20.7 関数の再帰的な定義

階乗 $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ を返す関数を定義してみましょう。このような関数を定義するには for 文を使って，掛け算を繰り返せばよいです：

- ファイル名：**kaijou1.py**

```
def kaijou(n):
    """ nの階乗n!を返す関数をfor文を使って定義する """
    a=1
    for i in range(1,n+1):
        a = a*i
    return a

for i in range(1,10):       #i=1,...,9に対して
    print kaijou(i),        #kaijou(i)の値を表示する
```

実行結果の例

```
1 2 6 24 120 720 5040 40320 362880
```

さて，関数 $f(n)$ を漸化式で

$$f(0) = 1, \quad f(1) = 1, \quad f(n) = n \cdot f(n-1)$$

で定義すると明らかに自然数 n に対して $f(n) = n!$ となります。このような関数の定義の仕方を**再帰的な定義** (recursive definition) といいます。Python では関数を再帰的に定義することができます。次のプログラムは階乗 $n!$ を再帰的に定義したものです：

- ファイル名：**kaijou2.py**

```
def kaijou(n):
    """ n!を再帰的に定義する"""
    if n==0 or n==1:
        return 1
    else:
        return n*kaijou(n-1)      # ここで自分自身の関数kaijou(n-1)呼び出す！！

for i in range(10):               #i=0,1,2,…,9に対して
    print kaijou(i),              #kaijou(i)をプリントする
```

実行結果の例

```
1 1 2 6 24 120 720 5040 40320 362880
```

ユークリッドの互除法はある種の手続きの繰り返しであることを利用して，最大公約数 gcd(a,b) を以下のように再帰的に定義することも出来ます：

- ファイル名：**gcd_rec.py**

```
def gcd(a,b):
    if b == 0:                  # もしb=0なら
        return a                # aを返す
    else:                       # そうでなければ
        return gcd(b,a%b)       # gcd(b,a%b)を返す

print gcd(1428,2618)
```

実行結果の例

```
238
```

上のプログラムでは，答えを得るまでに次のような動作が行われています：

1. a=1428, b=2618 として gcd(a,b) の中身を実行する。
2. $b = 0$ かどうかを調べたが，そうではないので gcd(2618,1428%2618) を返す。
3. 1428 ÷ 2618 の余りは 1428 なので gcd(2618,1428) を実行する。
4. 1428 は 0 ではないので，gcd(1428,2618%1428) が返される。
5. 2618%1428=1190 なので gcd(1428,1190) を実行する。
6. 1190 は 0 ではないので，gcd(1190,238) が返される。238=1428%1190。
7. 238 は 0 ではないので，gcd(238,1190%238) を返す。
8. 1190%238=0 と第 2 の変数が 0 になったので第 1 変数 238 を返す。
9. 返された値は print によって表示される。

このようにして，関数が繰り返しの動作によって定義可能な場合は，再帰的な定義によって非常に簡潔に関数を定義することが出来ます。

20.8 再帰的な定義の落とし穴と計算量

上で見たように関数が再帰的に定義できるというのはとても便利ですが，実はステップごとに計算量が増えていくような関数の定義には不向きです。以下でこのことを見てみましょう。ここではコンビネーション ${}_mC_n$ を定義します。これは m 個の異なるものの中から，n 個取り出す場合の数です。高校で習ったように

$$ {}_mC_n = \frac{m!}{(m-n)!n!} \tag{3} $$

です。一方で，これは次の漸化式によって表すこともできます：

$$ {}_mC_n = \begin{cases} 0 & \text{if } m < n \text{ or } n < 0 \\ 1 & \text{if } m = 0 \text{ or } m = n \\ {}_{m-1}C_n + {}_{m-1}C_{n-1} & \text{それ以外} \end{cases} \tag{4} $$

ここで $n < 0$ のときや $m < n$ のときは ${}_mC_n$ は意味がないので 0 としました。

(3), (4) に従って定義する関数 ${}_mC_n$ をそれぞれ C(m,n), D(m,n) としてプログラムを書いてみましょう：

- ファイル名：**combinat1.py**

```
1  def kaijou(n):    # kaijou(n)=n!を定義
2      if n == 0 or n==1:
3          return 1
4      else:
5          return n*kaijou(n-1)
6
7  def C(m,n):     # (1)で組み合わせ数を定義
8      return kaijou(m)/(kaijou(m-n)*kaijou(n))
9
10 def D(m,n):     # (2)で組み合わせ数を再帰的に定義
11     if n<0 or m<n:
12         return 0
13     elif m == 0 or m==n:
14         return 1
15     else:
```

```
        return D(m-1,n) + D(m-1,n-1)

for i in range(7):     #i=0,1,2,3,4,5,6に対して
    print C(6,i),       #C(6,i)を表示

print ''    # 改行する

for i in range(7):      #i=0,1,2,3,4,5,6に対して
    print D(6,i),      #C(6,i)を表示
```

実行結果の例

```
1 6 15 20 15 6 1
1 6 15 20 15 6 1
```

もちろん計算結果は同じになります。でも計算時間は大幅に異なります。次のプログラムで計算時間を見てみましょう：

- ファイル名：**combinat2.py**

```
# -*- coding:utf-8 -*-
from time import time    #timeというモジュールからtimeという関数をインポート

def kaijou(n):
   if n == 0 or n==1:
      return 1
   else:
      return n*kaijou(n-1)

def C(m,n):     # (1)で組み合わせ数を定義
    return kaijou(m)/(kaijou(m-n)*kaijou(n))

def D(m,n):     # (2)で組み合わせ数を再帰的に定義
    if n<0 or m<n:
       return 0
    elif m == 0 or m==n:
       return 1
    else:
       return D(m-1,n) + D(m-1,n-1)

time1 = time()  # この時間をtime1とする
print C(23,12)    # C(23,12)を計算
time2 = time()    # このときの時間をtime2とする
print '計算に要した時間は', time2-time1, '秒'
print D(23,12)    # D(23,12)を計算
time3 = time()    # このときの時間をtime3とする
print '計算に要した時間は', time3-time2, '秒'
```

実行結果の例

```
1352078
計算要した時間は 0.000201940536499 秒
1352078
計算要した時間は 1.78009104729 秒
```

(3) を使って計算した場合は計算時間は 0.0002 秒なのに対して，(4) を使った計算では 1.78 秒かかっています。これは漸化式 (4) を使った計算では非常に遠回りをして答えを出しているからです。例えば，漸化式 (4) から

C(5,2) を計算するのに，次のような膨大な計算を行っていることになります：

$$
\begin{aligned}
C(5,2) &= C(4,2) + C(4,1) \\
&= C(3,2) + C(3,1) + C(3,1) + C(3,0) \\
&= C(2,2) + C(2,1) + C(2,1) + C(2,0) + C(2,1) + C(2,0) + C(2,0) + C(2,-1) \\
&= 1 + C(1,1) + C(1,0) + C(1,1) + C(1,0) + C(1,0) + C(1,-1) + C(1,1) + C(1,0) \\
&\quad + C(1,0) + C(1,-1) + C(1,0) + C(1,-1) + 0 \\
&= 1 + 1 + C(0,0) + C(0,-1) + 1 + C(0,0) + C(0,-1) + C(0,0) + C(0,-1) + 0 + 1 \\
&\quad + C(0,0) + C(0,-1) + C(0,0) + C(0,-1) + 0 + C(0,0) + C(0,-1) + 0 \\
&= 1 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 0 + 1 + 1 + 0 + 1 + 0 + 0 + 1 + 0 + 0 \\
&= 10
\end{aligned}
$$

こんなやり方では，もっと大きな組み合わせの数 C(54,24)=1402659561581460 を計算するには約 100 年もかかってしまいます。ですから，コンビネーションを与える関数を定義するには公式 ${}_mC_n = m!/(m-n)!n!$ を使うべきです。このように簡単に再帰的に定義できる関数でも，事実上計算不可能な定義になってしまうことがあるので注意が必要です。

20.9 練習問題

与えられた自然数 n に対して

$$\sum_{j=1}^{n} \frac{1}{j^2} \tag{5}$$

を返す関数を再帰的に定義したい。上の和を返すような関数 Basel_sum(n) を定義しなさい。

- ファイル名：**Basel_sum.py**

```
def Basel_sum(n):
    if n<2:
        return 1
    else:
        ****** ************************

print Basel_sum(10)
print Basel_sum(100)
```

21 Set 型の操作と応用

21.1 基本的な集合の操作

以前少し紹介しましたが Python には集合 (set) というデータの型があります。これはデータの集まりであることはリストと同じですが，順番がない事と，重複が除かれる事がリストとは異なる所です。ですのでこれは数学の集合と類似の概念です。集合は set([1,3,4]) のように表します。集合同士の演算『union ∪, intersection ∩』をそれぞれ『|, &』で行うことができます。例えば，Python のインタラクティブシェルでの集合の操作は次のようにします：

```
>>> a = set([1,3,4])      # 集合aを定義
>>> b = set([3,7])        # 集合bを定義
>>> c = a | b             # cを集合aとbの和集合とする
>>> c
set([1, 3, 4, 7])
>>> d = a & b             # dを集合aとbの共通部分とする
```

```
>>> d
set([3])
```

集合の演算を行うには，union() や intersection() というメソッドを使うこともできます。また|=を使うことで，集合に他の集合の要素を加えることができます：

```
>>> a.union(b)              #a.union(b)は
set([1, 3, 4, 7])           #aとbの和集合を返す
>>> a.intersection(b)       #a.intersection(b)は
set([3])                    #aとbの共通部分を返す
>>> a
set([1, 3, 4])              # 上の操作でaの値が変わるわけではない
>>> b
set([3, 7])                 #bもかわらない
>>> a |= b                  #aにbの要素をすべて追加する
>>> a                       #aには
set([1, 3, 4, 7])           #bの要素が追加される
```

集合に要素を追加するには add，要素を削除するには remove を使います：

```
>>> a.add(8)          # 集合aに要素8を加える
>>> a                 # aの値を聞く
set([8, 1, 3, 4])     # aに要素8が追加された
>>> a.remove(1)       # aから要素1を除く
>>> a
set([8, 3, 4])        # aから1が削除された
```

21.2 Set 型の応用——四則演算で 10 を作る遊び

鉄道の切符に書かれている 4 桁の数字（または自動車のナンバーの 4 つの数字）から四則演算で 10 を作るゲームがあります。例えば，切符の数字が 2339 なら，

$$(2+9)-(3/3)=10$$

のようにして数字 10 を作ることができます。4 つの数字はどのような順番で計算してもかまいません。しかし 1111 のような数字からはどうやっても 10 を作ることは出来ません。どのような数字の組なら四則演算で 10 を作ることが出来るでしょうか？ 10 を作ることができるような数字の組を列挙するプログラムを Python で書いてみたいと思います。これを行うプログラムはいろいろな方法で書けると思いますが，ここでは Python らしく set を使ってみます。（ちなみに切符の端に書かれている 4 桁の数字は 0 をとらないらしいですが，ここでは 0 を含む場合も考えます。）

21.2.1 2 項演算

まずは 2 つの数字の組 a,b の四則演算で作られる数 $a+b, a-b, b-a, a*b, a/b, b/a$ を要素に持つ集合を返す関数を作ります。これを nikou(a,b) としましょう。この場合，得られる数の順番や重複は意味がないので，set を使うのが便利なのです。割り算のときに 0 で割る可能性を排除するために場合分けが必要です：

- ファイル名：**nikou.py**

```
def nikou(a,b):
    if a !=0 and b!=0:      # aもbも0でないときは
        return set([a+b,a-b,b-a,a*b,1.0*a/b, 1.0*b/a]) # 四則演算からできる集合
```

```
4     elif b == 0:            # b=0のときは
5         return set([a, -a, 0])
6     else:                   # a=0のときは
7         return set([b, -b, 0])
8
9 print nikou(3,4)     #3,4から四則演算で作ることができる数の集合を表示
```

実行結果の例

```
set([0.75, 1, 7, 12, 1.3333333333333333, -1])
```

上のプログラムで 1.0*あるのは演算『/』を浮動小数点数の割り算にしたいからです。

21.2.2　3 つの数の四則演算

つぎに，3 つの数 a,b,c の四則演算で作られる数の集合を返す関数 sankou(a,b,c) を作りたいと思います。まず 3 つの数はどの順番で計算するかによって 3 通りの順番があることに注意します。ある二項演算を $\circledast, \ominus$（つまり $\circledast, \ominus$ は＋－×÷のどれか）として

- $(a \circledast b) \ominus c$　：　a, b を先に計算してから次に c との演算を行う。
- $(a \circledast c) \ominus d$　：　a, c を先に計算してから次に d との演算を行う。
- $(b \circledast c) \ominus a$　：　b, c を先に計算してから次に a との演算を行う。

のように 3 つの計算の順番があります。

上のことに注意して 3 つの数 a,b,c に対する 2 項演算から作られる数の集合を返す関数 sankou(a,b,c) を定義してみましょう。

- ファイル名：**sankou1.py**

```
1  def nikou(a,b):
2      if a !=0 and b!=0:
3          return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
4      elif b == 0:
5          return set([a,-a,0])
6      else:
7          return set([b, -b, 0])
8
9  def sankou(a,b,c):          # a,b,cの四則演算で作られる集合を返す
10     ResultSet = set([])   # ResultSetを空の集合とする
11     for i in nikou(a,b):          # a,bから作られる数iに対して
12         ResultSet |= nikou(i,c) # i,cの二項演算から作られる集合をResultSetに追加
13     for i in nikou(b,c):
14         ResultSet |= nikou(i,a)
15     for i in nikou(a,c):
16         ResultSet |= nikou(i,b)
17     return ResultSet    # ResultSetを返す
18
19 print sankou(4,4,9)           # 4,4,9を1回ずつ使った四則演算で作られる集合
20 print ''                     # 改行
21 print 10 in sankou(4,4,9)    #sankou(4,4,9)には10は入っているか？
```

実行結果の例

```
set([-1.25000000000000, 3.25000000000000, 1.25000000000000, 1,
8.00000000000000, 9, 10.0000000000000, -3.55555555555556, 144, 17, 20,
25, -0.800000000000000, 0.562500000000000, 0, 32, 0.888888888888889, 40,
7, -1.75000000000000, 52, 1.75000000000000, 1.12500000000000,
4.44444444444444, 72, 0.307692307692308, -7, -32, 0.800000000000000,
3.55555555555556, -20, 0.111111111111111, -9, -8.00000000000000,
1.77777777777778, -1, 6.25000000000000])

True
```

さて，これで 3 つの数字の組から 10 を作れるかどうかを判定する事が可能になりました。そこで三つの数字 $0 \leq a \leq b \leq c \leq 9$ で四則演算によって 10 を作ることができるものをすべて列挙してみましょう：

- ファイル名：**sankou2.py**

```
# -*- coding:utf-8 -*-
def nikou(a,b):
    if a !=0 and b!=0:
        return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
    elif b == 0:
        return set([a,-a,0])
    else:
        return set([b, -b, 0])

def sankou(a,b,c):       # a,b,cの四則演算で作られる集合を返す
    ResultSet = set([])  # ResultSetを空の集合とする
    for i in nikou(a,b):         # a,bから作られる数iに対して
        ResultSet |= nikou(i,c) # i,cの二項演算でできる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet     # ResultSetを返す

counter = 0      # counterという変数を作り0にセットする

for i in range(10):
    for j in range(i,10):
        for k in range(j,10):
            if 10 in sankou(i,j,k):    # もし10がsankou(i,j,k)の要素なら
                print i,j,k            # i,j,kを表示して
                counter = counter + 1 # numberを1増やす

print '10を作れる数字の三つ組みの数は', counter, '個'
```

実行結果の例

```
0 1 9
0 2 5
...
...
9 9 9
10 を作れる数字の三つ組みの数は 75 個
```

21.2.3 4 つの数字の四則演算

つぎに 4 つの数 a,b,c,d から作ることのできる数の集合を返す関数を作りたい。どれか 2 つの数を先に計算することで，3 つ組の場合に帰着する。例えば，a,b を先に計算してその計算結果と c,d との四則演算で作ることのできる数の集合を作るには，a,b のすべての計算結果 i に対して sankou(i,c,d) を作ればよい。a,b,c,d のうちどの 2 つを先に計算するかについては，${}_4C_2 = 6$ 通りの場合がある。つまり，最初に計算す

る数は全部で (a,b),(a,c),(a,d),(b,c),(b,d),(c,d) の 6 通りになる。このような手順で作られる数字の集合を返す関数を yonkou(a,b,c,d) とすると，これは次のように定義すればよい

```
def yonkou(a,b,c,d):
    ResultSet = set([])
    for i in nikou(a,b):              # a,bの計算結果iに対して
        ResultSet |= sankou(i,c,d)   # i,b,cから作られる数をResultSetに追加
    for i in nikou(a,c):
        ResultSet |= sankou(i,b,d)
    for i in nikou(a,d):
        ResultSet |= sankou(i,b,c)
    for i in nikou(b,c):
        ResultSet |= sankou(i,a,d)
    for i in nikou(b,d):
        ResultSet |= sankou(i,a,c)
    for i in nikou(c,d):
        ResultSet |= sankou(i,a,b)
    return ResultSet
```

結局，0 から 9 までをとる 4 つの数字 a, b, c, d で $a \leq b \leq c \leq d$ かつこれらの四則演算で 10 を作ることができる組をすべて列挙するプログラムを作成するには次のようにすればよい：

- ファイル名：**make10.py**

```
# -*- coding:utf-8 -*-
def nikou(a,b):
    if a !=0 and b!=0:
        return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
    elif b == 0:
        return set([a,-a,0])
    else:
        return set([b, -b, 0])

def sankou(a,b,c):        # a,b,cの四則演算で作られる集合を返す
    ResultSet = set([])   # ResultSetを空のリストとする
    for i in nikou(a,b):          # a,bから作られる数iに対して
        ResultSet |= nikou(i,c) # i,cの演算から作られる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet    # ResultSetを返す

def yonkou(a,b,c,d):
    ResultSet = set([])
    for i in nikou(a,b):              # a,bの計算結果iに対して
        ResultSet |= sankou(i,c,d)   # i,b,cから作られる数をResultSetに追加
    for i in nikou(a,c):
        ResultSet |= sankou(i,b,d)
    for i in nikou(a,d):
        ResultSet |= sankou(i,b,c)
    for i in nikou(b,c):
        ResultSet |= sankou(i,a,d)
```

```
30      for i in nikou(b,d):
31          ResultSet |= sankou(i,a,c)
32      for i in nikou(c,d):
33          ResultSet |= sankou(i,a,b)
34      return ResultSet
35
36  counter = 0
37
38  for i in range(10):
39      for j in range(i,10):
40          for k in range(j,10):
41              for l in range(k,10):
42                  if 10 in yonkou(i,j,k,l):
43                      print i,j,k,l
44                      counter = counter+1
45
46  print '10を作れる数字の4つ組みの数は', counter, '個'
```

実行結果の例

```
.......
.......
.......
8 8 9 9
9 9 9 9
10を作ることができる4つの数字の組の数は 552 個
```

上のプログラムは結果的にはうまく動いていますが，本来なら桁落ちに注意が必要です。それは，二項演算を浮動小数点で行っているために，演算の結果が正確には 10 のはずなのに，誤差により 10 ではなくなっている可能性があるためです。ただ，10 に非常に近い数は 10 であると認識されます：

```
1  >>> 0.99999999999999999 == 1
2  True
3  >>> 0.9999999999999999 == 1
4  False
```

21.3 練習問題

5 つの数字 $a, b, c, d, e (0 \leq a \leq b \leq c \leq d \leq e \leq 9)$ を 1 回ずつ使い四則演算（加算，減算，乗算，除算）で 100 を作ることを考える。100 を作れるような全ての数の組及びその個数を表示するプログラムを作成せよ。ファイル名は `make100.py` とすること。

22 おまけ

22.1 文字列から式や文を作り出す

Python のプログラムは決められた文法によって記述しれなければなりませんが，それらは式 (expression)，文 (statement) によって作られます。例えば『`a=3`』は文ですが，『`1+3`』は式です。また，コーテーションマークで囲ってしまえば`'1+3'` も`'a=3'` も文字列です。

ここで一つ疑問が生じます，文字列を式や文に変換することはできるでしょうか？ Python では `eval` 関数と `exec` 関数という関数用意されており，これを行うことが出来ます。

eval 関数と exec 関数

- eval 関数は文字列で表されている式 (expression) を評価してその値を返す。
- exec 関数は文字列で表されている文 (statement) を実行する。

例：

```
>>> eval('1+3')          #1+3が評価されてその値が返される
4
>>> exec('a=3')           #a=3が実行される
>>> print a
3
```

ここで 1+3 は式，a=3 は文であり，'1+3' と'a=3' は文字列であることに注意。

eval を使うと文字列を変数名に変えることができます。例えば次のようなプログラムを作ったとしましょう：

```
def func1():
    return 2**2**2

def func2():
    return 'hogehoge'

def func3():
    return type(1.4)

def func4():
    return [1,2,3,4]

print func1()
print func2()
print func3()
print func4()
```

実行結果の例

```
16
hogehoge
<type 'float'>
[1,2,3,4]
```

func1, func2, func3, func4 と 4 つの関数を作ったのでこれをプリントする為に 4 回関数を呼び出す必要がありました。これを自動化するために，関数名からなる文字列を生成して eval で評価します：

```
def func1():
    return 2**2**2

def func2():
    return 'hogehoge'

def func3():
    return type(1.4)

def func4():
    return [1,2,3,4]
```

```
12
13 for i in range(1,5):
14     print eval('func'+str(i)+'()')
```

実行結果は先ほどと同じになります。ここで str 関数は中身を文字列に変換する関数だったことを思い出しましょう。

現実的には，ほとんどの場合で eval や exec を使わずにプログラムを書くことが出来ます。これらの関数を使うと，実行するまで何が起きるかわからないようなプログラムが書けてしまい，セキュリティー上問題が生じます。ですので，個人的な用途以外にはこれらの関数は使わないほうがよいでしょう。