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く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。

6.1.1 ブール論理の演算子

bool の基本演算子は、and、or、not の 3 種類だけです。もっとも優先順位が高いのは not で、and、or の順序で続きます。

not は単項演算子です。言い換えれば、-(3 + 2) という式のマイナス記号と同じように、1 個の値に適用されます。not が含まれている式は、元の値が False なら True、元の値が True なら False を返します。

cond/boolean_not_examples.cmd

```
>>> not True
False
>>> not False
True
```

上の例の not True は False、not False は True と言えばよいことです。ですから、ブール値に直接 not を適用することはあまりありません。普通は、ブール変数や複雑なブール式に not を適用します。同じことが and、or 演算子にも当てはまります。ですから、ここでのサンプルではブール定数にブール演算子を適用していきますが、節の最後では普通の使い方がどうなるかがわかるような例を見ていきます。

and は 2 項演算子です。left and right という式は、left も right も**ともに** True であれば True になり、それ以外の場合は False になります。

cond/boolean_and_examples.cmd

```
>>> True and True
True
>>> False and False
False
>>> True and False
False
>>> False and True
False
```

or も 2 項演算子です。**どちらか**の被演算子が True なら True を返し、両方とも False のときに限り False を返します。

cond/boolean_or_examples.cmd

```
>>> True or True
True
>>> False or False
False
>>> True or False
True
>>> False or True
True
```


この定義は、どちらか片方だけでなく、両方とも真も含むため、**包含的論理和** (inclusive or) と呼ばれます。しかし、英語の or は、**排他的論理和** (exclusive or) の意味になることもあります。たとえば、「ピザかタンドリーチキンが選べる」と言われたとき、両方を選べるという意味ではないでしょう。しかし、Python は、他のプログラミング言語の大半と同様に、or を包含的論理和の意味で解釈します。排他的論理和の作り方は、練習問題で取り上げます。

先ほど、ブール演算子は普通ブール定数ではなく、ブール式に適用されると言いました。cold と windy の 2 つの変数を使って「風が強く、寒いわけではない」ということを表現したい場合、まず自然言語の式が持つあいまいさを取り除かなければなりません。「寒くないけれども風が強い」のか「寒くないし風も強くない」のかということです。図 6.1 は、両方の意味で作った真理表で、次のコードは、同じことを Python に翻訳したコードです。

cold	windy	(not cold) and windy	not (cold and windy)
True	True	False	False
True	False	False	True
False	True	True	True
False	False	False	True

図6.1 関係、等価演算子

cond/boolean_expression.cmd

```
>>> (not cold) and windy
>>> not (cold and windy)
```

6.1.2 関係演算子

先ほど、True と False は値だと言いました。プログラムで使われているブール値の大半は、直接書き出して作ったものではなく、式の中で作り出されたものです。そして、ブール値を作り出す式としてもっとも一般的なのが、**関係演算子** (relational operator) を使った比較です。たとえば、3<5 は関係演算子 < を使った比較式で値は True、13 ≥ 77 は、≥を使った比較式で値は False です。

表 6.1 に示すように、Python は一般に使われている演算子をすべてサポートしています。ただし、≥ではなく >= が使われているように、1 字ではなく 2 字で演算子を表現しているものもあります。

表6.1 関係、等価演算子

記号	演算
>	より大きい
<	より小さい (未満)
>=	以上
<=	以下
==	等しい
!=	等しくない

表現の規則でもっとも重要なのが、等価演算子として = ではなく、== を使っていることです。こ

れは、=が代入のために使われているからです。初心者は、つい両方を混同して変数xが3かどうかを確かめるためにx = 3と書きがちです。こうすると、構文エラーになりますが、何に目を付けたらよいのかがわからなければ、理由がなかなかわからず苦労しますので注意して下さい。

関係演算子は、すべて2項演算子です。2つの値を比較して、TrueかFalseの値を生み出します。「より大きい」の>と「より小さい」の<は、予想通りの動作をします。

cond/relational_1.cmd

```
>>> 45 > 34
True
>>> 45 > 79
False
>>> 45 < 79
True
>>> 45 < 34
False
```

すべての関係演算子が整数と浮動小数点数を比較できます。このとき、整数は、23.3に14を加えるときと同じように、自動的に浮動小数点数に変換されます。

cond/relational_2.cmd

```
>>> 23.1 >= 23
True
>>> 23.1 >= 23.1
True
>>> 23.1 <= 23.1
True
>>> 23.1 <= 23
False
```

同じことが「等しい」と「等しくない」にも当てはまります。

cond/relational_3.cmd

```
>>> 67.3 == 87
False
>>> 67.3 == 67
False
>>> 67.0 == 67
True
>>> 67.0 != 67
False
>>> 67.0 != 23
True
```

もちろん、あらかじめ知っている2つの数値を比較しても、結果は最初からわかっていますから、

あまり意味がありません。ですから、関係演算子はほとんどかならず、次のように、変数とともに使われます。

cond/relational_var.cmd

```
>>> def positive(x):
...     return x > 0
...
>>> positive(3)
True
>>> positive(-2)
False
>>> positive(0)
False
```

6.1.3 比較の結合

私たちは今までに算術、ブール、関係の3種類の演算子を見てきました。これらを組み合わせて使うための規則をまとめておきましょう。

- 算術演算子は関係演算子よりも優先順位が高くなっています。たとえば、+と/は、<や>よりも先に評価されます。
- 関係演算子はブール演算子よりも優先順位が高くなっています。たとえば、比較は、andやorよりも先に評価されます。
- すべての関係演算子は、優先順位が同じです。

以上のルールから、1 + 3 > 7という式は、1 + (3 > 7)ではなく、(1 + 3) > 7と評価されます。このルールは、複雑な式でもたいていかっこが不要になるように作られているとも言えます。

cond/skipping_parens.cmd

```
>>> x = 2
>>> y = 5
>>> z = 7
>>> x < y and y < z
True
```

しかし、かっこを付けると下位式が見つけやすくなり、読者に演算の順序を明確に伝えられますので、普通はかっこを付けるようにした方がよいでしょう。

cond/parens_included.cmd

```
>>> (x < y) and (y < z)
True
```

数学では、値が特定の範囲に含まれているかどうか、つまり他の2つの値の間にあるかどうかをチェックすることがよくあります。Pythonでは、andで比較式を結合してこれを実現します。

cond/compare_range.cmd

```
>>> x = 3
>>> (1 < x) and (x <= 5)
True
>>> x = 7
>>> (1 < x) and (x <= 5)
False
```

しかし、この形は非常によく見られますので、Python では比較の**連鎖**(chain)を書けるようになっています。

cond/chain1.cmd

```
>>> x = 3
>>> 1 < x <= 5
True
```

ほとんどの組み合わせは、普通に予想した通りに動作しますが、まれにびっくりするようなものも含まれています。

cond/chain2.cmd

```
>>> 3 < 5 != True
True
>>> 3 < 5 != False
True
```

両方の式がともに True になるのはおかしい感じがします。しかし、第 1 の式が次の意味であるのに対し、

```
(3 < 5) and (5 != True)
```

第 2 の式は次の意味になります。

```
(3 < 5) and (5 != False)
```

5 は True でも False でもないので、どちらの式でも後半部分は True になり、式全体では True になるのです。

この種の式は、文法的に間違っていなくても、よくない式の例です†。比較の連鎖は、数学者にとって自然に見える場合のみ、つまり < と <=、または > と >= という組み合わせだけで使うようにすべきです。他の組み合わせを使ってみたくなっても、避けて下さい。and で単純比較を結合するのは、コードを読みにくくならないようにするためです。また、書いているコードの意味に少しでも不明確なところがあれば、ちゅうちょなくかっこを使うようにしましょう。

† 2個の巨大なクリームパフェを立て続けに食べた直後にジェットコースターに乗るようなものです。

6.1.4 ブール演算子による整数、浮動小数点数、文字列の操作

すでに説明したように、int と float が併用されている式では、Python が自動的に int を float に変換します。それと同じように、Python は数値を bool にも変換します。つまり、3 種類のブール演算子は、数値にも直接適用できます。その場合、0 と 0.0 は False、それ以外のすべての数値は True として扱われます。

cond/not.cmd

```
>>> not 0
True
>>> not 1
False
>>> not 5
False
>>> not 34.2
False
>>> not -87
False
```

and と or がかかわってくると、話が複雑になります。Python がどちらかの演算子を含む式を評価するとき、左から右に評価をします。そして、まだ見ていない被演算子が残っていても、全体の評価ができるだけの情報が得られれば、そこで評価をやめます。結果として返されるのは、最後に評価されたもので、かならずしも True や False ではありません。

口で説明するよりも、具体例を見た方がわかりやすいでしょう。and を使った式が 3 つあります。

cond/and.cmd

```
>>> 0 and 3
0
>>> 3 and 0
0
>>> 3 and 5
5
```

最初の式では、Python は False と評価される 0 を見ると、ただちに評価をやめます。式全体が True と評価されるのは両方の被演算子が True のときだけなので、次の 3 を見なくても、式全体が False と評価されることが明らかだからです(図 6.2 参照)。

しかし、第 2 の式の場合、最初の被演算子 (3) が False でないことがわかっただけでは式全体の値がどうなるかわからないので、両方の被演算子を評価しなければなりません。Python は、第 3 の式でも両方の被演算子を評価します。そして、式全体の値として、最後にチェックしたものの値を返します(この場合は 5)。

or の場合は、第 1 被演算子が True なら、第 2 被演算子をチェックせずに、or は式全体の値を判断します。なぜかと言うと、Python はすでに答えを知っているからです。True or X は、X の値が

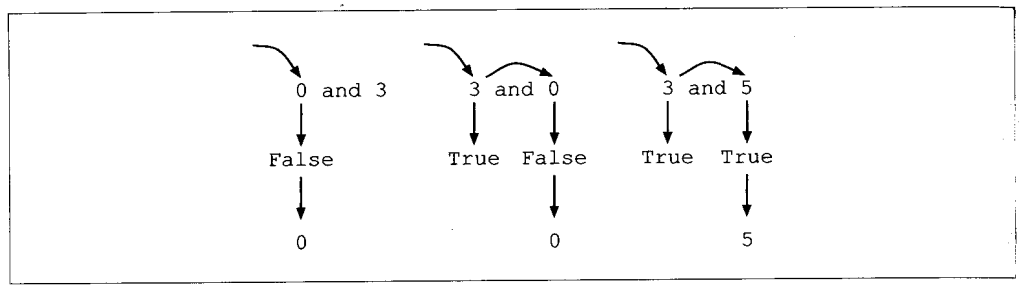

図6.2 短絡評価

何であれ True です。

しかし、第1被演算子が False なら、or は第2被演算子を評価しなければなりません。式全体の結果は、第2被演算子を評価した結果と等しくなります（コンピュータは、かならずしも小数を正確に表現できないということを思い出して下さい。次のコードの最後の値は、18.2 にできる限り近い値を探した結果です）。

cond/or.cmd

```
>>> 1 or 0
1
>>> 0 or 1
1
>>> True or 0
True
>>> 0 or False
False
>>> False or 0
0
>>> False or 18.2
18.199999999999999
```

or の第1被演算子が True なら、or は第2被演算子を評価すらせずに、すぐに式全体の値を評価すると言いましたが、本当にそうだということを確かめてみましょう。0 で除算する式を試してみると、次のようになります。

cond/div_zero.cmd

```
>>> 1 / 0
Traceback (most recent call last):
  File "<string>" , line 1, in <string>
ZeroDivisionError: integer division or modulo by zero
```

この式を or の第2被演算子として使ってみましょう。

cond/or_lazy.cmd

```
>>> True or 1 / 0
True
```

第1被演算子が True なので、第2被演算子は評価されず、そのためコンピュータは 0 による除算を試したりはしないのです。

世の中には賢すぎるということがある

y = x and 1/x のような式はエラーを起こさずに動作しますが、だからといってそういう式を使うべきだということにはなりません。まして、次のような式を

```
result = test and first or second
```

次のコードの略記法として使うのは避けるべきです。

```
if test:
    result = first
else:
    result = second
```

プログラムは読んで意味のわかるものでなければなりません。1行のコードに悩んだり、初めて見た誰かが誤解しそうなコードは、正しく実行されたとしても、よくないコードです。

数値を比較できるのと同じように、文字列を比較することもできます。文字列の中の文字は、整数で表現されています。たとえば、大文字の A は 65、スペースは 32、小文字の z は 172 です†。Python は、左から右に対応する文字の整数値を比較して、どちらの文字列の方が大きいかを判断します。片方の文字列の文字がもう片方の文字列の文字よりも大きければ、第1の文字列の方が第2の文字列よりも大きいということになります。すべての文字が等しければ、2つの文字列は同じです。比較の途中で片方の文字列の文字がなくなったら（つまり、その文字列の方が短ければ）、短い方が小さいと評価されます。いくつかの評価例を見てみましょう。

cond/string_compare.cmd

```
>>> 'A' < 'a'
True
>>> 'A' > 'z'
```

† このエンコーディング（符号化方式）は ASCII（American Standard Code for Information Interchange の略）と呼ばれます。この符号化方式では、すべての大文字がどの小文字よりも前にあるため、大文字の Z が小文字の a よりも小さいという特徴があります。

```
False
>>> 'abc' < 'abd'
True
>>> 'abc' < 'abcd'
True
```

空文字列は、0と同様にFalseと評価されます。他の文字列は、すべてTrueと評価されます。

cond/empty_false.cmd

```
>>> '' and False
''
>>> 'salmon' or True
'salmon'
```

Pythonは、ブール値を数値に変換することもできます。Trueは1に、Falseは0になります。

cond/truefalse.cmd

```
>>> False == 0
True
>>> True == 1
True
>>> True == 2
False
>>> False < True
True
```

そのため、ブール値を加減乗除できるということになります。

cond/bool_math.cmd

```
>>> 5 + True
6
>>> 7 - False
7
```

しかし、「できる」からといって「すべきだ」というわけではありません。5にTrueを加えたり、気温にcurrent_time<Noonを掛けたりすれば、コードはとても読みにくくなるでしょう。実際のプログラミングでは、ブールへの変換は非常によく使われますが、逆方向の変換が使われることはほとんどありません。

6.2 if文

if文の基本形式は、次の通りです。

```
if condition:
    block
```

condition（条件）は、name !='' やx < yなどの式です。ブール式である必要はないということに注意して下さい。「6.1.4 ブール演算子による整数、浮動小数点数、文字列の操作」で説明したように、ブール以外の値は、必要なときに自動的にTrueかFalseに変換されます。

特に、0、None、空文字列の''、空リストの[]は、どれもFalseと見なされるのに対し、それ以外の値はどれもTrueと見なされることに注意して下さい。

条件がTrueなら*block*（ブロック）の中の文が実行され、そうでなければ実行されません。ループや関数のブロックと同様に、このブロックはif文に属すことを表すためにインデントしなければなりません。インデントが適切でなければ、Pythonはエラーを起こすことがあります。もっと悪いのは、Pythonがコードを実行してしまう場合です。この場合、インデントが間違っているためにあなたの意図とは異なった形で実行されてしまいます。どちらの場合についても、後で簡単な具体例をお見せします。

表6.2に示すのは、pHの値による水溶液の分類表です。

表6.2 pH値による水溶液の分類

pHレベル	分類
0～4	強酸性
5～6	弱酸性
7	中性
8～9	弱アルカリ性
10～14	強アルカリ性

if文を使えば、ユーザーが入力したpHレベルが酸性のときに限り、何らかのメッセージを出力することができます（「3.6 ユーザー入力」で説明したように、比較をするためには、まずユーザー入力を文字列から浮動小数点数に変換しなければならないことを思い出して下さい）。

cond/if_basictrue.cmd

```
>>> ph = float(raw_input())
6.0
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
...
6.0は酸性です。
```

条件がFalseなら、ブロック内の文は実行されません。

cond/if_basicfalse.cmd

```
>>> ph = float(raw_input())
8.0
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
...
>>>
```

ブロックをインデントしなければ、インデントが必要だということを Python が教えてくれます。

cond/if_indenterror.cmd

```
>>> ph = float(raw_input())
6.0
>>> if ph < 7.0:
... print "%sは酸性です。" % (ph)
  File "<stdin>", line 2
    print "%sは酸性です。" % (ph)
        ^
IndentationError: expected an indented block
```

ブロックを使っていますので、複数の文を指定できます。これらの文は、どれも条件が True のときに限り実行されます。

cond/if_multilinetrue.cmd

```
>>> ph = float(raw_input())
6.0
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
...     print "それに対しては注意が必要です!"
...
6.0は酸性です。
それに対しては注意が必要です!
```

ブロックの先頭行をインデントすると、Python インタープリタはブロックの末尾まで、プロンプトを ... に変更します。ブロックの末尾は、空行で知らせます。

cond/if_multiline_indent_error.cmd

```
>>> ph = float(raw_input())
8.0
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
...
>>> print "それに対しては注意が必要です!"
それに対しては注意が必要です!
```


ブロック内のコードをインデントしないと、インタープリタはエラーを起こします。

cond/if_multiline_indent_error2.cmd

```
>>> ph = float(raw_input())
8.0
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
... print "それに対しては注意が必要です!"
  File "<stdin>", line 3
    print "それに対しては注意が必要です!"
        ^
SyntaxError: invalid syntax
```

プログラムがファイルにまとめられている場合、空行は不要です。インデントの終了とともに、Python はブロックも終了したと判断します。

cond/if_multiline_indent_error3.cmd

```
ph = 8.0
if ph < 7.0:
    print "%sは酸性です。" % (ph)
print "それに対しては注意が必要です!"
```

この小さな不一致が問題になることはなく、ほとんどの人は、このような違いがあることに気付きもしません。

もちろん、判断が 1 つだけでは足りない場合があります。チェックしなければならない基準が複数ある場合の処理方法は、複数あります。まず、複数の if 文を使う方法があります。たとえば、pH レベルが酸性かアルカリ性かによって異なるメッセージを表示したいものとします。

cond/multi_if.cmd

```
>>> ph = float(raw_input())
8.5
>>> if ph < 7.0:
...     print "%sは酸性です。"  % (ph)
...
>>> if ph > 7.0:
...     print "%sはアルカリ性です。"  % (ph)
...
8.5はアルカリ性です。
>>>
```

図 6.3 に示すように、実行できるブロックは 1 つだけだということがわかっているのに、両方の条件がかならず評価されています。elif キーワード(「else if」という意味です)を使って条件 / ブロックの対を追加すれば、両方の条件を 1 つにまとめることができます。1 つ 1 つの条件 / ブロックを

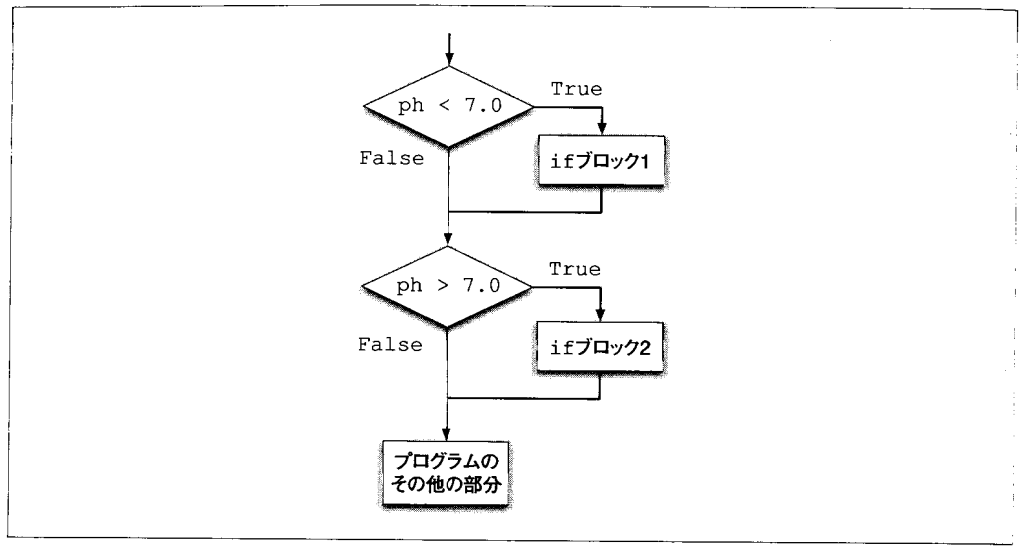

図6.3 if文

節(clause)と呼びます。

cond/elif_basic.cmd

```
>>> ph = float(raw_input())
8.5
>>> if ph < 7.0:
...     print "%sは酸性です。" % (ph)
... elif ph > 7.0:
...     print "%sはアルカリ性です。" % (ph)
...
8.5はアルカリ性です。
>>>
```

2つのコードの違いは、elifなら、その上のifがFalseだったときに限り実行されることです。図6.4を見れば、違いが一目でわかります。この図では、条件が菱形、四角はその他の文、矢印が制御フローを表しています。

if文には複数のelif節を付けることができます。次の少し長いサンプルは、化学式を日本語の名前に変換します。

cond/elif_longer.cmd

```
>>> compound = raw_input()
CH3
>>> if compound == "H2O":
...     print "水"
... elif compound == "NH3":
...     print "アンモニア"
... elif compound == "CH3":
...     print "メタン"
...
メタン
>>>
```


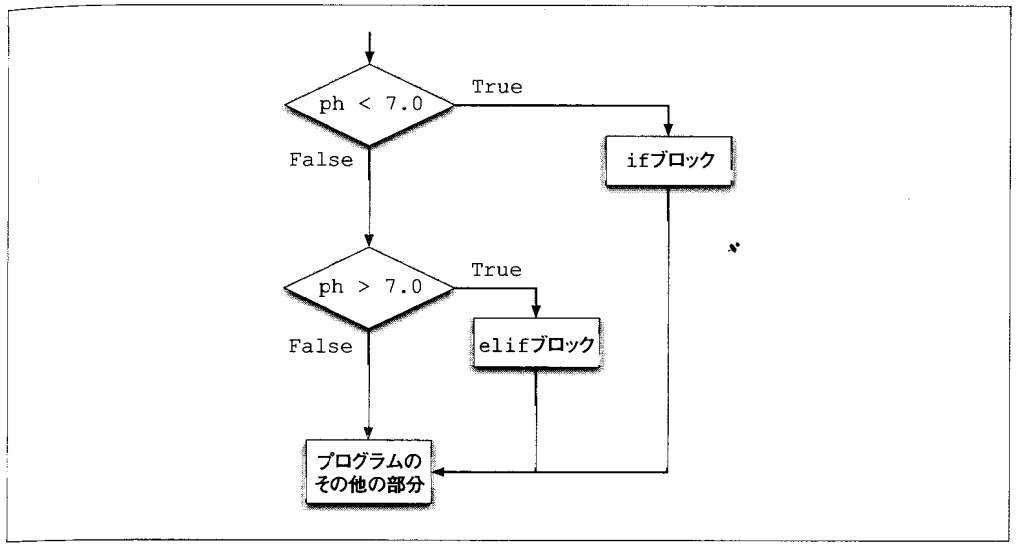

図6.4 elif文

if/elif文のどの条件も満たされなければ、Pythonはどのブロックも実行しません。しかし、それでは不便に感じることがあります。先ほどの例で言えば、知らない化学式が与えられた場合でも何かしら出力したいところです。連鎖の末尾にelse節を追加すれば、それを実現できます。

cond/else_basic.cmd

```
>>> compound = raw_input()
H2SO4
>>> if compound == "H2O":
...     print "水"
... elif compound == "NH3":
...     print "アンモニア"
... elif compound == "CH3":
...     print "メタン"
... else:
...     print "知らない化合物"
...
知らない化合物
>>>
```

if 文が持てる else 節は 1 つだけで、else 節はその文の最後の節でなければなりません。else には条件が付けられていないことに注意して下さい。論理的には、次の文は、

```
if condition:
    if-block
else:
    else-block
```

次の文と同じです。

```
if condition:
    if-block
if not condition:
    else-block
```

6.2.1 if文のネスト

if 文のブロックには、Python のあらゆるタイプの文を入れられます。そのため、ブロックに他の if 文を入れることもできます。if 文の中に別の if 文を書くことを if 文のネストと言います。

cond/nested_if.cmd

```
input = raw_input()
if len(input) > 0:
    ph = float(input)
    if ph < 7.0:
        print "%sは酸性です。" % (ph)
    elif ph > 7.0:
        print "%sはアルカリ性です。" % (ph)
    else:
        print "%sは中性です。" % (ph)
else:
    print "pH値が与えられていません!"
```

このコードは、ユーザーに pH 値の入力を求めますが、最初は文字列形式で受け取ります。最初の（**外側の**）if 文は、ユーザーが何かを入力しているかをテストします。そして、入力しているときに限り、**内側の** if 文で pH の値を解析します。

if 文のネストが必要になる場合もありますが、複雑でわかりにくくなりがちです。文が実行されるときのことを説明するために、頭の中で条件文を結び付けなければなりません。たとえば、「～は酸性です。」が実行されるのは、input 文字列の長さが 0 よりも大きく、pH < 7.0 が True と評価されるときだけだというようになってしまいます。

6.3 条件の保存

次のコードを見て、x にどんな値が格納されるのかを考えてみて下さい。

cond/assign_bool.cmd

```
>>> x = 15 > 5
```

「True だ」と思った人は正解です。15 は 5 よりも大きいので、比較式は True を生成します。そして、True は数値や文字列と同じような値ですから、変数に代入できるのです。

このようなコードは、主として決定表をソフトウェアに翻訳するときに必要になります。たとえば、年齢とボディマス指数（BMI）に基づく表 6.3 の規則に従って心臓病のリスクがどの程度あるかを計算したいものとします。

表6.3 年齢とボディマス指数(BMI)の関係

	45 歳未満	45 歳以上
BMI が 22.0 未満	低い	中くらい
BMI が 22.0 以上	中くらい	高い

たとえば if 文のネストを使えば、これをコード化できます。

```
if age < 45:
    if bmi < 22.0:
        risk = '低'
    else:
        risk = '中'
else:
    if bmi < 22.0:
        risk = '中'
    else:
        risk = '高'
```

しかし、この方法には、複数の場所で同じ条件をテストしていることがわかりにくいという問題点があります。たとえば、年齢と BMD で 4 つずつの区分がある場合には、内側の条件文は 16 個になりますので、それらがみな同じ条件だということは簡単にはわからないでしょう。

次のようにすれば、少しましになります。

```
young = age < 45
slim = bmi < 22.0
if young:
    if slim:
        risk = '低'
    else
        risk = '中'
else:
```

```
    if slim:
        risk = '中'
    else:
        risk = '高'
```

同じことを次のように書くこともできます。

```
young = age < 45
slim = bmi < 22.0
if young and slim:
    risk = '低'
elif young and not slim:
    risk = '中'
elif not young and slim:
    risk = '中'
elif not young and not slim:
    risk = '高'
```

整数に変換すればFalseは0、Trueは1になることを利用することもできます。

```
table = [['中', '高'],
         ['低', '中']]
young = age < 45
heavy = bmi >= 22.0
risk = table[young][heavy]
```

6.4 まとめ

この章では、次のことを学びました。

- Pythonは、ブール値のTrueとFalseを使って、何が真で何が偽かを表します。プログラムは、not、and、orの3つの演算子を使って、これらの値を結合できます。
- ブール演算子は数値にも適用できます。0と0.0はFalse、それ以外の数値はTrueと等しくなります。ブール値を数値に変換すると、Falseは0、Trueは1になります。
- 「等しい」、「より小さい」などの関係演算子は、値を比較してブール値を生成します。
- 1つの式の中でさまざまな演算子を組み合わせて使う場合、優先順位は高い方から順に、算術演算子、関係演算子、ブール演算子になります。
- ブロックを実行するかどうかの選択は、プログラムのふるまいを制御するためのもっとも基本的な方法の1つです。Pythonでは、if、elif、elseを使って、そのような選択を表現します。ifとelifは、論理式の値を基にして選択を行うのに対し、elseは他のすべてのテストが失敗したときに限り実行されます。

6.5 練習問題

練習問題で自分の力を試してみましょう。

1. 以下の式の値はそれぞれ何になるでしょうか。Pythonで式を入力して、自分の答えを確かめて下さい。
   a) `True and not False`
   b) `True or True and False`
   c) `not True or not False`
   d) `True and not 0`
   e) `52 < 52.3`
   f) `1 + 52 < 52.3`
   g) `4 != 4.0`

2. andのふるまいの定義として次のようなものを考えました。

   - 両方の被演算子がTrueなら、結果はその第2被演算子の値とします。
   - 被演算子のどちらかがFalseなら、結果はその第1被演算子の値とします。

   この規則は、Pythonが実際に使っているルールに合致しているでしょうか。そうでなければ、反例を示して下さい。

3. 変数xとyがあります。
   a) 両方の変数がTrueならTrue、そうでなければFalseと評価されるような式を書いて下さい。
   b) xがFalseならTrue、そうでなければFalseと評価される式を書いて下さい。
   c) 少なくとも1つの変数がTrueならTrue、そうでなければFalseと評価される式を書いて下さい。

4. full、empty変数があるとき、どちらか片方の変数がTrueならTrue、そうでなければFalseと評価される式を書いて下さい。

5. 明るさのレベルが0.01未満か、気温が氷点よりも上のどちらかなら、野生動物を撮るための自動カメラのスイッチをオンにし、両方の条件が満たされるときにはスイッチを入れないようにしたいものとします。そこで、まず次のようなコードを書きました。

```
if (light < 0.01) or (temperature > 0.0):
    if (light < 0.01) and (temperature > 0.0):
        pass
    else:
        camera.on()
```

友だちがこれを見て、「これは排他的論理和だから、次のようにすればもっと簡単に書けるよ」

と言いました。

```
if (light < 0.01) != (temperature > 0.0):
    camera.on()
```

友だちの言っていることは正しいでしょうか。もしそうなら、理由を説明して下さい。正しくない場合は、2つのコードで異なる結果になるような光と気温の値を用意して下さい。

6. 「2.7 組み込み関数」で、absという組み込み関数を取り上げました。変数xが与えられたときに、xとその絶対値が等しければTrue、そうでなければFalseと評価される式を書いて下さい。

7. aとbの2つの引数を取るdifferentという関数を書いて下さい。この関数は、aとbが異なる値を参照していればTrue、そうでなければFalseを返すものとします。

8. populationとland_areaという2つのfloat値が与えられているとします。
   a) 人口が1,000万以下のときに限り人口を表示するif文を書いて下さい。
   b) 人口が1,000万と3,500万の間なら人口を表示するif文を書いて下さい。
   c) 人口密度（単位面積当たりの人口）が100よりも大きければ「過密」と表示するif文を書いて下さい。
   d) 人口密度（単位面積当たりの人口）が100よりも大きければ「過密」、そうでなければ「過疎」と表示するif文を書いて下さい。

9. 「2.6 関数の基礎」で書いたto_celsius関数は、華氏を摂氏に変換します。しかし、Wikipediaでは、ケルビン、摂氏（セルシウス度）、華氏（ファーレンハイト度）、ランキン度、ドリー度、ニュートン度、レオミュール度、レーマー度の8種類の単位を取り上げています。これらについては、http://en.wikipedia.org/wiki/Comparison_of_temperature_scales、http://ja.wikipedia.org/wiki/セルシウス度を参照して下さい。
   a) source（元の）単位でtという温度をtarget（新しい）単位に変換するconvert_temperatures(t, source, target)関数を書いて下さい。source、targetとしては、"Kelvin"、"Celsius"、"Fahrenheit"、"Rankine"、"Delisle"、"Newton"、"Reaumur"、"Romer"の中の好きなものを指定できるものとします。
   ヒント：上記のWikipediaの英文ページには、縦7行横2列の表が8個含まれています。これをif文に変換すると、8種類の単位がそれぞれ他の7種類の単位に変換できるわけですから最低でも8×7という膨大な数のif文が必要になります。さらに悪いことに、他の単位を追加することになると、少なくとも16個のif文を追加することになります。新単位から8種類の既存単位への変換と8種類の既存単位から新単位への変換で、合わせて16個のif文が必要になります。
   それよりも、たとえば摂氏を基準スケールとして選んだ方がよいでしょう。変換関数は、source単位から摂氏への変換と摂氏からtarget単位への変換という2ステップで動作するように書くのです。
   b) 新しい単位を追加したとき、追加しなければならないif文はいくつですか。

10. pH値が3.0未満なら、強い警告メッセージを表示し、そうでなければただ酸性だと知らせるようにしたいものとします。そこで、次のif文を試してみます。

cond/elif_wrongorder.cmd

```
>>> if ph < 7.0:
...     print "%sは酸性です" % (ph)
... elif ph < 3.0:
...     print "%sは強酸性です。注意して下さい。" % (ph)
...
```

このコードでは、2.5というpHを入力すると、誤ったメッセージが出力されます。何が問題なのでしょうか。また、どうすれば直すことができるでしょうか。

11. 次のコードは、水溶液の酸性の度合いについてメッセージを表示します。

cond/acidity.cmd

```
ph = float(raw_input("ph値を入力して下さい: "))
if ph < 7.0:
    print "酸性です!"
elif ph < 4.0:
    print "強酸性です!"
```

   a) ユーザーが6.4と入力したら、どのようなメッセージが表示されますか。
   b) ユーザーが3.6と入力したら、どのようなメッセージが表示されますか。
   c) 4よりも小さな値を入力したときに、両方のメッセージが表示されるように、コードの中の1行を書き換えて下さい。

12. 「6.3 条件の保存」の最後の例は、誰かの体重が軽いかどうかではなく、重いかどうか（つまり、重さが境界値を超えているかどうか）を返しています。第2の代入文として、light = bmi < 22.0と書きたい場合、参照テーブルにはどのような変更を加えなければならないでしょうか。