関数定義とプログラミング

■ 関数定義

■ 基本的な関数定義

関数 $f(x) = x^2 + x + 1$ を定義するには，次のように入力する．

```
In[1]:= f[x_] := x^2 + x + 1
```

この定義式の意味は見てのとおり， f[x] を x^2+x+1 と定義するというものであるが，いくつかの注意がある．まず，定義式の左辺の x には，ブランクと呼ばれる記号 "_" （アンダーバー）が付いている．このブランクは何にでもマッチする記号であり，これを付けることによりこの f の引数にどのようなものがきても，この関数定義が有効となる．次に，関数定義には遅延割り当てと呼ばれる := が用いられていることである．

正確には， x にブランク _ が付いているのではなく，ブランク _ に x という名前がついているのである．また，関数定義には必ず遅延割り当て := を用いる必要はなく，即時割り当て = を用いるべき場合もある．たとえば上の例の場合には即時割り当てを用いた方がよいが，初心者はすべて遅延割り当てを用いた方が無難である．

この関数は普通の関数と同じように用いることができる．

```
In[2]:= f[a]

Out[2]= 1 + a + a^2

In[3]:= f[2]

Out[3]= 7
```

入力した関数の定義を解除するには Clear を用いる．特に，間違った定義をした後で再定義を行うときには，その前に以前の定義を解除しておくべきである．さらに安全を期すためには，関数定義を行うときには，その前に必ず Clear を用いるようにするとよい．

```
In[4]:= Clear[f]

In[5]:= f[2]

Out[5]= f[2]
```

■ 場合分けのある関数定義

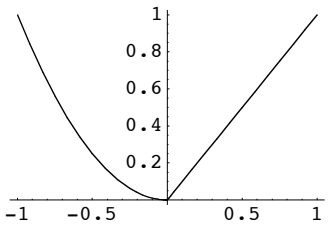
引数の値によって定義式が異なる関数 $g(x) = \begin{cases} x & (x \geq 0) \\ x^2 & (x < 0) \end{cases}$ を定義する．定義するための入力は，数学式とほとんど同じであり，遅延割り当てを用いた定義式の後に，スラッシュセミコロン "/;" に続いて，その定義式が適用されるための条件を書く．

```
In[6]:= Clear[g]

In[7]:= g[x_] := x /; x >= 0
        g[x_] := x^2 /; x < 0
```

この関数のグラフをプロットして，きちんと定義ができているかを確認してみる．

```
In[9]:= Plot[g[x], {x, -1, 1}]
```




```
Out[9]= - Graphics -
```

次に， x が 0 のときには 0， x が 0 以外の偶数のときには 2，x が奇数のときには 1,である関数 h(x) を定義する.

```
In[10]:= Clear[h]

In[11]:= h[0] := 0
         h[x_] := 2 /; EvenQ[x]
         h[x_] := 1 /; OddQ[x]
```

この定義において， x の値が 0 のときには，一番目の定義式 h[0]:=0 が適用されるので，2番目の定義式の条件に「0以外の」という条件を付ける必要はない.

```
In[14]:= Table[h[x], {x, -5, 5}]

Out[14]= {1, 2, 1, 2, 1, 0, 1, 2, 1, 2, 1}
```

練習問題： 実数 x の絶対値を返す関数 abs[x] を，組み込み関数 Abs を用いないで定義せよ.

■ 定義の優先順位

関数がどのように定義されているかを確認するには， ? に続いて関数名を入力すればよい.

```
In[15]:= ? h

        Global`h

         h[0] := 0

         h[x_] := 2 /; EvenQ[x]

         h[x_] := 1 /; OddQ[x]
```

一行目の Global`h はこの関数がユーザーが定義した関数であることを示している．この関数のように，定義式が複数個ある場合には，上から順に定義式が適用できるかどうかを見ていき，一番最初に適用できたものが，関数の定義となる．この優先順序はユーザーが関数定義を入力した順序にほぼ一致するが，異なる場合もある．それは，定義の左辺にある引数に x_ ではなく，具体的な値を指定したものは，優先順位が高くなる.

```
In[16]:= h[1] := -1
```

```
In[17]:= ?h

         Global`h

          h[0] := 0

          h[1] := -1

          h[x_] := 2 /; EvenQ[x]

          h[x_] := 1 /; OddQ[x]

In[18]:= Table[h[x], {x, -5, 5}]

Out[18]= {1, 2, 1, 2, 1, 0, -1, 2, 1, 2, 1}
```

また，この関数 h の定義では， h[1/2] のようなものは定義されていない．このような未定義の場合には，そのままの式を返す．

```
In[19]:= h[1 / 2]

Out[19]= h[1/2]
```

■ 引数の型指定

処理可能な関数の引数の型を限定したり，引数の型に応じて異なる定義式を用いたいときには，引数のブランク "_" に続いて型を書くことにより，引数の型を指定することができる．

```
In[20]:= Clear[f]

In[21]:= f[x_Integer] := x
         f[x_Rational] := x + 1
         f[x_Real] := x + 2
```

このように定義すると，引数が整数，有理数，実数の場合には，それぞれの定義が適用される．

```
In[24]:= f[2]

Out[24]= 2

In[25]:= f[1 / 2]

Out[25]= 3/2

In[26]:= f[1.2]

Out[26]= 3.2
```

しかし，引数が複素数の場合については定義されていないので，関数は入力された式そのままを返す．

```
In[27]:= f[2 + 3 I]

Out[27]= f[2 + 3 i]
```

■ 再帰的定義

ある関数を定義するのに，その関数自身を用いることを，再帰的定義と呼ぶ．再帰的定義を行うときに注意することは，再起呼び出しが無限に繰り返されることがないように，再起呼び出しの終了条件をきちんと定義しておくことである．再帰的定義で階乗 n! = 1×2×...×n を返す関数 fac[n] を定義してみる．0!=1, n! = (n-1)! × n (n>0) であることを用いると，次のような定義ができる．

```
In[28]:= Clear[fac]

In[29]:= fac[0] := 1
         fac[n_] := fac[n - 1] n /; n > 0

In[31]:= fac[100]

Out[31]= 933262154439441526816992388562667004907159682643816214685929638952175999932299156089
          414639761565182862536979208272237582511852109168640000000000000000000000000
```

次にもう一つ，$a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$ $(n \geq 3)$ で定義されるフィボナッチ数列を再帰的に定義してみる．

```
In[32]:= Clear[a]

In[33]:= a[1] := 1
         a[2] := 1
         a[n_] := a[n - 1] + a[n - 2] /; n >= 3

In[36]:= Table[a[n], {n, 10}]

Out[36]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

この計算の方法だと， a[n] を値を計算するために a[n-1] と a[n-2] の値を求める必要がある．したがって， a[n] の計算時間は a[n-1] の計算時間と a[n-2] の計算時間の和になる．このことを，Timing を用いて確認してみる．

Timing[expr] は expr の計算にかかった時間と，計算結果をリストにして返す．

```
In[37]:= Timing[a[20]]

Out[37]= {0.065698 Second, 6765}

In[38]:= Timing[a[21]]

Out[38]= {0.107099 Second, 10946}

In[39]:= Timing[a[22]]

Out[39]= {0.171976 Second, 17711}

In[40]:= Timing[a[23]]

Out[40]= {0.278454 Second, 28657}

In[41]:= Timing[a[24]]

Out[41]= {0.451313 Second, 46368}
```

このぶんだと， a[30] を計算するのに，1 時間以上かかることになる．

■ 計算した値を記憶する

さきほどの計算方法で a[30] の計算に 1 時間もかかる理由は，計算した値を忘れ去り二度と使わないことにある．そこで，一度計算した値は記憶しておくようにして，次に計算が要求されたときには，以前に計算した値を返すようにする．

f[x_]:= (f[x] = ...) は，関数 f[x] の値を一度目に計算するときには ... を定義式として計算し，その値を f[x] の値に割り当てる．二度目以降は，その割り当てられた値が優先されるので， ... が再び計算されることはない．

```
In[42]:= Clear[a]

In[43]:= a[1] := 1
         a[2] := 1
         a[n_] := (a[n] = a[n - 1] + a[n - 2]) /; n >= 3

In[46]:= a[4]

Out[46]= 3
```

今，上のように定義をやり直した後， a[4] の値を計算した．その後で， a の定義がどのようになっているかを確認してみる．

```
In[47]:= ?a

        Global`a

         a[1] := 1

         a[2] := 1

         a[3] = 2

         a[4] = 3

         a[n_] := (a[n] = a[n - 1] + a[n - 2]) /; n ≥ 3
```

このように， a[3], a[4] には値が割り当てられている．したがって，これ以降は a[3], a[4] の値を求めるのに， a[n] = a[n - 1] + a[n - 2] の計算式が用いられることはない．この計算方法で，a[30] の計算に何分かかるかを試してみる．

```
In[48]:= Timing[a[30]]

Out[48]= {0.000809 Second, 832040}
```

■ 手続きとしての関数

関数は値を返すだけでなく，大域変数に対して何かの操作を施すという手続きとして定義することもできる．次の関数 ab[x] は，大域変数 a に引数 x の値を加え，大域変数 b から引数 x の値を減じて，それらの結果を表示する，という操作を施す手続きである．

複数の実行文をまとめるには，セミコロン ";" で区切った実行文を括弧 "(", ")" で括る．

```
In[49]:= Clear[ab]

In[50]:= ab[x_] := (a += x; b -= x; Print["a=", a]; Print["b=", b])
```

ここで a += x は a = a + x と同で， b -= x は b = b - x と同じことである．

```
In[51]:= a = 0; b = 100;

In[52]:= ab[1]

        a=1

        b=99

In[53]:= ab[2]

        a=3

        b=97

In[54]:= ab[5]

        a=8

        b=92

In[55]:= {a, b}

Out[55]= {8, 92}
```

■ 純関数（匿名関数）

その場でしか使わないので，名前をつける必要がない関数を定義したいときには，純関数（匿名関数）と呼ばれるものを使う．

Function[body] または body & は， body で定義される純関数を表す． body の中では #1, #2, ... で引き数（#i が i 番目の引き数）を表す．ただし，引き数が 1 個の場合には， # でも表すことができる．

```
In[56]:= Function[# + Sin[#]]

Out[56]= #1 + Sin[#1] &

In[57]:= %[a]

Out[57]= 8 + Sin[8]

In[58]:= Function[#1 + Sin[#1 + #2]][a, b]

Out[58]= 8 + Sin[100]

In[59]:= Expand[#1^#2] &[a + b, 3]

Out[59]= 1000000
```

Function[{x1, x2, ...}, body] あるいは Function[x, body] は， x1, x2, ... あるいは x を引き数とし， body を本体とする純関数を表す．

```
In[60]:= Function[{x, y}, x + Sin[x + y]]

Out[60]= Function[{x, y}, x + Sin[x + y]]

In[61]:= %[a, b]

Out[61]= 8 + Sin[100]
```

純関数はリストに関数を施したいときによく用いる．たとえばリストの各要素 x に対して x+Sin[x] の値を計算したリストは， (#+Sin[#])& という純関数を Map させることにより得られる．

```
In[62]:= Map[(# + Sin[#]) &, {0.3, 0.4, 0.5}]

Out[62]= {0.59552, 0.789418, 0.979426}
```

■ プログラミング

■ 分岐

処理の流れを条件によって分岐させるには， If, Which, Switch を用いる.

- If[test, then, else] は test を評価してその結果が真であるときに then を，偽であるときに else を評価実行する.

```
In[63]:= If[E^Pi < Pi^E, 1, 2]

Out[63]= 2
```

- Which[test1, value1, test2, value2, ...] は， test1, test2, ... を順に評価し，初めて真になったのが testi であるとき valuei を返す．真になる testi がないときには，何も返さない.

```
In[64]:= Which[2 < 1, 1, 2 < 2, 2, 2 < 3, 3, 2 < 4, 4]

Out[64]= 3

In[65]:= Which[2 < 1, 1, 2 < 2, 2]
```

次の関数 f[x] は x<0 のとき -1 を，x=0 のとき 0 を， x >0 のとき 1 を返す.

```
In[66]:= Clear[f]

In[67]:= f[x_] := Which[x < 0, -1, x == 0, 0, x > 0, 1]

In[68]:= {f[-3], f[0], f[2]}

Out[68]= {-1, 0, 1}
```

Switch[expr, form1, value1, form2, value2, ...] は， expr を評価してその値を form1, form2, ... と順に比較し，最初にマッチしたものが formi であるとき valuei を返す.

次の関数 f[x] は x の値が 1 のときに 11 を， 2 のときに 22 を，Sin[何がし] のときに sinFunction を，それ以外のときに other を返す．ブランク "_" は何にでもマッチする記号なので，これを最後の選択肢に入れておくと，「それ以外の何でも」という条件になる.

```
In[69]:= Clear[f];
         f[x_] := Switch[x, 1, 11, 2, 22, Sin[_], sinFunction, _Integer, int, _, other]

In[71]:= {f[2], f[3], f[Sin[x]], f[Cos[x]]}

Out[71]= {22, int, sinFunction, other}
```

> 練習問題： 実数 x の符号（正のときは +1 ，負のときは -1， 0 のときには 0）を返す関数 sign[x] を，組み込み関数 Sign[x] を用いずに定義せよ.

■ 繰り返し

繰り返しを行うには Do, While, For などを用いる．これらの関数の body, start および incr にはセミコロンで区切ることにより複数の文を書くことができる．また test も複文にすることができるが，最後の文の真偽値が test の真偽値となる．

- Do[body, {i, imin, imax, di}] は， i を imin から imax まで di 刻みに動かし body を繰り返し評価実行する． imin, di を省略すると 1 となる．

```
In[72]:= s = 0;

In[73]:= Do[s += i; Print[s], {i, 10}]
        1
        3
        6
        10
        15
        21
        28
        36
        45
        55
```

- While[test, body] は test が真である限り body を繰り返し評価実行する．

```
In[74]:= a = 1;

In[75]:= While[a < 100, a *= 2; Print[a]]
        2
        4
        8
        16
        32
        64
        128

In[76]:= a

Out[76]= 128
```

- For[start, test, incr, body] は，最初に start を実行して初期化したのち，test が真である限り body と incr を繰り返し実行する．

```
In[77]:= For[s = 0; i = 0, i <= 100, i++, s += i]

In[78]:= s

Out[78]= 5050
```

■ 局所変数を用いたプログラミング

局所変数を用いた，複雑な関数や手続きを定義するには， Module を用いる．

- Module[{x, y, ...}, body] は局所変数 x, y, ... を備えた関数および手続きを定義する．body の一番最後で評価された式の値が，関数の値となる．

Module[{x=x0, y=y0, ...}, body] は局所変数に初期値を設定する．

次の関数 f[x] は 1 から x までの整数の和の二乗を返す．

```
In[79]:= Clear[f];

         f[x_] := Module[{a, s = 0}, For[a = 1, a <= x, a++, s += a]; s^2]

In[81]:= f[10]

Out[81]= 3025
```

この Module の中の a, s は局所変数なので， Module の外側にある大域変数とは関係ない．したがって， f を計算した後も大域変数の値は変化しない．

```
In[82]:= a = 5; s = 10;

In[83]:= f[5]

Out[83]= 225

In[84]:= {a, s}

Out[84]= {5, 10}
```

■ 演習問題

[1-1] 関数 f[x] を， x が 3 の倍数のときに 3 であり，x が 3 の倍数でなく 5 の倍数のときに 5 であり，それら以外のときに 0 であるものとして *Mathematica* で定義せよ．

[1-2] 西暦x年が閏年であるのは， x が 4 の倍数であり 100の倍数でないか，あるいは x が 400 の倍数であるとき，と定められている．西暦 x 年が閏年であるか否かを返す真偽値関数 f[x] を *Mathematica* で定義せよ．

[1-3] $a_1 = 1,\ a_n = n\,a_{n-1} + n^2 + 1\ \ (n \geq 2)$ で定義される数列の第 20 項の値を求めよ．

[1-4] データリスト $\{x_1, \ldots, x_n\}$ の平均と標準偏差を { 平均，標準偏差 } の形で返す関数を， Module を使って作れ．