

付録B.経験的Mathematica入門

松田裕幸 matsuda@symbolics.jp (2011.5.6)

Mathematicaはプログラミング言語

Mathematicaは超高級電卓だという話を聞いていた。しかし、面白そうなので『Mathematicaクックブック』を眺めるとなんか雰囲気が違う。Mathematicaっていったい何者？

Mathematicaはプログラミング言語の1つです。JavaやC++などと比べるとあり得ない機能をたくさん持ち、表記も一風変わっています。しかし、プログラミング言語です。

プログラミング言語というからにはそれを使ってプログラムを書きます。では、Mathematicaで書くプログラムとはどんなものでしょう？

はじめに確認しておかなければならないことは、Mathematicaではプログラムはすべて式の集まりであり、Mathematicaにおいては式は関数と等価です。通常、プログラムで式というと、こんな感じのものを思い浮かべるかもしれません。

$$3 - \sqrt{4}$$

Mathematicaでも同じように書けますが、この式を関数を使って書き直すと次のようになります。

Subtract[3, Sqrt[4]]

これを分解してみましょう。

Subtract	関数名Subtract(引き算)
[	普通関数というと()が一般的だがMathematicaでは[]を使用
3,	関数Subtractの第1引数
Sqrt	関数Subtractの第2引数、同時に関数Sqrt(平方根)
[	
4	関数Sqrtの引数
]	
]	

注：関数の中にまた関数がある場合、内側から計算していきます。

一般的にプログラムは「実行する」と言われます。しかし、Mathematicaのプログラムは式あるいは関数のため、実行するとは言わず、「評価する」(evaluate)といいます。ではさっそくこの式を評価したいのですが、その前にMathematicaにおけるプログラミング環境について確認しておく必要があります。拡張子 *.nbをもつこのファイルはノートブックと呼ばれます。そして画面右端にみえる  マークで囲まれた単位をセルと呼びます。(セルのマークはノートブックでは見えますが、PDFでは見えません。)

これで準備が整いました。Mathematicaの式があるセルをクリックし、Shift+Enterキーを押してください(Shiftキーを押したままEnterキーを押します)。

In[1]:= **Subtract[3, Sqrt[4]]**

Out[1]= 1

式の左側に現れたラベルが入力と出力を表します。In[]が評価前の式を表し、Out[]が評価結果を表します。当然、最初にした式も同じように評価できます。

```
In[2]:= 3 -  $\sqrt{4}$ 
```

```
Out[2]= 1
```

これをプログラムと言われても違和感を感じるかもしれません。どうみても電卓と同じじゃないか、どこがプログラミング言語なのか？何万行ものMathematicaプログラムなんてあるのか？当然の疑問です。でも答えはYESです。

通常は自分で書いた関数（方法は後述）が数行から数十行。そうした関数を使って、また別の関数を作り、などしていくと、あっというまに数百行に達します。MathematicaはOSや他の言語とのインタフェースも充実していますので、Mathematicaでシステム寄りのプログラムを書こうとすると自然、規模が大きくなっていきます。

Mathematicaの組み込み関数はすべて大文字から始まっているのですぐにわかります。逆に、ユーザが書く関数はこれと区別するために通常小文字で始めるのが基本です。また、すべての関数名はそれ自身が関数の意味を表すよう長い名前が推奨されています。

MathematicaはC++などやJavaなどと異なりインタプリタ言語です。したがって、プログラミング第1歩に登場する"Hello,World"を作る時もmain関数などの必要ありません。

```
In[3]:= Print["Hello,World"]
```

```
Hello,World
```

記号計算

ちょっと他の言語では表現できないものの1つが記号計算です。たとえばC言語で次の計算は可能でしょうか？変数に値が入ってないと計算できず、エラーとなるはずです。

```
In[4]:= a + 3
```

```
Out[4]= 3 + a
```

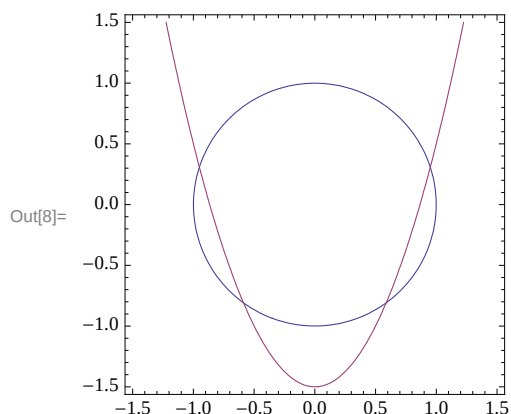
Mathematicaでは変数（正確にはシンボル）は変数のまま式の中に書けます。その一方、そのシンボルに対する値が与えられたときは式全体を評価し値を持つことができます。"/."は置換の意味。"a->9"は/左辺にあるaすべてを9で置き換えます（後述）。

```
In[5]:= a + 3 /. {a -> 9}
```

```
Out[5]= 12
```

Mathematicaはこの能力を使うことで初等数学の問題を解くことができます。（関数の説明は省略します。大まかにいうと、2つの方程式を使い、グラフを描き、方程式を解くことで交点を求め、今度はその交点をグラフ上に加えている、が分かれば十分です。）

```
In[6]:= exp1 = x2 + y2 == 1;
exp2 = y - 2 x2 + 3 / 2 == 0;
ContourPlot[Evaluate@{exp1, exp2}, {x, -1.5, 1.5}, {y, -1.5, 1.5}, ImageSize -> 200]
```

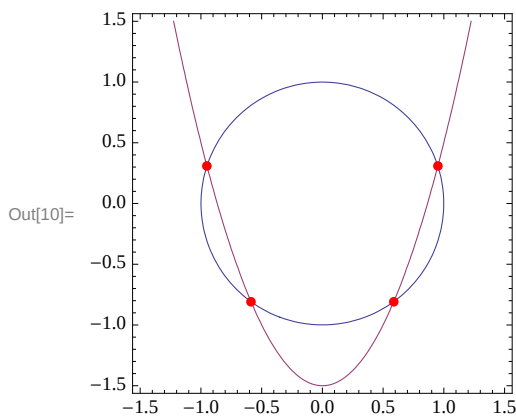


In[9]:= **pts = Solve[exp1 && exp2, {x, y}]** (*方程式を解く *)

Out[9]= $\left\{ \left\{ y \rightarrow \frac{1}{4}(-1 - \sqrt{5}), x \rightarrow -\frac{1}{2}\sqrt{\frac{1}{2}(5 - \sqrt{5})} \right\}, \left\{ y \rightarrow \frac{1}{4}(-1 - \sqrt{5}), x \rightarrow \frac{1}{2}\sqrt{\frac{1}{2}(5 - \sqrt{5})} \right\}, \right.$
 $\left. \left\{ y \rightarrow \frac{1}{4}(-1 + \sqrt{5}), x \rightarrow -\sqrt{\frac{5}{8} + \frac{\sqrt{5}}{8}} \right\}, \left\{ y \rightarrow \frac{1}{4}(-1 + \sqrt{5}), x \rightarrow \sqrt{\frac{5}{8} + \frac{\sqrt{5}}{8}} \right\} \right\}$

解いた答えを座標として赤点で描きます。円(exp1)と放物線(exp2)が交差する場所が連立方程式の解になっていることがわかります。

In[10]:= **Show[ContourPlot[Evaluate@{exp1, exp2}, {x, -1.5, 1.5}, {y, -1.5, 1.5}], Graphics[{Red, PointSize[Medium], Point[{x, y} /. pts]}], ImageSize -> 200]**



変数、シンボル、代入、束縛

通常のプログラミング言語では

x = 1

と書けば、変数xに値1を代入すると読みます。さらに具体的にいえば、変数xに代表されるメモリ番地に値1を格納することを意味します。

一方、同じ式のMathematicaの解釈は「シンボルxに値1を束縛する」です。変数と比較してみると、シンボルxはどこかの番地を「値」として持ち、その番地に1が格納されている、となります。なんか昔、アセンブラでやった間接参照みたいですね。

ただしMathematicaは特に指定しない限り、シンボルは登場するたびに評価される（つまり値が取り出される）ので、番地が顔をのぞかせることは通常はありません。

In[11]:= **x = 1**

Out[11]= 1

In[12]:= **y = x; (* xが参照するメモリの中身をyが参照するメモリの場所にコピーする *)**
x = 2;

y (* したがってxの値が変化してもyの値は変化しない *)

Out[14]= 1

次の例を見ると、=を代入ではなく束縛と呼ぶほうが相応しいとわかります。

In[15]:= **r := RandomReal[];**
{r, r, r, r}

Out[16]= {0.381909, 0.390127, 0.910411, 0.50408}

この:=（遅延束縛 late binding）を=に置き換えて再度実行してみます。

```
In[17]:= r = RandomReal[];
         {r, r, r, r}

Out[18]:= {0.988953, 0.988953, 0.988953, 0.988953}
```

Mathematicaではすべての評価はある種の「置き換え」になっています。シンボルが登場するとそのシンボルに束縛されている「もの」と置き換えます。これが評価の意味です。r=RandomReal[]の場合、1未満の乱数を1つ求めそれをシンボルrに束縛しました。したがって、そのrを4つ並べるとすべて同じ値になるのは当然です。

では、r:=RandomReal[]のように:=の場合はどうなるでしょう。この場合、シンボルrにはRandomReal[]を評価せず、そのまま記号として束縛させます。つまり、この段階でシンボルrに束縛されている値は RandomReal[]そのものです。そこで、別の場所でrが登場すると、rを評価、つまりrに束縛されたものを取り出し、その値を求めます。{r,r,r,r}の場合、rが登場するたびにRandomReal[]を計算するので、異なる値が並びます。繰り返しになりますが、最初の例の場合、{r,r,r,r}は次のものと等価です（ただし、乱数計算なので個々の値は異なりますから、相互に値が異なる点だけを確認してください）。

```
In[19]:= {RandomReal[], RandomReal[], RandomReal[], RandomReal[]}

Out[19]:= {0.972079, 0.144552, 0.610171, 0.331097}
```

他のプログラミング言語と異なり、Mathematicaではシンボルと値とを結びつける働きを=が行っています。したがって、この結びつきを解消することも可能となります。普通、変数に代入された値を消す（ゼロじゃないですよ）ことはできるでしょうか？出来ません。しかし、Mathematicaはできます。

```
In[20]:= y

Out[20]:= 1

In[21]:= Clear[y]; y

Out[21]:= y
```

関数Clearは引数にとるシンボルに対する束縛を解きます。シンボルyを評価して、シンボルそのものが返ってくる、のはシンボルに束縛されている値がないためです。その一方、こんなものもあります。

```
In[22]:= Remove[y]
```

こうするとMathematicaのシンボル空間の中からyというシンボルそのものを消し去ります。

```
In[23]:= Clear[x, y, r]
```

オブジェクトとリスト

Mathematicaにとってすべての計算対象はオブジェクト呼ばれます。オブジェクトはオブジェクト指向のオブジェクトではなく、関数引数の要素、あるいは関数の計算結果になるという意味でオブジェクトです。たとえば、次の例で関数Importは画像ファイルを取り込むのですが、取り込んだ画像ファイルというオブジェクトがシンボルiに束縛されます。

そして以降、このiに束縛された画像がオブジェクトとして操作されるのを見ていきます。

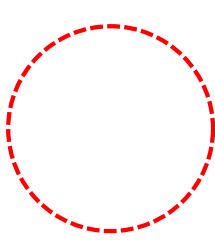
```
In[24]:= i = Import["ExampleData/lena.tif"]
```

```
Out[24]=
```



Mathematicaではデータの基本はオブジェクトですが、複数のオブジェクトを構成するにはリストを使います。もちろん、リストもオブジェクトですので、リストのリストも当然ありえます。リストは集合と異なり、異なるデータタイプが混在できます。(ここで、Graphics、Circleは関数です。Red,Thick, Dashedは関数Circleに対する属性、ImageSizeはグラフィックスの大きさを指定するオプションです。)

```
In[25]:= List[3.14, ImageMultiply[i, 3.14],  
Graphics[{Red, Thick, Dashed, Circle[]}], ImageSize -> 100]]
```

```
Out[25]= {3.14, , 
```

あるいは次のように記述します。ここに限らず、Mathematicaは同じことを表現するのに複数の表現を持ちます。どれを使うかは使う時の状況、使用者の好みに依存します。

```
In[26]:= {3.14, i, ImageMultiply[i, 3.14]}
```

```
Out[26]= {3.14, , 
```

```
In[27]:= ImageDimensions[i] (* 画像の大きさを縦横のピクセル数で与える *)
```

```
Out[27]= {150, 116}
```

Mathematicaではオブジェクトを関数によって直接操作し、結果も直接見ることができます。

```
In[28]:= ImageTake[i, 50]
```

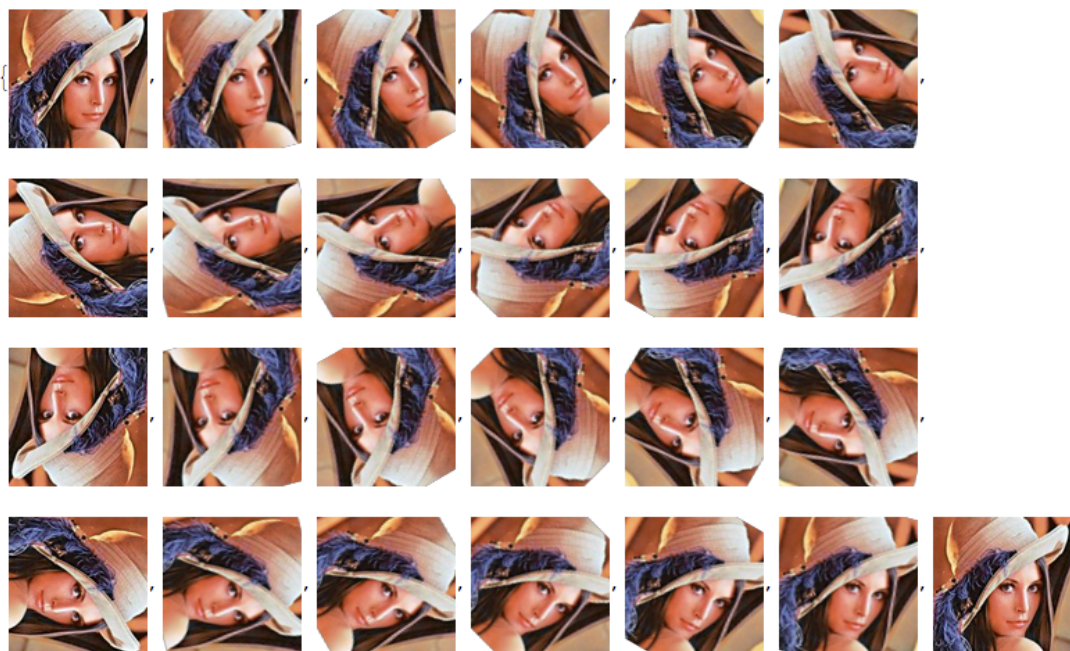
```
Out[28]= 
```

```
In[29]:= ImageRotate[i, Right]
```

```
Out[29]= 
```

リストデータを作るにはすでに見たように要素を直接指定する以外に、計算によって作り出すこともできます。次の例は、関数Tableを使い、画像を15°ずつずらしたもののリストを作り出しています。Table内でImageRotate関数を複数回実行し、それぞれの結果からなる画像のリストを用意します。ImageRotateの中のTableパラメータは θ で、これを0から 2π まで $\pi/12$ 刻みに増やしています。一方、ImageRotateは元の画像 i を θ 分回転しますが、その際、回転後の画像の大きさを{100,100}に指定し、かつ、標準で残る黒い背景をTransparentで消しています。

```
In[30]:= m = Table[ImageRotate[i,  $\theta$ , {100, 100}, Background -> Transparent], { $\theta$ , 0., 2 Pi, Pi / 12}]
```



```
Out[30]=
```

実行可能なハイパーリンク構造を持ったヘルプ

すでに書きましたようにMathematicaには4,000近い関数があります。当然、そのすべてを覚えることはできないし、かつ、覚えていても細かい使い方までは覚えていません。そこでノートブックから興味ある関数を見つける方法をお教えします。?でパターンにマッチする関数名を探します。次の例では関数名に Rotate を含むものをすべて列挙しています。概略を知りたい関数をクリックします。するとオレンジ線の下に関数引数と簡単な説明がでできます。そして一番最後の >>をクリックするとHelpのドキュメントに移動します。そこから関連する項目に移動することも可能です。

```
In[31]:= ? *Rotate*
```

▼ System`

ImageRotate	ImageRotated	Rotate	RotateLabel	RotateLeft	RotateRight
-------------	--------------	--------	-------------	------------	-------------

他のプログラミング言語のHelpと最も異なる点は**Help**画面もノートブックで出来ているので、そこで各関数を評価し様々なテストが可能になっていることです。これは極めて凄いことです。Help画面の中で、プログラムも実行できるなんて、他の言語ではありえるのでしょうか？実は本当のことをいえば、そういう言語もなくはありませんが、ノートブックという形でプログラム開発環境と透過な形で実現しているのはMathematicaだけだと思います。もし、同じようなものが世の中に存在するのであればぜひ筆者までお知らせください。

リスト操作

この回転画像のリストmの要素数は

```
In[32]:= Length[m]
```

```
Out[32]= 25
```

あるいは

```
In[33]:= m // Length
```

```
Out[33]= 25
```

です。さらには

```
In[34]:= Length@m
```

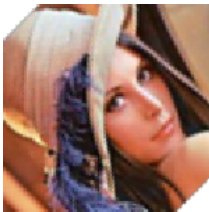
```
Out[34]= 25
```

なんて芸当も可能です。これはどれがベストな表記ということではなく、プログラム中の位置、あるいはプログラミングの状況に応じて自然と使い分けられていきます。

リスト中の特定の要素を取り出すにはPart関数を使います。ちなみに、多くのプログラミング言語と違い、リストを配列を見なすとindexは0からではなく1から始めます。本来多くのプログラミング言語がindexを0から始めたのは計算効率の理由が1番でそれ以上の理由は無く、0から始めることによって得られるメリットより危険性の方が大きいので、Mathematicaの姿勢は正しいと思います。

```
In[35]:= Part[m, 4]
```

```
Out[35]=
```



では最初の3つ、あるいは最後の2つとかを取り出すには2つの方法があります。

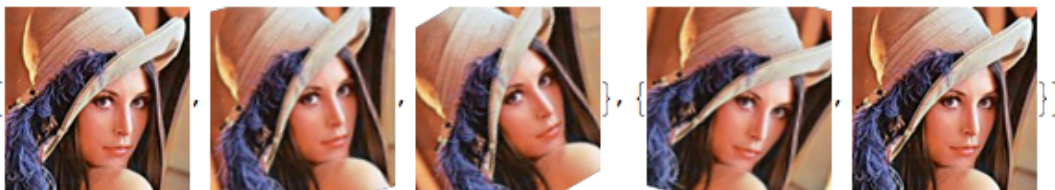
```
In[36]:= {Take[m, 3], Take[m, -2]}
```

```
Out[36]= {{
```



```
In[37]:= {m[[1 ;; 3]], m[[-2 ;; -1]]} (* [[ ]]はPart関数 *)
```

```
Out[37]= {{
```



Mathematicaのプログラミングに慣れるには、基本的なリスト構成関数、リスト操作関数をしっかり学ぶことをお勧めします。『Mathematicaクックブック』でよく取り上げられているものとして次のものがあります。

```
In[38]:= L = RandomInteger[{1, 10 000}, {20}] (* 1から10.000までの整数乱数20個のリスト *)
```

```
Out[38]= {3957, 8219, 5940, 5405, 9291, 2853, 633, 9049, 7436,
6804, 3606, 3413, 9700, 6415, 9815, 7310, 4459, 4065, 8463, 6059}
```

In[39]:= ? Table

Table[*expr*, {*i*_{max}}] *expr* の *i*_{max} 個のコピーのリストを作成する。
 Table[*expr*, {*i*, *i*_{max}}] *i* が 1 から *i*_{max} までの場合の *expr* の値のリストを作成する。
 Table[*expr*, {*i*, *i*_{min}, *i*_{max}}] *i* = *i*_{min} から開始する。
 Table[*expr*, {*i*, *i*_{min}, *i*_{max}, *di*}] *di* をステップとして使用する。
 Table[*expr*, {*i*, {*i*₁, *i*₂, ...}}] 連続する値 *i*₁, *i*₂, ... を使用する。
 Table[*expr*, {*i*, *i*_{min}, *i*_{max}}, {*j*, *j*_{min}, *j*_{max}}, ...]
 ネストしたリストを与える。 *i* に対応するリストが最も外側になる。 >>

In[40]:= ? Partition

Partition[*list*, *n*] *list* を重複しない長さ *n* のサブリストに分割する。
 Partition[*list*, *n*, *d*] *d* をオフセットとするサブリストを作成する。
 Partition[*list*, {*n*₁, *n*₂, ...}] ネストされたリストを *n*₁ × *n*₂ × ... のサイズのブロックに分割する。
 Partition[*list*, {*n*₁, *n*₂, ...}, {*d*₁, *d*₂, ...}] *list* でレベル *i* でオフセット *d*_i を使う。
 Partition[*list*, *n*, *d*, {*k*_L, *k*_R}] *list*
 の最初の要素が最初のサブリストの *k*_L の位置に、 *list* の最後の要素が最後のサブリストの
 *k*_R の位置またはそれ以降に現れることを指定する。要素の追加が必要な場合、Partition は
 list を循環的に扱い、充填する。
 Partition[*list*, *n*, *d*, {*k*_L, *k*_R}, *x*] 必要なら要素 *x* を反復して充填する。
 Partition[*list*, *n*, *d*, {*k*_L, *k*_R}, {*x*₁, *x*₂, ...}] 必要なら要素 *x*_i を循環的に反復し充填する。
 Partition[*list*, *n*, *d*, {*k*_L, *k*_R}, {}] パディングしないので異なった長さのサブリストを与えることがある。
 Partition[*list*, *nlist*, *dlist*, {*klist*_L, *klist*_R}, *padlist*] ネストされたリストで整形およびパディングを指定する。 >>

In[41]:= Partition[L, 4]

Out[41]= {{3957, 8219, 5940, 5405}, {9291, 2853, 633, 9049},
 {7436, 6804, 3606, 3413}, {9700, 6415, 9815, 7310}, {4459, 4065, 8463, 6059}}

In[42]:= ? ListConvolve

ListConvolve[*ker*, *list*] 核 *ker* と *list* のたたみ込みを形成する。
 ListConvolve[*ker*, *list*, *k*] *ker* の *k* 番目の要素が *list* の各要素と揃えられる循環たたみ込みを形成する。
 ListConvolve[*ker*, *list*, {*k*_L, *k*_R}] 最初の要素が
 list[[1]] ker[[*k*_L]] を含み、最後の要素が list[[-1]] ker[[*k*_R]] を含む循環たたみ込みを形成する。
 ListConvolve[*ker*, *list*, *klist*, *p*] *list* の最後が要素 *p* の反復で充填されるような、たたみ込みを形成する。
 ListConvolve[*ker*, *list*, *klist*, {*p*₁, *p*₂, ...}] *list* の最後が *p*_i で循環反復充填されるたたみ込みを形成する。
 ListConvolve[*ker*, *list*, *klist*, *padding*, *g*, *h*] *g*
 が Times の代りに、 *h* が Plus の代りに使われる一般化されたたたみ込みを形成する。
 ListConvolve[*ker*, *list*, *klist*, *padding*, *g*, *h*, *lev*] *ker* と *list* でレベル *lev* の要素を使い、たたみ込みを形成する。 >>

In[43]:= ListConvolve[{2, 3}, {a, b, c, d, e, f}]

Out[43]= {3 a + 2 b, 6 + 3 b, 9 + 2 d, 3 d + 2 e, 3 e + 2 f}

In[44]:= ? Select

Select[*list*, *crit*] crit[*e*_i] が True となる *list* のすべての要素 *e*_i を取り出す。
 Select[*list*, *crit*, *n*] crit[*e*_i] が True となる最初から *n* 個の要素を取り出す。 >>

In[45]:= Select[L, PrimeQ] (* 素数だけを取り出す *)

Out[45]= {8219, 9049, 3413}

In[46]:= ? Cases

Cases[{ $e_1, e_2, \dots$ }, *pattern*] パターンにマッチする要素 e_i をリストにして返す.
 Cases[{ $e_1, \dots$ }, *pattern* → *rhs*] パターンにマッチし, 要素 e_i に対応する *rhs* の値をリストにして返す.
 Cases[*expr*, *pattern*, *levelspec*] *levelspec*
 によって指定され, パターンにマッチするレベルにある式 *expr* のすべての部分をリストにして返す.
 Cases[*expr*, *pattern* → *rhs*, *levelspec*] パターンにマッチする *rhs* の値を返す.
 Cases[*expr*, *pattern*, *levelspec*, *n*] *expr* の中でパターンにマッチする最初の *n* 個の部分を返す. >>

In[47]:= Cases[L, a_ /; a > 8000] (* Lの中の要素で8000より大きな数のリスト。文法の意味は後述 *)

Out[47]= {8219, 9291, 9049, 9700, 9815, 8463}

In[48]:= ? Count

Count[*list*, *pattern*] *list* において *pattern* にマッチする要素の数を与える.
 Count[*expr*, *pattern*, *levelspec*] 式 *expr* の *levelspec*
 で指定されたレベルで適用され, *pattern* にマッチする部分式の数を与える. >>

In[49]:= Count[L, a_ /; a > 8000]

Out[49]= 6

関数型制御構造

Mathematicaは一般的なプログラミング言語が持つ制御構造をより高度に、より見やすくした表記をいくつも持っています。そしてこうした表記を使うと、Do、For、Whileなどより計算速度が高速になるというメリットもあります。（この付録Bを読んだあと、もし関数型の仕組みに興味をもたれたらぜひ付録Aも読んでみてください。）

■ Nest, NestList

In[50]:= a = x;
Do[a = f[a], {5}];
a

Out[52]= f[f[f[f[f[x]]]]]

In[53]:= Nest[f, x, 5]

Out[53]= f[f[f[f[f[x]]]]]

In[54]:= NestList[f, x, 5]

Out[54]= {x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]], f[f[f[f[x]]]]}

In[55]:= Nest[Sqrt, 2, 4]

Out[55]= $2^{1/16}$

■ FixedPoint, FixedPointList

以下の例は具体的な値無しに計算させるのが難しいので、評価させていません。

```
a = x;
While[a ≠ f[a], a = f[a]];
a

FixedPoint[f, x]
```

Fold, FoldList

```
In[56]:= a = x;
         Do[a = f[a, i], {i, 1, 3}];
         a

Out[58]= f[f[f[x, 1], 2], 3]

In[59]:= Fold[f, x, {1, 2, 3}]

Out[59]= f[f[f[x, 1], 2], 3]
```

■ Map

```
In[60]:= l = {};
         data = {10, 20, 30};
         Do[AppendTo[l, f[data[[i]]], {i, 1, 3}];
         l

Out[63]= {20, 40, 60}

In[64]:= Map[f, data]

Out[64]= {20, 40, 60}

In[65]:= Map[f, data] /. {f -> Sqrt}

Out[65]= {20, 40, 60}
```

ルール型プログラミング

実はMathematicaでは関数型プログラミングよりルール型プログラミングの方が難しいかもしれません。関数型はあくまで関数型の制御構造のみを理解すればいいのに、ルール型プログラミングに現れるパターンと実際の働きとの連関を掴むのが難しいからです。

しかし、すでに見た例から1つだけ取り上げ、説明してみます。

```
In[66]:= Count[L, a_ /; a > 8000]

Out[66]= 6

In[67]:= count = 0;
         Do[If[L[[i]] > 8000, count++], {i, Length[L]}];
         count

Out[69]= 6
```

`_`はパターンを表します。この場合、`L`の任意の要素を対象とし、`a_`とすることでそれに名前（シンボル）`a`を付与します。そして`/;`で条件を追加します。これを一般的なプログラミング言語風に書いてみると上のようになります。

MathematicaのHelpドキュメントには膨大な解説 `guide/RulesAndPatterns` が掲載されています。

＊ここまで読まれてきたにぜひ、確認していただきたいことがあります。あえてFortran風表記とMathematica風表記を併用させていただきましたが、Mathematica風表記が「恰好いい」から、そちらを使いなさい、ではないということです。もちろん、記述が簡潔な方が見やすいし、間違いも少なくなります。しかし、本当の理由は性能です。

Mathematicaのようなインタプリタ言語の場合、コンパイラ言語と異なりシンボル（変数）への参照は常にメモリ参照を引き起こし、速度に対し致命的です。

つまりMathematicaで少しでも速いプログラムを書くための鉄則の1つとして、不必要にシンボルを使わないことです。上記Fortran風コードを見てお分かりのように、多くの変数が使われています。これがコンパイラ言語ならば賢いコンパイラがこうした変数をすべてレジスタ参照に置き換えてくれるので、まったく気にする必要はないのですが、Mathematicaはインタプリタ言語です。ですので、繰り返しの中に不必要に変数を介在させることは望ましくありません。

関数定義

さて最後に自前の関数を作ってみます。

```
In[70]:= Clear[f, a, b];
```

```
In[71]:= f[a_, b_] :=  $\sqrt{a^2 + b^2}$ 
```

```
In[72]:= f[3, 4]
```

```
Out[72]= 5
```

何か変な感じしませんか？特に従来のプログラミング言語しか知らないとなおさらかと思います。上記定義を、一般的な表記に変えてみます。

$$f(a, b) = \sqrt{a^2 + b^2}$$

そう、こんな感じですよ。では、これに合わせて、上記定義を書き換えてみましょう。

```
In[73]:= Clear[f];
```

```
f[a, b] =  $\sqrt{a^2 + b^2}$ 
```

```
Out[74]=  $\sqrt{a^2 + b^2}$ 
```

ありゃ、何か変です。でも構わず先に行ってみます。

```
In[75]:= f[3, 4]
```

```
Out[75]= f[3, 4]
```

やっぱり変でした（笑）。前者と後者には変更点が2箇所（正確には3箇所）あります。1つは:=についてです。Mathematicaではシンボルの評価は「置き換え」だと書きました。関数の参照はまさに置き換えです。そのため、通常、関数を定義する際には、関数名とその引数に本体を束縛させます。したがって、最低限、次のような形に戻します。

```
In[76]:= Clear[f];
```

```
f[a, b] :=  $\sqrt{a^2 + b^2}$ 
```

それでも結果は変わりません。

```
In[78]:= f[3, 4]
```

```
Out[78]= f[3, 4]
```

どこがまずいのでしょうか。Mathematicaは数値のみならずシンボルも扱えることを思い出してください。そして、関数引数の場所では、代入という作業ではなく、パターンマッチングという作業が起きることを覚えてください。つまり、引数にa, bとあるということは関数fを呼び出す側にもaとbが必要だということです。

```
In[79]:= f[a, b]
```

```
Out[79]=  $\sqrt{a^2 + b^2}$ 
```

確かにうまく行きました。しかし、本来は任意の数をマッチさせたかったのに。どうすればいいんでしょうか？

Mathematicaには便利な記号があります。つまり、任意のものをマッチさせる記号があります。それが_です。

```
In[80]:= Clear[f];
```

```
f[_ , _] :=  $\sqrt{a^2 + b^2}$  ;  
f[3, 4]
```

```
Out[82]=  $\sqrt{a^2 + b^2}$ 
```

今度は確かに引数3と4はマッチしたようです。が、答えは望むものではありません。マッチさせるだけでなく、マッチさせたものをなんらかの方法で保持させなければなりません。そのため、マッチさせた後、保持させるために_の前にシンボル名を置きます。そして関数本体ではこのシンボル名を使い計算を行うようにすればいいわけです。

```
In[83]:= Clear[f];
```

```
f[a_ , b_] :=  $\sqrt{a^2 + b^2}$  ;  
f[3, 4]
```

```
Out[85]= 5
```

■ 局所変数

関数本体が大きくなってくると、一時的に使いたい変数が出てきます。そうした変数を気楽に使っているとよそに悪さを及ぼします。

```
In[86]:= c = 3;
```

```
f[a_ , b_] := (c =  $\sqrt{a^2 + b^2}$  ; c * c);  
f[3, 4]
```

```
Out[88]= 25
```

```
In[89]:= c
```

```
Out[89]= 5
```

さて何が問題でしょう？関数fの外にある変数cの値は元々3でした。それが関数fを使ったからといって突然5に変わってしまうのは嬉しくないはずです。なぜこんなことが起きたのでしょうか？関数fの中で一時変数cを無造作に使ったためです。そうか、では、絶対他では使いそうにない名前を付ければいいのではないだろうか？たとえば、ccccccとか。しかし、プログラム規模が大きくなればなるほど、未使用の確認は大変になってきます。また、後の修正でたまたま同じ名前の変数を使ってしまう可能性も出てきます。つまり、特殊な名前を使うアイデアは神経を使うだけ使って結局は問題を回避出来ません。

究極の解は、一時的に使う、つまり局所的に使う変数を指定する方法を使うことです。MathematicaにはそのためにModuleという機能が用意されています。

```
In[90]:= c = 3;
```

```
f[a_ , b_] := Module[{c =  $\sqrt{a^2 + b^2}$  }, c * c];  
f[3, 4]
```

```
Out[92]= 25
```

```
In[93]:= c
```

```
Out[93]= 3
```

複数定義？

一般的なプログラミング言語では同じ関数名に複数の定義を行うことは許されていません。しかし、Mathematicaはそれが可能です。これもパターンマッチングに関係します。Mathematicaではすべてのシンボルは評価において置き換えの対象になります。その置き換えが可能かどうか判断するためにパターンマッチングを使います。ただし、それが関数の場合は引数部分も含めてパターンと考えます。

すると次のケースはどうなるでしょう？関数名は同じ。そして引数パターンも同じ。ということは後から定義されたもの残り、前に定義されたものは消えます。

```
In[94]:= f[a_, b_] :=  $\sqrt{a^2 + b^2}$ 

In[95]:= f[a_, b_] := Module[{c =  $\sqrt{a^2 + b^2}$ }, c * c]

In[96]:= ? f
```

Global`f

```
f[a_, b_] := Module[{c =  $\sqrt{a^2 + b^2}$ }, c c]
```

次に同じ関数名だけど引数パターンが異なるものを追加したらどうなるでしょう？今度は引数パターンが異なるので、シンボルfには2つの定義が存在することになります。

```
In[97]:= f[x_] := x3

In[98]:= ? f
```

Global`f

```
f[a_, b_] := Module[{c =  $\sqrt{a^2 + b^2}$ }, c c]
```

```
f[x_] := x3
```

■ ルール型プログラミングの例

```
In[99]:= f[0] := 1;
          f[n_] := n * f[n - 1]

In[101]:= f[3]

Out[101]= 6
```

次の例では関数シンボル名も引数パターンもまったく同じです。ただし、引数に対する条件（/;以降に記載）が異なります。こうした条件もパターンマッチングの際、考慮されます。よってこれら3つの定義はパターンマッチングの観点から正確に区別されます。

```
In[102]:= Clear[f];
          f[a_] := a * 2 /; a > 0
          f[a_] := a / 2 /; a < 0;
          f[a_] := 0 /; a == 0

In[106]:= f[-9]

Out[106]= -  $\frac{9}{2}$ 

In[107]:= ? f
```

Global`f

$$f[a_] := a^2 /; a > 0$$

$$f[a_] := \frac{a}{2} /; a < 0$$

$$f[a_] := 0 /; a == 0$$

なお、この例は次のように書くこともできます。

$$\text{In[108]:= } f[a_ /; a > 0] := a * 2$$

おまけ

ノートブック上では多くの表記が使えます。以下主なものを、その表現とキーボード入力を併記します。詳細は以下の Help ドキュメントを参照してください。

tutorial/TwoDimensionalExpressionInputOverview

Mathematica メニューの [パレット] から。当該項目上にマウスカーソルを乗せるとショートカットキーの構成が浮かび上がってきます。時に、「:」という記号が出ることがありますが、これは Esc に対応します。

$M[[]]$	(* M Esc [[Esc Esc]] Esc, M[[]] もあり *)
$A \rightarrow B$	(* A -> <space> B , A ->B *)
$A \Rightarrow B$	(* A :> <space> B , A :> B *)
$A == B$	(* A == <space> B , A == B *)
$A \neq B$	(* A != <space> B , A != B *)
$\frac{A}{B}$	(* A Ctrl+/ B Ctrl+<space>, A/B *)
$\frac{(A+1)^3}{B}$	(* (A+1) Ctrl+6 3 Ctrl+<space> Ctrl+/ B Ctrl+<space>, (A+1)^3/B *)
$\sqrt{A+B}$	(* Ctrl+2 A+B Ctrl+<space>, Sqrt[A+B] *)
$\sum_{t=1}^{10} t^2$	(* Esc sumt Esc, Sum[t^2, {t,1,10}] *)
$\int x \, dx$	(* Esc intt Esc, Integrate[x,x] *)
θ	(* Esc theta Esc *)
$\alpha_1 + \beta_1$	(* Esc alpha Esc Ctrl+- 1 Ctrl+<space> + Esc beta Esc Ctrl+- 1 Ctrl+<space> *)