4章
モジュール

数学者は、すべての定理を最初から証明するわけではありません。先輩たちがすでに確立した事実という基礎の上に自分たちの証明を積み上げていきます。同じように、1人の人がプログラム全体を自分で書くことは非常にまれになっています。他のプログラマが以前に書いた数百万行ものコードを活用する方がずっと一般的ですし、生産的でもあります。

モジュール(module)とは、1つのファイルにまとめられた関数のコレクションです。モジュール内の関数同士は、普通は何らかの形で互いに関係があります。たとえば、mathモジュールには、cos(コサイン、余弦)やsqrt(平方根)などの数学関数が含まれています。この章では、Python付属の豊かなモジュール群をどのようにして利用すればよいか、また自分独自の新しいモジュールを作るにはどうすればよいかを説明していきます。Pythonを使ってイメージファイルの内容を探ったり、表示したりする方法も学びます。

4.1 モジュールのインポート

科学論文で他人の業績を参照したいときには、参考文献リストでその仕事を引用しなければなりません。同じように、モジュールの関数を使いたいときには、モジュールを**インポート**(import)する必要があります。たとえば、mathモジュールの関数を使いたいということをPythonに伝えるには、次のようなimport文を使います。

modules/import.cmd

```
>>> import math
```

モジュールをインポートしたら、組み込みのhelp関数を使って、その内容を確かめることができます†。

† 対話的にこれを行った場合、Pythonは1度に1画面分の情報しか表示しません。次ページを表示するための「More」プロンプトが表示されたときには、スペースキーを押すとページを移れます。

modules/help_math.cmd

```
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.
    (このモジュールはいつでも利用できます。C標準で定義されている
    数学関数へのアクセスを提供します。)

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
        (xの逆余弦(単位はラジアン)を返します。)

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.
        (xの逆正弦(単位はラジアン)を返します。)
...
```

これで、私たちのプログラムは、すべての標準数学関数を使えるようになります。しかし、ここで平方根を計算しようとすると、Python が sqrt 関数を見つけられないというエラーが返されてしまいます。

modules/sqrt.cmd

```
>>> sqrt(9)
Traceback (most recent call last):
  File "<string>" , line 1, in <string>
NameError: name 'sqrt' is not defined
```

どうすればよいのでしょうか。間にドットをはさんでモジュール名と関数名をつなげ、math モジュール内で関数を探せと明示的に Python に指示するのです。

modules/sqrt2.cmd

```
>>> math.sqrt(9)
3.0
```


なぜ、このようにモジュール名と関数名を結合しなければならないのでしょうか。それは、同じ名前の関数を持つモジュールが複数存在する場合があるからです。たとえば、次の floor 呼び出しは、数値の小数点以下を切り捨てる math モジュールの関数と、面積を引数として価格を計算する building モジュール（完全に架空のものです）の関数のどちらを参照しているのでしょうか（図 4.1 参照）。

modules/import_ambiguity.cmd

```
>>> import math
>>> import building
>>> floor(22.7)
```

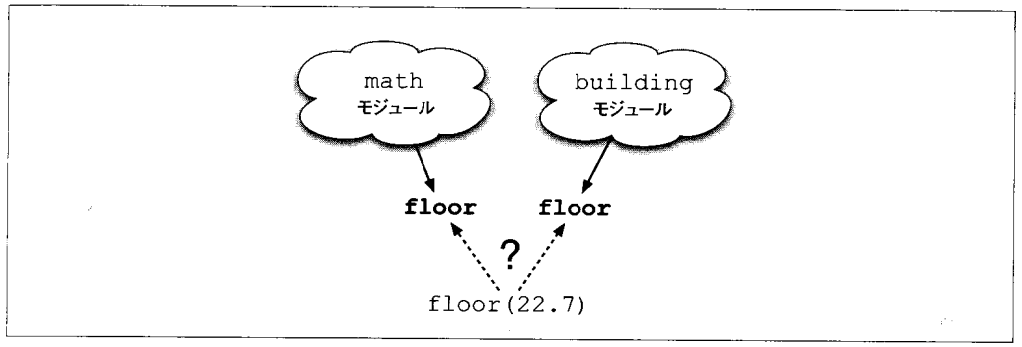

図4.1 importの仕組み

インポートされたモジュールは、プログラムが終了するまでメモリに居残ります。モジュールを「アンインポート」（メモリから消去）したり、プログラムの実行中に書き換えられたモジュールをインポートし直したりするための手段はありますが、ほとんど使われません。プログラムを止めて、再起動する方がほぼ間違いなく簡単です。

モジュールには関数以外のものを含めることもできます。たとえば、math モジュールは、pi などの変数も定義しています。モジュールをインポートすると、他の変数と同じようにこれらの変数も使えるようになります。

modules/pi.cmd

```
>>> math.pi
3.1415926535897931
>>> radius = 5
>>> print 'area is %6f' % (math.pi * radius ** 2)
area is 78.539816
```

モジュールからインポートされた変数に代入することさえ可能です。

modules/pi_change.cmd

```
>>> import math
>>> math.pi = 3 # 円を六角形に
>>> radius = 5
>>> print 'circumference is' , 2 * math.pi * radius
circumference is 30
```

しかし、こんなことをしてはいけません。πの値を変えるのは決してよいことではありません。そんなことをすべきではないということから、変数だけでなく、変更不能な定数を定義できるようにしている言語も多数あるくらいです。名前からもわかるように、定数の値は、定義後変更できないようになっています。πはいつでも3.14159と少しであり、SECONDS_PER_DAYは常に86,400です。このように値を「凍結」する手段を提供していないのは、Pythonが持つごく少数の欠点の1つです。

モジュール名とモジュール内に含まれているものの名前を結合する方法は安全ですが、不便に感じることもあります。そこで、次のようにすれば、インポートしたいものがモジュール内の何なのかをピンポイントで指定できます。

modules/from.cmd

```
>>> from math import sqrt, pi
>>> sqrt(9)
3.0
>>> radius = 5
>>> print '円周の長さは%6f' % (2 * pi * radius)
円周の長さは31.415927
```

この方法は、同じ名前の関数を異なるモジュールが提供しているときに問題を起こす場合があります。たとえば、magicというモジュールからspellという関数をインポートした後で、grammarというモジュールからspellという関数をインポートすると、第1の関数は第2の関数に押しのけられてしまいます。これは変数に新しい値を代入するときと同じで、最後に代入またはインポートされたものの勝ちになるのです。

モジュールに含まれているすべてのものをまとめてインポートできるimport *も、同じ理由から基本的に使わない方がよいでしょう。確かに、次のようにすれば、入力量が減ります。

modules/from2.cmd

```
>>> from math import *
>>> '%6f'  % sqrt(8)
'2.828427'
```

しかし、このようにモジュールをインポートしているプログラムは、モジュールに何かが追加されるたびに動かなくなるおそれがあります。

Pythonの標準ライブラリには、特定の日の曜日を調べるようなことから、Webサイトからデータを取り出すことまで、さまざまな機能を持つ数百種ものライブラリが含まれています。すべてのモジュールのリストは、http://docs.python.org/modindex.html で参照できます。1度に（あるいは1学期の講座で）すべて消化することはとてもできない量ですが、ライブラリを上手に使う方法がわかっているかどうかは、優れたプログラマとそうでないプログラマを分ける基準の1つです。

4.2 独自モジュールの定義

拡張子が.pyのファイルにコードを格納すれば、コードを保存して後で再利用できます。プロンプトに対して対話的にコマンドをいちいち入力しなくても、そのファイルに含まれているコードを実行するようにPythonに指示できるのです。しかし、話はそれだけでは終わりません。実は、すべてのPythonファイルがモジュールとして使えるのです。

モジュールの名前はファイル名から拡張子の.pyを取り除いたものです。

たとえば、「2.6 関数の基礎」で取り上げた次の関数について考えてみましょう。

modules/convert.py

```
def to_celsius(t):
    return (t - 32.0) * 5.0 / 9.0
```

この関数定義をtemperature.pyというファイルに格納し、さらにabove_freezingという関数も追加します。above_freezingは、引数の値が摂氏で氷点以上ならTrue、そうでなければFalseを返します。

modules/freezing.py

```
def above_freezing(t):
    return t > 0
```

おめでとうございます。これで、temperatureという名前のモジュールが完成しました（図4.2参照）。

modules/temperature.py

```
def to_celsius(t):
    return (t - 32.0) * 5.0 / 9.0

def above_freezing(t):
    return t > 0
```

ファイルを作ったら、他のモジュールと同じようにインポートできます。

modules/import_temp.cmd

```
>>> import temperature
>>> temperature.above_freezing(temperature.to_celsius(33.3))
True
```

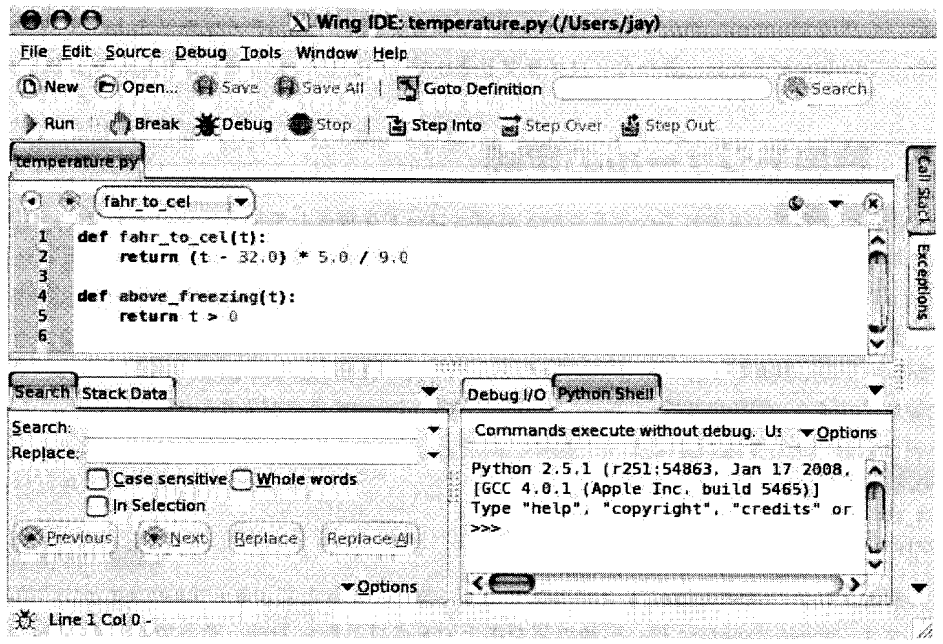

図4.2 Wing 101に読み込んだtemperatureモジュール

4.2.1 インポートのときに行われていること

それでは、別の実験をしてみましょう。experiment.pyというファイルに次のコードを格納します。

modules/experiment.py†

```
print "パンダの学名は'Ailuropoda melanoleuca'です。"
```

† 訳注：本書は入門書であり日本語版なので、print文で出力する文字列も極力日本語にするようにしています。しかし、Pythonはプログラムで使われている文字コードにとてもうるさい言語で、.pyファイルに漢字やひらがなを書き込んだだけで、たとえそれがコメントであっても、次のようなエラーを起こします。

```
  File "experiment.py", line 1
SyntaxError: Non-ASCII character '\x83' in file experiment.py on line 1, but no encoding
declared; see http://www.python.org/peps/pep-0263.html for details
```

「experiment.pyファイルの1行目にASCIIではない文字が含まれているのに、エンコーディングが宣言されていない」と言っているわけです。このエラーは、「エンコーディングを宣言」すれば解消されます。つまり、.pyファイルの先頭に次の1行を入れるのです。

```
#encoding: sjis
```

sjisの部分は、必要に応じてeucやutf8に置き換えて下さい。もちろん、sjisと指定した場合は、ファイルをShift-JISで書かなければなりません。以下のサンプルコードではいちいちこの行を入れませんが、.pyファイル内で日本語が使われている場合は、「エンコード宣言」が必要だということを忘れないで下さい。

__builtins__モジュール

Pythonの組み込み関数は、実際には__builtins__という名前のモジュールに含まれています。名前の前後に付けられたダブルアンダースコアは、それがPythonシステムの一部であることを示しています。この習慣については、また後で触れます。このモジュールに含まれているものは、help(__builtins__)を使えば見ることができます。一覧を見るだけでよければ、dirコマンドを使うこともできます(dirコマンドは他のモジュールにも使えます)。

modules/dir1.cmd

```
>>> dir(__builtins__)
['ArithmeticError' , 'AssertionError' , 'AttributeError' ,
'BaseException' , 'DeprecationWarning' , 'EOFError' , 'Ellipsis' ,
'EnvironmentError' , 'Exception' , 'False' , 'FloatingPointError' ,
'FutureWarning' , 'GeneratorExit' , 'IOError' , 'ImportError' ,
'ImportWarning' , 'IndentationError' , 'IndexError' , 'KeyError' ,
'KeyboardInterrupt' , 'LookupError' , 'MemoryError' , 'NameError' ,
'None' , 'NotImplemented' , 'NotImplementedError' , 'OSError' ,
'OverflowError' , 'PendingDeprecationWarning' , 'ReferenceError' ,
'RuntimeError' , 'RuntimeWarning' , 'StandardError' ,
'StopIteration' , 'SyntaxError' , 'SyntaxWarning' , 'SystemError' ,
'SystemExit' , 'TabError' , 'True' , 'TypeError' ,
'UnboundLocalError' , 'UnicodeDecodeError' , 'UnicodeEncodeError' ,
'UnicodeError' , 'UnicodeTranslateError' , 'UnicodeWarning' ,
'UserWarning' , 'ValueError' , 'Warning' , 'ZeroDivisionError' , '_' ,
'__debug__' , '__doc__' , '__import__' , '__name__' , 'abs' , 'all' ,
'any' , 'apply' , 'basestring' , 'bool' , 'buffer' , 'callable' ,
'chr' , 'classmethod' , 'cmp' , 'coerce' , 'compile' , 'complex' ,
'copyright' , 'credits' , 'delattr' , 'dict' , 'dir' , 'divmod' ,
'enumerate' , 'eval' , 'execfile' , 'exit' , 'file' , 'filter' ,
'float' , 'frozenset' , 'getattr' , 'globals' , 'hasattr' , 'hash' ,
'help' , 'hex' , 'id' , 'input' , 'int' , 'intern' , 'isinstance' ,
'issubclass' , 'iter' , 'len' , 'license' , 'list' , 'locals' , 'long' ,
'map' , 'max' , 'min' , 'object' , 'oct' , 'open' , 'ord' , 'pow' ,
'property' , 'quit' , 'range' , 'raw_input' , 'reduce' , 'reload' ,
'repr' , 'reversed' , 'round' , 'set' , 'setattr' , 'slice' , 'sorted' ,
'staticmethod' , 'str' , 'sum' , 'super' , 'tuple' , 'type' , 'unichr' ,
'unicode' , 'vars' , 'xrange' , 'zip']
```

Python 2.5の段階では、__builtins__に含まれている135種類のもののうち32種類までが、SyntaxErrorやZeroDivisionErrorのようにエラーを知らせるためのものです。また、Pythonの著作権者を返すcopyrightや、Pythonのかなり複雑なライセンスを表示するlicenseのような関数も含まれています。その他のメンバの中にも、後の章で取り上げるものが含まれています。

そして、それをインポートします(あるいは、Wing 101 の Run ボタンをクリックします)。

modules/import_experiment.cmd

```
>>> import experiment
パンダの学名は'Ailuropoda melanoleuca'です。
```

ここからわかるのは、**Python がモジュールをインポートするときに実行もしている**ということです。モジュールの中では他のプログラムで行うあらゆることをしてかまいません。Python からすれば、モジュールといっても実行される文の集まりにすぎません。

さらにまた別の実験をします。新しい Python セッションを開始し、experiment モジュールを 2 回続けてインポートしてみて下さい。

modules/import_twice.cmd

```
>>> import experiment
パンダの学名は'Ailuropoda melanoleuca'です。
>>> import experiment
>>>
```

2 度目にはメッセージが表示されなかったことに注目して下さい。これは、Python がモジュールをロードするのは、最初にインポートしたときだけだからです。Python は、すでに読んだことのあるモジュールを内部で管理しています。すでにリストに含まれているモジュールのロードを求められても、Python は要求を無視します。この動作は時間を節約するだけでなく、他のモジュールをインポートするモジュール(インポートしているモジュールがさらに他のモジュールをインポートすることもあり得ます)を書き始めたときに特に重要な意味を持ちます。すでにメモリにロードしてあるモジュールを管理していなければ、Python は math のようによく使われるモジュールを何十回もロードすることになってしまうでしょう。

4.2.2 __main__の使い方

今まで説明してきたように、すべての Python ファイルはコマンドラインや IDE から直接実行することもできますし、他のプログラムからインポートして使うこともできます。モジュールの中でどちらの形で実行されているかを知ることができれば便利な場合があります。つまり、モジュールがユーザーに実行を要求されたメインプログラムなのか、そのようなモジュールは他にあるのかということです。

Python は、どちらかを見分けやすくするために、すべてのモジュールで __name__ という特殊変数を定義しています。たとえば、echo.py に次のコードを入れたとします。

modules/echo.py

```
print "echo: __name__ は", __name__
```

このファイルを実行すると、この部分の出力は次のようになります。

modules/echo.out

```
echo: __name__ は __main__
```

Python は、約束通りに __name__ 変数を作っています。その値は "__main__" になっていますが、これは「このモジュールがメインプログラムだ」という意味です。

しかし、echo.py を直接実行せず、インポートしたときには、次のようになります。

modules/echo.cmd

```
>>> import echo
echo: __name__ は echo
```

echo モジュールをインポートする以外何もしないプログラムを書いても、同じことになります。

modules/import_echo.py

```
import echo
print "インポート後の __name__ は", __name__, "、echo.__name__ は", echo.__name__
```

このプログラムをコマンドラインから実行すると、次のように出力されます。

modules/import_echo.out

```
echo: __name__ は echo
インポート後の __name__ は __main__ 、echo.__name__ は echo
```

何が起きたのかを説明すると、モジュールをインポートしたときに、Python がモジュールの __name__ 変数に "__main__" という特殊変数ではなく、モジュールの名前をセットしているということです。これを利用すれば、モジュールは自分がメインプログラムかどうかを見分けられます。

modules/test_main.py

```
if __name__ == "__main__":
    print "私がメインプログラムである。"
else:
    print "誰かが私をインポートしている。"
```

このプログラムを直接実行したときとインポートしたときとで何が起きるかを確かめてみて下さい。

モジュールがインポートされているのかそうでないのかが見分けられると、うまいプログラミングテクニックが使えるようになります。1 つは、ライブラリというつもりで書かれたモジュールがコマンドラインから実行されたときにヘルプテキストを表示するというものです。たとえば、次のファイルをコマンドラインから実行したときと他のプログラムからインポートしたときにどうなるかを考えてみて下さい。

modules/main_help.py

```
'''
このモジュールは、何かが恐竜か否かを推測します。
'''

def is_dinosaur(name):
    '''
    引数が恐竜だと思われるならTrue、そうでなければFalseを返します。
    '''
    return name in ['Tyrannosaurus' , 'Triceratops']

if __name__ == '__main__' :
    help(__name__)
```

他の使い方は、後の章で説明していきます。

4.2.3 ヘルプの準備

temperature モジュールに戻り、温度の小数点以下を四捨五入するように書き換え、その結果を temp_round.py に格納します。

modules/temp_round.py

```
def to_celsius(t):
    return round((t - 32.0) * 5.0 / 9.0)

def above_freezing(t):
    return t > 0
```

to_celsius 関数のヘルプを要求すると、どうなるでしょうか。

modules/help_temp.cmd

```
>>> import temp_round
>>> help(temp_round)
Help on module temp_round:

NAME
    temp_round

FILE
    /home/pybook/modules/temp_round.py

FUNCTIONS
    above_freezing(t)

    to_celsius(t)
```


これではあまり役に立ちません。関数の名前、必要な引数の個数はわかりますが、それ以外のことはたいしてわかりません。もっと役に立つヘルプを提供するには、モジュールと関数に docstring と呼ばれるものを追加しなければなりません。追加した結果は、temp_with_doc.py に格納します。

modules/temp_with_doc.py

```
'''温度を操作する関数を集めてあります。'''

def to_celsius(t):
    '''華氏から摂氏に変換します。'''
    return round((t - 32.0) * 5.0 / 9.0)

def above_freezing(t):
    '''摂氏表現の温度が氷点以上ならTrue、そうでなければFalseを返します。'''
    return t > 0
```

このモジュールに対してヘルプを要求すると、ずっと役に立つ結果が得られます。

modules/help_temp_with_doc.cmd

```
>>> import temp_with_doc
>>> help(temp_with_doc)
Help on module temp_with_doc:

NAME
    temp_with_doc - 温度を操作する関数を集めてあります。

FILE
    c:\works\practical_programming\code\code\modules\temp_with_doc.py

FUNCTIONS
    above_freezing(t)
        摂氏表現の温度が氷点以上ならTrue、そうでなければFalseを返します。

    to_celsius(t)
        華氏から摂氏に変換します。
```

docstring は「documentation string」（ドキュメント文字列）の略です。docstring は簡単に作れます。ファイルや関数の最初の部分が代入されていない文字列なら、Python は help で表示できるようにその文字列を保存するのです。

こんなに小さな文字列なら、ドキュメントなど不要でしょうか。関数は 2 個しかありませんし、名前を見れば何をするのかだいたいわかります。しかし、ドキュメントは、ポイントを少し余分に稼ぐために書くようなものではありません。ソフトウェアを使えるものにするためには、ドキュメントが必要不可欠なのです。小さなプログラムでも、次第に大きなプログラムに成長して複雑になることがあります。ドキュメントを書かずに放っておいたり、プログラムと同じファイルで管理するようにしていなければ、何が何をしているのかは簡単にわからなくなってしまいます。

4.3 オブジェクトとメソッド

20世紀のプログラマは、数値と文字列だけを相手にしていても幸せでいられたかもしれませんが、今日のユーザーはイメージ、サウンド、ビデオなども操作したいと思うでしょう。mediaというPythonモジュールには、画像を操作、表示するための関数が含まれています。このモジュールは標準ライブラリには含まれていませんが、http://www.pragprog.com/titles/gwpy/source_code/からダウンロードできます（このモジュールを独立にダウンロードしなければならない理由については、練習問題の1つで考えていきます）。

mediaの仕組みを理解するためには、まず、現代のプログラム設計の基本概念を2つ学ばなければなりません。そして、そのためには、少し戻って文字列についてもう1度考える必要があります。

今までに私たちが見てきた文字列のための演算子は2つでした。2つの文字列を「加算する」連結(+)と、値の表示方法をコントロールする整形(%)です。しかし、文字列に対する操作としては、この他にも大文字化、先頭や末尾の空白の除去、部分文字列が含まれているかどうかのテストなど、何十種類もあります。これらすべてについて+とか-といった1文字の演算子を使うのは現実的ではありません。1文字をあっという間に使いきってしまうはずですし、2、3文字の組み合わせを使い始めると、今度は覚えられなくなります。

文字列を操作するすべての関数をモジュールにまとめて、ユーザーにそのモジュールをロードしてもらうというのも1つの方法ですが、この問題はもっと単純な方法で解決できます。Pythonの文字列は、**メソッド**（method）と呼ばれる特殊な関数の集合を「持って」いるのです。メソッドは、モジュール内の関数とまったく同じように呼び出せます。たとえば、'hogwarts'のような文字列があるとき、'hogwarts'.capitalize()を呼び出すと、先頭文字を大文字にした'Hogwarts'が返されます。同様に、villain変数に'malfoy'という文字列が代入されているときに、villain.capitalize()という式を実行すれば、'Malfoy'という文字列が返されます。

私たちが作るすべての文字列は、文字列型に属すすべてのメソッドを自動的に共有します。これらのメソッドの中でもっともよく使われるものを表4.1にまとめておきました。完全なリストは、Pythonのオンラインドキュメントを開くか、コマンドプロンプトにhelp(str)と入力すれば見られます。

メソッドは関数と同じように使えますが、ほとんどの場合、オーナーとなっているものに何か操作を加えたり、オーナーを使って何かをしたりします。たとえば、'species'という文字列を使ってstartswithメソッドを呼び出してみましょう。

modules/startswith.cmd

```
>>> 'species'.startswith('a')
False
>>> 'species'.startswith('s')
True
```

startswithメソッドは、文字列引数を取り、メソッドのオーナー文字列（ドットの左側の文字列）の先頭が引数の文字列になっているかどうかを示すboolを返します。文字列には、endswithメソッドもあります。

表4.1 よく使われる文字列メソッド

メソッド	説明
capitalize()	先頭文字を大文字にしたオーナーのコピーを返します。
find(s)	文字列の中でsが最初に現れる場所の添字を返します。sが文字列に含まれていない場合は、-1を返します。
find(s, beg)	文字列の添字beg以降の部分でsが最初に現れる場所の添字を返します。sがbeg以降の部分に含まれていない場合は、-1を返します。
find(s, beg, end)	文字列のbeg以降end以前の部分でsが最初に現れる場所の添字を返します。sがbeg以降end以前の部分に含まれていない場合は、-1を返します。
islower()	すべての文字が小文字になっているかどうかをテストします。
isupper()	すべての文字が大文字になっているかどうかをテストします。
lower()	すべての文字を小文字に変換したオーナーのコピーを返します。
replace(old, new)	すべての部分文字列oldをnewに置換したオーナーのコピーを返します。
split()	文字列をスペースで分割し、リストにして返します。
split(del)	文字列をdelで分割し、リストにして返します。
strip()	先頭と末尾の空白を取り除いたオーナーのコピーを返します。
strip(s)	sに含まれる文字を取り除いたオーナーのコピーを返します。
upper()	すべての文字を大文字に変換したオーナーのコピーを返します。

modules/endswith.cmd

```
>>> 'species'.endswith('a')
False
>>> 'species'.endswith('s')
True
```

他のメソッド呼び出しの戻り値からメソッドを呼び出せば、1行の中に複数のメソッド呼び出しを連鎖的に並べることができます。具体例を見ていきましょう。まず、小文字を大文字、大文字を小文字に置き換えるswapcaseを呼び出します。

modules/swap.cmd

```
>>> 'Computer Science'.swapcase()
'cOMPUTER sCIENCE'
```

このメソッドの結果は文字列ですから、結果のendwithメソッドを呼び出せば、最初の呼び出しが元の文字列の末尾数文字に対して正しい処理を行ったかどうかをチェックすることができます。

modules/swap_endswith.cmd

```
>>> 'Computer Science'.swapcase().endswith('ENCE')
True
```

図4.3は、連鎖呼び出しで実際にどのようなことが行われるかを示しています。Pythonは、swapcaseメソッド呼び出しの結果を保持できる一時変数を自動的に作り、その値のendswithメソッドを呼び出します。

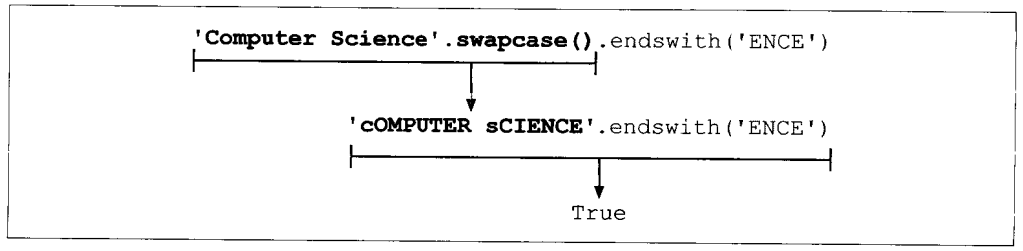

図4.3 メソッドの連鎖呼び出し

メソッドを持つもののことを**オブジェクト**(object)と言います。実は、0という数値を含め、Pythonの中の**すべてのもの**がオブジェクトです。

modules/int_help.cmd

```
>>> help(0)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible. A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base. It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |  (可能なら、文字列や数値を整数に変換します。浮動小数点数引数は、
 |  0に近い値に切り捨てられます(浮動小数点数を表している文字列は含まれません)。
 |  文字列を変換するときには、オプションのbase(基数)を使います。
 |  文字列以外のものを変換するときにbaseを指定するとエラーになります。
 |  引数が整数の範囲を越えている場合は、intの代わりにlongが返されます。)
 |
 |  Methods defined here:
 |  (このオブジェクトが持つメソッド)
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
...
```

現代のプログラミング言語の大半は、このような作りになっています。プログラムの中の「もの」はオブジェクトになっており、プログラム内のコードの大半は、それらのオブジェクトに格納されたデータを使うメソッドから構成されています。「13章 オブジェクト指向プログラミング」では、新しい種類のオブジェクトの作り方を説明します。ここでは、Pythonがイメージを格納、操作するために使っているオブジェクトについて見てみましょう。

4.3.1 イメージ

モジュール、オブジェクト、メソッドの基本的な働きは一通り説明しましたので、これらが実世界の問題をどのようにして解決するのかを見ていくことにしましょう。生きた例として、写真などのイメージを表示、操作するプログラムを書きます。

ハードディスクにpic207.jpgというファイルがあり、画面にそれを表示したいものとします。ファイルをダブルクリックすればオープンしますが、実際にはこのとき何が行われているのでしょうか。この問いに答えるための手始めとして、Pythonプロンプトに次のように入力して下さい。

modules/open_pic.cmd

```
>>> import media
>>> f = media.choose_file()
>>> pic = media.load_picture(f)
>>> media.show(pic)
```

ファイルダイアログがオープンしたら、ダイアログを操作してpic207.jpgをロードします。すると、図4.4のようにおそろしくかわいい写真が表示されるはずです。上のコマンドが実際に行ったことをまとめると、次のようになります。

1. mediaモジュールから関数をインポートします。
2. mediaモジュールのchoose_file関数を呼び出して、ファイル選択ダイアログをオープンします。この呼び出しは、写真ファイルのパスの文字列を返します。
3. mediaモジュールのload_picture関数を呼び出して、写真ファイルの内容をメモリに読み出します。こうすると、Pythonオブジェクトが作成されますので、作成されたオブジェクトをpic変数に代入します。
4. mediaモジュールのshow関数を呼び出します。すると、写真を表示するための別のプログラムが起動されます。他のプログラムを起動しなければならないのは、Pythonがコマンドラインに画像を出力できないからです。

ダブルクリックの方が間違いなく簡単でしょう。
しかし、マウスは次のことをしているのです。

modules/pic_props.cmd

```
>>> pic.get_width()
500
>>> pic.get_height()
375
>>> pic.title
'modules/pic207.jpg'
```

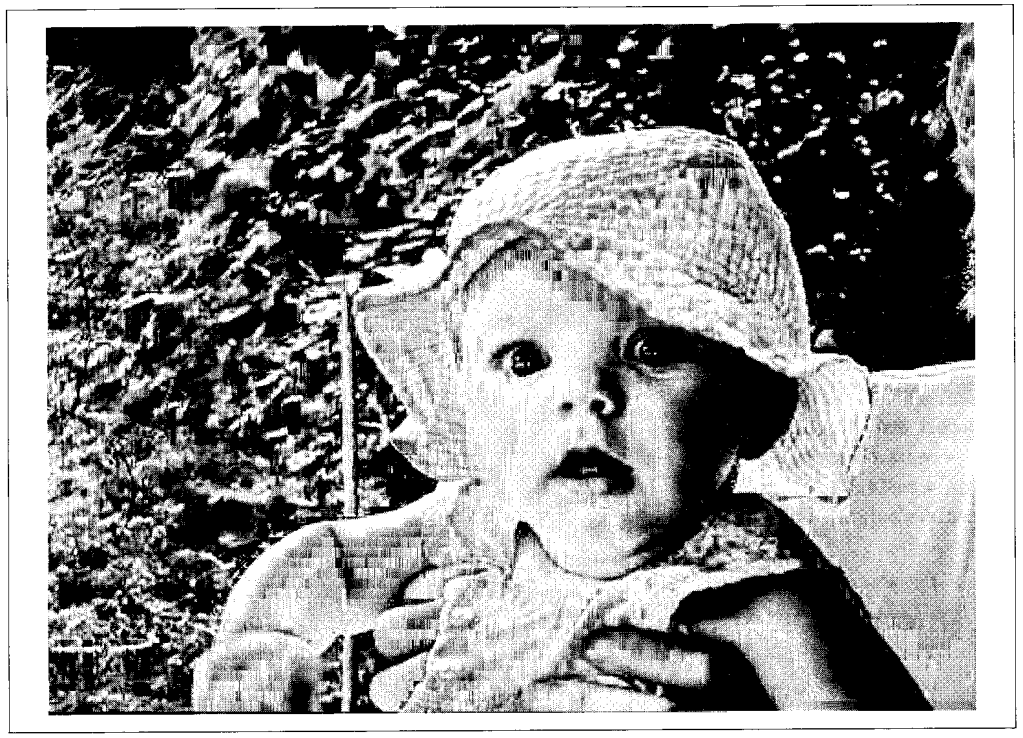

図4.4 マドレーヌ

最初の2つのコマンドは、画像の縦横の長さをピクセル単位で知らせてきます。第3のコマンドは、画像ファイルのパスを教えてくれます。

では、次のコマンドを試してみましょう。

modules/pic_crop.cmd

```
>>> media.crop_picture(pic, 150, 50, 450, 300)
>>> media.show(pic)
>>> media.save_as(pic, 'pic207cropped.jpg')
```

名前から想像されるように、crop_pictureは画像を切り取り(crop)ます。(150, 50)を左上隅、(450, 300)を右下隅にして切り取ると、画像は図4.5のようになります。

このコードは新しい画像を表示して、それを新しいファイルに書き込んでもいます。このファイルは、カレントディレクトリに保存されます。カレントディレクトリのデフォルトは、プログラムが実行されているディレクトリです。私たちのシステムでは、'/Users/pgries/' になっています。

では、マドレーヌの帽子に名前を書き込んでみましょう。mediaのadd_text関数を使います。実行結果は、図4.6のようになります。

図4.5 不要な部分を切り取ったマドレーヌ

modules/pic_text.cmd

```
>>> media.add_text(pic, 115, 40, 'Madeleine' , media.magenta)
>>> media.show(pic)
```

choose_file関数は対話的なプログラムを書くときには便利ですが、どのファイルをオープンすべきかがはっきりしている場合や複数のファイルを必要とする場合は、ダイアログ操作を省略できた方が簡単です。たとえば、1つのプログラムでマドレーヌの3枚の写真をすべてオープンしてみましょう。

modules/show_madeleine.py

```
import media

pic1 = media.load_picture('pic207.jpg')
media.show(pic1)
pic2 = media.load_picture('pic207cropped.jpg')
media.show(pic2)
pic3 = media.load_picture('pic207named.jpg')
media.show(pic3)
```

図4.6 名前を追加したマドレーヌ

ファイルをどのディレクトリから探すべきかを指定していませんので、プログラムはカレントディレクトリでファイルを探します。ファイルが見つからなければ、エラーメッセージが表示されます。

4.4 ピクセルと色

ほとんどの人は、単に表示したり、一部を切り取ったりするだけでなく、写真にさまざまな操作をしたいと思うはずです。たとえば、フラッシュによる「赤目」効果を取り除くことはできないでしょうか。また、印刷用に写真を白黒に変換したり、写っている特定のものを強調したりといったこともしたいところです。

これらの処理をするためには、イメージを構成する1つ1つの**ピクセル**（pixel）を操作しなければなりません。mediaモジュールは、**RGBカラーモデル**（RGB color model。後のコラム「RGBと16進数」参照）を使ってピクセルを表現します。mediaモジュールは、Color型と100種類以上の定義済みColor値を持っています。表4.2は、その一部を示したものです。黒は青、緑、赤のすべてが0の色として表現され、白は逆に3色すべてが最大値の色として表現されます。他の色は、この中間のどこかに入ります。

表4.2 カラー値の例

色	値
黒	Color(0, 0, 0)
白	Color(255, 255, 255)
赤	Color(255, 0, 0)
緑	Color(0, 255, 0)
青	Color(0, 0, 255)
マゼンタ	Color(255, 0, 255)
黄色	Color(255, 255, 0)
水色	Color(0, 255, 255)
ピンク	Color(255, 192, 203)
紫	Color(128, 0, 128)

mediaモジュールは、ピクセルの色を取得、設定する関数（表4.3参照）と色自体の操作関数（表4.4参照）を用意しています。

表4.3 ピクセル操作関数

関数	説明
get_red(pixel)	ピクセルの赤の要素を取得します。
set_red(pixel, value)	ピクセルの赤の要素をvalueにします。
get_blue(pixel)	ピクセルの青の要素を取得します。
set_blue(pixel, value)	ピクセルの青の要素をvalueにします。
get_green(pixel)	ピクセルの緑の要素を取得します。
set_green(pixel, value)	ピクセルの緑の要素をvalueにします。
get_color(pixel)	ピクセルの色を取得します。
set_color(pixel, color)	ピクセルの色を設定します。

表4.4 色操作関数

関数	説明
darken(color)	colorよりもわずかに暗い色を返します。
lighten(color)	colorよりもわずかに明るい色を返します。
create_color(red, green, blue)	(red, green, blue)という値のColorを返します。
distance(c1, c2)	c1とc2がどの程度離れているかを返します。

これらの関数の使い方を具体的に見ていきましょう。マドレーヌの写真に含まれているすべてのピクセルを操作して、夕暮れに撮ったような効果を生み出します。そのためには、各ピクセルから赤と緑の要素を少し取り除き、写真全体を暗く、そして赤くします†。

modules/sunset.py

```
import media
pic = media.load_picture('pic207.jpg')
media.show(pic)
for p in media.get_pixels(pic):
```

† 実際には赤の要素を加えませんが、青と緑の要素が減ると、目は錯覚によって赤くなったように感じます。

```
    new_blue = int(0.7 * media.get_blue(p))
    new_green = int(0.7 * media.get_green(p))
    media.set_blue(p, new_blue)
    media.set_green(p, new_green)

media.show(pic)
```

注意すべきポイントについてまとめておきます。

- カラー値は整数なので、青と緑に0.7を掛けた結果にint関数を適用して整数に変換しなければなりません。
- 画像の個々のピクセルに操作を加えているのは、forループです。ループについては「5.4 リストの要素の操作」で詳しく説明しますが、コードを読んでみれば、各ピクセルを順にp変数にセットし、青と緑の要素を取り出して新しい値を計算し、ピクセルに設定し直しているということが理解できるでしょう。

このコードであなたの写真を操作して、それらしい効果がどれくらい得られるかを確かめてみて下さい。

RGBと16進数

RGB(red-green-blue)カラーモデルでは、画像の各ピクセルが一定の度合いで3原色を持ち、各色の度合いは0から255までの範囲の整数(これは1個の8ビットバイトで表現できる数値の範囲です)で指定されます。

RGB値は、伝統的に10進数ではなく16進数、すなわち基数を16とする表記法で表されます。16進数の「桁」は通常の0から9までにAからF(またはaからf)までの文字を加えたものになります。そのため9_{16}の次の値は10_{16}ではなく、A_{16}になります。A_{16}の次はB_{16}、C_{16}と、F_{16}まで続き、F_{16}の次は10_{16}になります。10_{16}の後は$1F_{16}$まで続き、さらにその次が20_{16}になり、2桁の数はFF_{16}まで続きます。FF_{16}は、$15_{10}\times16_{10}+15_{10}$、すなわち$255_{10}$です。

そのため、RGBカラーは、6桁の16進数になります。最初の2桁で赤、次の2桁で緑、最後の2桁で青を表します。そのため、黒は#000000(どの色もなし)、白は#FFFFFF(すべての色が最高値)、青緑は#008080(赤なし、緑半分、青半分)になります。

4.5 テスト

現実のPythonプログラミングでは、プログラムがただ動くだけではなく、正しい答えを出していることを確かめるためにも、モジュールを使います。たとえば科学分野では、実験データの分析に使うプログラムに、少なくともそのデータを集めるために使った実験器具と同じくらいの信頼性がなければ、実験をする意味がなくなってしまいます。CATスキャナなどの医療機器を動かすプログラムは、患者の生命がかかっていますから、さらに高い信頼性を要求されます。また、プログラムが正しく動作することを確認するツールは、教師が学生の課題を評価するために使いますし、学生が提出するプログラムをチェックするためにも使います。

ソフトウェアの動作の正しさをチェックすることを**品質保証**、あるいはQA(quality assurance)と言います。プログラマたちは、50年の時間をかけて、品質とは、プログラムを書いた後にばらまけば効果の出る魔法の薬ではないことを学びました。品質保証は最初から設計に組み込まれていなければなりませんし、ソフトウェアはテストにテストを重ねて基準に達していることを確かめる必要があります。

それでも、QAに力を入れれば、生産性全体が高くなるという朗報もあります。その理由は、図4.7に示したベーム推移曲線にあります。バグを見つけたタイミングが遅ければ遅いほど、フィックスにかかるコストは高くなりますので、早い段階でバグをキャッチすれば全体の作業量を下げられるのです。

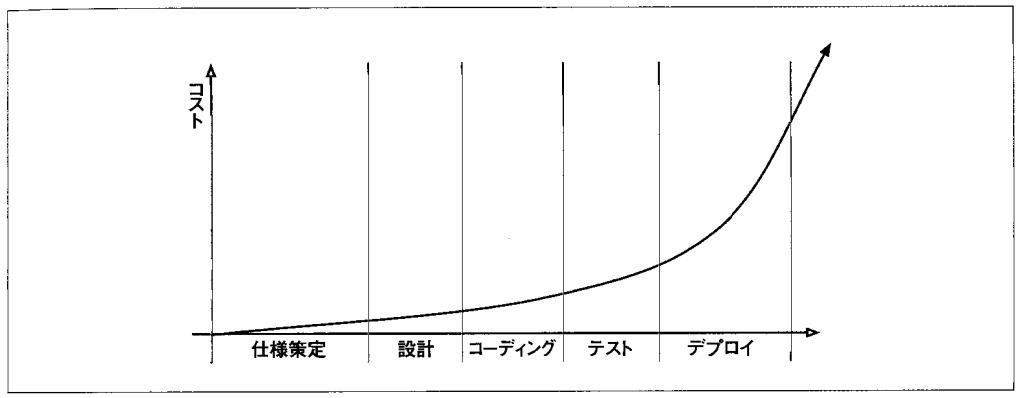

図4.7 ベームの推移曲線

今日の優れたプログラマは、コードを書く過程で単にテストを行うだけでなく、数か月後にまったく離れた場所にいる他人がもう1度実行できるようなテストを作ります。こうすると、最初にかかる時間は少し増えますが、バグを防ぐために1時間を費やせば、バグを探す2、3時間あるいは10時間がなくなりますから、プログラマ全体での生産性は上がります。

Python用のテストライブラリで人気のあるものにNoseがあります。Noseは、http://code.google.com/p/python-nose/ から無料でダウンロードできます。ここでは、temperatureモジュールを使ってNoseによるテストの方法を説明していきます。まず、test_temperature.pyという新しいPythonファイルを作ります。この名前は重要です。Noseは、起動されると先頭がtest_になっているファイルを自動的に探すのです。名前の第2の部分は自由に決められます。たとえば、test_hagrid.pyにしてもかまいません。しかし、わかりやすい名前を付ければ、他のプログラマが私たちのコードをテストしやすくなります。

すべてのNoseテストモジュールは、次のものを含んでいなければなりません。

- Nose とテスト対象のモジュールをインポートする文
- 実際にモジュールをテストする関数
- テスト関数の実行を開始させる関数呼び出し

テストモジュールの名前と同様に、テスト関数の名前も、test_ で始まる名前にします。以上の説明に従って、テストモジュールの第 1 歩に当たるものを作ると、次のようになります。

modules/structure.py

```
import nose
import temperature

def test_to_celsius():
    '''to_celsiusのテスト関数'''
    pass # 後で実装

def test_above_freezing():
    '''above_freezingのテスト関数'''
    pass # 後で実装

if __name__ == '__main__' :
    nose.runmodule()
```

まだ、テスト関数には docstring と pass 文しかありません。名前から想像できるように、pass は何もしません。後で何かコードを書かなければならないことを思い出すためのプレースホルダです。

このテストモジュールを実行すると、出力の先頭は 2 個のドットになります。これは 2 個のテストを実行して合格したという意味です（テストに合格しなかったときには、問題点に注意を引き付けるために、ドットではなく「F」を表示します）。破線の後ろのサマリを見ると、Nose が 2 個のテストを見つけて実行し、そのために 1,000 分の 1 秒もかかっていないこと、そしてテスト結果はすべて OK だったことがわかります。

modules/structure.out

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

私たちのテスト関数はまだ何もテストしていませんから、2 個のテストに合格したとしても驚くべきことではありません。次のステップは、実際にテストコードを書き、何か役に立つ処理をさせることです。テストの目標は、コードが正しく動くのを確かめることです。to_celsius の場合、正しい動作とは、華氏の温度を与えると、対応する摂氏の値が正しく返されることです。

あらゆる値を試してみることは明らかに現実的ではありません。現実の数値は無数にあります。そこで、いくつかの代表的な値を選び、関数がそれらの値に対して正しい処理をしていることを確かめます。

たとえば、「4.2.3　ヘルプの準備」で作った to_celsius の四捨五入バージョンが、華氏 32 度（摂氏 0 度）と華氏 212 度（摂氏 100 度）の 2 つの値に対して正しい結果を返すことを確かめてみましょう。また、念には念を入れて、きれいに変換されない値もテストしておきましょう。たとえば、華氏 100 度は、摂氏 37.777... 度になりますので、私たちの関数は 38 を返さなければなりません（小数点以下を四捨五入するため）。

個々のテストは、関数が**実際に返してきた値**と**返してくるはずの値**を比較して行います。この場合は、to_celsius(100) が 38 にならなければならないことを Nose に知らせるために assert 文を使います。

modules/assert.py

```
import nose
from temp_with_doc import to_celsius

def test_freezing():
    '''氷点のテスト'''
    assert to_celsius(32) == 0

def test_boiling():
    '''沸点のテスト'''
    assert to_celsius(212) == 100

def test_roundoff():
    '''四捨五入のテスト'''
    assert to_celsius(100) == 38 # 37.777...ではなく

if __name__ == '__main__' :
    nose.runmodule()
```

コードを実行すると、テストは 3 つの結果のどれかになります。

- **合格**：実際の値が期待通りの値になっています。
- **不合格**：実際の値が期待した値になっていません。
- **エラー**：テスト自体の中で問題が起きています。言い換えれば、テストコードにバグが含まれているということです。この場合、テストはテスト対象のシステムについて何も教えてくれません。

テストモジュールを実行してみましょう。出力は、次のようになるはずです。

modules/outcome.out

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

先ほどと同様に、ドットはテストに合格したことを示しています。Nose が正しい仕事をしていることを証明するために、to_celsius の結果を 37.8 と比較してみましょう。

modules/assert2.py

```
import nose
from temp_with_doc import to_celsius

def test_to_celsius():
    '''to_celsiusのテスト関数'''
    assert to_celsius(100) == 37.8
if __name__ == '__main__' :
    nose.runmodule()
```

こうすると、テストは不合格になり、テストに対応するドットは「F」に置き換わり、エラーメッセージが表示され、OK の代わりにエラーの数が表示されます。

modules/fail.out

```
F
======================================================================
FAIL: to_celsiusのテスト関数
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/python25/lib/site-packages/nose/case.py" , line 202, in runTest
    self.test(*self.arg)
  File "assert2.py" , line 6, in test_to_celsius
    assert to_celsius(100) == 37.8
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

エラーメッセージを見ると、不合格になったのは test_to_celsius の 6 行目だということがわかります。これだけでも役に立ちますが、個々の assert 文に何をテストしているかについての説明を付けておくと、不合格の理由がさらにはっきりするはずです。

modules/assert3.py

```
import nose
from temp_with_doc import to_celsius

def test_to_celsius():
    '''to_celsiusのテスト関数'''
    assert to_celsius(100) == 37.8, '四捨五入されていない結果を返している'

if __name__ == '__main__' :
    nose.runmodule()
```

すると、このメッセージが出力に組み込まれます。

modules/fail_comment.out

```
F
======================================================================
FAIL: to_celsiusのテスト関数
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Python25\Lib\site-packages\nose\case.py" , line 202, in runTest
    self.test(*self.arg)
  File "assert3.py" , line 6, in test_to_celsius
    assert to_celsius(100) == 37.8, '四捨五入されていない結果を返している'
AssertionError: 四捨五入されていない結果を返している

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

1 つの値を使って test_to_celsius をテストしましたが、他のテストケースが必要かどうかを決めなければなりません。テストケースの説明で正の値ということを言っているので、0 や負数でもテストした方がよいようにも感じます。しかし、本当に考えなければならないのは、コードのふるまいが 0 や負数になると変わるかどうかです。ここでは単純な算術計算をしているだけですから、わざわざ 0 や負数まで試してみる必要はないでしょう。後の章では、複雑な内容を持つため複数のテストを必要とするような関数も取り上げていきます。

では、test_above_freezing に移りましょう。これがテストしようとしている above_freezing 関数は、氷点よりも高い温度に対して True を返すことになっていますので、89.4 について正しくふるまうかどうかをテストします。また、氷点下の温度に対して正しくふるまうかどうかもテストすべきですから、-42 を対象とするチェックも追加します。

最後に、温度が氷点そのものであるという条件の境目での動作もテストすべきです。この種の値は、関数のふるまいが変わる境目の位置にあるため、**境界条件** (boundary case) と呼ばれます。他のものと比べて境界条件ではバグが含まれている確率が非常に高くなることが経験上わかっていま

すので、境界条件がどこにあるかを突き止め、それをテストすることには大きな意味があります。

以上に従えば、テストモジュール(コメント付き)は、次のようになります。

modules/test_freezing.py

```
import nose
from temp_with_doc import above_freezing

def test_above_freezing():
    '''above_freezingのテスト関数'''
    assert above_freezing(89.4), '氷点よりも高い温度'
    assert not above_freezing(-42), '氷点よりも低い温度'
    assert not above_freezing(0), 'ちょうど氷点'

if __name__ == '__main__' :
    nose.runmodule()
```

このテストを実行すると、出力は次のようになります。

modules/test_freezing.out

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

おやおや、ファイル内には3つのassert文が含まれているのに、Noseはテストが1つだと言っています。これは、Noseにとっては、関数が1つでテストが1個だからです。それらの関数の中に複数のことをチェックするものが含まれていても、それは関数の問題ということになりす。しかし、アサーションが1つ失敗すると、そのassertが含まれている関数の実行が停止してしまうのは問題です。test_above_freezingの最初のチェックが失敗すると、その後のテストについての情報は得られなくなってしまいます。一般に、1つの関数に何十個ものアサーションを入れるのではなく、ごく少数のポイントをチェックするだけの小さなテスト関数を多数書くようにすべきです。

4.6 コーディングスタイルについて

Pythonプログラムに入れられるものなら何でもモジュールに入れることができますが、だからといって何でもモジュールに入れるべきだというわけではありません。論理的に同じモジュールに属すべき関数と変数は、同じモジュールに入れるべきです。しかし、車の種類によって排出される一酸化炭素がどのくらいになるのかを計算する関数と直径や骨密度から骨の強さを計算する関数のように論理的なつながりのない関数は、たまたまあなたが両方を書いたのだしても、1つのモジュールにまとめるべきではありません。

もちろん、論理的な関係があるものが何で、ないものが何かについての意見は人によって異なることがよくあります。たとえば、Pythonのmathモジュールについて考えてみましょう。行列の乗算を行う関数はこのモジュールに入れるべきでしょうか、それとも別個の線形代数モジュールを作るべきでしょうか。基本統計関数はどうでしょうか。以前の節で燃費を計算する関数を取り上げましたが、これと一酸化炭素排出量を計算する関数は同じモジュールにまとめるべきでしょうか。2つの関数を同じモジュールに入れるべきでないという理由は、探せばかならず見つかりますが、それぞれ関数を1つずつ収めた1,000個のモジュールを作っても、人々(作者自身を含めて)はどうすればよいか戸惑ってしまうでしょう。

一応の目安として、モジュールに含まれるものが半ダース以下なら、そのモジュールはたぶん小さすぎますが、1、2文のdocstringでモジュールの内容や目的を要約できないなら、そのモジュールはおそらく大きすぎます。しかし、これらは目安にすぎません。最終的には、ベテランのプログラマたちがPython標準ライブラリに含まれているようなモジュールをどのように構成しているのかを学んで決めるべきことです。それを繰り返しているうちに、あなた独自のスタイルが確立していくでしょう。

4.7 まとめ

この章では、次のことを学びました。

- モジュールとは、ファイルに1つにまとめられた関数や変数のコレクションのことです。モジュールを使うには、最初にインポートしなければなりません。インポートしたモジュールの内容には、<モジュール名>.<変数/関数名>でアクセスできます。
- モジュールや関数の先頭には、内容や用途を説明するdocstringを置きます。
- Pythonのすべての「もの」は、オブジェクトです。オブジェクトは、関数とまったく同じようにふるまうメソッドを持ちます。メソッドは、オーナーとしてのオブジェクト型に対応付けられています。
- mediaモジュールを使えば、イメージを操作できます。mediaモジュールには、イメージ全体をロード、表示、操作する関数や個々のピクセル、色を取得、設定する関数が含まれています。
- ただ動くだけでは、使えるプログラムとは言えません。プログラムは、正しく動作しなければならないのです。正しい動作を保証するための手段の1つとして、テストがあります。Pythonではnoseモジュールを使えばテストを実行できます。普通はすべての条件をテストしつくすことはできませんから、境界条件を重点的にテストするようにします。

4.8 練習問題

練習問題で自分の力を試してみましょう。

1. mathモジュールをインポートし、その関数を使って次の課題をして下さい。

a) -4.3を四捨五入し、その結果の絶対値を返す1個の式を書いて下さい。
b) 34.5の正弦の天井値を返す式を書いて下さい。

2. Pythonのcalendarモジュールを使って各問いに答えて下さい。
   a) PythonドキュメントのWebサイトであるhttp://docs.python.org/modindex.htmlにジャンプして、calendarモジュールのドキュメントを読んで下さい。
   b) calendarモジュールをインポートして下さい。
   c) isleap関数の説明を読み、isleapを使って次の閏年を調べて下さい。
   d) 2000年から2050年までの間に閏年が何回あるかを調べられる関数をcalendarモジュールから探し、回数を答えて下さい。
   e) 2016年7月29日が何曜日になるかを調べられる関数をcalendarモジュールから探し、曜日を答えて下さい。

3. 文字列メソッドを使って、次のことをする式を書いて下さい。
   a) 'boolean'を大文字にして下さい。
   b) 'CO2 H2O'で最初に'2'が現れる位置を調べて下さい。
   c) 'CO2 H2O'で2度目に'2'が現れる位置を調べて下さい。
   d) 'Boolean'の先頭文字が小文字かどうかを調べて下さい。
   e) "MoNDaY"をすべて小文字に変換してから、先頭文字だけを大文字にして下さい。
   f) " Monday"の先頭の空白を取り除いて下さい。

4. import *の説明に使った例は、次のものでした。

modules/from2.cmd
```
>>> from math import *
>>> '%6f'  % sqrt(8)
'2.828427'
```

   2.828427という値の前後にクォートが付いているのはなぜか説明して下さい。

5. 「4.3.1 イメージ」で取り上げたmediaモジュールがPythonの標準ライブラリの一部になっていないのはなぜだと思いますか。Pythonの開発者たちは、何を標準ライブラリに入れるか否かをどのようにして決めたのだと思いますか。標準ライブラリに含まれていないモジュールが必要なとき、どこでどのようにすれば見つけられますか。

6. ユーザーがファイルを選択でき、選択された画像を2回表示するプログラムを書いて下さい。

7. ユーザーがファイルを選択でき、選択された画像の各ピクセルの赤要素を0にしてから画像を表示するプログラムを書いて下さい。

8. ユーザーがファイルを選択でき、選択された画像の各ピクセルの緑要素を半分にしてから画像を表示するプログラムを書いて下さい。

9. ユーザーがファイルを選択でき、選択された画像をモノクロに変えるプログラムを書いて下さい。このプログラムは、各ピクセルの赤、緑、青の値の平均を計算し、赤、緑、青としてその平均値を設定しなければなりません。

10. ユーザーがファイルを選択でき、選択された画像の各ピクセルの赤要素を倍にして画像を表示するプログラムを書いて下さい。倍にした値が255よりも大きくなったらどうなるでしょうか。

11. 新聞やテレビでは、写真の色を塗り直したり、2人の人の写真を結合して彼らを同じ写真に写らせたりといった写真の「拡張」を行うことがあります。メディアはそのような変更を加えていないイメージだけを使わせるようにすべきだと思いますか。現代の写真や動画はほぼすべてデジタルであり、表示のために何らかの処理が必要だということを前提とした場合、この規則は実際にどのような意味になると思いますか。

12. 2つのXY座標の距離を計算する関数をテストしたいものとします。

modules/distance.py
```
import math
def distance(x0, y0, x1, y1):
    '''(x0, y0)と(x1, y1)の距離を計算します。'''

    return math.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)
```

   a) to_celsiusの四捨五入バージョンとは異なり、この関数は浮動小数点数を返します。その分テストが難しくなっているのですが、その理由を説明して下さい。
   b) あなたの友だちが、次のようにして関数をテストすることを提案しています。

modules/test_distance.py
```
import nose
from distance import distance

def close(left, right):
    '''2つの浮動小数点数が十分に近いかどうかをテストします。'''

    return abs(left - right) < 1.0e-6

def test_distance():
    '''distance関数が正しく動作しているかどうかをテストします。'''

    assert close(distance(1.0, 0.0, 1.0, 0.0), 0.0), '同一の点で失敗'
    assert close(distance(0.0, 0.0, 1.0, 0.0), 1.0), '単位距離で失敗'

if __name__ == '__main__':
    nose.runmodule()
```