, 3}]
Out[41] = {x_1, x_2, x_3}

In[42] := wa = Apply[Plus, r]
Out[42] = x_1+x_2+x_3
```

{x2, x3, x1}

```
In[43] := seki = Apply[Times, r – RotateLeft[r]]
Out[43] = (x_1-x_2)(x_2-x_3)(x_3-x_1)

In[44] := rt[n_] := (r = Table[x_i, {i, n}];
                     Apply[Plus, r]/[Times, r – RotateLeft[r]])

In[45] := rf[n_] := Module[{s}, Table[xi, {i, n}];
                     Apply[Plus, s]/[Times, s – RotateLeft[s]])
```

１．５　書き換えルールの動的作成

評価を速くする方法の1つ　＝　計算した値を記憶
　⇒　SetDelayed関数書き換えルールの評価中に
　　Set書き換えルールを作る
f[arg_] := f[arg] = right-hand side

```
In[46] := fib[0] := 0
          fib[1] := 1
          fib[n_] := fib[n] = fib[n-1] + fib[n-2]
In[47] := fib[3]
Out[47] = 2
In[48] := ?fib

In[49] := fib[0] := 0
          fib[1] := 1
          fib[2] := 1
          fib[3] := 2
          fib[n_] := fib[n] = fib[n-1] + fib[n-2]
```

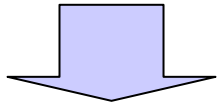
１．６　書き換えルールの順序

(1) ユーザ定義のルールが組込みルールより優先
(2) より特殊なルールが一般的なルールより優先

```
In[50] := f[x_] := x^2
In[51] := f[x_Integer] := x^3

In[52] := ?f
Global‘f
f[x^_Integer] := x^3
f[x_] := x^2

In[53] :=f[6.]
Out[53] = 36.

In[54] := f[6]
Out[54] = 216
```

２．変換ルール

書き換えルールを大域ルール・ベースに置くより
　１つの（内部で使われる）特定な式に適用させたいとき



Replace関数をRule, RuleDelayed関数と一緒に使い
⇒　適用される式の直後に局所的書き換えルール

２．１　Rule関数（->）

Rule［左辺，右辺］
表記法　　式/．左辺 -> 右辺
　　または　　式/．{左辺１ -> 右辺１，左辺２ -> 右辺２，・・}
左辺：　シンボル，数，name_
左辺，右辺が両方とも評価される
/．　ルールを式に当てはめるコマンド

```
In[55] := Table[x, {4}]/. x -> (-1)^Random[Integer]
Out[55] = {1, 1, 1, 1}

In[56] : = Trace[Table[x, {4}]/. x -> (-1)^Random[Integer]]
Out[56] = {{Table[x, {4}], {x, x, x, x}},
           {{{Random[Integer], 0}, (-1)^0, 1}, x -> 1},
           {x, x, x, x} /. x -> 1, {1, 1, 1, 1}}
```

x = 1で固定

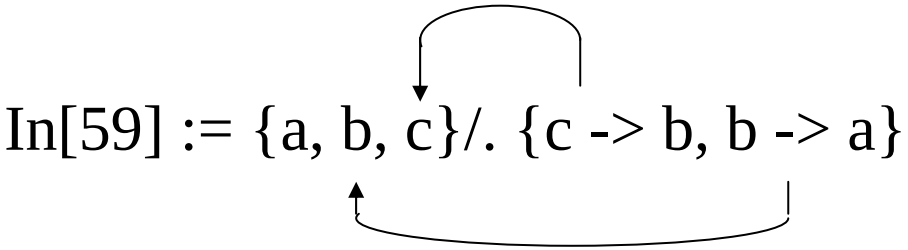
２．２　RuleDelayed関数（:>）

RuleDelayed［左辺，右辺］
表記法　　式/．左辺 :> 右辺
右辺は式に代入されるまで評価されない

In[57] := Table[x, {4}]/. x :> (-1)^Random[Integer]
Out[57] = {-1, 1, 1, -1}

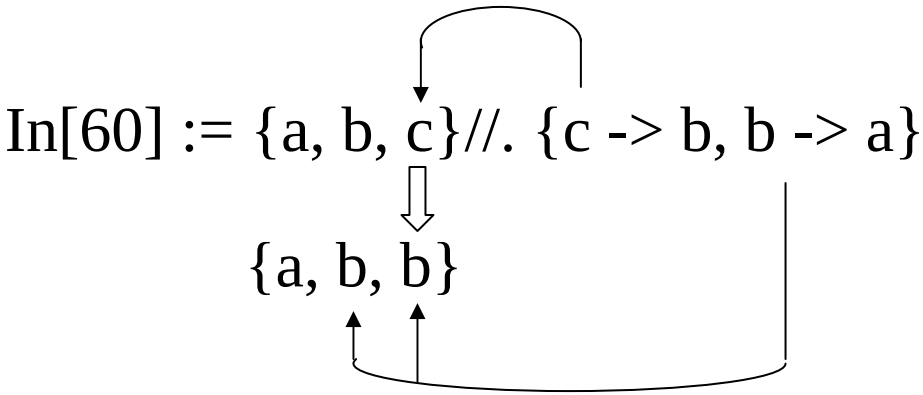
In[58] : = Trace[Table[x, {4}]/. x :> (-1)^Random[Integer]]
Out[58] = {{Table[x, {4}], {x, x, x, x}},
{{x, x, x, x}/. x:>$(-1)^{\text{Random[Integer]}}$, {$(-1)^{\text{Random[Integer]}}$,
$(-1)^{\text{Random[Integer]}}$, $(-1)^{\text{Random[Integer]}}$, $(-1)^{\text{Random[Integer]}}$},
{Random[Integer], 1}, $(-1)^1$, -1}
{Random[Integer], 0}, $(-1)^0$, -1}
{Random[Integer], 0}, $(-1)^0$, -1}
{Random[Integer], 1}, $(-1)^1$, -1}{-1, 1, 1, -1}}

２．３　変換ルールの繰返し適用



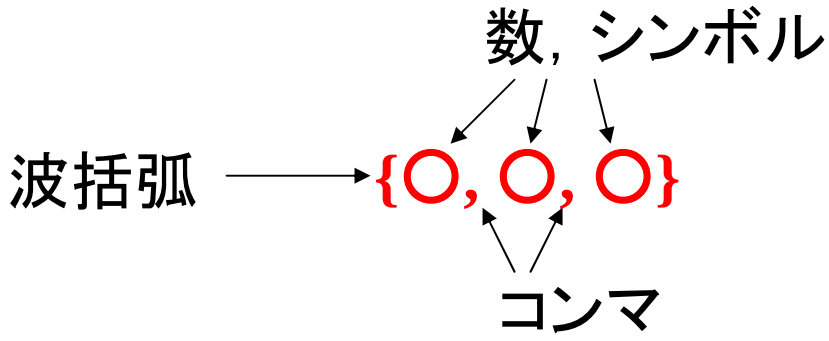
Out[59] = {a, a, b}

ReplaceRepeated関数



Out[60] = {a, a, a}

リスト



Length リストの長さ(要素の個数）

[[3]] リストの要素を表す

リストの番号

First 先頭の要素，　**Last** 最後の要素

{　, {　}} ネスト(リストのリスト，入れ子）

```
In[61] := moji = {1, c, f, 3, a, b}
Out[61]= {1, c, f, 3, a, b}
In[62] := Length[moji]
Out[62] = 6
In[63] := moji[[2]]
Out[63] = c
```

```
In[64] := nlst = {moji, {x, y}}
Out[64] = {{1, c, f, 3, a, b}{x, y}}
In[65] := nlst[[2]]
Out[65] = {x, y}
In[66] := nlst[[2, 1]]
Out[66] = x
```

リストの２番目の要素の１番目

```
In[67] := seisuu = {2, 6}; kigou = {x, y}; teisu = {0, e};
In[68] := combi = {seisuu, {kigou, teisuu}}
Out[68] = {{2, 6}, {{x, y}, {0, e}}
In[69] := FullForm[seisuu]
Out[69]//FullForm =
          List[2, 6]
In[70] := TreeForm[combi]
Out[70]//TreeForm =
```

個々のサブリスト

Range

Range[n] 1~ n の整数のリスト

Range[n, m] m ~ n の整数のリスト

Range[n, m, s] m ~ n, s ステップの整数のリスト

Table

Table[n(n+1), {n, 6}]

第1引数：式　第2引数：反復子，変数，$1 \le n \le 6$

$\{n, 2, 6\}$　$2 \le n \le 6$

$\{n, 2, 6, 2\}$　$2 \le n \le 6$ ステップ2

In[71] := **Table**[0, {6}]
Out[71] = {0, 0, 0, 0, 0, 0}

In[72] := **Table**[i-j, {i, j}, {j, 4}]
Out[72] = {{0, -1, -2, -3}, {1, 0, -1, -2}, {2, 1, 0, -1}}

In[73] := moji = {a, b, c, d, e, f}; **Range**[6]
Out[73] := {1, 2, 3, 4, 5, 6}

Take　指定した個数の要素を取出す
Part　指定した要素のみを取出す
Drop　指定した個数の要素を取り除いて残った要素のリスト
Append, Prepend, Insert　リストに要素を付け加える
Transpose　転置行列

```
In[74] := moji = {a, b, c, d, e, f}

In[75] := Take[moji, 3]
Out[75] = {a, b, c}

In[76] := Take[moji, {2, 5}]
Out[76] = {b, c, d, e}

In[77] := Take[moji, -4]
Out[77] = {c, d, e, f}

In[78] := Part[moji, {2, 5}]
Out[78] = {b, e}
```

In[79] := **Drop**[moji, 3]
Out[79] = {d, e, f}

In[80] := **Drop**[moji, {2, 4}]
Out[80] = {a, e, f}

In[81] := **Append**[moji, x]
Out[81] = {a, b, c, e, d, f, x}

In[82] := **Prepend**[moji, x]
Out[82] = {x, a, b, c, e, d, f}

In[83] := **Insert**[moji, x, 3]
Out[83] = {a, b, x, c, e, d, f}

In[84] := retsu = {suu, moji}
Out[84] = {{1, 2, 3, 4, 5, 6}, {a, b, c, d, e, f}}

In[85] := retsu//**MatrixForm**

Out[85]//**MatrixForm** =

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ a & b & c & d & e & f \end{pmatrix}$$

In[86] := tenchi = **Transpose**[retsu]

Out[86] = {{1, a}, {2, b}, {3, c}, {4, d}, {5, e}, {6, f}}

In[87] := tenchi//**MatrixForm**

Out[87]//**MatrixForm** =

$$\begin{pmatrix} 1 & a \\ 2 & b \\ 3 & c \\ 4 & d \\ 5 & e \\ 6 & f \end{pmatrix}$$

Flatten　リストのネストを取り除き，平らな（１重の）リストにする
Partition　指定した個数のサブリストのリストを作る
Depth　深さを知る
Dimensions　それぞれのレベルにおける要素の個数

In[88] := taira = **Flatten**[retsu]
Out[88] = {1, 2, 3, 4, 5, 6, a, b, c, d, e, f}

In[89] := **Partition**[taira, 3]
Out[89] = {{1, 2, 3}, {4, 5, 6}, {a, b, c}, {d, e, f}}

In[90] := nesuto = **Partition**[%, 2]
↳ リスト　⇒　さらに分割
Out[90] = {{{1, 2, 3}, {4, 5, 6}}, {{a, b, c}, {d, e, f}}}

In[91] := **Depth**[nesuto]
Out[91] = 4　　← 3行の場合　{{{

In[92] := **Dimensions**[nesuto]
Out[92] = {2, 2, 3}

In[93] := **Transpose**[nesuto, 1, 3, 2]

サブリストの中だけの置換. 2レベルと3レベルの入れ換え

Out[93] = {{{1, 4}, {2, 5}, {3, 6}}, {{a, d}, {b, e}, {c, f}}}

In[94] := **Dimensions**[%]

Out[94] = {2, 3, 2}

レベルを指定して部分的にネストをはずす

In[95] := **Flatten**[nesuto, 1]

Out[95] = {{{1, 2, 3}, {4, 5, 6}}, {{a, b, c}, {d, e, f}}}

In[96] := **Dimensions**[%]

Out[96] = {4, 3}

In[97] := **Partition**[taira, 5, 3]

送り要素数3

Out[97] = {{1, 2, 3, 4, 5}, {4, 5, 6, a, b}, {a, b, c, d, e}}

要素数5，6は省かれる

In[98] := **Partition**[taira, 5, 3, {1, 1}]

Out[98] = {{1, 2, 3, 4, }, {4, 5, 6, a, b},
{a, b, c, d, e}, {d, e, f, 1, 2}}

taira

1, 2, 3, 4, 5, 6, a, b, c, d, e, f

nesuto

Transpose

1	4	a	d
2	5	b	e
3	6	c	f

RotateRight リストの長さを変えずに，要素を右へ順送り
RotateLeft 左へ順送り
Sort 要素を標準的な順に並べかえる

In[99] := **RotateRight**[taira]
Out[99] = {f, 1, 2, 3, 4, 5, 6, a, b, c, d, e}

In[100] := **RotateRight**[taira, 3]
Out[100] = {d, e, f, 1, 2, 3, 4, 5, 6, a, b, c}

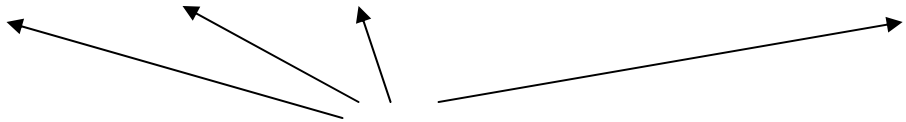
In[101] := **RotateRight**[taira, -1] または **RotateLeft**[taira]
Out[101] = {2, 3, 4, 5, 6, a, b, c, d, e, f, 1}

In[102] := bara = {b, e, 2, 4, a, 6, 3, c, f, 1, d, 5}
Out[102] = {b, e, 2, 4, a, 6, 3, c, f, 1, d, 5}

In[103] := **Sort**[bara]
Out[103] = {1, 2, 3, 4, 5, 6, a, b, c, d, e, f}

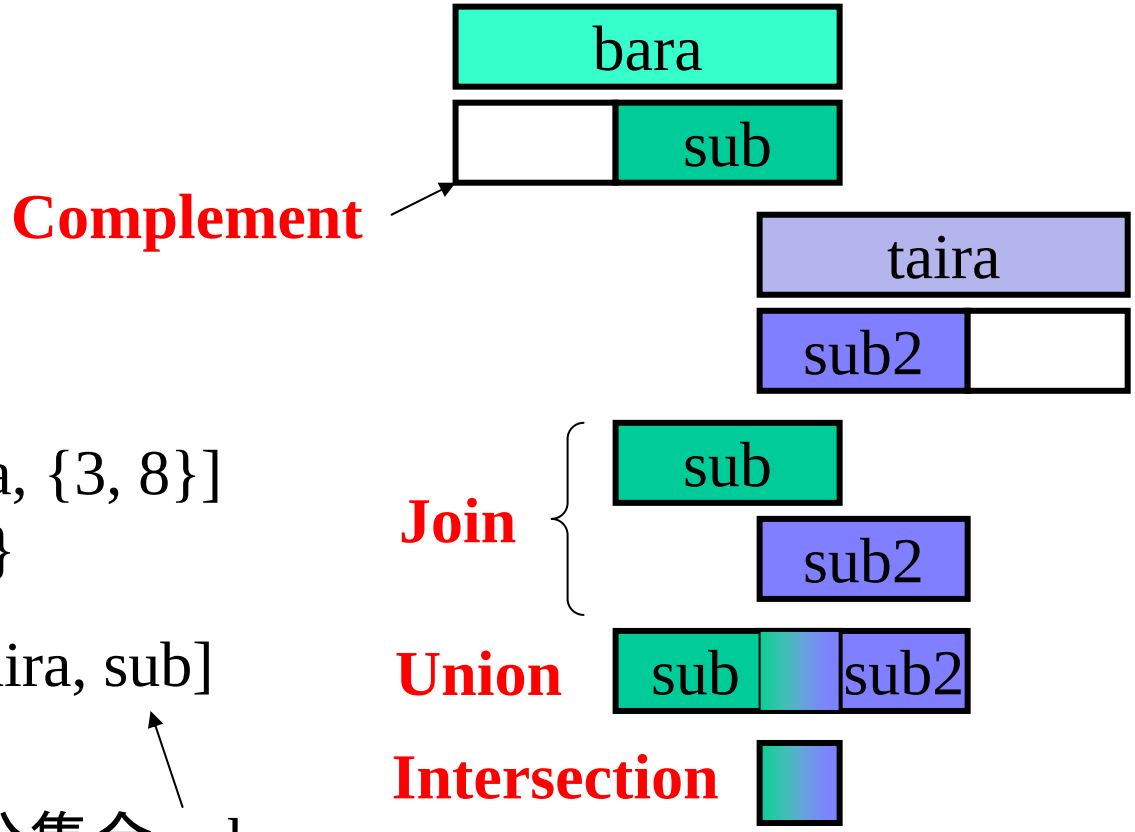
In[104] := **Partition**[bara, 2]
Out[104] = {{b, e}, {2, 4}, {a, 6}, {3, c}, {f, 1}, {d, 5}}

In[105] := **Sort**[%]
Out[105] = {{2, 4}, {3, c}, {a, 6}, {b, e}, {d, 5}, {f, 1}}



サブリストの場合，先頭の要素によってソート

Complement
Join
Union
Intersection



In[106] := sub = **Take**[bara, {3, 8}]
Out[106] = {2, 4, a, 6, 3, c}

In[107] := **Complement**[taira, sub]

集合 taira の部分集合 sub

Out[107] = {1, 5, 6, d, e, f}

In[108] := sub2 = **Take**[taira, {3, 8}]
Out[108] = {3, 4, 5, 6, a, b}

In[109] := **Join**[sub, sub2]
Out[109] = {2, 4, a, 6, 3, c, 3, 4, 5, 6, a, b}

In[110] := **Union**[sub, sub2]　または　sub⋃sub2
Out[110] = {2, 3, 4, 5, 6, a, b, c}

In[111] := **Intersection**[sub, sub2]　または sub⋂sub2
Out[111] = {3, 4, 6, a}

無名関数（anonym function），純関数（pure function）

ユーザ定義関数を作ることができ，使う前に名前をつけたり，入力したりすることなく，“その場（on the spot）”で使うことができる．

Function[x, body]　または　Function[{x, y, ･･}, body]

#, #1, #2 : 変数シンボル　　**(　)&[ ]**

In[112] := square[x_] := x^2　⟹　(#^2)&

In[113] := (#^2)&[5]
Out[113] = 25

In[114] := (#1^#2)&[5, 3]
Out[114] = 125

× In[115] := (#1^#2)&[{2, 3}]

○ In[116] := (#[[1]]^[[#2]])&[{2, 3}]
Out[116] = 8

高次関数

高次関数：　関数を引数としてとったり，結果として返す関数
Apply　　式のヘッドをすげ替える
Map　　関数の写像.
　　リスタブルでない関数をリスタブルにし，各要素に作用
MapThread　　ネストしたリストの各要素に作用
Listable　　ユーザ定義関数をリスタブルにする

```
In[117] := risuto = {a, b, c}; risuto// FullForm
Out[117]// FullForm =
                    List[a, b. c]

In[118] := Apply[Plus, risuto]// FullForm
Out[118] = FullForm =
                    Plus[a, b, c]

In[119] := Apply[Plus, risuto]
Out[119] = a + b + c
```

```
In[120] := Apply[Plus, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Out[120] = 55

In[121] := Apply[f, {{a, b}, {c, d}}]
Out[121] = f [{a, b}, {c, d}]

In[122] := Apply[f, {{a, b}, {c, d}}, 2]
Out[122] = {f [a, b], f [c, d]}
```

f がリスタブルでないと

```
In[123] := f [{a, b, c}]
Out[123] = f [{a, b, c}]
```

← 何もなしで返す

```
In[124] := Map[f [{a, b, c}]]
Out[124] = {f [a], f [b], f [c]}
```

ネストしたリスト構造に対して，Mapは外側のリスト，内側のリストまたは両者に適用

```
In[125] := Map[g, {{a, b}, {c, d}, {e, f}}]
Out[125] = {g[{a, b}], g[{c, d}], g[{e, f}]}
```

レベル2に適用

```
In[126] := Map[g, {{a, b}, {c, d}, {e, f}}, {2}]
Out[126] = {{g[a], g[b]}, {g[c], g[d]}, {g[e], g[f]}}
```

2回適用

```
In[127] := Map[g, {{a, b}, {c, d}, {e, f}}, 2]
Out[127] = {g[{g[a], g[b]}], g[{g[c], g[d]}], g[{g[e], g[f]}]}

In[128] := Map[Apply[g, #]&, {{a, b}, {c, d}, {e, f}}]
Out[128] = {g[{a, b}], g[{c, d}], g[{e, f}]}
```

In[129] := **MapThread**[g{{a, b, c}, {x, y, z}}]
Out[129] = {g[a, x], g[b, y], g[c, z]}

リストの対応した要素を順に並んだ組に組分け

In[130] := **MapThread**[**List**, {{a, b, c}, {x, y, z}}]
Out[130] = {{a, x}, {b, y}, {c, z}}

組込み関数はListable属性を持つ

In[131] := Log[{a, b, c, d}]
Out[131] = {Log[a], Log[b], Log[c], Log[d]}

ユーザー定義関数ff はリスタブルでない

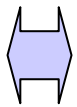
In[132] := ff [{a, b, c}]
Out[132] = ff [{a, b, c}]　←　何もなしで返す

ユーザー定義関数f をリスタブルにする

In[133] := Attributes[f] = Listable;

一方，ff に対してはMapを使い(In[119])

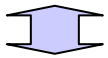
In[134] := f [{a, b, c}]
Out[134] = {f [a], f [b], f [c]}



In[135] := **Map**[ff, {a, b, c}]
Out[135] = {ff [a], ff [b], ff [c]}

リスタブルにしたf とリスタブルでないff の評価の比較

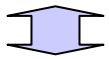
```
In[136] := f [{{a, b}, {c, d}}]
Out[136] = {{f [a], f [b]}, {f [c], f [d]}}
```



```
In[137] := Map[ff, {{a, b}, {c, d}}, {2}]
Out[137] = {{ff [a], ff [b]}, {ff [c], ff [d]}}
```

同一解を与える

```
In[138] := f [{a, b}, {c, d}]
Out[138] = {f [a, c], f [b, d]}
```



```
In[139] := Thread[ff [{a, b}, {c, d}]]
Out[139] = {ff [a, c], ff [b, d]}

In[140] := MapTread[ff, {{a, b}, {c, d}}]
Out[140] = {ff [a, c], ff [b, d]}
```

同一解を与える

Fold操作：　1つの関数，値，リストをとり，関数をその値に適用
⇒　その関数をその結果とリストの第一要素に適用
⇒　以下，続ける

Nest操作：　1つの関数を1つの値に適用
⇒　その関数をその結果に適用　⇒　以下，続ける

```
In[141] := Fold[Plus, 0, {a, b, c, d}]
Out[141] = a + b + c + d

In[142] := FoldList [Plus, 0, {a, b, c, d}]
Out[142] = {0, a, a + b, a + b + c, a + b + c + d}
```

In[143] := **NestList**[g, a, 4]　　aに対して g を4回
Out[143] = {a, g[a], g[g[a]], g[g[g[a]]], g[g[g[g[a]]]]}

In[144] := **NestList**[Sin, 0.7, 3]
Out[144] = {0.7, 0.644218, 0.600573, 0.565115}

In[145] := {0.7, Sin[0.7], Sin[Sin[0.7]], Sin[Sin[Sin[0.7]]]}
Out[145] = {0.7, 0.644218, 0.600573, 0.565115}

In[146] := **Nest** [g, a, 4]
Out[146] = g[g[g[g[a]]]]

FixedPointList, FixedPoint

Nest操作を止めるルール

(1)関数の適用が決められた回数

(2)結果が変わらなくなる

(3)ある述語条件が満たされる

```
In[147] := FixedPointList[Sin, 0.7, 3,
                          SameTest -> (#2 < 0.65 &)]
Out[147] = {0.7, 0.644218}

In[148] := FixedPointList[Sin, 0.7, 3,
                          SameTest ->( (#1 - #2) < 0.045 &)]
Out[148] = {0.7, 0.644218, 0.600573}
```

関数コールのネスト

ある引数の値を1つの関数に適用
⇒　その結果を別の関数に適用

```
In[149] := Tan[4.0]
Out[149] = 1.15782
```

```
In[150] := Sin[%]
Out[150] = 0.915931
```

```
In[151] := Cos[%]
Out[151] = 0.609053

In[152] := Cos[Sin[Tan[4.0]]]
Out[152] = 0.609053
```

無名関数のネスト

無名関数の形式 :　　(…#…)&　または　**Function**[var. body]

同一演算に対して色々な形式

```
In[153] := (Map[(#^2)&, #])& [{3, 2, 7}]
Out[153] = {9, 4, 49}

In[154] := Function[y, Map[Function[x, x^2], y]] [{3, 2, 7}]
Out[154] = {9, 4, 49}

In[155] := Function[y, Map [(#^2)&, y]][{3, 2, 7}]
Out[155] = {9, 4, 49}

In[156] := ( Map[Function[x, x^2], #])& [{3, 2, 7}]
Out[156] = {9, 4, 49}
```

関数入力のシンタックス

(1)通常:	関数名[引数]	Log[2]
(2)ポストフィックス記法:	引数//関数名	2//Log
(3)インフィックス記法:	引数～関数名～引数	

In[157] := 2//Log//Sin
Out[157] = Sin[Log[2]]

In[158] := Log[2]//N
Out[158] = 0.693147

In[159] := s = {a, b, c, d, e}; (sm = {s, s, s}//MatrixForm
Out[159]//MatrixForm =

$$\begin{pmatrix} a & b & c & d & e \\ a & b & c & d & e \\ a & b & c & d & e \end{pmatrix}$$

In[160] := n ~ Binomial ~ r
Out[160] = Binomial[n, r]

論理式，述語関数

論理式： 評価されると理論値 True, False を返す式

|| : Or, &&: And

[述語関数 …Q]

EvenQ : 引数が偶数か否か

PrimeQ : 素数を判定

MemberQ : $x \in S$ を判定（S：集合）

MatchQ : 表示（式）がパターンにマッチするか

In[161] := $2^{20} = = 16^5$

Out[161] = True

In[162] := 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = = 55

Out[162] = False

In[163] := $3^5 = = 5^3 \| 4^4 > 3^5$
Out[163] = True

In[164] := True && False
Out[164] = False

In[165] := **PrimeQ**[{101, 102, 103, 104, 105}]
Out[165] = {True, False, True, False, False}

In[166] := shugo = {a, b, c, d, e}; **MemberQ**[shugo, b]
Out[166] = True

乱数

Random[ ]	0.0 ~ 1.0 の実数乱数
Random[Real, {1., 2.}]	1.0 ~ 2.0の実数乱数
Random[Integer]	0 または1の乱数
Random[Integer, {0, 10}]	0 ~ 10の整数乱数
Random[Real, {range}, n]	有効数字 n桁

```
In[167] := Random[Integer, {1, 100}]
Out[167] = 85

In[168] := Table[Random[Integer, {10}]]
Out[168] = {0, 0, 0, 1, 1, 0, 0, 1, 0, 0}

In[169] := Random[Real, {1.1, 5.2}]
Out[169] = 2.35193

In[170] := Random[  ]
Out[170] = 0.0182791
```

厳密値，近似値

整数，分数，組込み値（π（Pi），e（E））：　厳密値を持つ
N　　：近似値に変換
Precision　：精度を調べる
Round　　：小数点第１位を四捨五入
Floor　　：小数点以下を切り捨て
Chop　　：数値計算誤差を切り捨て

```
In[171] := {Precision[N[e]], Precision[N[e, 50]], Precision[e]}
Out[171] = {16, 50, ∞}

In[172] := Round[{1.0, 1.2, 1.4, 1.6, 1.8, 2.0}]
Out[172] = {1, 1, 1, 2, 2, 2}

In[173] := Floor[{1.0, 1.2, 1.4, 1.6, 1.8, 2.0}]
Out[173] = {1, 1, 1, 1, 1, 2}
```

数の計算はなるべく厳密値を使い，近似値の変換は最後に

In[174] := Sin[100 N[π]]
Out[174] = 1.96439×10^{-15}　　$\sin 100\pi = 0$ であるが

In[175] := Chop[%]
Out[175] = 0

近似値と厳密値の計算　⇒　結果は近似値

In[176] := {15 + π, 15.2 + π, Sin[π/4], Sin[N[π/4]], e^2, $e^{2.0}$, 2.3^{π}}

Out[176] = {15 + π, 18.3416, $1/\sqrt{2}$, 0.707107, e^2, 7.38906, 13.6899}

データファイルの読み書き

>> （入力式 >> ファイル名）： ディレクトリ中にその名のファイルが作られ，計算結果が書きこまれる．新たな値が上書きされ，前のデータは消える．

>>> 新たな値が追記される

ReadList ファイルに書かれたデータを読み込む

!! ファイル名 外部ファイルの中身を知る

Export 表形式のデータをその形式でファイルに書き出す

Import 表形式のデータを読み込む

```
In[177] := Log[2.0] >> suu.dat

In[178] := Log[2.0] >>> suu.dat
```

データを数値として扱う

```
In[179] := data = ReadList[“suu.dat”, Number]
Out[179] = {0.693147, 0.788457}
```

```
In[180] := Export[“hyo.dat”, {{1.2, 3.4, 2.5}, {3.4, 5.3},
                              {-1.4, 3.6, 2.6}}, “Table”]
Out[180] = hyo.dat
```

第3引数. 表形式を指定

```
In[181] := !! hyo.dat
            1.2   3.4   2.5
            3.4   5.3
           -1.4   3.6   2.6

In[182] := Import[“hyo.dat”, “Table”]
Out[182] = {{1.2, 3.4, 2.5}, {3.4, 5.3}, {-1.4, 3.6, 2.6}}
```

グラフの表示

Plot	:	１変数関数 f(x) のグラフを描く
ParametricPlot	:	パラメータ表示の曲線を描く
ListPlot	:	データ点をプロットする
Show	:	図を表示する
Plot3D	:	３次元グラフィックス表示
DensityPlot	:	密度プロット
ContuerPlot	:	等高線プロット

```
In[183] := zu = Plot[ x (x – 5)^2, {x, -0.5, 7}]
```

関数　　$-0.5 \le x \le 7$

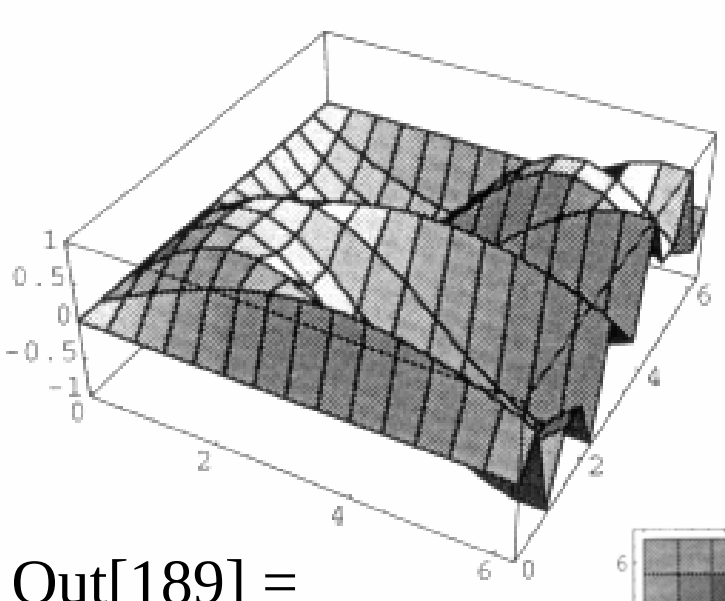
```
In[184] := ParametricPlot[{Sin[t], Sin[2t]}, {t, 0, 2π}]
In[185] := ten = {19, 10, 1.5, 20.3, 2.4, -1.5, 12.3};
In[186] := ListPlot[ten]
In[187] := ListPlot[ten, PlotJoined -> True]
```

点を直線で結んだグラフ

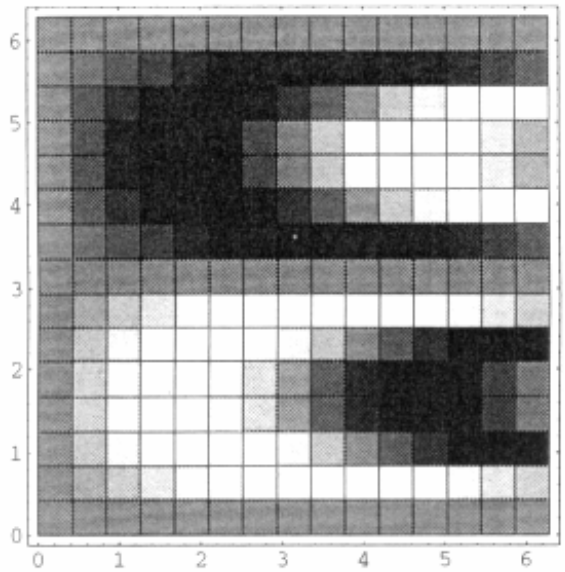
２つの図を表示

```
In[188] := Show[zu, %187]
```

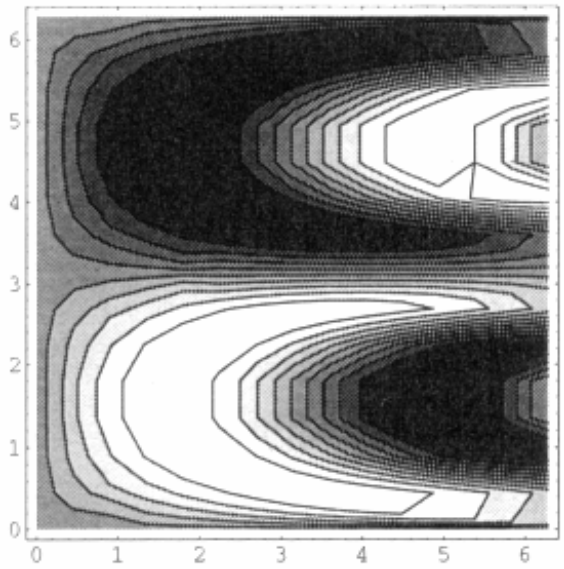
In[189] := Plot3D[Sin[xSin[y]], {x, 0, 2π}, {y, 0, 2π}]
In[190] := DensityPlot[Sin[xSin[y]], {x, 0, 2π}, {y, 0, 2π}]
In[191] := ContouerPlot[Sin[xSin[y]], {x, 0, 2π}, {y, 0, 2π}]



Out[189] =
-SurfaceGraphics-

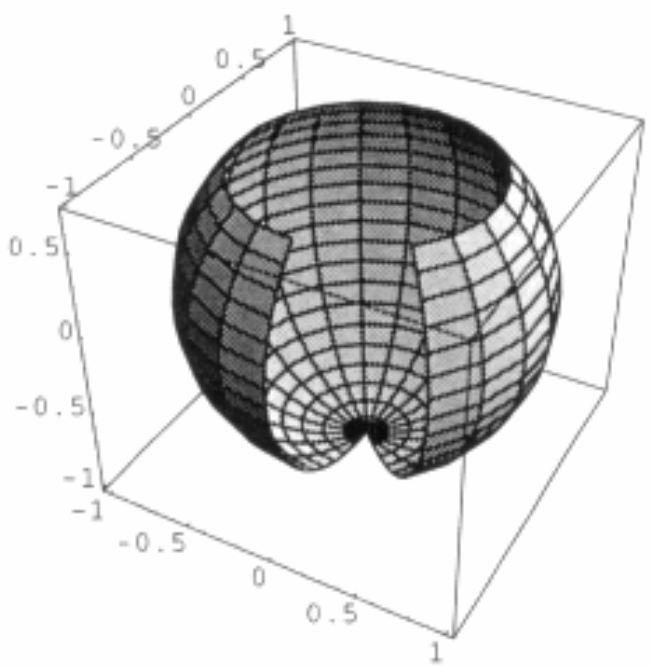


Out[190] =
-DensityGraphics-



Out[191] =
-ContourGraphics-

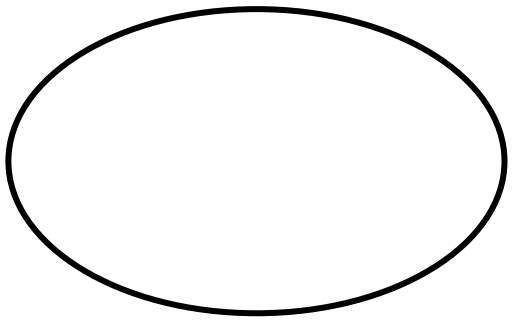
```
In[192] := ParametricPlot3D
           [{Sin[u]Cos[v], Sin[u]Sin[v], Cos[u]},
            {u, π/4, π}, {v, -π/4, 3π/2}

Out[192] = -Graphics3D-
```



プリミティヴ ： 図形の構成要素
Circle ： 円の組込み関数．中心の座標，半径
Graphics ： プリミティヴからグラフィックス・オブジェクトを作る
Show ： グラフィックス・オブジェクトを表示
AspectRatio ： 縦横比を指定．Automaticで1:1にする
（デフォルト値は1/黄金分割比）

```
In[193] := grf = Graphics[Circle[{1, 0}, 2]]
Out[193] = - Graphics –
In[194] := Show[grf]
In[195] := grf2 =
           Show[grf, AspectRatio -> Automatic, Axes -> Automatic]
```

Out[194] = -Graphics-



Out[195] = -Graphics-