September 2010

3-2

TSNETスクリプト通信

TSC編集委員会

目次

巻頭言	jscripter	...	2
スクリプトで数学を可視化する	海鳥	...	4
Pythonの文法 草稿　第3回	機械伯爵	...	9
よしおさんとロボ太	海鳥	...	39
マルチバイト文字はlength()==2の夢を見るか？	Ｙさ	...	43
編集後記	jscripter	...	48

表紙デザイン ： jscripter

海鳥さんのコムギ粘土製「3D ロボ太」の貴重な(既に黴びてしまったそうで^^;)写真を単純に引き伸ばさせていただきました。シュールな感じになっていれば最高なんですが。

巻頭言

jscripter

TSNETスクリプト通信は本号で、第3巻第2号、記念すべき通算第10号になります。今号は猛暑の夏の中ですが、海鳥さんから投稿をいただいたことをきっかけになんとか刊行に辿りつけました。

海鳥さんの記事は、「スクリプトで数学を可視化する ～ RubyとGraphvizでCollatz予想 ～」というタイトルで、「TSNETで数学でも勉強しよう」というキャッチフレーズが生まれそうです。また「よしおさんとロボ太」シリーズ連載は、ロボ太誕生の秘密が明かされる4連作を大盤振る舞いしました。合わせて、表紙には新作の3Dロボ太を使わせていただきました。

機械さんの「Pythonの文法」は第3回、毎度ご苦労様です。第3回は「6. オブジェクト」です。いよいよ、Pythonのオブジェクトに迫ります。

Ｙささんからは、私が刊行延長を申し入れたことが幸いし、なんとか記事をまとめていただいて投稿していただきました。ありがたし^^)/　元ネタはFGALTS時代の『「貴方もawkでプログラミングを」講座』にあるものだそうで懐かしい。

私自身は、8月末刊行を目指して、twitter関連のスクリプトを弄っていたのですが、なかなか話としてまとまらないので一週間刊行を延ばさせていただきました。しかし、8月末を境にして認証に関連するスクリプトが動かなくなって、今日(9/5)ようやくAPI terms of Serviceに「August 31, 2010 Basic Auth has been deprecated.」の記述を発見。Net::Twitterにも、そのためにNet::Twitter::Role::OAuthモジュールがあることに気がついたという体たらく。8月末に刊行していたら、とんでもなくいい加減なものになっていたところで、無理は止めようということに致しました。もう少し自在に使えるようになってからということにしたいと思います。

WebアプリケーションのAPIのスクリプトへの使用については、AmazonのAPIなどを含めて趣味に使う場合には多少神経質になってしまいます。TwitterのAPI terms of Serviceは、zed2twitter.plを書いた昨年は気づかなかったので存在していなかったような気もします。新しいOAuthの認証ではアプリケーション毎に登録する必要があり、大した制約ではありませんが、守るべきルールを満たす必要があります。そういう事情も加わって、今回は私自身の記事はパスということにしました。現在では、zed2twitter.plはそのままでは動かなくなっています。

WebアプリケーションのAPIについて趣味的、非営利的利用についての取り扱いの記載があるとうれしいという感じはいたしますね。日曜プログラマといたしましては。

編集後記で、Twitterの技術リソースと「国立国会図書館サーチ」のリソースについて少しまとめています。参考にしてください。

Have much fun.

(2010 年 9 月 5 日投稿受付)

スクリプトで数学を可視化する

～ Ruby と Graphviz で Collatz 予想 ～

海鳥

0. 概要

Graphviz ライブラリを使って、数論に関する未解決問題の一つ「Collatz 予想」を可視化する Ruby スクリプトを紹介する。

1. はじめに

数学の一分野、数論は主に整数の性質について研究する学問である。一般に、整数より分数などの有理数が、さらに有理数より実数の方が難しいとされており、実際に学校でもその順番で習う。また、数論の問題といえばフェルマーの最終定理に代表されるように、素人にも問題の理解が容易であることが多い。しかし、実際に数学の問題を考える時、解を整数や有理数に限るとものすごく難しくなる。実数には「連続性」という極めて強力な性質があるが整数にはそれがなく、従って、整数の研究には、微分や積分といった武器が使えないからである。この「理解は簡単、解くのは難しい」という整数論の世界は、学者のみならず、数多くのアマチュア数学者の興味をひいてきた。ちなみにフェルマーもそんなアマチュア数学者の一人であった。

さて、フェルマーの最終定理は解決されたが、整数論には素数の分布(リーマン予想)や双子素数の問題など、未解決問題が数多く残されている。本稿では、そのうちの一つ、Collatz(コラッツ)予想を取り上げる。なんでも良いから、整数を一つ思い浮かべてみよう。それが偶数であれば 2 で割り、奇数であれば 3 倍して 1 を足す。この操作を数が 1 になるまで繰り返す。例えば「10」からスタートすると、偶数だから 2 で割って「5」、5 は奇数だから 3 倍して 1 を足すと 16、以下 8,4,2,1 と 5 ステップで 1 になる。「どんな数でも、この操作を繰り返せばいつかは 1 になる」というのが Collatz の予想である。これは角谷の問題とも呼ばれ、1937 年に提出されて以来、プロアマを問わず世界中の数学者の挑戦を受けてきたが、未だ解決していない。

この問題を解くのはもちろん無理であるが、この問題を可視化すること考えてみよう。先ほどの例では、10 をスタートとすると 10→5→16→8→4→2→1 という列を作る。また、32 からスタートすれば 32→16→8→4→2→1 という数列となり、10 からはじめた数列に合流する。これをグラフとして考えることにしよう。すなわち、1 が根(root)となり、次々に数字が分かれて伸びて行く二分木だとみなすことにしよう。Collatz 予想とは「このグラフがループを作らない」もしくは「すべての整数が 1 につながっている」(すなわちグラフが「木」になっている)と言い換えることができる。このグラフを作るのが本稿の目的である。

2. Graphviz のインストールと使い方

グラフの可視化に Graphviz を使うことにする。Graphviz は AT&T 研究所が開発したオープンソースソフトウェアで、ライブラリとして様々な言語から使うことができる。今回は Ruby から Graphviz を呼び出すことにしよう。gem が使えるなら、コマンドラインから

```
$ gem install ruby-graphviz
```

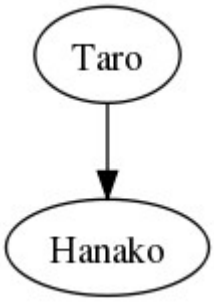
とするだけでインストールされるはずだ。後はスクリプトで

```
require 'rubygems'
require 'graphviz'
```

と宣言すれば使うことができる(gem から入れた場合は rubygems を require するのを忘れずに)。簡単な例を挙げよう。

```
require 'rubygems'
require 'graphviz'

g = GraphViz::new("G", { :type => "digraph", :use => "dot", :output => "png",
    :file => "test.png"})
n1 = g.add_node("Taro")
n2 = g.add_node("Hanako")
g.add_edge(n1,n2)
g.output()
```

このスクリプトを実行すると以下のような図が出力される。



簡単にスクリプトの説明をしよう。GraphViz のコンストラクタで type や use などでパラメータを指定する。重要なのはファイルタイプを指定する output、ファイル名を指定する file である。作成した GraphViz のインスタンスを g で受け取り、g.add_node でノードの作成、g.add_edge で、エッジと呼ばれるノード間の関係を作ることができる。最後に output を呼べばファイルが保存される。g.add_node にはノードの名前を文字列で渡す。すると、ノードオブジェクトが返ってくるので適当に受け取ろう。こうして Taro、Hanako という名前のノードを n1、n2 という変数で受け取ったあと、二つの関係を g.add_ege(n1,n2)で与えている。これにより、n1 から n2 に、すなわち Taro さんから Hanako さんに矢印が引かれる。以後、同様にいくらでもノードやエッジを増やすことが可能だ。

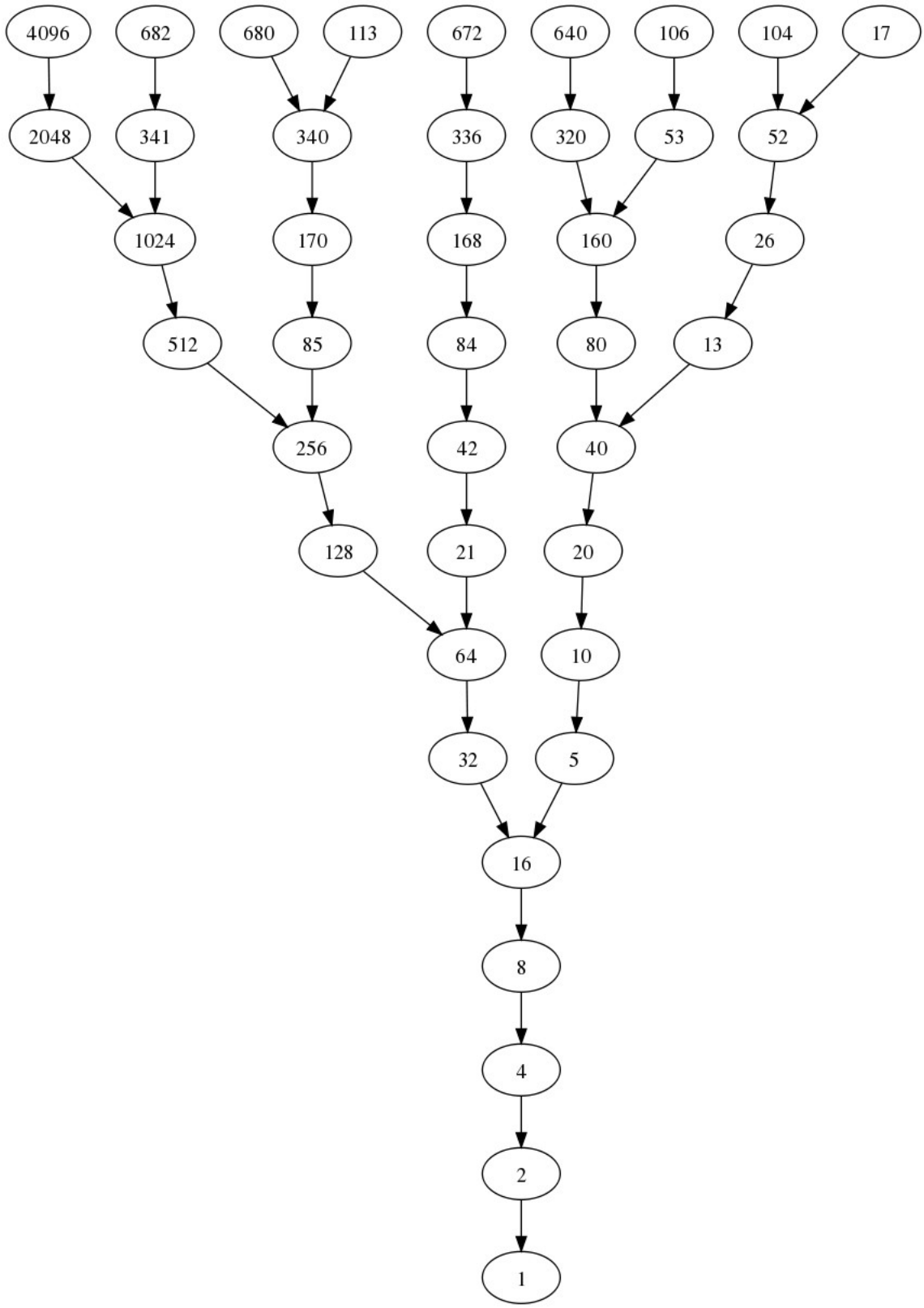
2. Collatz 予想の可視化

ではいよいよ Collatz 予想を可視化するスクリプトを作ってみよう。通常、Collatz 予想では適当な数字から 1 を目指すが、可視化のために 1 から順番に枝をのばして行き、適当な深さまで再起することにしよう。これを実現するスクリプトは以下の通り。

```
require 'rubygems'
require 'graphviz'

class Point
```

```
    attr_reader :node
    def initialize (n,g,parent)
      @n = n
      @node = g.add_node(n.to_s)
      g.add_edge(@node,parent.node) if parent
    end
    def next(g,nodes)
      nodes.push(Point.new(@n*2,g,self))
      if @n > 15 and @n % 3 == 1 and (@n-1)/3 %2 != 0
        nodes.push(Point.new((@n-1)/3,g,self))
      end
  end
end

def makegraph (g,nodes)
  new_nodes = Array.new
  nodes.size.times{|n|
    nodes[n].next(g,new_nodes)
  }
  return new_nodes
end

g = GraphViz::new("G",{:type => "digraph", :use => "dot", :output => "png", :file =>
"collatz.png"})
p1 = Point.new(1,g,nil)
nodes = [p1]
12.times{
  nodes = makegraph(g,nodes)
}
g.output()
```

あえてアルゴリズムの説明はしないが、少し考えればすぐどうなっているかわかる・・・と思う。12.times となっているところを変えれば、再帰深さを変えることが可能だ。これを使って出力された図を次のページに掲載しておく。ちなみに Wikipedia における Collatz 予想の項目にある図も Graphviz によって作られている。


4096
682
680
113
672
640
106
104
17
2048
341
340
336
320
53
52
1024
170
168
160
26
512
85
84
80
13
256
42
40
128
21
20
64
10
32
5
16
8
4
2
1

3. おわりに

Ruby と Graphviz により、整数論の未解決問題の一つ、Collatz 予想を可視化してみた。

Collatz 予想に限らず、文章で読むだけでは難しい問題も、可視化してみると簡単に解法を思いつくこともある。また、何かの問題を可視化することで、それまで思い至らなかった別の側面が見えてくることもあるだろう。GraphViz は多様な出力形式をサポートしており、扱えるグラフの種類も多い。これを機に、いろんな問題の可視化に挑戦してみれば、何か面白いことを見つけることができるかもしれない。

(2010 年 8 月 25 日投稿受付;2010 年 8 月 26 日改訂)

Python の文法

TSNET スクリプト通信版　草稿　第３回

6.オブジェクト

6-1 オブジェクト総論

6-1-1 Python オブジェクト

「概論」でも少し詳しく触れましたが、Python で言うところのオブジェクト（Python オブジェクト）とは、つまるところ被演算子(operand)の全てを指します。

被演算子とは、演算子(operator)によって演算される対象、という意味です。

この被演算子が全てクラス・インスタンスである、というのが「Python は全てオブジェクト」という意味です。
オブジェクトにはどのようなものがあるのか、具体的に列挙してみましょう。

- **数値などの原子オブジェクト**
- **リストなどのコンテナオブジェクト**
- **クラス定義によって（あるいは type 関数によって）定義されたクラスオブジェクト**
- **クラスより生成されたインスタンスオブジェクト**
- **クラスのクラスにあたるメタクラスオブジェクト**
- **関数オブジェクト、メソッドオブジェクト、組み込み関数及びメソッドオブジェクト**
- **反復子オブジェクト、及び生成子オブジェクト**
- **モジュールオブジェクト及びパッケージオブジェクト**

Python のオブジェクトは、全て「第一級オブジェクト(first-class object)」であるといわれます。
第一級オブジェクトとは、あるプログラミング言語において、その言語における他のオブジェクトと比べて制限なしに使用できるオブジェクトのことを指しますが、「何が出来れば制限なしか」という問題は非常に微妙ですので、とりあえずここでは以下の条件を満たすものを「第一級」と考えることにします。

- **変数として名前がつけられる**
- **データ構造に組み込める**
- **手続きに引数として渡せる**
- **手続きの結果として返される**

以下に、「概要」で述べた項目をさらに詳しく説明しながら、見比べてみたいと思います（⇒「」内がファーストクラスの要件)

・識別子(identifier)に束縛(bind)できる。すなわち代入可能⇒「変数として名前がつけられる」
Python からはオブジェクトそのものにはアクセスできません。
そのため、識別子という名前（名札）を用いてアクセスします。
識別子はあくまで単なる名札であり、ユーザはオブジェクトに対して自由に識別子を割り当てることができます。
識別子というシンボルにオブジェクトが関連付けられることを束縛(bind)といいますが、Python ではこの行為を代入(assignment)と言っています（もう少し正確に言うなら「識別子はオブジェクトを参照している」といいます)
識別子の参照先のオブジェクトは、自由に付け替え可能です。
よって、for 文などでは、一つの識別子に次々とオブジェクトを連続して代入します。
また、別名(alias)をつけることにより、ドットによる参照名を含む長い名前を、簡単で手軽な名前

で扱うことができます。

・属性として代入可能⇒「データ構造に組み込める」
識別子として代入可能なオブジェクトは、基本的にオブジェクトの属性として代入することも可能です。
ただし関数オブジェクトなど、属性として登録することにより特殊な働きをするオブジェクトも存在します（正確には、クラスオブジェクトに属性として登録したのち、インスタンスを生成して、そのインスタンスから参照したとき、ということになります）

・タプルなどのコンテナ（コレクション）に収納可能⇒「データ構造に組み込める」
識別子と同様に、全てのオブジェクトはコレクションのアイテムとして登録可能です。
コレクションの参照は、属性や識別子と全く同じ形で機能します。
関数リテラル(lambda 式)にコレクションの参照を利用すると、文字通り「無名関数(anonymous functions)」を作って登録しておくことも可能です。

・関数の引数としての引渡しが可能⇒「手続きに引数として渡せる．手続きの結果として返される」
オブジェクトである以上、全てを関数の引数として引き渡すことが可能です。
この特徴は、特に「高階関数（関数を引数や戻り値として取る関数）」の実装のしやすさとして語られることが多いようです。
戻り値については書きませんでしたが、引数として渡せるのと同様、何の制限も無く戻り値としても返せます。

・必ず値を持つので、真偽判定可能
'Flase'オブジェクト、'None'オブジェクト、数値の 0、空のコレクション及び、'__bool__'メソッドで'False'を返すよう設定されたオブジェクト以外は、全て真となります。
なお、戻り値を設定していない関数は全て'None'を返すので、実行結果は'False'になります。
また、どんなオブジェクト同士でも'=='や'!='による同値比較、あるいは'is'や'is not'による同一性比較が可能です（以前は全てのオブジェクトの組み合わせに対して大小比較も可能でしたが、Python3 では廃止されました）

なお、全てのオブジェクトが同等かといえば、それは微妙です。
たとえば、Python には'marshal', 'pickle', 'shelve'といった、オブジェクトの直列化(serialize オブジェクトコードにデータ化すること)及び保存用のモジュールがあります。
これらのモジュールによって、全てのオブジェクトが直列化及び保存できるかというと、そうではありません。
'marshal'は原始的な直列化ツールで、このツールで直列化できるものが基本的に「データ化できる」ということになるのですが、原子オブジェクトおよび文字列、そしてそれらをのみをアイテムとする（入れ子になっていても構わない）コレクションしか、直列化できません。

'marshal'は基本的にコンパイルの過程で使用するツールであり、作成したオブジェクトコードのバージョン間の互換性も保証されていませんので、永続的オブジェクトとしては使用しないほうが無難です。もし、オブジェクトの永続化が目的であれば、'pickle'や shelve のような永続化ツールをお勧めします。

また、'pickle'や'shelve'は'marshal'と異なり、カスタムオブジェクトも直列化できますが、その仕組みはかなり行き当たりばったりです。
具体的に言うなら、呼び出された場所で新しくオブジェクトを作って（'__init__'による初期化は二度行いません）データ属性を複製して関連付けます。
そのためには、呼び出した先でそのオブジェクトの属するクラスが存在することが条件であり、さらにインスタンス属性以外は全て、呼び出した先でのクラス（例え内容が違っていても、名前が同じ

なら同じとする）に従います。
この'pickle'や'shelve'でも、関数オブジェクトの収納は出来ません。
関数オブジェクトをコード化するには、関数オブジェクトの持つ'__code__'属性を直列化し、その後、隠しクラスである'function'クラスを呼び出して、環境辞書を与えて再構成するしかありません。
もし、どうしても実行環境を越えて関数をやり取りしたい場合は、'eval'や'exec'といった関数を用いてその場で組み立てるのが無難でしょう。

'function'は'types'モジュールの'FunctionType'として呼び出すか、あるいは様々な使用法が可能な'type'組み込みクラスに関数オブジェクトを与えれば呼び出せます。しかし、オブジェクトの情報には、固有の情報の他に『環境の』情報も含まれるので、'eval'か'exec'(可能ならば'eval')で組み立てた方が無難です。

【補足】属性は概ねオブジェクトと同様の操作が可能ですが、プロパティーや記述子を用いることで、そのアクセスをコントロールすることが可能です。ただし、属性というのは基本的に単なる参照なので、トップレベルの名前に束縛(代入)してしまえば、その振る舞いは通常のオブジェクトと全く同じになります（ただし、参照アクセスコントロールで代入不可能となっていない場合に限ります）

6-1-2 識別子(identifier)のルール

識別子については、これまで簡単にしか説明してきませんでしたので、ここで命名規約を確認しておきます。
まず、Python で識別子に使用できる文字はアルファベットと数値とアンダースコアです。
本当は日本語などのユニコード文字も使えたりするのですが……

```
>>> 富士山 = 3776
>>> 富士山 * 2
7552
>>>
```

……マルチバイト文字を識別子に利用することは、お勧めできません（マルチバイト文字は、文字列として使用するにとどめておいたほうが無難です）
シングルバイト文字(ASCII 文字)を利用する場合は、予約語として登録されている言葉は、識別子としては使用できません。

予約語(reserved word = keyword)は、'keyword'モジュールの'kwlist'というリストに入っています

また、文字の先頭はアンダースコアかアルファベットのどちらかで、数値は使用できません（数値が先頭にくると、数値リテラルと判定されます）
２文字目以降は、アルファベットかアンダースコアか数値のどれかを使用します
どれだけ長い名前をつけても構わないようですが、システムは頭から 255 文字までしか認識しないようです。
スペースやタブはオブジェクトの区切りと見做されるので、どうしても文節を区切りたい場合はアンダースコアか、大文字小文字を混ぜて使います。
識別子の大文字と小文字は完全に区別されます。
システムに直結する特殊な名前は、前後にダブルアンダースコアを置くのが普通('__main__'など)ですので、この形式の識別子はユーザが自由に設定して使わないほうが良いでしょう。
さて、モジュールの項で、'__all__'属性を用いて'from'-'import *'構文による一括導入をブロックする方法を説明しました。
特定のオブジェクトについて'from'-'import *'構文から外したい場合は、識別子の先頭にアンダースコアを一つつけます（２つ以上つけても可）

```
# tmod.py
x = 100
```

```
_x = 200
__x = 300
def f(): return 100
def _f(): return 200
```

```
>>> from tmod import *
>>> x
100
>>> _x        #【導入されていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_x' is not defined
>>> __x              #【導入されていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '__x' is not defined
>>> f()
100
>>> _f()      #【導入されていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_f' is not defined
>>>
```

なお、この書式は'from'-'import *'文を抑制するだけであって、'from'-'import'文による名指しでの参照や、モジュールからの参照をブロックする機能は一切ありません。

```
>>> from tmod import _x
>>> _x        #【導入可能】
200
>>> import tmod
>>> tmod._f()        #【参照可能】
200
>>>
```

6-1-2-1 属性名の重複を防ぐ(name mangling)

識別子の一種ではありますが、属性名には特殊属性や特殊メソッド以外に、特別な命名ルールがあります。

ネームマングリング(name mangling 名前切り刻み？)と呼ばれるこのルールは、アンダースコア二つで始まる属性名を「クラス定義文内に」書いた時に、以下のように自動的に変更します。

【定義時】__属性名 ⇒ 【定義後】_クラス名__属性名

これは、クラス属性／インスタンス属性双方に作用します。

説明だけではなかなか判りづらい概念なので、例を見ていきましょう。

```
>>> class C:
...    __x = 100
...    def setx(self):
...      self.__x = 200
...
>>> c = C()
>>> C.__x     #【変更前の名前は削除されている】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'C' has no attribute '__x'
>>> C._C__x  #【変更されている】
```

```
100
>>> c._C__x
100
>>> c.setx() #【定義内のメソッド定義でも……】
>>> c._C__x  #【変更されている】
200
>>> C.__y = 500     #【しかし、直接代入を行うと】
>>> C.__y    #【変更されない】
500
>>> c.__y
500
>>> C.f = lambda self:self.__x   #【同じく代入によるメソッド定義も】
>>> c.f()                  #【自動変更は行われない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
AttributeError: 'C' object has no attribute '__x'
>>>
```

繰り返しますが、ネーム・マングリングが行われるのは、クラス定義を行った時だけです（クラス定義内で定義されたメソッドも該当します）

後から追加されたクラス属性／インスタンス属性／メソッドなどは全て、この処理は行われません。

このような機能が何の役に立つのかといえば、インスタンス変数やクラス変数を「プライベートな値の受け渡し」に利用するタイプのクラスを「複数継承した」場合などに、混乱せずに済むということです。

以下は「初めての Python」に出ていた例に手を加えたものです。

```
>>> class C0:
...   def m1(self): self.v = 100
...   def m2(self): print(self.v)
...
>>> class C1:
...   def mA(self): self.v = 500
...   def mB(self): print(self.v)
...
>>> class C2(C0, C1): pass
...
>>> c = C2()
>>> c.m1()
>>> c.m2()
100
>>> c.mA()
>>> c.mB()
500
>>> c.m2()
500
>>>
```

これを見ると、"C0"で"m1"→"m2"への値の受け渡しとして使用されていた属性"v"が、"C1"では"mA"→"mB"への値の受け渡しとして使われていたために、"mA"での変更が"m2"という別のクラスから継承されたメソッドに影響を及ぼす結果となります。

これを防ぐために、"v"ではなく、ネーム・マングリングを行う"__v"を使用すると、次のようになります。

```
>>> class C0:
...   def m1(self): self.__v = 100
...   def m2(self): return self.__v
```

```
...
>>> class C1:
...   def mA(self): self.__v = 500
...   def mB(self): return self.__v
...
>>> class C2(C0, C1): pass
...
>>> c = C2()
>>> c.m1()
>>> c.m2()
100
>>> c.mA()
>>> c.mB()
500
>>> c.m2()   #【c.mA の影響を受けていない】
100
>>> c._C0__v #【C0 の__v】
100
>>> c._C1__v #【C1 の__v】
500
>>>
```

ネーム・マングリングは不意の変数汚染を防ぐ効果はありますが、例によって隠蔽の効果はありません。

多重継承を前提とした部品クラスを作る際などに、変数汚染防止として使用するのが一般的であると考えられます。

属性名が煩雑になりますので、多用は禁物です

6-1-3 クラスオブジェクト

6-1-3-1 クラスオブジェクトの概要

クラスとは、第一義的にはインスタンスの鋳型ですが、Python のクラスはそれだけではありません。

クラスオブジェクトは、'type'というクラス（メタクラス）のインスタンスであるオブジェクトです。

クラス内はモジュールと同様、独立した名前空間です。

クラス定義とは、この独立した名前空間の中で実行される処理と考えることが出来ます。

ただし、クラスの名前空間はモジュールと異なり（トップレベルもモジュール扱いです）、入れ子にはなりません。

クラスの中で定義された関数、即ちメソッドも、その『外側』はクラス内ではなくその外（トップレベルあるいはモジュール）です。

メソッドにクラス内部の名前空間のスコープを導入するには、クラスメソッドを設定します。

クラスメソッドの第一引数は、クラスへの参照となります。

クラスの中には、クラス属性としてオブジェクトを格納することが可能です。

このため、クラスを属性のコンテナ（特に写像オブジェクトである辞書の代わり）として使用することも可能です。

機能が似通ったツール的な関数定義をまとめるには、クラスは非常に便利です（オブジェクトとしての『持ち運び』が若干不便な関数は、モジュールの特定クラスにまとめておくと呼び出しやすくて便利です）

ただし、単なるデータ格納辞書の代用として使用するなら、インスタンスオブジェクトの方が優れています。

理由は、第一に、クラスには予め登録された属性が存在しますが、インスタンス属性は初期化しない限り空です。

第二に、インスタンス属性は普通の辞書で実装されているため、辞書に変換するのが容易ですが、クラス属性は「辞書のようなもの」で実装されているため、変換に一手間余分にかかります。

インスタンスの鋳型としてのクラスは、大規模なプログラミングでなければ実はさほど活躍しません。
Python は組み込みオブジェクトが豊富なので、それらを使用すれば大概の用事は済んでしまいます。
もちろん、簡単に使えるクラス定義や変幻自在なインスタンスオブジェクトは、効率的に使えば非常に強力なツールです。
しかし Python は「クラス定義」がプログラミングそのものと同義であるというタイプの「純粋な」オブジェクト指向プログラミング(OOP)言語ではありません。
「便利なものを要領よく、できればシンプルに」が Python スタイルです。

6-1-3-2 名前空間としてのクラス

クラスは、モジュールとは一部異なった形式の名前空間です。

実際のルートの実行環境であるトップレベルは、モジュールと全く同じ

クラス定義は、クラスという名前空間で処理を行える唯一の処理です。
クラス定義が終了した後は、クラスは単なるオブジェクトのコンテナとして扱われます。
コンテナとしてのクラスの説明は次章に回し、ここではクラス定義で行われる処理について見ていきましょう。
他の言語では、クラス定義は内部のメソッドの定義と同義であるものが多いようですが、Python におけるクラス定義では、メソッド定義は（重要ではありますが）一つの要素に過ぎません。
クラス定義の中では、基本的にどんな実行文も行うことができます。
ただし、クラス定義内の処理とトップレベルでの処理では、以下の点で異なります。

・クラス定義内に記述された処理は、クラス定義終了時に即座に実行される
・クラス定義内で定義された変数や関数は、クラスの名前空間に属する属性として登録される（定義文が実行された結果）
・クラス定義内は、その周囲のモジュール（あるいはトップレベル）に対してスコープを持つ
・クラス定義内の変数の参照先は、グローバルからローカルに途中で切り替わってもよい（関数との違い）

スコープについては「式と文」の「スコープルール」の章で詳しく話しましたので繰り返しませんが、クラスと関数の中の変数の扱いは異なることを確認しておいてください。
既に説明しましたが、加えて次の点も重要です。

・クラス内で定義された関数（メソッド）内では、クラスの外側（モジュール／トップレベル）に対してスコープを持つが、クラス内にはスコープを持たない

他言語を修得された方がこぞって首をかしげる点（実は私も、その意味を理解するまで首を傾げました）ですが、メソッド内からクラス定義内へのスコープは一切ありません。
クラス定義内の変数や関数にアクセスする際には、トップレベルからクラスにアクセスする時のようにクラス名から参照するか、あるいはクラスメソッドを定義し、その第１引数にクラスへの参照（ポインタ）を取得することによってアクセスするしか方法はありません。
例を見てみましょう。

```
>>> class C:
...   x = 666
...   def mA(): print(x)
...   def mB(): print(C.x)
...   @classmethod
```

```
...   def mC(cls): print(cls.x)
...
>>> x = 777
>>> C.mA()   #【トップレベルを参照してしまう】
777
>>> C.mB()   #【外側からクラス属性にアクセス】
666
>>> C.mC()   #【クラスメソッドとしてクラス参照から】
666
>>>
```

スコープルールの細則については「スコープルール」の章を参照してください。
ところで、Python のクラス定義は先にも述べた通り「実行文」です。
定義に何の関係もない処理をおこなっても、一向に構いません。
実行文は定義直後に実行されます。

```
>>> class C:
...   print('Hello World!')
...
Hello World!
>>>
```

名前空間が独立しているということは、変数の汚染せずに'for'文などを使用することが出来ることを意味します。

```
>>> x = 100
>>> class C:
...   for x in range(10):
...     print(x, end=' ')
...   else:
...     print()
...
0 1 2 3 4 5 6 7 8 9
>>> x        #【x は変更されていない】
100
>>>
```

以上から、「再利用するわけではないので関数化して実行するのは面倒だが、変数を多用するので変数汚染を気にせずに計算するメモ的なサブルーチンブロックが欲しい」という時などに、ちょっと便利です。

実際、そのようなサブルーチンブロックが必要なのは、多分プログラミングに慣れていない初心者の方か、あるいは構造化以前のかなり古典的な言語に慣れ親しんだ方にに限られると思います。汚染する可能性のあるような変数名はもともと「不変である」ことを前提にプログラムは書きませんし、トップレベルの変数の融通は極力避け、クラスやオブジェクトのブロックの中に収納するのが普通でしょう。

もちろん「参照（属性参照やインデックス参照）」には注意してください。
単なる代入は別オブジェクトへの参照の切り替えですが、参照を通した変更は、オブジェクトそのものの変更になることには代わりはありません。

```
>>> x = []
>>> class C:
...   x.append(10)  #【リストに要素を追加】
...
>>> x
[10]                #【変更されている!】
```

```
>>>
```

クラス内で定義されることは、トップレベルでオブジェクトが定義されることと本質的には同じです。
クラス定義／メソッド定義は、クラス定義内の実行文の実行によって行われるというメカニズムを理解しておくと、Python のクラスシステムを柔軟に使用することができるようになるでしょう。

6-1-3-3 コンテナとしてのクラス

通常のインスタンスオブジェクトと同じく、クラスオブジェクトも属性に他のオブジェクトを収納するコンテナとして使用することができます。
ただし、通常のデータの容れ物として考えるのなら、クラスではなくインスタンスを利用する方が便利です。
理由は概要で少し触れましたが、３点あります。

・クラスオブジェクトには'__dict__'属性に予め収納されている属性（'__dict__','__module__','__weakref__','__doc__'など）があるので、辞書に変換する際に面倒
・クラスオブジェクトの'__dict__'属性は、辞書ではなく'dict_proxy object'＝辞書の代用オブジェクト、つまり「辞書的な動作はするが辞書ではない」オブジェクトなので、辞書として使うには一手間かかる
・コンテナデータを継承（単純／多重継承問わず）すると、扱いがややこしくなる

当然の話ですが、出来る限りシンプルにすることが間違いに対する最大の防御です。
もしクラスをコンテナとして使用するなら、一般データでなく、関数のコンテナとしてのみ使用すべきでしょう。
しかも副作用を伴った『手続き』的な関数ではなく、引数を取って戻り値を利用する、関数らしい関数のコンテナに向いています。
Python では(Python3 以降)、何のデコレータもつけずに定義した関数を『クラス名』から呼び出すと、静的メソッド(static method)として扱われます。
静的メソッドとは、メソッドでありながら第１引数に何の特別な意味も無いメソッド（つまりただの関数）を指します。

Python では、クラスに登録されているメソッドをインスタンスから参照すると、その第１引数はインスタンス自身のセルフポインタ（自身参照）になります（詳しくは後述の「インスタンス」の章を参照のこと）

クラスを関数のコンテナとして利用すると、以下のようなメリットがあります。

・識別子を節約できる
クラスにまとめると、トップレベルで管理する識別子はクラス名だけですので、トップレベルでのプログラミングに影響を与えにくくなります。

・整理しやすい
ファイルをディレクトリ（フォルダ）で管理するのと同じ感覚で、関数を管理できます。

以上二つは、基本的にインスタンスオブジェクトで管理しても同じということになります。
「Python で」クラスを関数のコンテナとして使うと便利な理由は以下の通りです。

・通常の関数と同じフォーマットでクラス定義内に定義できる
クラスで定義し、インスタンスで利用すると、基本的なメソッドのルールとして、第１引数を（利用しようとしまいと）インスタンス自身の参照用として確保する必要があります。
タイプ数自体は大したことはありませんが（お望みなら"self"とせずに"s"一文字でも結構です）不要なモノを設定するのは、コード自体の見栄えとして決して良いものではありません。

もしクラスから参照するように静的メソッドとして使用するなら、'@staticmethod'というデコレータを追加する必要があります（コレのタイプ数はちょっといただけませんよね？）
また、インスタンス属性として設定する（代入による定義）場合、確かに第１引数に対する制限はなくなりますが、関数リテラル(lambda 式)ならばその内容に制限があり、外側で関数定義を行うならば、識別子の節約にはならないでしょう（後で消去すればいいわけですが、面倒ですよね）

・インスタンスを作るより、一手間少ない

当然ですが、クラスをつくらなければインスタンスを作れないわけで、クラスで出来る（あるいはクラスのほうが使いやすい）ものであれば、あえてインスタンスを作る必要は無いでしょう。

尤も、'type'クラスを特殊な方法で使用することにより作成される『無名クラス(anonymous class)』のインスタンスを作ることは可能です。データ属性のコンテナとしてのみ使用するインスタンスオブジェクトは、この方法で作ると手軽です。

```
>>> o = type('',(),{})()
>>> o.x = 100
>>> o.x
100
>>>
```

詳細は、「オブジェクト」の「'type'関数」の章で説明します。

さて、クラスを使った方が便利な関数コンテナとしての使用法についてのみ、ここでサンプルを見ていきましょう。

```
>>> class C:
...   def sm(x): return x        #【引数を返すだけの関数】
...   def im(self, x): return x  #【使わないのに二つ引数が要る】
...
>>> C.sm(100)                    #【非常にシンプル】
100
>>> c = C()
>>> c.sm(100)                    #【引数が多すぎる】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm() takes exactly 1 positional argument (2 given)
>>> c.sm()                 #【意図しない使い方に】
<__main__.C object at 0x00BF3A90>
>>> c.im(100)                    #【これが正常】
100
>>> c.imL = lambda x: x          #【lambda でもできるものならこれでも可能は可能】
>>> c.imL(100)
100
>>> def imD(x): return x  #【難しいものなら、外部で定義が必要】
...
>>> imD(100)               #【こういうもの】
100
>>> c.imD = imD
>>> c.imD(100)
100
>>>
```

関数オブジェクトはオブジェクトの中でも特殊な存在で、直列化（データ化）するのが難しいため、モジュールレベルで導入され（モジュールは基本的にスクリプトファイルかコンパイルされたスクリプトファイル）、必要な場所で再構築されて利用されます（動的言語として、Python ではプログラム実行中に全てのオブジェクトの定義／実行が可能です）

モジュールの中身をすっきりと整理するために、クラスで適当に仕分けするのは効果的でしょう。
ところで、データを共有する関数群を実装する際は、基本的にはオブジェクトを使用しますが、インスタンスのコピーを多用しない場合は、クラスだけで作ることも可能です。
ただしクラス属性を参照するメソッドは、クラスメソッドとして定義しておく必要があります。
'@classmethod'というデコレータを何度も書くのは面倒（コピペすらイヤ）という方は、"cm = classmethod"などとして、"@cm"といった略号で定義する方法が一番簡単のようです。
ローカル辞書を呼び出す'locals'関数を用いて反復で設定する方法が無いわけではありませんが、あまり簡単にはならないようです。
とはいえ、参考までにサンプルを書いておきましょう。

```
>>> class C1:
...   x, y, z = 100, 200, 300
...   cm = classmethod
...   @cm
...   def cmx(cls): return cls.x
...   @cm
...   def cmy(cls): return cls.y
...   @cm
...   def cmz(cls): return cls.z
...
>>> c1 = C1()
>>> c1.x, c1.y, c1.z = 1, 2, 3
>>> c1.cmx(), c1.cmy(), c1.cmz()
(100, 200, 300)
>>> class C2:              #【呼び出す関数が比較的簡単な場合は有効】
...   x, y, z = 100, 200, 300
...   for k in 'xyz':
...     locals()['cm'+k] = classmethod(eval('lambda cls:cls.'+k))
...
>>> c2 = C2()
>>> c2.x, c2.y, c2.z = 1, 2, 3
>>> c2.cmx(), c2.cmy(), c2.cmz()
(100, 200, 300)
>>>
```

"C2"は、単にPythonの記述性の高さを示すために書いただけのものです。
可読性を重視するPythonに於いて、このような書き方は明らかに邪道ですので、実際に使うべきではないでしょう。

'locals'関数、'eval'関数については、「組み込みオブジェクト」の章で詳しく説明します。

6-1-3-4 インスタンスの鋳型としてのクラス

クラスの第一義的にはインスタンスの鋳型としての役割、ということになります。
しかし実際のPythonのプログラミングでは、ユーザがカスタムクラスを定義しなければならない場面というのは、意外に多くありません。
組み込み型と呼ばれるクラス群のインスタンス（整数なども含む）さえ使えば、大概のプログラムは組めてしまいます。
ただし、プログラム規模が大きくなればなるほど、カスタムクラスの有効性は高まりますし、そうでなくともクラスの仕組みの知識は、組み込み型のオブジェクトを有効利用する際にも利用できます。
クラス定義そのものについては、「定義」の章で詳しく説明しましたので、ここでは「クラス」自体の性質を、「インスタンスの鋳型」という観点から見ていきましょう。

まずクラス定義文の本質は、前章・前々章でも述べたとおり、「クラスという名前空間に於ける実行文の処理」です。

実行の結果、変数が定義されればクラス変数に、関数が定義されればメソッドになります。
クラス変数は、インスタンス変数で遮蔽されない限り、インスタンスから普通に属性参照できます（ただしインスタンスから属性代入すると、それはインスタンス属性になってしまいます）
メソッドも同じく、インスタンス属性で遮蔽されない限りは、インスタンスから呼び出して使用することが出来ます。
属性名の識別子は、関数とその他のオブジェクトを区別しないことに気をつけてください。
クラスで"meth"という名前のメソッドを設定していたとしても、インスタンスで"meth"という属性に代入すれば、直接の参照はできなくなってしまいます。
また、インスタンス属性に代入した関数は、「メソッド」にはならず、単なるインスタンス属性の関数オブジェクトという扱いになりますので注意してください（具体的には第１引数に自身参照を持たない）
例を見てみましょう。

```
>>> class C:
...   x = 100
...   y = 200
...   def meth(self):
...     return 300
...
>>> c = C()
>>> c.x             #【クラス属性を参照】
100
>>> c.x = 500       #【インスタンス属性で遮蔽】
>>> c.x
500
>>> c.meth()
300
>>> c.meth = 0      #【インスタンス属性で遮蔽】
>>> c.meth()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> c.meth = lambda self:600     #【関数を定義】
>>> c.meth()               #【メソッドにはならない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 positional argument (0 given)
>>> c.meth(c)
600
>>> C.meth2 = lambda self: self.y      #【クラスに定義すると】
>>> c.meth2()                          #【メソッドとして有効】
200
>>> c.y = 2000
>>> c.meth2()       #【しかもしっかりインスタンス変数を参照】
2000
>>>
```

メソッドのスコープについては様々なところで紹介してきましたが、基本的に自由変数（関数内で定義していない変数をいきなり参照すること）的な使用法は避けるのが、判りやすいコードを書くポイントです。
メソッドでは関数内に定義された変数と引数、そして第１引数の自身参照のみを使用するのが普通です。

当然、組み込みオブジェクトを呼び出す場合を除きます

6-1-4 インスタンスオブジェクト

6-1-4-1 Python とオブジェクト指向

Python は ver.1.0 として登場した時から、オブジェクト指向プログラミング言語(Object Oriented Programming language)であると宣言しています。

しかし同時に、Python の OOP はオプションであり、Python の総体は「多主義対応(マルチパラダイム multi-paradigm)言語」であるとも言っています。

たしかに Python は、手続き型（構造化）言語としても、関数型言語としても使用できますが、どれ一つとして要件を完全に満たすことは出来ません。

構造化言語の要件すら満たさない筈は無い、と思われる方は、反復ループ構造の名前空間が独立していないことに注目してください。関数型は、参照透過性でアウトです。

よって、といっては何ですが……OOP に必須といわれる「カプセル化」や「隠蔽」の機能は、Python には一切無いと言っても過言ではありません。

Python オブジェクトの属性は基本的に無制限にアクセス可能（C 言語として書かれたモジュール内で制限された場合を除く）で、外部からの変更の制限も殆どありません。

プロパティなども、クラスを経由して破壊することは可能です

この性質は、他の OOP 言語の流儀に慣れた方には、恐怖として映るようです。

某有名な言語の作者は、インスタンス変数へ直接アクセスできることを「内臓が外に飛び出している」という表現をしましたが、Python はまさにそういったスプラッタな言語なのです。

しかし、Python には記述子／プロパティなどの「誤操作防止用の制御機構」や、多重定義による重複意義を解消するためのネーム・マングリングルールなどはあります。

故意の破壊は食い止められませんが、誤操作防止のセーフティはあるということです。

そもそも動的に変化することを目的とするスクリプト言語ですから、扱うデータの安全性はともかく、スクリプトコード自体の隠蔽機構などほとんどありません（コードの著作権を護りたいのなら、その部分は Python ではなく C を使わなければなりません）

故意の破壊については、それをブロックするのはプログラミング言語ではなく、アクセスさせないシステム全体の役目です。

Python のプログラミング環境にアクセスする権利のある人は、自己責任で無制限に Python の全環境に関与できる、というのが Python の基本思想です。

・プロパティーをインスタンスから変更する方法

プロパティーで保護されているモノをわざわざ壊す場合は、相応の覚悟をしてください。

```
>>> class C:
...   @property
...   def x(self): return self._v
...   @x.setter
...   def x(self, v): pass
...   @x.deleter
...   def x(self): pass
...   def __init__(self, v=100):
...     self._v = v
...
>>> c = C()
>>> c.x
100
>>> c.x = 500
>>> c.x
```

```
100
>>> del c.x
>>> c.x
100
>>> c.__dict__
{'_v': 100}
>>> c._v = 500              #【隠し変数を変更】
>>> c.x
500
>>> type(c).x = 600         #【クラスを呼び出して属性変更】
>>> c.x
600
>>>
```

以上、Python の OOP は単にプログラマのコーディングを楽にするだけのものであって、決して安全性を保証するものではない、ということに注意してください。

6-1-4-2 組み込み型とカスタムクラス

実は、インスタンスオブジェクトというオブジェクトがあるわけではなく、全てのオブジェクトはインスタンスです。

クラスオブジェクトも type オブジェクト（メタクラス）のインスタンスですし、type オブジェクト自身も type のインスタンスです。

よって、インスタンスオブジェクトに共通な情報とは、全てのオブジェクトに共通な情報とも言えます。

しかし、同じインスタンスであっても、組み込み型(built-in type)のインスタンスであるか、ユーザが定義したカスタムクラスかによって、内容は若干異なります。

まず、ユーザが定義したクラスは、__setattr__などのアクセス制御していない限り、自由に属性に代入することが出来ます（ただし特殊メソッドのスロットに妙なモノを代入すると、思わぬ結果を招くことはありますが)

それに対して、組み込み型や、ある種のモジュールは Python コードではなく C で実装されているため、属性の参照はともかく、殆どの場合、属性への代入は制限されています。

カスタムクラスのインスタンスは'__dict__'属性を持つことによってインスタンス属性を辞書で管理していますが、クラスオブジェクト('type'オブジェクトのインスタンス)の__dict__属性は'dict_proxy object'であり、扱いが異なります。

その他の組み込み型は、そもそも'__dict__'属性を持っていないので、インスタンス属性を保存するスペースがありません。

こうした、属性への代入の可否以外は、組み込み型でもカスタムクラスでも扱いは変わりません。

以前は組み込み型は'type'、カスタムクラスは'class'と分けていたのですが、現在では統合され、どちらも'class'扱いです。それに伴って、'type'はメタクラスになりました。

Python では Smalltalk などと異なり、組み込み型はカスタマイズできませんが、組み込み型を継承したクラスをカスタマイズすることはできます。

組み込み型の拡張は、カスタムクラスを作る要領を学ぶ上で便利ですので、是非一度チャレンジしてみてください。

6-1-4-3 カスタムオブジェクト

組み込みオブジェクトに関しては、この章の後にそれぞれ見ていきますが、ここではまず、カスタムオブジェクトの基本を確認して置きましょう。

カスタムオブジェクトは、カスタムクラスを定義し、そのインスタンスを作ることによって生成します。

カスタムクラスは、特に何らかの機能を継承するのでなければ、そのまま基底クラス無しとして記

述します。

> **Python 2 までは、'object' クラスを継承しないと、プロパティーなどの新しい機能が使えなかったのですが、Python 3 になった現在では、特に'object' クラスを継承すると宣言しなくとも、全ての機能は使用できます（'object' は暗黙的に全てのクラスの基底クラスとなります）**

クラス定義については「定義」の章で詳しく説明したので、ここではインスタンス属性について少しお話します。

インスタンス属性は、個々のインスタンスオブジェクトによって管理されている属性です。

'__dict__' 属性は、インスタンス属性を格納する辞書ですので、この属性を参照すればインスタンス属性の中身を簡単に知ることが出来ます。

また、辞書オブジェクトと互換性がありますので、追加したり入れ替えたりが自由自在です。

```
>>> class C: pass   #【とりあえずクラスは空で】
...
>>> c = C()
>>> c.__dict__             #【インスタンス属性無し】
{}
>>> c.x = 100
>>> c.__dict__             #【インスタンス属性追加の確認】
{'x': 100}
>>> c.__dict__ = {'y':200}       #【インスタンス属性辞書をごっそり入れ替え】
>>> c.x                          #【今まであったインスタンス属性が無効に】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'x'
>>> c.y                          #【そして新たなインスタンス属性が】
200
>>> c.__dict__.update({'x':100}) #【辞書式の「更新」が可能】
>>> c.x
100
>>> c.y
200
>>> d = c.__dict__         #【識別子に繋いでみる】
>>> d
{'y': 200, 'x': 100}
>>> d.update({'z':300})          #【外から更新】
>>> c.z                          #【繋がっています】
300
>>> c.w = 400                    #【逆も真なり】
>>> d
{'y': 200, 'x': 100, 'z': 300, 'w': 400}
>>>
```

辞書オブジェクトのメソッドについては、「組み込みオブジェクト」の章で詳しく説明しますが、インスタンスオブジェクトの属性辞書は辞書そのものなので、様々な操作が可能です。

逆に、辞書にオブジェクトのカバーを被せると、文字列をキーとするアイテムを入力しやすくなります。

```
>>> d = {}
>>> class Dx: pass
...
>>> dx = Dx()
>>> dx.__dict__ = d
>>> d['a'] = 100    #【これが辞書のアイテムを入力する正式な方法】
>>> dx.a
```

```
100
>>> dx.b = 200              #【[]や''のようなシフト文字が無いので入力が楽】
>>> d
{'a': 100, 'b': 200}
>>> del dx.a        #【消去も可能】
>>> d
{'b': 200}
>>> del d['b']              #【ちなみに正式な辞書のアイテム消去方法】
>>> dx.__dict__
{}
>>>
```

これ以外にも「文字列をキーとする辞書」の裏技はありますが、詳細は「組み込みオブジェクト」で

インスタンス属性に関数を代入しても、メソッドにはなりません。
クラスを通さない限り、第１引数に自身参照(self reference)を持ってくる形にはなりません。
どうしても参照を渡したければ、素直に引数として渡すのが無難です。

クラスに登録する方法もありますが、当然他のインスタンスに影響も出ますので、あまりお勧めは出来ません。

最後に、lambda のように、クラス定義をせずに（空の）オブジェクトをいきなり作る方法を紹介しておきます。
これは、メタクラスである'type'を使う方法です。
ついでに、式でクラスを作る方法も書いておきます。

```
>>> c = type('',(),{})()          #【クラスを作らずいきなりオブジェクト】
>>> c
<__main__. object at 0x00BA4BB0>
>>> C = type('C',(),{})                  #【式でクラスを……↑はこれを利用】
>>> C
<class '__main__.C'>
>>>
```

'type'オブジェクトは非常に曲者ですので、後ほどじっくりとご説明します。

6-1-4-4 特殊メソッドとオブジェクト機能の拡張

特殊メソッド(special method)とは、実装することにより、通常のメソッドとしての働き以外に特別な働きをするメソッドです。

Python のドキュメントでは、正式には"specal method name"つまり「特殊メソッド名」として登場します。つまり、決められた特殊な「名前」のメソッドが、特別な意味を持つという意味です。ただし本書では面倒なので「特殊メソッド」と記述します。

特殊メソッドにも、「通常のメソッドとしての働き」はあることに注意してください。

特殊メソッドは、いくつかの例外(__new__やアクセッサなど)を除いて、通常のメソッドの使用法として通常の動作をします。

特殊メソッドは書式として、識別子の頭と終わりにダブルアンダースコア'__'をつけます。
特殊メソッドにはいくつかに大別できますが、とりあえずこれまでに見てきたものをまとめましょう。

＜「定義」の章で説明＞
・生成と消滅の特殊メソッド(__init__,__des__,__new__)
・属性参照に関する特殊メソッド(__getattr__, __getattribute__, __setattr__, __delattr__)
・記述子(descriptor)に関する特殊メソッド(__get__, __set__, __delete__)

＜「式」の章で説明＞
・演算に関する特殊メソッド(__add__, __radd__, __sub__, __rsub__, __mul__, __rmul__, __truediv__, __rtruediv__, __floordiv__, __rfloordiv__, __mod__, __rmod__, __pow__, __rpow__, __lshift__, __rlshift__, __rshift__, __rrshift__, __and__, __rand__, __xor__, __rxor__, __or__, __ror__, __lt__, __le__, __eq__, __ne__, __gt__, __ge__, __neg__, __pos__, __invert__)

以上については、各々の説明している箇所を参照してください。
以降は、それ以外の特殊メソッドについて説明します。

6-1-4-4-1 基本的な拡張

リファレンスマニュアルの「基本的な拡張(basic customization)」の項目に説明されているメソッドは、'__new__', '__init__', '__del__', '__repr__', '__str__', '__format__', '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__hash__', '__bool__' で、このうち、今まで説明していないものを説明します。

■表現(representation)メソッド
書式：__repr__(self) ⇒ 文字列
動作：インタープリタで表示される文字を返す（戻り値は必ず文字列）

この値自身を評価した際やに表示される文字を決めます。
また、repr 関数に渡した際の戻り値（文字列）も、このメソッドの結果が使用されます。
さらに、'__str__'メソッドが設定されていなかった場合には、print 関数に渡した時の表示や、str 関数に渡した際に得られる文字列などに使用されます。

■文字表現(string representation)メソッド
書式：__str__(self) ⇒ 文字列
動作：print 関数に渡した時に表示される文字を返す（戻り値は必ず文字列）

このオブジェクトを print に渡した際に表示される文字を決めます。
また str 関数に渡した際に得られる文字列を決定します。
ただし、'__repr__'メソッドの代わりにはなりません。

```
>>> class C: #【repr と str を設定】
...   def __repr__(self): return 'repr C'
...   def __str__(self): return 'str C'
...
>>> class D: #【repr を設定】
...   def __repr__(self): return 'repr D'
...
>>> class E: #【str を設定】
...   def __str__(self): return 'str E'
...
>>> c, d, e = C(), D(), E()
>>> c, d, e  #【str は repr の代わりにはならない】
(repr C, repr D, <__main__.E object at 0x00BF3A50>)
>>> print(c, d, e)  #【repr は str の代わりになる】
str C repr D str E
>>> repr(c), repr(d), repr(e)
('repr C', 'repr D', '<__main__.E object at 0x00BF3A50>')
>>> str(c), str(d), str(e)
```

```
('str C', 'repr D', 'str E')
>>>
```

■書式指定(format)メソッド
書式：__format__(self, format_spec)： ⇒ 文字列
動作：format_spec に記されたオプションによる文字表現を返す

これは、'str'型の'format'メソッドの章、あるいは組み込み関数'format'の章で詳しく説明する機能ですが、ここではカスタムオブジェクトの視点から、さらっと説明しておきましょう。
数値は、精度を指定して表示する（文字列に変換する）ことが可能です。
例えば、小数点２桁まで表示してみましょう。

```
>>> format(10,'.2f')
'10.00'
>>>
```

小数点２桁の精度で表示するには、オプションの'.2f'を用います。
これは、文字列の書式機能（後述）でも、同じ表現が利用できます。

```
>>> '答えは{}です'.format(10)
'答えは10です'
>>> '答えは{:.2f}です'.format(10)
'答えは10.00です'
>>>
```

このように、オプションに対応して表現を変えるような場合に、'__format__'が使えます。

```
>>> class C:
...   def __repr__(self):
...     return 'class C item'
...   def __format__(self, format_spec):
...     if format_spec == 'x':
...       return 'X option! C'
...     else:
...       return self.__repr__()
...
>>> c = C()
>>> format(c,'x')
'X option! C'
>>> format(c,'y')
'class C item'
>>> '{}'.format(c)
'class C item'
>>> '{:x}'.format(c)
'X option! C'
>>>
```

この機能を有効活用すると、パラメータを表示する時などに、文字列をテンプレートとして活用することが可能となります。
たとえば、あるオブジェクトの属性を表示する場合の例を見てみましょう。
文字列の format メソッドは、オブジェクトの属性も表示できます。

```
>>> class C:
...   x = 100
...
>>> c = C()
```

```
>>> print('{.x}'.format(c))
100
>>>
```

ところが、隠し属性としてネーム・マングリングされている上に、アクセスでは「値を返す」のではなく「表示する」だけというような、念入りに保護してある属性を、表示として使用したい場合などに、__format__属性は威力を発揮します。

サンプルのような不変体なら問題ありませんが、リストを受け取った時など、うっかり返すと変更される可能性があるので。

```
>>> class C:
...   def __init__(self, x=0, y=0, z=0):
...     self.__securet_x = x     #【ネーム・マングリングする厄介な名前】
...     self.__securet_y = y
...     self.__securet_z = z
...   def __repr__(self):
...     return ''
...   def x(self):         #【内容を返さず表示するだけ】
...     print(':x =', self.__securet_x)
...   def y(self):
...     print(':y =', self.__securet_y)
...   def z(self):
...     print(':z =', self.__securet_z)
...   def __format__(self, fs):  #【しかし format には対応】
...     if fs == 'x':
...       ret = self.__securet_x
...     elif fs == 'y':
...       ret = self.__securet_y
...     elif fs == 'z':
...       ret = self.__securet_z
...     else:
...       ret = self
...     return str(ret)
...
>>> c = C(100, 200, 300)
>>> c.x()    #【表示】
:x = 100
>>> c.y()
:y = 200
>>> c.z()
:z = 300
>>> #【シンプルなテンプレート】
... f0 = 'c:x = {}\nc:y = {}\nc:z = {}'
>>> #【頑張って属性指定】
... f1 = 'c:x = {0._C__securet_x}\nc:y = {0._C__securet_y}\nc:z = {0._C__securet_z}'
>>> #【スマートにフォーマット】
... f2 = 'c:x = {0:x}\nc:y = {0:y}\nc:z = {0:z}'
>>> #【表示するときゃ、大変だ】
... print(f0.format(c._C__securet_x, c._C__securet_y, c._C__securet_z))
c:x = 100
c:y = 200
c:z = 300
>>> #【頑張った甲斐あって、スマートに表示】
... print(f1.format(c))
c:x = 100
c:y = 200
c:z = 300
>>> #【でも、これでもOK!】
... print(f2.format(c))
c:x = 100
```

```
c:y = 200
c:z = 300
>>>
```

プロパティや記述子よりシンプルに、値を保護することが可能です。

■ハッシュ可能化メソッド
書式：__hash__(self)： ⇒ 整数
動作：'hash'関数に渡した際の戻り値を設定。__eq__メソッドを設定したオブジェクトが、辞書のキーや集合などの要素になることができる

カスタムクラスに於いて、同値性テストを行う__eq__メソッドを設定していない場合は、この__hash__メソッドは単に'hash'関数に渡した結果を設定するだけの動作しかしません。
また、__eq__メソッドを設定していなければ、そもそもカスタムクラスのオブジェクトは辞書のキーや集合の要素になることができます。
しかし「同一性と別レベルで同値性の比較が可能」なカスタムオブジェクトを、辞書のキーや集合の要素として扱いたい時には、__hash__メソッドを設定しておく必要があります。
例を見てみましょう。
属性'v'の比較で同値性を判断するオブジェクトを作ってみます。

```
>>> class C:
...   def __init__(self, v=0):
...     self.v = v
...   def __repr__(self):
...     return 'C:' + str(self.v)
...   def __eq__(self, other):
...     if self.v == other.v:
...       return True
...     else:
...       return False
...   def __hash__(self):
...     return int(self.v)
...
>>> c0 = C(100)
>>> c1 = C(100) #【c0と同値】
>>> c2 = C(200)
>>> c0, c1, c2
(C:100, C:100, C:200)
>>> c0 == c1
True
>>> c0 == c2
False
>>> {c0, c1, c2} #【C:100が一つ処分される】
{C:200, C:100}
>>>
```

「リテラル」を持つオブジェクトを作る時には、こういった機能が便利になります。

■真偽値メソッド
書式：__bool__(self)： ⇒ 真偽値
動作：真偽チェックの結果を設定する

カスタムオブジェクトは、真偽テストで基本的には「真」と見做されます。
もし、常に「偽」と見做されたり、あるいは条件によっては「偽」とみなされたりしたい場合には、この__bool__メソッドを設定します。

```
>>> class C:
...   def __init__(self, v):
...     self.v = v
...   def __bool__(self):
...     if self.v == 0:
...       return False
...     else:
...       return True
...
>>> F = C(0)
>>> T = C(1)
>>> print('真' if F else '偽')
偽
>>> print('真' if T else '偽')
真
>>>
```

__bool__の値がFalseを返す場合、『偽(false)』と見做されます。

6-1-4-4-2 属性アクセスの拡張

属性アクセスの拡張(customizing attribute access)の章に説明されているのは、'__getattr__', '__getattribute__', '__setattr__', '__delattr__', '__dir__'の５つです。

このうち'__dir__'以外は全て「属性アクセスに関するメソッド」で説明されています。

■属性閲覧制御メソッド
書式：__dir__(self) ⇒ リスト
動作：'dir'関数に渡された時に返すリストを設定する

'dir'関数は非常に使いでのある関数で、特にPythonのハックには不可欠です。

しかし、この関数で得られる表示は非常に長ったらしのが普通です。

適当に空のオブジェクトをつくって、その表示を見てみましょう。

```
>>> class C:
...   pass
...
>>> c = C()
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribut
e__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__re
duce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_
_weakref__']
>>>
```

これを制御するのが'__dir__'属性です。

とりあえず「見るな！」というのであれば、空リストを返します。

```
>>> class C:
...   def __dir__(self):
...     return []
...
>>> c = C()
>>> dir(x)
[]
>>>
```

これではあまりにもそっけない、という場合は、インスタンス属性として追加されたもののみの表示、などというコトもできます。

```
>>> class C:
...   def __dir__(self):
...     return list(self.__dict__)
...
>>> c = C()
>>> c.x, c.y, c.z = 100, 200, 300
>>> dir(c)
['x', 'y', 'z']
>>>
```

Python はそもそも、プログラミング構造上の情報の隠蔽には向いていませんので、本気でハッキングすれば属性を見ることなど簡単です。
'__dir__'属性を設定しなおす場合は、'dir'関数を使用するユーザが便利に使えるような設定にすることをお勧めします。

6-1-4-4-3 呼び出し可能オブジェクトのエミュレート

呼び出し可能オブジェクト(callable object)とは、関数のように識別子の後に括弧（タプル）をつけて、関数のように呼び出せるオブジェクトのことです。

■呼び出し可能メソッド
書式：__call__(self[,arg...])
動作：カスタムオブジェクトを関数のように呼び出す

例を見てみましょう。

```
>>> def func(x):
...   return x * 100
...
>>> func(10)
1000
>>> class Fnc:
...   def __call__(self, v):
...     return v * 100
...
>>> fnc = Fnc()
>>> fnc(10)
1000
>>>
```

書式については、メソッドとして第１引数に自身参照を持つ以外は、関数の書き方と全く同じです。
この自身参照を用いて、閉包(closure)をわりと簡単にエミュレートすることが出来ます。

閉包とは、関数自身が値を持つ関数です。

```
>>> def closure(x):#【本来、閉包はこう書く】
...   def in_closure(y):
...     nonlocal x
...     x += y
...     return x
...   return in_closure
...
>>> c = closure(10)
```

```
>>> c(5)
15
>>> c(6)
21
>>> c(7)
28
>>> class FakeClosure:    #【クラスで書くとこんな感じ】
...   def __init__(self, v):
...     self.__v = v
...   def __call__(self, w):
...     self.__v += w
...     return self.__v
...
>>> fc = FakeClosure(10)
>>> fc(5)
15
>>> fc(6)
21
>>> fc(7)
28
>>>
```

上の例は「累算器(accumulator)」と呼ばれるモノです。

閉包と異なり、内部変数に比較的アクセスしやすいのが問題ですが、逆にパラメータを変更する必要のある関数などを作るには、部分関数の合成などを使うより、いっそすっきりするかもしれません。

関数として使用する文には、関数と呼び出し可能オブジェクト（実はクラスもそうです）は同等ですが、一点だけ異なるので覚えておいてください。

関数以外の呼び出し可能オブジェクトをクラス属性として設定し、インスタンスから呼び出したとしても、メソッドにはならず、単なるクラス属性として扱われます。

```
>>> def f1(self):
...   return self.v
...
>>> class F2:
...   def __call__(self):
...     return self.v
...   v = 'F2'
...
>>> class C:
...   m1 = f1
...   m2 = F2()
...
>>> c = C()
>>> c.v = 'C'
>>> c.m1()   #【c を参照している】
'C'
>>> c.m2()   #【F2 を参照している】
'F2'
>>>
```

__call__の自身参照はあくまでもそのメソッドを設定したクラス及びそのインスタンスです。

6-1-4-4-4 コレクションのエミュレート

Python には、SNOBOL(AWK)や ICON のような、豊富なコレクション(collection)があります。

コレクション(collection = container type)はオブジェクトを様々な形で収納するコンテナ型オブジェクト。

この豊富なコレクションこそが、Python プログラミングの切り札と言っても過言ではないでしょう。

ちなみにコレクションの種類ではSmalltalkには及びませんが、SNOBOL-AWK-ICON系の操作書式を継承したPythonの圧倒的な使いやすさは、そのアドバンテージに匹敵するでしょう。

コレクションの詳細については、この後の「組み込みオブジェクト」の章で念入りに説明しますが、ここではコレクションの基本的な動作をエミュレートする（つまり、コンテナのように動作する……ように見える）特殊メソッドの説明をします。

この特殊メソッド群は、要するに'[]'という後置演算子の動作に対するものだと思ってください（前項の「呼び出し可能オブジェクト」が'()'後置演算子の動作に対する特殊メソッドであるのと同じです）

なお、この'[]'後置演算子が使用できるのは、順列(sequence シーケンス)と写像(mapping マッピング)の２種類で、集合(set)のように'[]'後置演算子に対応していないものもあります。

'[]'の中に指定されるオブジェクトは、順列では「インデックス(index)」、写像では「キー(key)」と呼ばれますが、インデックスはキーの特殊形と考えて、ここでは「キー」で統一します。

また、キーを使って指定する要素をアイテムとします。

キーを使った指定は、基本的に属性の指定と同じであり（本質的にも同じですが）同じ種類の操作が設定できます。

■キーによる参照の指定
書式：__getitem__(self, key) ⇒ object
動作：キーによって参照された時に実行

要するに'object[key]'として参照された時にどうなるかを決めます。

■キーによるアイテムの代入の指定
書式：__setitem__(self, key, value)
動作：キーによってアイテムが代入された時に実行

'object[key] = value'となった時の動作です。

■キーによるアイテムの削除の指定
書式：__delitem__(self, key)
動作：キーによってアイテムが削除された時に実行

'del object[key]'となった時の動作です。

```
>>> class FakeDic:  #【単なる辞書のラッパー】
...   def __init__(self, dic={}):
...     self.__dic = dic
...   def __getitem__(self, key):
...     return self.__dic[key]
...   def __setitem__(self, key, value):
...     self.__dic[key] = value
...   def __delitem__(self, key):
...     del self.__dic[key]
...   def __repr__(self):
...     return str(self.__dic)
...
>>> fd = FakeDic({'a':100})
>>> fd
{'a': 100}
>>> fd['a']
```

```
100
>>> fd['b']=200
>>> fd
{'a': 100, 'b': 200}
>>> fd['b']
200
>>> del fd['a']
>>> fd
{'b': 200}
>>>
```

以下は、属性とは異なった操作要求への対応の設定です。

■反復子への対応
書式：__iter__(self) ⇒ 反復子
動作：反復子を要求された時に実行

'iter'関数に渡された時や、for-in 構文で in の後に置かれた時に返す反復子を設定します。

■メンバーテストへの対応
書式：__contains__(self, item) ⇒ 真偽値
動作：'in'演算子でメンバーテストを行った時に実行

メンバーテストの結果への対応を設定します。

```
>>> class FakeDic:
...   def __init__(self, dic={}):
...     self.__dic = dic
...   def __getitem__(self, key):
...     return self.__dic[key]
...   def __setitem__(self, key, value):
...     self.__dic[key] = value
...   def __delitem__(self, key):
...     del self.__dic[key]
...   def __repr__(self):
...     return str(self.__dic)
...   def __iter__(self): #【辞書とは異なり、値の反復子を……】
...     return (x for x in self.__dic.values())
...   def __contains__(self, item):    #【こちらも値を基準に……】
...     if item in self.__dic.values():
...       return True
...     else:
...       return False
...
>>> fd = FakeDic({'a':100,'b':200,'c':300})
>>> for x in fd:
...   print(x)
...
100
300
200
>>> print(*fd)
100 300 200
>>> 100 in fd
True
>>> 'a' in fd
False
>>>
```

この章は、コレクションオブジェクトの章を読んだ後に、もう一度ごらんください。

6-1-4-4-5 終了処理構文への対応

Python の比較的新しい構文である「with」文は、ファイルハンドルのクローズなど「終わったら忘れずに自動的に実行したい」処理を自動化するための構文です。

このwith構文に対応するための特殊メソッドが、'__enter__'と'__exit__'の２種類です。

■with構文開始時処理
書式：__enter__(self) ⇒ self
動作：with文開始直後、この処理によって返される値がas節に記述された識別子に代入される

with構文では、渡されたクラスのインスタンスを初期化した後、'__enter__'メソッドを実行し、「そのメソッドの戻り値」を、as節以降の識別子に代入します。

メソッドが返す値は別段selfでなくてもかまいませんが、その場合はwith構文の中でas以下の識別子として使用されるオブジェクトは、selfつまり渡されたクラスのインスタンス以外ということになります（当然、'__enter__'メソッド内で'self'に対して行った操作は、反映されません）

■with構文終了時処理
書式：__exit__(self, exc_type, exc_value, traceback)
動作：with文終了直後に行う処理を記述

このメソッドは、基本的にはwith構文終了時に行う処理を記述するものですが、その処理を確実に行う為に、一時的に（あるいは永久に）例外を捕捉し保留する仕組みがあります。

'__exit__'メソッドの戻り値が指定されていなかったり、あるいはNoneを返す場合は、この終了処理が行われた後に上へ例外が投げ返されます（つまり例外は通過した後に、正常に発生します）

そうでない場合、例外はこのオブジェクトに捕捉されたままになります。

例外が発生しなかった場合は、self以降の三つの引数には全てNoneが入りますが、例外が発生した場合は、それぞれ

例外の型（クラス）　⇒exc_type
例外の値（内容）　⇒exc_value
トレースバックオブジェクト⇒traceback

が引き渡されます。

with文で、複数のクラスが指定された場合、たとえば……

```
with W1() as w1, W2() aw w2, W3() as w3:
  <statement>
```

とした場合は、処理される順序は以下のようになります。

１）W1.__enter__
２）W2.__enter__
３）W3.__enter__
４）<statement>
５）W3.__exit__
６）W2.__exit__
７）W1.__exit__

（スタックで処理されていると考えればいいかと思います）

```
>>> class Wv:
...   def __init__(self):
...     self.v = 'Wv:value'
```

```
...   def __enter__(self):
...     print('enter:Wv')
...     return self
...   def __exit__(self, exec_type, exec_value, traceback):
...     print('exit:Wv')
...     print('exec_type:', exec_type)
...     print('exec_value:', exec_value)
...     print('traceback:', traceback)
...     return self.v     #【None 以外の値を返す】
...
>>> class Wn:
...   def __init__(self):
...     self.v = 'Wn:value'
...   def __enter__(self):
...     print('enter:Wn')
...     return self
...   def __exit__(self, exec_type, exec_value, traceback):
...     print('exit:Wn')
...     print('exec_type:', exec_type)
...     print('exec_value:', exec_value)
...     print('traceback:', traceback)
...     print('exit')
...     return None #【None を返す】
...
>>> with Wv() as wv:      #【例外は捕捉されたまま】
...   print(wv.v)
...   print(1/0)    #【ZeroDivisionError】
...
enter:Wv
Wv:value
exit:Wv
exec_type: <class 'ZeroDivisionError'>
exec_value: int division or modulo by zero
traceback: <traceback object at 0x00BFD1C0>
>>> with Wn() as wn:      #【例外は上へ投げられる】
...   print(wn.v)
...   print(1/0)
...
enter:Wn
Wn:value
exit:Wn
exec_type: <class 'ZeroDivisionError'>
exec_value: int division or modulo by zero
traceback: <traceback object at 0x00BFD170>
exit
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: int division or modulo by zero
>>> with Wv() as wv, Wn() as wn:#【例外は Wn⇒Wv で捕捉】
...   print(wv.v, wn.v)
...   print(1/0)
...
enter:Wv
enter:Wn
Wv:value Wn:value
exit:Wn
exec_type: <class 'ZeroDivisionError'>
exec_value: int division or modulo by zero
traceback: <traceback object at 0x00BFD210>
exit
exit:Wv
exec_type: <class 'ZeroDivisionError'>
exec_value: int division or modulo by zero
traceback: <traceback object at 0x00BFD210>
```

```
>>> with Wn() as wn, Wv() as wv: #【例外は Wv で捕捉されたまま】
...   print(wn.v, wv.v)
...   print(1/0)
...
enter
enter
Wn:value Wv:value
exit:Wv
exec_type: <class 'ZeroDivisionError'>
exec_value: int division or modulo by zero
traceback: <traceback object at 0x00BFD210>
exit:Wn
exec_type: None
exec_value: None
traceback: None
exit
>>>
```

6-1-4-4-6 特定の関数に対する対応

特殊メソッドの中には、特定の関数へ渡された時の対応を記述したものがあります。

たとえば、コレクションの要素の数を返す len 関数は、特殊メソッドである'__len__'を呼び出しているだけのものです。

つまりこの手の関数は、特殊メソッドのラッパ(wrapper)となっているわけです。

オブジェクト指向言語を学んだ人にとっては、わざわざメソッドで記してあるものをラッパ関数で呼び出すのに疑問を持つかもしれませんが、これには理由があります。

Python のメソッドは Smalltalk のメソッドのように単独で使用することが出来ないため、複数のメッセージを渡すカスケード構文のような書き方が基本的に不可能です。

たとえば、以下のような書き方は、Python では出来ません。

```
>>> class C:
...   def a(self): return 0
...   def b(self): return 1
...   def c(self): return 2
...
>>> c = C()
>>> for x in (a,b,c):     #【メソッドを単独で書けない！】
...   print(c.x())
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

しかし、ラッパー関数を使うなら、似たようなコトが可能です。

```
>>> def wa(obj):
...   return obj.a()
...
>>> def wb(obj):
...   return obj.b()
...
>>> def wc(obj):
...   return obj.c()
...
>>> for x in (wa,wb,wc):  #【関数は 1st クラスオブジェクト】
...   print(x(c))
...
```

```
0
1
2
>>>
```

ラッパー関数に対応する特殊メソッドは、ラッパー関数名が"???"なら、特殊メソッドは、前後にダブルアンダースコアを置いた"__???__"となります。
特殊メソッドにそのまま対応している組み込み関数（及び組み込みオブジェクト）を下に列挙します。
「⇒」の指定がある場合は、戻り値はこの指定の型(あるいはそのサブクラス)のオブジェクトである必要があります（一部、そのオブジェクト'x'以外の引数に型の指定があるものもあります)

- abs(x)
- bool(x) ⇒ 真偽値(bool)
- dir(x) ⇒ リスト(list)
- divmod(x, y)
- float(x) ⇒ 浮動小数点数(float)
- format(x, format_spec[←文字列(str)]) ⇒ 文字列(str)
- getattr(x, key[←文字列(str)])
- hash(x) ⇒ 整数(int)
- int(x) ⇒ 整数(int)
- iter(x) ⇒ 反復子(iter 型やジェネレータなど)
- len(x) ⇒ 整数(int)
- next(x)
- pow(x, y)
- repr(x) ⇒ 文字列(str)
- reversed(x)
- str(x) ⇒ 文字列(str)

とはいえ、戻り値の指定が無くとも、関数の名前が連想するようなものとはかけ離れた戻り値は、オブジェクトの設計上好ましくないのは言うまでもありません。
'setattr'関数については、'__setattr__'の動作に準拠しますが、戻り値は必ずNoneになりますので、指定できません。
なお、'dict'と'set'（どちらも関数ではなくクラス）については、特殊属性の'__dict__'と'__set__'とは無関係ですので、注意してください。

'dict'は辞書クラス、'set'は集合クラスです。それに対して'__dict__'はインスタンス属性の辞書、'__set__'に至っては記述子のメソッドです。

6-1-5 モジュールオブジェクト

モジュールをオブジェクトとして扱うのは、Pythonのかなり珍しい特徴ではないかと思います。
その方法については「式と文」の章の「import 文とモジュール」で詳細に説明しましたが、ここではその意味をもう少し深く掘り下げて見ていきましょう。
モジュールには、組み込みモジュールと、その他のモジュールがありますが、'__builtins__'モジュール以外は全て'import'して使用します。
この『組み込みモジュール(built-in module)』は、Java などの言語をご存知の方には、なかなか理解できない性質ではないかと思います。
Java の'import'は、使用するモジュールを組み入れるために使用しますが、Python では、既に走っているシステムの中に組み込まれている機能('os'や'sys'など)を呼び出すためにも'import'文を使用します。
つまりPythonの'import'文は、本質的には名前管理の文法です（勿論、コンパイル／リンクされていないモジュールを呼び出した場合は、その場でコンパイル／リンクします)

組み込みモジュールは、'sys.builtin_module_names'の中で確認できます。

【若干、打ち出し結果の改行を編集してあります】

```
>>> import sys
>>> sys.builtin_module_names
('__main__', '_ast', '_bisect', '_codecs', '_codecs_cn', '_codecs_hk',
 '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections',
 '_csv', '_functools', '_heapq', '_io', '_json', '_locale', '_lsprof',
 '_md5', '_multibytecodec', '_pickle', '_random', '_sha1', '_sha256',
 '_sha512', '_sre', '_struct', '_subprocess', '_symtable', '_thread',
 '_warnings', '_weakref', 'array', 'atexit', 'audioop', 'binascii',
 'builtins', 'cmath', 'datetime', errno', 'gc', 'imp', 'itertools',
 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'operator', 'parser', 'signal',
 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')
>>>
```

これらのモジュールは、Python のシステムの中に予め組み込まれていますが、使用する際には、上記の'sys'のように'import'する必要があります。

導入(import)されたモジュールは、一個のオブジェクトとして取り扱います。

```
>>> import sys, os, operator, functools
>>> L = [sys, os, operator, functools] #【リストの要素になる】
>>> sys.platform
'win32'
>>> L[0].platform   #【一般のオブジェクトと同様に呼び出せる】
'win32'
>>>
```

モジュールとその中のオブジェクトの呼び出し方は、オブジェクトとその属性の呼び出し方と全く同じく'.'(ドット)を用いて連結します。

ただし Java のように、モジュールの中の特定オブジェクトをドットで繋いで呼び出す（"import module.object"のような書き方）ことはできません（from-import 構文を使用して下さい）

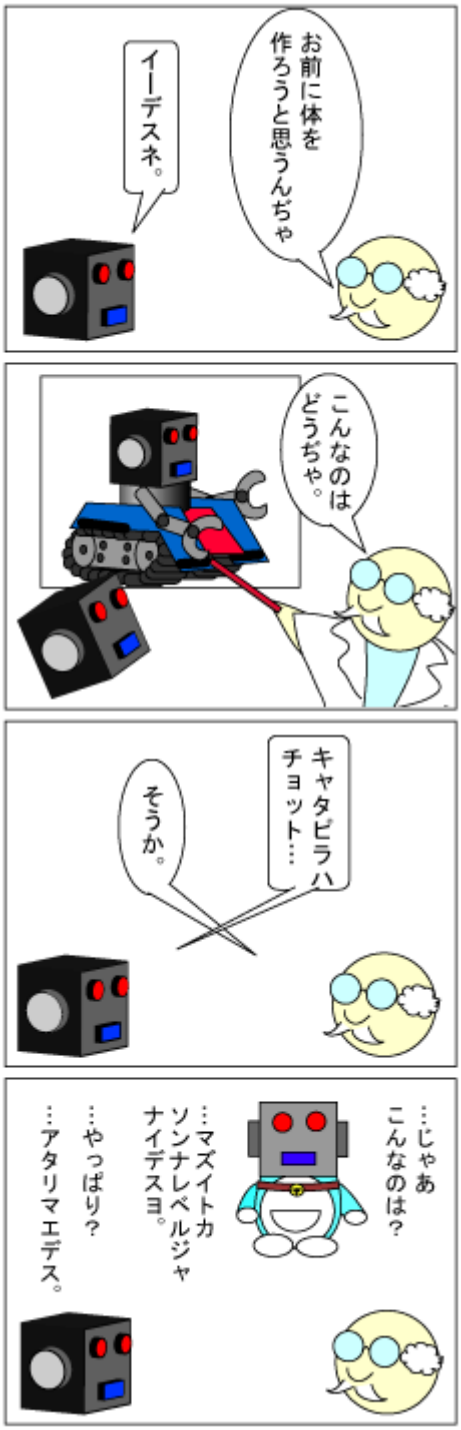
パッケージとモジュールに関しては、"import package.module"という呼び出し方が可能です。パッケージは複数のモジュールをまとめたものです。

(2010 年 8 月 29 日投稿受付)

よしおさんとロボ太

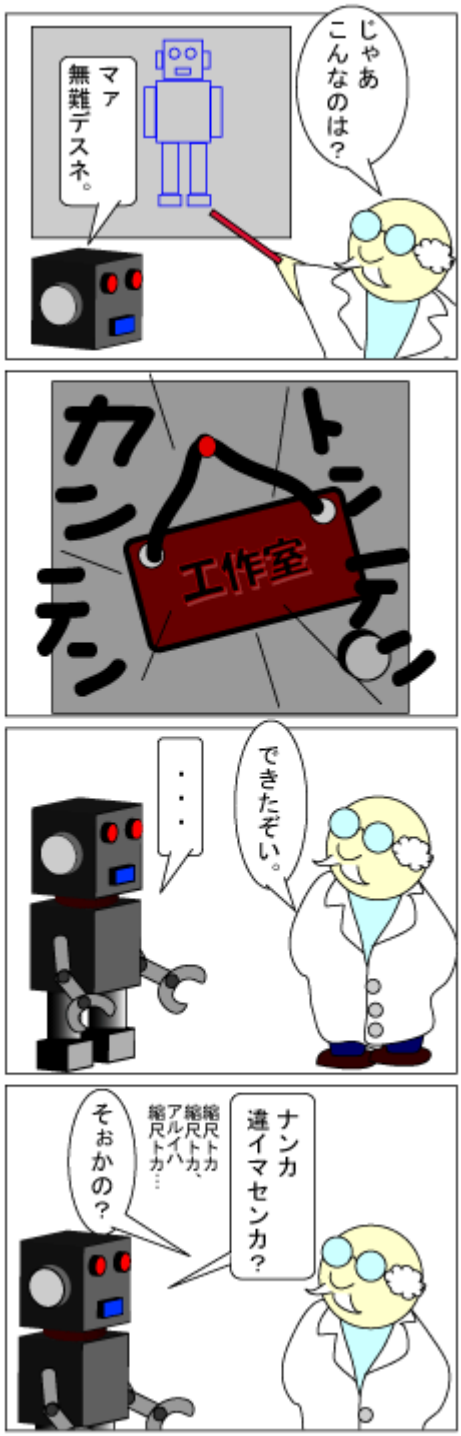
海鳥作

「ロボに体を」



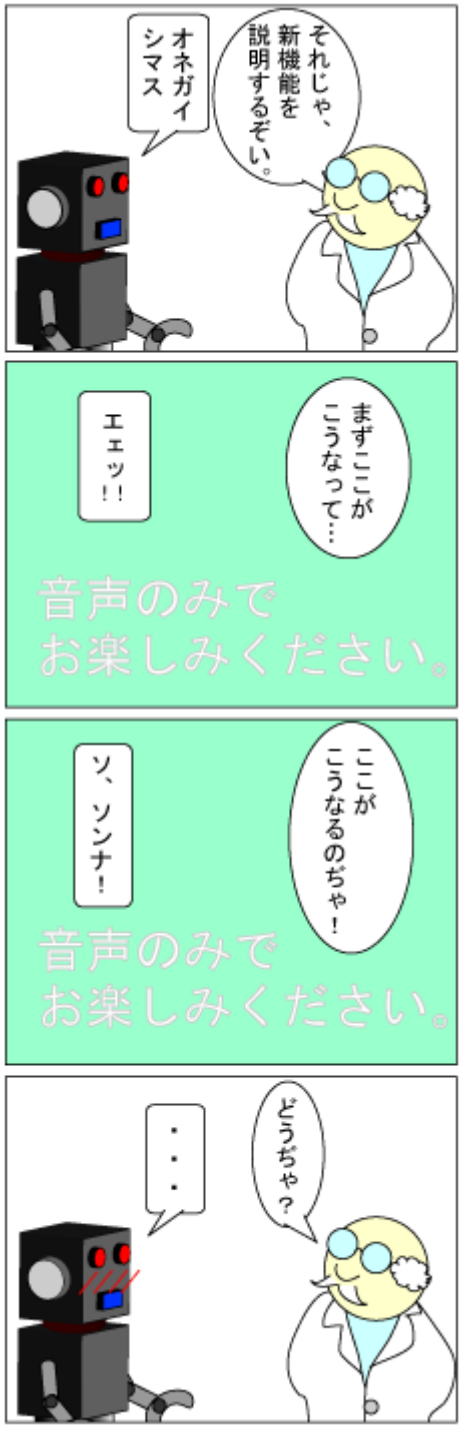
ついに手抜きをやめて体を描こうという作者の決意(挫折気味)。

「続・ロボに体を」



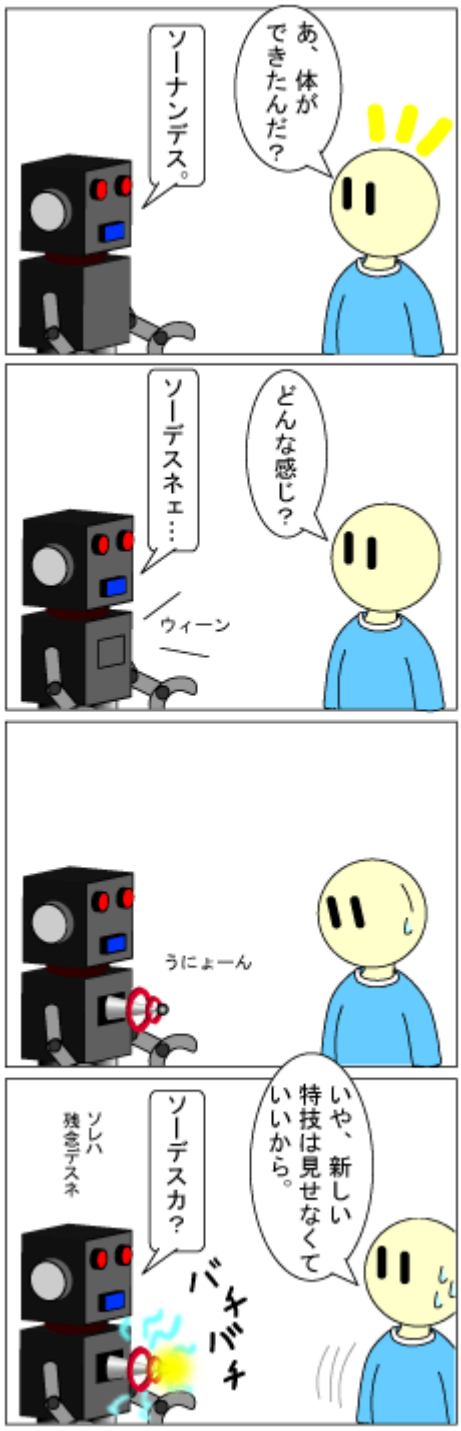
夏休みの工作レベル。

「秘・ロボに体を」



白衣と裸の二人があんなことやこんなことを(違)。

「激・ロボに体を」



どんな特技だったんだろう・・・。

（2010 年 9 月 5 日転載）

マルチバイト文字は length()==2 の夢を見るか？
または メイキング・オブ・みにくい awk の子

written by Ｙさ

1. この記事について

シフト JIS(2 バイト文字)を扱っている『昔作った awk スクリプトが動かないでござる』問題にぶつかった時の記録です。
もしかしたら何方かの何かの役に立つかもしれないので、書いて残しておこうと思った次第です。

2. 動作環境について

執筆時の動作確認は、
GNU Awk 3.1.8, mawk 1.3.3 MBCS R27
で、WinXP の DOS 窓で行なってます。

3. 問題点と『みにくい』解決方法について

マルチバイト文字を扱う awk 処理系プログラムの実装上の都合なのであろうかとは思いますが、

・gawk3.1.8 などは、length(),index()等が"文字数"を扱う
・mawk32(1.3.3 MBCS R27) などは、"バイト数"を扱う

といった違いがあります。

と言う訳で、mawk32 でシフト JIS(2 バイト文字)を文字列の位置決め打ちで substr()処理するスクリプトが、そのままのロジックでは gawk3.1.8 で動かなかったのです。
orz

また、よく分かっていないのですが環境変数 LANG とかでどうにかなるかもと思って少し実験してみました。gawk3.1.8 で UTF8 の以下のスクリプト

```
-----
BEGIN{ print length("∩") }
-----
```

を、[1]環境変数 LANG が定義されていない状態, [2]LANG=C で動かしたところ、

・[1] ⇒「1」と出力
・[2] ⇒「3」と出力

となりました。今さらですがいわゆる'漢字'が2バイトになるとは限らないことを認識しましたw

...んで、結局どうしたかというと、

1) マルチバイト文字列に後で消せる文字を繋げて、2バイト決め打ちで substr()処理できるように辻褄を合わせ、
2) 一連の処理の最後に、辻褄合わせのために繋げた文字を削除する

```
-----
fake=(length("∩")==2)?("")：("o");
putBit1[1]="∩" fake;
:
wk=putBit1[bit[i]] substr(wk, 1+length(putBit1[bit[i]]));
gsub(fake, "", wk);
-----
```

といった方法を思いつきましたとさw

4. サンプルプログラムについて

みにくいawkスクリプトのサンプルは、いわゆる基数変換器です。

```
>gawk -f ybin.awk[Enter]
~~~~~~~~~~~~~~~~~~~~~~~~
```

等として、実行してください。10進数で数字を入力すると2進数にして表示します。

ただし最大10ビット(10進数で1023)までしか正確に表示できません。
またなぜか表示が日本語(記号)のため、それなりの環境が必要です(;^^ゞ
ちなみにWinXPのDOS窓で、

・gawk3.1.8では環境変数 LANG を定義せずにUTF8のスクリプトで
・mawk32(1.3.3 MBCS R27)ではシフトJISのスクリプトで

実行確認しました。

オリジナル初出は、awk版として1999/7にFGALTSにアップしたものです。
当時のアーカイブ(?)が jscripter さんの手によって「講座」としてまとめられています。

http://homepage1.nifty.com/kazuf/awkdepg/main.htm

現在の原稿の形になったのは2010/9なので、もう10年以上も昔の話なんですね...
d(^^;

5. その他

本記事のサンプルプログラムはフリーソフトです。
著作権は作者であるＹさにあります。
ただし転載、再配布、改造、消去は自由です。
(できましたら素晴らしい改造を加えた後にTSNetへ投稿してくださいませ)
また、このソフトを使用した事による損害が発生したとしても、損害に対しては一切の責任を負いかねます。

[ybin.awk]

```
# 10進数→10bitバイナリ変換器　written by Ｙさ

BEGIN{
  #ビット表示用データ
  fake=(length("∩")==2)?(""):("o");
  putBit1[0]="　";       putBit2[0]="__" fake;  putBit3[0]="∪" fake;
  putBit1[1]="∩" fake;  putBit2[1]="||";       putBit3[1]="　";
}
{
  for(i=1; i<=4; ++i) ptbl[i]="";
  setYbin($0+0);
  for(i=1; i<=4; ++i) printf("%s¥n", ptbl[i]);
}

#バイナリ変換器本体
function setYbin(num,  i){
  if(num>31) setL(int(num/32)%32);
  setR(num%32);
}

#下位５ビットセット
function setR(num,  wkTbl,i,pos,addStr){
  setBit(num);
  wkTbl[1]=wkTbl[2]="";
  for(i=5; i>=2; --i)  wkTbl[1]=wkTbl[1] putBit1[bit[i]];
```

```
  wkTbl[1]=wkTbl[1] ″  ″;
  for(i=5; i>=3; --i)  wkTbl[2]=wkTbl[2] putBit2[bit[i]];
  wkTbl[2]=wkTbl[2] ((num!=int(num/4)*4 || bit[2])?(putBit2[bit[2]]):(″_ ″));
  wkTbl[2]=wkTbl[2] ″  ″;
  wkTbl[3]=(bit[5])?(″( ″):(fake ″∪″);
  for(i=4; i>=2; --i){
    pos=(5-i)*2+1;  addStr=putBit3[bit[i]];
    if(i==4 && (num==16 || num==18)){
      pos=2;  addStr=fake ″∪ ″;
    }
    if(i==3 && (num!=0 && num!=2 && (num%8==0 || num==int(num/8)*8+2))){
      pos=4;  addStr=fake ″∪ ″;
    }
    if(i==2 && ((num+1)%8==0 && bit[2]==1)){
      pos=6;  addStr=fake ″／_″;
    }
    if(i==2 && bit[3]==0 && bit[2]==1 && bit[1]==1){
      addStr=″ _″;
    }
    wkTbl[3]=appendR(wkTbl[3], addStr, pos);
  }
  if(num==0 || num==2) wkTbl[3]=appendR(wkTbl[3], fake ″⊂) ″,  7);
  else if(bit[1]==0)   wkTbl[3]=appendR(wkTbl[3], fake ″⊂)  ″, 6);
  else                 wkTbl[3]=appendR(wkTbl[3], fake ″⊃″,    9);

  for(i=1; i<=3; ++i){ gsub(fake, ″″, wkTbl[i]);  ptbl[i]=ptbl[i] wkTbl[i]; }
  #                  0....5....
  ptbl[4] = ptbl[4] ″ |    |   ″;
}
function appendR(src,tgt,pos){ return substr(src, 1, pos-1) tgt; }

#上位５ビットセット
function setL(num,  wkTbl,i,pos,addStr){
  setBit(num);
  wkTbl[1]=wkTbl[2]=″  ″;
  for(i=4; i>=1; --i)  wkTbl[1]=wkTbl[1] putBit1[bit[i]];
  wkTbl[2]=wkTbl[2] ((0<=num && num<8)?(″ _″):(putBit2[bit[4]]));
  for(i=3; i>=1; --i)  wkTbl[2]=wkTbl[2] putBit2[bit[i]];
  wkTbl[3]=(num==0 || num==8 || num>15)?(″  ″):(″   ″);
  for(i=4; i>=1; --i){
    addStr=putBit3[bit[i]];
    if(i==4 && num>=28)
      addStr=″_＼″ fake;
```

```
    if(i==4 && bit[3]==0 && bit[4]==1 && bit[5]==1)
      addStr="_ ";
    if(i==3 && (num>=28 || num==4 || num==5 || num==12 || num==13))
      addStr=" ";
    if(i==2 && (num==5 || num==13))
      addStr=fake "∪ ";
    if(i==2 && (num==2 || num==6 || num==10 || num==14))
      addStr=" ";
    if(i==1)
      addStr=(num<16 && bit[1])?("⊃"):((bit[1])?("⊂")):("∪" fake));
    wkTbl[3]=wkTbl[3] addStr;
  }
  if(num==0 || num==8) addStr=" (⊃" fake;
  else if(bit[5]==0)   addStr="  (⊃" fake;
  else                 addStr="⊂" fake;
  wkTbl[3]=addStr substr(wkTbl[3], 1+length(addStr));

  for(i=1; i<=3; ++i){ gsub(fake, "", wkTbl[i]);  ptbl[i]=wkTbl[i] "  "; }
  #          0....5....0.
  ptbl[4] = "   |    |   ";
}

#ビットカウント
function setBit(num,  cnt,i){
  do{ bit[++cnt]=num%2; }while((num=int(num/2))>0);
  while(cnt<=5) bit[++cnt]=0;
}
```

(2010 年 9 月 2 日投稿受付)

編集後記

jscripter

TSNET スクリプト通信第 10 号が出る。僕自身のまとまった記事が書けなかったのは残念だが、編集後記に Twitter 関連の開発に役に立ちそうな、もう少し参考になるメモを残しておこう。

実際のところ、3 ヵ月毎に何か記事をまとめるのはなかなか難しい。特に大量の文章を続けてまとめられている機械伯爵氏は大変だろうと思う。僕自身は 2-3 日集中すればなんとかなると高を括っていたのが失敗のもと。少しずつ時間を掛けてスクリプトには触ってはいたのだが、Twitter の API や Net::Twitter の変化が把握できていなかった。普段十分に使っていないものは危険だということ。問題に気づいたり、関連情報に触れるチャンスが少ないからである。

たまたま、今日(9/5)、twitter の perl_api から次の記事を知った。

On the demise of basic authentication. - Twitter Development Talk | Google グループ

仕事で使っている人には当然のことでも、アマチュアに情報が伝わるのは、問題を自分の身で不具合を知ってからになることが多い。

Create cool applications! | dev.twitter.com

に、twitter の技術リソースが置いてある。TSNET スクリプト通信 2.2 の「Zed 入門 II - Twitter on Zed」の記事で紹介したリソースは、次の Wiki である。Wiki から、上記のサイトに移行中なのである。

Twitter API Wiki / FrontPage

日頃からチェックするなら、Twitter を使うとよいだろうと今更ながら思う。

@TwitterAPI
http://twitter.com/twitterapi

perl_api に投稿された記事の Google グループにも参加することにした。

Twitter Development Talk | Google グループ
http://groups.google.com/group/twitter-development-talk

さて、Twitter ネタはこれぐらいにして、新しい話題としては、「国立国会図書館サーチ」の公開がある。8 月 17 日に「外部提供インタフェース仕様書」の暫定版が公開されている。これをネタにスクリプトを書くことも可能だろう。スクリプトが活躍できる場は拡

がるばかりだ。是非ともみなさんも記事をものにして投稿していただきたい。情報へのリンクを残しておこう。

国立国会図書館サーチ（NDL Search）
メタデータについて《 国立国会図書館サーチ（開発版）について
API について《 国立国会図書館サーチ（開発版）について

（2010 年 9 月 5 日投稿受付）

ＴＳＮＥＴスクリプト通信

ISSN 1884-2798　出版地: 広島市

2010 年 9 月 7 日　　3.1.001 版

投稿規程

TSNETWiki ： 「投稿規程」のページを参照のこと

編集委員会(投稿順)

海鳥　kaityo256[at]nifty[dot]ne[dot]jp
機械伯爵　kikwai[at]livedoor[dot]com
Ｙさ　saw[at]mf-nokuchi2pho[dot]ne[dot]jp
jscripter　jscripter9[at]gmail[dot]com

著作権

1. 各記事及びその他の著作物については、著作者が著作権を保持します。
2. 「TSNET スクリプト通信」の二次著作権は各記事及びその他の著作物の著作者より構成される編集委員会が保持します。

使用許諾・配布条件

1. 編集委員会は「TSNET スクリプト通信 3.2.xxx 版」を、ファイル名が「tsc_3.2.xxx.pdf」の PDF ファイルとして無償で配布します。また、ファイル名、ファイル内容を一切改変しない状態での電子的再配布および印刷による再配布を無償で許諾します。
2. 関連するスクリプトファイルなどのプログラムについては、使用および再配布を無償で許諾しますが、改変後の再配布についてはオリジナルの著作権を併記することを条件に無償で許諾します。
3. 記事およびスクリプトファイルなどのプログラムに著作者の使用許諾・配布条件の記載がある場合は、著作権の項および上記 2 項に優先するものとします。

免責事項

「TSNET スクリプト通信」の内容および同時に配布されるスクリプトなどの使用は、すべて使用者の自己責任によるものとし、使用によって生ずる一切の結果等について、編集委員会および著作者は責任を負いません。

編集ソフトウェア

OpenOffice.org 3.2.1 Writer
ConcatPDF for Windows 1.1.11.0

発行所

一次配布所: TSNET スクリプト通信刊行リスト
http://text.world.coocan.jp/TSNET/?TSNET%E3%82%B9%E3%82%AF%E3%83%AA%E3%83%97%E3%83%88%E9%80%9A%E4%BF%A1%E5%88%8A%E8%A1%8C%E3%83%AA%E3%82%B9%E3%83%88

TSNET スクリプト通信 第 3 巻第 2 号 (通算第 10 号)
発行 : TSC 編集委員会 発行日 : 2010 年 9 月 7 日
ISSN: 1884-2798 出版地 : 広島市 創刊 : 2008 年 5 月 7 日