TSNET スクリプト通信

創刊二周年記念号

TSC 編集委員会

May 2010, Vol. 3 Issue 1　ISSN 1884-2798

目次

表示デザイン : jscripter

古い自著の王羲之の蘭亭序を撮影し、白黒反転し、着色したものを背景に使用している。テキスト処理を考えることは言語を考えることに通ずる。旧きを訪ねて新しきを知る。最先端のプログラミングも旧きを訪ねてこそ新しいものが生まれてくるのではという思いを込めた。

巻頭言

jscripter

TSNETスクリプト通信は第9号、二周年を迎えます。2008年5月に創刊の辞を書いて以来、年4号刊行の三年目に入りました。著者のみなさんが多忙の中で生み出したスクリプトが輝いています。

コンピューティングの世界はiPadの発売やGoogle TVの発表などで盛り上がっています。手のひらで実用的なコンピュータが軽やかに動作し、インターネットと交信しています。10年前と比較するとほとんど夢のようなことが実現しているのです。急速にコンピューティング環境は進化しており、ここはスクリプティング言語の出番ではないかと我田引水したいところです。

問題は「TSNETスクリプト通信」の認知度でしょう。記事をPDFに閉じ込めるよりはWebに展開した方がよいのかもしれません。ePub化してWebストアから配信することなども検討してみたいですね。Webストアを作ってみてもよい。

みなさんお忙しいので、今号は無理をせずに刊行予定日までに集まった記事を編集することにしました。

機械さんの「Pythonの文法」は第2回も60ページに及ぶ大作です。第2回は「5. 式と文」に入ります。

ムムリクさんこと、きしだあつしさんの「RMagickでカレンダー壁紙を作る」はRubyでImageMagickを取り扱うにはどうするのかを教えてくれます。

海鳥さんの「よしおさんとロボ太」は「この親にして」を一つだけ転載します。今回は少し出し惜しんでみようと思いました。ドクターYが登場します。

私こと、jscripterは「日曜工作的プログラミング - ラジオエアチェック管理サーバーを作る」です。デスクトップのデータリソースに自在にアクセスする方法を実現します。

Yささんは投稿締切まで頑張られ新作ゲームを投稿していただいた。そのタイトルもカッコよく「ダイヤル(Aから)Mを廻せ！」です。

Have fun.

(2010年5月30日投稿受付;2010年5月31日更新)

Python の文法

TSNET スクリプト通信版　草稿　第２回

5. 式と文

5-1 Python における式と文

5-1-1 式と文の違い

Python プログラムは文(statement)で構成されています。
文の構成単位のうち、一番簡単なものが式(expression)です。
式はそれだけで式文(expression statement)を構成できます。
式は式の中に記述できますので、基本的にいくつ組み合わせても一つの式に合成可能です。
式は値を持ちますので、全体で一つのオブジェクトと見做すことができます（lambda を使って関数リテラルにすれば、実際全部まとめてオブジェクトに出来ます)
式でない文を非式文(non expression statement)といいます。
非式文には、前章までに説明した定義文、代入文、宣言文の他に、各種制御文があります。
注意すべきは、Python に於いては代入文が式ではない、ということです。
これによって、Python の式は大幅に制限されることになります。
非式文は、基本的に式の中に組み込んで式にすることは出来ないからです。

※実際には、ちょっとした裏技を使えば出来ないこともないのですが、コードが極端に汚くなる上、様々な制限があるのでお勧めできません。できません……が、第二部で説明する予定です。

非式文が存在することを以て、Python が使いにくいと評する声も聞きます。
特に関数型言語系のユーザーの方は、なまじ Python が様々な機能を有しているが故に、これを大きな欠点と見做すようです。
しかし、関数型言語が優秀なのは常識だとしても、関数型言語が「難しい」ということも常識です。
Python は、言語に熟達したウィザードな方々ばかりを対象にした言語ではありません。
寧ろ、プログラミングを始めたばかり、あるいはプログラミング言語を「知っている」程度の人に優しい言語です。
しっかりとした文構成の言語ですから、テクニックを生かす機会は難しいのですが、自他ともに理解しやすいコードが書けるようになっています。

でも、ウィザードな方々もご安心ください。Python のダークサイドは、恐ろしい魑魅魍魎の棲まう魔界なのです。それも第二部にて説明予定

5-1-2 式の範囲

式という言葉を正式に定義するのなら「評価した結果、値をもつもの」ということになります。
つまり式そのものが「式オブジェクト」とでも言えるようなものでもあり、オブジェクトの記述できるところには（実行結果はともあれ文法上は）全て式を書くことができます。
式は即座に評価されますが、関数リテラル(λ 式)としてパックすることで評価を遅延させることが可能です。
逆にいうならば、全てのオブジェクト（リテラルで記述されるものを含む）は、単体で式であるといえます。
式を要求する箇所には（同じく実行結果はともあれ文法上は）全てオブジェクトを置くことが可能です。
式は再帰定義的に式（式の式は式になる）であり、再帰の出発点はオブジェクトということになるでしょう。
ただし式の結果となる値は、必ずしも意味のあるものだとは限りません。
オブジェクトのメソッドなどの中には、副作用を目的としたものも多数あり、そのようなメソッドを呼び出した結果戻る'None'オブジェクトには、事実上意味がありません。
また式の実行順序については、明示的なものもありますが、リストやタプルの中身のように、基本的に実行順序が決まっていないものもあります（シーケンスの評価が先頭から行われると決め付けて

プログラムを書いた場合、その動作が正常に行われるかどうかは保証の限りではありません）
実際、非式文を排して式文だけでもかなりのプログラムを書くことが「可能」ですが、無理に式文にまとめると、かえって非効率で見難いコードが出来てしまうことが多いので、便利で簡潔に記述できる利点だけを適度に利用することをお薦めします。

5-1-3 文という単位

文は、Python の実行単位です。
文は、改行及びセミコロン(';')によって区切られます。
文は、特に制御構文などを用いない限り、スクリプトの先頭から順に実行します（順次構造）
式文は式のみで構成された文ですが、式文自体も式ですので、文の実行順序に従います（式の中の評価の順序は、必ずしも表記順序に従うとは限りません）
非式文には、代入文、定義文、宣言文、そして各種の制御構文があります。
定義文には、既に述べた通り関数定義とクラス定義があります。
制御構文や宣言文は（スコープについての宣言を行う一部宣言文を除き）全ての関数定義文、クラス定義文の内部で実行することができます（例えば導入宣言である'import'文は、関数やクラスの定義内で使用可能です）
構文ブロックには、特定の開始／終了記号（begin-end や{}など）を使用しません。
構文ブロックの区切りは、デリミタ(':')に続ける場合は行末まで、改行する場合はインデントを以って行います。
なお、構文によっては節について改行を要求するものもあります（'if'-'else'文'、'try'-'except'文など）

5-2 式

5-2-1 演算子を使った式

演算子を使った式は、式の基本形であるとも言えます（オブジェクト１個を以て式、という言い方も出来ますが、この章ではあつかいません）
演算子の定義を広く取ると、オブジェクト（リテラルと変数）と演算子以外で構成される式は皆無になりますが、ここでは演算子の定義を狭く取って、通常の単項演算子と二項演算子だけを取り上げます。
なお演算子の動作は、オブジェクトによって変わりますので、組み込みオブジェクトに対する動作だけを扱います。

Python では、演算子の多重定義(overload)によって、オブジェクトの演算子に対する振る舞いを定義できます。詳しくはオブジェクトの項で説明します。

5-2-1-1 四則演算

加減乗除の４つの演算を行う、ごく普通の計算記号です。
皆さんも小学校の頃から（好きか嫌いかはともかく）おなじみのものです。
違いと言えば、『＋』と『－』はそのまま'+'と'-'ですが、『×』の代わり'*'、『÷』の代わりに'/'を使うくらいでしょうか。
演算子の優先順位も、加減算に対して乗除算が優先されるところも同じです。

言語によっては、演算記号の優先順位が無いものもありますが、Python は常識準拠です。Python が教育などで使用されやすいのも、こういった細かい配慮があちこちにあるからだということも理由の一つでしょう。

```
>>> 1 + 2 * 3              #【9にはならない】
7
>>>
```

整数、実数、虚数入り乱れでも OK です（虚数は'i'でなく'j'）

```
>>> 3 + 8.5 - 5.5 * 3j
(11.5-16.5j)
>>>
```

除算演算子'/'については注意してください。

他の演算子は、整数同士の演算では必ず整数の値を返しますが、除算演算子は実数(浮動小数点数)を返します。

整数同士の除算で、小数点以下を切り捨てた整数値を求めたい場合は、フロー除算演算子(floor division)'//'を代わりに使用してください。

なお、余りを求めたい場合は、剰余演算子'%'を使用します。

```
>>> 13 / 5
2.6
>>> 13 // 5
2
>>> 13 % 5
3
>>>
```

5-2-1-2 結合演算、反復演算

数値オブジェクトに対して加算演算子として働く'+'は、シーケンスに対しては、結合演算子として働きます。

```
>>> "123" + '456'
'123456'
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + [4, 5, 6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
>>>
>>> 1 + '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

文字列と文字列、タプルとタプル、リストとリストは'+'で結合できます。

文字列はシングルクォートでもダブルクォートでも同じ文字列ですが、タプルとリストは別物ですから直接結合はできません。

また、数値と文字列もしっかり区別されます（Pythonは「型が緩やかな(type lose)言語」ではありません）

type loseとは、例えば数値オブジェクトを文字列としても使えるなど、オブジェクトの型の区別が緩やかな言語。

数値オブジェクトに対して四則演算子として働く'*'は、シーケンスに対しては、反復演算子として働きます。

```
>>> '123' * 4
'123123123123'
>>> (1, 2) * 3
(1, 2, 1, 2, 1, 2)
>>> [5, 6] * 2
[5, 6, 5, 6]
>>>
```

シーケンスに対して整数を掛けると、その回数だけ繰り返されます。

反復演算子に使用できる数値は整数だけで、実数オブジェクトや複素数オブジェクトは使えません。なお、マイナスの値（ゼロ以下の値）を使うと、空のシーケンスが返ります

5-2-1-3 正負符号と累乗演算

算数ではなく数学の世界に入ると、正負の符号（負の値が登場する）や平方根などが登場します。

四則演算の項で思いっきり虚数をやってしまったことは忘れてください。

正負の符号'+'と'-'は、単項演算子として働きます。

オブジェクトの前の単項演算子は、当然オブジェクトに近い方から結合していきます。

※'+'と'-'では、どの順序で処理されているのか結果からは一切判りませんが、後述の反転の単項演算子'~'などを混ぜるとわかります）

```
>>> 5+3
8
>>> 5+-3
2
>>> 5+--3
8
>>> 5+---+3
8
>>> 5+-*-+3  #【ヘンな所に混ぜると】
  File "<stdin>", line 1
    5+-*-+3
       ^
SyntaxError: invalid syntax
>>> 5*+--+3  #【コレはOK】
15
>>>
```

なお、二項演算子と単項演算子を続けて書くと見苦しいので、通常はこのように書きます。

```
>>> 5 * -3
-15
>>>
```

累乗演算子'**'は、基本的には『～乗』という計算に使います（累乗は冪乗(べきじょう)とも言います)

他のプログラミング言語では、累乗に'^'を使うものが多いので気をつけてください。

累乗演算子の気をつけるべきところは、右結合（右から順に結合する）である、ということです。

```
>>> 3 ** 2          #【普通に３の２乗】
9
>>> 3 ** 1 ** 2     #【これは３の、１の２乗、乗】
3
>>> 3 ** (1 ** 2)   #【確認してみる】】
3
>>> (3 ** 1) ** 2   #【こうではない】
9
>>>
```

同じ（あるいは同じ優先度）の二項演算子は通常左結合なのですが、この累乗演算子に限っては右結合です（単項演算子も）。

さて、ここまでで話は終わり、ではありません。

少しレベルの高い数学を習った方なら、「負の整数乗」は分数になることをご存知ですね。

つまり「２の－２乗」は「１割る、２の２乗」で「４分の１」。

さらに高度な数学を学んだ方は、平方根は「２分の１乗」である、ということをご存知でしょう。

三乗根(立方根)は「３分の１乗」。

つまり、累乗に関しては、負の数も実数や複素数も適用可能です。

虚数乗や複素数乗を、使う機会があるかどうかは別ですが

```
>>> 2 ** 2   #【通常の２の２乗】
4
>>> 2 ** -2  #【２の－２乗】
0.25
>>> 2 ** (1/2)      #【２の、２分の１乗】
1.4142135623730951
>>> a = 2 ** (1/2)
>>> a * a    #【確認】
2.0000000000000004
>>> print(a * a)    #【計算誤差修正】
2.0
>>> b = 2 ** (1/3)
>>> b * b * b
2.0
>>> b
1.2599210498948732
>>> c = 2 ** 2j            #【虚数乗の意味はわかんないけど……】
>>> c
(0.18345697474330172+0.9830277404112437j)
>>> c ** 2j         #【指数計算ルールからいけば、こうすれば……】
(0.06250000000000001+0j)
>>> print(c ** 2j)  #【やっぱり】
(0.0625+0j)
>>> 1/16
0.0625              #【コレですよね】
>>>
```

5-2-1-4 論理演算

コンピュータの演算で、数値の単純計算以上によく行われるのが、論理演算です。

後述のビット演算も論理演算の一種ですが、ここで扱うのは真偽値が関係する本来の論理演算です。

論理演算子は、論理積演算子'and'、論理和演算子'or'、論理否定演算子'not'の三つです。

'and'と'or'は二項演算子、'not'は単項演算子です。

優先順位は'not' ＞ 'and' ＞ 'or'の順になるので、少し気をつけてください。

論理演算に使用できるのも、論理演算の結果得られるのも真偽値だけの筈ですが、論理演算だけは比較的緩やかなルールに則って行われます。

結果から言えば、全てのオブジェクトが真か偽かのどちらかの値を持ちます。

○『偽(False)』の値を持つオブジェクト

真偽値オブジェクト　False
特殊オブジェクト　None
数値オブジェクトの０相当
整数　0
実数　0.0
虚数　0j
複素数　0+0j
空コレクション
空文字列　''
空タプル　()
空リスト　[]
空辞書　{}
空集合　set()
空凍結集合　frozenset()
空バイト列　b''
空バイト配列　bytearry(b'')
特殊メソッド'__bool__'の戻り値をFalseにセットしたオブジェクト

○『真(True)』の値を持つオブジェクト

偽の値を持つオブジェクト以外全部

それでは、論理演算子の動作について説明します。

・論理積演算子'and'
書式：A and B
動作：Aが真ならBを、Aが偽ならAを返す

論理積とは本来、「両方が真の時だけ真で、あとは偽」という演算です
論理積であるなら真偽値(True, False)を返すのが当然と思われるかもしれませんが、理屈は合っています。
つまり……

1. 全てのオブジェクトに真／偽がある
2. Aが真ならばBが結果、つまりBが真なら真に、Bが偽なら偽になる
3. Aが偽ならBを判定せずとも（本当に判定しません）偽なので、Aを返す

ということです
このため、'and'と次に説明する'or'は『短絡判定演算子(short-circuit operator)』と呼ばれます。

Lispなどで、選択文を書く代わりに論理演算を使われた歴史的な使用方法が残ったものと思われます。ただし、現在では後に説明する条件付評価式で同等のものを記述できるので、なるべくこちらを使うべきでしょう。なお、論理演算子を用いて選択文を書く方法は、後に紹介します。

・論理和演算子'or'
書式：A or B
動作：Aが真ならAを、Aが偽ならBを返す

論理和とは本来、「両方が偽の時だけ偽で、あとは真」という演算です。
動作の理屈は論理積と全く同じです。

・論理否定演算子'not'
書式：not A
動作：Aが真ならFalseを、偽ならTrueを返す

論理否定とは、その真偽値と「反対の値」を意味します。
ですから、論理演算で判定される値と逆の値を真偽値で返します。
なせこれだけが真偽値か……考えて見ればわかりますが、真となる値と偽となる値は、真偽値以外は全然対称ではないからです。
結果として、これだけが一番論理演算っぽくなります。

```
>>> 1 and 2
2
>>> 2 and 1
1
>>> 1 or 2
1
>>> 2 or 1
2
>>> 0 and 1
0
>>> () and 1        #【空タプルも偽】
()
>>> 0 or ()
()
>>>
>>> 1 and 2 or 3    #【左優先？】
2
>>> 2 or 0 and 3    #【いいえ、'and'が優先】
2
>>> (2 or 0) and 3  #【左優先なら、こう】
3
>>> not 1
False
>>> not 1 and 3
False
>>> not 0 and 3            #【単項演算子は大体優先順位が高い】
3
>>>
```

【補足】論理記号をご存知の方のために、論理記号と Python の論理演算子(真偽)及びビット論理演算子（あるいは文や関数）の対応を下に揚げておきます

記号	真偽	ビット	その他	意味
∧	and	&		論理積、連言、合接（および
∨	or	|		論理和、選言、離接（または）
¬	not	~		否定（ではない）
⇒			if 文	条件（もし～ならば）
∀			all 関数	全称量化（全ての）
∃			any 関数	存在量化（ある）
		^		（排他的論理和）

5-2-1-5 比較演算

比較演算とは、比較可能なオブジェクト同士の関係についての結果を、真偽値(True, False)で返す演算です。

比較には同値性、同一性、大小関係の三つがあります。

同一性は、オブジェクト同士が全く同じ実体を参照している場合にのみ真とします。

同値性は、オブジェクトの持つ値が同じなら真とします。

大小比較は、同値性と同様、オブジェクトの持つ値の大小関係を比較します。

メンバーテストは、あるオブジェクトがコレクションの要素かどうかを判定します。

5-2-1-5-1 同一性

同一性を調べるには、'is'演算子と'is not'演算子を使用します。

・比較演算子'is'

書式： A is B

動作：AとBが同じ実体を指していれば真、違う実体を指していれば偽を返す

数値以外でこの値が成立するのは、同じものを代入されている場合に限ります。

```
>>> class C: pass
...
>>> c = C()
>>> d = C()
>>> cc = c
>>> c is cc
True
>>> c is d
False
>>>
```

数値は「整数」「実数」といった種類まで同じなら同一実体を参照しているとみなされますが、違っていてはダメです。

コレクションに至っては、同一のオブジェクトを代入したものでない限りダメです。

```
>>> 10 is 10        #【同じ値は○】
True
>>> 10 is 10.0      #【オブジェクトの種類が違う】
False
>>> 10 is 5+5               #【結果が全く同じなら○】
True
>>> 10 is 5+5.0     #【結果のオブジェクトの種類が違う】
False
>>> 10 is 5+0j      #【複素数もダメ】
False
>>> [] is []        #【別のオブジェクトとみなされる】
False
>>> a = b = []      #【同じオブジェクトを代入すれば○】
>>> a is b
True
>>>
```

実体と参照について不明な場合は、「4-2-1 基本形」に少し詳しく書いてあるので、参考にしてください。

・比較演算子'is not'
書式：A is not B
動作：AとBが同じ実体を指していれば偽、違う実体を指していれば真を返す

'is not'は、単に'is'の逆です。('is not'で一つです。間違っても'is'と'not'ではありません！)

```
>>> 5 is 5.0
False
>>> 5 is not 5.0
True
>>> 5 is (not 5.0) #【こうではない！】
False
>>> False is (not 5.0)
True
>>>
```

5-2-1-5-2 同値

同値テストは、'=='演算子と'!='演算子を用いて行います。

・比較演算子'=='
書式：A == B
動作：AとBが同値なら真を、違う値なら偽を返す
・比較演算子'!='
書式：A != B
動作：AとBが同値なら偽を、違う値なら真を返す

同値テストは、数値においては、通常に同じ値と思われるものは同じ値です。

数値に於いては、大体想像通りに動きます。

```
>>> 5 == 5.0          #【整数と実数】
True
>>> 5 == 5+0j               #【整数と複素数】
True
>>> 5.0 == 5+0j       #【実数と複素数】
True
>>>
```

文字列も大体予想通りだと思います。

```
>>> 'abc' == 'abc'
True
>>> 'abc' == 'ABC'  #【当然、大文字小文字は区別】
False
>>> 'abc' == 'ａｂｃ'        #【半角／全角も別物ですよ】
False
>>>
```

タプルやリストなどは、一つ一つのアイテムを順に比較して、全てが同値なら同値と見ます。

```
>>> [1,2,3] == [1,2,3]
True
>>> [1,2.0,3+0j] == [1.0,2+0j,3] #【全てが同値ならば同値】
True
>>> (1,2,3)==(1,2,3)
True
>>> (1,2,3)==[1,2,3]              #【ただし、タプルとリストは違います】
False
>>> (1,2,3)==(3,2,1)              #【順番は守りましょう】
False
>>> {1:10,2:20} == {2:20,1:10}    #【辞書は順番関係なし】
True
>>> {3,4,5} == {5,3,4}            #【集合も順番関係なし】
True
>>>
```

数値でもコレクションでもない普通のオブジェクトの場合、’__eq__’メソッドが設定されていなければ、基本的に別オブジェクトは別値と見做されます。

```
>>> class C: pass
...
>>> c0 = C()
>>> c1 = C()
>>> c0 == c1        #【同じクラスのインスタンスでも別値】
False
>>> class Cx:               #【同値テストにパスするようにする】
...   def __eq__(self, other):
...     return self.v == other.v
...   v = 0
...
>>> cx0 = Cx()
>>> cx1 = Cx()
>>> cx0 == cx1      #【同値であると認められる】
True
>>> c0 != c1        #【比較否定テスト】
True
>>> cx0 != cx1      #【同値テストと連動】
False
>>> cx1.v = 1
>>> cx0 == cx1
False
>>> cx0 != cx1
True
>>>
```

5-2-1-5-3 大小関係

大小比較は、そもそも比較対象となる条件が厳しく、さらにシーケンスの比較などはその比較方法がわかりづらいので気をつけてください

＜大小比較演算子４種類＞

- **比較演算子'>'**
  書式：Ａ ＞ Ｂ
  動作：ＡがＢより大きいとき、真。そうでなければ偽
- **比較演算子'<'**
  書式：Ａ ＜ Ｂ
  動作：ＡがＢより小さいとき、真。そうでなければ偽
- **比較演算子'>='**
  書式：Ａ ＞= Ｂ
  動作：ＡがＢより大きいか同値のとき、真。そうでなければ偽
- **比較演算子'<='**
  書式：Ａ ＜= Ｂ
  動作：ＡがＢより小さいか同値のとき、真。そうでなければ偽

以下に、比較対象となる条件を列挙します。

1. 数値以外は、比較対象オブジェクト同士が基本的に同じ（か、あるいはどちらかの派生クラス）であること
2. ２．数値の場合は実数値（整数オブジェクト及び実数オブジェクト）のみ。虚数・複素数の大小関係は判定しない
3. コレクションの場合は、シーケンス以外は比較しない（マッピングも集合も不可）
4. シーケンスは先頭の要素から順に比較し、先に大きな値が出てきた方の値を大と判定する。同値の要素が続いた場合、先に要素が尽きた方を小とする。
5. 文字列はシーケンスに従う。１文字同士はコード番号の大きい方が大と判定する
6. その他のオブジェクトでも、比較に関する特殊メソッドが設定されていないものは判定不能。

場合が多すぎるのでごく一部ですが、例を見てみましょう。

```
>>> 1 < 2
True
>>> 1 < 2.0  #【整数と実数でも比較可能ですが】
True
>>> 1 < 2+0j #【複素数はダメ】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: no ordering relation is defined for complex numbers
>>> 1 >= -5
True
>>> 1 <= 5
True
>>> 'a' > 'A'       #【コードは小文字の方が大きい】
True
>>> 'abc' > 'Abc'
True
>>> 'abc' >= 'abc'
True
>>> 'abc' >= 'aBc'
True
>>> (10, 2.3, 5) > (10, 2.2, 100)      #【二番目の要素の大きさで見ている】
True
>>> class C:
...   def __lt__(self, other): return self.v < other
...   def __gt__(self, other): return self.v > other
...   def __init__(self, v): self.v = v
...
>>> x = C(5)
>>> y = C(10)
>>> x > y
False
>>> x < y
True
>>> x < 5
False
>>> x < 10
True
>>>
```

演算子に関係する特殊メソッドについては「オブジェクト」の項を参照してください。

5-2-1-5-4 メンバーテスト

コレクションが有するアイテムの中に特定のオブジェクトと同値のオブジェクトが存在するかどうか（要素であるかどうか）という判断をメンバーテストといいます。

メンバーテストに使用される演算子は'in'です。

- **比較演算子'in'**
  書式：A in B
  動作：AがBの要素ならば真を返し、そうでないなら偽を返す

メンバーテストは同値であり、同一性は問いません。

```
>>> t = (0,1,2,3,4,5,6,7,8,9)
>>> 1 in t                  #【要素である】
True
>>> 11 in t                 #【要素ではない】
False
>>> (1,2) in t              #【部分集合であって要素ではない】
False
>>> all(x in t for x in (1,2))   #【要素を一つずつテスト】
True
>>> {1,2}.issubset(set(t))       #【部分集合を見るなら、こちら】
True
>>>
```

「要素であること」と、「部分集合(subset)」であることは別物ですので注意してください。

'all'関数は、受け取ったコレクションの全ての要素が'True'なら'True'を、一つでも'False'なら'False'を返す関数。逆に一つでも'True'なら'True'を返し、全て'False'のときのみ'False'を返す'any'関数もある

集合オブジェクトのメソッドについては「オブジェクト」の章を参照

・比較演算子'not in'

書式：A not in B

動作：AがBの要素ならば偽を返し、そうでないなら真を返す

比較演算子'in'の逆に「メンバーでない」ことを調べる演算子です。

'not'と'in'は二つで一つ（不可分）ですので、注意してください（分けるとエラーになります）。

基本的には比較演算子'in'の使用されている部分には使用できますが、当然のことながら'for'-'in'のように、構文の一部となっている'in'の代わりに無理矢理使用することはできません。

```
>>> p = {1,2,3,4,5}
>>> 0 in p
False
>>> 0 not in p      #【基本形】
True
>>> all(x in p for x in {1,2})
True
>>> all(x not in p for x in {1,2})      #【これは可能】
False
>>> all(x in p for x not in {1,2})      #【これはダメ】
  File "<stdin>", line 1
    all(x in p for x not in {1,2})
                     ^
SyntaxError: invalid syntax
>>>
```

5-2-1-6 ビット演算

Python のビット演算子は、整数オブジェクトに対してビット単位で論理演算やビットシフトを行います。

よって演算子のオペラント（対象となるオブジェクト）は整数オブジェクトのみです。

ビット演算子は、優先度の高い順に、以下の通りです。

・ビット反転演算子'~'
書式：~Ｘ
動作：Ｘのビット表現を反転する
・ビット右シフト演算子'>>'とビット左シフト演算子'<<'
書式：Ｘ >> 右Ｎ桁（整数）、Ｘ << 左Ｎ桁（整数）
動作：Ｘのビット表現を左右にシフトする。その結果、数値的には右シフトは２のＮ乗で割り（小数点以下切捨て）、左シフトは２のＮ乗を掛けることとなる（ただし負の整数はその限りではない）
・ビット論理積演算子'&'
書式：Ｘ & Ｙ
動作：整数ＸとＹの論理積を返す

論理積の意味については「論理演算」の項を参照してください。
ビット演算の場合は０を偽、１を真として、二進法表現の各桁毎に真偽（１か０）を判定します。
論理積の場合は、ある桁について両方の数が１の場合のみ１として整数を組み立てます。

例）５＆６＝４

```
　　１０１…５
&）１１０…６
―――――――
　　１００…４
```

・ビット排他的論理和演算子'^'
書式：Ｘ ^ Ｙ
動作：整数ＸとＹの排他的論理和を返す

排他的論理和(XOR とも呼ばれる)は、片方のみ１のとき１を返す判定です。

例）５^６＝４

```
　　１０１…５
&）１１０…６
―――――――
　　０１１…３
```

排他的論理和は、論理積と論理和と否定を組み合わせれば表現できますが、電算用論理演算では頻出する（コンピュータの基礎電子回路に出てくる）ため、特に記号を設けてあります。

『x ^ y』は『(x | y) & ~(x & y)』と同等

・ビット論理和演算子'|'
書式：ＸＹ
動作：整数ＸとＹの論理和を返す

論理和の意味については「論理演算」の項を参照してください。

例）５|６＝４

```
　　１０１…５
&）１１０…６
―――――――
　　１１１…７
```

ビット演算について詳しく書くと本一冊分を軽く超えてしまいますので、ここではほんのさわりだけ紹介します。

```
>>> 5 & 6     #【論理積】
4
>>> 5 | 6     #【論理和】
7
>>> 5 ^ 6     #【排他的論理和】
3
>>> (5 | 6)&~(5 & 6)        #【排他的論理和と同じ】
3
>>> 0b101     #【二進法表記】
5
>>> 0b110
6
>>> bin(5)    #【二進法に直して表示】
'0b101'
>>> bin(6)
'0b110'
>>> bin(0b101 & 0b110)     #【論理積を二進法表記】
'0b100'
>>> bin(0b101 | 0b110)     #【論理和を二進法表記】
'0b111'
>>> bin(0b101 ^ 0b110)     #【排他的論理和を二進法表記】
'0b11'
>>>
```

5-2-1-7 演算子の多重定義と特殊メソッド

演算子の適用できるオブジェクトは、演算子に対応した特殊メソッドをもっています。

オリジナルのオブジェクトをクラス定義するとき、演算子に対応するメソッドを実装すると、その定義に応じた動作をします。

ただし、演算子の使い方（二項／単項演算子）や優先順位、結合ルールは変更できません。

特殊メソッドは、かなり沢山あるのですが、演算子の実装に直接関わるもののみ、ここで紹介します。

■二項演算子

式	特殊メソッド	特殊メソッド（逆位）
X + Y	__add__(X, Y)	__radd__(Y, X)
X - Y	__sub__(X, Y)	__rsub__(Y, X)
X * Y	__mul__(X, Y)	__rmul__(Y, X)
X / Y	__truediv__(X, Y)	__rtruediv__(Y, X)
X // Y	__floordiv__(X, Y)	__rfloordiv__(Y, X)
X % Y	__mod__(X, Y)	__rmod__(Y, X)
X ** Y	__pow__(X, Y)	__rpow__(Y, X)
X << Y	__lshift__(X, Y)	__rlshift__(Y, X)
X >> Y	__rshift__(X, Y)	__rrshift__(Y, X)
X & Y	__and__(X, Y)	__rand__(Y, X)
X ^ Y	__xor__(X, Y)	__rxor__(Y, X)
X | Y	__or__(X, Y)	__ror__(Y, X)
X < Y, Y > X	__lt__(X, Y)	
X <= Y, Y >= X	__le__(X, Y)	
X == Y, Y == X	__eq__(X, Y)	
X != Y, Y != X	__ne__(X, Y)	
X > Y, Y < X	__gt__(X, Y)	
X >= Y, Y <= Y	__ge__(X, Y)	

■単項演算子

-X	__neg__(X)
+X	__pos__(X)
~X	__invert__(Y)

サンプルも、左右がわかりやすい＋演算子と－演算子のみ説明します。

```
>>> class C:
...   def __add__(self, other):         #【＋演算子実装】
...     return self.v + other
...   def __init__(self, v=0):
...     self.v = v
...
>>> x = C(100)
>>> x + 3
103
>>> 3 + x                            #【でも逆はダメ】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'C'
>>> x + x                            #【これもダメ】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __add__
TypeError: unsupported operand type(s) for +: 'int' and 'C'
>>> C.__radd__ = lambda self, other: other + self.v #【逆パターン実装】
>>> 3 + x
103
>>> x + x
200
>>> +x                               #【でもコレは別】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for unary +: 'C'
>>> C.__pos__ = lambda self: self.v           #【符号実装】
>>> +x
100
>>> x - 3                                     #【次は引き算か？】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'C' and 'int'
>>> C.__sub__ = lambda self, other: self.v - other
>>> C.__rsub__ = lambda self, other: other - self.v
>>> C.__neg__ = lambda self: -self.v
>>> x-3
97
>>> 3-x
-97
>>> x-x
0
>>> -x
-100
>>>
```

5-2-2 関数評価式（関数呼び出し）

5-2-2-1 基本

式の中で、演算子を用いた式以上に利用されているのが、関数評価式(function evaluate expression)、即ち関数呼び出し(function call)です。

関数呼び出しの形式は以下の通りです。

書式：オブジェクト名([arg0[,arg1 ...]])

動作：オブジェクトを関数として呼び出す

関数は「関数オブジェクト(function object)」という、れっきとしたオブジェクです。

しかしここでの話題は、純正の関数に限らず、関数呼び出しの形式で使用できるオブジェクト全てについて通用します。

関数を含め、関数呼び出し形式で使用できるオブジェクトを、「呼び出し可能オブジェクト(callable object)」といいます。

【呼び出し可能オブジェクト】
・関数（functionオブジェクト）
・コンストラクタ（＝クラスオブジェクトを関数呼び出しで使ったもの）
・メソッド（ドット演算子による属性参照まで一つの『関数名』として扱う
・その他の呼び出し可能オブジェクト（'__call__'特殊メソッドに動作が記述されているもの）

関数オブジェクトとその他の「呼び出し可能オブジェクト」の振る舞いがはっきりと異なるのは、呼び出した時ではなく、クラス属性として代入した時です。

関数オブジェクトはメソッドになりますが、その他のオブジェクトはデータ属性として登録されてしまいます（詳細は「オブジェクト」の項で説明します）

実際「組み込み『関数』」として紹介されているもののうち、多くが関数オブジェクトではなく呼び出し可能オブジェクトの一種であるクラスオブジェクトです（例：'range'など）

しかし関数オブジェクトであろうが、偽物の「呼び出し可能オブジェクト」だろうが、呼び出した時の振る舞いに違いは一切ありません。

以降この章での「関数」という言葉は、「関数オブジェクト及びその他の呼び出し可能オブジェクト」を意味します。

関数は呼び出されると即時に評価されます。

関数呼び出しの後の括弧を後置の単項演算子として見た場合、その優先順位は「関数呼び出しでない括弧群」に次いで最上位にあります（同位の呼び出しやドットなどが並んだ場合は、左結合で処理します）

関数は評価されると、その場で戻り値（返り値）に入れ替えられます。

戻り値が特に設定されていない場合は、'None'オブジェクトを戻したものと見做されます。

よって、関数呼び出しは、式として他の式の中に組み込むことが可能です。

関数呼び出しは「式」であり、「値が無い」ということはありません。'None'オブジェクトは値です。値を返さない関数呼び出しはありえません。Pythonで「手続き」的なものも全て「関数」と呼ぶのは、そのためです。この違いはPythonでは意外に大きく、いかなる関数も式の中に組み込むことは出来ますが（その結果はどうあれ、文法的には）、非式文を式の中に組み込むことは出来ません（代入文など）

例を見てみましょう。

```
>>> f = lambda : 100
>>> f()              #【数値を返すだけの最も簡単な関数】
100
>>> f() + 50
150
>>> f() - 50
50
>>> 200 - f()
100
>>> 10 * f()
1000
>>> f() / 5
20.0
>>>
>>> def f():         #【自分自身を返す関数】
...   f.x += 1
...   return f
...
>>> f.x=0
>>> f()
<function f at 0x00C00D20>
>>> f.x
1
>>> f().x            #【こういう書き方や……】
2
>>> f().x
3
>>> f()()()().x      #【こういう書き方もできる】
7
>>> f()()()()()().x
13
>>>
```

5-2-2-2 キーワード引数

関数呼び出しの際、仮引数の並び順ではなく、関数に定義してある仮引数(parametor)を指定して引数を引き渡すことができます。

```
>>> def f(a,b,c):
...   return a,b,c
...
>>> f(1,2,3)
(1, 2, 3)
>>> f(b=1, c=2, a=3)
(3, 1, 2)
>>>
```

仮引数を指定するには、『仮引数名＝値』と書いて関数を呼び出します。

この書式は、仮引数にデフォルト引数が設定してある場合、一部だけを変更する時などに便利です。

```
>>> def f(a=0, b=1, c=2):
...   return a, b, c
...
>>> f(b=100)         #【b のみを指定】
(0, 100, 2)
>>> f(c=5)
(0, 1, 5)
>>> f(a=3, c=11)     #【a と c を指定】
(3, 1, 11)
>>>
```

通常の引数とキーワード引数を混ぜる際は、普通の引数（順序引数）を先に、キーワード引数を後に指定します（逆にするとエラーになります）

```
>>> f = lambda a=0, b=1, c= 2: (a, b, c)
>>> f(10, b=300)
(10, 300, 2)
>>>
>>> f(a=10,20,30)   #【キーワード引数が先だとエラー】
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

5-2-2-3 引数オプション

定義の項で多彩な仮引数オプションがあったように、Python では引数の引渡しオプションもあります。

まずは、引数リスト（この場合のリストとは'list'オブジェクトに限定せず、一列に並んだもの、という意味）の展開オプションです。

・引数リストの展開オプション
書式：関数名(*引数リスト)
動作：引数リストの[引数1, 引数2, 引数3...]を『呼び出し名』

『関数』は、前項に引き続き『関数及びその他の呼び出し可能オブジェクト』の省略形です。

リスト展開は、仮引数で引数リストを取るときの、ちょうど逆の書式になります。

リスト展開は、関数が多くのパラメータを使用する場合（例えば仮引数でリストを受け取る場合）などに有効です。

例を挙げて見ましょう。

'print'関数は、引数で受け取ったものを（文字表現にして）画面出力する簡単な（手続き的）関数です。

'print'は、Python 2 以前ではキーワードで、'print'文という構文だったのですが、様々な理由から現在は関数に変更されました。

```
>>> print(1,2,3)
1 2 3
>>> t = 1,2,3
>>> print(t)
(1, 2, 3)
>>> print(*t)
1 2 3
>>> l = [1,2,3]
>>> print(*l)
1 2 3
>>>
```

引数展開は、生成子や反復子なども展開することが可能です。

```
>>> range(5)        #【range 関数は反復子を返す】
range(0, 5)
>>> print(range(5))
range(0, 5)
>>> print(*(range(5)))    #【展開】
0 1 2 3 4
>>> k = (x ** 2 for x in range(5))
>>> k
<generator object <genexpr> at 0x00BAF878>
>>> print(k)        #【生成子は普通そのまま表示】
<generator object <genexpr> at 0x00BAF878>
>>> print(*k)             #【展開】
0 1 4 9 16
>>> print(x ** 2 for x in range(5))
<generator object <genexpr> at 0x00BAF8C8>
>>> #               #【こうやって展開することも……】
... print(*(x ** 2 for x in range(5)))
0 1 4 9 16
>>>
```

引数辞書の引数版は、辞書をキーワード引数へ展開する公式です。

・辞書をキーワード引数に展開する
書式：関数名(引数辞書)**
動作：引数辞書{'キーワード1':値1, 'キーワード2':値2, 'キーワード'3:値3...}を、関数名(キーワード1=値1, キーワード2=値2, キーワード3=値3...)に展開

注意点がいくつかあります。

まず、キーワードは文字列で書く必要があります。

また、キーワードは、関数で指定されているもの以外は受け付けません（どんなキーワードでも受け取る引数辞書が設定されているならともかく）

前出の'print'関数には、引数リスト以外に、'sep', 'end', 'file'の三つのキーワード専用(keyword only)仮引数が設定されています。

このうち、出力のセパレータ設定の'sep'と、行末処理の'end'を設定した辞書を作ってみましょう。

```
>>> op = {'sep':'-', 'end':'[end]\n'}
>>> print(1,2,3,**op)
1-2-3[end]
>>> t = 1,2,3
>>> print(*t, **op)        #【リスト展開と併用可能】
1-2-3[end]
>>> print(**op, *t)        #【ただし順番は、リスト，辞書】
  File "<stdin>", line 1
    print(**op, *t)
            ^
SyntaxError: invalid syntax
>>>
```

5-2-3 条件付評価式

条件付評価式(conditional expression 「if/else 式」と訳した本もあった)は、C言語の三項演算子に相当する文法です。

・条件評価式
書式：A if 条件式 else B
動作：条件式が真の時にはAを、偽の時にはBを返す

例を見た方が早いですね。

```
>>> 10 if 0 else 20
20
>>> 10 if 1 else 20
10
>>>
```

'if'の後の数値に注意してください。

'if'の後が偽（この場合０）ならば、'else'の後ろの数値が、'if'の後が真（この場合１）なら、前の数値が返されています。

代入文に使ったり、簡単な判断式に応用すると便利です。

```
>>> m = lambda x, y: x if 0 < x - y else y #【大きな方を返す関数】
>>> m(8,3)
8
>>> m(10,15)
15
>>>
```

【悪用法】

条件付評価式は、本来的でない方法で、さまざまなトリック（イタズラ）を仕掛けることが出来ます。

例えば、メソッドも関数の一種ですから値を返さないわけではないのですが、オブジェクトの内容を変更するようなメソッドは、大体'None'を返すことが多いようです。

例えば、リストの特定の要素を入れ替えるメソッド'__setitem__'は値を返しません。

リストの内容を変更しつつ、その値をすぐに使いたい時など、非常に面倒です。

かといって代入文を使えば簡単ですが、一式では書けなくなってしまいます。

こんなとき、条件付評価式は威力を発揮します。

さて、シチュエーションを決めておきましょうか。

リストの先頭の値に+1 して、その値を 1000 倍した値を返します。

```
>>> L = [5, 6, 7]
>>> L[0] += 1       #【通常はこうします】
>>> L[0] * 1000
6000
>>> L         #【L[0]は変更されています】
[6, 6, 7]
>>> (0 if L.__setitem__(0,L[0]+1) else L[0]) * 1000 #【一発で】
7000
>>> L         #【L[0]は変更されています】
[7, 6, 7]
>>>
```

なお、最初に【悪用法】と書いた通り、このような使い方は「実際には百害あって一利なし」です。

普通に実用的なプログラムを書く時は、決して使うべきではありません。

しかし、「出来る」ことを知っていること自体は無駄にはなりません。

また、ヒマな時に「一行で全て解く」を考えるのは、適度な脳トレになるでしょう（プログラミング言語を、遊びに使ったっていいじゃないですか）

5-2-4 その他の式(コンマ、ドット、インデックス、括弧)

基本的にオブジェクトは単独でも「式」です。

ですから、リテラルの書式は全て式の書き方と見做してもいいでしょう（ごちゃごちゃになるので、リテラルとして整理しましたが）

例えば、','(コンマ)を演算子として見れば（事実『コンマ演算子』という呼び方もある）、オブジェクトをタプルとして連結する式になります。

```
>>> x = 1,2,3
>>> x
(1, 2, 3)
>>> s = ()
>>> s = 1,s  #【簡易スタック】
>>> s = 2,s
>>> s = 3,s
>>> s
(3, (2, (1, ())))
>>> x, s = s #【取り出しは後のものから】
>>> x
3
>>> x, s = s
>>> x
2
>>> x, s = s
>>> x
1
>>> s
()
>>>
```

属性を指す'.'も「ドット演算子」と言われるように、基本的には演算子扱いです。
つまり属性参照は、属性参照式ということになります。
同様にインデックスによる参照も、参照式の一種と言えるでしょう。

```
>>> class C:
...   v = 100
...
>>> c = C()
>>> t = [c]
>>> t[0].v   #【インデックス⇒属性参照】
100
>>> (t[0]).v #【これと同じ】
100
>>>
```

属性参照やインデックス参照による変更は、オブジェクト本体の属性や要素を変更してしまうことに注意して下さい。

```
>>> x = 100
>>> y = x    #【x の指すオブジェクトを共有】
>>> y
100
>>> x = 50   #【x が指すオブジェクトが変更される】
>>> y        #【しかし y には変更が無い】
100
>>> x = [100]
>>> y = x
>>> x[0] = 50       #【x はリストへの参照を保ちながら内容を変更】
>>> y        #【リストへの参照は共有しているので、同じく変更される】
[50]
>>> class C:
...   v = 100
...
>>> c = C()
>>> x = c
>>> c.v
100
>>> x.v
100
>>> c.v = 50 #【参照で属性を変更】
>>> x.v             #【参照先が変更】
50
>>>
```

最後に、括弧も演算子の一種です。

後置の単項演算子としてオブジェクトの後に置けば、ブラケットであればインデックス、パーレンであれば関数呼び出しということになります。

しかし、通常に「はさむ」括弧も、非常に優先順の高い（最高位）演算子ということになります。

普通のパーレン括弧で囲むと、その中の処理は外の処理より「先に」行われ、評価されます。

リテラルはルールがあり、ドットや後置の括弧の制限がなされることがあるのですが、括弧によりその優先順位を変えることも可能です。

例えば当然ですが、数字の後にドットをつければ、それは小数を表すことになるわけで、整数オブジェクトのメソッドなどを使用する場合は、まず数値を評価するために括弧で括り（評価されて、オブジェクト自身に入れ替わる）、そこからメソッドを呼び出します。

```
>>> 5.__add__(8)    #【リテラルの続きとされ、拒否される】
  File "<stdin>", line 1
    5.__add__(8)
          ^
SyntaxError: invalid syntax
>>> (5).__add__(8) #【これならば通る】
13
>>>
```

他に、関数リテラル('lambda'式)を使用する場合にも、括弧で囲むといきなり関数が実行できます。

```
>>> (lambda x,y: x ** y)(9,2)
81
>>>
```

ブラケットはリスト、ブレースは辞書と集合のリテラルとなるのは、コレクションで説明したとおりです。

5-3 文

5-3-1 基本三構造

古典的なプログラミングの基礎に「構造化定理」という言葉があります。

構造化定理とは、プログラムを局所的な処理に分割して、プログラムソース全体を読みやすく理解しやすい形にする、ということです。

ここで、「順次(sequence)」「選択(selection)」「反復(iteratation)」という、プログラミングの基本３構造という概念が説明されています。
宣言文、代入文、式文などのように、普通に上から順番に実行される構造を順次（あるいは連続）構造と言い、プログラミングの基本です。
選択構造は、'if'文などに代表されるような、条件に対して制御を分岐する構造を指します。
反復構造は、'while'文などに代表されるように、一定の条件が満たされる間（あるいは一定回数）実行文のブロックを実行し続けます。
プログラムは基本的にこの３構造で全て記述できることが、構造化定理を提唱したエドガー・ダイクストラによって証明されています。
本来、構造化定理とは、無条件ジャンプによる制御の複雑化によるデバッグの困難を解消するために提唱されました。
現在では、無条件ジャンプによるフロー制御の構文が、基本的に存在しないようなプログラミング言語が大半です。
あるいはあったとしても、その言語の修得の際には、使用箇所をはっきりと限定することを習う筈です（例えばＣ言語では、多重ループを一気に抜ける時以外は、『goto』は使うべきではないと習います）
Python では、基本３構造に加え、「例外処理」という特殊な構造を導入しました。
例外処理は、基本３構造だけではやや冗長になるコードをすっきりとまとめて構造化し、さらに見やすくする効果を持っています。
また、比較的新しく導入された'with'文は、ファイルのクローズなど、忘れやすい処理を自動的に行ってくれる便利が構文です。
Python の構文は、必要最低限度しかないので、覚えるのは簡単ですが、実際の場面で活用するにはよく使うテクニックを一通り見ておくといいでしょう。

5-3-2 代入文

5-3-2-1 単純代入と演算代入

定義文の項で単純代入については少し述べましたが、代入子の中には、演算した結果を代入するものもあります。
まず、下の例を見てください。

```
>>> x = 5
>>> x = x + 1
>>> x
6
>>> x += 1
>>> x
7
>>>
```

『 x = x + 1 』という書き方は、コンピュータプログラム独特の書き方で、『x に 1 を足した数値を、改めて x に代入しなおす』という意味です。
『 x += 1 』は、これと全く同じ意味です。
'+='のような記号を本書では「演算代入子」と名前をつけますが、要するに省略形（構文糖衣）だと思っていただければ結構です。
演算代入子には以下のような種類がありますが、『 x ?= y 』は全て『 x = x ? y 』の略（?のところに演算子が嵌る）です。

+=　　加算代入子
-=　　減算代入子
***=　　乗算代入子**
/=　　除算代入子
//=　　フロー除算代入子
%=　　剰余算代入子
****=　　累乗算代入子**
>>=　　ビット右シフト代入子
<<=　　ビット左シフト代入子
&=　　ビット論理積代入子
^=　　ビット排他的論理和代入子
|=　　ビット論理和代入子

各演算については、「演算子を使った式」を参照してください。

5-3-2-2 参照先への代入と代入の仕組み

オブジェクトが持つアイテムや属性など、間接的な参照先への代入は、通常の代入とは様子がかなり異なります。

先ずはインデックスによる参照の例を見てみましょう。

```
>>> x = 100
>>> y = x     #【代入】
>>> x
100
>>> y         #【x と同じ】
100
>>> x = 200   #【代入先を変更】
>>> x
200
>>> y         #【y は変更なし】
100
>>> x = [100]        #【リストを代入】
>>> y = x
>>> x
[100]
>>> y
[100]
>>> x[0] = 200       #【リストの要素を変更】
>>> x
[200]
>>> y         #【y の要素も変更されている！】
[200]
>>>
```

直接代入では、同じオブジェクトを代入した別の変数に影響を与えませんが、間接代入では、変更が行われています。

属性参照の例も見てみましょう。

```
>>> class C: pass
...
>>> c = C()
>>> c.x = 100
>>> d = c
>>> c.x
100
>>> d.x
100
>>> c.x = 200
>>> c.x
200
>>> d.x              #【参照先が変化している！】
200
>>>
```

もし、仕組みではなくルールで覚えるのであれば、直接代入では代入先では影響がないが、間接参照で代入した場合には変更が反映される、と覚えればいいでしょう。

しかし、仕組みが気になるという方のために、少し難しくなりますが、やや深く Python の変数と代入の仕組みについて、お話します。

Python オブジェクトは、基本的に全て『参照』です。

変数は「オブジェクトそのもの」ではなく「オブジェクトにつけた名札（ラベル）」だと思ってください。

代入の本当の意味は、ただ「名札をつけるだけ」の行為です。

つまり『 x = 100 』は、『整数オブジェクト 100 に"x"という名札をつける』という意味です。

図で書くと、こんな感じでしょうか。

"x" → 100

名前は、評価されるとオブジェクトそのものを指します。

『 y = x 』は、『"x"が指していた整数オブジェクト 100 に新たに"y"という名札をつける』という意味です。

"x" → 100 ← "y"

ここで、"y"は"x"を経由せず、直接『100』につながっていることに注意してください。

さらに『 x = 200 』は、『整数オブジェクト 100 につけていた名札"x"をはがして、200 に付け替える』作業を指します。

200 ← "x"　　100 ← "y"

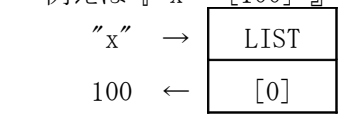
これは名札"x"の操作ですから、名札"y"には何の影響もありません（100 が 200 に変わったわけではない）

しかし、オブジェクト自身が名札を持っている場合は事情が異なります。

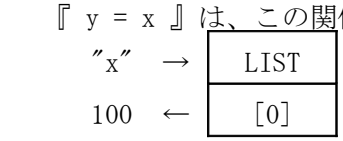
属性やインデックスは、オブジェクト自身の名札です。

オブジェクト自身の名札は、オブジェクトを指す名札から、ドット演算子'.'やブラケット'[]'を用いて参照できます。

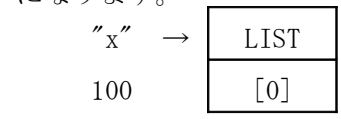
例えば『 x = [100] 』というコードでは、以下のような参照関係が成立します。



『 y = x 』は、この関係に"y"の参照が加わります。



参照を通した代入『 x[0] = 200 』は、オブジェクト自身が持つ名札が示す内容を変化させたことになります。



この場合、"x"自身の参照ではなくリストオブジェクトの名札の参照先が変化したことに注意してください。

つまり"x"を通して、"y"の参照先のオブジェクト自身（の一部）が変更されてしまったことになります。

これが、"y"を評価した結果の変更のカラクリです。

Python のオブジェクトが常に参照であることを意識する必要はありませんが、属性やインデックス参照による代入の結果だけは知っておくべきでしょう。

5-3-3 制御構文と構文ブロック

順次構文以外の選択／反復の構文では、構文ブロックを用います。

Python の構文ブロックは、有名な「インデント」で表現されます。

インデントブロックについては、好き嫌いが激しく分かれる部分の一つですので、その是非については一切主張する気はありませんが、Python の作者の GvR の言によれば「インデントは普通に誰でも使ってる書式。Python は単に、ブロックを囲う括弧や『BEGIN END』のキーワードを省いただけ」とのことです。

しかし、実は Python の構文ブロックの落とし穴は、インデントではなく「スコープ」にあります。

スコープとは、少し後に説明しますが、オブジェクト名（変数）の有効範囲のことです（また、スコープを共有するエリアを名前空間といいます）

通常のプログラミング言語では、「ブロック」と称するものの中と外では、スコープが異なっているのが普通です。

しかし Python の構文ブロックの中は、外のスコープと完全に地続きです（名前空間を共有しています）。

つまり構文ブロック内でのオブジェクトの変更は、構文ブロックの外側のスコープでの変更と同一の意味を持ちます。

スコープを切り離す（名前空間を独立させる）には、関数定義やクラス定義を使うか、完全にモジュール分割する必要があります。

具体的な例については、夫々の制御文でその都度説明しますが、やや判りづらい部分なので、気をつけてください。

【補足】インデントについて：なお、インデントはタブでもできますが、ホワイトスペースを利用する方が無難ですし、奨励されています。チュートリアルでは４スペースを採用しているようですが、私自身は２スペースを使います。なお、例文などではブロックをはっきり見せるために８スペースのタブを使用することもありますが、上記の通り、通常のコーディングではタブはお勧めできません（書いた後にエディタで一気に変換するか、あるいはエディタのタブ設定をホワイトスペースに変換するようにしておくと便利かもしれません）

5-3-4 選択構文

Python の選択構文は、基本的に'if'文だけです。

'if'文の基本は以下の通りです。

・'if'文（基本形）
書式：if〈条件式〉：〈実行文〉
動作：条件式が真なら、実行文を実行

条件「式」ですから、ここには非式文は入りません（逆に式なら何でもＯＫです）

実行文は、一行で書くならデリミタであるコロンの後に続けて書けば、インデントする必要はありません。

複数行で書く時、あるいは一行でも改行して書く時には以下のように書きます。

```
if〈条件式〉：
        〈実行文１〉
        〈実行文２〉
        ...
```

'if'文は基本的に、ある実行文の塊を条件式に従って、実行するかしないかだけを決める文です。

条件式が偽の時に実行するための実行文を指定するためには、'else'節を追加します。

```
if〈条件式〉:
        〈実行文１〉
else:
        〈実行文２〉
```

これで、条件式が真の場合には実行文１が、偽の場合には実行文２が実行されるようになります。
実行文は構文ブロックを用いれば複数式が可能ですし、上記のように一行だけなら……、

```
if〈条件式〉:〈実行文１〉
else: 〈実行文２〉
```

……でも一向に構いません。
もし、実行文が複数であっても、短い文であれば、';'を使って１行にまとめることが可能です。

```
if〈条件式〉:〈実行文 1-1〉;〈実行文 1-2〉;〈実行文 1-3〉
else: 〈実行文 2-1〉;〈実行文 2-2〉;〈実行文 2-3〉
```

この辺は、全体の見映えのバランスを見て判断して下さい（';'と改行は併用できません）
ただし、'else'節の前には必ず改行を必要とします。
さて、では選択肢を三つ以上設定したいときにはどうするのでしょうか。
一応、'if'-'else'だけでも十分記述できます。

```
if〈条件式１〉:
        〈実行文１〉
else:
        if〈条件式２〉:
                〈実行文２〉
        else:
                〈実行文３〉
```

ただし、見ての通りネスト（入れ子）が深くなってしまいますので、実行面もさることながら、非常に見づらいコードになります（インデントを揃えなければならないので……）
これが十択とかになると、もう誰も見たくないでしょう。
こんな時に'else if'を意味する'elif'というキーワードを使った'elif'節が便利です。
上の例を'elif'節を用いて書き直してみましょう。

```
if〈条件式１〉:
        〈実行文１〉
elif〈条件式２〉:
        〈実行文２〉
else:
        〈実行文３〉
```

ネストが無くなって条件と式が並ぶ単純な構造になりました。
'elif'節は場合に応じていくつでも設定できます。
そして、条件式が一つも真でなかった時のみ、'else'節が実行されます。

'elif'節を書いて、'else'節を書かなくても一向に構いません。ただし順番は'if'-'elif'-'else'の順で書きます。

なお、条件付評価式の'if'-'else'は式ですが、'if'文は「非式文」ですので注意してください。

```
>>> x = 1
>>> if x == 1:              #【条件式は真】
...   print('x == 1')
...
x == 1
>>> x = 0
>>> if x == 1:              #【条件式は偽】
...   print('x == 1')
...
>>> if x == 1:
...   print('x == 1')
... else:           #【条件式が偽のとき……】
...   print('x != 1')
...
x != 1
>>> x = 3
>>> if x == 1:
...   print('x == 1')
... elif x == 2:    #【x != 1でx == 2のとき】
...   print('x == 2')
... elif x == 3:
...   print('x == 3')
... else:
...   print('x is not 1 or 2 or 3')
...
x == 3
>>>
```

'if'文を使用せずに条件分岐を書く方法はいくつかありますが、式にまとめてしまうことが必ずしも良いとは限りません。

'if'文は一目で内容が理解しやすい構造ですので、特に理由が無ければ素直に'if'文を使うことをお勧めします。

とはいえ'if'-'elif'-'else'文も、あまりにも実行内容が長かったり、あるいは選択肢が多い場合は、内容を関数化した上でリストや辞書を'if'-'elif'-'else'文の代わりに使用したほうがすっきり書ける場合もあります。

5-3-5 反復構文

5-3-5-1 while文

Pythonには反復の制御構文が２種類ありますが、構造が単純なのはこの'while'文です。

- **'while文'（基本形）**
  書式：while〈条件式〉:〈実行文〉
  動作：条件式が真である限り、実行文を繰り返し実行する

基本的な構造と動作は'if'文と酷似しています。

違いは、'if'文は実行文を基本的に一回だけ行うのに対し、'while'文は条件式が真である限り、いつまでも繰り返し行う（＝ループする）ことです。

書き方も'if'文と同じく、一行にまとめても、構文ブロックを書いても構いません（'while'文の内容は何度も繰り返されるため、構文ブロックで書くのが一般的ですが）

```
while〈条件式〉:〈実行文〉

while〈条件式〉:
        〈実行文〉
```

条件式の内容が変化しなければ、実行文を実行しつづける、ということは、通常「実行文の中で条件式の真偽を変更させるなんらかの操作がある」ということです（でなければ、終わりません）

簡単な実例を見ていきましょう。

```
>>> x = 0
>>> while x < 10:
...   print(x, end=' ')
...   x += 1
...
0 1 2 3 4 5 6 7 8 9 >>>
>>>
```

'print'関数の'end'パラメータは、基本的には改行コードの'¥n'になっていますが、これを変更することで表示の区切りの改行を抑制することができます。ただし上記の通り、最後まで改行されないので、プロンプトの表示が汚くなります。これを解消する方法は後ほどお話します。

上の例で'x'は'while'ループの中を一巡する度に+1 されます。

その結果、最初 0 だった'x'が 10 になると、「x < 10(x は 10 未満)」という条件を満たさなくなり、'while'ループは終了します（ループを抜ける、と言います）。

なお、プログラミングテクニックの一つで、わざと「終わらないループ（無限ループ）」を書く方法もありますが、当然その場合は式の中に終了条件があります（でなければ、コンピュータが暴走してしまいます）。

無限ループの書き方と、そしてそれに関連した「反復条件式を後に書く反復構文」の書き方は、「'break'文、'continue'文」の項で説明します。

5-3-5-2 for 文

Python の'for'文は、同名のＣ言語の『for』文や、Pascal や BASIC の『FOR NEXT 文』とは全く異なります。

bash（シェルスクリプト）の『for 文』や、Perl の『foreach 文』をご存知の方は、そちらに近い形となります。

Python の'for'文の動作は、正確に定義すると若干ややこしい話になりますので、本書では異例の書き方ですが、まず通常によく用いる形の実例からお話します。

'while'ループを使わず'for'ループを使う場合は何通りか考えられますが、その中で「決まった回数だけループする」ことがわかっている場合に使用されます。

'for'文を「定回数のループ」として使う場合、以下のイディオムに従います。

- **'for'文（定回数ループ）**
  書式：for〈変数〉in range(〈回数〉):〈実行文〉
  動作：〈回数〉だけ実行文を実行。〈変数〉には 0 から順次 1,2,3...と数値が入る

実例を見てみましょう。

```
>>> for x in range(10):   #【わずか２行（１行でも可）】
...   print(x * 10, end=' ')
...
0 10 20 30 40 50 60 70 80 90 >>>
>>> x = 0                 #【まず、カウンタをセット】
>>> while x < 10:
...   print(x * 10, end=' ')
...   x += 1               #【カウンタをインクリメント】
...
0 10 20 30 40 50 60 70 80 90 >>>
>>>
```

このように、'while'文ではカウンタを予めセットした後、本文でカウンタのインクリメント(+1 すること)をしなければならないのに対し、'for'文ではストレートに記述できます。

'range'関数（実は'range'クラスなのですが）に与えるパラメータを変更すれば、変数の変化も調節することが可能です。

```
>>> #【0 から 5 未満まで 5 個】
... for x in range(5): print(x, end=' ')
...
0 1 2 3 4 >>>
>>> #【5 から 10 未満まで】
... for x in range(5, 10): print(x, end=' ')
...
5 6 7 8 9 >>>
>>> #【カウンタを２ずつ増やす】
... for x in range(0, 10, 2): print(x, end=' ')
...
0 2 4 6 8 >>>
>>> #【逆順、10 から 5 より上まで１ずつ減らす
... for x in range(10, 5, -1): print(x, end=' ')
...
10 9 8 7 6 >>>
>>>
```

'range'関数のパラメータの設定方法については、詳細は「オブジェクト」の項に譲ります。

さて、もう一つの'for'の使い方は、'range'の代わりにリストなどのコレクションを渡すことです。

基本的に内容が一つずつ順に渡されます。

```
>>> for x in [100, 200, 300]: print(x, end=' ')
...
100 200 300 >>>
>>> #【文字もシーケンス】
... for x in 'aiueo': print(x, end=':') #【判りやすいように、':'で区切る】
...
a:i:u:e:o:>>>
>>> #【辞書はインデックスを返す】
... for x in {1:100, 2:200, 3:300}: print(x, end=' ')
...
1 2 3 >>>
>>>
```

ちなみに、辞書や集合のようなシーケンス以外のコレクションは、インデックスを順に返すとは限らないので注意してください（順不同です）

そろそろ仕掛けがわかってきた方もいるかと思いますが、'for'文は本来、このようにコレクションの要素を一つずつ取り出して変数に順番に入れていくための文であり、'range'は指定されたシーケンスを生成する関数（クラス）です。

……と、説明できれば簡単だったのですが（実際、Python2 まではそうだった）現実（Python3）は若干話が厄介になっています。

ここまでで、とりあえずの'for'文の使い方はご理解できたと思いますので、これ以降の説明は、Python の動作を深く知りたい方のみお読みください。

☆ ☆ ☆

Python3 では、「反復子(iterator)」というものがその仕組みの中に大々的に取り入れられました。

反復子とは何かということについては、リテラルや定義の項で生成子に関連して少し触れましたが、「何のために」ということはお話していませんでした。

簡単に言えば、「メモリ節約のため」です。

Python2 以前は、'range'関数（この時は本当に関数だった）は、リストを生成する関数でした。

でも、現在は違います。

```
>>> range(10)
range(0, 10)
>>>
```

現在の'range'関数(クラス)が生み出すのは、'range'オブジェクトという反復子です。

巨大なコレクションは、相応のメモリを必要とします。

データベースのように、必要なものならば仕方ありませんが、たとえば 1000 回繰り返すだけの為に作られる 0 から 999 までの 1000 個の要素のリストは、無駄以外の何ものでもありませんでした。

この無駄を抑えるために、Python2 では'xrange'という省メモリ型の関数が生まれました。
この関数が生み出す'xrange'オブジェクトは、'for'ループの中で呼び出される度に、必要なアイテムをその都度生成していました。
そうすることによって、膨大な数のオブジェクトを一時的にリザーブしておく必要が無くなったというわけです。
これが、Python 初の反復子（生成子）だったと記憶しています。
おりしも、CPU の性能が上昇し、多少 CPU に余計な仕事をさせても、メモリをダダ漏れに使うよりは良い、というハードウェアの転換期でもありました。
この'xrange'の導入に伴って、対応する'for'文のメカニズムも内部で変更されていきました。
ところが、Python2⇒3 の大改訂の際、この反復子のメカニズムが大々的に取り入れられました。
例えば、'enumerate'関数、'zip'関数、'map'関数や'functools.reduce'関数（詳細は「オブジェクト」の章で）など、一時的なシーケンスを作成する関数は、ほぼ全て反復子を返す仕様に変更され、また、直接処理されていたコレクションも、内部で反復子を生成してから渡すような仕組みに統一されたのです。

反復子を一々作ったらメモリが逆にたくさん必要だ、と思われる方、ご安心を。反復子は元オブジェクトに対するポインタとしての機能しか無く、例えば'iter'関数で反復子を生成した後、元のオブジェクト（例えばリスト）の内容を変更すると、反復子が'next'関数でコールされた時には、その変更が適用されています。

```
>>> L = [1, 2, 3, 4, 5]
>>> i = iter(L)
>>> L[2] = 300             #【3→300】
>>> L.append(6)            #【6 を追加】
>>> for x in i: print(x, end=' ')
...
1 2 300 4 5 6 >>> #【変更結果が反映されている】
...
>>>
```

よって、現在の'for'文の正確な書式はこうなります。

書式：for〈変数〉in〈反復子及び反復可能オブジェクト〉：〈実行文〉
動作：反復可能オブジェクト(iteratable object)に対しては反復子を作り、それを next 関数に一回づつ渡した結果を〈変数〉（複数も可能）に収め、〈実行文〉を実行する

以上のように、メカニズムを深く理解しようと思うと面倒ですが、動作自体は単純ですので、気楽に使ってください。
なお、反復子ではなく本物のシーケンスが欲しい時は、'list'関数(クラス)や'tuple'関数(クラス)に渡せば、簡単にシーケンスが得られます。

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

反復子とハサミは使いよう、ということで……

5-3-5-3 ループの制御(break 文、continue 文)

'while'文や'for'文は非常に便利なのですが、条件判定の式が先頭に固定されていたり、あるいは回数が固定されていたりと、ある意味あまり小回りが利きません。
ループを臨機応変に扱うために、'break'文や'continue'文などのループ制御の構文があります。
'break'文が実行されると、ループがそこで中断されます。
'continue'文が実行されると、ループの構文ブロックの残りの部分がとばされ、ブロックの先頭に制御が移ります。
まず、'break'文の例からみてみましょう。

```
>>> x = 100
>>> while 1:
...   print(x, end=' ')
...   x -= 10
...   if x < 0:
...     break
...
100 90 80 70 60 50 40 30 20 10 0 >>>
>>>
```

これは、他のプログラミング言語では『do-untile 文』と呼ばれている『制御の最後にループを続けるかどうかの判定式がある』というタイプのループです。

Python には『DO UNTILE 構文』は無いのですが、'if'文と'break'文を組み合わせて似たような構文を作ることが出来ます。

なお"while 1:"は、条件式(1)が必ず真ですので、ここではループは止まりません（無限ループの書き方のイディオムです)

ただし、この例だと「終了条件」であり「継続条件ではない」と気になる方は、こう書きます。

```
>>> while 1:
...   print(x, end=' ')
...   x -= 10
...   if x >= 0:
...     continue
...   else:
...     break       #【else 節に入れなくても可】
...
100 90 80 70 60 50 40 30 20 10 0 >>>
>>>
```

'if'文が真ならばその後の'break'文を飛ばしますが、そうでなければ'break'文にひっかかって問答無用に終了します。

これでも同じじゃないか、と思われる方もいるでしょうか？

```
>>> while x >= 0:
...   print(x, end=' ')
...   x -= 10
...
100 90 80 70 60 50 40 30 20 10 0 >>>
>>>
```

結果は同じですが、実は違います。

"x"がもし -1 だったら、と考えてください。

通常の'while'の制御では、実行文は一切実行されませんが、実行文の最後に制御がある場合は、一度だけ実行されます。

プログラムの内容によっては、一度だけは実行して、その後に終了させたい場合などもあります。

'continue'は'break'を伴わずとも単独でも勿論使えます（今度は'for'文で別の例を書いてみましょう。

```
>>> for x in range(100):
...   if x > 10 and x % 10: #【10 を超えたら 10 刻み】
...     continue
...   else:
...     print(x, end=' ')
...
0 1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 >>>
>>>
```

'continue'は、上記の例のように、条件に合わない（合う）ものについてそれ以降の実行を「行わない」為に用いることも可能です。

5-3-5-4 反復構文の else 節

さて、通常のプログラミング言語であれば、反復文の話はここでおしまいなのですが、Python の場合はおまけがあります。

Python の反復制御である'while'文と'for'文には、は'if'文と同様に'else'節が書けるのです。
まず、'while'文から見ていきましょう。

while〈条件式〉:
〈実行文１〉
else:
〈実行文２〉

この'else'節は、'if'文と同じく、「'while'の条件式が満たされない時」に「１回だけ」実行されます。
ここで注意してください。
'while'ループの本体（実行文１）は、通常、条件式の条件を変更し、その結果、「条件式を満たさなく」なって終了します。
つまり言い換えると、'while'文の'else'節は「ループが正常に終了した後に行われる処理」という扱いになります。
'if'文の'else'節と異なり「どちらかを実行する」でも、「'while'が正常に終了しなかった場合」でもないことになります。
ただし、条件式が最初から満たされず、ループに入らなかった場合には、'else'節だけ実行されることになります。
では、'else'節が「実行されない」場合とはどのような場合でしょうか。
ゆっくり整理していくとわかりますが「'while'の条件文が真であり」ながら「ループが終了」する場合、即ち「'break'文」によって強制的にループを抜けた場合にのみ、'else'節は実行されない、ということになります。

前項で行ったように、『do-until構文』を'break'文を利用して実装した場合は、'else'節は実行されませんから注意してください。

'while'に限らず、反復文の'else'節は使い方が難しい（というより、無くてもさして困らない）のですが、例を上げてみましょう。

```
>>> def base(n, base=3):
...   if not (1 < base < 11):
...     return 'RANGE OVER'
...   s = ''
...   while n >= 0:      #【0 未満は、はじく】
...     n, sx = divmod(n, base)
...     s = str(sx) + s
...     if n > 0: continue
...     else: break
...   else:              #【0 未満のみひっかかる】
...     return 'ERROR'
...   return '0[' + str(base) + ']' + s
...
>>> base(100,1)
'RANGE OVER'
>>> base(100,2)
'0[2]1100100'
>>> base(100,11)
'RANGE OVER'
>>> base(100,10)
'0[10]100'
>>> base(100,5)
'0[5]400'
>>> base(25,5)
'0[5]100'
>>> base(81)
'0[3]10000'
>>> base(80)
'0[3]2222'
>>>
```

二進法や八進法や十六進法表記はありますが、ｎ進法表記は無いようなので、作ってみました（ただし、2～10 進法限定）。

ここでは、'else'節は、入ってきた数値が 0 未満だった場合にエラーとしてはじくのに利用されています。

'for'文も'while'文同様、'else'節を設定することが出来ます

for〈変数〉in〈反復可能オブジェクト〉：
〈実行文１〉
else:
〈実行文２〉

そもそも条件式で判定しているわけではないので'else'節の実行条件が難しいのですが、要するに'break'で終わった時には、'else'節は実行されないと思ってください（内部的には終了の例外をキャッチするかしないか、ということらしいのですが）

'for'文の'else'節は、要素の中に特定の条件を満たすものがあるかどうかチェックするようなコードには利用できます。

```
>>> def check4(L): #【数字の 4 が要素にあればチェックする】
...   for x in L:
...     if x == 4:
...       print('4 がありました！')
...       break
...   else:
...     print('4 はありませんでした')
...
>>> check4(range(1,10))
4 がありました！
>>> check4(range(1,10,3))
4 がありました！
>>> check4(range(0,10,3))
4 はありませんでした
>>>
```

ただし、上記のように特定のアイテムがあるかどうか調べるだけなら"4 in L"のように、すれば一行で終わってしまうので、実際にはもっと面倒な判定の場合に利用します。

ここで、裏技的になりますが、実用的な'else'節の使い方を一つ。

いままで'for'文の中で'print'関数を使う時、末尾制御('end'オプション)を改行以外に変えると、最後まで改行がされず、やや見苦しい格好になりました。

```
>>> for x in range(10): print(x, end=' ')
...
0 1 2 3 4 5 6 7 8 9 >>> #【←最後が改行されないので見苦しい】
...
>>>
```

この「反復構文」の章では、一貫してその表示について無視しつづけてきたのですが、実は簡単に解決できます。

```
>>> for x in range(10):
...   print(x, end=' ')
... else: #【最後に実行】
...   print()
...
0 1 2 3 4 5 6 7 8 9
>>> #【↑改行されてすっきり】
...
>>>
```

5-3-6 例外

5-3-6-1 基本形

例外(exception)の処理は、Python の構文を利用する上で、一つのポイントとなります。

例外とは、構文や不正な処理を警告する機能です。

放っておくと、この Python インタープリタはそこで処理を止めてしまいます。

```
>>> k = 1 / 0 #【数値をゼロで割っている】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>> #【ゼロ除算エラー】
...
>>>
```

例外処理は、プログラム実行中に生じた様々なエラーをどう処理するかの構文ということになります。

例外処理の構文である'try'文の基本形は以下の通りです。

```
try:
        〈実行文〉
except〈例外の種類〉:
        〈その例外発生時の処理〉
```

'try'節の実行文が実行された結果、'except'節に指定してある例外が発生した場合、'except'節の内容が実行されます。

基本的にプログラムの実行に関わる重大な例外（ほぼエラー）は、予め登録してあります。

よく見る（いや、見たらマズいのですが）エラーには、以下のようなものがあります（ほんの一例）

- NameError：デフォルトでもなく、ユーザーも定義していない名前を使った時などに発生
- ZeroDivisionError：数値をゼロで割った時に発生
- TypeError：操作に合ったオブジェクトを使っていない（数値を文字列に足したりした時など）
- ValueError：値に変換できないときなど（int 関数に'abc'とか渡した時など。同じ文字列でも'200'とかなら可能だから）

なお、構文エラーを意味する'SyntaxError'だけは、例外処理の'try'文でキャッチされるまでもなく、プログラムが停止します。

```
>>> try:
...   k = 1 / 0
... except ZeroDivisionError:
...   print('Zero Division Error!')
...
Zero Division Error!
>>>
```

'except'節では、複数のエラーをキャッチすることができます。

また、複数の'except'節を設定することもできます。

```
>>> try:
...   x = unknown          #【名前エラー】
... except ZeroDivisionError:
...   print('Zero Division Error!')
... except NameError:
...   print('Name Error!')
...
Name Error!
>>> try:
...   x = unknown
... except (NameError, ZeroDivisionError):
...   print('Name or Zero Division Error!')
...
Name or Zero Division Error!
>>>
```

古い情報の本では、Python3 では２種類以上のエラーをキャッチする際に、括弧で囲う必要がなくなる、と書かれているものがあります。確かに当初はそのような動きがあったようですが、結果的には Python2 以前の通り、括弧で括ってタプルにすることが必須になりました。

'except'でエラーの種類を特定しない場合、('SyntaxError'以外の)全てのエラーをキャッチすることが出来ますが、何のエラーか特定できないため、バグを残すことになりかねないので、あまりお勧めできません。

```
>>> try:
...   k = 1 / 0
... except:
...   print('Something Error?')
...
Something Error?
>>>
```

このような場合、とりあえずエラーをキャッチした上で、エラーを上（?）に送るため、'raise'文を使うのが常套手段です。

```
>>> try:
...   k = 1 / 0
... except NameError:     #【NameErrorだけキャッチ】
...   print('Name Error')
... except:         #【それ以外はエラーを再送】
...   print('Not Name Error')    #【確認メッセージ】
...   raise               #【エラーを再送】
...
Not Name Error
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
```

'raise'文は、例外を発生させる構文で、この他にも様々な使い方がありますが、別項に分けてでしっかりと説明します。

5-3-6-2 終了処理と正常処理

例外が発生する／しないに関わらず、実行しておきたい処理、というものはあります。

特に、バッチ型（プログラムが最後まで自動的に処理を行う）ではなく対話型のプログラムに於いて、ユーザの想定外の入力よるエラーなどは日常茶飯事ですが、それによってプログラムが中断されると、途中経過などが失われてしまって再開することが困難になることが往々にしてあります。

たとえミスがあって処理を止めざるを得ない場合でも、必ず行わなければならない操作は、'try'文の'finally'節に記述することが出来ます。

ここで例として挙げるために、二つの組み込み関数を簡単に説明しておきます。

一つは'input'関数で、ユーザの入力を文字列として受け取る関数です。

もう一つは'eval'関数で、文字列をPythonの式として評価する関数です。

この二つの組み込み関数は「オブジェクト」の項で詳しく説明します。

```
>>> log = ''
>>> try:
...   x = input('<<< ')          #【入力する】
...   y = eval(x)          #【入力内容を評価する】
... finally:
...   log = log + x + '\n'       #【失敗しても記録(log)は取る】
...
<<< 100 / 0
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<string>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>> log                          #【ログ内容の確認】
'100 / 0\n'
>>>
```

処理が失敗しても、記録にはちゃんと残されていることがわかります。

'finally'節は、'except'節と一緒に書くことが出来ますが、一番最後に書きます(″final″ですから)

```
>>> log = ''
>>> try:
...   x = input('<<< ')
...   y = eval(x)
... except ZeroDivisionError:
...   print('Zero Division !')
... finally:
...   log = log + x + '\n'
...
<<< 555 / 0
Zero Division !
>>> log
'555 / 0\n'
>>>
```

Python2.4 以前は'except'節と'finally'節は同時に使えませんでした。Python2.5 以降及び Python3 では、一緒に使うことができます。

さて、'if'文のみならず'for'文、'while'文にまで存在する'else'節は、'try'文には無いのでしょうか？

いいえ、あります。

'try'文でも、'except'節の後、'finally'節の前という位置に書くことができます。

'finally'節は必須ではありませんが、'except'節は一つは必要です。

なぜなら'else'の意味の「～そうでなければ」の「そう」が何なのかわからないからでしょう。

そうであるにも関わらず、'else'節の内容は、'try'節で行われた全ての処理が、例外を「一切」送出しなかった際のみに行われます。

'except'節に指定されている以外の例外が送出されても、'else'節は実行されないことに気をつけてください。

```
>>> try:
...   1 / 100
... except NameError:
...   print('Name Error')
... else:           #【例外が出なければ実行】
...   print('OK?')
...
0.01
OK?
>>> try:
...   1 / 0
... except NameError:
...   print('Name Error')
... else:           #【NameError 以外の例外でも実行されない】
...   print('OK?')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
```

'finally'節、'else'節をまとめて、'try'文の完全な書式を見てみましょう。

try:〈実行文〉
[except〈例外〉:〈その例外時の実行文〉]
[except:〈例外をキャッチできなかった全ての例外発生時の実効文〉]
[else:〈例外が送出されなかった時の実行文〉]
[finally:〈例外の送出のあるなしに関係なく実行する文〉]

なお、'try'文は、'except'節か'finally'節のどちらか一つは必ず持ちます。

繰り返しになりますが、構文エラー('SyntaxError')はキャッチできません。

5-3-6-3 例外の取得と例外の送出

複数の例外をキャッチする場合、その例外が何であったかがわかると便利です。

```
>>> try:
...   1 / 0
... except(NameError,ZeroDivisionError) as e:
...   print(e)            #【情報の表示】
... except:
...   raise
...
int division or modulo by zero
>>> try:
...   x = spam
... except(NameError,ZeroDivisionError) as e:
...   print(e)
... except:
...   raise
...
name 'spam' is not defined
>>>
```

'except'節にで'as'キーワードを用いてオブジェクトを指定すると、発生した例外オブジェクトを捕捉することができます。

さて、例外はプログラマが故意に送出することも可能です。

```
>>> raise ZeroDivisionError      #【クラスを送出】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
>>> raise ZeroDivisionError()    #【インスタンスを送出】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
>>> raise ZeroDivisionError('0 de watta')     #【インスタンスに引数を】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: 0 de watta
>>>
```

'raise'文は、例外を送出する（発生させる）構文です。

・例外送出'raise'文
書式：raise〈例外〉
動作：〈例外〉を送出する

出来合いの例外だけでなく、プログラマが新しい例外を作ることも出来ます。

例外は、'Exception'クラスを継承して作ります。

```
>>> class NewException(Exception):      #【新しい例外を作る】
...   def __init__(self, value=''):
...     self.value = value
...   def __str__(self):
...     return 'New Exception: '+self.value
...
>>> raise NewException('test')          #【例外を送出】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NewException: New Exception: test
>>> try:
...   raise NewException('test')
... except NewException as e:           #【例外をキャッチ】
...   print(e)
...
New Exception: test
>>>
```

上記のものは一例であり、'__str__'の戻り値を設定することでエラー表示ができることだけわかれば、あとは自由に設定してもかまいません（ただし「あまり大量の情報を含ませるべきではない」とチュートリアルには書いてあります)

自作のソフトウェアを作る際に、ユーザの入力について、想定される様々なシチュエーションに対して適切なエラーメッセージを出すことは賢明です。

たとえば、自然数（ 0 より大きい整数）のみの入力（文字列）を受け付けたい場合の判定は、「全ての文字が"0123456789"のどれか」であり「先頭が"0"でない」の二つで十分でしょう。

```
>>> class InputError(Exception):
...   def __init__(self, value=''):
...     self.value = value
...   def __str__(self):
...     return 'input error: ' + self.value
...
>>> def inputNN():
...   try:
...     n = input('自然数を入力してください\n<<< : ')
...     if n[0]=='0':
...       raise InputError('数字の先頭が 0 です')
...     else:
...       for x in n:
...         if x not in '0123456789':  #【メンバーテスト】
...           raise InputError('数字以外が混じっています')
...       else:
...         print('正常な入力を確認しました')
...         return int(n)
...   except InputError as ie:
...     print(ie)
...     inputNN()
...
>>> inputNN()
自然数を入力してください
<<< : 150
正常な入力を確認しました
150
>>> inputNN()
自然数を入力してください
<<< : 015
input error: 数字の先頭が 0 です
自然数を入力してください
<<< : o-157
input error: 数字以外が混じっています
自然数を入力してください
<<< : 157
正常な入力を確認しました
>>>
```

ユーザに入力を促すソフトウェアでは、例外処理は非常に強力なツールになりえます。

5-3-6-4 例外処理の応用

例外処理というと、予期せぬ事態、あるいは不都合な事態に対しての処理というイメージが多いのですが、実は「別に異常でもなんでもない」事態でも例外を使うことができます。

これは、実はPythonの文法の内部でも利用されています。

例えば反復子は要素を一通り吐き出してしまうと、'StopIteration'という例外を送出します。

実は'for'文は、内部でこれを捕捉してループを終了しているのです。

つまり例外は、「異常事態処理」に限定する必要は一切無いのです。

ところで、基本三構造のところで「無条件ジャンプを極力使わない」ということの例外として「多重ループを一気に抜ける場合を除く」というくだりに少し触れました。

Pythonでは様々なツールが揃っていますので、多重ループを書く必要は殆ど無いのですが、全く皆無というわけではありません。

例えば、三重構造のタプルがあったとします。

```
>>> tt = tuple(
...   (
...     tuple(
...       tuple(
...         range(x,x + 10, 5)
...       ) for x in range(y,y+2)
...     )
...   ) for y in range(1,3)
... )
>>> tt
(((1, 6), (2, 7)), ((2, 7), (3, 8)))
>>>
```

この中を検索して、ある要素があるかどうか調べるために'break'文を使って書くと、やや冗長になります。

```
>>> def memberB(L, n):
...   Get = False
...   for x in L:
...     for y in x:
...       if n in y:
...         Get = True
...         s = 'Get ' + str(n) + '!'
...         break   #【内側のループを抜ける】
...     if Get:
...       break           #【外側のループを抜ける】
...   else:
...     s = 'Not Get ' + str(n) + '!'
...   print(s)
...   return Get
...
>>> memberB(tt, 3)
Get 3!
True
>>> memberB(tt, 5)
Not Get 5!
False
>>>
```

例外を用いれば、同じものがわりとすっきり書けます。

```
>>> def memberE(L, n):
...   class GGet(Exception): pass
...   Get = False
...   try:
...     for x in L:
...       for y in x:
...         if n in y:
...           raise GGet()       #【ループを一発で抜ける】
...   except GGet:
...     s = 'Get ' + str(n) + '!'
...     Get = True
...   else:
...     s = 'Not Get ' + str(n) + '!'
...   print(s)
...   return Get
...
>>> memberE(tt, 3)
Get 3!
True
>>> memberE(tt, 5)
Not Get 5!
False
>>>
```

ループが複雑になった場合など、例外を使えば一気にジャンプすることが出来ます。

【補足】上の例は、説明用のかなり強引な使用法であり、実際にはこのように関数化されている事例では分岐処理の後、

'return'文で抜けてしまうのが一番賢いやり方でしょう。ただし、例外を補足する条件が多数あったり、例外があるのと無いので共通の処理が多かったりする場合など、つまり処理の記述を集中させるには非常に便利です。ただ、そのようなやや大規模なコードを書くと読んで理解する方も大変なので、今回は、本来「使うまでもない」例に使用して、感覚を知っていただくことを目的としていることを補足しておきます。

実は、三重ではなく二重ループの例も考えたのですが、流石に'in'メンバーテストを無視するのは不自然なので、面倒な三重構造タプルを考えました。でも、よく考えたら、リストの方が式は簡単でした……

```
tl = [([list(range(x,x + 10, 5)) for x in range(y,y+2)]) for y in range(1,3)]
```

5-3-7 終了処理

プログラムでファイルを使用した場合、終了時点でファイルハンドルを解放(close)する必要があります。

一応、Pythonインタープリタが終了する際に自動的に解放されますが、不必要にファイルハンドルを占有するのは無意味ですし要らぬ事故も招きます。

終了処理(clean up)の'with'文は、主にこのファイルの処理のために出来た構文です。

```
>>> with open('myfile.txt') as f:
...   for line in f:
...     print(line)
...
aaaaa

bbbbb

ccccc

ddddd

>>> for x in f:            #【既に閉じられている】
...   print(x)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>> f
<_io.TextIOWrapper name='myfile.txt' encoding='cp932'>
>>>
```

二度目の処理の処理は、ファイルが閉じられているかどうか確認するためです。

・終了処理構文'with'

書式：with〈終了予定オブジェクト〉as〈変数〉：

動作：ブロックが終了したとき、終了動作を行う。

Python 3.1以降では、『with A() as a, B() as b:』のように、複数の終了予定オブジェクトを設定することができるようになりました。

'with'文は、ファイルオブジェクトのように終了操作が予め設定してあるオブジェクトに使用します。

追加モジュールなどで'with'文を使うよう指示されるオブジェクトはこれから出てくるでしょう。

「終了予定オブジェクト(clean up object)」を自作することも、勿論可能です。

まず、終了予定オブジェクトは'__enter__'メソッドと'__exit__'メソッドを実装します（引数の数や実行のタイミングは、以下の例を参照してください）

```
>>> class W:
...   def __enter__(self):
...     print("enter")           #【__enter__実行のタイミング】
...     return 'aiueo'
...   def __exit__(self, exc_type, exc_value, traceback):
...     print("exit sequence")   #【__exit__実行のタイミング】
...     print("exc_type:",exc_type)
...     print("exc_value:",exc_value)
...     print("traceback:",traceback)
...
>>> with W() as w:
...   print(w)                   #【実行文の実行のタイミング】
...
enter
aiueo
exit sequence
exc_type: None
exc_value: None
traceback: None
>>>
```

以上のコードを見ればご理解いただけるかと思います（引数の数に過不足があると、エラーを引き起こします)

引数を取る場合は、'__enter__'メソッドではなく、普通に'__init__'を使用します。

```
>>> class W:
...   def __init__(self, v):
...     self.v = v
...   def __enter__(self):
...     return self.v
...   def __exit__(self,a,b,c):
...     return 0    #【__exit__の戻り値は捨てられます】
...
>>> with W('hello') as w:
...   print(w)
...
hello
>>>
```

Python の凄いところは、生成子にせよ修飾子にせよこの'with'文にせよ、増設した文は単一の利用法に止めず、必ず汎用性を持たせて、他の利用が可能になるようにしているところです。

皆さんも Python 開発陣の期待に応えて、面白い利用法を考えてみてください。

5-3-8 注釈文(コメント)

Python のプログラムコードの中で、'#'以降改行するまでの文は、コメントと見做されます。

但し（当然ではありますが）文字列の中に書かれた'#'は文字として扱います。

コメントは、継続中の行（閉じていない括弧など）の中にも書くことができます。

```
>>> 1 + 2 #【ここに書くのは普通】
3
>>> 1 + #【こう書くとエラーになるが……】
  File "<stdin>", line 1
    1 + #【こう書くとエラーになるが……】

SyntaxError: invalid syntax
>>> (1 + #【これはＯＫ】
... 2)
3
>>> #【コメントから始まると……】
... 1 + 2 #【次に続く】
3
>>>
```

正確にはコメントではありませんが、説明文(docstring)のように、実行文中に挿入された文字列のみの文は、コメントとして使用できます。

特にPythonには複数行にわたるコメントを書く書式が無いので、三重引用符の文字列を代わりに使用することが多いです。

なお、コメントではないということは「評価される」ということですので、何度も繰り返し実行するループの中にあまり長い無駄な文字列を入れるべきではないでしょう。

```
>>> del C
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'cls']
>>> 'コメント'
'コメント'
>>> '''
... 複数行にわたるコメント
... 実行内容に影響は無いけど、
... 評価されるので、演算の無駄
... '''
'\n複数行にわたるコメント\n実行内容に影響は無いけど、\n評価されるので、演算の無駄\n'
>>>
```

5-3-9 スコープルール

プログラムコードの現在の行で、どのオブジェクトにアクセスできるかということを知っておくのは、プログラマにとって非常に重要なことです。

オブジェクトにアクセスできる範囲をスコープ(scope 範囲／射程という意味)といい、そのルールをスコープルール(scope rule)といいます。

スコープを深く理解するには、名前空間(name space)と呼ばれるものを理解する必要があります。

名前空間とは、オブジェクトの名前を管理しているデータベースのことです。

しかし名前空間は、Pythonというシステム側から見た言葉なので、ユーザ側からはやや判りづらい概念です。

そこで、ここではまず、Pythonプログラミングに必要な「スコープルール」の話に絞ります。

名前空間という上位概念を理解すれば、スコープルールの説明はほぼ必要なくなりますが、スコープルールを完全に理解するために必要な名前空間に関する知識はかなり広範囲なので、とりあえずここでは、通常のプログラミングに必要な知識を選んで集めることにしました。名前空間を理解した後にこの章を読み直すと、全ての理由が氷解するのではないかと思います。

5-3-9-1 トップレベル

トップレベル(top level = 最上層)、という用語は、Pythonプログラミングにおいて度々言及されるので、知っておくべきでしょう。

といっても、全然難しい話ではありません。

「コードを書き始めた時点で、関数定義の中でもクラス定義の中でも無い場所」のことを指します。

Ｃ言語やJava言語をご存知の方は、どちらの言語にもトップレベルに相当するものは無いことをご存知かと思います。Ｃ言

語は全て関数内に定義しますし、Java では全てクラス内に定義します。その意味では、Python は BASIC や FORTRAN やのような、クラッシックな言語に近いスタイルを取っているともいえます。

インタープリタで、プロンプトが『>>>』になっているところがトップレベルです。

```
>>> x = 100  #【ここはトップレベル】
>>> def f():
...   x = 200       #【ここはトップレベルではない】
...   return x
...
>>> f()             #【この結果はトップレベル】
200
>>> x         #【トップレベルで x の変更はない】
100
>>>
```

ただし、構文ブロックの中は、プロンプトが『...』になりますが、トップレベルです。

```
>>> if True:
...   x = 200       #【構文ブロックの中】
...
>>> x         #【トップレベルが変更されている】
200
>>>
```

クラス定義や関数定義のブロック（定義ブロック）と、制御構文の構文ブロックをはっきりと区別してください。

基本的に定義ブロックの中は非トップレベル（ローカル）です。

そして構文ブロックの中は（そのコード自体がトップレベルにあるなら）常にトップレベルになります。

判りづらい例を一つ。

```
>>> x = 0
>>> for x in range(5):
...   print(x, end=' ')    #【改行せずに表示】
... else:
...   print()                      #【最後に改行】
...
0 1 2 3 4
>>> x                      #【カウンタに使った x が変更されている】
4
>>>
```

明らかに'for'文内で使用するだけのカウンタ（アイテムホルダ）の x ですが、内容が変更されてしまいます。

上記の例では、生成子リテラルを使えばトップレベルの変更は制御できます。

```
>>> x = 100  #【x を設定】
>>> l = [ x ** 2 for x in range(5)]
>>> l
[0, 1, 4, 9, 16]
>>> x         #【x の値に変化なし】
100
>>> print(*(x for x in range(5)))
0 1 2 3 4
>>> x         #【x の値に変化なし】
100
>>>
```

関数リテラル（'lambda'式）も関数扱いですので、同じです。

トップレベルに変化を与えるのは、地の文と構文ブロックと覚えておいてください。

Python2 では、リスト内包記法は生成子とは異なった書式でしたので、その中の"x"(内部変数)は、実は'for'文と同様、変化していました。Python3 ではその点が改訂され、全て生成子を用いた式という扱いになり、内部の変数はトップレベルに影響を及ぼすことはなくなりました（つまりローカル変数となります）

5-3-9-2 関数定義内のスコープ

5-3-9-2-1 基本

ここでいう『関数』とは純粋に'function'オブジェクトの定義のことであり、『呼び出し可能オブジェクト』を指しません。それらはまとめて次項の『メソッド定義内のスコープ』で扱います。

関数定義内のスコープルールは、基本的には以下のようになっています。

・関数の外側の値を参照することはできるが、その値を直接代入によって変更することは出来ない

例を挙げます。

```
>>> def f0():
...   print(x)              #【外側の x を参照】
...
>>> def f1():
...   x = 666               #【ローカルに x を設定】
...   print(x)
...
>>> x = 500
>>> f0()             #【外側の x】
500
>>> f1()             #【ローカルの x】
666
>>> x                #【外側には影響なし】
500
>>>
```

"f0"の"x"のように、関数内で定義されず、外側の値を参照する変数を、Scheme などの言語では自由変数(free variable)と言います。

ここで気をつけなければならないことが、一点あります。

Python では、自由変数（として見て欲しい変数）とローカルに定義した変数を混在させるとエラーになるのです。

例を挙げます。

```
>>> x = 100
>>> def f():
...   print(x)
...   x = 200
...   return x
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

最後のエラーは、「ローカル変数が定義前に参照されています」という意味です。

関数定義の最初の x は外側の参照、代入した後はローカル変数として見ればいいのですが、この関数が関数オブジェクトとして生成された過程で、関数定義内の x は全てローカル扱いとなり、そして定義前に使用された変数はエラーとなるのです。

これは、一つの定義内で、参照を複雑にしないための処置だと思われます。

もし、外部の値を取り込みたければ、普通に引数として渡すのが普通です。

```
>>> def f(x):                #【引数は最初からローカル】
...   print(x)
...   x = 200
...   return x
...
>>> x = 100
>>> f(x)             #【トップレベルの x を値として渡す】
100
200
>>>
```

あるいは、以下のような方法もありますが、問題があるのでお勧めできません。

```
>>> x = 100
>>> def f(x=x):              #【内部 x←外部 x という式】
...   print(x)
...   x = 200
...   return x
...
>>> f()
100
200
>>> x = 500
>>> f()                      #【変更には非対応】
100
200
>>>
```

コメントにあるように、引数は一度しか評価されませんから、「その時点での」"x"が評価されたらそれっきりです。

『x = x』のような書式は、古いPythonでは閉包を作るテクニックとして存在しましたが、後に説明する'nonlocal'キーワードが導入された現在ではほぼ不要となりました。

関数を返す関数を関数リテラル(lambda式)で書いて、引数を引き渡したい時など以外は、『x = x』という書き方はしないほうが良いでしょう。

```
【関数を返す閉包の一種】
>>> f = lambda x:lambda y, x=x: y + x
>>> f10 = f(10)
>>> f10(5)
15
>>> f20 = f(20)
>>> f20(5)
25
>>>
```

5-3-9-2-2 参照による変更

参照に関するスコープルールは多少厄介です。

本項を読む前に、「参照先への代入と代入の仕組み」を一読しておくことをお勧めします。

要点をまとめれば、以下の通りになります。

・関数定義の中であっても、引数や自由変数（前項参照）に対して、リストのインデックスやオブジェクトなどの属性参照による代入を行うと、オブジェクトそのものの内容が変更されてしまう

例を挙げます。

```
>>> def f(x):
...   x.y = 500            #【属性 y を 500 に変更】
...
>>> class C: pass
...
>>> c = C()
>>> c.y = 100
>>> f(c)
>>> c.y                    #【属性の値が変更されている】
500
>>> def fl(L):
...   L[0] = 500    #【インデックスで変更】
...
>>> l = [100]
>>> fl(l)
>>> l                #【アイテムが変更されている】
[500]
>>> class X: pass
...
>>> x = X()
>>> x.y = 1000
>>> xl = [1000]
>>> def fx():
...   x.y = 500
...   xl[0] = 600
...
>>> x.y
1000
>>> xl
[1000]
>>> fx()      #【自由変数の属性やアイテムの変更】
>>> x.y
500
>>> xl
[600]
>>>
```

変数そのものでなく、参照を通して変更すると、その変更は関数内のものであっても有効になります。

これは、代入によって同じオブジェクトを指す変数の、一方から参照を用いてオブジェクトの属性やアイテムを変更した場合に、オブジェクト自身が変更されてしまうことと全く同じです。

Python では、スコープ的には、オブジェクトを参照できることと、オブジェクトの属性を変更できることがイコールになります。

もし、オブジェクトの属性を不用意に変更させたくない場合は、記述子やプロパティーなどを用いて保護するか、あるいは不変オブジェクトであるタプルを使用するなどの対策が必要かもしれません。

5-3-9-2-3 入れ子になった関数のスコープ

関数定義は入れ子にすることが出来ます。

入れ子になった関数のスコープは、基本的に自分の近くから検索します。

```
>>> x = 100
>>> def f():
...   x = 200
...   def ff():
...     x = 300
...     def fff():
...       print(x)  #【一番近くの x を検索】
...     fff()
...   ff()
...
>>> f()
300
>>>
```

以前の Python では、スコープルールがかなりややこしかったのですが、Python3 以降はすっきりと整理されています。

自由変数の参照も、一番内側の変数から検索されています。

```
>>> x = [0]
>>> def f():
...   x = [1]
...   def ff():
...     x = [2]
...     def fff():
...       x[0] = 100
...     fff()
...     print(x)
...   ff()
...   print(x)
...
>>> f()
[100]
[1]
>>> x
[0]
>>>
```

スコープルールは、「定義された位置」によって決まります。
「呼び出された位置」ではないので、注意しましょう。

```
>>> def fx():
...   x[0] = 100    #【この関数のすぐ外側はトップレベル！】
...
>>> x = [50]
>>> def ff():
...   x = [60]
...   fx()          #【ここで実行しても……】
...   return x
...
>>> x
[50]
>>> ff()            #【関数定義内は反映されず……】
[60]
>>> x               #【トップレベルに反映されてしまう】
[100]
>>>
```

本書は文法のルールを解説する本なので、こういった自由変数の話も説明しますが、基本的には自由生起(occu free)型の変数である自由変数は使用せず、値は明示的に引数として受け渡し、値は戻り値として受け取るのが、判りやすい（透過性の高い）良いプログラムの書き方でしょう。

5-3-9-3 クラス定義内のスコープ

5-3-9-3-1 クラス定義文のスコープ

クラス定義内のスコープルールは独特です。

というのもクラス自体が、スコープルール自身の規定を左右する「名前空間(name space)」そのものだからです。

名前空間には「クラス」と「モジュール」があります。モジュールについては後ほどお話します。

まずは、トップレベルとクラス定義の関係を見てみましょう。

```
>>> v = 100
>>> class C:
...   x = v  #【トップレベルの'v'を参照】
...   v += 1 #【クラス内の'v'を定義】
...
>>> C.x             #【名前空間'C'の'x'】
100
>>> C.v             #【名前空間'C'の'v'】
101
>>> v         #【トップレベルの'v'には影響なし】
100
>>> x         #【トップレベルの'x'は定義されていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

一見、関数と似ているようですが、微妙に違います。

クラス定義一行目の文の"v"は、明らかにトップレベルの"v"を参照しています。

しかし、二行目の文の"v"は、トップレベルの"v"でもあり、クラス内の"v"でもあります。

つまり……

v = v + 1

の、左辺の"v"はクラス内の、右辺の"v"はトップレベルの"v"です。

つまり、実際にはこのように展開されていると考えてください。

C.v = v + 1

このように、クラス定義内ではトップレベルとクラス内のオブジェクトが混在しているように見えても、名前空間ではっきりと分割されているので問題ないのです。

同じコードを関数定義で行ってみてください。関数は特に名前空間を区切るわけではないので（スコープは局所的(local)になりますが）、実行時にエラーが出ます。

クラス内とトップレベルの関係はシンプルで、クラス内からは参照できるが、新規／変更したオブジェクトはクラスという名前空間に登録されることになります。

ただし……例によって「参照代入」は、新しくオブジェクトを定義するわけではありませんので、参照という扱いでトップレベルのオブジェクト（オブジェクト自身）に影響を与えます。

```
>>> x = [500]
>>> class C:
...   x[0] = 777    #【クラス内で参照代入】
...
>>> x               #【クラス外でも……】
[777]
>>>
```

クラス定義内に関数定義を書くことについては、次の「メソッド」の項で説明しますが、逆の「関数の中でクラスを定義する場合」について少し触れておきます。

```
>>> x = [100]
>>> y = [200]
>>> def f():
...   y = [300]     #【関数定義内の'y'】
...   class C:
...     x[0] = 1000
...     y[0] = 2300
...   return y[0]
...
>>> f()
2300
>>> x               #【変更された】
[1000]
>>> y               #【変更されていない】
[200]
>>>
```

クラスが関数内で実行されると、そこを基点として名前を検索しているのがわかります。

もう一つ見てみましょう。

```
>>> def ff():
...   class C:
...     k += 1
...   return C.k
...
>>> k = 100
>>> ff()
101
>>>
>>> k
100
>>>
```

関数内部であっても、クラスによる名前空間がある場合、『k += 1』は、『C.k = k + 1』と分解されるのです。

でも、コレはダメです。

```
>>> def fx():
...   class C:
...     x[0] = 666
...   x = [100]
...   return x
...
>>> x = [200]
>>> fx()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fx
  File "<stdin>", line 3, in C
NameError: free variable 'x' referenced before assignment in enclosing scope
>>>
```

クラス定義内とはいえ、"x"が関数内ローカルなのかトップレベルなのかはっきりしない場合は、エラーが出ます。

関数内の場合は、評価順序としては後に来るコードにも気をつけてください。

最後に、クラス内クラスのスコープを見てみましょう。

```
>>> x = 100
>>> class C:
...   x = 200
...   class D:
...     print(x)
...
100
>>>
```

これが多分、最大の難関になると思います。

これを見てわかるように、クラスの「スコープ」は関数と異なり『入れ子になりません』。

"C"には、"D"というオブジェクトが登録されますが、"D"の『外側』は、名前空間"C"ではないのです。

"D"から"C"を参照するには、"C"の名前を使用するしかないのですが、当然"D"の実行時には"C"という名前は無いので、使用することは出来ません。

```
>>> x = 100
>>> class C:
...   x = 200
...   class D:
...     print(C.x)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in C
  File "<stdin>", line 4, in D
NameError: name 'C' is not defined
>>>
```

順序としてはこうなるからです。

"C"内部の実行（含む"D"の内部の実行）⇒"C"の定義

このような操作を行う場合は、即時実行するクラスではなく、遅延して実行するメソッドを使用するしかありません。

```
>>> x = 100
>>> class C:
...   x = 200
...   class D:
...     def p(): print(C.x)
...
>>> C.D.p()
200
>>>
```

メソッドについては次章に説明しますが、名前空間の名前を、その名前空間以外から参照しようとすれば、名前を明示的に記述して参照するしかない、ということを覚えておいてください。

5-3-9-3-2 メソッド定義文のスコープ

「定義」の章で少し説明しましたが、ここでメソッドのスコープについて改めて確認します。

クラス定義文のスコープで見たとおり、クラス定義内は局所的(local)ではなく別の名前空間として扱います。

ですからその中で定義された関数（つまりメソッド）から、「外側」を覗こうとしても、それは無駄です。

メソッドの「外側」は、基本的にはそのままトップレベル（あるいは全てが関数定義の中であれば、関数定義ローカル）であると考えてください。

```
>>> x = 100
>>> class C:
...   x = 200
...   def m():
...     print(x)
...
>>> C.m()    #【クラスの外のトップレベルを参照】
100
>>> def f():
...   class C:
...     def m():
...       nonlocal x      #【関数定義の中なので nonlocal 宣言可能】
...       x = 500
...   x = 100
...   C.m()
...   return x
...
>>> f()                    #【変更されました】
500
>>>
```

「静的メソッド」の定義のところで説明しましたが、現在の Python3 では、クラスからの呼び出しは基本的に全て静的メソッド（つまり第１引数に特別な意味のない、ただクラスという名前空間に属する関数）として扱います。

クラスメソッドにすると、クラスから属性参照しようが、インスタンスから属性参照しようが、第１引数はクラスそのものになります。

```
>>> class CC:
...   @classmethod #【クラスメソッド】
...   def m(cls):
...     print(cls.x)
...   x = 'CC in'
...
>>> CC.m()
CC in
>>> cc = CC()
>>> cc.m()
CC in
>>> cc.x = 'cc local'      #【インスタンス属性を設定しても……】
>>> cc.m()          #【クラス属性を参照】
CC in
>>>
```

'nonlocal'文で閉包を書く方法は既に紹介しましたが、同じ原理で、専用のオブジェクトからしかアクセスできないオブジェクトを作ることができます。

```
>>> def f():
...   counter = 0
...   class Cx:
...     def count(self):
...       nonlocal counter
...       counter += 1
...       return counter
...   return Cx()
...
>>> k = f()
>>> k.count()              #【'k'からしか'counter'にはアクセスできない】
1
>>> k.count()
2
>>> k.count()
3
>>> kk = f()
>>> kk.count()             #【'kk'の'counter'は別物】
1
>>> k.count()
4
>>>
```

クラスの名前空間と、関数のローカル空間を使い分けると、アクセッサとは別の方法でアクセスのコントロールが可能になります。

5-3-9-4 変数のスコープの変更

変数のスコープの変更については、「宣言文」の章で説明した「'global'文」と「'nonlocal'文」を参照してください。

また「メソッド定義文のスコープ」の章に、'nonlocal'の追加情報があります。

ここでは、説明の漏れたいくつかの次項を補足的に説明するにとどめます。

クラス定義は新しい名前空間であってローカルではありませんので、'nonlocal'文は使用できません。

```
>>> class C:
...   x = 100
...   def k():
...     nonlocal x
...     x = 200
...   k()
...
SyntaxError: no binding for nonlocal 'x' found
>>>
```

それに対して、global 文は普通に使えます。

```
>>> x = 777
>>> class C:
...   def k():
...     global x
...     x = 666
...   k()    #【トップレベルの'x'を変更】
...
>>> x
666
>>> x = 888
>>> C.k()    #【クラスを通して呼び出す】
>>> x
666
>>>
```

スコープルールと'global'文、'nonlocal'文はワンセットですので、両者を見比べて使用してください。

5-3-10 import 文とモジュール

モジュールという言葉は、プログラミング言語によって様々な意味を持ちます。
Python では、モジュールとは単に、分割されたファイルに書かれたコード、及び、そこに登録されたオブジェクト群をとりまとめるパッケージオブジェクトです。
モジュールはクラスと同じく名前空間ですが、クラスと異なり、モジュール内のオブジェクトの『外側』として参照できる関数的な局所空間でもあります。

【実験に使うモジュールファイル（実行パス内に置いておく）】

```
# tmod.py
print('test module')

def func0():
  print(x)

x = 'in module'

def func1():
  print(y)

class mC:
  def m(self):
    print(x)
```

【実行】

```
>>> import tmod
test module
>>> x = 'out of module'
>>> tmod.func0()    #【モジュール内を検索】
in module
>>>
```

・モジュール導入構文'import'
書式：import〈モジュール名〉
動作：〈モジュール名〉.py という名前のファイルの内容を読み込み、〈モジュール名〉の名前空間で実行する

クラスと同じく、モジュールオブジェクトも生成した時点で内部のコードを実行します（定義も実行文の一種）
そしてモジュール内の関数のスコープは、モジュール内を検索します。
トップレベルは、モジュール内の関数のスコープにありません。
ので、もしトップレベルのオブジェクトの参照が必要ならば、引数として引き渡してやるか、あるいは「モジュール空間」に値を渡してやる必要があります。

```
>>> y = 'out of module'
>>> tmod.func1()    #【トップレベルは検索しない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c:\prog\python\tmod.py", line 10, in func1
    print(y)
NameError: global name 'y' is not defined
>>> tmod.y = 'insert module'
>>> tmod.func1()    #【モジュール内を検索】
insert module
>>>
```

上記の例の通り、モジュールオブジェクトとして、トップレベルにはモジュール名が存在しますので、モジュールの属性として値を代入すると、関数はその値を検索します。

モジュール全体ではなく、モジュールの中から特定のオブジェクトのみを導入したい場合は、'from'節を追加します。

・モジュール内オブジェクト導入構文'from'-'import'
書式：from〈モジュール名〉import〈オブジェクト名〉
動作：〈モジュール〉内の〈オブジェクト〉のみを導入する。〈オブジェクト〉は〈オブジェクト名〉として使用できる

```
>>> from tmod import func0
test module
>>> func0()
in module
>>>
```

導入されたのが一つのオブジェクトだけだったとしても、モジュール内部の文は実行されます。
ただし、その実行は「一度きり」で、既に導入されている（モジュールの一部だとしても）モジュールの実行文は、２度は実行されません。
以下のコードは前のコードの続きですが、行頭の'print'関数は実行されていないのがわかります。

```
>>> from tmod import mC          #【既にprintは実行されていて、新たにオブジェクトを導入するだけ】
>>> c = mC()
>>> c.m()                   #【メソッドのスコープの実験⇒関数に同じ】
in module
>>>
```

モジュール内クラスのメソッドのスコープも、モジュール内関数と同じです。
'import'文には、いくつかのオプションがあります（'from'節もその一つです）
'as'節は、導入したモジュールを任意の名前に割り当てる、手軽で便利な使い方です。
これは、'import'単独でも、'from'節を含めた'from'-'import'形式でも使用できます。

```
>>> from tmod import func0 as f
test module
>>> f()
in module
>>> import tmod as t
>>> t.func0()
in module
>>>
```

・モジュール／モジュール内オブジェクトのラベリング導入構文['from'-]'import'-'as'
書式：import〈オブジェクト名〉as〈導入名〉
from〈モジュール名〉import〈オブジェクト名〉as〈導入名〉
動作：導入後のオブジェクト名が〈導入名〉になる

'as'は、この構文が導入された直後、しばらくキーワードではありませんでしたが、現在はキーワード扱いです。

さて、この項の最後になりますが、'from'-'import'形式で、モジュール内のオブジェクトを一括して登録する方法があります。

・モジュール内オブジェクト一括導入構文'from'-'import *'
書式：from〈モジュール名〉import *
動作：〈モジュール〉内の全てのオブジェクトを、〈モジュール〉に登録された名前のまま導入する

例を見ます。

```
>>> from tmod import *
test module
>>> func0
<function func0 at 0x00C00DF8>
>>> func1
<function func1 at 0x00C00E40>
>>> x
'in module'
>>> mC
<class 'tmod.mC'>
>>>
```

この書式は非常に便利なのですが、使い方を間違えると非常に危険です。
例を見てみましょう。

```
>>> print('hello')
hello
>>> from tmod2 import *
>>> print('hello')
'over write!'
>>>
```

これは何事か……実は、"tmod2"の中身を見れば理由は判明します。

```
# tmod2.py

def print(*x, **y):
  return 'over write!'
```

つまり、'print'関数を上書きしてしまっているのです。

組み込み関数を上書きすることは、別にタブーではありませんので、意識して使えば何の問題もありません。

しかし'import'文のワイルドカード'*'による一括導入は、上書きが明示的に行われないので、知らずに使えば非常に危険です（『名前空間の汚染』と呼ばれることもあります）

その点を踏まえて、一括導入の一般的な使用基準を以下に記しておきます。

1. 一括導入するモジュールが自作であり、導入されるオブジェクト名を全て把握していること
2. 一括導入による使用を念頭に置いて作られたモジュール(tkinter など)であること。ただしこの場合は、モジュールの仕様を詳細に把握しておくか、あるいはそのモジュールを一括導入する「モジュール」を分割して、導入するのが望ましい

'as'の他、オブジェクトは常に代入による変名が可能なので、よほどの場合でない限り、ワイルドカードを使用する必要は無いと思われますが、使用する際は注意してください。

なお、モジュールに'__all__'属性を記述しておくと、ワイルドカードでインポートできるオブジェクトを制限できます。

【"tmod"に以下の１行を追加】

```
__all__=['x', 'func0']
```

```
>>> from tmod import *
test module
>>> func0()
in module
>>> func1()          #【これは対象外】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'func1' is not defined
>>>
```

5-3-10-1 補足：実行パスの調べ方

実行パスをインタープリタ実行中に調べるには、以下のように行います。

```
>>> import sys
>>> print(*sys.path, sep='\n')
c:\prog\python
C:\Python31\python31.zip
C:\Python31\DLLs
C:\Python31\lib
C:\Python31\lib\plat-win
C:\Python31
C:\Python31\lib\site-packages
>>>
```

※sys.path には path 名がリストになって入っているので、表示方法はご自由に

5-3-10-2 補足２：モジュールと文字コード

本来、本書では Python の文法について説明するものであって、特定の環境や実装については触れな

いことにしていますが、読者が自作のモジュールを作った際に問題になりそうな点だけを、軽く触れておきたいと思います。

Python モジュールは基本的には Unicode の UTF-8 エンコーディングが準拠となっています。UTF-8 エンコーディングは ASCII ファイルがそのまま使用できる規格なので、モジュールに ASCII 文字のみを記述する場合は何の問題もありません。

問題はそのファイル内に日本語などのマルチバイト文字を入れる場合です。

日本で日常的に使用されている S-JIS エンコーディングでモジュールを使用する場合は、以下の一行を行頭に加えてください。

```
# encoding:shift_jis
```

正式にはもっとコテコテと書くようですが、これで十分です。

ちなみに、（Windows 付属のメモ帳でもできますが）UTF-8 のエンコードが指定できるエディタであれば、それで保存すれば、行頭に上記のようなモノを書く必要はありません。

また、UNIX でよく使用される EUC-JP を使用する場合は、

```
# encoding:euc_jp
```

で使用できます。

詳しくは codecs のモジュールの説明などを参照してください。

5-3-11 パッケージ

Python には、モジュールをまとめるパッケージという機能があります。

パッケージ化の話題はかなり高度になるので、ここでは簡単な導入方法と使用法だけを書いておきます。

まず、パッケージ化するモジュールを入れたフォルダを実行パス上に置きます。

パッケージフォルダの中には、'__init__.py'という名前のファイルをおきます。

このファイルは、パッケージ使用時の各種初期設定を行うのですが、とりあえず内容は空でも構いません。

空のテキストが登録しづらい時は、先頭行にコメントでも書いておいて下さい

あとは、モジュールの容れ物でもあるかのように、指定すれば OK です。

【構成】"tpac"というフォルダに"tpacmod.py"というファイルがあり、"func"というメッセージを表示するだけの関数が入っている。

【呼び出し方１】

```
>>> from tpac import tpacmod
>>> tpacmod.func()
in package
>>>
```

【呼び出し方２】

```
>>> import tpac.tpacmod
>>> tpac.tpacmod.func()
in package
>>>
```

【呼び出し方１】

```
>>> from tpac.tpacmod import func as f
>>> f()
in package
>>>
```

なお、下記の例は、通りそうで通りません。

```
>>> import tpac
>>> tpac.tpacmod.func()          #【tpacmod と名前は無いと言われる】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'tpacmod'
>>>
```

また、'__init__.py'が空のままだと、コレもダメです。

```
>>> from tpac import *
>>> tpacmod.func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'tpacmod' is not defined
>>>
```

最後の例に関しては、'__init__.py'ファイルに、一行つけくわえると、通ります

【'__init__.py'に下の１行を追加】

```
__all__=['tpacmod']
```

```
>>> from tpac import *
>>> tpacmod.func() #【tpacmodが追加されている】
in package
>>>
```

これはモジュール内のオブジェクトのワイルドカードによる検索を制限する'__all__'属性と結果は全く同じ効果（導入できるオブジェクトやモジュールを指定する）ですが、モジュールのワイルドカードは、指定が無ければ全てのオブジェクトを導入するのに対し、パッケージのワイルドカードは指定が無ければどのモジュールも導入しません。

(2010 年 5 月 22 日投稿受付)

RMagick でカレンダー壁紙を作る

きしだあつし

なにやらカレンダーの話題が続いてしまいますが、もともとこれがはじまりでした。

携帯電話の 2G 網が終了することを受けて、提供されていた様々なサービスも終了していき、便利に使っていたカレンダーのはいった壁紙も利用できなくなりました。携帯電話の機能としてカレンダーやスケジューラといったものを、待ち受け画面として表示させておくこともできるかと思いますが、それだと単にカレンダーだけで味気ないものです。季節のイラストや写真といったものにカレンダーがはいっていたら、壁紙としての機能も満足させてくれます。

基本的にカレンダーの形態で、数字を合成すればいいだけですから、画像処理のソフトなどで作ればよかろうと、当初は思っていました。しかし、手作業で数字を貼り付けていくのは、どうにも手のかかる作業に思えます。それならば、プログラムにするほうがよくはないか。画像処理を Ruby で行うためのものがなにかあったと思ったはずと、記憶をたどって思いだしたのが、ImageMagick です。

ImageMagick はさまざまなプログラムで利用することができるようになっているうえ、行える処理も非常に多彩です。もちろん、文字や数字といったテキストを合成することもできます。ここでは、主にテキストとベースとなる画像との合成を紹介しますが、画像の解像度を一定のものにまとめて変換するなど、ちょっとした、それでいてまとまった作業を自動化することに、さまざま応用が利くはずです。

■インストール

必要になるのは以下のプログラムです。（ここでは、Windows 環境について説明しています）

・Ruby 1.8.7
・ImageMagick　(http://www.imagemagick.org/script/index.php)
・RMagick　(http://rmagick.rubyforge.org/)

RMagick は、Ruby から ImageMagick を利用するために使われるライブラリです。Windwos 用には、バイナリパッケージが用意されています。このパッケージには、ImageMagick も同梱されているので、ダウンロードは RMagick のサイトから Win32 用バイナリパッケージだけをダウンロードすれば大丈夫です。ImageMagick のバージョンが、かならずしも最新ではない場合もありえますが、動作が確実に保証されているパッケージのまま使用しておくのが、ひとまずは安心かと思います。

RMagick サイトのダウンロード項目（一番下のほうにあります）にある、

> MS Windows binary Gem(A pre-compiled Win32 Gem bundled with ImageMagick)

のリンクをたどり、「rmagick-win32」のなかにある、

> 2.12.0 binary gem for Ruby 1.8.6　　October 4, 2009
> RMagick-2.12.0-ImageMagick-6.5.6-8-Q8.zip

というファイルをダウンロードします。（2010/5/25 現在）

アーカイブを展開し、まず ImageMagick-6.5.6-8-Q8-windows-dll.exe （数字部分はバージョンによっては異なります）を実行して、ImageMagick をインストールします。

続いて Ruby にパスの通ったコンソールを開き、gem ファイルの置かれたフォルダで、

```
>gem install rmagick-2.12.0-x86-mswin32.gem
```

として RMagick を gem ファイルからインストールします。（ x86 より前の数字は、バージョンによっては異なります）

なお、Windows において Ruby 1.9.1 には対応していません。1.8.6 もしくは 1.8.7 を使用します。RMagick そのものは、1.9 にも対応していますが、Windows 用のバイナリでは 1.9 用が提供されていません。現在、Ruby は 1.9 系への移行時期にあるので、いずれは 1.9 系用のバイナリも登場するかもしれません。

■ImageMagick を試してみる

ImageMagick に用意されている処理はたくさんありますが、そのいくつかを実際に試して実感してみましょう。まずは、画像ファイルを用意してください。とりあえず試すだけですから 320*240 ピクセル程度の小さめのファイルが便利かもしれません。仮に適当な小さめの画像がないとしても、ImageMagick の機能を使って縮小することもできます。

画像のフォーマットタイプについては、一般的な多くのタイプに対応していますので、手元にある画像で、まず、問題ありません。JPEG、PNG、GIF、TIFF などから用意すればよいでしょう。

以下に示す List.1 を rm_sample.rb などとして、画像ファイルと同じフォルダに置いてください。

List.1 : rm_sample.rb

```
--------------------
require 'rubygems'      # <========= rmagick を gem でインストールしたので必須
require 'rmagick'

file = '********.jpg'  # <========= 実際の画像ファイル名を入れます
img = Magick::ImageList.new file

img.flop.write 'flop.jpg'        # <== write('filename') で保存
img.flip.write 'flip.jpg'
img.resize(80,60).write 'resize.jpg'
img.oil_paint(radius = 3.0).write 'oil_paint.jpg'
img.sepiatone.write 'sepiatone.jpg'
img.wave(amplitude = 22.0, wavelength = 130.0).write 'wave.jpg'
img.swirl(180).write 'swirl.jpg'
--------------------
```

準備ができたら、画像とプログラムを置いたフォルダでコンソールを開き（あるいは、コンソールを開いた後に、そのフォルダに移動して）、実行します。

>ruby rm_sample.rb

実行すると、７つの新しい画像ファイルが作られます。画像と該当するプログラムを見てみましょう。

元の画像
（画像：GC111_03.jpg）

img.flop
（画像：flop.jpg）

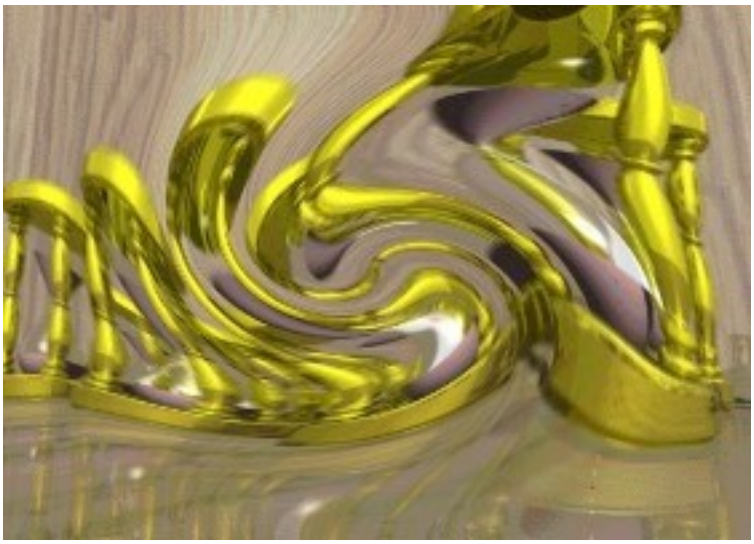
img.flip
(画像：flip.jpg)

img.resize(80,60)
(画像：resize.jpg)

img.oil_paint(radius = 3.0)
(画像：oil_paint.jpg)

img.sepiatone
（画像：sepiatone.jpg）

img.wave(amplitude = 22.0, wavelength = 130.0)
（画像：wave.jpg）



img.swirl(180)
（画像：swirl.jpg）

細かなことは抜きにして、どんな効果をもたらすのかが視覚的に理解できるかと思います。数字があるものについては、少し変えて試してみると、さらにその変化が見えてきます。実際に加工するときにも、少しずつ変えて試しながら、適当な値をみつけていけばよいでしょう。

RMagick のドキュメント（英文ではありますが）には、さらにたくさんの機能が書かれているので、そこに示された例を参考にして、いろいろ試してみてください。それによって、実際どのような機能なのかが、わかってくるでしょう。

【ポイント】
ImageMagick を Windows で使用する際に、注意しなくてはならないことがあります。それは文字コードにまつわる問題です。ImageMagick は UTF-8 で処理します。そのため、画像にテキストを合成したりする際には UTF-8 でエンコードされた文字を渡す必要があります。

同様にファイル名やフォルダ名にも気をつけなくてはなりません。日本語 Windows では Windows-31J（一般には Shift-JIS と呼ばれています）が使われているため、日本語のフォルダやファイルを指定すると、ImageMagick では処理ができません。この場合、フォルダ名などを UTF-8 でエンコードしなおして渡しても、ImageMagick 内では正しく認識されるでしょうが、Windows 側では文字化けしてしまうことになり、結局うまく動作しません。プログラムと画像ファイルを、同じフォルダに置いて作業したのは、そういう理由もあってのことです。

プログラムに使用するフォルダ名やファイル名には、日本語を使わず英数字のみを使うようにします。仮に上位のフォルダに日本語名が使われていても、同じフォルダに必要なファイルをおいて、カレントファイルとして実行する分には、影響がありません。

もしも、任意のフォルダからファイルを選択して読み込ませるようなプログラムを作る場合には、注意が必要です。

■カレンダーを合成する

それでは本題に戻って、画像にカレンダーを合成してみましょう。リストは次のようなものになりました。ただし、ここでは 240*320 ピクセルの画像ファイルで作成することを想定しています。サイズが異なる場合には、初期値の値などを適宜変更して試してください。

実行は、

```
>ruby calkgs.rb xxxxxx.png 2010 6
```

のようにします。使用する画像ファイル、年、月を指定して実行します。

List.2 : calkgs.rb

```
require "rubygems"
require "rmagick"
require "date"
require "date/holiday"

def usage
  print "usage: ruby calkgs.rb base_picture_filename yyyy mm¥n"
  exit()
end

usage if ARGV.size < 3

base_picture = ARGV.shift
year = ARGV.shift.to_i
month = ARGV.shift.to_i

days_of_month = Date.new(year, month, -1).day   # 作成する月の日数
cal_img = Magick::ImageList.new(base_picture)   # ベースになる画像

def text_print( xpitch, w_day, xpos, ypos, text, text_color, text_point)
  @cal.pointsize(text_point)
  @cal.fill("white")          #                    白い文字を
  @cal.stroke("white")        #       白で縁取りした
  @cal.stroke_width(6)        # 広めに
  @cal.text(xpitch * w_day + xpos, ypos, text)  #             描画
  @cal.fill(text_color)       # その上に、
  @cal.stroke("none")         #           縁取りなしで文字を
  @cal.text(xpitch * w_day + xpos, ypos, text)  #           描画
end

@cal = Magick::Draw.new  # カレンダーを描画するオブジェクトを用意
year_month = Date.new(year, month)
# xpos, ypos : 横方向の初期値, 縦方向の初期値
# xpitch, ypitch : 横・縦方向の文字間隔
# month_fig_xx : 月数の色、文字サイズ
```

```
# month_name_xx : 月名の色、文字サイズ
# date_fig_xx : 日付の文字サイズ
xpos = 35
ypos = 90
xpitch = 28
ypitch = 40
month_fig_color = "blue"
month_name_color = "blue"
month_fig_point = 50
month_name_point = 20
date_fig_point = 20

# 月数を作成
text_print( 0, 0, 50, 60, "#{month}", month_fig_color, month_fig_point)

# 月英名を作成
month_name = year_month.strftime("%B")
text_print( 0, 0, 90, 40, month_name, month_name_color, month_name_point)

#
# 日付を作成
(1..days_of_month).each do |d|
  w_day = Date.new(year, month, d).wday   # 曜日
  #
  # 日曜・祝日は赤色に、土曜は青色に、平日は黒色に
  if Date.new(year, month, d).national_holiday?
    text_color = "red"
  elsif w_day == 0
    text_color = "red"
  elsif w_day == 6
    text_color = "blue"
  else
    text_color = "black"
  end

  @cal.text_align(Magick::CenterAlign)  # 文字はプロット座標に対して中央揃えで
  text_print( xpitch, w_day, xpos, ypos, d.to_s, text_color, date_fig_point)

  #
  # 土曜日を描画したら改行する
  ypos += ypitch if w_day == 6
end

# カレンダーを描画
@cal.draw(cal_img)

f_name = "cal" + year_month.strftime("%y%m") + ".jpg"
cal_img.write(f_name)
-----------------
```

今回は祝日判定をするために、 holiday.rb をインストールしてあります。

http://www.funaba.org/ruby.html からダウンロードし、展開したら README ファイルに記載されたインストール方法に従ってインストールしてください。展開したフォルダにはマニュアルもあるので、参考にしてください。

順に説明していきます。

```
-----------------
def usage
  print "usage: ruby calkgs.rb base_picture_filename yyyy mm¥n"
  exit()
end

usage if ARGV.size < 3
-----------------
```

実行時に使用する画像ファイル名、作成するカレンダーの年月を入力するようにしているので、万一それらが不足しているときの簡単な予防として、使い方が表示されるようにしています。ただし、厳密にそれらをチェックしているわけではありません（引数が 3 つあるかどうかを見ているだけです）。

```
-----------------
def text_print( xpitch, w_day, xpos, ypos, text, text_color, text_point)
  @cal.pointsize(text_point)
  @cal.fill("white")          #                    白い文字を
  @cal.stroke("white")        #       白で縁取りした
  @cal.stroke_width(6)        # 広めに
  @cal.text(xpitch * w_day + xpos, ypos, text)  #             描画
  @cal.fill(text_color)       # その上に、
  @cal.stroke("none")         #           縁取りなしで文字を
  @cal.text(xpitch * w_day + xpos, ypos, text)  #           描画
end
-----------------
```

カレンダーの日付・月数・月の英名を描画する部分は、テキストを描画するということについては共通なので、汎用的に使えるようにまとめてあります。渡されるそれぞれの変数については、続く初期設定値の部分で簡単に説明をいれてあります。

ここでの特徴は、文字の回りに白く縁取りを施す方法です。縁取りがないと、画像と重なった場所によっては、文字が読みにくくなってしまいます。stroke が縁取りの色を、stroke_width がその幅を指定していますが、描画された文字の内と外に向かって均一に縁取り処理がされます。このため、文字サイズが小さいと、わずかな縁取り幅でも文字そのものがつぶれてしまい、文字そのものの線がわからなくなってしまいます。

次のように修正して実行すると、その様子がわかります。

```
-----------------
```

```
def text_print( xpitch, w_day, xpos, ypos, text, text_color, text_point)
  @cal.pointsize(text_point)
#  @cal.fill("white")          #                      白い文字を
  @cal.fill(text_color)    # <=== "white" を text_color に変更
  @cal.stroke("white")        #       白で縁取りした
  @cal.stroke_width(6)        # 広めに
  @cal.text(xpitch * w_day + xpos, ypos, text)  #             描画
#  @cal.fill(text_color)       # その上に、
#  @cal.stroke("none")         #           縁取りなしで文字を
#  @cal.text(xpitch * w_day + xpos, ypos, text)  #           描画
end
-----------------
```

そこで、縁取りは縁取りで描画し、その上に縁取りしない文字を重ねることで、文字の幅を圧縮しない縁取りを作っています。

```
-----------------
xpos = 35
ypos = 90
xpitch = 28
ypitch = 40
month_fig_color = "blue"
month_name_color = "blue"
month_fig_point = 50
month_name_point = 20
date_fig_point = 20
-----------------
```

ここでは初期値を設定しています。それぞれを変えて試してみてください。

月数と月の英名を描画したら、つぎは日付です。

```
-----------------
 1:(1..days_of_month).each do |d|
 2:   w_day = Date.new(year, month, d).wday   # 曜日
 3:   #
 4:   # 日曜・祝日は赤色に、土曜は青色に、平日は黒色に
 5:   if Date.new(year, month, d).national_holiday?
 6:     text_color = "red"
 7:   elsif w_day == 0
 8:     text_color = "red"
 9:   elsif w_day == 6
10:     text_color = "blue"
11:   else
12:     text_color = "black"
13:   end
14:
15:   @cal.text_align(Magick::CenterAlign)  # 文字はプロット座標に対して中央揃えで
16:   text_print( xpitch, w_day, xpos, ypos, d.to_s, text_color, date_fig_point)
```

```
17:
18:  #
19:  # 土曜日を描画したら改行する
20:  ypos += ypitch if w_day == 6
21:end
-----------------
```

1: で、 1 からその月の日数(days_of_month)まで値を増やしつつ、その数字について処理を進めます。

2: で、その数字が示す月日の曜日を調べ、それによって日付の色を決定します。同時に、national_holiday? で祝日なのかどうかも判定します。

15: で、文字を描画する座標を中心として中央揃えに指定しているのは、合成縁取り処理のためでもあり、日付の数字がきれいに揃うためにも役立っています。

できあがりは、実行したフォルダに保存された calxxxx.jpg ファイルを表示してみてください。こんな感じになります。

（画像：cal1006.jpg）

■その先へ

ひとまず、それらしいものができあがりました。祝日にも対応しているので、まずまず実用的かもしれません。

それでも、まだ不満があります。出来上がりは、その都度ファイルを見なければわかりません。また、サンプルのような単純な画像では問題ありませんが、キャラクターのイラストなどを使用した場合には、数字が表情などを隠してしまって残念なときもあるかもしれません。そんな時に、初期値などを手作業で修正しては実行し、ファイルを確認する、を繰り返しているのは、なんとも面倒そうです。

実行したら、そのまま結果が見られると便利です。さらに、細かな調整がそこで行えたらまた便利です（Ruby/Tk などの出番ですね）。文字のフォントを変えたらまた雰囲気が変わりそうです。発展の余地はまだまだありそうです。

==========

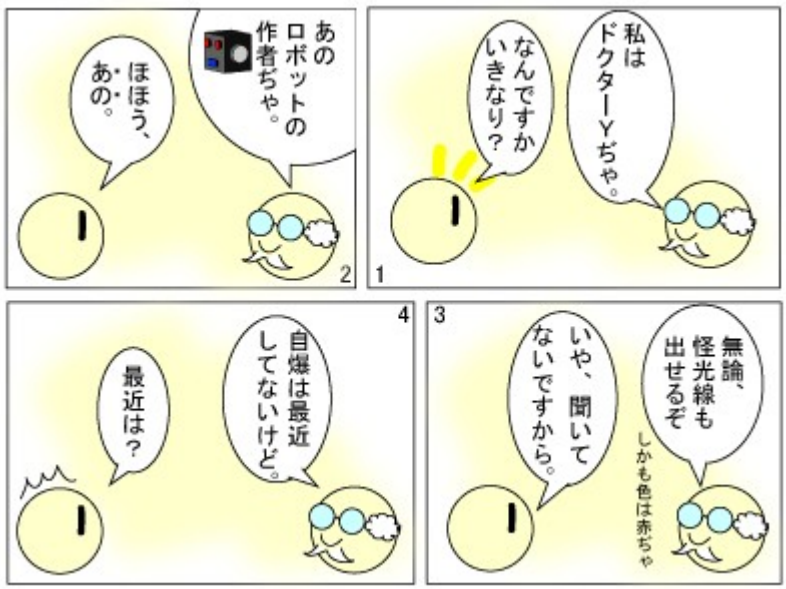
＊使用したサンプル画像は、著作権フリー CG 画像データ集「CG photo library -photograph & design-」（株式会社ツァイト：企画・制作、株式会社ジーイー企画センター：画像制作・販売、(c)1995 zeit）収録のものです。

==========

(2010 年 5 月 28 日投稿受付;2010 年 5 月 30 日更新)

よしおさんとロボ太
「この親にして」

海鳥作



特技はあんたの趣味か。

日曜工作的プログラミング
ラジオエアチェック管理サーバーを作る

jscripter

1. はじめに

本稿は、ローカルサーバー(デスクトップサーバー)で動作する CGI(Perl によるデスクトップ CGI)に関するものである。課題は二つある。

課題の一つは CGI から Windows の日本語名フォルダに存在するファイルをリストアップしてアクセスできるようにすることである。もう一つの課題は、サーバーのデフォルトのフォルダ以外のフォルダをサーバーのフォルダとしてアクセスできるようにすることである。

この課題を解決すれば、デスクトップ CGI は Windows のデスクトップを自在に動き回れるようになる。

具体的には、USB RADIO の録音ファイルが格納されるフォルダにアクセスし、録音ファイルを Web ブラウザ上で再生する CGI を作成する。

2. USB RADIO

昨年の 3 月に購入した「SUNTAC RD-PC-101 USB AM/FM RADIO」は、PC に USB 接続して、PC 上で FM/AM ラジオを聞くことができ、録音のセッティングをすればラジオ番組のタイマー録音が可能である。PC をスリープ状態にしておけば、自動的に PC を起動して所定の番組を録音してスリープ状態に復帰することができる。

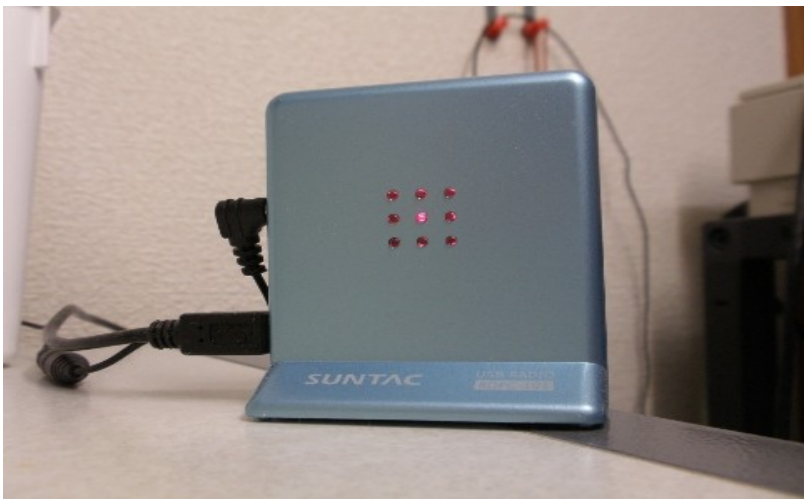


図 SUNTAC RD-PC-101 USB AM/FM RADIO

USB RADIO のアプリケーションのウインドウは次図のようである。About を調べると、ソフトウェアのバージョンは 2.0、ハードウェアのバージョンは 1.01.01 となっている。購入当初はタイマー録音がうまく動作しない問題があったが、ソフトウェアがアップグレードされて問題はなくなった。

図 USB RADIO の起動画面

日本語 Windows のフォルダは SJIS 日本語で表示できるので、フォルダの意味を示すのに大変便利である。USB RADIO のソフトウェアは「予約名」を録音ファイルを格納するフォルダ名として使う。録音のセッティング用のウインドウは次図のようである。

図　USB RADIO の録音設定画面

録音設定の前に受信するラジオ局を FM/AM について設定しておく必要がある。この設定画面で、録音ファイル保存先を指定する。受信設定用のウインドウは次図のようである。

図　USB RADIO の受信設定用画面

実際に生成された録音ファイル保存先のフォルダ(E:¥RadioREC)の状態は次図のようになる。

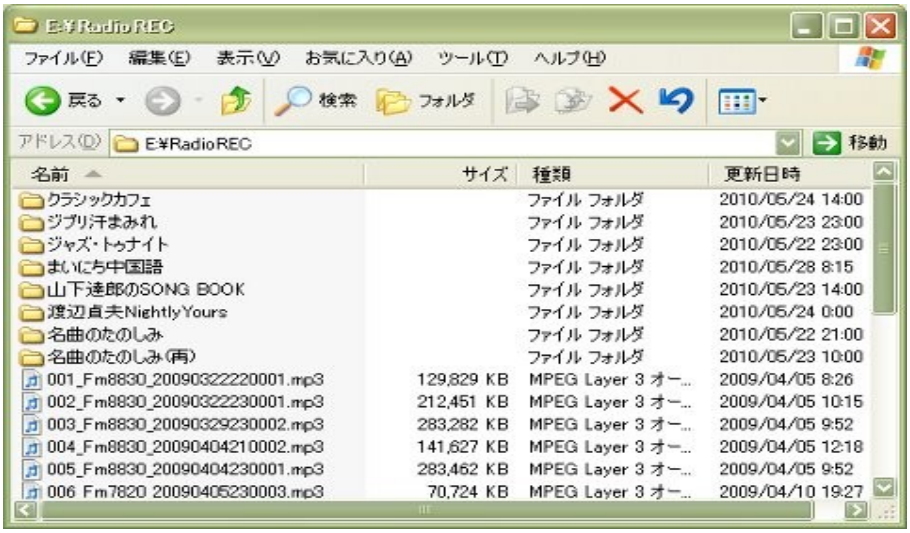


図　USB RADIO の録音ファイル保存先

3. どのプロトコルを使うか

ファイルにアクセスするためにどのプロトコルを使うかという問題がまずある。ファイルパスからアクセスするなら、ファイル・プロトコル(file protocol)を使うのが自然だが、セキュリティの問題から、Web ページに「file://ホスト名/ファイルパス」のリンクを記載してもリンクが機能しないようになっている。

Firefox では users.js を加えることによってセキュリティを維持しながらファイル・プロトコルを使えるようにするという記事も見かけるが、著者が試す限りは動作しない。

結論は、HTTP プロトコル(http protocol)を使う。

4. 日本語名フォルダの HTTP プロトコルによる表現

日本語のままでは HTTP プロトコルには載らない。URL エンコーディングが必要になる。日本語フォルダ名を Perl の readdir()で読み取った場合、文字コードは ShiftJIS(cp932)になっているのだろうが、確かめたこともない。

ファイル・プロトコルは Web ブラウザのアドレスに直接に入力した場合には動作する。例えば、日本語名フォルダにある録音ファイルをエクスプローラ(Explorer)から Firefox のアドレス欄にドラッグ・アンド・ドロップすると次のように表示される。

file:///E:/RadioREC/%E6%B8%A1%E8%BE%BA%E8%B2%9E%E5%A4%ABNightlyYours/20100208_000000.mp3

この状態で、Enter を入力すると、URL エンコーディングされた部分「%E6%B8%A1%E8%BE%BA%E8%B2%9E%E5%A4%AB」は「渡辺貞夫」と表示が変化して、Firefox 上で QuickTime が起動する。

日本語漢字 4 文字が 12 バイトのコードに表現されている。UTF-8 に自動的に変換されているようだ。これを HTTP プロトコルに変えればよいだけだ。

http://localhost:80/E:/RadioREC/%E6%B8%A1%E8%BE%BA%E8%B2%9E%E5%A4%ABNightlyYours/20100208_000000.mp3

ファイルプロトコルでは、ホスト名の localhost が省略されている。HTTP プロトコルでは省略できない。また、HTTP サーバー側の設定も必要である。

5. HTTP サーバーの設定

Apache の設定について説明しておく。次のような設定を httpd.conf に追加しよう。

```
<Directory "E:/RadioRec">
    AllowOverride None
    Deny from all
    Allow from 127.0.0.1 192.168.xx.x ...(アクセスを許可する PC の Address を設定)
    Order deny,allow
</Directory>
Alias /radio "E:/RadioREC"
```

ローカルサーバーであっても LAN で使う場合にはサーバーのディレクトリにアクセスできる PC を IP Address で登録しておくとセキュリティ上安心である。

また、具体的なフォルダ名でなく、簡単なアライアスを設定しておくと便利だしセキュリテイ性も高まるだろう。ここでは、E:/RadioREC に/radio でアクセスできる設定としている。

録音ファイルは MP3 の 128kbps のフォーマットで FM の 1 時間番組が 57MB 程度のサイズになる。著者の場合は、500GB の USB ハードディスクに録音ファイル保存先を設定している。

6. FM/AM エアチェック管理サーバー用 CGI

動作するかどうかを確かめるための最初の実験的実装なので、機能はシンプルである。

CGI は単に録音ファイル保存先にアクセスして、予約名単位で録音ファイルをリストアップし、HTTP プロトコルでリンクを生成出力する。CGI 出力するフレームを設定できる。

予約名からなる日本語名フォルダは ShiftJIS(cp932)としてデコードし、UTF-8 にエンコードして、さらに URL エンコーディングを URI::Escape モジュールで行う。

気になったのはファイルテスト演算子がディレクトリ名などを取得(grep)する際に有効に動作しないことだ。radiorecm.pl では、パターンマッチだけで予約名フォルダを”.”ディレクトリやファイル名と切り分けている。Perl は ActivePerl の v.5.10.0.1004 を使っている。

スクリプト自体は短く簡単なものである。前述したこと以外に特に難しいところはない。次に示す。

Perl スクリプト: radiorecm.pl

```
#!/Perl5.10/bin/perl.exe
```

```
use URI::Escape;
use Encode;

### Setting
my $server = "192.168.xx.x:xxxx";# host ip address and port number
my $radiorecdir = "E:/RadioREC";# Radio Recording Directory
my $target_frame = "status";# Target frame
###

opendir(DIR, $radiorecdir);
my @dmp3s = grep {/¥.mp3/i} readdir DIR;
rewinddir(DIR);
my @rdirs = grep {!/(^[.][.]?¥z|¥.mp3)/i} readdir DIR;
closedir(DIR);

print <<HEADER;
Content-type: text/html; charset=UTF-8

<htmL>
<body>
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<ul>
<li>$radiorecdir
HEADER

print "¥t<ul>¥n";
foreach my $dmp3 (@dmp3s){
      my $dmp3_url = "http://$server/radio/$dmp3";
      print "¥t<li><a href=¥"$dmp3_url¥" target=¥"$target_frame¥">$dmp3</a>¥n";
}
print "¥t</ul>¥n";

foreach my $rdir (@rdirs){
      my $encode_rdir = encode("utf8", decode("cp932", $rdir));
      print "<li>$encode_rdir¥n";
      my $esc_rdir = uri_escape($encode_rdir);
      opendir(DIR, "$radiorecdir/$rdir");
      my @rmp3s = grep(/¥.mp3/i, readdir(DIR));
      closedir(DIR);
      print "¥t<ul>¥n";
      foreach my $rmp3 (@rmp3s){
```

```
            my $rmp3_url = "http://$server/radio/$esc_rdir/$rmp3";
            print "¥t<li><a href=¥"$rmp3_url¥"
target=¥"$target_frame¥">$rmp3</a>¥n";
      }
      print "¥t</ul>¥n";
}

print <<FOOTER;
</ul>
</body>
</html>
FOOTER
```

次に CGI の動作画面を示す。著者お馴染みのデスクトップ CGI の画面だが、フレームは五つに分割される。左は items、memocalnder の 2 フレーム、右側は上から下に submain、main と status の 3 フレームに分割されている。CGI のファイル名のリストは submain フレームにセットしてある。録音ファイルへのリンクをクリックすると、status フレームで再生される。動作画面では、QuickTime が動作している。

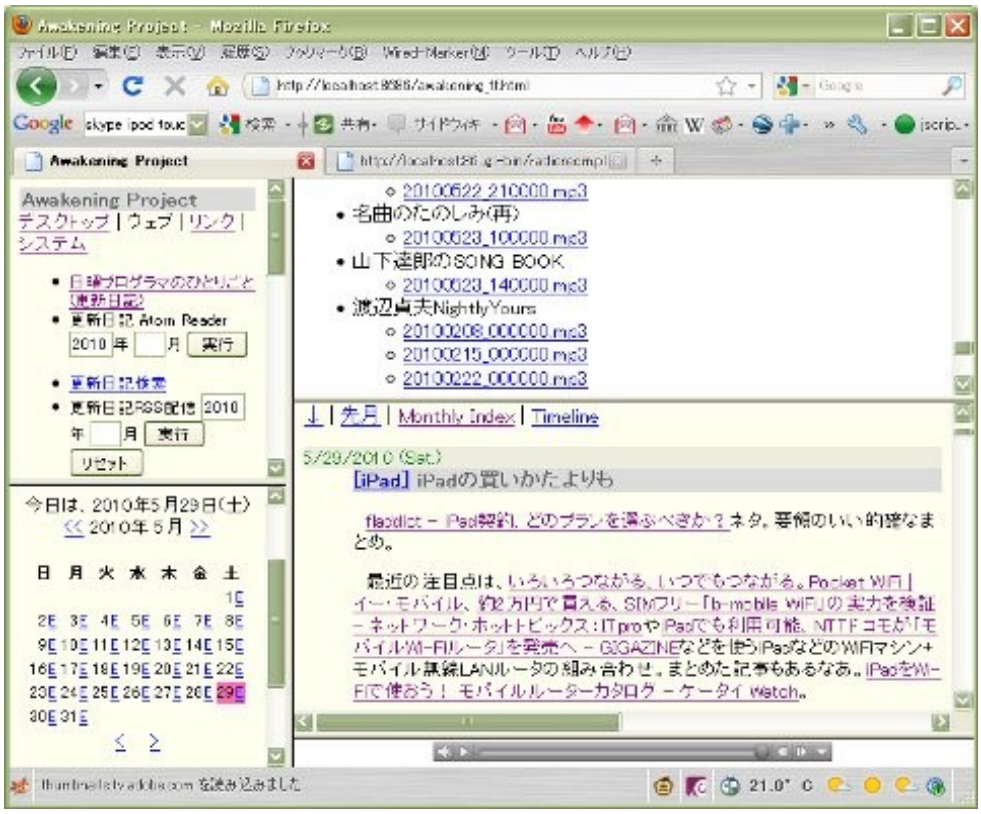


図　radiorecm.pl の動作画面

7. おわりに、そして次の構想

さて、日曜工作的プログラミングもハードウェアやソフトウェア環境が整ってくると広がりが出てくる。世の中の経済的激変にも関わらず大変楽しみなことである。

今回の FM/AM エアチェックサーバーに iPod touch からもアクセスして、Safari 上で録音ファイルを再生できる。PSP では、インターネットブラウザーから CGI にアクセスしてリストを表示し、リンクをクリックすると、MP3 ファイルを「/MUSIC」に保存するかどうかを聞いてくる。保存して、MP3 ファイルを XMB(クロスメディアバー)の「ミュージック」から選んで再生することができる。CGI のサーバーのホスト名を localhost にしていないのは他の PC やモバイルデバイスからのアクセスを想定しているからである。

Larry Wall が「これからは複数のシステムをつなぐ糊言語(glue language)が重要になる」というような意味のことを述べたことがある。Windows やその上で動作するアプリケーション、Web 上で動作するシステムを結び合わせると何ができるのか。我々はスクリプティング言語という万能ツールを手にしているのである。

FM/AM エアチェック管理サーバーも Web の番組やブログなどの情報と結びつけるとおもしろいものになるかもしれない。また、録音ファイルを RSS 2.0 で Podcast としてローカルに配信するようなことも考えられる。ブログの録音した番組の記事に特殊なタグを付けて、ローカルで録音ファイルを再生するようなこともできるだろう。

様々なアイデアが渦巻いている。実現するために必要なのは、本物の必要性と時間だけである。FM 番組についてブログでコメントする。そこに録音ファイルへのタグを残しておく。デスクトップ CGI はブログ記事と Podcast としての録音ファイルへのリンクを糊づけて、RSS 2.0 としてデスクトップサーバーに出力する。そんな感じにすれば汎用的なシステムとして統合可能だろう。

と言いながらも次はもっと別なネタで記事を書きそうな予感もしている。何かコメントをいただければその方向性に進むかもしれない。

なお、本稿で取り上げた USB RADIO については下記のリンクを参照のこと。ソフトウェアは録音レベルを最適化したものが、Ver.2.2 として出ている。売れているみたいだな。実際、なかなかよくできていると思う。

USB 接続 AM/FM ラジオ | SUNTAC
http://www.suntac-brand.jp/products/usb/rdpc101.html

楽しんでください。

(2010 年 5 月 30 日投稿受付)

ダイヤル(Aから)Mを廻せ！

written by Ｙさ

1. このゲームは

※タイトルはヒッチコック監督作品の映画の邦題から拝借していますが、内容は全く関係ありません...

"ダイヤル"を回して、全ての隣り合うダイヤルの上下左右の数字を揃えてください。

2. 動作環境は

例によって、最近のawkならOKだと思います。
ちなみに動作確認は、
GNU Awk 3.1.8, mawk 1.3.3 MBCS R27
で(WinXPのDOS窓で)行なってます。

3. 遊び方は

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

等で起動するとゲームスタートです。

1～4の数字が振られた"ダイヤル"が13個表示されます。
回す"ダイヤル"を'A'～'M'で指定し、左右のどちらに90° 回転させるかを'L','R'で指定してください。
例) AL (大文字小文字は区別しません)

"ダイヤル"は90° ずつしか回転させることができませんが、
上下左右方向の直線上に隣り合う他の"ダイヤル"があれば、歯車のように連動してそれらも回転します。

例)

```
  .2.     .2.     .2.
1:(A):3 1:(B):3 1:(C):3
  '4'     '4'     '4'
  .2.     .2.     .2.
1:(D):3 1:(E):3 1:(F):3
  '4'     '4'     '4'
```

↓ AR = "ダイヤル"A を右に 90° 回転

```
  .1.     .3.     .1.
4:(A):2 2:(B):4 4:(C):2
  '3'     '1'     '3'
  .3.     .2.     .2.
2:(D):4 1:(E):3 1:(F):3
  '1'     '4'     '4'
```

・"ダイヤル"A～C および D が連動して回転。
（隣り合う"ダイヤル"A～C および D の上下左右の数字が揃っている状態）
・"ダイヤル"E, F は連動して回転しない

隣り合うダイヤルの上下左右の数字を全て揃えると、得点を表示してゲーム終了します。
ゲーム開始時に表示する規定回数以内に揃えた場合はボーナス得点が加算されます。

なお、

```
  -v DIAL_PATTERN=1
  ~~~~~~~~~~~~~~~~~
```

といった感じでオプションを付けて起動すると、ダイヤルの配置パターンを変更できます。
数字は 1～9 を指定できます。デフォルトは 9 となっています。

※も少しヒントを出すと、phase() でダイヤルの「場所」と「揃っている状態の向き」を配列にセットしていますので、参考にして適当に要素を変更するのも有りかも。
d(^^)（パターン番号が大きい方が気持ち難しくなるように、色々細工している箇所もありますが...）

また、

```
 -v TRAINING_MODE=0
  ~~~~~~~~~~~~~~~~~~
```

のオプションを付けて起動すると、練習モードとなり、初期状態から好きなだけ"ダイヤル"を回して動きを実験することができるようにしてみました。

4.開発環境など

ソースは、awk 版として新規に作成したものです。ThinkPad の WinXP 上のテキストエディタとコマンドプロンプトな環境でやってます。
そういえば何時の間にやら GAWK のバージョンが上がっていますね...
ちなみに本作は、最初は前作「cube」の名前から"ルービックキューブ"っぽいものって

どうよ？ ってな感じで思いついて作り始めてみたのですが、気がつけば全く似ても似つかないものになっていましたとさ。

...しかもこのテキストを打ち込んでいる今現在は 5/31 のまもなく午前 4 時(^^; やれやれ

5. その他

本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。

AWK スクリプト: dial.awk

```
##  dial   written by Ｙさ

function rnd(N){ return int(N * rand()); }  ##  乱数

BEGIN{
  srand();

  # ゲームモード, 配置パターン決定
  if(TRAINING_MODE!=1) TRAINING_MODE=0;
  if(DIAL_PATTERN<1 || 9<DIAL_PATTERN) DIAL_PATTERN=9;
  Pattern=DIAL_PATTERN;

  # ゲーム準備
  initialize();

  # ゲームメイン
  sc=0;  # 得点
  times=0;
  do{ display(); which(); }while(notYet());
  exit;
}
END{
  display();
  sc += times + DIAL_PATTERN*3;
```

```
  if(limit>=times) sc += 10*limit + (limit-times)*100; # ボーナス加算
  printf("¥n¥n****** CONGRATULATIONS !! ******¥n¥n");
  printf("        (SCORE:%05d0)¥n", sc);
}

# ダイヤル初期配置データ
function phase(p,s)
{                       # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==1){ split("1, 2, 3, 4, 6, 9,11,14,16,17,19,21,24",p,",");
                  split("1, 3, 1, 3, 3, 1, 1, 3, 3, 1, 1, 1, 3",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==2){ split("3, 5, 6, 7, 8, 9,12,15,16,17,18,19,21",p,",");
                  split("1, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==3){ split("1, 2, 3, 6,11,13,14,16,19,21,22,23,24",p,",");
                  split("1, 3, 1, 3, 1, 1, 3, 3, 1, 1, 3, 1, 3",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==4){ split("2, 3, 7, 9,10,11,12,13,14,15,17,21,22",p,",");
                  split("1, 3, 3, 3, 1, 3, 1, 3, 1, 3, 3, 3, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==5){ split("0, 1, 3, 6, 8,11,12,13,14,17,19,22,24",p,",");
                  split("1, 3, 3, 1, 1, 3, 1, 3, 1, 3, 3, 1, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==6){ split("1, 2, 3, 4, 6, 9,10,11,14,15,18,19,23",p,",");
                  split("1, 3, 1, 3, 3, 1, 3, 1, 3, 1, 3, 1, 1", s, ",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==7){ split("1, 5, 6, 8, 9,11,12,13,15,16,18,19,23",p,",");
                  split("1, 1, 3, 3, 1, 1, 3, 1, 1, 3, 3, 1, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==8){ split("1, 2, 3, 5, 6, 8, 9,12,13,16,17,20,21",p,",");
                  split("1, 3, 1, 1, 3, 3, 1, 3, 1, 3, 1, 3, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
  if(Pattern==9){ split("2, 6, 7, 8,10,11,12,13,14,16,17,18,22",p,",");
                  split("1, 1, 3, 1, 1, 3, 1, 3, 1, 1, 3, 1, 1",s,",");
  }                     # 1] 2] 3] 4] 5] 6] 7] 8] 9]10]11]12]13]
}

# 各種初期化設定
function initialize(  n,p,s)
{
  # 上下左右の差分：dX[], dY[]
  # (上下左右の番号 = 1:上, 2:右, 3:下, 4:左)
  dX[1]=0;  dY[1]=-1;
```

```
  dX[2]=1;  dY[2]=0;
  dX[3]=0;  dY[3]=1;
  dX[4]=-1; dY[4]=0;

  # ゲーム盤：Board[]
  # 配置場所：Place[]
  # ダイヤルの向き：State[]
  phase(p,s);
  split("A,B,C,D,E,F,G,H,I,J,K,L,M", Kind, ",");
  for(n=0; n<5*5; ++n) Board[n]=State[n]=0;
  for(n=1; n<=13; ++n){
    Place[n] = p[n]+0;
    Board[Place[n]] = n;
    State[Place[n]] = s[n]+0;
    atoi[Kind[n]]=n;
  }

  # 規定回数(=ランダムにダイヤルを回した数)：limit
  if(TRAINING_MODE==1){
    print "¥n### TRAINING MODE ###¥n";
  }else{
    # 作業用に rndList[], rotateList[] を使う
    limit=makeRndList();
    for(i=0; i<limit; i++) turn(Place[rndList[i]],rotateList[i]);
    if(analyze(2)==0 && analyze(3)==0){
      # 少なくとも１箇所は変更する
      turn(Place[rndList[limit]],rotateList[limit]);
      ++limit;
    }
    printf("¥n### SOLVING MODE ###  (BONUS-DEADLINE:%dtimes)¥n¥n", limit);
  }
}

# ダイヤル表示
function display(  x,y,l)
{
  print "";
  for(y=0; y<5; ++y)
    for(l=0; l<3; ++l){
      for(x=0; x<5; ++x) dispDial(y,x,l);
      print "";
    }
  print "";
```

```
}
function dispDial(y,x,l,  p)
{
  if(State[(p=y*5+x)]==0){ printf("%*s", 8, ""); return; }
  if(l==0) printf("   .%d.  ",   number(1,p));
  if(l==1) printf(" %d:(%s):%d", number(4,p), Kind[Board[p]], number(2,p));
  if(l==2) printf("   '%d'  ",   number(3,p));
}
function number(dir,pos,  n)
{
  if((n=dir+State[pos]-1)>4) n-=4;
  return n;
}

# ランダムにダイヤルの向きを変える
# (作業用に rndList[], rotateList[] を使う)
function makeRndList(  list,num,i,prev,pcnt,r)
{
  delete rndList;  delete rotateList;
  num=pickUp(  0,3);
  num=pickUp(num,2);
  prev=0;
  for(i=0; i<num; ++i){
    if(prev==rndList[i]){ rotateList[i]=rotateList[i-1]; continue; }
    rotateList[i]=(rnd(1000)<500)?1:-1;
    prev=rndList[i];
  }
  for(i=0; i<num; ++i){
    r=rnd(num);
    swap(rndList, r, i);
    swap(rotateList, r, i);
  }

  # 同じダイヤルを3つ以上連続にしたりしないように調整
  prev=pcnt=0;
  for(i=0; i<num; ++i){
    if(prev!=rndList[i]){ prev=rndList[i]; pcnt=0; continue; }
    if(pcnt<1 && rotateList[i-1]==rotateList[i]){ ++pcnt; continue; }
    #詰める
    for(r=i+1; r<num; ++r){
      rndList[r-1]=rndList[r];
      rotateList[r-1]=rotateList[r];
    }
```

```
    --i; --num;
  }

  r = (Pattern<5)?(Pattern+1):6;
  r = (rnd(1000)<500)?r:(2*r);
  return (r>num-1)?(num-1):r;
}
function pickUp(num,dir,  list,v,l,r,cnt,i)
{
  for(v=0; v<5; ++v){
   l=(dir==3)?v:(5*v);
   if((cnt=getUse(l,dir,list))>0){
      r=rnd(cnt);
      for(i=((rnd(1000)<500)?1:2); i>0; --i) rndList[num++]=list[r];
    }
  }
  return num;
}
function getUse(l,dir,list,  n,p)
{
  delete list;
  n=0;
  p=find(dir,l);
  while(p>0){
    if(State[p]>0) list[n++]=Board[p];
    p=getNextPos(dir,p);
  }
  return n;
}
function swap(tbl,p1,p2,   t){ t=tbl[p1];  tbl[p1]=tbl[p2];  tbl[p2]=t; }

# 向きが揃っているかチェック
function notYet()
{
  if(analyze(2)==0 && analyze(3)==0){
    if(TRAINING_MODE!=1) return 0;

    display();
    print "** COMPLETE ! **¥n";
    which();
  }
  return 1;
}
```

```
function analyze(dir,  v,p,np,e)
{
  for(v=0; v<5; ++v){
    p=v*((dir==3)?1:5);
    do{
      if((p=find(dir,p))<0) break;
      while((np=getNextPos(dir,p))>=0 && State[np]>0){
        if((e=State[p]+2)>4) e-=4;
        if(e!=State[np]) return -1;
        p=np;
      }
      if(np<0) break;
      p=np;
    }while(p>=0);
  }
  return 0;
}
function find(dir,pos,  p)
{
  p=pos;
  while(State[p]==0) if((p=getNextPos(dir,p))<0) return -1;
  return p;
}
function getNextPos(dir,pos,  x,y,p)
{
  x=(pos%5) +dX[dir];
  y=int(pos/5) +dY[dir];
  if(x<0 || 5<=x) return -1;
  if(y<0 || 5<=y) return -1;
  if((p=(y*5+x))<5*5) return p;
  return -1;
}

# ダイヤル指定
function which(  p,r,tmp)
{
  printf("[#%d] Which Dial?>", ++times);
  do{ tmp=""; getline tmp; tmp=cnv(tmp); }while(cannotTurn(tmp));
  p=substr(tmp,1,1);
  r=(substr(tmp,2,1)=="L")?1:-1;
  turn(Place[atoi[p]],r);
}
function cannotTurn(src,  s)
```

```
{
  if(length(src)!=2) return 1;
  if(atoi[substr(src,1,1)]<1) return 1;
  s=substr(src,2,1);
  if(s!="L" && s!="R") return 1;
  return 0;
}
function cnv(src){ src=toupper(src); gsub(/[^A-MR]/,"",src); return src; }
function turn(pos,r,  d,t,p)
{
  rotate(pos,r);
  for(d=1; d<=4; ++d){
    p=pos;
    t=r;
    while((p=getNextPos(d,p))>=0 && State[p]>0){
      t*=-1;
      rotate(p,t);
    }
  }
}
function rotate(pos,r,  q)
{
  q = State[pos] +r;
  if(q<1) q=4;
  if(4<q) q=1;
  State[pos] = q;
}
```

(2010 年 5 月 31 日投稿受付)

編集後記

jscripter

TSNET スクリプト通信第 9 号は創刊二周年記念号となった。記念号を出せてよかったとほっとしている。

アマゾンの Kindle やアップルの iPad の登場とともに日本においては電子書籍元年と言われている。Web には様々な言説があるが、結局は読者にとって価値のあるものが残るということだろう。

私の予想はごく当たり前のことである。紙媒体による出版は印刷コストが掛かるからある程度の部数が売れる見込みがないと出版自体が行われない。しかしながら、電子出版は実質的に出版にコストが掛からない。少部数しか売れないものは電子出版しか選択肢はない。

もう一つの当たり前の予想は、紙媒体でも売れるものを電子媒体で販売するとその分だけ紙媒体の販売は減るために印刷の固定費が上がるということになる。電子書籍として売れて、そのほうが利益率が高いなら、選択は決まってくる。

紙の本も電子書籍もいずれにも便利な特徴がある。読者がどちらを選ぶかはおそらく価格設定が大きな役割をはたすのではないか。紙の本はコスト低減を工夫すると共にオンデマンド印刷に生き残る道を求めたい。紙の本が欲しい人もいることは間違いないので、電子書籍の導入によって紙の本の価格が高騰するのは悲しむべきことだろう。

印刷・紙の本の流通を除く、企画、デザイン、編集、広告などを含む出版の機能に価値がないわけではない。しかしながら、今度、700 円の特価で iPad 用電子書籍が販売されると噂の京極夏彦さんの小説「死ねばいいのに」（講談社） は、著者自らが InDesign で入稿されているのだという。著者自身がデザイナーであり、装幀家でもあるという。いくら才能のある人の試みであるとはいえ、この変革はコンピュータがもたらしたものである。流れが変わる大きな変革の時が訪れたと実感している。

TSNET スクリプト通信は読者層の開拓が課題であり、広告の部分が重要なのだろう。そして、本当の読者がどれだけいるのかを探らなければならない。

日曜プログラマ連絡協議会みたいなものを発足させようかと考えている。同志よ、集まれ^^;)/~~

(2010 年 5 月 30 日投稿受付)

ＴＳＮＥＴスクリプト通信

ISSN 1884-2798　出版地: 広島市

2010 年 5 月 31 日　　3.1.001 版
2010 年 5 月 30 日　　3.1.000 版(プレチェック版)

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)

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

著作権

使用許諾・配布条件

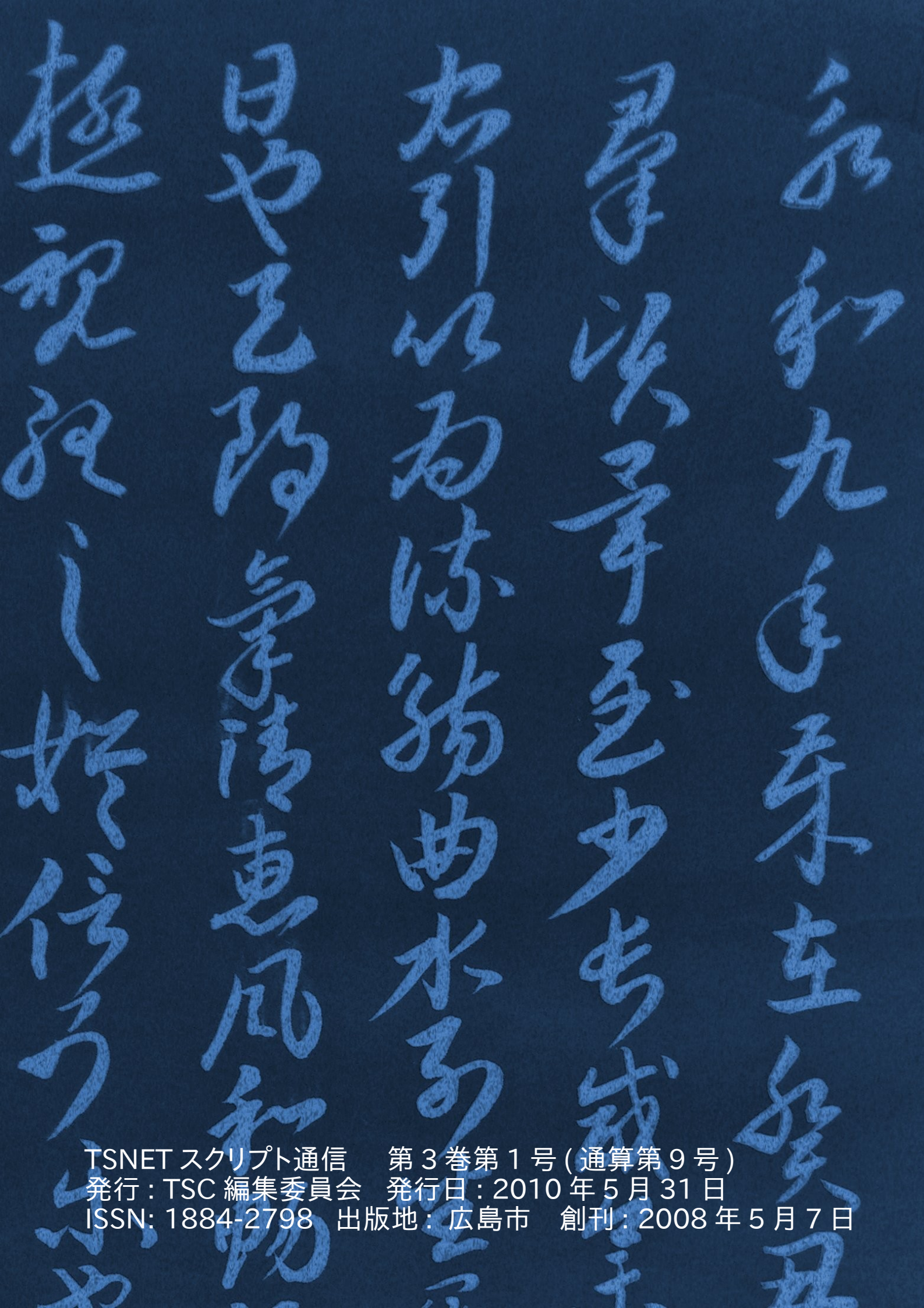
免責事項

編集ソフトウェア

OpenOffice.org 3.1.1 Writer

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１号 (通算第９号)
発行 : TSC 編集委員会　発行日 : 2010 年５月 31 日
ISSN: 1884-2798　出版地 : 広島市　創刊 : 2008 年５月７日