あなたの友だちがしようとしていることを説明して下さい。また、できる限り穏やかな表現で、彼のアプローチの問題点を２つ指摘して下さい。

5章
リスト

今まで私たちが作ってきた変数は、１つの数値または文字列として参照されていました。この章では、Python の list という型を使って、データのコレクションを操作します。リストは 0 個以上のオブジェクトを格納するもので、90 件の実験結果や 10,000 人の学生の ID などのデータを管理できます。この章では、ファイルアクセスの方法や、ファイルの内容をリストとして表現する方法も学びます。

5.1 リストと添字

表 5.1 は、http://www.acschannelislands.org/2008CountDaily.pdf から引用したもので、2008 年春の 2 週間にコールオイルポイント自然保護区（カリフォルニア州サンタバーバラ）の近くで観測されたコククジラの頭数を示しています。

表5.1 コククジラの頭数調査

日	頭数
1	5
2	4
3	7
4	3
5	2
6	3
7	2
8	6
9	4
10	2
11	1
12	7
13	1
14	3

私たちが今までに学んだものを使うなら、これらの数値を管理するために 14 個の変数を作らなければならないところです（図 5.1 参照）。1 年分の観察記録を追跡したければ、366 個もの変数が

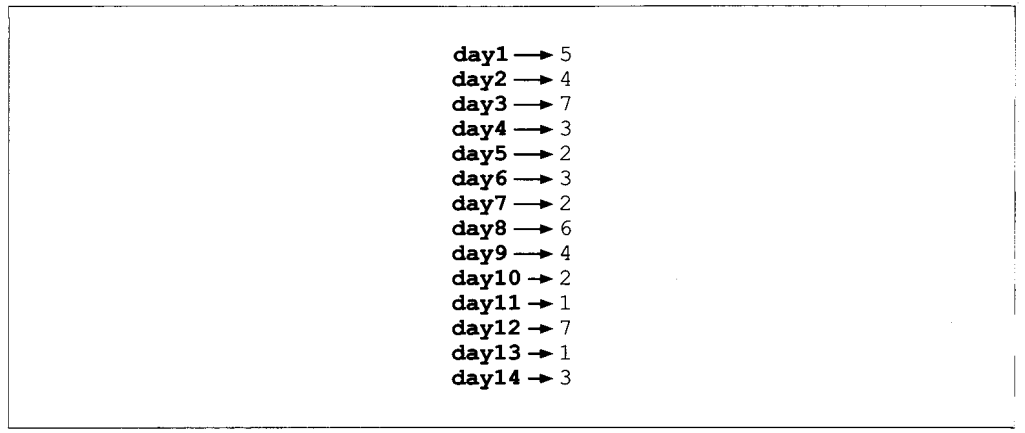

図5.1 リストがなければどうするか

必要です（閏年に合わせると）。さらに、クジラの観測をどれだけの間続けるかがあらかじめわかっていなければ、いくつの変数を作ったらよいかもわかりません。

この問題は、すべての値をまとめて**リスト**（list）に格納すれば解決できます。リストは、実生活の中のあちこちで見かけます。クラスの学生たち、ニューギニア産の鳥の種類などです。Pythonでは、角かっこの中にカンマ区切りで値を入れるだけでリストを作れます。

lists/whalelist.py

```
# 1日に観測されたコククジラの頭数
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```

リストはオブジェクトです。他のオブジェクトと同様に、リストは変数に代入できます。

lists/whales1.cmd

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales
[5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
```

図5.2は、代入後のwhalesのメモリモデルを示しています。覚えておきたい大切なことは、リスト自体は1個のオブジェクトですが、他のオブジェクトに対する参照（図では矢印で表現しています）を格納できるということです。

では、リストに含まれるオブジェクトにアクセスするにはどうすればよいのでしょうか。アクセスしたいものを指定する**添字**（index）を使うのです。リストの先頭要素は添字0の位置、2番目の要素は添字1の位置にあります†。リストの特定の要素を参照するには、リストの参照（たとえば変数名）の後ろに角かっこで添字を囲んだものを使います。

† 自然言語と同じように、添字の先頭を1にした方が、おそらく自然でしょう。しかし、Pythonは、CやJavaなどの言語と同じ習慣に従い、添字を0から数えることにしています。

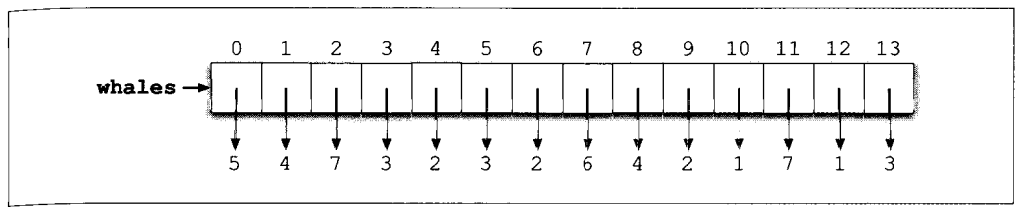

図5.2 リストの例

lists/whales2.cmd

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[0]
5
>>> whales[1]
4
>>> whales[12]
1
>>> whales[13]
3
```

使える添字は、0からリストの長さマイナス1までの範囲だけです。14個の要素を持つリストの場合、有効な添字は0、1、2、...、13までです。範囲外の添字を使おうとすると、0で除算しようとしたときと同じようにエラーになります。

lists/whales3.cmd

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[1001]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

ほとんどのプログラミング言語とは異なり、Pythonではリストの末尾から数える添字も使えます。この場合、最後の要素の添字が-1、その1つ前が添字-2のようになります。

lists/whales4.cmd

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> whales[-1]
3
>>> whales[-2]
1
>>> whales[-14]
5
```

リスト内の値は、他の変数に代入できます。

lists/whales5.cmd

```
>>> whales = [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
>>> third = whales[2]
>>> print 'Third day:', third
Third day: 7
```

5.1.1 空リスト

ゼロは便利な数です。そして、「3.1 文字列」で説明したように、空文字列も同じように便利です。リストにも**空リスト**(empty list)、すなわち要素を1つも持たないリストがあります。空リストは、[]と書きます。空リストに添字でアクセスしようとすると、かならずエラーになります。

lists/whales6.cmd

```
>>> whales = []
>>> whales[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> whales[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

これは、有効な添字の定義によるものです。

- N個の要素を持つリストの有効な添字は、集合$\{i\colon 0 \le i < N\}$に含まれる整数です。
- 空文字列の長さは0です。
- 空文字列の有効な文字列は、そのため集合$\{i\colon 0 \le i < 0\}$の要素です。
- この集合は空集合なので、空リストには有効な添字はありません。

5.1.2 異種要素を格納できるリスト

リストは、整数、文字列、さらには他のリストを含め、あらゆる型のデータを格納できます。次のリストは、クリプトンの名前、元素記号、融点(単位は摂氏度)、沸点(単位は摂氏度)の4つの情報から作られたリストです。このようにリストを使って関連情報を管理するのは、エラーを起こしやすい方法です。これよりも優れているもののより上級者向けの方法については、「13章 オブジェクト指向プログラミング」で説明します。

lists/krypton1.cmd

```
>>> krypton = ['Krypton', 'Kr', -157.2, -153.4]
>>> krypton[1]
'Kr'
>>> krypton[2]
-157.19999999999999
```


5.2 リストの書き換え

希ガス†のリストを入力していて指が滑ってしまいました。

lists/nobles1.cmd

```
>>> nobles = ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
```

'neon'ではなく、'none'と入力してしまったのです。このようなとき、リスト全体を入力し直すのではなく、リストの特定の要素に新しい値を代入することができます。

lists/nobles2.cmd

```
>>> nobles = ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
>>> nobles[1] = 'neon'
>>> nobles ['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

図5.3は、nobles[1]に対する代入によって何が起きたかを示しています。また、この図はリストが**ミュータブル**(mutable)であること、つまりその内容が作成後も書き換えられることを示しています。それに対し、数値や文字列は**イミュータブル**(immutable)で、たとえば作成後の文字列の文字を変更することはできません。変更をしているように見えるupperなどのメソッドは、実際には新しい文字列を作っています。

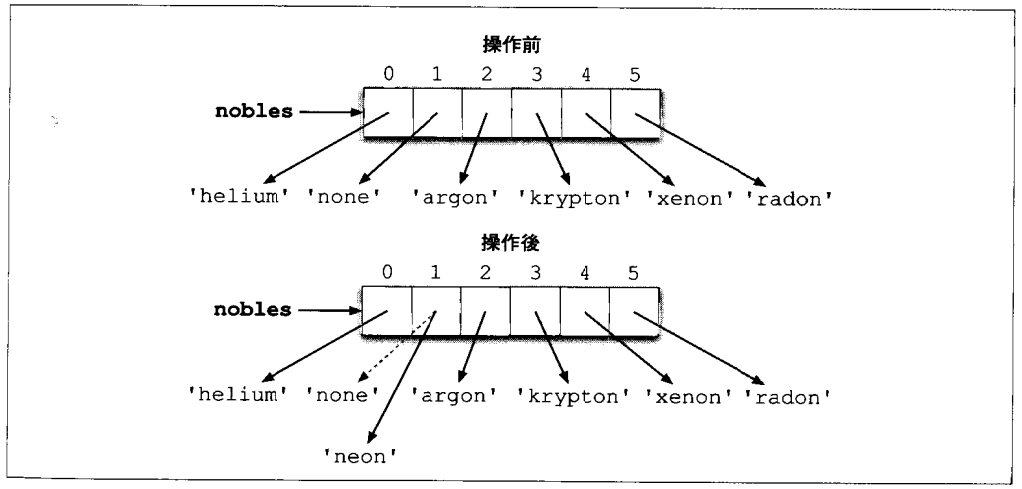

図5.3 リストの書き換え

† **希ガス**(noble gas)は、最外殻電子が閉殻となっているために、化学的に不活性なガスです。

lists/strings_immutable.cmd

```
>>> name = 'Darwin'
>>> capitalized = name.upper()
>>> print capitalized
'DARWIN'
>>> print name
'Darwin'
```

L[i] という式は、単純な変数と同じようにふるまいます（「2.4 変数と代入文」参照）。右辺にある場合は、「リストLの添字iの位置にある値を取り出せ」という意味になります。左辺にある場合は、「上書きするために、リストLの添字iがどこにあるかを調べよ」という意味になります。

5.3 リストの組み込み関数

「2.6 関数の基礎」では、Python の組み込み関数の一部を紹介しました。その中でも len のようなものはリストにも適用できますし、まだ紹介していない関数の中でリスト操作に使えるものもあります（表 5.2 参照）。

表5.2 リスト関数

関数	説明
len(L)	リストLに含まれている要素の数を返します。
max(L)	リストLの最大値を返します。
min(L)	リストLの最小値を返します。
sum(L)	リストLの値の合計を返します。

次のコードでは、プルトニウム同位体の半減期†をまとめたリストにそれらの関数を適用しています。

lists/plu4.cmd

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> len(half_lives)
5
>>> max(half_lives)
376000.0
>>> min(half_lives)
14.4
>>> sum(half_lives)
406749.14000000001
```

† 放射性同位体の半減期とは、その物質が半分に減るまでの時間のことです。この時間の倍の時間が経過すると、物質の $\frac{3}{4}$ がなくなっています。3 倍の時間が経過すると $\frac{7}{8}$ がなくなっています。

式の中では、組み込み関数の結果を使えます。たとえば、次のコードは、添字が範囲内に収まっているかどうかをチェックしています†。

lists/plu5.cmd

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> i = 2
>>> 0 <= i < len(half_lives)
True
>>> half_lives[i]
6537.0
>>> i = 5
>>> 0 <= i < len(half_lives)
False
>>> half_lives[i]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

他のオブジェクトと同様に、リストは型を持っていますので、異なる型のものを不適切な方法で結合しようとすると、Python はエラーを起こします。たとえば、リストと文字列を「加算」しようとすると、次のようになります。

lists/add_list_str.cmd

```
>>> ['H', 'He', 'Li'] + 'Be'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

このエラーメッセージには面白いことが書かれています。文字列と文字列を連結して新しい文字列を作るのと同じように、リストとリストを連結して新しいリストが作れるようなことが書かれています。実際に試してみると、その通りになることがわかります。

lists/concat_lists.cmd

```
>>> original = ['H', 'He', 'Li']
>>> final = original + ['Be']
>>> final
['H', 'He', 'Li', 'Be']
```

図 5.4 に示すように、連結をしても、元の 2 つのリストは変更されず、個々の要素が元のリスト

† 訳注：ここで使われている <= や < といった演算子、True や False といった値については、6 章で詳しく説明します。たとえば、0 <= i < len(half_lives) は、「i は 0 以上で len(half_lives) 未満か？」という意味で、その通りなら True、そうでなければ False が返されます。

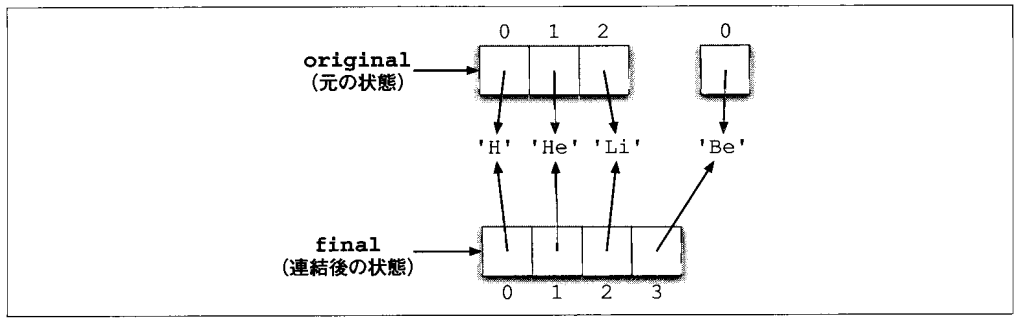

図5.4 リストの連結

の要素と同じものを指している新しいリストが作られます。

リストに対して + を実行できるのなら、文字列のリストに対して sum を実行することはできるのでしょうか。sum([1, 2, 3]) が 1 + 2 + 3 と同じなら、sum('a', 'b', 'c') は 'a' + 'b' + 'c'、そして 'abc' と同じになるべきではないでしょうか。しかし、次のコードを見ると、そこまで類推を利かせるわけにはいかないことがわかります。

lists/sum_of_str.cmd

```
>>> sum(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

それに対し、リストに整数を掛けると、元のリストをその回数だけ繰り返して作った新しいリストが得られます。

lists/mult_lists.cmd

```
>>> metals = 'Fe Ni'.split()
>>> metals * 3
['Fe', 'Ni', 'Fe', 'Ni', 'Fe', 'Ni']
```

連結のときと同様に、元のリストは書き換えられず、新しいリストが作成されます。なお、string.split を使って 'Fe Ni' という文字列を ['Fe', 'Ni'] という 2 要素のリストに変換していることに注目して下さい。これは、Python プログラムでよく見られるトリックです。

5.4 リストの要素の操作

リストは、1,000 個の値を格納するために 1,000 個の変数を作らなくても済むようにするために作られました。同じ理由から、Python には **for ループ** (for loop) があります。for ループを使えば、個々の要素のために 1 つずつ文を書かなくても、リスト内の各要素を順に処理することができます。for ループの一般形式は、次の通りです。


```
for variable in list:
    block
```

「2.6 関数の基礎」で説明したように、*block* (ブロック) は 1 個以上の文を並べたものにすぎません。*variable* と *list* は、単純に変数とリストです。

Python は、ループを検出すると、リスト内の各要素のために 1 度ずつループのブロックを実行します。ブロックの毎回の実行を**イテレーション** (iteration) と呼び、イテレーションを開始する直前に Python はリストの次の値を指定された変数に代入します。こうすることにより、プログラムは個々の値に順に何らかの操作を加えることができるわけです。

たとえば、次のコードは、落ちるものの速度をメートル法とヤードポンド法の両方で表示します。

lists/velocity_loop.cmd

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for v in velocities:
...     print "メートル法:", v, "m/秒、",
...     print "ヤードポンド法:", v * 3.28, "ft/秒"
...
メートル法: 0.0 m/秒; ヤードポンド法: 0.0 ft/秒
メートル法: 9.81 m/秒; ヤードポンド法: 32.1768 ft/秒
メートル法: 19.62 m/秒; ヤードポンド法: 64.3536 ft/秒
メートル法: 29.43 m/秒; ヤードポンド法: 96.5304 ft/秒
```

このループに関しては、他に次の 2 点に注目して下さい。

- 英語では、「for each velocity in the list, print the metric value, and then print the imperial value」(リストに含まれる個々の速度データについて、そのメートル法による値を表示し、さらにヤードポンド法の値を表示する) と表現しますが、Python でもほぼ同じ表現を使います。
- 関数定義と同様に、ループブロックの文はインデントされます (本書では 4 個のスペースを使っていますが、他の方法を使うかどうかは、あなたの講座の先生に確かめて下さい)。

この場合、ループ内でリストから取り出した現在の値を格納するために、v という新しい変数を作っていますが、既存の変数を使うこともできます。その場合でも、ループはリストの先頭要素からスタートします。ループを実行する前に変数に格納されていた値は失われます。

lists/velocity_recycle.cmd

```
>>> speed = 2
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for speed in velocities:
...     print "メートル法:", speed, "m/秒"
...
メートル法: 0.0 m/秒
メートル法: 9.81 m/秒
メートル法: 19.62 m/秒
メートル法: 29.43 m/秒
```

```
>>> print "最終:", speed
最終: 29.43
```

いずれにしても、ループ終了時には、最後の値を保持した状態で変数は残ります。このプログラムの最後のprint文がインデントされていないことに注意して下さい。この文はループの一部ではありません。ループ終了後に1度だけ実行される文です。

5.4.1 ループのネスト

先ほど、ループ内のブロックには任意の文を入れられると言いましたが、それは他のループを入れることもできるという意味です。

たとえば、次のプログラムは、outerリストの各要素について、innerリスト全体をループで処理します。

lists/nested_loops.cmd

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
...     for halogen in inner:
...         print metal + halogen
...
...
LiF
LiCl
LiBr
NaF
NaCl
NaBr
KF
KCl
KBr
```

外側のループがN_o回繰り返され、内側のループが外側のループの各イテレーションごとにN_i回ずつ繰り返される場合、内側のループは、合計でN_oN_i回繰り返されることになります。外側のループと内側のループが長さNの同じリストを反復処理するのは、これの特殊条件になり、内側のループはN^2回実行されます。これを使うと、かけ算の早見表が作れます。ヘッダ行を出力した後、ループのネストを使って表の各行を出力していきます。列の位置揃えにはタブを使います。

lists/multiplication_table.py

```
def print_table():
    '''1から5までのかけ算九九の表を出力します。'''

    numbers = [1, 2, 3, 4, 5]

    # ヘッダ行を出力
    for i in numbers:
        print '\t' + str(i),

    print # ヘッダ行を終了(改行)

    # 行番号と表の内容の出力
    for i in numbers:
        print i,
        for j in numbers:
            print '\t' + str(i * j),
        print # 現在行を終了(改行)
```

print_tableの出力は、次のようになります。

lists/multiplication_out.txt

```
>>> from multiplication_table import *
>>> print_table()
        1       2       3       4       5
1       1       2       3       4       5
2       2       4       6       8       10
3       3       6       9       12      15
4       4       8       12      16      20
5       5       10      15      20      25
```

2種類の異なる整形処理が行われていることに注意して下さい。プログラムの末尾のprint文は外側のループを次に進めるときの改行を出力するのに対し、内側のループのprint文は各要素の前にタブを挿入しています。

5.5 スライシング

遺伝学者たちは、3文字の略記号を使ってシー・エレガンス(線虫、ネマトーダ)のマーカー遺伝子を表します。たとえば、Emb(embryonic lethality：幼生のうちに死亡)、Him(high incidence of males：高い確率で雄)、Unc(uncoordinated：非協調的)、Dpy(dumpy：太く短い)、Sma(small：小さい)、Lon(long：長い)などです。

lists/celegans.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

しかし、DpyとSmaは互いに見分けにくく、複雑な遺伝系統のマーカー遺伝子としてはあまり役に立ちません。そこで、celegans_markersリストをスライスし、DpyとSmaのない新しいリストを作ることにします。

lists/celegans1.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> useful_markers = celegans_markers[0:4]
```

こうすると、識別できる4種類のマーカーだけから構成される新しいリストが作られます（図5.5参照）。

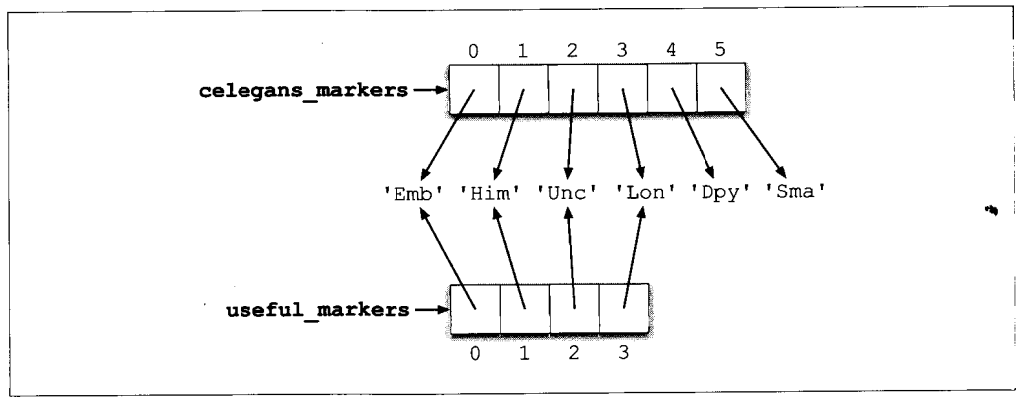

図5.5 スライシングはリストに変更を加えない

スライスの第1の添字は、先頭を表します。第2の添字は、取り込みたい最後の要素の添字に1を加えた値です。より厳密に言うと、list[i:j]は、元のリストの添字i以上j未満の要素によるスライスです†。

先頭要素からスライスするなら第1の添字を省略でき、末尾の要素までスライスするなら第2の添字を省略できます。

lists/celegans2.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_markers[:4]
['Emb', 'Him', 'Unc', 'Lon']
>>> celegans_markers[4:]
['Dpy', 'Sma']
```

リスト全体のコピーを作るなら、両方の添字を省略して、先頭要素から末尾の要素までを「スライス」したリストを作ります。

† Pythonがこのような習慣を使っているのは、0からリストの長さマイナス1までという有効な添字の規則との一貫性を保つためです。

lists/celegans3.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_copy = celegans_markers[:]
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

5.6 エイリアシング

エイリアス（alias）とは、同じものの別名のことです。Pythonでは、同じ値を参照する2つの変数はエイリアスになっています。たとえば、次のコードは2個の変数を作りますが、どちらの変数も同じリストを参照しています（図5.6参照）。どちらか片方の変数を使ってリストに変更を加えると、もう片方の変数から見える値も同じように変わります。

lists/celegans4.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_copy = celegans_markers
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
```

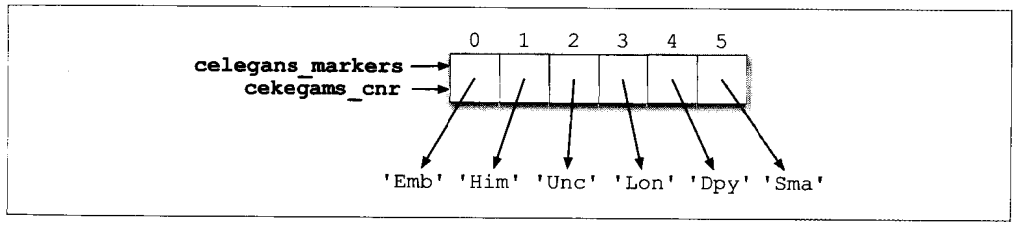

図5.6 リストのエイリアシング

ミュータブルという概念は重要ですが、その理由の1つがエイリアシングです。たとえば、xとyが同じリストを参照しているとき、xからリストに加えた変更はyからも見えますし、逆も同様です。自分のプログラムは何も代入していないはずなのに、何かの魔法でリストの値が変わってしまうという見つけにくいエラーがありますが、エイリアシングはそういうエラーの原因になり得ます。文字列のようなイミュータブルな値では、このようなエラーは起きません。文字列は作成後に変更することはできませんので、エイリアスがあっても危険なことは起きないのです。

5.6.1 関数呼び出しにおけるエイリアシング

関数の仮引数は変数ですから、エイリアシングはリスト引数を使うときにも発生します。

次に示すのは、引数としてリストを取り、ソートしてから順序を反転する関数です。

lists/alias_parameters.cmd

```
>>> def sort_and_reverse(L):
...     '''ソートして逆順にしたリストLを返します'''
...     L.sort()
...     L.reverse()
...     return L
...
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> sort_and_reverse(celegans_markers)
['Unc', 'Lvl', 'Lon', 'Him', 'Emb', 'Dpy']
>>> celegans_markers
['Unc', 'Lvl', 'Lon', 'Him', 'Emb', 'Dpy']
```

この関数はリストLを変更し、Lはcelegans_markersのエイリアスなので、celegans_markersも変更されてしまいます。

5.7 リストメソッド

リストはオブジェクトなので、メソッドを持っています。表5.3に、もっともよく使われるメソッドをまとめてあります。

表5.3 リストメソッド

メソッド	説明
L.append(v)	リストLに値vを追加します。
L.insert(i, v)	リストLの添字iに値vを挿入します。スペースを作るために要素を後ろにずらします。
L.remove(v)	リストLに含まれる最初の値vを削除します。
L.reverse()	リストLの値の順序を逆にします。
L.sort()	リストLの要素を昇順(文字列の場合はアルファベット順)にソートします。
L.pop()	リストL(空リストではいけません)の最後の要素を取り除いて返します。

次のコードは、これらのメソッドを使って虹に含まれるすべての色から構成されるリストを構築する方法を示しています†。

† 訳注：英語でも虹の色は7色とされていますが、ここではなぜか6色で話が終わってしまっています。http://www1.umn.edu/ships/updates/newton1.htmによれば、ニュートンは最初虹の色を5色としていましたが、後から橙と藍を入れて7色としたそうです。7という数字は、楽音の数に合わせたもので、調和を表現しているということです。英語版Wikipediaは、7色としてblueの後にindigoを入れ、purpleの代わりにvioletを入れて7色にしています。日本語では、赤、橙、黄、緑、青、藍、紫（菫）の7色ですが、このコードは英語のままとしました。

lists/colors.cmd

```
>>> colors = 'red orange green black blue'.split()
>>> colors.append('purple')
>>> colors
['red', 'orange', 'green', 'black', 'blue', 'purple']
>>> colors.insert(2, 'yellow')
>>> colors
['red', 'orange', 'yellow', 'green', 'black', 'blue', 'black', 'purple']
>>> colors.remove('black')
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
```

これらすべてのメソッドが新しいリストを作るのではなく、リストに変更を加えるということに注意して下さい。これらのメソッドが新しいリストを作らないのは、リストは非常に大きなものになることがあるからです。たとえば、100万の患者レコードのリストは、10億個分の磁気フィールドを使います。そのようなリストに変更を加えたくなったときにいちいち新しいリストを作っていると、Pythonの動作は極度に遅くなり、使い物にならなくなってしまいます。いつコピーを作り、いつリストを直接操作すべきかをPythonに判断させようとしても、Pythonは判断に困ることになるでしょう。

もう1つ覚えておかなければならないのは、popを除くすべてのメソッドが「役に立つ情報はない」、「ここには何もない」という意味のNoneという特殊な値を返すことです。Pythonは、値がNoneになっているものを表示せよと求められても何も表示しません。しかし、printを使えば、値がNoneだということがわかります。

lists/none.cmd

```
>>> x = None
>>> x
>>> print x
None
```

なお、append呼び出しは+とは異なります。まず、appendは1個の値を追加しますが、+は被演算子として2個のリストを要求します。第2に、appendは新しいリストを作らず、リストを書き換えます。

5.8 リストのネスト

「5.1.2 異種要素を格納できるリスト」で、リストは任意の型のデータを格納できると説明しました。これは、ループ本体に他のループを入れられるのと同じように、リストに他のリストを入れることができるということです。たとえば、次のネストされたリストは、さまざまな国の平均寿命を表しています。

リストはどこに消えたか

リストメソッドの多くは、新しいリストを作って返すのではなく、Noneを返してきますが、初心者プログラマはそのことを忘れがちです（ベテランでも、忘れることがあります）。そのため、彼らのリストはときどき消えたようになってしまいます。

lists/colors2.cmd

```
>>> colors = 'red orange yellow green blue purple'.split()
>>> colors
['blue', 'green', 'orange', 'purple', 'red', 'yellow']
>>> sorted_colors = colors.sort()
>>> print sorted_colors
None
```

「4.5 テスト」で説明したように、テストを書いて実行すれば、この種のエラーはすぐにキャッチできます。

lists/lifelist.py

```
[['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
```

図5.7に示すように、外側のリストの各要素は、それ自体2個の要素を持つリストになっています。次のように、外側のリストの要素には標準の記法でアクセスできます。

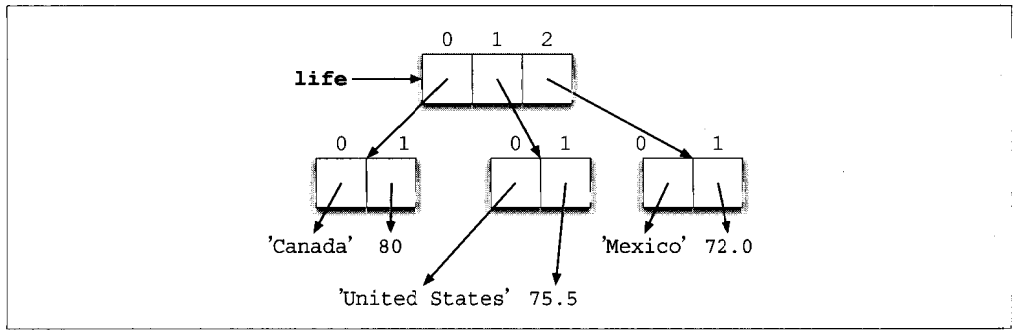

図5.7 リストのネスト

lists/life0.cmd

```
>>> life = [['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
>>> life[0]
['Canada', 76.5]
>>> life[1]
['United States', 75.5]
```



```
>>> life[2]
['Mexico', 72.0]
```

外側のリストの要素はそれ自身リストですから、メソッド呼び出しを連鎖させたり、ある関数呼び出しの結果を別の関数呼び出しの引数として渡したりするのとまったく同じように、再び添字を付けるだけで、下位リスト（内側のリスト）の要素にアクセスできます。

lists/life1.cmd

```
>>> life = [['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
>>> life[1]
['United States', 75.5]
>>> life[1][0]
'United States'
>>> life[1][1]
75.5
```

変数に下位リストを代入することもできます。

lists/life2.cmd

```
>>> life = [['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
>>> canada = life[0]
>>> canada
['Canada', 76.5]
>>> canada[0]
'Canada'
>>> canada[1]
76.5
```

変数に下位リストを代入すると、その下位リストのエイリアスが作られます（図5.8参照）。先ほどと同様に、下位リスト参照を通じて変更を加えると、外側のリストを介して下位リストにアクセスしたときにも同じように変更された形で見えます。

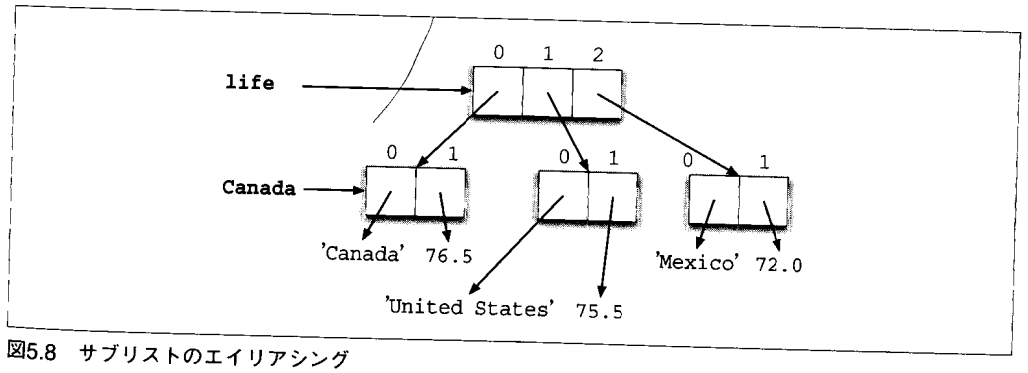

図5.8 サブリストのエイリアシング

lists/life3.cmd

```
>>> life = [['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0]]
>>> canada = life[0]
>>> canada[1] = 80.0
>>> canada
['Canada', 80.0]
>>> life
[['Canada', 80.0], ['United States', 75.5], ['Mexico', 72.0]]
```

5.9 その他のシーケンス

リストは、Python の唯一のシーケンスというわけではありません。すでに、他のシーケンスのうちの 1 つは見てきました。すなわち文字列です。正式な形で言えば、文字列は文字のイミュータブルなシーケンスです。この定義の「シーケンス」の部分は、リストと同じように添字でアクセスでき、スライスすると新しい文字列が作られることを意味しています。

lists/string_seq.cmd

```
>>> rock = 'anthracite'
>>> rock[9]
'e'
>>> rock[0:3]
'ant'
>>> rock[-5:]
'acite'
>>> for character in rock[:5]:
...     print character
...
a
n
t
h
r
```

Python はイミュータブルなシーケンス型として**タプル**（tuple）と呼ばれるものもサポートしています。タプルは、角かっこではなく普通のかっこを使って書きます。タプルは、文字列やリストと同様に、添字でアクセスしたり、スライスしたり、ループ処理したりすることができます。

lists/tuples1.cmd

```
>>> bases = ('A', 'C', 'G', 'T')
... for b in bases:
...     print b
A
C
G
T
```


タプルには、注意しておかなければならないことが 1 つあります。() は空タプルを表しますが、1 個の要素を持つタプルは (x) ではなく、(x,) と書かなければなりません（要素の後にカンマを付けます）。構文があいまいになるのを避けるために、こうすることが必要なのです。末尾にカンマを付けることが必須になっていなければ、(5 + 3) は、8（普通の計算のルールにより）という意味にもなれば、値として 8 だけを含むタプルという意味にもなってしまいます。これは、Python の構文規則にちょっと物足りなさが残る数少ない場面の 1 つです。

タプルは、作成後に書き換えることはできません。

lists/life4.cmd

```
>>> life = (['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0])
>>> life[0] = life[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

しかし、タプルに含まれているオブジェクト自体は書き換えることができます。

lists/life5.cmd

```
>>> life = (['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0])
>>> life[0][1] = 80.0
>>> life
(['Canada', 80.0], ['United States', 75.5], ['Mexico', 72.0])
```

ですから、何かがタプルの「中に」あるという言い方は、表現として完全ではありません。正確に言おうと思うなら、「タプルに格納されている参照は、タプル作成後には変更できませんが、参照されているオブジェクト自体は書き換えることができます」と言わなければなりません。

Python 初心者は、タプルの存在理由を疑問に思うことが多いようです。答えは、タプルがあれば効率がよくなる処理や安全になる処理があるからです。本書では前者についてはあまり深く追究しませんが、後者については「9 章　集合と辞書」で取り上げます。

5.10 リストとしてのファイル

ほとんどのデータは、バイトを順に並べたシーケンスであるファイルに格納されています。バイトは、文字、ピクセル、郵便番号などを表現します。ここで重要なのは、バイトシーケンスに順序があるということです。そのため、リストは一般にファイルを操作するための自然な方法として使えます。

ファイルからデータを読み出すためには、まず、Python の組み込み関数の open を使って、ファ

イルをオープンしなければなりません。

lists/open_basic.cmd

```
>>> file = open("data.txt", "r")
```

open の第1引数はファイル名を表す文字列で、第2引数はオープンのモードです。読み出し用の "r"、書き込み用の "w"、追加用の "a" の3種類があります（書き込みはファイルの既存の内容を消去しますが、追加は既存の内容の末尾に新しいデータを追加する、という違いがあります）。

open の結果は、ファイルの内容ではありません。open が返すのは**ファイルオブジェクト**で、ファイルオブジェクトのメソッドを使えばファイルの内容にアクセスできます。

これらのメソッドの中でも基礎中の基礎と言えるのが、read です。引数なしで read を呼び出すと、ファイルに含まれるすべてのデータが読み込まれ、文字列として返されます。read に正の整数の引数を渡すと、その文字数だけしか読み出しをしません。これは、非常に大きなファイルを扱っているときに役立ちます。いずれにしても、ファイルに読み出すべきデータが残っていなければ、read は空文字列を返します。

read を使えばファイル内のバイト情報にアクセスできますが、普通はもっと高い水準のメソッドを使います。たとえば、ファイルがテキストを格納している場合には、1度に1行ずつ処理できればと思うことでしょう。そのようなときには、ファイルオブジェクトの readline メソッドを使います。このメソッドは、ファイルから次の行を読み出します。1行とは、次の EOL マーカー（「3.3 マルチライン文字列」参照）までのすべての文字（マーカーも含む）と定義されています。ファイルに残されたデータがない場合は、readline も read と同様に空文字列を返します。

readline のもっともよいところは、for ループ内でファイルオブジェクトを使ったときに Python が自動的に readline を呼び出してくれることです。次のデータが data.txt というファイルに格納されているとします。

lists/data.txt

```
Mercury
Venus
Earth
Mars
```

次のプログラムは、ファイルをオープンし、各行の長さを表示します。

lists/fileinputloop.cmd

```
>>> data = open('data.txt', 'r')
>>> for line in data:
...     print len(line)
...
8
6
```



```
6
5
```

出力の最後の行をよく見て下さい。Mars という単語は4文字しかないのに、プログラムはその行が5文字だと言っています。こうなるのは、ファイルから読み出している各行には改行文字が付いているからです。先頭と末尾の空白文字（スペース、タブ、改行）を取り除いた文字列のコピーを返す string.strip を使えば、改行を取り除くことができます。

lists/fileinputloop2.cmd

```
>>> data = open('data.txt', 'r')
>>> for line in data:
...     print len(line.strip())
...
7
5
5
4
```

次のコードは、先頭と末尾に空白が含まれている文字列に strip を適用した結果を示しています。

lists/strip_basic.cmd

```
>>> compound = "     \n  Methyl butanol   \n"
>>> print compound

 Methyl butanol



>>> print compound.strip()
Methyl butanol
```

文字列の中に含まれている空白文字が取り除かれていないことに注意して下さい。string.strip は、文字列の先頭と末尾の空白文字だけを取り除きます。

string.strip を使えば、ファイルからデータを読み出すときに正しい出力を作れるようになります。

lists/fileinputloop_strip.cmd

```
>>> file = open('data.txt', 'r')
>>> for line in file:
...     line = line.strip()
...     print len(line)
...
7
```

```
5
5
4
```

5.10.1 コマンドライン引数

data.txt ファイルには惑星の名前が格納されています。このサンプルの締めくくりとして、ファイルは読み出すものの、特定の範囲の行だけを表示するようにしてみましょう。プログラムを実行するときに、先頭と末尾の行番号を指定します。たとえば、最初は1行目から3行目、次のときは2行目から4行目までを読み出したいものとします。

先頭と末尾は、コマンドライン引数で指定できます。プログラムを実行するときには、関数やメソッドに引数を渡せるのと同じように、プログラムに引数を渡すことができます。これらの値は、システムモジュール sys の argv という特殊変数に格納されます。argv は、コマンドライン引数(文字列型)のリストです。

sys.argv[0] は、常に実行される Python プログラムの名前です。この場合は、read_lines_range.py です。コマンドライン引数の残りの部分は、sys.argv[1]、sys.argv[2] などとして渡されます。

次のプログラムは、ファイルからすべてのデータを読み出し、先頭から末尾までの範囲の行だけを表示します。

lists/read_lines_range.py

```
'''指定された先頭から末尾までのdata.txtの行を表示します。

呼び出し形式: read_lines_range.py 先頭行 末尾行'''

import sys

if __name__ == '__main__':

    # 先頭と末尾の行番号の取得
    start_line = int(sys.argv[1])
    end_line = int(sys.argv[2])

    # ファイルの各行を読み出してリストに格納
    data = open('data.txt', 'r')
    data_list = data.readlines()
    data.close()

    # 先頭から末尾までの範囲の行を表示
    for line in data_list[start_line:end_line]:
        print line.strip()
```


5.11 コメント

今の行読み出しプログラムは、私たちがこれまで見てきたプログラムの中で最長のものです。長いので、docstring だけでなく、**コメント**(comment)も追加しました。docstring は、主としてプログラムを使う人のためのもので、プログラムが何をしてくれるのかを説明しますが、どのようにそれを実現するかは説明しません。

それに対し、コメントは、将来コードを見る開発者のために書かれます†。コメントは、先頭が # 文字でその行の末尾まで続きます。Python はコメントを完全に無視しますので、コメントには何を書いてもかまいません††。

優れたコメントを書くためのルールをまとめておきます。

- 読者があなたと同じくらい Python のことを知っているという前提で書きます(たとえば、文字列とは何かとか、代入文が何をするかといったことは説明しないようにします)。
- 当たり前のことをコメントしないようにします。たとえば、次のコメントは意味がありません。

```
count = count + 1 # countに1を加える
```

- これから書いたり、修正して磨き上げたりすべき部分を思い出せるようにするために、コード内に「TODO」とか「FIXME」で始まるコメントを残すことがよくあります。
- コードを書くときによく考えなければならなかった場合、その部分にはコメントを書き、次にコードを読む人が同じ思考を繰り返さなくても済むようにすべきです。特に、簡単な箇条書きの説明からスタートして、1つ1つの記述をコードになるまで磨き抜くという方法でプログラムや関数を開発する場合には、その箇条書きをコメントとして残すとよいでしょう(このスタイルの開発については、「10章　アルゴリズム」で詳しく説明します)。
- 同様に、バグが見つけにくかった場合や、フィックスが難しかった場合には、それを説明するコメントを書くべきです。書いておかなければ、次にプログラムのその部分を担当するプログラマは、コードが不必要に複雑だと考え、せっかくあなたががんばって作ったコードを解体してしまいます。
- 逆に、コードがしていることを説明するためにコメントをたくさん書かなければならない場合には、コードをクリーンアップすべきです。たとえば、関数内の15個のリストが何のために必要なのかを読者に絶えず思い出してもらわなければならないような場合、その関数を小さな部品に分割し、それぞれがリストの一部だけで動作するようにすべきです。

そして、もう1つルールを追加しておきます。

- 古くなったコメントが残っているコードは、コメントがまったく書かれていないコードよりも

† 将来のあなた自身を含みます。プログラムに変更を加えたり、バグフィックスが必要になったりしたときに、あなた自身もプログラムの詳細を忘れている場合があります。

†† 訳注：ただし、ここに示したように日本語のコメントを入れる場合には、先頭行としてエンコード宣言(たとえば、#encoding: sjis)を入れなければエラーになります。詳しくは、「4.2.1　インポートのときに行われていること」の訳注を参照して下さい。

かえって悪質です。コメントをよく読み、正確さに欠ける場合には書き直すようにしなければなりません。

5.12 まとめ

この章では、次のことを学びました。

- リストは0個以上のオブジェクトを管理するために使われます。リストに含まれているオブジェクトのことをリストの要素と呼び、0からリストの長さマイナス1までの添字という位置情報を使って要素を参照します。
- リストはミュータブルです。つまり、リストの内容は書き換えられます。リストは他のリストを含む任意の型のデータを格納できます。
- 元のリストと同じ値を持つ新しいリストや一部の値だけで作られた新リストを作るには、スライシングを使います。
- 2つの変数が同じオブジェクトを参照するとき、それらをエイリアス(別名)と呼びます。
- Python シーケンスには、タプルというものもあります。タプルはリストとほぼ同じですが、イミュータブルだという違いがあります。
- ファイルをオープンして読み出すとき、その内容は文字列のリストに格納されるのが普通です。

5.13 練習問題

練習問題で自分の力を試してみましょう。

1. alkaline_earth_metals という変数に6種類のアルカリ土類金属の元素番号(ベリリウム =4、マグネシウム =12、カルシウム =20、ストロンチウム =38、バリウム =56、ラジウム =88)を格納するリストを代入して下さい。

2. ラジウムの原子番号にアクセスするにはどの添字を使ったらよいですか。正の添字と負の添字とで2通りの答えを書いて下さい。

3. alkaline_earth_metals に含まれる要素数を教えてくれるのは、どの関数ですか。

4. alkaline_earth_metals でもっとも高い原子番号を返すコードを書いて下さい(ヒント:表5.2のリスト関数を使います)。

5. print 'a' と書くのと、print 'a', と書くのとではどのような違いがありますか。

6. 「5.5 スライシング」の half_lives リストに含まれるすべての値を1行に1つずつ表示するforループを書いて下さい。

7. 「5.5 スライシング」の half_lives リストに含まれるすべての値を同じ行に表示するforループを書いて下さい。

8. アジアの国々(および地域)の人口(単位は100万人)のリストを作る次の文について考えてみましょう。

```
country_ populations = [1295, 23, 7, 3, 47, 21]
```

すべての値を合計して total 変数に結果を格納する for ループを書いて下さい(ヒント:total に初期値0を与えてから、ループ本体で現在の国の人口を total に加算します)。

9. 25.2、16.8、31.4、23.9、28、22.5、19.6 という値を持つ摂氏の気温のリストを作り、temps 変数に代入して下さい。

10. あるリストメソッドを使って、temps を昇順にソートして下さい。

11. スライシングを使って、それぞれ20度未満、20度以上の気温を集めた cool_temps と warm_temps という2つの新しいリストを作って下さい。

12. リスト演算を使って、cool_temps と warm_temps を結合し、temps_in_celsius という新しいリストを作って下さい。

13. temps_in_celsius の値をすべて華氏に変換し、変換後の値を temps_in_fahrenheit という新しいリストに格納する for ループを書いて下さい。

14. アルカリ土類金属の原子番号と原子量のリストを要素とするネストされたリストを作って下さい。値はベリリウムが4と9.012、マグネシウムが12と24.305、カルシウムが20と40.078、ストロンチウムが38と87.62、バリウムが56と137.327、ラジウムが88と226です。

15. 元素ごとに原子番号と原子量を1行に並べた形で、alkaline_earth_metals のすべての値を表示する for ループを書いて下さい。

16. alkaline_earth_metals の要素を同じ順序でネストしない形で格納する number_and_weight という新しいリストを作る for ループを書いて下さい。

17. 次のような内容の alkaline_metals.txt ファイルがあるものとします。

```
4 9.012
12 24.305
20 40.078
38 87.62
56 137.327
88 226
```

alkaline_metals.txt を読み出し、原子番号と原子量のリストを要素とするネストされたリストに、その内容を格納する for ループを書いて下さい(ヒント:string.split を使いましょう)。

18. 次の文の効果を示すメモリモデルを描いて下さい。

```
values = [0, 1, 2]
values[1] = values
```

19. 次の関数には、docstringやコメントがありません。関数が何をどのようにしているのかを次の担当者が簡単に理解できる程度にdocstringとコメントを追加して下さい。次に、少なくとも他の2人の解答と自分の解答を比較してみて下さい。どのくらい似ていますか。違いが出るのはなぜでしょうか†。

```
def mystery_function(values):
    result = []
    for i in range(len(values[0])):
        result.append([values[0][i]])
        for j in range(1, len(values)):
            result[-1].append(values[j][i])
    return result
```

20. 「5.2　リストの書き換え」で、文字列はイミュータブルだと説明しました。ミュータブルな文字列があったとして、それが役に立つ理由はどのようなものでしょうか。Pythonが文字列をイミュータブルにしたのはなぜだと思いますか。

21. 数値と文字列が混ざっている[1, 'a', 2, 'b']というようなリストをソートするとどうなりますか。これは、「3章　文字列」の規則や、「6章　条件分岐」で説明している数値と文字列に対する<などの比較演算子の動作規則に合致したものだと言えますか。これは、Pythonの動作として「正しい」でしょうか、それとも他の動作の方が役に立つでしょうか。

† 訳注：range関数については7章を参照して下さい。range(len(values[0]))は0からvalues[0]よりも小さい最大の整数までのすべての整数のリストを返します。range(1, len(values))は1からlen(values)よりも小さい最大の整数までのすべての整数のリストを返します。

6章
条件分岐

この章では、制御構造というプログラミングの重要な基礎概念の1つを説明します。操作しているデータによってプログラムのふるまいを変えたいときには、かならずこれをしなければなりません。たとえば、水溶液が酸性かアルカリ性かによって異なることをする場合などに使うものです。

この章で取り上げる構文は、コンピュータがプログラムをどのように実行するかを左右するため、**制御フロー**（control flow）、あるいは制御構文と呼ばれます。制御フローの1つはすでにこの本でも登場しています。それは、「5.4　リストの要素の処理」で取り上げたループです。また、この後の章で取り上げる制御構文もあります。これらは、プログラムに「個性」を与える存在です。

フロー制御を掘り下げていく前に、真偽値を表現するPythonの型を紹介しておかなければなりません。今までに学んだ整数、浮動小数点数、文字列とは異なり、この型は2種類の値と3種類の演算子しか持ちませんが、非常に強力です。

6.1　ブール論理

1840年代に、数学者ジョージ・ブールは、「真」と「偽」の2つの値だけを使った純粋に数学的な形式で古典的な論理規則が表現できることを示しました。1世紀後、クロード・シャノン（その後、情報理論の発明者となる人物）は、ブールの業績を活用すれば電気機械時代の電話交換機の設計を刷新できることを示しました。コンピュータ回路の設計で**ブール論理**（Boolean logic）を使うのも、彼の業績に直接結び付いています。

ブールの業績をたたえるために、現代のほとんどのプログラミング言語は、真偽を管理するための型に、彼にちなんだ名前を付けています。

Pythonでは、その型はboolと呼ばれています（「e」を取り除いた形）。無数の値があり得るintやfloatとは異なり、boolはTrueとFalseの2通りの値しかありません。TrueとFalseは、0や-43.7などの数値と同じように値です。「true」や「false」は、日常会話では他の言葉に適用される形容詞ですから、このように考えるのは最初は変な感じがするかもしれませんが、プログラムではTrueとFalseを名詞として扱うのが自然な形です。