TSNET

スクリプト通信

Scripts Communication

March/2010

TSC編集委員会

ISSN 1884-2798

目次

巻頭言	jscripter	... 2
みんなのスナップショット	jscripter	... 3
Flex SDK と ActionScript 3.0 でゲーム	海鳥	... 5
Ruby/Tk でカレンダーを作ってみよう	ムムリク	... 18
Python の文法(仮)	機械伯爵	... 28
よしおさんとロボ太	海鳥	... 80
Zed 入門 III 「TAG 実行」を使ってみる	jscripter	... 84
Sjis.pm ミニ入門	jscripter	... 91
ゲーム Cube	Ｙさ	... 94
編集後記	jscripter	...103

巻頭言

jscripter

3月ももうすぐ終りますが、少し肌寒い日々が続いています。22日の振替休日に、筆の里工房の展覧会「絵職人 男鹿和雄の世界」を見に熊野に出かけたときは広島の黄金山の麓のほうはかなり咲いていました。もう春がやってきます。実はその前の週には広島県立美術館でムーミン展を見ました。隣合わせの縮景園には桜の蕾が弱々しく綻び始めていました。

さて、TSNETスクリプト通信は第2巻第4号、通算第8号を刊行することになりました。2月刊行予定が1月遅れましたが、機械さんの「Pythonの文法(仮)」という大作が投稿され、過去最高のページ数、第5号の80ページを超えることは疑いないでしょう。続編もあるので今後が楽しみです。Pythonの勉強はTSNETスクリプト通信で!

今号には久しぶりに海鳥さんからプログラムを投稿していただきました。Flex SDKでFlashのゲームを作ろうというものです。FlexのLinuxへのインストールからゲームの作成まで順を追って丁寧に説明していただいているので大変参考になるでしょう。ActionScriptはプロトタイプベースのJavaScriptに近いのかと思っていたらむしろクラスベースのJavaなのだそうです。やはりJavaを勉強しておいて損はなさそうだと思います。しかし、ただただ、何もしないままに時は過ぎ行く・・・jscripterがjava scripterになるのはいつの日か^^;)

もちろん、海鳥さんの「よしおさんとロボ太」も併載します。前号のロボ太の原点5編に続く7編を一挙掲載という大サービス。お忙しい海鳥さんは在庫が少なくなって少し慌て気味です。

ムムリクさんからは、Ruby/Tk活用シリーズ応用編「Ruby/Tkでカレンダーを作ってみよう」をご投稿いただきました。RubyでTkしてみようという方は、第7号の基礎編「Ruby/Tkしてみませんか」と合わせて、是非チャレンジしてください。

私は、Zedに新たに導入された「TAG実行」機能を使って、テキスト編集画面上のISBN番号からAmazon Webサービスの書籍情報を取り出してブラウザに表示させるというPerlスクリプトを「Zed入門 III」として紹介しました。さらに同じスクリプトをSjis.pmを使ってShift_JIS文字コードで書くとどのように書けばよいかについて「Sjis.pmミニ入門」として紹介します。

最後にYささんからは「ゲームCube」という視覚的には立体的ではない不思議なゲームが投稿されました。チャレンジしてみてください。

今号は表紙や各ページのヘッダに見ていただけるようにブックデザインにも取り組み始めました。デザインはmono966.org(モノクロームと読みます)に応援を頼んでいます。

これまでは文字だけの表紙は寂しいので、表紙に写真を飾っていましたが、写真投稿は継続して受け付けたいと思います。読者や著者の交流のための「みんなのスナップショット」というページを設けますので、お気軽に投稿ください。ペンネーム(もちろん、実名でも結構です)と一緒に一言添えてください。場所や撮影日がわかるといいですね。投稿は基本的にすべて掲載します。1024x768のサイズでご投稿ください。よろしくお願いします。今回は先陣を切って私が投稿しました^^)

それでは楽しんでください。

(2010月3月27日投稿; 2010年4月2日改訂)

みんなのスナップショット

jscripter の巻

3 月 20 日 13 時 47 分 原爆ドームと路面電車　広島市、相生橋上にて

2010年3月28日　広島市黄金山山頂にて

ほとんど咲いていない中でたくさんの花をつけたソメイヨシノ

Flex SDK と ActionScript 3.0 でゲーム
～ 完全無料で Flash のゲームを作ってみる ～

海鳥

0. 概要

無料の ActionScript 開発環境である Flex を使って簡単なゲームを作ってみる。

1. はじめに

これを読んでいる方なら当たり前かもしれないが、JavaScript と Java は全く違う言語だということを知らない人は多い。どちらもオブジェクト指向言語であるが、Java は強い静的片付けを持ったクラスベースの言語であり、JavaScript はプロトタイプベースの言語である。クラスベースとプロトタイプベースを厳密に区別することは難しいが、実際に使ってみると、まずクラスありきで考えるのか、インスタンスありきで考えるのかの設計哲学の差は結構感じる。C++などはクラスベース、Self などはプロトタイプベースである。普通、プログラム言語は中心的な設計哲学があり、言語は拡張されることはあってもその哲学が変更されることは少ない。その、途中で設計哲学が大きく変わった数少ない言語の一つがここで紹介する ActionScript である。

ActionScript は、もともと Adobe のフラッシュの作成の補助的な役割を担っていたスクリプトを言語的に精密化することで生まれた。最初は ECMAScript ベース、つまり JavaScript の兄弟のような言語であったが、後のバージョンアップでクラスベースに生まれ変わった。個人的な話になるが、僕は昔フリーソフトウェア作家をやっており、DOS や Windows でゲームをいくつか作っていた。しかし、ブラウザで手軽にできる Flash ゲームがその後広まり、フリーゲームをわざわざダウンロードして遊ぶ、ということは(よほどの大作でない限り)なくなった。

当然、それなら Flash でゲームを作ってやろうと ActionScript を見てみたが、言語仕様を見て「こりゃだめだ、これでゲーム作るのは僕には無理」とあきらめていた。ところがある時、言語仕様についての調べものをしていて「ActionScript がクラスベースになった」という噂を聞いた。見てみると、もうまるで Java そのもの。これならできそうだ、といろいろ調べてみると、開発環境である Flex SDK が無料で使えるという。実際にインストールしてちょっと触ってみると、あっというまにキー入力でキャラクタが動くところまでいったのにうれしくなり、その勢いで昔作ったゲームを Flash に移植することができた。なお、開発に使ったのは Linux 機であり、OS から開発環境からすべて無料のソフトウェアでブラウザゲームを作れることになる。

そんなわけで、本稿では ActionScript の言語仕様を簡単に紹介しつつ、Flex SDK で簡単なゲームを作ってみたいと思う。

2. 開発環境

2.1. Java 実行環境の確認

Flex SDK は Java で作られている。したがって、Java の実行環境がないといけない。Windows や Mac なら最初から Java の実行環境があるだろうが、Linux だと GNU の Java が入っている可能性が高い。その場合には、まず Sun の Java をインストールする必要がある。まず Java が入っているか、入っていたらどんな Java が入っているか確認しよう。

```
% which java
/usr/bin/java
```

```
% java --version
java version "1.4.2"
gij (GNU libgcj) version 4.1.2 20071124 (Red Hat 4.1.2-42)
```

このように表示されたらGNUのもので、これではFlexは動かない。SunのサイトからJavaを落としてこよう。

http://java.sun.com/javase/downloads/index.jsp

インストールしたあと、先にSunのJavaを見るようにパスを通しておこう。デフォルトなら /usr/javaに入るので、

```
export JAVA_HOME=/usr/java/default
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/jre/lib:$JAVA_HOME/lib:$JAVA_HOME/lib/tools.jar
```

などとすれば大丈夫。

2.2. Flex SDKのインストール

次にFlex SDKをダウンロードしよう。

http://www.adobe.com/cfusion/entitlement/index.cfm?e=flex3sdk

からダウンロードするか、以下のようにいきなりwgetしてしまっても良いだろう。

```
% wget http://download.macromedia.com/pub/flex/sdk/flex_sdk_3.zip
```

あとは適当なところに展開し、そこにパスを通しておくだけでインストール作業は終了。flex3というディレクトリに展開したなら、

```
export PATH=~/flex3/bin:$PATH
```

とでもしておけばよいだろう。コンパイラはmxmlcという名前のコマンドである。

コマンドプロンプトから

```
% mxmlc -help
```

と入力して、

```
Adobe Flex Compiler (mxmlc)
Version 3.3.0 build 4852
Copyright (c) 2004-2007 Adobe Systems, Inc. All rights reserved.
```

のような表示が返ってきたら正しくインストールされている。これで環境はととのった。

3. ActionScriptの文法

ActionScriptの言語仕様はほぼJavaなので、Javaを知っていれば特に苦労することなく使うことができるだろう。以下、Javaとの違いを中心にActionScriptの言語仕様を簡単に紹介する。せっかくなので、まずはお決まりのHello Worldを作ってみよう。

以下の内容のファイルを HelloWorld.as という名前で作成する。クラス名は大文字小文字を区別し、かつファイル名と一致しなければならない。Java と同じ仕様である。

```
package {
  import flash.display.*;
  import flash.text.*;
  public class HelloWorld extends Sprite {
    public function HelloWorld() {
      var tf:TextField = new TextField();
      tf.text = "Hello World!";
      addChild(tf);
    }
  }
}
```

できたらコンパイルしてみよう。

```
% mxmlc HelloWorld.as
```

ミスがなければ、HelloWorld.swf が作成されたはず。それをブラウザで開いてみよう。無事に Hello World が表示されていれば成功。

ファイルを構成する一番大きな要素はパッケージである。ほぼ Java のパッケージと同じと考えてよい。特に大きなプロジェクトでなければ、単に package {} で一つのクラスを囲んでしまってよいだろう。その中で、使うライブラリによって import をするのも Java と同じ。

クラス宣言も Java と同じだが、中で行われる変数宣言はかなり異なる。ActionScript では、変数(var)であるか定数(const)であるかをまず宣言する。次に識別子：型、という順番である。たとえば整数型の変数 i を宣言するなら、

```
var i:int;
```

とする。もちろん初期値の代入もできる。

```
var i:int = 0;
```

定数を宣言するには

```
const C:int = 0;
```

などとする。クラスのフィールドを宣言する場合には、さらに public か private も指定する。

```
private var field:int = 0;
```

クラスが一つしかないのであれば、とりあえずフィールドやメソッドは全て private にしてしまってよい。

また、細かい違いだが for 文のスコープが Java より広いのも注意しておきたい。C++や Java では

```
for(int i=0;i<N;i++){
  foo;
}
```

```
for(int i=0;i<N;i++){
  bar;
}
```

という書き方ができるが、ActionScript で

```
for(var i:in=0;i<N;i++){
  foo;
}
for(var i:int=0;i<N;i++){
  bar;
}
```

とすると、二回目のｉの宣言でエラーが出る。一つ目のｉのスコープがその後に及ぶからである。

これだけ理解すればもうあとは簡単。早速ゲームを作ろう。

4. ゲーム作り概観

4.1. ゲームの方針

ゲームを作るといっても、いきなり RPG やアクションは難しい。そこで、マウスのクリックだけでできる、UFO ゲームを作ろう。UFO ゲームとは、ボタンを押している間は上に加速し、離していると下に落ちていく UFO を操縦して、障害物を乗り越えていくゲームである。アーケードゲームの黎明期に誕生した、非常にプリミティブなゲームだが、ゲームの三要素(描画、入力、タイマー制御)を備えている。これが作れたら本格的なゲーム作りに進むのも容易であろう。

そこで、線の描画だけで画面を構成し、マウスだけの入力により線が右に右に進みながら障害物を越えていくゲームを作る。安直だが Lines という名前をつけておこう。クラス名は Lines、ファイル名は Lines.as とする。

4.2. プログラムを作る前に

これからゲームを作るが、その前にやらなければならないことがある。それは環境設定である。作られるフラッシュがどのようなプロパティを持つかをコンパイラに指示しなくてはならない。その指示するファイルは、

クラス名-config.xml

という名前で、メインのクラスと同じディレクトリにおいておく。今回の場合なら Lines-config.xml とする。たとえば以下のような内容である。

```
<flex-config>
<output>Lines.swf</output>
<default-size>
  <width>320</width>
  <height>340</height>
</default-size>
<default-background-color>0x000000</default-background-color>
<use-network>false</use-network>
```

```
<benchmark>true</benchmark>
<compiler>
 <incremental>true</incremental>
</compiler>
<metadata>
 <title>Lines</title>
 <description></description>
 <publisher></publisher>
 <creator>Kaityo</creator>
</metadata>
</flex-config>
```

この中で一番重要なのはoutputとdefault-sizeの項目である。まずoutputで指示されたファイル名でswfが出力される。ここではLines.swfとしておこう。

default-sizeは、そのサイズでブラウザで表示したときに等倍で表示されるサイズのことである。このサイズを正しく指定しておかないと、ブラウザで表示されなかったり、縦横比がおかしくなったりする。

残りの項目はあまり重要でない。compilerのincrementalをtrueにしておくと、初回コンパイル時にキャッシュファイルを作り、次から部分コンパイルをするためにコンパイルが高速になる。metadataも、もし指定したければ指定しておいても良い。全体をflex-configタグで囲っておくこと。

コンパイルしたらLines.swfができるので、そのswfを表示するHTMLファイルも作っておこう。たとえば以下のような内容となる。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="ja">
<head>
<title>
Lines
</title>
</head>
<div style="text-align:center">
<H1>Lines</H1>
<embed src="./Lines.swf" type="application/x-shockwave-flash" width="320"
height="340" bgcolor="#000000"
pluginspage="http://www.adobe.com/go/getflashplayer_jp"/>
</div>
<hr>
</body>
</html>
```

ここでも、もっとも重要なのはwidthとheightの指定である。これを先ほど作成したLines-config.xmlと整合する値にしておかなければ、表示がおかしくなる。

さて、これであとはゲームを作るだけである。

4.3. ゲームデザイン

穴が開いた棒が右から左にスクロールしてくるのを次々と潜り抜けるゲームとする。「自機」は赤い線、「敵(障害物)」は水色の線として、水色の線にぶつかったり、一番上や下にぶつかったりしたらアウトとする。単に潜り抜けるだけでは面白くないので、ぎりぎりですり抜けたら高得点としよう。具体的には、水色の線の穴のふちに黄色い短い線が

あり、そこを触るとボーナスで、連続して触り続ければコンボボーナスをつけることにする。

「自機」の操縦はマウスクリックのみ。マウスをクリックしている間は上に加速、離していれば重力で下に加速する。これは内部で速度を保持しておいて、マウスクリックで下がることにする。

スタートボタン、スコア、コンボ数、ハイスコアはテキスト(TextField)で表示することにすれば、プログラムで必要なのはテキストと線の描画、そしてマウスイベント処理のみである。

5. ゲームの処理の実際

5.1. クラスの外形とテキストフィールド

実際に組んだコードを最後に添付するので、それを見ながら処理を確認していこう。まず、Lines クラスを作る。これはSpriteクラスから派生させる。イベント関連のインポートimport flash.events.*;などもしておく。

ActionScriptでテキストを表示するには TextFieldクラスを使う。後でテキストの内容をいじるので、Spriteクラスのフィールドとして宣言する。たとえば得点を表示するためのテキストフィールドは

```
private var tfScore:TextField = new TextField();
```

と宣言されている。このテキストフィールドを、Linesクラスに「追加」して、表示されるようにするためにはSpliteクラスのメソッド、addChildを使う。LinesはSpriteクラスのサブクラスなので、当然そのままaddChildが使える。

テキストフィールドの位置は、たとえば

```
tfScore.x = 10
tfScore.y = 320
```

などと指定できる。テキストはtext、色はtextColor、背景色はbackgroundColorなど、ほとんどそのまんまなので見てわかるだろう。selectableは、マウスで選んでクリップボードにコピーなどできるようにするか決めるプロパティで、今回はfalseにしておく。今回のコードでは、共通する設定項目が多いので、addTextFieldというメソッドを作ってそこでaddChildを行っている。

5.2. 線の描画

線の描画もほぼJavaと同じで、 graphics.moveToやgraphics.lineToを使えば良い。ただし、Spriteクラスに直接線を描いていくと、どんどん処理が重くなるので注意が必要である。これは、Spriteクラスが「どんな線を描いたか」を全て覚えている、つまり内部的にベクタデータで保持しているためである。この機能のおかげで拡大、縮小しても線がぎざぎざになったりしないのだが、ゲームの用途には向かない。
そこで、描画用にBitmapDataクラスを用意して、そこにラスタライズした線を描いていくことにする。具体的には、

(1) Shapeクラスを作り、
(2) lineStyleで色などを設定し、
(3) moveTo、lineToでShapeクラスに描画し、

(4) BitmapData クラスの draw メソッドでラスタライズする

という段階を経る。本来なら全ての要素をスプライトで保持、管理して、当たり判定もスプライトのメソッドを使うのがフラッシュ作成の正しい方針なのであるが、とりあえずプログラムもレトロな感じでいくことにしよう。そのために、あらかじめ BitmapData クラスのインスタンスを作っておき、それを Lines クラスのコンストラクタで addChild してある。あとは BitmapData で draw すればそれが画面に反映される。

5.3. イベント処理

イベントは addEventListener によりリスナを登録する。これも Java 使いにはおなじみであろう。C 使いにはコールバック関数と言ったほうが通りが良いかもしれない。TextField などにもリスナを登録できるので、これでスタートボタンを作ることにしよう。具体的には、Lines クラスに onClick というメソッドを作っておき、tfStartButton:TextField にそのメソッドを

```
tfStartButton.addEventListener(MouseEvent.CLICK,onClick);
```

としてリスナ登録をする。第一引数が処理するイベントの種類、第二引数がイベントリスナである。これでスタートボタンをクリックしたら onClick が呼ばれるので、その中に処理したいことをかけばよい。今回は、クリックしたらゲームの初期化処理とタイマー処理をするようにしてある。

5.4. タイマー処理

単純なアドベンチャーや RPG ならともかく、シューティングやアクションなど、時間の流れがあるゲームにはタイマー処理は必須である。DOS のゲームなどでは割り込みなどを使ったりして相当面倒だったが、ActionScript を使えばすこぶる簡単にできる。

Sprite クラスは setInterval というメソッドを持っており、その引数にタイマーイベントのリスナが登録できるので、その中で処理を書けばよい。具体的には onTimer というメソッドを用意しておき、コンストラクタで

```
setInterval(onTimer,50);
```

とすれば、50 ミリ秒に一度 onTimer 関数が呼ばれるようになるので、onTimer 関数の中でゲームの処理を行えばよい。

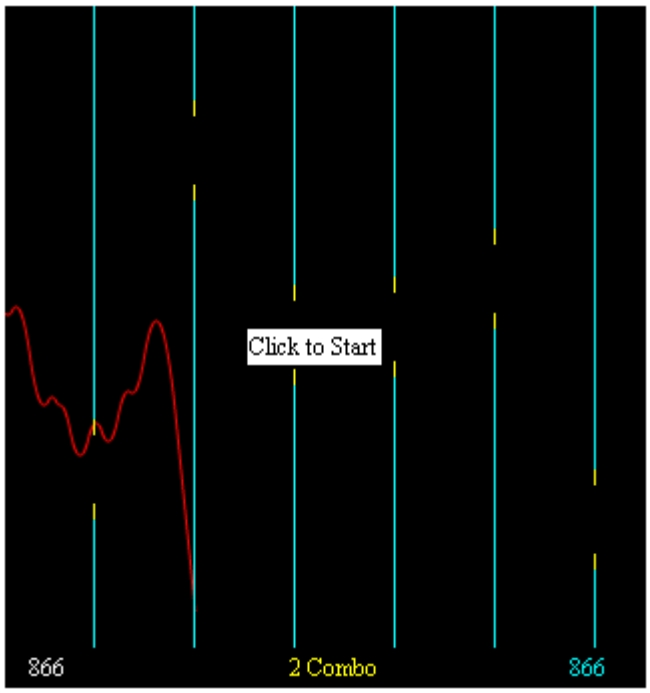
本当はゲームプレイ中だけタイマーイベントを起こすようにするのが良いのだが、今回は手抜きでずっとタイマーイベントを発生させておき、onTimer の内部で

```
if(GS_PLAYING == gameState){
   drawAll()
}
```

と、いまゲームプレイ中かどうかを判定し、プレイ中なら drawAll()を呼ぶようにしている。drawAll はゲームを一コマ進めるメソッドである。敵にあたるなどしてアウトになったら、gameState を GS_STOP に設定することで、drawAll が呼ばれなくなる(GS_PLAYING も GS_STOP もここで勝手に定義した定数)。

5.5. ゲームの遊び方

こうして作成したゲーム画面は以下の通り。



赤い線が「自機」、水色の線が「敵」、黄色い線が「ボーナスゾーン」、画面下に得点その他、画面真ん中に「スタートボタン」である。スタートボタンは最初とアウトになった直後に表示し、ゲーム中は非表示としておく。

最初に「Click to Start」と表示されているテキストをクリックするとゲームが始まる。マウスを押している間は上に、離している間は下に下がっていくので、水色の線にさわったり、一番上や一番下に触ったりしないように進んでいこう。どんどん得点は増えていくが、単に穴をくぐるだけでは高得点は狙えない。基本は1ドット進むごとに1点、穴をくぐれば100点だが、水色の線の穴に黄色く短い線があり、そこは触ってもミスにならないだけでなく、連続して触れば「コンボ」となる。0 Comboに対して、1 Comboは得点の上がり方が2倍、2 Comboは4倍・・・と、コンボをつなげればつなげるほど得点の増え方が激しくなる。黄色い線を触らないとコンボは下がってしまうので、なるべく高いコンボを維持することで高得点を目指そう。

6. 終わりに

以上、非常に駆け足であるが、ActionScriptでのゲーム作りの概要を説明した。非常にプリミティブで、サウンドもBGMもないが、それでもたった200行程度でちゃんとブラウザで遊べるゲームが作れてしまう。しかも、OSがLinux（僕が使ったのはCentOS)、開発環境 Flex SDK、それを動かすためのJavaと、全てフリーでそろえることができる。ブラウザで動くゲームなので、ブラウザさえフラッシュに対応していればWindowsだろうがMacだろうがLinuxだろうがどこでも遊ぶことができる。そしてJavaさえ動けば、WindowsだろうがMacだろうがLinuxだろうが、どこでもゲームを開発することができる。改めてすごい時代になったものである。

さて、以下はまったく個人的な歴史認識なので、間違いがあるかもしれない。僕はさほど年寄りではないと思っているが、インターネットユーザの中では最古参の一人である。まだブラウザがMosaicしかなく、端末がUNIXだった時代からインターネットを触っていた。CGIやSSIを作ったのもかなり早い部類だと思う。そのうち、Netscapeができ、IEができ、Yahooが、そしてGoogleが誕生、インターネットは飛躍的な発展を遂げ、いまでは携帯からネットにつながる時代となった。

そんななか、かなり早い段階からブラウザ上でアプリケーションを動かそうという動きがあった。Sun の Java と Java Applet である。詳しくはわからないが、Sun は Java Applet に相当自信を持っていたようだ。Applet によりスタンドアローンのアプリケーションは必要なくなり、全てブラウザ上で仕事が済んでしまう、そういう時代がくると思っていたふしがある。

しかし、Java Applet は非常に重かった。昔 Java Applet で業務用のアプリケーションを組む仕事をアルバイトでやったことがあるが、なんでもできるかわり、ちょっと複雑なことをやろうとすると、すぐに IE ともども落ちてしまい、ひどいときには Windows ごと固まってしまうこともあった。そんな中、動作が軽い Flash や ShockWave が誕生し、Applet は完全に駆逐されてしまった(一部業務用にはまだ使われているようだが)。

ところが今、その Flash は ActionScript により作ることができるようになった。その SDK は Java で作られているため、Java さえ動けばどんなプラットフォームでも Flash アプリが作れるようになった。Java のスローガン「Write once, run anywhere」の面目躍如といったところである。Java Applet を駆逐した Flash が、Java によってコンパイルされるという事実を、歴史の皮肉と見るか、最終的な Java の勝利と見るかは人によって違うであろうが、単純に面白い。

今後インターネット、そしてプログラム言語がどんな発展を遂げるか楽しみである。

7. ソースコード

以下を Lines.as として保存し、本文中の Lines-config.xml と同じディレクトリにおいて

```
% mxmlc  Lines.as
```

とすれば Lines.swf ができる。ブラウザで lines.html を開けばゲームができるはずである。

改造、改良、再配布自由にしていただいてかまわないので、より面白いゲームを作って欲しい。

[Lines.as]

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.system.*;
  import flash.utils.*;
  import flash.geom.*;
  import flash.events.*;
  public class Lines extends Sprite {

    private const WIDTH:int = 320;
    private const HEIGHT:int = 320;
    private var canvas:BitmapData = new BitmapData(WIDTH,HEIGHT,true,0);

    private var tfScore:TextField = new TextField();
    private var tfHighScore:TextField = new TextField();
    private var tfCombo:TextField = new TextField();
```

```
    private var tfMessage:TextField = new TextField();
    private var tfStartButton:TextField = new TextField();

    private var myHeight:int = HEIGHT/2;
    private var myAcceleration:int = 0;
    private const MAX_ACCELERATION:int = 10;
    private var myMouseDown:Boolean = false;
    private const LINE_SIZE:int = 96;
    private var pastHeight:Array = new Array(LINE_SIZE);

    private const MAX_HOLE:int = 10;
    private var holeHeight:Array = new Array(MAX_HOLE);
    private var holePosition:Array = new Array(MAX_HOLE);
    private var holeSize:Array = new Array(MAX_HOLE);
    private const HOLE_GAP:int = 50;
    private const COMBO_GAP:int = 8;

    private var gameState:int = GS_STOP;
    private const GS_STOP:int = 0;
    private const GS_PLAYING:int = 1;

    private var score:int = 0;
    private var highScore:int = 0;
    private var combo:int = 0;

    public function Lines() {
      addChild(new Bitmap(canvas));
        addTextField(tfScore,10,320,"0",0xFFFFFF);
        addTextField(tfCombo,140,320,"0 Combo",0xFFFF00);
        addTextField(tfHighScore,280,320,"0",0x00FFFF);
        addTextField(tfStartButton,120,160,"Click to Start",0x000000);
      tfStartButton.backgroundColor = 0xFFFFFF;
      tfStartButton.addEventListener(MouseEvent.CLICK,onClick);
      setInterval(onTimer,50);
      this.stage.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
      this.stage.addEventListener(MouseEvent.MOUSE_UP,onMouseUp);
    }

    private function addTextField(tf:TextField, x:int, y:int,
text:String,c:uint):void{
      tf.autoSize = TextFieldAutoSize.LEFT;
      tf.selectable = false;
      tf.border = true;
      tf.background = true;
        tf.backgroundColor = 0x000000;
      tf.text = text;
        tf.x = x;
      tf.y = y;
        tf.textColor = c;
        addChild(tf);
      }

    private function init():void{
      score = 0;
      combo = 0;
```

```
    myHeight = HEIGHT/2;
    myAcceleration = 0;
    var i:int = 0;
    for(i=0;i<LINE_SIZE;i++){
      pastHeight[i] = myHeight;
    }
    for(i=0;i<MAX_HOLE;i++){
      holePosition[i] = WIDTH/2+i*HOLE_GAP;
      holeSize[i] = 50;
      holeHeight[i] = int(Math.random()*(HEIGHT-holeSize[i]));
    }
  }

  private function onTimer():void{
    if(GS_PLAYING == gameState){
      drawAll();
    }
  }

  private function drawAll():void{
    canvas.fillRect(new Rectangle(0,0,WIDTH,HEIGHT),0x0);
    drawMyLine();
    drawHoles();
    score = score + Math.pow(2,combo);
    tfScore.text = score.toString();
    tfCombo.text = combo.toString() + " Combo";
    if(isDead()){
      if(score>highScore){
        highScore = score;
        tfHighScore.text = highScore.toString();
      }
      gameState = GS_STOP;
      tfStartButton.visible = true;
    }
  }

  private function isDead():Boolean{
    if(0 == myHeight) return true;
    if(HEIGHT == myHeight) return true;
    for(var i:int =0;i<MAX_HOLE;i++){
      if(LINE_SIZE==holePosition[i]+2){
        if(myHeight < holeHeight[i])return true;
        if(myHeight > holeHeight[i]+holeSize[i])return true;
        //combo check
        if(myHeight < holeHeight[i]+COMBO_GAP ||
           myHeight > holeHeight[i]+holeSize[i] - COMBO_GAP){
           combo++;
        }else{
          combo--;
          if(combo<0){
            combo = 0;
          }
        }
        score = score + Math.pow(2,combo)*100;
      }
```

```
      }
      return false;
    }

    private function drawMyLine():void{
      myHeight = myHeight + myAcceleration;
      if(myHeight<0){
        myHeight = 0;
      }else if(myHeight > HEIGHT){
        myHeight = HEIGHT;
      }
      if(myMouseDown){
        myAcceleration--;
      }else{
        myAcceleration++;
      }
      if(myAcceleration >MAX_ACCELERATION){
        myAcceleration = MAX_ACCELERATION;
      }else if(myAcceleration < -MAX_ACCELERATION){
        myAcceleration = -MAX_ACCELERATION;
      }
      var i:int = 0;
      for(i=0;i<LINE_SIZE-1;i++){
        pastHeight[i] = pastHeight[i+1];
      }
      pastHeight[LINE_SIZE-1] = myHeight;
      var s:Shape = new Shape();
      s.graphics.lineStyle(1,0XFF0000);
      s.graphics.moveTo(0,pastHeight[0]);
      for(i =1;i<LINE_SIZE;i++){
        s.graphics.lineTo(i,pastHeight[i]);
      }
      canvas.draw(s);
    }

    private function drawHoles():void{
      var i:int = 0;
      for(i=0;i<MAX_HOLE;i++){
        holePosition[i]--;
      }
      if(holePosition[0] < 0){
        for(i=0;i<MAX_HOLE-1;i++){
          holePosition[i] = holePosition[i+1];
          holeHeight[i] = holeHeight[i+1];
          holeSize[i] = holeSize[i+1];
        }
        holePosition[MAX_HOLE-1] = holePosition[MAX_HOLE-2] + HOLE_GAP;
        holeSize[MAX_HOLE-1] = 50;
        holeHeight[MAX_HOLE-1] = int(Math.random()*(HEIGHT-holeSize[MAX_HOLE-
1]));
      }
      var s:Shape = new Shape();
      s.graphics.lineStyle(1,0X00FFFF);
      for(i=0;i<MAX_HOLE;i++){
        s.graphics.moveTo(holePosition[i],0);
```

```
        s.graphics.lineTo(holePosition[i],holeHeight[i]);
        s.graphics.moveTo(holePosition[i],holeHeight[i]+holeSize[i]);
        s.graphics.lineTo(holePosition[i],HEIGHT);
      }
      canvas.draw(s);
      s.graphics.lineStyle(1,0XFFFF00);
      for(i=0;i<MAX_HOLE;i++){
        s.graphics.moveTo(holePosition[i],holeHeight[i]);
        s.graphics.lineTo(holePosition[i],holeHeight[i]+COMBO_GAP);
        s.graphics.moveTo(holePosition[i],holeHeight[i]+holeSize[i]-COMBO_GAP);
        s.graphics.lineTo(holePosition[i],holeHeight[i]+holeSize[i]);
      }
      canvas.draw(s);
    }

    private function onClick(e:MouseEvent):void{
      if(GS_STOP == gameState){
        init();
        gameState=GS_PLAYING;
        tfStartButton.visible = false;
      }
    }

    private function onMouseDown(e:MouseEvent):void{
      myMouseDown = true;
    }

    private function onMouseUp(e:MouseEvent):void{
      myMouseDown = false;
    }
  }
}
```

Ruby/Tk でカレンダーを作ってみよう

ムムリク

仕事でプログラムをしていたり、プログラムが趣味として染み付いてしまっているような人は別として、まったく未経験の人が「プログラムってのもなんだか面白そうだね」と思い、プログラムを始めてみようとしてみても、なにから手をつけてよいのか悩んでしまい、入門書を読んでみたものの、なんだかそこで終わってしまうということは案外ありがちなことかもしれません。

はじめるきっかけとして、恐らく有望かと思うのは、なにか作りたいと思うものがあるときかもしれません。それがあまりに複雑で高度なものを夢見てしまった場合には、早々と挫折するかもしれませんが、そこそこの目標を持ったとしたら、段階を追って拡張していくことで達成感と同時にある程度の知識も身についてくるということはあると思います。

なにより、どんなことができるのかがよく分からない段階でもあり、なにを作っていいか分からない状態では、ただ入門書をなぞるだけで終わってしまうというところかも。

もちろん、その入門書を発展させることができれば、それもひとつの方法ですが、いずこからかテーマをもらってきて取り組んでみるというのもひとつの方法かもしれません。さらにいえば、日々 PC を使うなかで、自分が不便に感じる操作や処理を自動化したり肩代わりさせるようなものができないかと考えることも、きっかけのひとつになるのでは。

もちろん、このご時世ですからネットを探せば希望するプログラムはそれこそ掃いて捨てるほど見つかるかもしれません。けれども自分で作ることの楽しさと、その自由度を一度知ると「それはそれとして」と思えることも少なくないはずです。

今回は、そんなテーマの提案のひとつとしてカレンダーを作ってみましょう。

（ここでは、ruby 1.8.7 (2010-01-10 patchlevel 249) [i386-mswin32] by ASR、ActiveTcl 8.5.7.0 を使用しています。Ruby 1.9.x でも動作はしますが、警告が逐一でるために動作はやや緩慢になります。）

まずは型枠をつくる

カレンダーをイメージして思いつくのは、「 1　2　3　4　5　6 」といったように、週ごとの数字を並べた列を順に表示するといったものかと思います。日々目にするカレンダーの形です。

ただ、月が変わると曜日との位置が変わってくるので、数字の位置合わせなどが面倒になりそうです。

そこで、よくある書き込み式のカレンダーような一日ずつ枠に囲まれたイメージで数字のはいる場所をわけてしまうことにします。そうすればそれぞれの位置の数字を書き換えることで月の変更をすることもできるはずです。

こうした座標管理のためのライブラリなどもあるようですが、まずはそうしたものは使わずに作ってみることにしましょう。（格子状の配置を管理する grid がそのものずばりです）

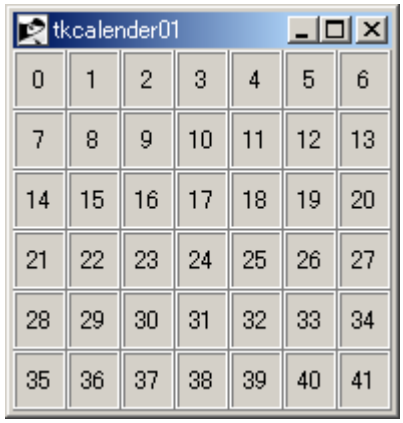
まずは、日付数字のはいる枠を作ってみます。

[tkcalender01]

```
require 'tk'

f = Array.new(6)
l = Array.new(42)

6.times do |x|
  f[x] = TkFrame.new.pack('side' => 'top')
  7.times do |y|
    z = 7 * x + y
    l[z] = TkLabel.new(f[x]) {
                              text z
                              relief 'ridge'
                              height 2
                              width 3
                              pack('side' => 'left')
                             }
  end
end
Tk.mainloop
```



カレンダーは 1 日の曜日如何によって最大で 6 週間分必要になります。そこでフレームを 6 つ分、曜日は 7 つですから 6 * 7 = 42 の日付の収まる場所を用意します。relief は縁取りの効果を指定するもので、ほかに raised（凸）、sunken（凹）、flat（平坦）、solid（線枠）、groove（溝枠）があります。

42 個のラベルを用意して、そこに表示する日付の数字を保存することにします。後から変更しやすいようにあらかじめラベルひとつひとつに l という配列を用意します。配列の添え字は 0 からはじまります。ここでは添え字をそのまま表示させているので 0 から 41 までの数字が並んでいます。

この配列に日付の数字を順に設定しておいてひとつずつ読み取って表示するようにします。たとえば l => [1, 2, 3, 4, 5, ...] といった並びだったとすると、先の図の 0 の場所に「 1 」が、1 の場所に「 2 」がといったように表示されることになります。仮に、l => ['', '', 1, 2, 3, ...] といった並びだったとすると、0 1 の場所に

は数字は表示されず、2 の場所に「 1 」が表示され、以降順次続きます。

日付を表示させる

では、実際に日付を設定してみましょう。

まず、実行したときに表示されるカレンダーは当月のものとします。当月の年月はそれぞれ次のようにして求めることができます。

```
Date.today.year => 今日の年
Date.today.month => 今日の月
```

カレンダーを作るにあたって欲しい情報としては、さらにその月の日数（大の月、小の月、うるう月など）、一日の曜日があります。それらは次のようにして求めることができます。

```
Date.new(年数, 月数, -1).day => 月の日数
Date.new(年数, 月数, 1).wday => 一日の曜日
```

Ruby の Date による曜日は、日月火水木金土の順に 0, 1, 2, 3, 4, 5, 6 という値を返します。日本式な日曜日から一週間を表示するカレンダーには、これがうってつけの効果をもたらしてくれます。

先ほどの図の数字の並びを思い出すと、一行目がそのまま同じであることがわかります。つまり表示したい月の一日の曜日を求め、その曜日の数字と同じ添え字のラベルから順に数字を入れていけばよいことがわかります。

ひとまず別の配列を用意して、そこに日付イメージを格納してみましょう。

```
p_fig = Array.new(42, '')
days = Date.new(p_year, p_month, -1).day
i = Date.new(p_year, p_month, 1).wday
(1..days).each do |d|
  p_fig[i] = d
  i += 1
end
```

これをつかって日付を表示するように修正します。

[tkcalender02]

```
require 'tk'
require 'date'

p_year = Date.today.year
p_month = Date.today.month

p_fig = Array.new(42, '')
days = Date.new(p_year, p_month, -1).day
i = Date.new(p_year, p_month, 1).wday
(1..days).each do |d|
  p_fig[i] = d
  i += 1
end

f = Array.new(6)
l = Array.new(42)

6.times do |x|
  f[x] = TkFrame.new.pack('side' => 'top')
  7.times do |y|
    z = 7 * x + y
    l[z] = TkLabel.new(f[x]) {
                              text p_fig[z]
                              relief 'ridge'
                              height 2
                              width 3
                              pack('side' => 'left')
                             }
  end
end

Tk.mainloop
```

それらしい感じにはなりましたが、やはり表示している年月があったほうがいいですね。日付の上の部分に追加することにしましょう。

```
fm = TkFrame.new('height' => 2).pack
lm = TkLabel.new(fm, 'height' => 2).pack
lm.text = "#{p_year} 年 #{p_month} 月"
```

```
f = Array.new(6)
```

の前に追加します。

月の移動を可能にする

さて、このままでは今月分しか見ることができません。せめて前後に移動できるボタンをつけてみましょう。

最後の行の、

```
Tk.mainloop
```

の前に以下を追加します。

```
fb = TkFrame.new.pack('side' => 'bottom', 'fill' => 'x')
TkButton.new(fb) {
              text "前の月"
              command proc {
                           }
              pack('side' => 'left')
              }
TkButton.new(fb) {
              text "次の月"
              command proc {
                           }
              pack('side' => 'right')
             }
```

まだ具体的なボタンの処理はいれてありません。画面のイメージは図のような感じです。

ここで、それぞれの command proc { ＊ } の＊部分に処理を書けばよいわけですが、ふたつの処理はほぼ同じようなことをするということが想像されます。年月をひとつ減らすか、ひとつ増やすかして、その年月から日付を表示しなおすという処理になりそうです。とすると、年月を増減するところは別としても、その数字を使って表示しなおすのは同じ処理で済みます。そしてそれは先に表示させたものと同じ処理です。それぞれにその処理を書くことでもちろん動作するはずですが、同じものが複数箇所にあるのはなんとも無駄な感じです。できればひとつにまとめたいものです。

それを踏まえて全体を見直してみることにします。

[tkcalender03]

```
require 'tk'
require 'date'

p_year = Date.today.year
p_month = Date.today.month

def set_fig(p_year, p_month, l, lm)
  42.times {|x| l[x].text = ''}                # ゴミを残さないために初期化
  days = Date.new(p_year, p_month, -1).day
  i = Date.new(p_year, p_month, 1).wday
  (1..days).each do |d|
    l[i].text = d                              # 直接ラベルテキストを変更
    i += 1
  end
  lm.text = "#{p_year} 年 #{p_month} 月"
end

fm = TkFrame.new('height' => 2).pack
lm = TkLabel.new(fm, 'height' => 2).pack
lm.text = "#{p_year} 年 #{p_month} 月"

f = Array.new(6)
l = Array.new(42)

6.times do |x|
```

```
  f[x] = TkFrame.new.pack('side' => 'top')
  7.times do |y|
    z = 7 * x + y
    l[z] = TkLabel.new(f[x]) {
                              text ''
                              relief 'ridge'
                              height 2
                              width 3
                              pack('side' => 'left')
                             }
  end
end

fb = TkFrame.new.pack('side' => 'bottom', 'fill' => 'x')
TkButton.new(fb) {
              text "前の月"
              command proc {
                            if p_month  == 1
                              p_month = 12
                              p_year -= 1
                            else
                              p_month -= 1
                            end
                            set_fig(p_year, p_month, l, lm)
                           }
              pack('side' => 'left')
              }
TkButton.new(fb) {
              text "次の月"
              command proc {
                            if p_month == 12
                              p_month = 1
                              p_year += 1
                            else
                              p_month += 1
                            end
                            set_fig(p_year, p_month, l, lm)
                           }
              pack('side' => 'right')
             }
set_fig(p_year, p_month, l, lm)
Tk.mainloop
```

実際に月を変更してみて正しく表示されているでしょうか。

ここでは年月の増減部分については、それぞれのボタンの処理のなかに書いていますが、それらも外に出してしまうという方法もあります。そのほうがすっきりするかもしれません。

応用に向けて

ごく簡単ですがカレンダーができました。月の移動もできるようになりました。しかし、

まだ不満もあります。曜日を示す文字を入れたり、曜日によって色分けをしたり、祝日にも対応したいかもしれません。ひと月ずつ前後するのではなく、任意の年月に移動できたほうが便利かもしれません。また、日曜からはじまるのではなく月曜からはじまるほうが使いやすいという人もいるでしょう。

そうした機能を自分で追加していけるのがプログラムすることの便利なところです。

はじめはなかなか意図するような動作をしないことのほうが多いかもしれません。いろいろの書籍を参考にしたり、時には詳しい人に助言を求めたりしてすこしずつ身についていくのだと思います。なにより、そうした工夫をしていく過程は楽しいものです。

あなたなりのカレンダーに発展させてみてください。

参考：

■date/holiday.rb
かつて date2 というライブラリに収められていた日本の祝日を判定するものが holiday.rb です。現在では date2 の基本部分は date に取り込まれていますが、当然ながら日本固有のライブラリとなる holiday.rb は含まれていません。利用するためには、独自にインストールする必要があります。詳細については作者のサイトを参照してください。

http://www.funaba.org/ruby.html

■Windows での Ruby バイナリ
ActiveScriptRuby はとても便利ですが、ActiveScript が必要なければ Rumix というパッケージもあります。同じようにインストールパッケージになっているのでインストールはクリックするだけです。
実行をサポートするコンソールは tcsh ライクな工夫が施されていて、ファイルの種別などのカラー表示ができたりとなかなか便利です。

http://ruby.morphball.net/rumix/

[tkcalender sample]

```
require 'tk'
require 'date'

class TkCal
  def initialize
    @p_year = Date.today.year
    @p_month = Date.today.month
  end

  attr_accessor :p_year, :p_month

  def set_fig(p_year, p_month, l, lm)
    42.times {|x| l[x].text = ''}
    days = Date.new(p_year, p_month, -1).day
    i = Date.new(p_year, p_month, 1).wday
    (1..days).each do |d|
```

```
      l[i].text = d
      i += 1
    end
    lm.text = "#{p_year} 年 #{p_month} 月"
  end

  def backward(p_year, p_month, l, lm)
    if p_month  == 1
      @p_month = 12
      @p_year -= 1
    else
      @p_month -= 1
    end
    set_fig(@p_year, @p_month, l, lm)
  end

  def forward(p_year, p_month, l, lm)
    if p_month == 12
      @p_month = 1
      @p_year += 1
    else
      @p_month += 1
    end
    set_fig(@p_year, @p_month, l, lm)
  end

end

tc = TkCal.new

fm = TkFrame.new('height' => 2).pack
lm = TkLabel.new(fm, 'height' => 2).pack
lm.text = "#{p_year} 年 #{p_month} 月"

f = Array.new(6)
l = Array.new(42)

6.times do |x|
  f[x] = TkFrame.new.pack('side' => 'top')
  7.times do |y|
    z = 7 * x + y
    l[z] = TkLabel.new(f[x]) {
                              text ''
                              relief 'ridge'
                              height 2
                              width 3
                              pack('side' => 'left')
                             }
  end
end

fb = TkFrame.new.pack('side' => 'bottom', 'fill' => 'x')
TkButton.new(fb) {
              text "前の月"
              command proc {
```

```
                                tc.backward(tc.p_year, tc.p_month, l, lm)
                               }
               pack('side' => 'left')
               }
TkButton.new(fb) {
               text "次の月"
               command proc {
                                tc.forward(tc.p_year, tc.p_month, l, lm)
                               }
               pack('side' => 'right')
              }
tc.set_fig(tc.p_year, tc.p_month, l, lm)
Tk.mainloop
```

Pythonの文法(仮)

TSNETスクリプト通信版　草稿　第１回

1. はじめに

本稿は「Pythonの文法(仮)」としてまとめる予定の本の原稿となります。このまま読んで活用して戴けるように書いているつもりではありますが、位置的にはドラフト（草稿）的なものですので、内容の活用に際してはご用心ください。また、内容の引用については自由にして戴いて構わないのですが、引用元はドラフトであることを確認するために、掲載誌である「ＴＳＮＥＴ通信」でお願いします。
※本来は上記の文の代わりに、序文や凡例などが載る予定です。

2. 概略

2-1 文法分析と構成要素

文法の話をするには、まずその構成要素を明確に示す必要があります。
前述の通りリファレンスの要素は、精確／厳密ではありますが、人間が理解するにはやや不便です。
そこで一般の人が理解／利用しやすい文法を再構築するため、用語／分類などを敢えて再定義します。
勿論、正式な用語は支障のない限り取り入れ、再定義の幅は出来る限り抑えます。

2-2 構成要素(component)

まず、構成要素を、以下のように分類します。

・構文キーワード(syntax keyword)
・オブジェクト(object)
・演算子(operator)
・代入子(assignor)
・空白(white space)
・特殊記号(special symbol)

よく、『Pythonは全てオブジェクトで出来ている』といわれますが、これは被演算子(operant=演算子によって演算される対象)に限ります。
Schemeのように演算子が全て関数で、それが全てオブジェクトとして扱える、というほど徹底していません（例えば'+'記号のリストは出来ません）。
また、Smalltalkのように「オブジェクト←メッセージ」で全てを記述しているわけではなく、制御構造などの記述には、ごく古典的な構文キーワードを用いた文法を採用しています。
この点、Pythonの基本構文は、C++やJavaのような文法に近いといえるでしょう（尤も、関数はオブジェクト扱いになりますが)
Pythonでは、文(statement)と式(expression)がはっきりと区別されています。
式は単独で文(式文 expression statement)になりますが、文の全てが式というわけではありません。
構文キーワードを用いた制御構造や代入文などは、一部を除き、式扱いできない文(非式文 not-expression statement)となっています。
文の区切りは、文末がバックスラッシュ('¥')で終わらない改行で行われます（バックスラッシュで終わる行の改行は空白扱いとなります)
ただし、各種『括弧(parenthesis)』が使用されていた場合、『暗黙の行の接続(implicit line joining)』によってカッコが終了するまでは一行は終了しないものとします。
ちなみに括弧は、リファレンスでは『デリミタ(delimiter)』扱いですが、本書では『演算子(operator)』に含めて説明します。
'='などの代入に関する記号も、リファレンスではデリミタ扱いですが、本書では独自に『代入子(assignor)』という用語を用いて、識別子の代入(assignment)に用いるものをまとめました。
他言語では代入に関わる記号も代入演算子として演算子の一種として扱うものもありますが、Pythonでは演算子と代入子をしっかりと区別します。
なぜなら、演算子を用いた結果は式となりますが、代入子を用いた代入文(assignment statement)は式扱いにはならないからです。
空白（スペースとタブ）は、キーワード(構文キーワードと演算子キーワード)と識別子(identifier)を区切るためと、制御ブロックのインデントのために用いられます。
読みやすさのために、要素と要素の間には適宜空白を使用することが許されていますが、Pythonのインデントルールのため、行頭のスペースについては、厳格です。

※例えば、ブロックの中でもないのに、先頭に空白を置くだけでエラーになります。

特殊記号(special symbols)には、上記のいずれにも当てはまらない記号について集めました。
この分類の記号については、複数の役割を持っている場合が多いので注意が必要です。
例えば、コロン(':')は制御ブロックのヘッダ部分のデリミタ(delimiter)として使用される他、辞書リテラルでキーとアイテムを分ける部分で使用されたり、リストの演算子の一種であるスライスで使用したり、引数の型勧告に使用したりします。
他の特殊記号としては、複数行纏め書きのためのセミコロン（';'）、引数リストなどに用いるアスタリスク('*')、修飾子の開始記号であるアットマーク('@')などがあります。（アスタリスクは、乗算の演算子としても使用されます)

2-3 キーワード(keyword)／予約語(reserved word)

キーワードや予約語と呼ばれる、識別子に使えない特殊な文字の組み合わせには、機能的に３種類のものが含まれています。
以下の分類は、この文書独自のものです。

オブジェクトキーワード(object keyword):
　　False,None,True
演算子キーワード(operator keyword):
　　and, in, is, not, or
構文キーワード(syntax keyword):
　　as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, lambda, nonlocal, pass, raise, return, try, while, with, yield

オブジェクトキーワードは、識別子として再定義できないキーワードであるだけで、扱いは通常のオブジェクトに準じます。
演算子キーワードは、記号の代わりに特定の文字列を用いたもので、扱いは演算子に準じます。
ちなみにメンバーテスト演算子 in は、for と一緒に使うことによって、違う意味を持ちます。
構文キーワードには、基本構文の範疇に無い拡張構文を導入する際に使用します。
殆どの拡張構文は非式文となりますが、一部は式文を形成します（lambda,や if-else など)
また、一部キーワードは、ループや関数定義の中など、特別な状況下でしか使用できないものもあります（break や return など)

2-4 オブジェクト(object)

オブジェクトという言葉には様々な定義があり、プログラミング言語の用法によって扱いが異なる厄介な言葉ですが、Python では被演算子(operant)となりうるもの全てを指します（式は被演算子ではありませんが、式が評価された値や式オブジェクトは被演算子です)
初期の Python では、オブジェクトに相当しない原始的な型のデータ(primitive type data)が存在しましたが、現在では、原始型扱いされていた数や文字列も全てオブジェクトとなりました。
モノとしてではなく性質としてのオブジェクト(Python object)とは、クラスインスタンスであることを指します。
つまり Python では、全てのオブジェクトがクラスインスタンス(class instance)です。

※クラスインスタンスとは、クラスという雛形から作られたオブジェクトという意味

Python ではメタクラスプロトコル(metaclass protocol)を採用しているので、クラス自身も、メタクラスのインスタンスです。

※メタクラスはクラスのクラス

さらに Python では、関数(function)やメソッド(method)も全てオブジェクトとして扱います。

※ただし関数やメソッドの扱いは、やや特殊です。

括弧演算子を用いた関数適用を行わなければ、関数そのもののオブジェクトとして扱います。
オブジェクトの共通の性質は、以下のようなものがあります。

・識別子(identifier)の名前に束縛(bind)できる。すなわち代入可能
・同じく属性として代入可能
・タプルなどのコンテナ（コレクション）に収納可能
・必ず値を持つので、真偽判定可能

・関数の引数としての引渡しが可能

※プロパティー(property)は本来オブジェクトですが、例外的に標準的なオブジェクトの扱いを受けないことがあります。

メソッドデータ属性などの属性(attribute)を持つオブジェクトは、識別子や演算子によって、メソッドやデータ属性にアクセスできます。
Pythonには、属性の外部からのアクセスを、根本的に制限する仕組みはありません（例外はプロパティーです）。
オブジェクトには、可変オブジェクト(mutalbe object)と不変オブジェクト(immutable object)の二種類があります。
可変オブジェクトは、オブジェクト生成後にそのオブジェクトの内容が変更可能（再代入ではなく、そのオブジェクト自身の属性が変更可能）ですが、不変オブジェクトでは属性にアクセスすることはできても、変更することはできません。
なお、可変オブジェクトであっても、組み込み型(builtin type)のオブジェクトであった場合は、属性の追加などはできません。
これらの組み込み型を拡張するには、組み込み型を継承した新しいクラスを定義する必要があります（関数クラスなど、一部継承不可能なクラスもあります）。
その他、Pythonではモジュール(module)もオブジェクト扱いされます。

2-5 演算子(operator)

演算子は、Pythonに於いて関数呼び出し以外で演算を行うために選ばれた記号群及び特定の文字列(キーワード演算子)です。
Pythonの演算子は、日常にありふれた数学表現に極力対応しているため、その適用順序も簡単な左結合ではありません。
例えば加算記号の'+'は、括弧を使わない限り、乗算記号の'*'より必ず『優先順位(precedence)』が低くなります。
Pythonでは、ユーザーが定義したオブジェクトに関しては、演算子がどのように働くかを定義することができますが、演算子の優先順位や結合順位を変更することはできません。
また、演算子として使う記号は、決まったもの以外、使用できません。
さらに、単項演算子／二項演算子といった使い方についても、言語として規定された通りにしか使えません。
例えば'+'は単項演算子としても二項演算子としても使えますが、'/'は二項演算子としてしか使えませんので、必ず両辺に被演算子を必要とします。
Pythonで使用できる演算子と優先順位は以下の通りです（優先順位の低い順）

順位	演算子	説明
低	,	カンマ演算子(comma)
↑	*arg, **arg	引数オプション
	lambda, if-else	λ式(Lambda expression),条件付評価式
	or	論理和(Boolean OR)
	and	論理積(Boolean AND)
	not X	論理否定(Boolean NOT)
	in, not in	メンバーテスト(Membership tests)
	is, is not	同一性テスト(Identity tests)
	<, <=, >, >=, !=, ==	比較(Comparisons)
	|	ビット演算の論理和(Bitwise OR)
	^	ビット演算の排他的論理和(Bitwise XOR)
	&	ビット演算の論理積(Bitwise AND)
	<<, >>	ビットシフト(Shifts)
	+, -	加減算(Addition and subtraction)
	*, /, //, %	乗除算、剰余算(Multiplication, division, remainder)
	+x, -x	正負符号(Positive, negative)
	~x	ビット反転(Bitwise not)
	**	累乗算(Exponentiation)
	x[index], x[index:index], x(arguments...), x.attribute	添え字(Subscription), スライス(Slicing)
↓		関数／クラス呼び出し(Call), 属性参照(Attribute reference)
高	(expressions...), [expressions...], {expressions...}	束縛、タプル表示、ジェネレータ式

(Binding, tuple display, generator expressions)
リスト表示(List display)
辞書表示、集合表示(Dictionary or set display)

二項演算子の同順位のものは、原則左から適用されますが、累乗を示す'**'については、右から適用されます。
lambda キーワードによる lambda 文と、条件付評価式(if-else 式)は、キーワードを用いた結果が式になるので、優先順位の表に含めましたが、これらは本書ではリテラル及び特殊構文の項で説明します。

2-6 括弧(parenthesis)、引用符(quotation mark)

括弧は、リファレンスでは全てデリミタ(delimiter)に含まれていますが、前述の通り本書では『グループ化／演算順位変更のための演算子』として扱います。
括弧には種類があり、英語でも混乱しているので、本書での言葉でどの記号を指すのかの対応表を以下に記しておきます。

種類	英語表記	漢字／仮名表記
括弧の総称	parenthesis	括弧
(...)	paren	丸括弧、パーレン
[...]	bracket	角括弧、ブラケット
{...}	brace	中括弧、ブレース

細かく言えば、日本語の括弧は『括(くく)る弧』なので、当然丸いモノを指します。
また、英語でも bracket や brace がドレを指すのか混乱していたりします。

例)　square bracket　=> []
　　bracket　=> 括弧一般 [],(),<>,{}

よって上記の区分は本書のみでの呼称だと思って下さい。
当然ですが括弧の順位は全て同じであり、より内側が優先されます。
引用符は、Python では文字列(string)のリテラル(literal)として利用します。
単引用符も二重引用符も同義ですが、互いの引用符を中に記述することが出来ます（つまり単引用符の中では二重引用符が、二重引用符の中では単引用符が、単なる文字として使用できます)
三重引用符は、単引用符の三回連続と、二重引用符の三回連続で表示されますが、呼称は区別しません。

種類	英語表記	漢字／仮名表記
'...'	(single)quote	単引用符、クォート
"..."	double quote	二重引用符、ダブルクォート
'''...'''	triple quote	三重引用符、トリプルクォート
""".."""	triple quote	三重引用符、トリプルクォート

2-7 代入子(assignor)

リファレンスでは、代入文(assignment statements)および増加代入文(augmented assignment statements)といった名称しか見えず、記号自体には名称が無いので、代入文を生成する'='などの記号を、ここでは代入子と名づけます。
代入子に分類される記号は以下の通りです。

x = y
　単純代入(定義代入)
x += y
　加算代入 ⇒ x = x + y
x -= y
　減算代入 ⇒ x = x - y
x *= y
　乗算代入 ⇒ x = x * y
x /= y
　真性除算代入 ⇒ x = x / y
x //= y
　フロー除算代入 ⇒ x = x // y

x %= y
剰余算代入 ⇒ x = x % y
x **= y
累乗代入 ⇒ x = x ** y
x &= y
ビット論理積代入 ⇒ x = x & y
x |= y
ビット論理和代入 ⇒ x = x | y
x ^= y
ビット排他的論理積和代入 ⇒ x = x ^ y
x >>= y
ビット右シフト代入 ⇒ x = x >> y
x <<= y
ビット左シフト代入 ⇒ x = y << y

代入子と演算子では、適用した結果に生成されるものが異なります。
演算子では、その結果が『式』となりますが、代入子では非式文の『代入文』となります。
よって、式の中に代入子を使った代入文を含めることは、基本的にはできません。
代入文の非式文扱いについては、比較演算子の'=='と代入子の'='を書き間違えないように、とよく説明されます。
なお、定義に使用できる代入子は'='だけですので、注意して下さい。

2-8 特殊記号(special symbol)

演算子にも代入子にもリテラル表記あてはまらない、拡張構文で用いられる記号と用法を、以下に簡単に列挙します。

*　アスタリスク(asterisk)
仮引数リスト、引数リスト展開、余剰(rest)要素収納

**　ダブルアスタリスク(double asterisk)
キーワード引数、引数辞書展開

@　アットマーク(at-mark, at-sign)
修飾子(デコレータ decorator)適用

:　コロン(colon)
ブロックヘッダーデリミタ(block header delimiter)
辞書のキーとアイテムのセパレータ
スライス(slice)

;　セミコロン(semicolon)
複文接続

#　シャープ(sharp)
コメント

¥　バックスラッシュ(back slash)
複数行継続

詳細については、それぞれの項目を参照してください。

3. リテラル

3-1 Pythonのリテラル

プログラミング言語で、オブジェクトの内容を直接書き込む書式を『リテラル(直定数,即値 literal)』といいます。
本書ではPythonのリテラルを、『原子リテラル』『集団リテラル』『その他のリテラル』の３種類に分類します。
リファレンスが定義するリテラルに対して、本書ではリテラルの定義がかなり広くなっています。
これは、構文や式に対して一定の規則の上で理解しやすいように構造化した結果、その例外的な書式を全てリテラルに含めたためです。

よって、このリテラルの項目は、構文や式の落穂拾い的な項目にもなっています。
尚、この章ではリテラル(記法/書式)について述べるにとどめ、生成されるオブジェクトの詳細については別項にまとめますが、各オブジェクトの外部表現については、リテラルと密接な関係にある（リテラルそのままであることが多い）ので若干触れます。

※外部表現(output representation)：コンソールでオブジェクト名を打ち込んでリターンを押した時に表示される文字列。またはオブジェクトをrepr関数に渡した時、返される文字列。あるいはオブジェクトの__repr__メソッドおよび__str__メソッドによって返される文字列。

3-2 原子リテラル(atom literal)

3-2-1 原子リテラルと数値

表記したものが、部分分割不可能な１オブジェクトの内容をあらわす時、それを原子リテラルといいます。
Pythonの原子リテラルは、数値リテラルしかありません。

※単独の文字専用のリテラルは、Pythonには無いので、文字は全て文字列という集団リテラル扱いになります。

数値リテラルには、整数リテラル、実数リテラル、そして虚数リテラルがあります。

※有理数(rational number)を扱うオブジェクトもありますが、リテラルはありません。fractionsモジュールのFractionクラスを利用してください。

3-2-2 整数リテラル(integer literal)

整数はデータの基本形です。
初期のPythonでは、整数はクラスインスタンス扱いではありませんでした
また、ごく最近まで、整数には短整数(int)と長整数(long)のリテラルが別々に存在しました。
現在では、整数は全てintクラスのインスタンスであり、intクラスのリテラルが唯一の整数リテラルです。

※Python 2までは、long型の長整数の表記には'100L'のように、末尾に'L'あるいは'l'を添付していました。

整数リテラルには、通常の十進法表記の他に、十六進法や八進法、そして最近追加された二進法表記などがあります。
以下は全て同じ値となります。

```
255 (外部表現)
0xff   0xFF   0Xff   0XFF
0o377         0O377
0b11111111    0B11111111
```

通常、二・八・十六進法は、コンピュータの内部表現に用いられますが、Pythonの整数はあくまで『記数法』ですので、負の数などは以下のように記述します。

```
-255 (外部表現)
-0xff  -0xFF  -0Xff  -0XFF
-0o377        -0O377
-0b11111111   -0B11111111
```

内部表現などを扱う場合は、集団リテラルの『バイト列』などを使用してください。

※なお、二進法、八進法、十六進法表記は、それぞれbin, oct, hex関数で文字列として表示できます。また、strオブジェクトのformatメソッドを用いると、数値の整形が可能です。

3-2-3 実数リテラル(real literal)

実数のリテラルは、小数点を加えた形で表記されます。

例えば、255 という値は、実数リテラルでは以下のように表現されます。

255.0（外部表現）

小数点以下の０は、いくら続けても良いし、省略してもかまいません。

255.00　　255.

また、ピリオドより前の０も省略できます。

.255 ⇒ 0.255

科学演算に使用される、有効数字的な表現も可能です。

2.55e2　　2.55E3

これは『真数＋常用対数』型の表示であるので、真数がどのような値であってもつじつまがあっていれば構いません。
以下は全て 255.0 の実数リテラルとして有効です。

0.255e3　　25.5e1　　2550e-1　　25.5e+1
255e0　　255e+0　　255e-0

ただし、あくまでリテラル(記述法)であり演算ではないので、対数部分は整数に限られます。

×　　80.638080334293676e0.5（対数部分が整数でないので不可）

実数は float オブジェクト(浮動小数点数オブジェクト)で実装されています。
浮動小数点数オブジェクトは、内部では有限ビット数の二進法で実装されているので、桁数が増えると自動的に丸められてしまう他、十進法では表現可能な数値でも、二進法では表示できない数値などが近似値に置き換わってしまうこともあります。
よって、実数リテラルは、『文字通り(literal)』の正確な実数が再現されるとは限りません。

※ただし、実数は整数より範囲の大きい数値として、実数と整数の計算結果は実数になります。

なお、実数オブジェクトの外部表現は、その数値の桁数などの事情によって、小数点つきの数値('255.0'など)のみで表されるか、あるいは真数＋常用対数表記('2.55e+17'など)のどちらかで表されます。

3-2-4 虚数リテラル(imaginary literal)、複素数リテラル(complex literal)

虚数(imaginary)は、通常数値に'j'を加えて記述します（数学で広く使われているｉではないので注意してください）
以下は、数学に於ける 255i のリテラルです。

255J　255J　255.ｊ　2.55e2ｊ　2550e-1ｊ

虚数表記は実数表記に'j'あるいは'J'を末尾に加えれば、殆ど通用します。
ただし、内部では全て浮動小数点数で扱っている関係もあり、整数×ｉの虚数であっても、整数リテラルのような十進法以外の記数法は使用できません。
ただし、float オブジェクトの外部表現では、整数値相当であっても整数オブジェクトと区別するために'.0'がつきましたが、桁数が多いものを除き、整数×ｉの虚数では'.0'の表示は省略されます（桁数の多いものは、真数＋常用対数 型表記になります）。
なお、一文字の'j'や'J'は、ただの識別子なので、代入されない限り'1j'を意味しません。
虚数リテラルは complex オブジェクト(複素数オブジェクト)を生成します。
複素数オブジェクトについては特にリテラルは存在せず、実数(あるいは整数)オブジェクトと虚数オブジェクトの加減算によって生成されます（外部表現もそれに準じます）

※虚数という言葉は、実数部を含まない『純虚数』だけでなく『実数以外の複素数』という意味もありますが、本文中では純虚数の意味で用いています。

そのルールは、通常の加減算のルールに従います。

複素数オブジェクトは実数(整数)＋虚数の形で表示されますが、実数部分が無い場合は表示は省略されます。
逆に実数部分がある場合は、虚数部分があろうがなかろうが、'(255+0j)'のように複素数表示されます。

3-2-5 真偽値リテラル(boolean literal)

初期のPythonでは、真偽を表す真偽値には、整数の１と０が使用されていました。
現在は、真は'True'、偽は'False'というキーワードが真偽値のリテラルとして使用されます。

3-3 集団リテラル(collection literal)

3-3-1 集団リテラルとは

Pythonのデータ構造として、リストや辞書などのコレクションがあります。
コレクションとは、０～複数の「要素(item)」を持つコンテナオブジェクトのことです。
組み込みのコレクションに関しては、クラス（コンストラクタ）から生成する方法、（可変コレクションの場合）要素を追加する方法などがありますが、要素を直接列挙して記述するリテラルも存在します。
組み込み型のコレクションには、文字列、タプル、リスト、辞書、集合、バイト列、バイト配列などがあります。

3-3-2 文字列リテラル(string literal)

3-3-2-1 基本

Pythonでは、単引用符『'』および二重引用符『"』で囲まれた部分を、文字列(strオブジェクト)として扱います。
どちらを用いても意味は同じですが、　単引用符で定義された文字列の中では、文字として二重引用符を使用することができます（逆も可能）
文字列は、必ずコレクションであり、1文字の文字自身をあらわすリテラルは特にありません。
つまり

```
'a'
```

は、『文字オブジェクト』ではなく、長さ1のコレクションである『文字列オブジェクト』と見做されます。

3-3-2-2 エスケープシーケンス(escape sequence)

文字列の中ではバックスラッシュ『¥(コード0x5C)』はエスケープ文字としてエスケープシーケンスを入力するために使用されます。

```
<エスケープシーケンスと意味>
¥(改行)              改行の無効化
¥¥            バックスラッシュそのもの
¥'            単引用符(')
¥"            二重引用符(")
<ASCIIの制御コード>
¥n            改行 / Linefeed (LF) = '¥x0A'
¥t            タブ / Horizontal Tab (TAB) = '¥x09'
¥a            Bell (BEL) = '¥x07'
¥b            Backspace (BS) = '¥x07'
¥f            Formfeed (FF) = '¥0C'
¥r            Carriage Return (CR) = '¥x0D'
¥v            Vertical Tab (VT) = '¥x0B'
<文字コードによる指定>
¥ooo          八進法による文字コード(ASCII)
               ('¥'＋八進法の３桁の数値)
¥xhh          十六進法による文字コード(ASCII)
               ('¥x'＋十六進法の２桁の数値)
¥uxxxx        16ビットキャラクターコード
               ('¥u'＋十六進法の４桁の数値)
```

```
¥Uxxxxxxxx    Character with 32-bit
               hex value xxxxxxxx
               ('¥U'＋十六進法の８桁の数値)
¥N{name}      Unicode のデータベースに
              収められた名称を入力
```

ASCII の制御コードは、端末によっては無効なので、改行'¥n'とタブ'¥t'以外は期待すべきではないでしょう。
八進法の文字コードについては、指定された桁数未満の数値であれば対応しますが、続けて数値を入れると誤作動するので、頭に 0 を入れて桁を揃えることをお勧めします。
例えば、'¥n'を入力するなら、'¥12'でもいいのですが、'¥012'と書いたほうが安全です。
なぜなら'123¥n456'のつもりで'123¥12456'と入力すると、'¥12'ではなく'¥124'と見なされて'123T56'となってしまいます。
この場合、'123¥012456'と書けば、文字どおりの結果が得られます。
十六進法のコードは、要求された桁数が必要になります（×'¥xa'　○'¥x0a'）
Unicode データベースの ID を参照して文字列に変換する場合は、以下のように使います(name は大文字でも小文字でもかまいません)

```
＜例＞
    '¥N{space}'   ⇒ ' '
    '¥N{SPACE}'   ⇒ ' '
    '¥N{ampersand}'      ⇒ '&'
    '¥N{comma}'   ⇒ ','
    '¥N{colon}'   ⇒ ':'
    '¥N{hyphen}'  ⇒ '-'
    '¥N{left parenthesis}'            ⇒ '('
    '¥N{right square bracket}' ⇒ ']'
    '¥N{hiragana letter a}'           ⇒ 'あ'
```

なお、ターゲットの文字の Unicode データベースのの ID は、unicodedata モジュールの name 関数を用いて調べることができます。

3-3-2-3 raw 文字列(raw string)

『¥』自身を文字として認識するには、'¥¥'と二回続けて書けば良いのですが、正規表現による検索などで文字列の中に多数の『¥』が交じる場合、一々二度づつ書くのも大変ですし、見た目も見づらくなります。
raw(生)文字列は、このような場合に効果を発揮します。
raw 文字列として記述すると、Python はエスケープ記号を『¥』という普通の文字列として扱います。

※raw 文字列自体はリテラルの一種なので、それによって記述され、生成された文字列自体は普通の文字列です。

raw 文字列で記述するときは、引用符の前に'r'か'R'を書きます。

```
r'hello¥n' ⇒ 'hello¥¥n'
```

ただし、r'¥'や r'aaa¥'のように、最後に『¥』がある（正確には末尾に『¥』が奇数個連続している）とエラーになるので注意してください。
通常の文字列の場合は、末尾の『¥』が奇数個並んでいると『¥'』が括弧をエスケープしてしまいます。
ところがこのルールは、実は raw 文字列でも起こります。
つまり、そのままで文字列として成立する文字と記号の並び以外は、raw 文字列としても記述できません。

3-3-2-4 三重引用符(triple quote)による、改行を含む文字列

文字列に限らず、Python コードでバックスラッシュで終了する行は、次の行への継続とみなされ、改行は無視されます。
文字列の場合も、最後にバックスラッシュで終了する場合は、その改行が無視されます。
しかし、三重引用符を用いると、改行をそのまま改行コードとして文字列の中に読み込むことがで

きます。
　三重引用符は、単引用符か二重引用符のどちらかを三回続けて書きます。

```
'''...'''      """..."""

'''aiueo
aiueo'''       ⇒ 'aiueo¥naiueo'
```

　三重引用符は、同種の（単引用符なら単引用符、二重引用符なら二重引用符）の『三連続』まで継続するので、二連続までの同種の引用符を文字列に含めることができます。

```
'''Say "It's"''' ⇒ 'Say "It¥'s"'
```

3-3-2-5 空文字列リテラル(empty string literal)

　空文字列は単引用符か二重引用符の、２連続及び６連続で記述します（外部表現は『''』）

3-3-3 タプルリテラル(tuple literal)

　タプル（tuple オブジェクト、不変リスト）は、基本的にはコンマ(カンマ)演算子','の適用によって生成されます。

```
1,2,3  ⇒ (1, 2, 3)
1,     ⇒ (1,)
```

　外部表現には括弧がつきますが、タプルがタプルであるためにはコンマが必要です。
　よって、要素１個のタプルを作る場合には、必ずコンマを付与します。
　一方、開き丸括弧と閉じ丸括弧の間に、空白と改行以外の要素が無い（あるいは連続）場合、空タプルのリテラルとなります。

```
( )    ⇒ ()
```

　他の場合は必須のコンマが、このときだけは必要ありません（書くとエラーになります）

```
(,)    ⇒ エラー
```

　タプルの外部表現では（１要素のタプルを除き）必ずコンマの後に空白が一つ入ります。

3-3-4 リストリテラル(list literal)

　リスト(list オブジェクト)は、要素を角括弧で囲うことにより直接記述できます（タプルと異なり、一つの要素の場合でも要素の後ろにコンマは要りません）

```
[1] ⇒ [1]
[1,2,3] ⇒ [1, 2, 3]
```

　タプル同様、開き角括弧と閉じ丸括弧の間に、空白と改行以外の要素が無い場合、空リストのリテラルとなります。

```
[   ] ⇒ []
```

　リストの外部表現では（１要素のリストを除き）コンマの後に空白が一つ入ります。

3-3-5 辞書リテラル(dictionary literal)

　辞書(dict オブジェクト)は、キー（key 不変オブジェクトであればなんでも可能）と値（value 全てのオブジェクト）のペア（item アイテム）をコンマで区切ってブレース'{}'の中に記述します。

```
{'a':1,'b':2,'c':3}
  ⇒  {'a': 1, 'b': 2, 'c': 3}
```

　空辞書のリテラルは、ブレースのペアで書きます。

{}

辞書の外部表現では（１アイテムの辞書を除き）必ずコンマとコロンの後に空白が一つ入ります。

3-3-6 集合リテラルと凍結集合リテラル

集合(set)のリテラルは、Python 3になって登場したものであり、歴史的な経緯で同じブレース括弧を用いる辞書との競合のため、やや変則的な書式になっています。
要素を全て書き出す場合は、ブレース括弧の中に要素を書き込みます(重複は自動的に消えます)

{0,1,2,3,2,5,7} ⇒ {0, 1, 2, 3, 5, 7}

ただし、空集合に関しては"{}"が空辞書のリテラルなので、以下のように書きます。

set()

これを「リテラルが無い」と評した人もいたようですが、正確に言うならば「これが空集合のリテラル」なのです。

※個人的にはキーワードを増設して'Phi'（＝Φ）とかだと格好良かったかな、と思いますが

凍結集合(frozenset ： 固定／不変集合)は、変更不可能な集合のバージョンです。

frozenset((5,8,7,9,6,5)) ⇒ frozenset({8, 9, 5, 6, 7})

※raw文字列でr'...'という表記があるのだから、f{...}でも良かったのでは？と思うのですが……

凍結空集合のリテラルも、集合とほぼ同じです。

frozenset()

3-3-7 バイト列リテラル、バイト配列リテラル

バイト列(bytes)は、Python 3になって廃止された従来のstrにあたる、バイトデータのシーケンスです。
バイト列のリテラルは特殊で、基本的に文字列の書き方の拡張バージョンになります。

1. 文字列のように'か"で囲み、その前に'b'をつける
2. 文字コードを指定する時のように、'¥xNN'の形式で書き込む（ただし、xは必ず小文字！）
3. 文字コードが、ちょうどASCIIコードの文字のあるナンバーに重なる場合は、ASCII文字で代用できる

以下に、実際にリテラルと外部表現を書いてみます。

b'¥x01¥x11¥x21¥x31¥x41¥x51¥x61¥x71' ⇒ b'¥x01¥x11!1AQaq'

バイト列は不変オブジェクトですが、その可変バージョンがバイト配列(bytearray)です。
バイト配列コンストラクタはバイト列を引数に取ってバイト配列を生成するのですが、リテラルや外部表現も、その形式がそのまま使用されます。

bytearray(b'abc') ⇒ bytearray(b'abc')

バイト列、バイト配列は、文字列のような書式ですが、要素はあくまで整数です。
詳細は、オブジェクトの項を参照してください。

3-4 その他のリテラル

3-4-1 遅延評価に関するリテラル

ここでは、原子でも集団でもない、特殊なリテラルについて記述します。
具体的にはλ式(lambda expression)や生成子(generator)式などです。

これらの扱いは複雑で、lambda はリファレンスでは λ 演算子(lambda operator)と呼ばれ演算子扱いですし、生成子は式扱いです。
本書でこれらをリテラルに分類する理由は以下の通りです。

１．評価内容を具体的に記述する書式（リテラル）である

２．評価結果がオブジェクトとして通用するため、構文扱いはやや不適切。定義文の一種と見ることもできるが、式の中で扱えるため、応用範囲が異なる

３．式一般（演算子及び関数の適用）のルールからは逸脱しているため、特殊な書式が多いリテラル扱いが妥当

要約すれば「リテラルとして見るのが一番分かりやすいだろう」ということです。
もちろんこれは、筆者独自の考えであり、多く支持された考えではありません。
素直に考えるのなら、λ 式は関数定義文の変形、反復子式は for 構文の変形、そしてリテラルでは扱いませんが、条件付評価式(conditional expression)は if-else 構文の変形、となるでしょう。
しかし、文法の根幹となる構文（定義文や制御文）の例外を増やすことは、Python 文法全体を不必要に複雑化してしまう懸念がありました。
その点リテラルは、文法の中でも特異的な書式がもともと多いので、ここで扱ったほうが、全体の構造の解釈を大改編するよりも理解しやすいと判断しました。

要するに「その他のリテラル」に属する書式は、かなり例外的なものです。
これを巧く使用すれば、プログラム構造をかなり単純化することが可能となる反面、どちらも無分別に多用すると Python が最重要視する『可読性』が損なわれてしまう可能性が大いにあることが指摘されているものでもあります。
くれぐれも、扱いは慎重にしましょう。

3-4-2 関数リテラル(function literal)

lambda キーワードを使うと、適用することで値を返す関数を、そのまま記述する関数リテラルが書けます。

```
lambda x: x + 1
```

上記は、引数に 1 を加える関数のリテラル、つまり関数自身となります。
関数を適用させるには、

```
f(x)
```

とするので、一度変数に代入して

```
f = lambda x: x + 1
f(10) ⇒ 11
```

とすることもできますし、リテラル自身に適用して、

```
(lambda x: x + 1)(10) ⇒ 11
```

とすることもできます。
lambda の一般書式は以下の通りです。

```
lambda [仮引数[, 仮引数[...]]]: 式
```

仮引数は、'lambda'と':'の間に書きます。
仮引数は、何個でも（無くても）構いません。
引数リストやキーワード引数の書式は、関数定義と同じですので、関数定義の章を参照してください。
なお、lambda キーワードの『演算子』としての優先順位はさほど高くない（というか最低）なので、lambda の適用範囲は、必要に応じて適宜括弧を使用します。
たとえば、引数を第一要素、10 を第二要素とするタプルを返す関数を定義する際、

```
lambda x: x, 10
```

とすると、うまくいきません（関数と10のタプルになってしまいます）
　この場合、正しくは

　　lambda x:（x, 10）

とします。

3-4-3 生成子リテラル(generator literal)

　生成子(generator)は、要素を一定の法則にしたがって生成するタイプの反復子(iterator)です。
　反復子と生成子の詳細な説明については「オブジェクト」の項を参照してもらうとして、ここではごく大ざっぱに説明します。
　反復子とは、next関数に引き渡される毎に規定された要素を戻すオブジェクトです。
　通常、反復子は要素を規定したコレクション（主にシーケンス）をiter関数に引き渡すことにより生成されますが、生成子は生成子定義関数などに要素の生成ルールを書いて作ります。
　この、生成子定義関数で作ることのできる反復子(生成子)のうち、単純な繰り返しによるものは、生成子リテラルとして記述可能です。

※生成子自体は「継続」に近い概念も再現できますので、生成子リテラルで記述できる生成子は、生成子の可能性のごく一部です。

　反復子は、next関数の引数として渡される度に、記述された特定の数値を返します。
　簡単な実例を見て見ましょう。

```
>>> i = iter([1, 2, 3])
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

　最初にリストで指定した要素が、次々に返されています。
　生成子は、このようなものを式で記述します（そのまま返してもつまらないので、ちょっとヒネってあります）

```
>>> g = (x + 1 for x in [0,1,2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

　先頭の式'x + 1'は、next(g)の返り値のテンプレートになります。
　forに続く変数xの中に、inの後にあるリストなどの反復可能オブジェクト(iteratable object)の要素が順に代入されます（ここに反復子を入れても、勿論OKです）
　例えば、１回目はリストの先頭の0がxに代入されて、'x + 1' ⇒ 1 となり、１が返ります。
　そして２回目は１が代入されて……という具合です。
　リストなどの反復可能オブジェクトの要素が尽きてしまったら、その後は反復子は例外'StopIteration'を発生しつづけます。
　式の変数として２種類以上の変数に対しforを用いた場合、後ろのforから順に適用されます。

　　(x + y for x in (10,20,30)

```
              for y in (1, 2, 3))
```

の場合、生成される数値は、

11⇒12⇒13⇒21⇒22⇒23⇒31⇒32⇒33

の順に生成されます。

```
>>> g = (x + y for x in (10,20,30) for y in (1, 2, 3))
>>> for k in g:
...   print(k, end=' ')
... else:
...   print()
...
11 12 13 21 22 23 31 32 33
>>>
```

for 文の詳細については、制御構文の項を参照してください。
ここでは、next 関数で得られる値を順に取得して表示しています。

シーケンスに条件を加えてフィルタリングすることもできます。

```
(x for x in (1,2,3,4,5,6) if x % 2 == 0)
```

この反復子は、リストの中の偶数のみを生成します（比較演算子'=='や、剰余演算子'%'については『演算子』の章参照）

```
>>> gg = (x for x in (1,2,3,4,5,6) if x % 2 == 0)
>>> for k in gg:
...   print(k, end=' ')
... else:
...   print()
...
2 4 6
>>>
```

条件は、for の数だけ設定できます。

3-4-4 内包表記(comprehension)

一般的に生成子リテラルは、for 文と一緒に用いるか、あるいは反復可能オブジェクトを引数とする関数に引き渡されます。

```
sum(x for x in (1,2,3,4,5)) ⇒ 15
```

生成子が生成する数値をコレクションにおさめるために特殊な構文糖衣が用意されています。

※構文糖衣(syntax sugar)とは、同じ内容を簡単に書くための略記法です。生成子リテラルも構文糖衣の一種と言えます。

生成子が生成する数値をリストにおさめるには、本来は以下のように書きます。

```
list(x for x in (1,2,3)) ⇒ [1, 2, 3]
```

構文糖衣を使うと、以下のように書けます。

```
[x for x in (1,2,3)] ⇒ [1, 2, 3]
```

このような、生成子によるコレクションの構文糖衣を内包表記(comprehension)と言います。
集合の内包表記も同じです。

```
set(x for x in (1,1,2,3)) ⇒ {1, 2, 3}
{x for x in (1,1,2,3)} ⇒ {1, 2, 3}
```

辞書の内包表記は、少し変わっています。
生成したい辞書が{1:10,2:20,3:30}の場合、

```
{x:y for x,y in ((1,10), (2,20), (3,30))}
        ⇒ {1: 10, 2: 20, 3: 30}
```

と書きます。
もし、xとyを別々のforでくくると、思ったとおりの結果は得られないでしょう。
内包表記を使わないと、こうなります。

```
dict(((x,y) for (x,y) in ((1,10),(2,20),(3,30))))
        ⇒ {1: 10, 2: 20, 3: 30}
```

内包表記を使わなければ面倒ですが、内包表記を使ってもあまり便利そうではありません。
実際には、zip関数やrange関数などの組み込み関数と併用することで、内包表記は威力を発揮します。

```
>>> {x: y for x, y in zip((1, 2, 3),(10, 20, 30))}
{1: 10, 2: 20, 3: 30}
```

※上の例は「ちっとも便利ではなさそうだ」と思えるかもしれませんが、キーとアイテムがそれぞれ別々のコレクションとして与えられていた場合は、結構便利です。

4. 定義

4-1 定義とは

プログラミング言語では、オブジェクトを生成する際に宣言、定義(definition)を行います。
宣言は変数などのシンボルの使用の告知、定義は実際のとメモリの確保、初期化(initialize)は定義されたオブジェクトが使えるようにすることです。
Pythonでは通常、宣言を省略して定義や初期化のみでオブジェクトを生成します。
また、定義は一部を除き、同時に初期化を行います。
しかし、新出の識別子を自動的に定義する機能は無く、未定義の識別子を代入などの対象以外に使用するとエラーとなります。
また、識別子自体は単なるラベルですので、どんなオブジェクトにも再定義可能ですが、オブジェクト自身はきっちりと判断され、その場に応じて適切に変化する、といったことはありません。

※例えば、文字列と数値に'+'を適用すると、数値を文字列とみなしたりすることなく、エラーを発します。

つまり、いわゆるタイプルーズ(type loose)の言語ではない、ということです。
なお、λ式や内包表記によるオブジェクト定義は、第２章でリテラルとして扱っていますのでそちらを参照してください（基本的に式扱いです）

4-2 代入文による定義

4-2-1 基本形

定義及び初期化を最も簡単に行う方法は、リテラルや式の代入によるものです。
もっとも基本的な形式は以下の通りです。

```
変数 = 式
```

右辺の式が評価された結果が、変数に代入(assignment)されます。
変数名(シンボル)が、今まで使われていないものであれば、代入の瞬間、宣言と定義と初期化が行われます。
ちなみにPythonのオブジェクト名は全ていわゆる参照（リファレンス／ポインタ）のラベルですので、正確には以下のような手順を取ります。

1. 式の結果を表すオブジェクトが生成される。
2. 左辺の変数(シンボル)にラベルとして、オブジェクトの番地が渡される。

ラベルとしてシンボルを与えられないオブジェクトは、式の評価されている時のみ存在します。
そうでなければ、ラベルやコンテナからの全ての参照が終了した時点で、オブジェクトは破棄され

ます。

※以上の理由から、本来ならば「変数(variable)」や「代入(assignment)」という用語は誤解を招くので避けた方が良いのかもしれません（assginment は「代入」と訳されるが、本来「割り当て」という意味）しかし、「名札(label)」や「記号(symbol)」および「束縛(bind)」などの用語（関数型プログラミング言語でよく使われる用語で、こちらの方が実情に近い）よりは「変数」「代入」の方が一般のプログラマの方々に馴染みがあると考えて、以降も敢えてこのように書きます。また、本質的に参照であるといっても、オブジェクトの実体が不変(immutable)である場合は、実質上の扱いは変数／代入となんら変わりません（可変オブジェクトは若干影響します）

なお、代入子の仲間に'+='などの演算代入子(operate assignor)がありますが、'='と異なり、定義済みの変数にしか使用できませんので、これによって初期化／定義することはできません。

4-2-2 一括代入書式

'='代入子は、連続して使用することができます。

変数1 = 変数2 = 変数3 = 式

上記の場合は、最右辺の式の値が、変数1,2,3に順次代入されます。
原理的にはいくつ繋いでも構いません。
また、複数の代入（多重代入）を一括して書くこともできます。

変数1, 変数2, 変数3 = 式1, 式2, 式3

これは、意味合いとしては以下と同じです。

変数1 = 式1
変数2 = 式2
変数3 = 式3

左辺のターゲットが複数で、右辺がコレクションオブジェクトの場合、その要素が展開されて代入されます。

```
>> lst = [10, 20, 30]
>> a, b, c = lst
>> print(a, b, c)
10 20 30
```

※Python では、オブジェクトをコンマで区切ると、タプルというシーケンス型のコレクション構造となるので、リストの展開と多重代入は、実は仕組は同じです。これをシーケンスの開梱(unpack)といいます。

さて、ここまではターゲットの数とコレクションのアイテムの数が同じでなければなりませんでした。
Python3 から導入された「残余(rest)」を扱う書式では、ターゲットの一部をリストとして扱うことにより、余った部分を収納することができるようになりました。
書式としては、ターゲットの変数の前にアスタリスク('*')を置くことによって、残余を収納する変数を決めます。

```
>>> a, b, c, d, *e = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> print(a,b,c,d,e)
0 1 2 3 [4, 5, 6, 7, 8, 9]
>>> a, b, c, *d, e = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> print(a,b,c,d,e)
0 1 2 [3, 4, 5, 6, 7, 8] 9
>>> a, b, *c, d, e = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> print(a,b,c,d,e)
0 1 [2, 3, 4, 5, 6, 7] 8 9
>>> a, *b, c, d, e = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> print(a,b,c,d,e)
0 [1, 2, 3, 4, 5, 6] 7 8 9
>>> *a, b, c, d, e = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> print(a,b,c,d,e)
```

```
[0, 1, 2, 3, 4, 5] 6 7 8 9
>>>
```

なお、開梱の逆として、一つの変数に複数の値を収納する梱包(pack)という書式もあります。

```
>>> a = 1, 2, 3
>>> a
(1, 2, 3)
>>>
```

しかし、カンマはタプルを作る演算子でもあるので、これは特殊な書式というよりは、カンマで区切られたタプルを単に左辺の変数に代入しているだけ、という見方も出来ます（詳しくは演算子の項を参照してください）

4-2-3 ラベルの消去(del 文)

使用している識別子を消去するためには、del 文を用います。

```
del 識別子[, 識別子..]
```

del 文で消去されるのは識別子だけであって、オブジェクト自体は消去(破棄)されません（オブジェクトは全ての参照が無くなった時に破棄されます）
del 文で、コレクションアイテムの一部や、オブジェクトの属性なども消去できます。

```
del コレクション[インデックス]
del オブジェクト.属性
```

del 文の適用はオブジェクトによって異なりますので、詳しくは「オブジェクト」の項を参照してください。

4-3 定義文

4-3-1 定義文によって定義されるオブジェクト

定義文によって定義されるオブジェクトは、関数オブジェクトとクラスオブジェクトの２種類です。正確には、クラスオブジェクトに属するメソッドオブジェクトやプロパティーオブジェクトも定義文によって定義しますが、少なくともどちらも書式的には関数定義にそっくりです。さらに、継続する反復子(iterator)を生産(yield)する生成子も、関数と同様の書式で書かれます。この「定義文」の項では、関数定義とクラス定義の２種類を扱います。メソッド定義はクラス定義の中で、プロパティー定義は 3-4 の「修飾子」の項で、生成子定義関数は関数の項で説明します。

4-3-2 pass 文

関数やクラスの定義の際、内容を一切書かないことも可能です。
しかし、Python では定義の区切りを表す括弧や定義(ブロック)終了の end キーワードを使用しませんので、どこで終了するのか知らせる必要があります。
pass 文は、関数やクラスの内容が一切ないことを示す文です。
pass というキーワードを書くだけのシンプルなものですが、これによって空の定義を書くことが出来ます。
なお、空の関数はスタブ（トップダウン開発のための仮の単体モジュール）に、空のクラスはＣ言語の構造体や Pascal のレコードのような使い方をする場合があります。

4-4 関数定義

4-4-1 基本

プログラムでは、再利用される処理の単位をなんらかの形で纏めます。
クラッシックな言語ではサブルーチンという形で、構造化された言語では「手続き(procedure)」や「関数(function)」という形で纏められます。
Python ではこういった処理ブロックの単位は、一括して「関数」と呼ばれるもので纏めます。
Python では、関数という言葉を「呼び出し可能オブジェクト」の意味として扱います。
よって、マニュアルの一部には（わざと）呼び出し可能な「関数以外の」オブジェクトを「関数」と呼んでいる場合もあります。

ここで扱う書式によって生成されるのは、'function'クラスのインスタンスである関数オブジェクトです（lambda 式を使う方法は「第３章　リテラル」の関数リテラルの項を参照してください。また、関数オブジェクトについての詳細な情報は「第５章　オブジェクト」を参照してください）
関数定義は、一般に以下のように行います。

```
def 関数名([変数1,変数2...]):
        実行文１
        実行文２
        …
```

関数名と変数（仮引数）名は、識別子のルールに従います。
関数は呼び出されると、実行文に書かれた部分を実行します。
実行文は、インデントしたブロックの中に記述します（この中に、さらに制御構造などの構造文が入らない限り、インデントの深さは一定とします）
関数を呼び出す際の引数(実引数 argument)は、仮引数(parameter)として０～いくつでも指定することが出来ます。
関数は呼び出す際、仮引数として定義に書いた数だけ引数を必要とします（過不足があってはいけません）
関数内で定義された識別子は、基本的にその関数内だけで有効です。
関数の周囲の環境の識別子は、関数内から参照することができます。
しかし、識別子への代入は定義ですので、たとえ環境で使用されている識別子であっても、関数内で値を代入された瞬間、それは「関数内で定義された識別子」扱いとなります（つまり、外の識別子へは影響を与えません）

```
>>> x = 100
>>> def f():
...   x = 200
...
>>> f()
>>> x
100
>>>
```

なお、識別子への代入ではなく「変更」であった場合、その変更は外部の識別子が指し示すオブジェクトを変更させます。

```
>>> x = [0]
>>> def f():
...   x[0] += 1
...
>>> x
[0]
>>> f()
>>> x
[1]
>>> f()
>>> x
[2]
>>> f()
>>> x
[3]
>>>
```

なお、global 宣言を用いれば、外部環境の識別子の内容を変更させることもできます。

4-4-2 戻り値と return 文

副作用を目的としない関数は、戻り値(return value)を返す必要があります。

※正確には、副作用を目的とする処理ブロックは手続き(procedure)と呼ばれる方が相応しいのでしょうが、Python では副作用や戻り値の有無に関係なく「関数」という用語で統一されています。

戻り値は関数定義文中の、return 文で設定します。

return 式

return 文は、関数定義(及びメソッドなど、def キーワードで始まる類似した定義)内で書かれなければ、エラーとなります。

```
>>> def func(x, y):
...   return x + y
...
>>> func(5, 10)
15
>>>
```

実行文の内容が、上記のように１行で済むようなものなら、ブロックを作らずに':'の後に続けて書くことも出来ます。
なお、ついでに言えば、単純計算をするような式であれば、関数リテラルを使ったほうが簡単で明快かもしれません。

```
>>> def func2(x, y): return x + y
...
>>> func2(5, 10)
15
>>> func3 = lambda x, y: x + y
>>> func3(5, 10)
15
>>>
```

return 文(及び後述の yield 文)の無い関数及び、return キーワードに続く式を省略した場合は、戻り値として特殊オブジェクト None を返します。
None の詳しい性質はオブジェクトの項に譲りますが、基本的には捨てられる値です。

※もし、C 言語のいくつかの標準関数のように、実行内容の成功や失敗の結果を返したいのなら、True か False を返すよう、にすべきでしょう。

return 文が実行されると、関数の処理はそこで直ちに終了され、処理が呼び出し元に戻ります。
よって、retrun 文の後に実行文を書いたり、複数の return 文を書く場合は、if 文などの条件分岐構文を入れなければ、最初の return 文が実行された時点で以降は全て無効になります。

```
>>> def func():
...   return 100
...   a = 10
...   b = 20
...   return a + b
...
>>> func()
100
>>>
```

なお、複数の値を返す場合は、カンマで区切ってタプルにして返すのが一般的です（シーケンスの開梱による多重代入が可能です）

```
>>> def func():
...   return 10, 20, 30
...
>>> func()
(10, 20, 30)
>>> a, b, c = func()
>>> a
10
>>> b
20
>>> c
30
>>>
```

4-4-3 生成子定義関数(function of defining generators)とyield文

生成子(generator)は、要素を一定の法則にしたがって生成するタイプの反復子です。
生成子定義関数は、生成子の要素生成の定義を記述する関数です。
生成子はリテラルの項で生成子リテラルとして記述する方法が出てきましたが、正式にはこの生成子定義関数を用いて定義します。
ここでは生成子の動作から、生成子定義関数の定義の仕方を見て行きましょう。

```
>>> def gendef(a):
...   yield a * 10
...   yield a * 20
...   yield a * 30
...
>>> gen = gendef(10)
>>> next(gen)
100
>>> next(gen)
200
>>> next(gen)
300
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>> gen2 = gendef(10)
>>> for x in gen2:
...   print(x, end=' ')
... else:
...   print()
...
100 200 300
>>>
```

生成子をnext関数に渡すと、生成子定義関数のyield文で指定された部分の値を返します。
この点はreturnによく似ているのですが、生成子はここまでの実行経緯を記憶していて、次に呼び出された時には、その次の行から実行を再開します。
生成子定義関数は通常、生成子リテラルのようにwhile文やfor文などの反復構造を持つ構文が中に入ります。
(下の例は、生成子リテラルの項で(x + y for x in (10,20,30) for y in (1, 2, 3))として作ったものと同じです)

```
>>> def gendef():
...   for x in (10, 20, 30):
...     for y in (1, 2, 3):
...       yield x + y
...
>>> g = gendef()
>>> for x in g:
...   print(x, end=' ')
... else:
...   print()
...
11 12 13 21 22 23 31 32 33
>>>
```

しかし実際には反復子の名に反して反復構造を持つ必要は無く、適宜中断／再開可能なブロックとして使うことも可能です。
なお、yield文に渡す式の部分を省略すると、return文と同じようにNoneを返します。

4-4-4 引数のデフォルト値(default argument values)

関数は、呼び出し時に仮引数で設定されたのと同じ数の引数を必要とします。
しかし、ある引数が通常的によく使用される場合、デフォルト値をセットすることにより、省略す

ることが出来ます。
　デフォルト値は、『仮引数＝デフォルト値』として設定します。

```
>>> def func(x, y=10):
...   return x + y
...
>>> func(5,20)
25
>>> func(5)
15
>>>
```

　なお、全ての引数にデフォルト値を設定することはできますが、デフォルト値を設定した仮引数の後に、デフォルト値を設定しない通常の引数はセットできません。
　また、デフォルト値は一度だけしか評価されませんので、可変オブジェクトを設定した場合は、呼び出し間でその値が共有されることに注意してください。
（以下は、Python 公式ドキュメントのチュートリアルに掲載されている例です）

```
>>> def func(x, y=[]):
...   y.append(x)
...   return y
...
>>> func(1)
[1]
>>> func(2)
[1, 2]
>>> func(3)
[1, 2, 3]
>>>
```

　リストについての詳細はオブジェクトの項を参照してください。
　ここでは、デフォルト変数として初期化されたリストがそのまま２回目以降も継続されて使用されていることが確認できます。
　もし、毎回リストを初期化したい場合は、

```
def func(x, y=None):
  if y == None:
    y = []
  y.append(x)
  return y
```

とすると良いでしょう（日本語版のチュートリアルの初版のサンプルコードは、インデントが間違っています）
　なお、これは一種のクロージャ(閉包)として働くため、この性質を逆に利用してコードを書くことも可能です。
　以下は呼び出す度に値を加算する累算器(accumulator)の一種です。

```
>>> def ac(x, y=[0]):
...   y[0] += x
...   return y[0]
...
>>> ac(1)
1
>>> ac(2)
3
>>> ac(3)
6
>>>
```

　ちなみに関数リテラル(lambda 式)でも、デフォルト引数は書けます。
　最初の例を lambda 式で書くと、

```
func = lambda x, y=10: x + y
```

のようになります。
　しかし、累算器を書こうとすると、かなりトリッキーな形になります（以下のコードは「遊び」ですので、無理に理解する必要はありません）

```
>>> ac = lambda x, y=[0]: 0 if y.__setitem__(0, y[0]+x) else y[0]
>>> ac(1)
1
>>> ac(2)
3
>>> ac(3)
6
>>>
```

【教訓】なんでも lambda で書こうとしてはいけません(笑)

4-4-5 任意の個数の引数に対応する自由引数リスト(arbitrary argument list)

　Python はリストやタプルなどのコレクションを引数として関数に渡すことができるので、任意の個数の引数に対応する必要はあまり無いのですが、簡便的にそのような関数を書きたい場合には、方法があります。

```
def 関数名(*引数タプル変数名):
    実行文
```

　たとえば、組み込み関数の sum は、コレクションのアイテムの値の合計を返す関数ですが、直接書き込む場合には、

```
sum((1, 2, 3))
```

と書き込まなければなりません（当然です。直接書くなら、+で繋げば済むのですから）
　これを無理やり「引数を合計する関数」に変えるなら、以下のように行います。

```
>>> def xsum(*args):
...   k = 0
...   for x in args:
...     k += x
...   return k
...
>>> xsum(1,2,3)
6
>>>
```

　仮引数の前に'*'をつけると、その変数は引数をまとめてタプルとして受け取ります（よく『引数リスト』と言われますが、タプルです）

```
>>> def f(*x):
...   return x
...
>>> f(1,2,3)
(1, 2, 3)
>>>
```

　タプルはリストと同じシーケンス型のコレクションなので、添え字(index)を用いて参照することができます。

```
>>> def f(*x):
...   return x[2]
...
>>> f(1,2,3)
3
>>>
```

　ミニ言語っぽく使用する関数群を用意する時などには、この任意個数の引数を受け取る方法が役に立つかもしれません。

4-4-6 キーワード引数を辞書で受け取る

関数は呼び出す際に、引数を単に順にならべれば順に関数に引き渡され、関数内では仮引数として書かれた部分に代入されます。
Python では、関数内での仮引数の名前が分かっているのなら、仮引数の名前を指定して、ターゲットの仮引数に代入することができます。
これをキーワード引数といいます（詳しくは「式」の章の「呼び出し」の項を参照）
ここでは、簡単にキーワード引数の動作を見ておきます。

```
>>> def f(a, b, c):
...   return a * b - c
...
>>> f(2, 5, 10)
0
>>> f(5, 10, 2)
48
>>> f(c=2, a=5, b=10)
48
>>>
```

'=' を使って仮引数名を指定すると、指定された仮引数に引数の値が渡されることがわかります。
さて、このキーワード引数による呼び出しを辞書として利用したい場合は、以下のように記述します。

※辞書はマップ型のコレクションオブジェクトで、リストのインデックスが文字などになったもので、他言語では連想配列、ハッシュなどと呼ばれています。詳しくは「オブジェクト」の章を参照してください。

```
def 関数名(**引数辞書変数名):
        実行文
```

以下に、実行例を上げます。

```
>>> def f(**d):
...   return d
...
>>> D = f(a=10,b=20,c=30)
>>> D
{'a':10, 'c':30, 'b':20}
>>> D['a']
10
>>> D['b']
20
>>> D['c']
30
>>>
```

任意個数の引数に対応する時のように、ミニ言語を作ったり、あるいは他のスクリプト言語への橋渡しなどに使用すると威力を発揮するでしょう。
Python モジュールでは、Tcl 言語とのインターフェイスである tkinter モジュールが、このような形式を多用しています。

4-4-7 特殊仮引数の順序

デフォルト値(デフォルト値つきの仮引数)、引数タプル、引数辞書(キーワード引数の辞書)など、Python では様々な仮引数が設定できますが、これらが混在する場合には、順序が決まっています。

```
def 関数名(通常の仮引数, デフォルト値, 引数タプル, 引数辞書):
```

どれとどれを組み合わせても構いませんが、順序を間違えるとエラーが出ます。
ちなみに、引数タプルや引数辞書は、通常の仮引数やデフォルト値と一緒に使うと、これらの仮引数で指定されていない引数（引数の残余）のみを収納します。
なお、当然ですが、２個以上の引数タプルや、２個以上の引数辞書を書くことはできません。

【全てのキーワードを入れてみた例】

```
>>> def allparam(a, b=100, *c, **d):
...   return a,b,c,d
...
>>> allparam(1,2,3,4,5)
(1, 2, (3, 4, 5), {})
>>> allparam(1,2,3,x=4,y=5)
(1, 2, (3,), {'y': 5, 'x': 4})
>>> allparam(1,2,3,4,c=5)  #【引数タプルの'c'は、指定なしと見なされる】
(1, 2, (3, 4), {'c': 5})
>>> allparam(1,2,3,4,d=5)  #【引数辞書の'd'は、指定なしと見なされる】
(1, 2, (3, 4), {'d': 5})
>>> allparam(1,2,3,c=4,d=5)
(1, 2, (3,), {'c': 4, 'd': 5})
>>>
```

【上記の関数でエラーになる引数指定】

・キーワード引数は通常の引数の後にかかなければならない

```
>>> allparam(a=1,2,3,4,5)
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

・最初の引数(位置引数)が'a'に代入されるので、'a'が多重代入となる

```
>>> allparam(1,2,3,4,a=5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: allparam() got multiple values for keyword argument 'a'
>>>
```

4-4-8 関数に対する情報

4-4-8-1 Pythonインタープリタと閲覧(inspect)

この項で取り上げる説明文(documentation string, docstring)や注釈(annotation)は、直接実行には影響を及ぼしませんが、関数のユーザー（そしてそれは、自分の記述を忘れてしまった未来のあなたかもしれない）に、help関数を通して情報を与えるための記述方法です。

コンパイル型の実行環境、あるいはソースファイルを読ませるのが主な実行環境の言語になじみの方には、この項目の意味がややわかりづらいかもしれませんが、Pythonではソースではなくインタープリタでテストしながら使うことも多いのです。

PythonインタープリタはLisp系言語のREPL(read-evalute-print loop)に近い環境で、その実行内容には一切制限はなく、実行速度とメモリの利用効率以外、ソースファイルを直接渡してコンパイル（自動）して実行するのと変わりません。

関数型言語のHaskellのインタープリタは関数定義はできませんでしたが、Pythonインタープリタでは関数定義どころかクラス定義だろうがなんだろうが可能ですし、インタープリタ環境のままパッケージを導入したりモジュールを読み込んで実行したりすることも可能です。

そして、help関数を使えば、(Smalltalkのオブジェクトブラウザには劣りますが)オブジェクトに関する様々な情報を閲覧することができます。

※help関数、dir関数などは、実行環境の情報取得専用の関数です。詳しくは「実行環境の情報の閲覧」の項を参照してください。

4-4-8-2 説明文(docstring=document string)

関数やメソッド（そしてクラス）の実行文の一行目に、文字列を書くと、help関数に渡された際に表示される説明文（ドキュメンテーション文字列）となります(下の例の説明文は、トリプルクォートによって改行を含んでいますが、普通の文字列で１行でも構いません)

```
>>> def f():
...   """No function.
...     No action."""
...   return 0
...
>>> f()
0

>>> help(f)
Help on function f in module __main__:

f()
   No function.
   No action.

>>>
```

※オブジェクト自身は式ですから、コード中に数字やクォートに挟まれた文字列をそのまま書いても、エラーにはなりません。ただし'#'でコメントアウトしておかないと、無駄な処理をすることになります。

レイアウトは、２行目以降の空行以外の行の先頭をインデントの基点とします（１行目はその基準行に揃えられます)
なお説明文は、__doc__属性に文字列として保存されています。
help関数を自作したい(?)などに、参考にしてください。
特にルールではありませんが、次のような書き方の「しきたり」があるそうです。

【一般的な説明文の書き方】
・一行目：簡単な説明
・二行目：空行
・三行目以降：詳細な解説

```
>>> def addxy(x, y):
...     """２引数を加算する。
...
...     加算した合計を返す。"""
...     return x + y
...
>>> help(addxy)
Help on function addxy in module __main__:

addxy(x, y)
    ２引数を加算する。

    加算した合計を返す。

>>>
```

4-4-8-3 注釈(annotations)

関数についての注釈の文法はPython3に導入された文法です。
Pythonでは引数の型を強制することは出来ません（検査してエラーを出すことは出来ますが）が、どのような引数を取ることを期待して関数を書いたか、ということをユーザに知らせる方法があります。

※ユーザはその「期待」通りに「振舞う」オブジェクトを渡して使えば良いのです。

注釈は、引数と返り値について次のように書きます

```
def 関数名(変数1:式1, 変数:式2, ...)->式R :
```

式1,式2...(お望みならいくつでも)は、それぞれ変数1,変数2...の型を書きます。
式Rには、戻り値の型を書きます。
'式'となっているのは、そこで評価されたものが表示されるからです(1+1と書けば2と表示される)。
実際、この「注釈」のルールはかなりカオスで、式として評価してエラーの出ないものならなんで

も取り入れます。
　ただし逆に、れっきとした型であっても、実行している名前空間に名前として辞書に登録されていないものは、直接書き込めません。
　例えば、型(クラス)を表示するtype関数に、関数オブジェクトを渡すと、'function'という型が表示されます。
　しかし「関数を返す関数を書こう」と思って『def func()-> function:』のように書くと、「functionという名前が定義されていない！」とエラーが出ます。
　というわけで、『def f()-> 'function':』と書くしかないようです。
　なお、注釈情報は__annotations__属性に収納されています。

【重ねて注意】annotationsは注釈であって、引数の型宣言ではありません。C,C++,Java,Pascalなど、型が強制される言語を通常使っている方は勘違いしないようにしてください。Pythonには、型を指定／強制する仕組みは存在しません。

4-5 クラス定義(class definition)

4-5-1 クラス定義の概要

　クラス(class 種類)とは、第一義的にはオブジェクトのテンプレートですが、同時に「クラスオブジェクト(class object)」というオブジェクトでもあります。
　クラスオブジェクトとしての振る舞いは「クラスオブジェクト」項に譲るとして、この「クラス定義」の項ではオブジェクト（インスタンス）を定義するものとしてのクラスという視点を中心に話を進めます。

※クラス定義をテンプレートとして生成されたオブジェクトを、インスタンス(instance 事例)といいます。

　クラス定義は、通常のメソッドや属性の設定の他に、継承のルール、メソッドフックによる演算子の多重定義(overload)、あるはメタクラスプロトコル(metaclass protocol)など様々な要素を含みます。
　それぞれについて、全て一通り触れることになりますが、この項は定義についての話題なので、その他詳細については別項に譲ることにします。

4-5-2 基本型

　オブジェクトのテンプレートとしてのクラスの定義は、基本的に以下のように定義されます。

```
class クラス名:
        定義文１
        定義文２
        ...
```

　クラスの定義も、関数定義と同様、インデントされたブロックの中に記述されます。
　定義文では、主に属性やメソッドの設定を行います。
　なお、メソッドはクラス内に定義された関数オブジェクトで、メソッド特有のルールはありますが属性の一種なので、属性のルールに全て従います

※注意：メソッドは「クラス内『に』定義された関数」であり、「クラス内『で』定義された関数」だけではありません。クラス定義の外で定義された関数をメソッドとして代入することも可能です。詳細は「メソッド」の項を参照してください。

　一番単純なクラス定義は、以下のようになります。

```
>>> class C:
...   pass
...
>>> c = C()
>>>
```

　cはクラスCのインスタンスで、内容的には何もありません。
　クラス名を関数のように呼び出すことにより、インスタンスが作られます。
　この関数(として呼び出されたクラス)を、以後C++言語などに倣ってコンストラクタ(constructor)と呼びます。
　コンストラクタは、基本的にはインスタンスを生成し、それを初期化して返します。

初期化については、後に述べる__init__メソッドによって規定できます（詳しくは後述の「特殊メソッド」で説明します）

※実はコンストラクタの動作をコントロールする__new__という特殊メソッドもあるのですが、非常に複雑な動きをするものなので、Python のオブジェクトモデルに十分慣れてからの使用をお勧めします。__new__特殊メソッドについては「特殊メソッド」及び第二部の「オブジェクト指向プログラミング」で詳しく説明します。

クラスにも説明文(docstring)を書いて、help で呼び出すことができます（書き方は関数と全く同じです）

```
>>> class C:
...   'クラスの説明です'
...   pass
...
>>> help(C)
Help on class C in module __main__:

class C(builtins.object)
 |  クラスの説明です
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

>>>
```

4-5-3 属性(attribute)

属性は、クラス及びインスタンスに属する変数です。
クラス、インスタンスに共通する属性は、クラス定義で単純に代入することにより、設定できます。

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

属性はインスタンス及びクラスから、ドット演算子('.')を介して呼び出されます。

オブジェクト.属性

クラス属性に値が直接代入された場合、クラス属性を参照するインスタンス属性も変化します。

```
>>> C.x = 200
>>> C.x
200
>>> c.x
200
>>>
```

しかし、インスタンス属性に直接値が代入されると、その時点でインスタンス独自の(クラス属性を参照しない)インスタンス属性が生成されます。
これを遮蔽(shadow)あるいは上書き(override)といいます。
遮蔽は、通常の OOP 言語では上位クラスと下位クラスの間で起こりますが、インスタンスが独自の属性を持つことの出きる Python では、クラスーインスタンス間でも遮蔽が起きます。

```
>>> C.x
200
>>> c.x
200
>>> c.x = 300
>>> C.x
200
>>> c.x
300
>>> del c.x
>>> c.x
200
>>>
```

上記の通り、インスタンス属性を消去(delete)してやると、インスタンス属性のみが消去され、再びクラス属性を参照するようになります。
クラス属性にない属性を、インスタンス独自に設定することも可能です。

```
>>> class C:
...   x = 100
...
>>> c = C()
>>> c.y = 200
>>> c.y
200
>>>
```

インスタンスの属性変更は、クラスに一切影響を及ぼしません。
※インスタンスからクラス自身を参照して、クラスの属性を変更することは可能です。詳細は「オブジェクト」の項で説明します。

逆にクラスの変更は、全インスタンスに影響を及ぼしますが、インスタンス独自に設定した属性には変化を及ぼしません。

```
>>> class C:
...   x = 100
...
>>> c0 = C()
>>> c1 = C()
>>> c1.x = 200
>>> del C.x
>>> c0.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'x'
>>> c1.x
200
>>>
```

基本的にクラスとインスタンスの、属性についての関係はシンプルで、インスタンス独自の属性がなければクラスを参照する、というものです。
インスタンス属性からクラスを経由して参照しても、オブジェクトの振舞は「一つの例外を除き」全く変わりません。
その例外とは、クラス定義内で定義された関数及び、クラス属性として代入された関数オブジェクトです。
インスタンスがクラスを経由して関数を参照すると、関数は「メソッド」として働きます。

※ここが微妙なのですが、クラス属性として関数を参照すると、普通の関数扱いになります。

メソッドの詳細は次項「メソッド」で詳しく解説します。
なお「メソッド以外の属性」は「データ属性(data attribute)」と呼ばれています。

4-5-4 メソッド(method)

4-5-4-1 メソッドの概要

メソッドの扱いはPythonのオブジェクトモデルの中でも異色の存在ですので、若干定義の範囲を越えた解説を交えながら話を進めます（全般的な情報については「カスタムオブジェクト」の「インスタンス」の項を参照してください）
メソッドは、クラス属性として設定された関数オブジェクトを「インスタンスから呼び出した」ものです。
メソッドの最大の特徴は、第１引数にインスタンス自身を受け取ることです（この引数の仮引数名は慣例的に'self'となっていますが、強制ではありません）
簡単な例を見てみましょう。

```
>>> class C:
...   def x(self):
...     return self
...
>>> c = C()
>>> C.x
<function x at 0xca4f8>
>>> c.x
<bound method C.x of <__main__.C object at 0xc7a50>>
>>> c.x()
<__main__.C object at 0xc7a50>
>>> c
<__main__.C object at 0xc7a50>
>>>
```

c.xがC.xを参照しているにもかかわらず、片やメソッド(正確には「束縛されたメソッド bound method」)、片や関数(function)と表示されています。
そして、第１引数をそのまま戻すc.x()の結果は、c自体が戻されている（つまりc自身が暗黙的に代入されている）ことが分かります。
同じことをCから行ってみましょう。

```
>>> C.x(c)
<__main__.C object at 0xc7a50>
>>>
```

C.xのほうは、設定通り１引数を必要とします。
C.xの引数を省略したり、あるいはc.xに余分な引数を渡して実行すると、エラーになります。
なお、C経由で実行した場合のこの書式は単なる裏技ではなく、「継承」の項で使用します。

ちなみに、メソッドは基本的に関数ですので、いくつでも変数を指定できます。
さらに、関数でおなじみの引数オプションも使えます。

```
>>> class C:
...   x = 0
...   y = 0
...   z = ()
...   def res(self, v, *vv):
...     self.x += 1
...     self.y += v
...     self.z += (v,) + vv
...     print('total=', self.y, ' count:', self.x)
...     print('log=', self.z)
...
>>> c = C()
>>> c.res(1,10)
total= 1  count: 1
log= (1, 10)
>>> c.res(2,20)
total= 3  count: 2
log= (1, 10, 2, 20)
>>> c.res(3,30,30)
total= 6  count: 3
log= (1, 10, 2, 20, 3, 30, 30)
```

```
>>> c.res(4,40)
total= 10  count: 4
log= (1, 10, 2, 20, 3, 30, 30, 4, 40)
>>>
```

4-5-4-2 メソッドのスコープ

Pythonのスコープ(範囲 scope = 識別子の有効範囲)と名前空間(name space)のルールは、Pythonの文法の中でも直感的に判りづらいところなので、別項で詳しく説明しますが、ここではメソッド定義に関して最低限知っておかなければならないことを話しておきます。
まず、関数の場合はその周囲の識別子を直接参照することができますが、メソッドはクラス内やインスタンス内に設定されている識別子を直接参照することはできません。

```
>>> def f():
...   print(v)       #【外部のｖを参照】
...
>>> class C:
...   x = 100        #【クラス属性にｘをセット】
...   def call_x(self):
...     print(x)     #【外部のｘを参照】
...   def call_y(self):
...     print(y)     #【外部のｙを参照】
...
>>> c = C()
>>> c.y = 200        #【インスタンス属性としてｙをセット】
>>> v = 300          #【実行環境レベルでｖをセット】
>>> f()
300                  #【参照可能】
>>> c.call_x()       #【クラス属性を参照→失敗】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in call_x
NameError: global name 'x' is not defined
>>> c.call_y()       #【インスタンス属性を参照→失敗】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in call_y
NameError: global name 'y' is not defined
>>> x = 400          #【実行環境レベルでｘをセット】
>>> c.call_x()
400                  #【関数と同様に参照可能】
>>>
```

もうお分かりだと思いますが、メソッドが第１引数にインスタンス自身を取るのは、インスタンスへの暗示的な参照ができないからです。

※このような仕様の理由は、プログラミング言語に対する哲学的なものが多分に含まれているのでしょうが、手間をかける最大のメリットの一つは、メソッド内で使用される変数の由来がはっきりしたり、仮引数との名前の衝突を避けたりできます。もちろんそれがコストに見合うかどうかを判断するのは、それこそ設計者の哲学でしょう。

4-5-4-3 その他

引数に関するオプションなどは、関数と全く同じです（詳細は「関数」の項を参照してください）
また、関数リテラルで書いた関数をクラス属性として代入しても、問題なくメソッドとして機能します（ただし__init__など、値を返さない特殊メソッドなどは、関数リテラルでは定義できません）
以下の例は、インスタンスを作った後にクラスに属性として関数オブジェクトを登録し、インスタンス経由でメソッドとして使用しています。

```
>>> class C:
...   pass
...
>>> c = C()
>>> def f(self):
...   return 100
```

```
...
>>> C.f = f
>>> C.l = lambda self: 200
>>> c.f()
100
>>> c.l()
200
```

もちろん、後付けのインスタンス変数も参照可能です。

```
>>> c.x = 500
>>> C.l2 = lambda self: self.x
>>> c.l2()
500
```

ただし、「クラス経由」でないと、ただの関数になってしまう（第１引数に暗示的にインスタンス自身を取らない）ので注意してください。

```
>>> c.L = lambda self:self.x
>>> c.L()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes exactly 1 positional argument (0 given)
>>> c.L(c)
500
>>>
```

多分、C++やJavaなどのプログラミングをご存知の方は、このファジーさには眩暈がするかもしれませんが、良きにせよ悪しきにせよ、これがPythonスタイルです。

4-5-5 継承

継承は非常に重要な概念ですが、定義の書式自体は至って簡単です。

```
class クラス名(基底クラス1[,基底クラス2 ...]):
```

Pythonは多重継承を(割と無頓着に)導入しています。
複数指定された基底クラスへの属性検索順序は、左から順というコトになります。
一番左の基底クラスが何らかのクラスの派生クラスであった場合、その基底クラスまで順にさかのぼります。
後述しますが全てのクラスは'object'クラスが基底クラスとなりますし、重複したクラスから派生していることもありますが、検索前に全てのオブジェクトの派生関係を整理した後に検索にかかるので、同じクラスを二度検索することはありません。
多重継承は、よく言われている通り使い方が難しいので、導入は慎重に行いましょう。

4-5-6 メタクラス

メタクラスの話題は、かなり詳しい本でもあまり触れられないPythonのオブジェクト指向プログラミング言語システムの最深部に当たるものです。
扱いが難しく、使用する場面が本当にあるのかどうかは微妙なところですが、ここでは簡単に書式だけ触れておきたいと思います。
ちなみにメタクラス（metaclass 超クラス？）とは、「クラスのクラス」即ち「クラスオブジェクトのクラスにあたるもの」です。
Smalltalkなど一部のOOP言語では、そもそもメタクラスを使わないとクラスが記述できないのですが、Pythonは別にそういったことはありません。

```
class クラス名([基底クラス1[,基底クラス2 ...]], metaclass=メタクラス):
```

基底クラスは必ずしも記述する必要はありませんが、もし書くなら、メタクラス指定より前になります。
メタクラスの詳しい使用法は「オブジェクト」の項目で説明します。

4-5-7 クラス定義と関数定義の違い

クラス定義と関数定義は似て非なるものです。
書式としては、関数定義とクラス定義は不必要なほどそっくりです（私はもう慣れてしまいましたが、クラス定義の基底クラスを記述する際に、関数の引数と同様のスタイルを取ることに異論のある人も多いと思います）
しかし、クラス定義と関数定義では、細かなところで判りづらい違いがあります。
まず、内部に記述された文の実行タイミングですが、関数の場合は最初に呼び出されるまで一切行われません。
それに対し、クラス定義文は、定義文が登録された直後に実行されます。
次の例を見てください。

```
>>> def f():
...   f.x = 100
...
>>> f.x              #【関数が実行されていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'x'
>>> f()
>>> f.x              #【関数が実行されている】
100
>>> class C:
...   x = 100
...
>>> C.x              #【定義後既に実行されている】
100
>>>
```

Python のクラスは、基本的には名前空間（オブジェクトの名前と対応を所持したデータベース）です。
そしてクラス定義は、独立した名前空間の中で行われる処理そのものです。
呼び出されるまで評価されない（コンパイルされてオブジェクトにはなりますが、実行はされない）関数定義とは、この点が根本的に異なります。
また、スコープルールに関してもかなり大きな違いがあります。
このことについては、「スコープルール」の章で触れます。

4-6 特殊メソッド

4-6-1 特殊メソッド(special method)の概要

Python ではメソッド定義の際に、ある特殊な名称を用いることにより、インスタンスオブジェクトを初期化させたり、様々な演算子が利用できるように演算子の多重定義(overload)を行ったり、あるいは属性への入出力を管理することが可能です。
特殊メソッドは必ずダブルアンダースコアに挟まれた名前を持ちます（'__init__'など）。
特殊メソッドとして登録されている名前と重複しなければ、前後ダブルアンダースコアのメソッド及びデータ属性を設定しても構わないのですが、特殊メソッド及び特殊属性オブジェクトは膨大な数がある（上に、仕様変更でよく追加される）ので、自分のオリジナルとして設定するデータ属性／メソッドにはそのような名前を使うのは止めたほうがいいでしょう。
この「定義」の項では、特殊メソッドのうち、定義に関係の深いメソッドのみを取り上げます。
演算子の多重定義に関係するものや、特定の関数の戻り値として設定されているものは、「オブジェクト」の項に譲ります。
ただし、演算子の一種に分類される、属性参照演算子の'.'(別名：ドット演算子)については、定義との関わりが深いので、この項目で取り上げます。

4-6-2 生成／消去に関するメソッド

4-6-2-1 生成と消去

インスタンスはクラスオブジェクトを関数呼び出し（クラス呼び出し）して、生成し、del 文によって（あるいは『ごみ集め＝メモリ回収』によって自動的に）消去されますが、その際のオブジェクトの振る舞いを、以下の特殊メソッドで記述できます。
ここでは、非常に使用頻度の高い（多分、普通に作るクラスの半数以上で使用する）'__init__'メソッド、使用場面は限られるものの効果的に使えば便利な'__del__'メソッド、更にはオブジェクト指向プログラミングにある程度精通した玄人向けの'__new__'メソッドの三つを紹介します。

4-6-2-2 初期化メソッド('__init__')

書式：__init__(self[, arg ...])
動作：インスタンスを初期化する。引数を指定した場合は、コンストラクタ経由で渡される。

特殊メソッドの中で、多分一番使用頻度の多いものは、この__init__メソッドでしょう。
このメソッドは、インスタンスが作成された際に、デフォルトでそのインスタンスを初期化します。
例を挙げてみましょう。

```
>>> class C:
...   def __init__(self):
...     self.x = 10
...     self.y = 20
...
>>> c = C()
>>> c.x
10
>>> c.y
20
>>> class Cv:
...   def __init__(self, x, y):
...     self.x = x
...     self.y = y
...
>>> cv = Cv(5, 10)
>>> cv.x
5
>>> cv.y
10
>>>
```

最初の例では、引数なしの初期化です。
第１引数を用いて、__init__の中で初期化していきます。
次の例は、引数を含めた方法です。
変数x,yは引数でインスタンス生成時に引数として受け取ったものです。
self.x, self.yはインスタンス属性をあらわします。
つまり、インスタンス属性を明示的に指定することにより、おなじ変数名でも混乱することなく設定することが出来ます。
なお、__init__メソッドはインスタンス自身を初期化するメソッドなので、return文を書いてはいけません（エラーになります）

4-6-2-3 消去メソッド('__del__')

書式：__del__(self)
動作：オブジェクト消去時の動作を規定

このメソッドは、いわゆるデストラクタ(destructor)に相当するもので、インスタンスが消去／破壊された時に実行されます。
つまり、インスタンスが消去されるときに実行したい内容を記述しておけばよいのです。
以下の例は、インスタンスに蓄えられた文字列を、インスタンス消去時にログに落とす操作を登録したものです。

```
>>> class D:
...   def __init__(self):
...     self.s = ''
...   def __del__(self):
...     f = open('log.txt','a')
...     f.write(self.s + '\n')
...     f.close()
...
>>> d1 = D()
>>> d2 = D()
>>> d1.s += 'aiueo\n'
>>> d1.s += 'kakikukeko\n'
```

```
>>> d2.s += 'sasisuseso\n'
>>> d2.s += 'tatituteto\n'
>>> del d1
>>> del d2
>>> exit()

c:\prog\python>type log.txt
aiueo
kakikukeko

sasisuseso
tatituteto
```

ファイルオブジェクトを直接扱うと、ファイルオブジェクトが存在している間ファイルハンドルを独占することになりますが、このように貯めた情報を一気に記録すれば、ファイルハンドルを独占する時間が短くて済みます。
その他、終了操作などが必要なもの（言語としての儀式は必要ありませんが）については、このメソッドを活用することができます。

4-6-2-4 生成メソッド('__new__')

書式：__new__(cls[, arg..])
動作：クラスがコンストラクタとして呼び出された際の動作を定義する（クラスメソッド）

__new__メソッドは、真のコンストラクタとでも言うべき特殊メソッドで、クラス名を関数呼び出しされると自動的に呼び出される関数です。
通常、このメソッドを設定しなおす必要はありません。
インスタンスの初期化は__init__で行いますので、通常の設定は__init__メソッドを使えば十分です。
この__new__メソッドは、デザインパターンの導入など、高度なオブジェクト指向プログラミングを行う際にのみ再設定が必要になります。
もし、インスタンス生成過程をカスタマイズする必要があり、__new__メソッドを設定しなおす必要がでてきた場合は、以下の点に注意してください。
まず１点目、__new__メソッドで通常どおりインスタンスを生成する場合は、それらを明示的に示す必要があります。
__new__メソッドは、簡単に言えばクラス名を関数呼び出しした際に行う動作を設定します。
通常、クラス名を関数呼び出しした際にインスタンスが生成されるのですが、__new__メソッドを設定した場合は、必要ならば設定しなければなりません。
インスタンス生成方法については、下に例を挙げています。
なお、生成されたインスタンスが返された場合、__init__メソッドが設定されていれば（その後に）実行します（なお、インスタンスを生成しなかったり、別の値を返したりした場合は、__init__メソッドは働きません）
２点目、__init__メソッドと異なり、__new__メソッドではインスタンスを戻り値として設定する必要があります（でないと、インスタンスの代わりにNoneが返されます）
３点目、__new__メソッドはインスタンスに先立つ存在なので、第１引数はインスタンスではなく「クラス」になります（これは、後で説明する変則参照メソッドの一種の「クラスメソッド」扱いであることを意味します。
以下に、インスタンス生成数をカウントするカウンタを__new__メソッドに組み込む例を挙げます。

【インスタンスの数を数えるカウンタを作る】

```
>>> class C:
...   _cnt = 0
...   def __new__(cls, *args):
...     cls._cnt += 1
...     ins = object.__new__(cls)
...     return ins
...   def __init__(self, *args):
...     self.args = args
...   def getcount(self):
...     return C._cnt
...
>>> c0 = C(1,2,3)
>>> c0.args
```

```
(1, 2, 3)
>>> c0.getcount()
1
>>> c1 = C(4,5,6)
>>> c1.args
(4, 5, 6)
>>> c1.getcount()
2
>>> c0.getcount()
2
>>>
```

__new__の第１引数名がself でなくclsになっているのはクラスであることを強調するための慣例です。
よってcls._cntはインスタンス属性ではなくクラス属性(クラス変数)であることに注意してください。
これは、getcountメソッドでC._cntという形で参照されているものと同じです（Pythonは属性に対しては必ず明示的に参照先を記します)
__new__の後の仮引数タプルは、__init__が変数を取らないなら必要ありません。
なお、__new__の引数の形式と__init__の引数の形式が違っていても、実際にクラス名で引数を取って呼び出された形式にどちらでも適合するのであれば、問題ありません。
ins = object.__new__(cls)は、クラス(cls)からインスタンス(ins)を生成するための慣用句(idiom)です（詳細は、「オブジェクト指向プログラミング」の項で説明します)
ちなみに、object.__new__(cls)自身は__init__インスタンスを生成するだけで、cls.__init__は実行しませんので注意してください（クラス内で別のオブジェクトを生成して返す時など)
なお、上の例は__new__メソッドの動作を説明するためのもので、同じことは__init__メソッドを用いても簡単にできます。

```
>>> class C:
...   _cnt = 0
...   def __init__(self, *args):
...     C._cnt += 1
...     self.args = args
...   def getcount(self):
...     return C._cnt
...
>>> c = C(1,2,3)
>>> c.getcount()
1
>>> c0 = C(4,5,6)
>>> c.getcount()
2
>>> c0.getcount()
2
>>>
```

__new__メソッドを本当に必要とするのは、インスタンスの生成に関わる過程にかなり大きく関与したい時に限られます。
以下は、デザインパターンで有名なSingletonパターンを__new__を用いて実装したものです。

※Sigletonは、クラスのインスタンスがただ一つであることを保証するパターンです。

```
>>> class Singleton:
...   instance = None
...   def __new__(cls, *args):
...     if cls.instance == None:
...       ins = object.__new__(cls)
...       ins.args = args
...       cls.instance = ins
...     else:
...       ins = cls.instance
...     return ins
...
>>> s = Singleton(1,2,3)
>>> s.args
```

```
(1, 2, 3)
>>> s2 = Singleton(4,5,6)
>>> s2.args
(1, 2, 3)
>>> s2.args = 4,5,6
>>> s2.args
(4, 5, 6)
>>> s.args
(4, 5, 6)
>>>
```

※↑の例ではｓとs2は同一のオブジェクトです

繰り返しになりますが、__new__メソッドの活用には、Pythonの知識のみならず、オブジェクト指向プログラミングの知識が必要となりますので、ご注意ください。

4-6-3 属性アクセスに関するメソッド(accessor)

4-6-3-1 Pythonの属性アクセス

基本的にPythonは属性（データ属性、メソッド問わず）直接参照したり代入したり消去したりすることが自由に出来ます。
しかし、存在しない属性を参照すれば当然エラーが出ますし、メソッドが内部で使用しているメソッドをうっかり上書きすると、内部のメカニズムが崩れてしまう可能性もあります。
このような場合に使用するのが、属性アクセスをフックするメソッド群です。

※フック(hook)する……鈎(hook)でひっかけて横取りする、というような意味で、要するにある書式についてその処理を横取りするような動作をすること。Pythonではこのような属性アクセスのフックの他、演算子の適用をフックすることにより、演算子の多重定義などを可能にしている。

しかし、メソッドの中ですら明示的な参照を用いてプログラミングを行うPythonでは、属性への制限は、それらへのアクセスの制限にもつながります。
つまりややこしい話ですが、アクセス制限のメソッドを設定する際に内部で属性にアクセスしようとすれば、制御がループすることになってしまいます。
ここで紹介する__getattr__と__setattr__には、それぞれそういったループをしないための防止策がありますが、__getattribute__という究極のアクセス制限メソッドにはそれがありません。

4-6-3-2 属性参照メソッド('__getattr__'と'__getattribute__')

書式：__getattr__(self, name)
動作：存在しない属性に対する参照をフックする

書式：__getattribute__(self, name)
動作：ドット演算子による全ての属性参照をフックする

属性参照をフックするメソッドは'__getattr__'と'__getattribute__'ですが、この二つを同時に使用することはありません。
取り扱いについては、__getatte__の方が簡単ですので、こちらを先に説明しましょう。
__getattr__特殊メソッドは、インスタンスが持たない属性を参照したときの振舞を規定します。
たとえば、xがyという属性を持たなかった際に、x.yを参照するとエラーが生じます。
しかしx.__getattr__メソッドが設定されていれば、処理はこのメソッドに委譲されます。
変数nameには参照された属性名を文字列として受け取ります（例えば'y'）ので、それを基準に処理を分岐させることもできます。
注意すべきは、__getattr__がフックするのは、データ属性としてだけではなく、メソッド呼び出しの場合、例えばx.y()のようなものにも反応するということです。
オブジェクトがどのような使われ方をするか十分予測した上でないと、__getattr__は有効に機能できないかもしれません。
以下は__getattr__の動作確認です。

【予想外の呼び出しにとりあえず対応する】

```
>>> class X:
...   def __getattr__(self, name):
...     return name #【呼び出し名をそのまま返す】
```

```
...   v = 111
...   def f(self):
...     return 999
...
>>> x = X()
>>> x.x
'x'                  #【データ属性としての呼び出しに対応】
>>> x.x()            #【メソッドとしても呼び出されるが、対応できていない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
>>> x.v
111                  #【存在するデータ属性はフックしない】
>>> x.f()
999                  #【存在するメソッドもフックしない】
>>> x.y = 666        #【新規にインスタンス属性を追加】
>>> x.y
666                  #【追加された属性もフックしない】
>>>
```

__getattribute__は__getattr__の強化版で、ドット演算子による参照一切をフックします。
これは、クラスの中で設定するメソッド内部にも適用されるので、メソッドを設定する場合には要注意です（そもそも__getattribute__メソッド内で参照をすると、無限ループします）
利用法としては、__getattribute__内部でインスタンスの属性参照の動作を全て委譲してしまうことが挙げられます。

【内容を保護されたインスタンス】

```
>>> class X:
...   def __getattribute__(self, name):
...     if name == 'f':
...       return lambda *args: args     #【f属性はメソッドとして設定】
...     elif name == 'x':
...       return 666                    #【x属性はデータ属性として設定】
...     else:
...       return None                   #【それ以外は反応なし】
...
>>> x = X()
>>> x.x
666
>>> x.f(1,2,3)
(1, 2, 3)
>>> x.k              #【反応しない】
>>>
>>> x.x = 777        #【x属性を上書き】
>>> x.f = None       #【f属性を上書き】
>>> x.x
666                  #【x属性を参照していないので影響されない】
>>> x.f(5,6,7)
(5, 6, 7)            #【f属性を参照していないので影響されない】
>>>
```

なお、__getattribute___メソッドがフックするのはドット演算子'.'による参照だけですので、演算子をフックする特殊メソッドなどは設定できます。
しかし、内部でインスタンス属性を参照する際はもれなく__getattribute__を経由することになりますので、設定には相当の注意が必要となります。

4-6-3-3 属性設定メソッド('__setattr__')

書式：__setattr__(self, name, value)
動作：属性への代入をフックする

__setattr__は属性の代入をフックするメソッドです。
x.y = v ⇒ x.__setattr__('y',v)

__setattr__は__getattr__と異なり、既存の属性があろうが無かろうが属性がへの代入を「１つの例外を除いて」フックします。
例外とは__dict__という属性です。
__dict__属性は、オブジェクトの全ての属性の情報を集めた辞書のようなオブジェクトで、属性名(文字列)をキーとして、オブジェクトの属性を参照したり属性に代入したりすることができます。

x.__dict__['y'] = v ⇒ x.y = v

この形式は__setattr__にフックされないので、__setattr__本体や他のメソッドで属性代入が必要な箇所で利用することができます。

【新規の属性には頭に'd'をつけることを強要する】

```
>>> class X:
...   def __setattr__(self, name, value):
...     if name[0] == 'd': #【頭文字が'd'のもののみ許す】
...       self.__dict__[name] = value
...   cx = 666                    #【クラス内での変数】
...   def __init__(self):
...     self.a = 111              #【__init__による初期化　ルール違反】
...     self.d = 222              #【__init__による初期化　ルール通り】
...     self.__dict__['v'] = 333 #【__dict__による迂回】
...   def z(self):          #【メソッド】
...     return 444
...
>>> x = X()
>>> x.a                          #【__init__内でもルール適用】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'X' object has no attribute 'a'
>>> x.d                          #【ルール通り】
222
>>> x.v                          #【__dict__迂回OK】
333
>>> x.cx                    #【クラス変数はOK】
666
>>> x.z                          #【メソッドもOK】
<bound method X.z of <__main__.X object at 0x00B9E490>>
>>> x.z()
444
>>> x.y = 555               #【ルール違反】
>>> x.y                     #【通過せず】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'X' object has no attribute 'y'
>>> x.dy = 555              #【ルール通り】
>>> x.dy
555
>>> x.__dict__['y'] = 777 #【__dict__迂回OK】
>>> x.y
777
>>>
```

__setattr__は、属性に「誤って」代入してオブジェクトのメカニズムしないよう、防護柵を作ることなどが可能です。
ただし、Pythonのシステムで、故意の破壊を防ぐことは至難のワザです（ほとんど出来ません）

4-6-3-4 属性消去メソッド('__delattr__')

書式：__delattr__(self, name)
動作：属性の消去をフック

__delattr__は先の__setattr__と似たような動作をします。
__dect__を経由したアクセス以外の「del オブジェクト.属性」という書式をフックします。
以下に、簡単な例として、属性の消去をブロックする方法を挙げます。

【属性消去をブロックする】

```
>>> class X:
...   def __delattr__(self,name):
...     print("Don't Delete!")   #【消去しようとしたら警告】
...
>>> x = X()
>>> x.x = 100
>>> x.x
100
>>> del x.x
Don't Delete!              #【消去しようとしたので警告】
>>> x.x
100
>>> del x.__dict__['x']    #【でも__dict__から迂回すれば可能】
>>> x.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'X' object has no attribute 'x'
>>>
```

用法も__setattr__と同じで、属性保護などが考えられます。

4-7 変則参照メソッド

4-7-1 セルフポインタを有しないメソッド

基本的にメソッドの第１引数は、そのオブジェクト自身です。
しかし、クラスメソッドと静的(static)メソッドは、第１引数がオブジェクト自身を指しません。
オブジェクトのメソッドでありながら、オブジェクト自身の属性を参照したり変更したりしないこの２つのメソッドは、追加された文法に共通の「巧く使えば便利だが、下手に使うとコードを見づらくする」という性質を持っています。
基本的に使わずとも同等のコードを書けるので、導入は慎重に行ってください。

4-7-2 クラスメソッド(class method)

クラスメソッドは、メソッドの第１引数がそのオブジェクトそのものではなく、そのオブジェクトの「クラス」です。
クラスメソッドを導入するには、デコレータを用いるのが簡単です。
例を見てみましょう。

```
>>> class C:
...   x = 100
...   def imeth(self):
...     print(self.x)
...   @classmethod  #【クラスメソッド】
...   def cmeth(self):
...     print(self.x)
...
>>> c = C()
>>> c.x = 500
>>> c.imeth()
500                #【インスタンス属性を参照】
>>> c.cmeth()
100                #【クラス属性を参照】
>>>
```

'imeth'と'cmeth'の内容は、'cmeth'が'@classmethod'の後に記述されている以外、全く同じですが、'imeth'の'self'がオブジェクト自身であるインスタンスを指しているのに対し、'cmeth'の'self'は、クラスを指していることがわかります。
このため、クラスメソッドの第１引数は、オブジェクト自身を指す'self'の代わりに、クラスを表す'cls'を使うのが一般的です。
classmethodクラス(クラスオブジェクトです)は、クラスメソッドを生成するクラスで、修飾子とし

て使わなくとも使えます。
　また、無理にクラスメソッドを使わずともクラス属性を参照する方法もあります。

```
>>> class C:
...   x = 100
...   @classmethod                #【修飾子を使う】
...   def cm0(cls):
...     print(cls.x)
...   def cm1(cls):               #【普通のメソッド】
...     print(cls.x)
...   cm1 = classmethod(cm1)      #【クラスメソッドに変える】
...   def cm2(self):              #【普通のメソッド】
...     cls = type(self)          #【クラスを呼び出す】
...     print(cls.x)
...   def im(self):               #【比較のための普通のメソッド】
...     print(self.x)
...
>>> c = C()
>>> c.x = 500                     #【インスタンス属性変更】
>>> c.cm0()
100                 #【クラス属性参照】
>>> c.cm1()
100                 #【クラス属性参照】
>>> c.cm2()
100                 #【クラス属性参照】
>>> c.im()
500                 #【インスタンス属性参照】
>>>
```

　type 関数は、オブジェクトのクラスを返す関数なので、これを使えばクラスメソッドをわざわざ使う必要はありませんが、クラス属性を頻繁に参照／変更するようなら、クラスメソッドにするメリットもあるでしょう。
　それでは、実用サンプルとして、インスタンスから属性を参照された回数を表示するメソッドを考えてみましょう。

```
>>> class C:
...   x = 0
...   @classmethod
...   def getx(cls):
...     cls.x += 1  #【カウンタ加算】
...     print(cls.x, '回目の参照です')
...
>>> c0 = C()
>>> c1 = C()
>>> c0.getx()
1 回目の参照です
>>> c0.getx()        #【参照が加算】
2 回目の参照です
>>> c1.getx()        #【別オブジェクトからでも……】
3 回目の参照です
>>> c0.getx()        #【回数加算】
4 回目の参照です
>>> c1.getx()
5 回目の参照です
>>>
```

4-7-3 静的メソッド(static method)

　静的メソッドは、言わば「クラスで定義されただけの、ただの関数」です。
　第１引数はそのままメソッド呼び出しの際の第１引数になり、何の変更もありません。

```
>>> class C:
...   def im(self):        #【普通のメソッド】
...     print(self)
```

```
...   @staticmethod        #【修飾子】
...   def sm0(self):
...     print(self)
...   def sm1(self):
...     print(self)
...   sm1 = staticmethod(sm1)    #【方法２】
...
>>> c = C()
>>> c.im()                 #【自分自身を出力】
<__main__.C object at 0x00BFE530>
>>> c.sm0()                #【引数不足でエラー】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm0() takes exactly 1 positional argument (0 given)
>>> c.sm1()                #【同じく】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm1() takes exactly 1 positional argument (0 given)
>>> c.sm0('hello')         #【ただの関数】
hello
>>> c.sm1('hello')
hello
>>> c.sm0(c)               #【普通のメソッドをエミュレート】
<__main__.C object at 0x00BFE530>
>>>
```

静的メソッドの成立には、歴史的な事情が関わっています。
Python2 までは、クラス属性として呼び出されたメソッドは『非束縛メソッド(unbound method)』といい、その第１引数に必ずそのクラスのインスタンスを引き渡さなければならない仕様でした。

```
>>> #【Python 2までのメソッドの使い方】
... class C:
...   def meth(self, x):
...     print(x)
...
>>> c = C()
>>> c.meth(100)
100
>>> C.meth(c, 100)  #【インスタンスに関係なくとも必ずインスタンスを渡す】
100
>>>
```

「非束縛」とは「暗示的に第１引数がインスタンスに束縛されていない」という意味なのですが、束縛されていないから自由かといえばそうでなく、必ず「そのクラスのインスタンス」のどれかを渡してやる必要があったのです。
更にこれは、インスタンスを作らないと、クラスに記述されているメソッドが一切使えないことも意味します。
クラス属性として呼び出したメソッドにインスタンスを引き渡して使う方法は、「継承」のテクニックとして現在も使用されていますが、要するにそういった用途でしか考えられていなかったと思われます。
クラスは独立した名前空間ですから、関数の容れ物(function holder)としての用途も当然考えられていました。
しかし関数ホルダーとして使うには、Python のクラス仕様は非常に使いづらかったのです。
そこで登場したのが『静的メソッド』です。
静的メソッドはクラスと独立しているので、無理やり第１引数にインスタンスを取る必要はありません。
これは、インスタンス無しでクラスのメソッドが使えることも意味します。
つまり静的メソッドの書式は、クラスを関数ホルダー的に使うために出来たのです。

ところが、この事情が Python3 の改訂で一変します。
Python3 では、クラスに登録されたメソッドをクラスの属性として呼び出すと、無条件に『普通の関数』つまり『静的メソッド』として扱うようになったのです。
これによって、関数ホルダーとして使うのであれば、わざわざ『静的メソッド』宣言する必要が無くなりました。
Python3 以降の静的メソッドは、インスタンスから呼び出しても第１引数をインスタンス自身にする

必要の無いメソッドなどを設定する際に使用します。
　例えば、そのオブジェクトの操作によく使用する関数的なツールを、クラスの中にまとめる場合などに便利です。
　とはいえ、Python2 以前に比べると必要度は若干減ったと言えるかもしれません。

4-7-4 第 1 引数に関する注意事項

　普通のメソッド（インスタンスメソッド）とクラスメソッドと静的メソッドは、第一引数が指し示すものが若干変化します。
　ここで比較のために、簡単な実験をしてみましょう。

```
>>> class C:
...   x = 'class attribute'
...   def im(arg): print(arg.x)         #【普通のメソッド】
...   @classmethod
...   def cm(arg): print(arg.x)         #【クラスメソッド】
...   @staticmethod
...   def sm(arg): print(arg.x)         #【静的メソッド】
...
>>> C.im()   #【引数不足】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: im() takes exactly 1 positional argument (0 given)
>>> C.cm()   #【暗示的にクラス属性をポイント】
class attribute
>>> C.sm()   #【引数不足】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm() takes exactly 1 positional argument (0 given)
>>> C.im(C)  #【クラス属性参照】
class attribute
>>> C.cm(C)  #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> C.sm(C)  #【クラス属性参照】
class attribute
>>> c = C()
>>> c.im()   #【クラス属性参照】
class attribute
>>> c.cm()   #【クラス属性参照】
class attribute
>>> c.sm()   #【引数不足】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm() takes exactly 1 positional argument (0 given)
>>> c.im(C)  #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: im() takes exactly 1 positional argument (2 given)
>>> c.cm(C)  #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> c.sm(C)  #【クラス属性参照】
class attribute
>>> C.im(c)  #【クラス属性参照】
class attribute
>>> C.cm(c)  #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> C.sm(c)  #【クラス属性参照】
class attribute
>>> c.im(c)  #【引数過多】
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: im() takes exactly 1 positional argument (2 given)
>>> c.cm(c)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> c.sm(c)   #【クラス属性参照】
class attribute
>>> c.x = 'instance attribute'          #【インスタンス属性セット】
>>> c.im()    #【インスタンス属性参照】
instance attribute
>>> c.cm()    #【クラス属性参照】
class attribute
>>> c.sm()    #【引数不足】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sm() takes exactly 1 positional argument (0 given)
>>> c.im(C)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: im() takes exactly 1 positional argument (2 given)
>>> c.cm(C)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> c.sm(C)   #【クラス属性参照】
class attribute
>>> C.im(c)   #【インスタンス属性参照】
instance attribute
>>> C.cm(c)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> C.sm(c)   #【インスタンス属性参照】
instance attribute
>>> c.im(c)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: im() takes exactly 1 positional argument (2 given)
>>> c.cm(c)   #【引数過多】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cm() takes exactly 1 positional argument (2 given)
>>> c.sm(c)   #【インスタンス属性参照】
instance attribute
>>>
```

まとめましょう。

【第１引数の示すもの】

	クラス属性呼び出し	インスタンス属性呼び出し
通常のメソッド	明示的指定必要	インスタンス
クラスメソッド	クラス	クラス
静的メソッド	明示的指定必要	明示的指定必要

余談ですが、クラスメソッドとインスタンス属性呼び出しの通常のメソッドのクラスは'method'、静的メソッドとクラス属性呼び出しの通常メソッドのクラスは'function'(関数)です。
メソッドには関数にはない'__self__'という属性があり、ここに第１引数の指し示すものを格納しています。

4-8 宣言文

4-8-1 オブジェクト宣言に関する特殊文

オブジェクトの宣言は、スコープに関する特別扱いの際にのみ行います。
宣言に用いるキーワードはglobalとnonlocalの２種類です。

4-8-2 global文

グローバル宣言はglobal文を用いて行います。

書式：global global_value
動作：オブジェクトのスコープをグローバルにする

関数内においてglobal文でグローバル宣言をしたオブジェクトが、その関数内で変更された場合、その変更はトップレベルで変更されます。

```
>>> a = 100
>>> def f():
...   global a
...   a = 200
...
>>> a
100
>>> f()
>>> a
200
>>>
```

グローバル宣言はネストされた（入れ子になった）関数の中で行われても、トップレベルに影響するものと判断されます。

```
>>> a = 100
>>> def f0():
...   def f1():
...     global a
...     a = 200
...   f1()
...
>>> a
100
>>> f0()
>>> a
200
>>>
```

グローバル宣言はクラス定義内でも行うことができますが、クラス定義は関数定義と異なり「定義された時点で実行」されますので、あまり意味はありません。
むしろ話がややこしくなるだけなので使うべき所は無いでしょう。

```
>>> x = 100
>>> y = 100
>>> class C:
...   global x      #【グローバル宣言】
...   x = 666
...   y = 777
...
>>> x
666                 #【クラス定義内の変更】
>>> y
100                 #【クラス内属性なので変更無し】
>>> C.x             #【クラス属性にならない】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'C' has no attribute 'x'
>>> C.y             #【クラス属性】
```

```
777
>>>
```

※もしクラス内におけるグローバル宣言に利用法があるとするならば、クラスをローカルの名前空間を持ったブロックとして使用し、その中で一連のコードを実行して、結果をグローバルとして返す場合などが考えられます（名前空間の汚染を避けた隔離ブロック）。しかし、クラス内のコードの実行タイミングを知らない人にとっては、決してわかりやすいコードにはなりません。大した手間もかかりませんから、素直に手続き的な関数を定義して実行すべきでしょう。

global 文によるグローバル宣言は、関数の副作用を増やし透明性を下げますので、必要以上は使用すべきでないのはプログラミングの常識の通りです。

4-8-3 nonlocal 文

nonlocal 文による非ローカル宣言は、グローバル宣言以上に使用する場所が限られます。
非ローカル宣言は「ネストされた関数又はメソッド内」に於いて「外側の関数内で定義されたオブジェクト」に対し「内側で宣言」されます。
非ローカル宣言されると、内側の関数でのそのオブジェクトの変更が外側の関数のオブジェクトの値に影響を及ぼします。

```
>>> def f():
...   x, y = 100, 100
...   def ff():
...     nonlocal x
...     x, y = 200, 200
...   ff()
...   return x, y
...
>>> f()
(200, 100)
>>>
```

非ローカル宣言された x のみが、内部関数 ff において変更されたことがわかります。
非ローカル宣言は、三重以上ネストした関数でも有効です。

```
>>> def f():
...   x = 100
...   def ff():
...     def fff():
...       nonlocal x
...       x = 200
...     fff()
...   ff()
...   return x
...
>>> f()
200
>>>
```

しかし、三重以上のネストには慎重な扱いが必要です。
もし、内側で変更が行われてしまえば、非ローカル宣言自体が無効になるからです。

```
>>> def f():
...   x = 100
...   def ff():
...     def fff():
...       nonlocal x
...       x = 200
...     fff()
...     x = 400            #【非ローカル宣言の取り消し】
...   ff()
...   return x
...
>>> f()
100                 #【非ローカル宣言が無かったことになる！】
```

```
>>>
```

ついでにコレもダメです

```
>>> def f():
...   x = 100
...   def ff():
...     def fff():
...       nonlocal x
...       x = 200
...     x = 400            #【ff 内に x をつくってしまう】
...     fff()       #【ff の x を変更】
...   ff()
...   return x
...
>>> f()
100
>>>
```

二番目の例は、途中で print してみると話がわかりやすくなります。

```
>>> def f():
...   x = 100
...   def ff():
...     def fff():
...       nonlocal x
...       x = 200
...     x = 400
...     fff()
...     print(x)
...   ff()
...   return x
...
>>> f()
200          #【ff 内の x】
100
>>>
```

nonlocal 宣言は、閉包(closure)などを実装する際に使用されることが想定されているようです。
ここでは閉包の一種である加算器(accumulator)の例を出してみます。

```
>>> def accumulator(x):
...   def acc(y):
...     nonlocal x
...     x += y
...     return x
...   return acc
...
>>> a = accumulator(50)
>>> b = accumulator(100)
>>> a(1)
51
>>> a(2)
53
>>> b(1)
101
>>> b(2)
103
>>> a(3)
56
>>>
```

accumulator 関数の引数である'x'が非ローカル宣言により、内部関数 acc 内部からしかアクセスできないようになっています。
新しく accumulator 関数から加算器が作られた場合には、別のメモリ空間に新しい'x'が作成されま

す（コードの通り、'a'と'b'は独立しています）

4-9 修飾子(decorator)

修飾子(デコレータ decorator)は、クラスメソッドや静的メソッドの定義などに使用された書式が一般化されたものです。
修飾子は関数あるいはクラスなど『呼び出し可能な』オブジェクトで、基本的には以下のように記述します。

```
@decorator
def function(..):
      定義
```

これは、以下の書式と全く同じ(構文糖衣)です。

```
function = decorator(function)
```

修飾子は引数を取ることが出来ます。

```
@dx(arg)
def func(...):...
```

これは、以下の書式と同等です。

```
func = dx(arg)(func)
```

さらに、修飾子を二重以上に重ねることもできます(『The Python Language Reference』より)

```
@f1(arg)
@f2
def func(): pass
```

これは、以下の書式と同等です（出典同上）

```
def func(): pass
func = f1(arg)(f2(func))
```

修飾子については、システムで使用されている例（クラスメソッド、静的メソッド、プロパティなど）で使用されている例を参考にした上で、使用場面を考えてください。
ここでは、簡単な使用例を一つだけ挙げておきます。

```
>>> def plus1(func):
...   return lambda x: func(x) + 1
...
>>> @plus1
... def f(x):
...   return x
...
>>> f(100)
101
>>>
```

元々修飾子は、文法中の例外を一般化するために生まれたような文法なので、応用はやや難しいようですが、関数の合成などをシンプルに記述できる強みがありますので、使い方によっては強力なツールになりえます。
しかし、可能だからといって、元の関数から類推できる使用法（引数のフォーマットなど）を強引に変更するような修飾子の使い方は、コードを見難くするので極力避けるべきでしょう。

```
>>> def d(func):
...   return lambda :func(100)    #【引数を取らない関数が返される】
...
>>> @d
... def f(x):                #【一つの引数を取る関数】
...   return x * 100
```

```
...
>>> f(50)                    #【修飾子の適用前の形式が無効】
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes no arguments (1 given)
>>> f()                              #【修飾子の適用前のとは違う形式が有効】
10000
>>>
```

なお、一時的に取った引数を無効にするようなデバッグテスト的な修飾子の使い方は、勿論可能でしょう。

4-10 記述子とプロパティ

4-10-1 記述子(descriptor)

記述子(ディスクリプタ descriptor)とは、オブジェクトの基本的な三つの振る舞い、すなわち「値を参照された時」「値を代入された時」「消去された時」に対して、どう反応するかを自由に設定できる『属性』を作るための記述方法です。
クラスから属性全体へのアクセスをコントロールするためには、__setattr__などの属性アクセスに関するメソッド(accessor)を設定すれば良いのですが、オブジェクトに属する一部の属性のアクセスだけをコントロールしたい場合には、この方法は大仰になります。
また、複数のオブジェクトで同様の振る舞いをする属性を持ちたい場合にも不便です。
このような場合には、記述子を用いると便利です。
記述子とは、'__get__','__set__','__delete__'の全て、あるいはいずれかのメソッドを実装したオブジェクトです。
記述子があるクラスの属性として登録されると、そのクラスのインスタンスの属性は、記述子のアクセスメソッドに記述されたとおりの振る舞いを行います。
つまり、値を参照された時は'__get__'メソッドに記述された振る舞いを、値を代入された時には'__set__'に記述された振る舞いを、そして消去された時には'__delete__'に記述された振る舞いを行います。
お話だけではなかなかイメージがわきませんので、簡単な実例を見ていきましょう。

```
>>> class dsc:      #【記述子の定義】
...   def __get__(self, instance, owner):
...     print('get!')
...   def __set__(self, instance, value):
...     print('set!')
...   def __delete__(self, instance):
...     print('delete!')
...
>>> class C:
...   d = dsc()     #【記述子を属性として登録】
...
>>> c = C()
>>> c.d             #【値の参照】
get!
>>> c.d = 100       #【値の代入】
set!
>>> del c.d         #【オブジェクトの消去】
delete!
>>> c.d
get!
>>>
```

それでは、記述子の三つのアクセスメソッドについて説明します。

書式：__get__(self, instance, owner)
動作：記述子が属性参照された時の振る舞いを記述する

'__get__'メソッドには、記述子が属性参照された時にどう振る舞うかを記述します。
self は記述子自身、instance はクラス経由で記述子を属性参照しているインスタンス、owner は、記述子を属性として記述したクラスです。

書式：__set__(self, instance, value)
動作：記述子に値が代入された時の振る舞いを記述する

'__set__'メソッドは、代入子'='などで値を代入された時の振る舞いを記述します。
value は代入された値です。

書式：__delete__(self, instance)
動作：記述子が属性消去された時の振る舞いを記述する

'__delete__'メソッドは、del 文でインスタンスから属性消去された時の振る舞いを記述します。
通常、インスタンスからクラス属性は直接消去できませんので（クラスを呼び出せば別ですが）、このような振る舞いに至ること自体、かなり特殊でしょう。
あるとすれば、同名の属性にインスタンス属性として値を代入した後に、インスタンス属性を消去する、といった使い道でしょうか。

さて、記述子の主な利用方法は、属性の保護のための隠蔽でしょうか。

```
>>> class Dummy:    #【ダミー記述子】
...   def __get__(self, instance, owner):
...     if hasattr(instance, '_v'):
...       return instance._v
...     else:
...       return owner._v
...   def __set__(self, instance, value):
...     instance._v = value
...   def __delete__(self, instance):
...     if hasattr(instance, '_v'):
...       del instance._v
...
>>> class C:
...   d = Dummy()   #【記述子を実装】
...   _v = 100            #【実際の値】
...
>>> c = C()
>>> c.d
100
>>> c.d = 200
>>> c.d
200
>>> del c.d
>>> c.d
100
>>>
```

ちなみに、ここまで通常の属性の振る舞いをエミュレートしてしまうと、逆にあまり意味は無くなります（インスタンスが持たない属性をインスタンスから属性消去した時にエラーは出ませんが）
同じように値を隠蔽／保護するとして、少し応用してみましょう。

```
>>> class Dummy:
...   def __get__(self, instance, owner):
...     if not hasattr(instance, '_v'):
...       instance._v = 0  #【隠蔽属性が無ければ初期化】
...     return instance._v
...   def __set__(self, instance, value):
...     instance._v = value
...   def __delete__(self, instance):
...     instance._v = 0          #【属性消去されたら初期化】
...
>>> class C:
...   d = Dummy()
...
>>> c = C()
>>> c.d                          #【隠蔽属性を初期化して参照】
0
>>> c.d = 100
```

```
>>> c.d
100
>>> del c.d
>>> c.d                                  #【隠蔽属性を初期化】
0
>>>
```

こうすれば、クラス属性として一々初期化することなく記述子を導入することができます（また、最初の参照があった時点でクラスの中に属性を設定／初期化することも、勿論可能です）
このように、記述子を定義する方法は、複数のクラスに同様な振る舞いを行う属性を設定する場合には便利です。
しかし、特定のクラスの特定の属性の振る舞いのみを設定する場合には、クラス外に記述子を書く方法は、やはり大仰なものになってしまいます。
この点をコンパクトに記述する方法が、以下に説明する「プロパティ」です。

4-10-2 プロパティ(property)

プロパティ(『資産』という意味)は、他のプログラミング言語では属性と同様の意味で使われることもありますが、Pythonでは「振る舞いを再定義された属性」という意味になります。
基本的には記述子をクラスの中で記述してしまうだけのことですので、用途については前項の「記述子」の項をご覧ください。
さて、とりあえずシンプルなプロパティーの実装例です(『The Python Standard Library』より)

```
>>> class C:
...     def __init__(self):
...         self._x = None
...     def getx(self):
...         return self._x
...     def setx(self, value):
...         self._x = value
...     def delx(self):
...         del self._x
...     x = property(getx, setx, delx, "I'm the 'x' property.")
...
>>> help(C.x)
Help on property:

   I'm the 'x' property.

>>> c = C()
>>> c.x
>>> c.x = 100
>>> c.x
100
>>> c._x
100
>>> del c.x
>>> c.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in getx
AttributeError: 'C' object has no attribute '_x'
>>>
```

propertyクラスのインスタンス(propertyはクラスです)である'x'は、'getx','setx','delx'によって、振る舞いを記述されています。
この場合、'x'は属性'_x'の完全なエイリアスとなっているので、'x'を通じて'_x'にアクセスするように設定されています。
まぁ、このコード自体は説明用でしかありませんので、潔いくらい何の意味もありません。
とりあえず書式をまとめておきましょう。

書式：property(getter=None, setter=None, deleter=None, doc=None)
動作：プロパティ(propertyクラスのインスタンス)を生成する

propertyクラスは、その名の通りプロパティのクラスです。

getter はプロパティを参照した時の振る舞いを記述したメソッド、setter はプロパティに値を代入した時の振る舞いを記述したメソッド、deleter はプロパティを消去しようとした時の振る舞いを記述したメソッドです。
property は、与えられたパラメータから記述子を生成するファクトリクラスと思われます。
とりあえずコレを使えば、外部に記述子を設定せずとも、手軽に『振る舞いを記述した属性』つまり『プロパティ』を記述することができます。
できます……が、「これ、まだあんまり手軽じゃない」とお嘆きの向きの為に、修飾子(decorator)を使ったさらに手軽なプロパティ記述方法をご紹介します(再び『The Python Standard Library』より)

```
>>> class C(object):
...     def __init__(self):
...         self._x = None
...     @property
...     def x(self):
...         """I'm the 'x' property."""
...         return self._x
...     @x.setter
...     def x(self, value):
...         self._x = value
...     @x.deleter
...     def x(self):
...         del self._x
...
>>> c = C()
>>> c.x
>>> c.x = 100
>>> c.x
100
>>> c._x
100
>>> del c.x
>>> c._x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '_x'
>>>
```

まず property を修飾子として getter 部分を記述し、プロパティである x の属性（として勝手に設定されている）setter と deleter を記述する、という形式になります。
さて、ここまでお膳立てが揃ったのに『無意味な例』だけでも面白く無いので、私がブログに書いたコードを掘り出して書いてみます。

```
>>> class Undead:
...   def __init__(self, v):
...     self._v = v
...   @property
...   def x(self): return self._v
...   @x.setter
...   def x(self, v): print('Go to hell!')
...   @x.deleter
...   def x(self): print('I shall be back!')
...
>>> zombie = Undead(666)
>>> zombie.x
666
>>> zombie.x = 777
Go to hell!
>>> del zombie.x
I shall be back!
>>>
```

変更しようとすると毒づき、消そうとしても消えないゾンビオブジェクト。
ついでに、これを旧式の書き方で書いてみます。

```
>>> class Undead:
...   def __init__(self, v):
...     self._v = v
...   def getx(self):
...     return self._v
...   def setx(self, v):
...     print('Go to hell!')
...   def delx(self):
...     print('I shall be back!')
...   x = property(getx, setx, delx)
...
>>> zombie = Undead(666)
>>> zombie.x
666
>>> zombie.x = 777
Go to hell!
>>> del zombie.x
I shall be back!
>>>
```

さらにおまけに、前出の記述子で書いてみましょう。

```
>>> class Necromancy:
...   def __get__(self, instance, owner):
...     return instance._v
...   def __set__(self, instance, value):
...     print('Go to hell!')
...   def __delete__(self, instance):
...     print('I shall be back!')
...
>>> class Undead:
...   x = Necromancy()
...   def __init__(self, v):
...     self._v = v
...
>>> zombie = Undead(666)
>>> zombie.x
666
>>> zombie.x = 777
Go to hell!
>>> del zombie.x
I shall be back!
>>>
```

見比べると、プロパティと記述子の共通点や相違点がはっきりわかると思います。

【第二回に続く】

(2010 年 3 月 15 日投稿；2010 年 3 月 27 日改訂)

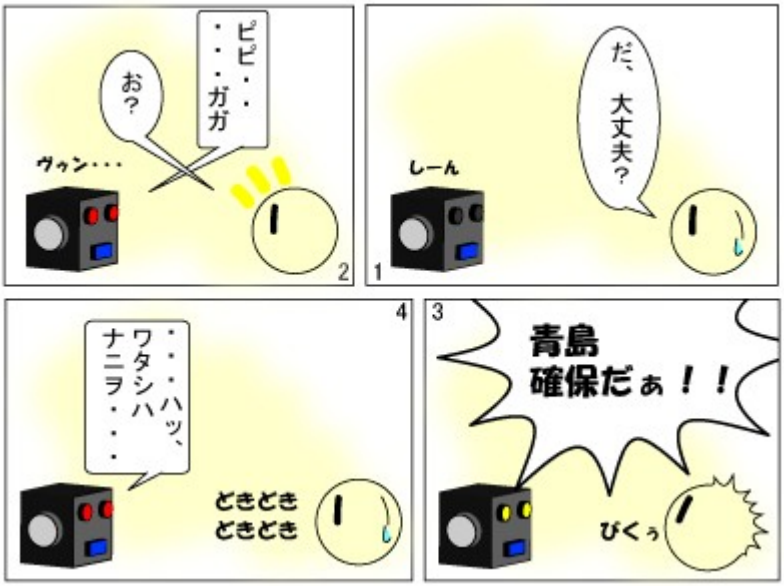
よしおさんとロボ太

海鳥

「乾燥注意報(前編)」

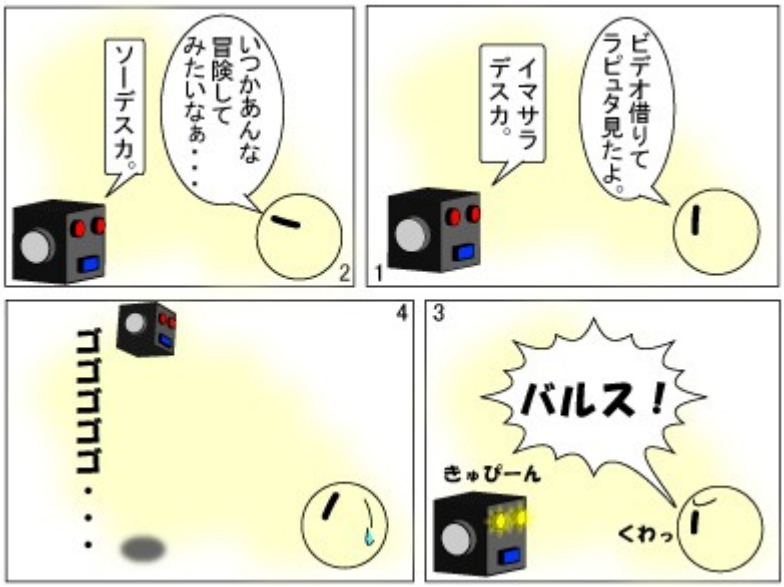
ロボ太？ロボ太ぁあああああああ！！

「乾燥注意報(後編)」



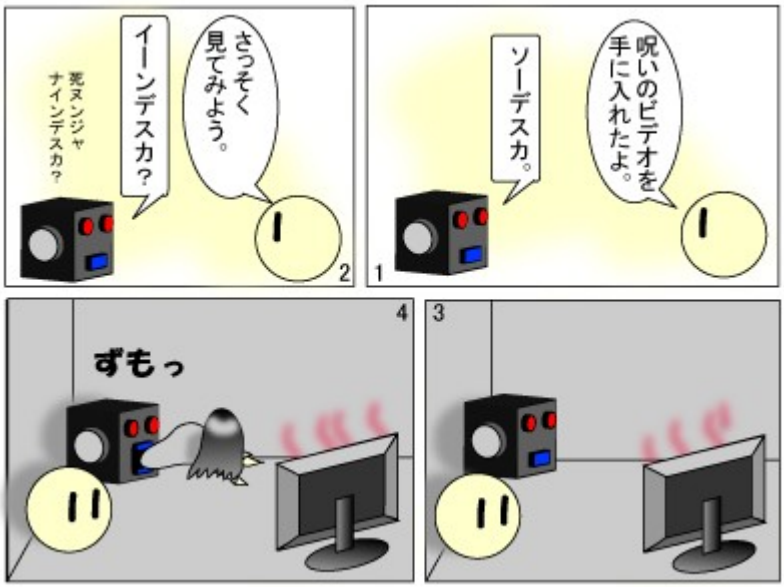
混信？

「ラピ○タ」



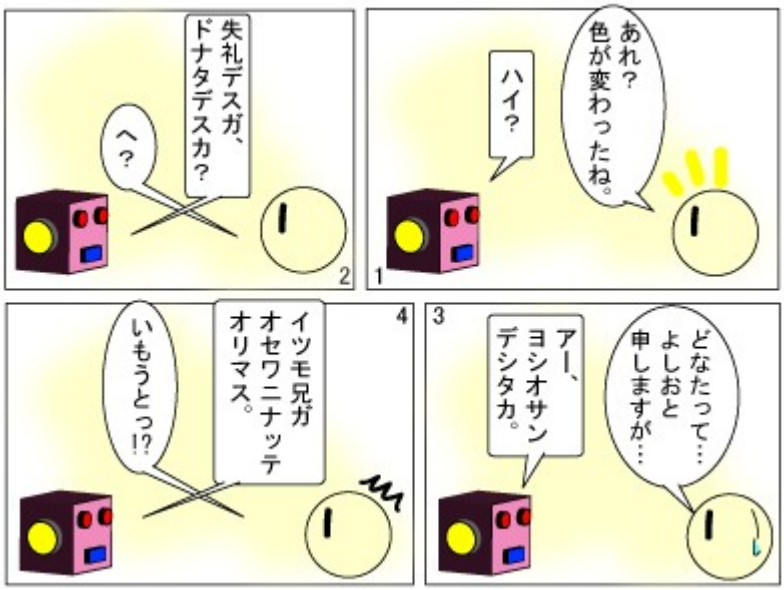
あのち～へい～せ～ん～♪ か～が～やく～の～は～♪

「ビデオ」



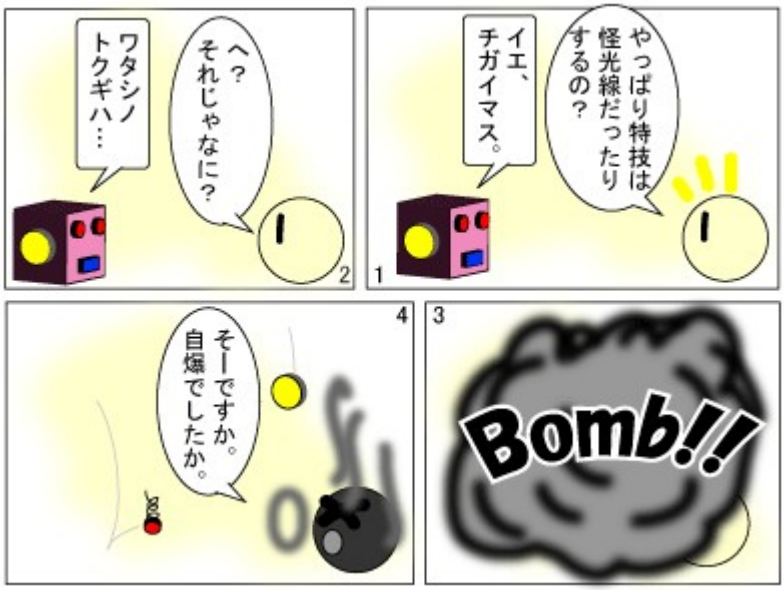
そこからっ！？

「色違い」



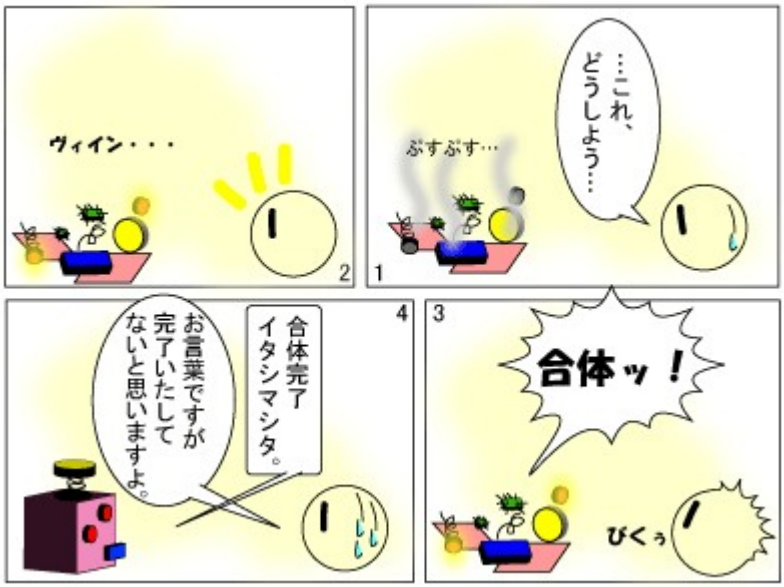
新キャラ出現(手抜きにも程がある)。

「続・特技」



ロボ子(仮)？ロボ子ぉおおおおおおおおお！！！

「真・特技」



ピカソ？

（転載：2010 年 3 月 27 日）

Zed 入門 III

「TAG 実行」を使ってみる

jscripter

1. はじめに

本稿では、Zed に新たに導入された「TAG 実行」機能を活用して、エディタ上のテキストの価値を高める方法を考える。

2. テキスト主義のための Zed

MS-DOS 時代は、今や伝説のテキストエディタ VZ のマクロユーザーからテキスト主義が生まれ、UNIX のテキスト処理ツールの流れから、SE 編集部編「MS-DOS テキストデータ料理学 sed,awk のある UNIX 流パソコン環境」(翔泳社、1992 年)や志村拓・鷲北賢・西村克信共著「AWK を 256 倍使うための本」(アスキー出版局、1993 年)などの本が出版される現象が見られた。

TSNET の母体であった FGALTS(テキスト&スクリプト)はそのまったただ中にいたと言えるだろう。VZ マクラー(マクロの使い手)は FGALDC(ドキュメント)に集結していたのだが、FGALTS にも顔を出されていたと思う。sed、awk、perl、python、ruby、tcl/tk などのテキスト処理ツールを取り扱う FGALTS と FGALDC は親和性が高かった。

テキスト主義においては、テキストファイルこそがコンピューティングにおける知的自由を確保する唯一の方法であり、自在にテキストを取り扱う方法が VZ のマクロであった。ワープロなどのアプリケーションに対するアンチテーゼであったのだ。

今や、時は 2010 年、既に 20 年近くが経過し、Web の発明とインターネットの進展とともにコンピューティング環境は様変わりしているが、コンピューティングをパーソナルな知的生産に役立てようという気持ちはまったく変わりがない。テキストの世界はインターネットの発展とともにより豊かなものとなっている。

著者はテキストエディタは大概のものに触っている。もちろん、VZ も例外ではない。秀丸も使わせていただいた。しかしながら、スクリプトをエディタ上でフルに使いたいがために、Dana、TeraPad、そして Zed へと遍歴を重ねている。

Dana や TeraPad は完成の域にあるのに対して、Zed はまだ α 版であり、発展途上にある。うれしいことにまだまだ進化する可能性が高い。そして、lua のスクリプトエンジンを編集用に内蔵するなど作者自身がスクリプトにも興味をお持ちである。現在では Perl などのスクリプトを自在に編集メニューに登録できるようになった。また、サブウインドウを持っているので、スクリプトの実行過程やエラーなどの表示も可能である。スクリプトの実行環境として大変興味深いものである。

3. isbn2amazon.pl と TAG 実行

最近、Zed には「TAG 実行」という機能が加わった。最初はどのようなものかよくわからなかったのだが、久世さんに促されて試してみた。使ってみて、エディタ上のテキストを編集するというよりは、テキスト自体からさらに情報を引き出すように使えることがわかった。典型的には、既に作者により実装されているが、URL 文字列から Web ブラウザを起動して表示するというものである。これだけなら他のエディタでも標準で実装しているものも多いだろう。しかし、このような機能を自分で自由にエディタに付加できるとしたらどのような可能性が生まれるだろう。

久世さんから促されたのは、以前作った isbn2amazon.pl である。Net::Amazon モジュールを使い、isbn 番号から Amazon の Web サービスを利用し書籍情報を HTML に変換して得るというものである。元々 Web 日記の編集に利用するスクリプトとして作成したものである。このスクリプトが「TAG 実行」で使えるのではないかということだった。元のスクリプトは次の記事にある。

Net::Amazon モジュールを使う
http://homepage1.nifty.com/kazuf/renewal_2008_09.html#perl_1221839696
Net::Amazon モジュールを使う II
http://homepage1.nifty.com/kazuf/renewal_2008_09.html#perl_1221869701

4. isbn2amazon.plのProduct Advertising API対応

スクリプトをTAG実行用に改造する前に、昨年必要となった「Product Advertising API対応」したものを次に示す。Zed上のテキスト編集に対応したものである。

Product Advertising API対応では、Amazon Webサービスへのアクセスキーだけでなく、シークレットキーも取得してセットする必要がある。次のリンクから取得できる。

Product Advertising API
https://affiliate.amazon.co.jp/gp/advertising/api/detail/main.html

次のスクリプトは、著者の環境に合わせて、Shift_JIS出力となっているので、必要に合わせて変更して欲しい。スクリプトはUTF-8の文字コードで記述する必要がある。現在の状況ではスクリプトは編集用スクリプトのディレクトリに格納しておく。Zedの編集におけるスクリプトの使用方法は、下記のURLから著者のTSNETスクリプト通信に掲載している「Zed入門」シリーズを参照していただきたい。

Zed Watching
http://text.world.coocan.jp/TSNET/?Zed%20Watching

スクリプトの動作は、例えば、

isbn: 978-4-00-431227-7

のようなエディタ上の文字列行を範囲選択して、Zedの編集メニューからISBN2AMAZONを選択して実行すると、

<img src="http://ecx.images-amazon.com/images/I/41Y8XJoGw-L._SL160_.jpg" align=LEFT hspace=20>村井 純著「インターネット新世代（岩波新書)」(岩波書店, 2010年)<br clear=ALL>

が置換出力される。

[isbn2amazon.pl]

```
#!/Perl5.10/bin/perl.exe
# Title = ISBN2AMAZON
# Input
# Output

# Copyright 2009,2010 jscripter(jscripter9[at]gmail[dot]com)

use Net::Amazon;
use utf8;
binmode STDOUT, ':encoding(cp932)';# Shift_JIS出力
$|=1;

my $isbn = "";
if($ARGV[0]){
	($isbn = $ARGV[0]) =~ s/^i*s*bn: (.+)$/$1/i;
}else{
while(<>){
	if(index($_, "¥x1A") > -1){
		last;
	}
	chomp;
	($isbn = $_) =~ s/^i*s*bn: (.+)$/$1/i;
}
}
my($isbn13, $localenum) = &isbn10to13($isbn);
my $locale = "";
if($localenum == 0){
	$locale = 'us';
```

```
}elsif($localenum == 1){
	$locale = 'uk';
}elsif($localenum == 2){
	$locale = 'fr';
}elsif($localenum == 3){
	$locale = 'de';
}elsif($localenum == 4){
	$locale = 'jp';
}

# Set your Amazon Access Key ###
my $ua = Net::Amazon->new(token => '<-- Your Amazon Access Key -->',
                    secret_key => '<-- Your Amazon Secret Access Key -->',
                    locale => $locale);
###

# Get a request object
my $response = $ua->search(isbn => $isbn13);

if($response->is_success()) {
	for ($response->properties){
		print "<img src=¥"", $_->ImageUrlMedium(),"¥" align=LEFT hspace=20>",
                              join("/", $_->authors), "著「", $_->title(), "」(",
                              $_->publisher(), ", ", $_->year(), "年)<br clear=ALL>";
	}
} else {
    print "Error: ", $response->message(), "¥n";
}

sub isbn10to13{
	my($isbn10) = @_;
	my $isbn13 = "";
	my $localenum = 0;
	$isbn10 =~ s/-//g;
	if($isbn10 =~ /^(¥d{9}).$/){
		$isbn13 = "978" . $1;
		$isbn13 =~ /^(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)$/;
		$localenum = $4;
		$e = $2 + $4 + $6 + $8 + $10 + $12;
		$o = $1 + $3 + $5 + $7 + $9 + $11;
		($f = $e * 3 + $o) =~ s/^¥d*(¥d)$/$1/;
		if($f == 0){
			$isbn13 .= "0";
		}else{
			$isbn13  .= 10 - $f;
		}
	}else{
		$isbn13 = $isbn10;
		$localenum = (split(//, $isbn13))[3];
	}
	return $isbn13, $localenum;
}
```

5. TAG実行用isbn2html.pl

まず、TAG実行用スクリプトの動作を説明しよう。現時点(Zed Ver.0.46)ではTAG実行用スクリプトは編集用スクリプトと同じ場所に格納しておく。

次にTAG実行の設定を行う必要がある。Zedメニューバーの「表示」→「タイプ別設定」ウインドウの「基本」→「ファイル名」→「タイプ別設定」ウインドウの「TAG実行」から「TAG名」、「TAG条件」、「実行コマンド」を設定する。設定用の画面を次に示す。

図 タイプ別設定の TAG 実行設定画面

TAG 条件は正規表現で記述する。図の TAG 条件にある「(i*s*bn:* *[¥d-]+)」という条件は「isbn: 978-4-00-431227-7」あるいは簡略化して「bn: 978-4-00-431227-7」、あるいは「ISBN978-4-00-431227-7」などにマッチする。I オプションは付いている状態のようだ。TAG 条件にマッチした文字列の()内の部分は「実行コマンド」の引数として「”¥1”」のように記述する。TAG 実行においてもスクリプト先頭行の「#!○○○ 」の指定は有効なので、実行コマンド部分ではスクリプト名を指定するだけでよい。数値部分を抜き出すのはスクリプト側に任せている。

編集テキストの行中に「ISBN978-4-00-431227-7」という文字列を含むと、カレットが行の上に重なると、「ISBN2HTML (Ctrl+クリック)」という黄色の下地に黒文字のタグが出る。その状態で、「Ctrl+クリック」すると、次のような画面になる。画像ではカレットが表示されていないが、行の上にある。

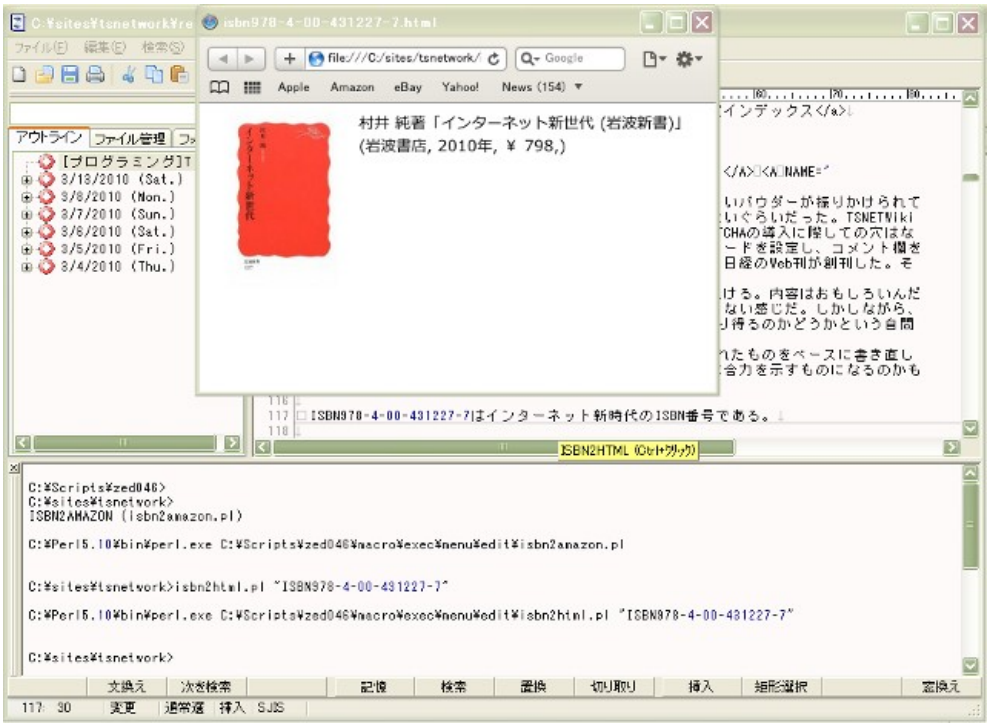


図 isbn2html.pl の TAG 実行画面

Amazon Web サービスの書籍情報を含む HTML を表示する Web ブラウザには Safari を採用している。シンプルなデザインが小さく表示したい場合には特に向いている。ブラウザのサイズや表示位置を指定するために、amazon2html.pl の出力する HTML に JavaScript を仕掛けてある。次の部分である。

```
<SCRIPT language="JavaScript">
<!--
function rsize(){
window.resizeTo(480,360);
x=(screen.width-480)/2
window.moveTo(x,0);
}
// -->
</SCRIPT></head>
<body onLoad="rsize()">
```

body が読み込まれたときに rsize メソッドが起動され、480x360 ピクセルサイズのブラウザが画面上端の中央に表示される。

本スクリプトは Amazon Web サービスから、UTF-8 文字コードの文字列を受け取るので、printf 出力では、UTF-8 で encoding し、UTF-8 の HTML を出力する。スクリプトは UTF-8 で保存しておく。

編集用のスクリプトの場合は、編集指定範囲を標準出力に置換する処理を期待するわけだが、TAG 実行の場合は、編集が目的ではなく、編集テキストの情報からパターンマッチングに基づいて別のアプリケーションを起動することになる。本スクリプトでは、ブラウザで表示させたい HTML を出力し、その HTML ファイルを関連付けから system 関数によって start コマンドでブラウザを起動している。次の部分である。

```
system("start isbn${isbn}.html");
```

[isbn2html.pl]

```
#!/Perl5.10/bin/perl.exe
# Title = ISBN2HTML
# Input

# Copyright 2009,2010 jscripter(jscripter9[at]gmail[dot]com)

use Net::Amazon;
use utf8;
use Encode;
$|=1;

my $isbn = "";
if($ARGV[0]){
	($isbn = $ARGV[0]) =~ s/^i*s*bn:* *([¥d¥-]+)$/$1/i;
}else{
while(<>){
	if(index($_, "¥x1A") > -1){
		last;
	}
	chomp;
	($isbn = $_) =~ s/^i*s*bn:* *([¥d¥-]+)$/$1/i;# ISBN番号の取得
}
}
my($isbn13, $localenum) = &isbn10to13($isbn);
my $locale = "";
if($localenum == 0){
	$locale = 'us';
}elsif($localenum == 1){
	$locale = 'uk';
}elsif($localenum == 2){
```

```
	$locale = 'fr';
}elsif($localenum == 3){
	$locale = 'de';
}elsif($localenum == 4){
	$locale = 'jp';
}

open(OUT,"> isbn${isbn}.html");
print OUT <<HEADER;
<html>
<head>
<meta http-equiv=¥"Content-Type¥" content=¥"text/html; charset=UTF-8¥">
<SCRIPT language="JavaScript">
<!--
function rsize(){
window.resizeTo(480,360);
x=(screen.width-480)/2
window.moveTo(x,0);
}
// -->
</SCRIPT></head>
<body onLoad="rsize()">
HEADER

my $ua = Net::Amazon->new(token => '<-- Your Amazon Access Key -->',
                    secret_key => '<-- Your Amazon Secret Access Key -->',
	locale => $locale);

# Get a request object
my $response = $ua->search(isbn => $isbn13);

if($response->is_success()) {
	for ($response->properties){
		print OUT encode('utf8',
		sprintf("%s%s%s%s%s%s%s%s%s%s%s","<img src=¥"", $_->ImageUrlMedium(),
		"¥" align=LEFT hspace=20>", join("/", $_->authors), "著「",
		$_->title(), "」(", $_->publisher(), ", ", $_->year(),
		"年)<br clear=ALL>"));
	}
} else {
    print "<p>Error: ", $response->message(), "</p>¥n";
}

print OUT "</body></html>¥n";
close(OUT);

system("start isbn${isbn}.html");

sub isbn10to13{
	my($isbn10) = @_;
	my $isbn13 = "";
	my $localenum = 0;
	$isbn10 =~ s/-//g;
	if($isbn10 =~ /^(¥d{9}).$/){
		$isbn13 = "978" . $1;
		$isbn13 =~ /^(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)$/;
		$localenum = $4;
		$e = $2 + $4 + $6 + $8 + $10 + $12;
		$o = $1 + $3 + $5 + $7 + $9 + $11;
		($f = $e * 3 + $o) =~ s/^¥d*(¥d)$/$1/;
		if($f == 0){
			$isbn13 .= "0";
		}else{
```

```
                $isbn13  .= 10 - $f;
            }
        }else{
            $isbn13 = $isbn10;
            $localenum = (split(//, $isbn13))[3];
        }
        return $isbn13, $localenum;
}
```

6. TAG 実行により Zed の可能性が広がる

著者はこれまで Zed を、HTML を編集し、編集ファイルを FTP で Web サイトに送り込むためのツールとして活用してきたが、TAG 実行機能はさらに活用領域を広げるのではと感じている。

本稿の isbn2html.pl は言わばエディタ上のマッシュアップである。エディタ上の情報と Web サービスを結びつけた例である。このような応用はいくらでも考えつくかもしれない。デスクトップのデータベースやファイルなど他のデスクトップリソースと結びつけることも考えられるかもしれない。

このような応用はテキストを編集するためのものではなくテキストをコマンド化するものと考えられる。しかし、この機能はやはりテキストを生成編集出力することにつなげていくべきだろう。新しいものが生成されなくてはおもしろくない。

編集用のスクリプトも TAG 実行に切り替えた方が、範囲指定をしなくても済むので便利になる場合が多いかもしれない。

現在、久世さんは Grep 機能の実装をされているそうだ。TAG 実行機能と連動していく可能性があるなあと思っている。著者は内蔵されている Lua を使って、Zed の持つ本来の編集機能を活用したアプリケーションも考えてみたいと思っている。まだ、見通せない霧の彼方にあるだけの思いだけなのだが・・・

さらに Zed が発展することを願ってやまない。

(2010 年 3 月 27 日投稿)

Sjis.pm ミニ入門

jscripter

ActiveState 社が、最近、jperl のベースとなる APi522e.exe など古いバージョンへのアクセスを制限した。Business Edition 以上の ActivePerl が必要となったのである。長年お世話になっているのではあるが、趣味のユーザーには、$999/Server/year の負担は厳しい。

Commercial open source language distribution & support for Perl, Python, Tcl
http://www.activestate.com/business_solutions/compare/

新たな趣味の jperl ユーザーが生まれているのかどうかは定かでないが、今は、稲葉準さんの Sjis.pm がある。

Sjis.pm
http://text.world.coocan.jp/TSNET/?Sjis.pm

Perl5.005_03、Perl5.8/5.10 で Sjis.pm を使えば、Shift_JIS で Perl スクリプトが書ける。著者も試し始めたばかりだが、一つの成果として、前稿の isbn2html.pl の SJIS 版を作成して動作を検証したので紹介しよう。文字コードの変換には、jacode.pl を使用する。

jacode.pl
http://text.world.coocan.jp/TSNET/?jacode.pl

下記のスクリプトを SJIS で保存しよう。jacode.pl の使い方は、jcode.pl とほぼ同じ。UTF-8 が取り扱えるので、Amazon Web サービスから返る UTF-8 文字列を SJIS に変換するために使っている。本スクリプトは SJIS の HTML を出力する。

本スクリプトの動作内容については UTF-8 の文字コードを出力する前稿のスクリプトを参照し、対比して欲しい。スクリプトは SJIS 文字コードで保存して使用する。

[isbn2sjishtml.pl]

```
#!/Perl5.10/bin/perl.exe
# Title = ISBN2SJISHTML
# Input

# Copyright 2009,2010 jscripter(jscripter9[at]gmail[dot]com)

use Net::Amazon;
use Sjis;
require 'jacode.pl';
$|=1;

my $isbn = "";
if($ARGV[0]){
	($isbn = $ARGV[0]) =~ s/^i*s*bn:* *([¥d¥-]+)$/$1/i;
}else{
while(<>){
	if(index($_, "¥x1A") > -1){
		last;
	}
	chomp;
	($isbn = $_) =~ s/^i*s*bn:* *([¥d¥-]+)$/$1/i;
}
}
my($isbn13, $localenum) = &isbn10to13($isbn);
my $locale = "";
if($localenum == 0){
	$locale = 'us';
```

```
}elsif($localenum == 1){
	$locale = 'uk';
}elsif($localenum == 2){
	$locale = 'fr';
}elsif($localenum == 3){
	$locale = 'de';
}elsif($localenum == 4){
	$locale = 'jp';
}

open(OUT,"> isbn${isbn}.html");
print OUT <<HEADER;
<html>
<head>
<meta http-equiv=¥"Content-Type¥" content=¥"text/html; charset=Shift_JIS¥">
<SCRIPT language="JavaScript">
<!--
function rsize(){
window.resizeTo(480,360);
x=(screen.width-480)/2
window.moveTo(x,0);
}
// -->
</SCRIPT></head>
<body onLoad="rsize()">
HEADER

# Set your Amazon Access Key ###
my $ua = Net::Amazon->new(token => '<-- Your Amazon Access Key -->',
                    secret_key => '<-- Your Amazon Secret Access Key -->',
                    locale => $locale);
###

# Get a request object
my $response = $ua->search(isbn => $isbn13);

if($response->is_success()) {
	for ($response->properties){
		print OUT "<img src=¥"", &jcode'sjis($_->ImageUrlMedium(),'utf8'),
		"¥" align=LEFT hspace=20>",
		&jcode'sjis(join(' ',$_->authors), 'utf8'), "著「",
		&jcode'sjis($_->title(),'utf8'), "」(",
		&jcode'sjis($_->publisher(), 'utf8'), ", ",
		&jcode'sjis($_->year(), 'utf8'),
		"年)<br clear=ALL>";
	}
} else {
    print "<p>Error: ", $response->message(), "</p>¥n";
}

print OUT "</body></html>¥n";
close(OUT);

system("start isbn${isbn}.html");

sub isbn10to13{
	my($isbn10) = @_;
	my $isbn13 = "";
	my $localenum = 0;
	$isbn10 =~ s/-//g;
	if($isbn10 =~ /^(¥d{9}).$/){
		$isbn13 = "978" . $1;
		$isbn13 =~ /^(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)(¥d)$/;
```

```
            $localenum = $4;
            $e = $2 + $4 + $6 + $8 + $10 + $12;
            $o = $1 + $3 + $5 + $7 + $9 + $11;
            ($f = $e * 3 + $o) =~ s/^¥d*(¥d)$/$1/;
            if($f == 0){
                  $isbn13 .= "0";
            }else{
                  $isbn13  .= 10 - $f;
            }
      }else{
            $isbn13 = $isbn10;
            $localenum = (split(//, $isbn13))[3];
      }
      return $isbn13, $localenum;
}
```

@nifty の@homepage サービスの新規受付は最近なくなってしまったが、ソラリスの Perl5.005_03 が CGI では動作している。通常なら、Shift_JIS で書いた CGI は使えないが、Sjis.pm を使えばできるようになる。他の CGI サービスでも 5.005_03 をまだ使っていれば試してみるとよいだろう。

もちろん、CGI サービスでは普通の Perl がインストールされていることがほとんどのはずなので、SJIS で CGI スクリプトを書くことはできない。これまで日本語圏ではほとんどの CGI スクリプトは euc-jp で書かれていたはずである。CGI スクリプトが SJIS で書ければ、少なかった SJIS の Web ページが増えるかもしれないのだ。Sjis.pm の応用範囲は意外と広いかもしれない。現在の Sjis モジュール(0.52)は他のモジュールに依存しないので、レンタルサーバーなどでも使用しやすいはずである。

小形克宏の「文字の海、ビットの舟」――文字コードが私たちに問いかけるもの
http://internet.watch.impress.co.jp/www/column/ogata/index.htm

を読むと、Shift_JIS がなくならない理由がよくわかる。すべてが Unicode になっていくはずというのは幻想なのかもしれない。文字コードのことなど気にしていない人がほとんどだし、どこにどれだけ、どの文字コードを使った文書が存在しているかなど誰にも把握できないことなのだ。そのような状況の中で OS が特定の文字コードしか使えないということはありえないのである。

無論、Sjis.pm を使わなくても、SJIS 文字コードのテキストは Perl で取り扱える。さて・・・どっちが簡単なのかよく考えてみよう^^)

(2010 年 3 月 27 日投稿)

ゲーム Cube

written by Yさ

1. このゲームは

※タイトルが某家庭用ゲーム機にとても良く似た感じになってますが、全く関係ありません。

元ネタの映画"Cube"は、立方体(キューブ)で構成されトラップが張り巡らされた謎の迷宮に、突如放り込まれた男女の脱出劇を描く1997年カナダの作品で、日本公開は1998年との事です。
(ウィキペディアより)

迷宮は、立方体の小部屋が複数繋がって構成されています。小部屋の上下左右前後の6面には扉があり、それぞれが別の小部屋、あるいは出口と繋がっています。

どの部屋も基本構造は全く同じですが、部屋によっては殺人的なトラップが仕掛けられている場合があります。

閉じ込められたあなたは、死のトラップが張り巡らされたこの立方体の迷宮の出口を見つけて無事脱出してください。

サイコロを振って...

2. 動作環境は

例によって、最近のawkならOKだと思います。

ただし、表示が日本語のため、それなりの環境が必要です(;^^ゞ

ちなみに動作確認は、

GNU Awk 3.1.7, 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 cube.awk[Enter]
```

等で起動するとゲームスタートです。

最初に迷宮のサイズ(＝小部屋の数)を入力してください。

27部屋(=3^3)を最小に1000部屋(=10^3)まで、3～10で指定できます。

ゲーム中は、現在いる部屋の状況を表示します。各部屋には部屋番号000～999が振られているので、一度入った事がある部屋を覚えておくのに利用してください。

全ての部屋の向きは固定で、扉の番号は 1が「天」, 6が「地」, 5が東, 2は西, 3は南, 4は北と定義しています。
(いってんちろく とうございに なんざんほくし)

この扉の番号を指定して別の部屋へ移動します。なお、部屋と部屋を繋ぐ扉の番号はゲーム中は基本的には固定です。

例えば、部屋Aから天井の扉1で部屋Bに移動できれば、部屋Bから床の扉6で部屋Aに戻れます。

ただし、物理的にはありえませんが、ある部屋から同じ部屋に複数の別々の扉で繋がっていることもあります。また、一部の扉が他の部屋と繋がっていない部屋もあります。
(元ネタには無い、本ゲーム固有の"仕様"です^^;)

「罠のある部屋に入る」＝常に「即死」というわけではなく、元ネタと同様に罠やセンサーの種類によっては回避したり、発動させずに通過することもできます。

部屋のタイプはゲーム終了するまで変わりません。同じ部屋に入ると何度でも同じイベントが発生します。

部屋のタイプ(≒発生するイベント)としては以下があります。

・ノーマル：罠やセンサーが何も無く、扉を指定して別の部屋へ自由に移動できる
・トラップ：サイコロを振り、特定の目が出ないと罠を回避できず死亡する
　あるいは、振られたサイコロの目を当てないと罠を回避できず死亡する
・移動制限：別の部屋へ移動する扉がランダムに制限され、自由に指定できない
　あるいは、サイコロを振り、特定の目が出るまでその部屋から動けない
・強制移動：強制的に部屋を移動させられる
　(例. 「3部屋移動」
　　→ 今いる部屋の扉, 次の部屋の扉, その次の部屋の扉 の3回分が強制的に決まる
　　なお、複数の部屋を一度に移動する場合は、移動途中の部屋の罠は動作しない)
　あるいは、部屋に入るとゲーム開始時の部屋に飛ばされる＝振り出しに戻る

後、脱出できずに終了した際、同じ部屋の配置の迷宮で再挑戦できるようにしてみました。
(配置を紙にマッピングしてプレイするような奇特な方がいらっしゃるかもしれないので...)

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。

想像が付く方もいらっしゃるかと思いますが、年初に『お正月といえば凧揚げ、独楽回し...やっぱ双六？』ってな感じで思い付き、"貧相な画面のゲーム"としてまとめてみました。

...原稿の形になったのは3月末でしたけど d(^^;

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。

[cube.awk]

```
##  cube   written by Ｙさ

BEGIN{
  # サイズ決定
  MAXSIZE=10;
  printf("¥nSize (3-%d) ?>", MAXSIZE);
  do{ max=""; getline max; }while(max<3 && MAXSIZE<max);
  rooms = max*max*max;
  way = int(rooms/2);

  # ゲーム準備
  initialize();
```

```
  # ゲームメイン
  do{
    nowPos=0;

    # 履歴表示用
    walk=0;
    delete history;
    delete times;
    history[0]=roomNo[0];
    times[roomNo[0]]=1;

    life=(max>5)?5:max;
    sc=0;  # 得点
    bonus=20;
    do{ disp(); which(); }while(life>0 && nowPos!=way-1);
    if(sc>0) sc += 10*max; # ボーナス加算
    if(life>0){ s="*** 脱出成功! おめでとう!! ***"; }
    else{       s="... あなたは しにました"; }
    printf("¥n¥n%s¥n¥n     (SCORE:%05d0)¥n", s, sc);
  }while( life==0 && tryAgain()=="y" );
  exit;
}
END{
  printf("¥n¥n## 部屋移動の履歴 ##");  pushEnter();
  delete times;
  printf("[%03d](%d)¥n", history[0], ++times[history[0]]);
  for(i=1; i<walk; ++i)
    printf(" -> [%03d](%d)¥n", history[i], ++times[history[i]]);
}
function tryAgain(  yn)
{
  print "¥n¥n もう一度、同じ迷宮に挑戦しますか？";
  printf("y:はい, n:もう止める >");
  do{ yn=""; getline yn; yn=tolower(yn); }while(yn!="y" && yn!="n");
  return yn;
}
function swap(tbl,p1,p2,   t){ t=tbl[p1];  tbl[p1]=tbl[p2];  tbl[p2]=t; }
function rndRange(size,   i)
{
  delete rndList;
  for(i=0; i<size; ++i) rndList[i]=i;
  for(i=0; i<size; ++i) swap(rndList, rnd(size), i);
}
function rnd(N){ return int(N * rand()); }  ##  乱数


##
## 初期化
##
function initialize()
{
  srand();

  # 作業用に rndList[] を使う
  rndRange(rooms);
  for(i=0; i<rooms; ++i){
    roomNo[i]=rndList[i];
    # i 番目の部屋の [,0]:タイプ、[,1～6]:扉と繋がっている部屋番号
    for(d=0; d<=6; ++d) room[i,d]=-1;
  }

  setRoomType(0, 0);      # room[0,] がスタート
  setRoomType(way-1, 0); # room[way-1,] がゴール
```

```
  # 少なくとも１本はゴールに着くルートを確保する
  for(i=0; i<=way-1; ++i){
    connect(i, i+1, rnd(6)+1); setRoomType(i, 4);
  }
  connect(0, rooms-1, rnd(6)+1); setRoomType(rooms-1, 6);
  # スタートはなるべく全ての扉を使えるようにする
  rndRange(rooms-1 -3);
  ofs=0;
  for(d=1; d<=6; ++d){
    do{
      if((p=rndList[d+ofs]+1)>=way-2) p+=3; # +3=スタートとゴールを繋がない
      if((cnt=startDetect(p))>0) ++ofs;
    }while(cnt>0);
    connect(0, p, d); setRoomType(p, 6);
  }
  # 残りを適当に繋ぐ
  m1=int((1   + way-2   +1)/2);
  m2=int((way + rooms-1 +1)/2);
  connectRange(1, m1-1,   m2,   rooms-1);
  connectRange(m1,way-2,  way,  m2-1);
  connectRange(1, way-3,  way+1,rooms-1);
  for(i=1; i<rooms; ++i) setRoomType(i, 7);
}
function setRoomType(pos, kind)
{
  if(room[pos, 0]==-1) room[pos, 0]=(rnd(10)>7)?0:(rnd(kind));
}
function startDetect(p,   d,c)
{
  c=0;
  for(d=1; d<=6; ++d) if(room[p,d]==0) ++c;
  return c;
}
function connect(base, tgt, st,   d)
{
  d=st;
  while(room[base, d]>-1 || room[tgt, 7-d]>-1){
    if(++d>6) d=1;
    if(d==st) return 0;
  }
  room[base, d]=tgt;
  room[tgt, 7-d]=base;
  return d;
}
function connectRange(b1_s, b1_e, b2_s, b2_e,   i,n,c,p,x)
{
  c=b1_e+1 - b1_s;
  n=b2_e+1 - b2_s +c;
  for(i=0; i<n; ++i){
    if((p=rnd(n))==i) continue;
    connect((x=cnvPos(i,c,b1_s,b2_s)), cnvPos(p,c,b1_s,b2_s), rnd(6)+1);
  }
}
function cnvPos(b, lv, t1, t2){ return (b + ((b>=lv)?(t2-lv):t1)); }


##
## 扉指定
##
function which(  p,tmp,x,y)
{
  if(life==0 || nowPos==way-1) return;
  do{
```

```
    printf("Which Door?>"); do{ tmp=""; getline tmp; }while(tmp<1 || 6<tmp);
    p=room[nowPos, tmp];
    if(p==-1){
      print "¥n そこには何もありません。";
    }else if(p>=10000){
      print "¥n その扉はロックされています。";
    }
  }while(p<0 || p>=10000);
  move(p);
}
function move(nextPos)
{
  history[++walk]=roomNo[(nowPos=nextPos)];
  if(++times[roomNo[nowPos]]==1) ++sc;
  print "¥n 移動しました。";
}


##
## 状況表示
##
function disp(  n,d,type,cnt,visit,used)
{
  type=room[nowPos, 0];
  # 表示前に扉のロック/ロック解除を行う
  if(type==1){
    doorUnlock();
    if((cnt=unique()-1)>=1)
      event01(cnt);              # 一部の扉をロック
    else
      type=(room[nowPos, 0]=2);  # ロック対象外が作れないなら、全ロックに変更
  }
  if(type==2) doorLock();

  dispRoom();
  print "";
  if(type==1){
    print " 扉がロックされました!";
    print " 移動できる部屋が制限されました。";
  }else if(type==2){
    n=(rnd(10)>1)?2:3;  d=rnd(6)+1;
    print " すべての扉がロックされました!";
    printf(" %d が%d 回連続して出るまで他の部屋へ移動できません。¥n", d,n);
    event02(n,d,bonus);
  }else if(type==3){
    if((n=makeVisit(visit,used))>0){
      printf(" %d 部屋強制的に移動させられます!¥n", n);
      event03(n,visit,used);
    }
  }else if(type==4){
    n=rnd(5)+1;
    print " サイコロが振られ、出た目が１つ記録されました。";
    printf(" %d 回以内に何の目か当てないとトラップが発動します。¥n", n);
    event04(n);
  }else if(type==5){
    n=rnd(5)+1;  rndRange(6);
    print " センサーに引っ掛かりました!";
    printf(" %s の目が出ないとトラップが発動します。¥n", rndListToStr(n));
    event05(n);
  }else if(type==6){
    printf(" 部屋番号%03d へ飛ばされます!¥n", roomNo[0]);
    pushEnter();
    event06();
```

```
  }
}
function safe()
{
  print " トラップを回避できました。";
  sc+=10;
  dispRoom();
}
function dead()
{
  print " 回避不能! トラップが発動しました!!";
  if(--life>0)
    printf(" ダメージを受け、生命力が１つ減りました。(残り%d)¥n", life);
  pushEnter();
  if(life>0) dispRoom();
}
function shootDice(unfair,  e)
{
  printf(" サイコロを振ります。"); pushEnter();
  printf("%d が出ました。¥n¥n", e=(unfair>0)?unfair:(rnd(6)+1));
  return e;
}
function rndListToStr(n,  s,d,list,i,j)
{
  delete list;
  for(d=0; d<n; ++d) list[d]=rndList[d]+1;
  for(i=0; i<n-1; ++i)
    for(j=i+1; j<n; ++j) if(list[i]>list[j]) swap(list,i,j);
  s=sprintf("%d", list[0]);
  for(d=1; d<n; ++d) s=s sprintf("か%d", list[d]);
  return s;
}
function pushEnter(  tmp){ printf(" << push [Enter] >>");  getline tmp; }
function dispRoom(  d,n,p)
{
  n=times[(p=roomNo[nowPos])];
  printf("¥n現在位置：部屋番号%03d (%s)¥n", p, (n>1)?(n+""):("初"));

  for(d=1; d<=6; ++d){
    p=room[nowPos, d];
    printf(" [%s]", (p==-1)?(" "):((p>=10000)?("x"):(d+"")));
  }
  print "";
  for(d=1; d<=6; ++d){
    if((p=room[nowPos, d])>=10000) p-=10000;
    if(p==-1) printf(" ***"); else printf(" %03d", roomNo[p]);
  }
  print "";
}
function unique(  c,d,list,p)
{
  delete list;
  for(d=1; d<=6; ++d) if((p=room[nowPos, d])!=-1) ++list[p];
  for(d in list) ++c;
  return c;
}
function doorLock(  d)
{
  for(d=1; d<=6; ++d)
    if(room[nowPos, d]!=-1) room[nowPos, d]+=10000;
}
function doorUnlock(  d)
{
```

```
  for(d=1; d<=6; ++d)
    if(room[nowPos, d]>=10000) room[nowPos, d]-=10000;
}


##
## 移動制限 (type1)
##
function event01(cnt,   n,d,p,c,ofs)
{
  rndRange((n=rnd(cnt)+1));
  c=0;
  ofs=rnd(6)+1;
  do{
    d=rndList[c]+ofs;
    if(room[nowPos, d]==-1){
      if(++ofs>6) ofs=1;
    }else{
      room[nowPos, d]+=10000; ++c;
    }
  }while(c<n);
  for(d=1; d<=6; ++d){
    if((p=room[nowPos, d])>=10000){
      p-=10000;
      for(c=1; c<=6; ++c)
        if(c!=d && room[nowPos, c]==p) room[nowPos, c]+=10000;
    }
  }
}


##
## 移動制限 (type2)
##
function event02(n, d, bonus,   c,e)
{
  printf(" (%d 回以内に解除できると、ボーナス点を獲得できます)¥n¥n", bonus);
  c=1;
  do{
    printf(" %d/%d 回目 :", c,n);
    if(shootDice((bonus>0)?0:d)==d) ++c; else c=1;
    if(--bonus<0) bonus=0;
  }while(c<=n);
  print " 扉のロックが解除されました。";
  if(bonus>0) printf(" (ボーナス点 %05d0)¥n", bonus); sc+=bonus;
  doorUnlock();
  dispRoom();
}


##
## 強制移動 (type3)
##
function event03(cnt,visit,used,   i,p,nx)
{
  printf(" %3s%d: 扉[%d]¥n", "", 1, used[0]);
  for(i=1; i<=cnt-1; ++i) printf(" -> %d: 扉[%d]¥n", i+1, used[i]);
  pushEnter();
  p=nowPos;
  for(i=0; i<=cnt-1; ++i){
    nx=room[p, used[i]];  p=nx;
    if(i!=cnt-1){
      print "¥n 移動しました。";
```

```
      printf("¥n 現在位置：部屋番号%03d¥n", roomNo[p]);
    }
  }
  move(p);
  disp();
}
function makeVisit(visit,used,    last,able,cnt,nx,p,same,h,d)
{
  delete visit;
  delete used;
  last=rnd(3)+1;
  p=visit[0]=nowPos;
  cnt=1;
  do{
    delete able;  num=0;
    for(d=1; d<=6; ++d){
      nx=room[p,d];
      if(nx==-1 || nx==way-1) continue;
      same=0;
      for(h=0; h<cnt; ++h) if(nx==visit[h]) ++same;
      if(same==0) able[num++]=d;
    }
    if(num==0) exit;
    visit[cnt]=room[p, (used[cnt-1]=able[rnd(num)])];
    p=visit[cnt];
  }while(cnt++<=last)
  return cnt-1;
}


##
## トラップ (type4)
##
function event04(n,  ans,tmp)
{
  ans=rnd(6)+1;
  do{
    printf(" 残り%d 回¥n", n);
    printf("Which ?>"); do{ tmp=""; getline tmp; }while(tmp<1 || 6<tmp);
    if(ans==tmp) break;
    print " ...はずれ!";
  }while(--n>0);
  if(ans!=tmp){
    printf(" (正解は%d)¥n¥n", ans);
    dead();
  }else
    safe();
}


##
## トラップ (type5)
##
function event05(n,   e,d)
{
  e=shootDice(0);
  for(d=0; d<n; ++d)
    if(e==rndList[d]+1){ safe(); return; }
  dead();
}


##
```

```
## 振り出しに戻る (type6)
##
function event06()
{
  ++times[roomNo[(nowPos=0)]];
  dispRoom();
}
```

(2010 年 3 月 30 日投稿)

編集後記

jscripter

著作者の皆様、ご苦労さま。まだ最後の追い上げで頑張っていただいている方もおられるのだが、私自身の仕事もまだ終わらない。編集者は最後の走者であるから当然だけど。

通算第8号は90ページを超え、100ページも超えてしまうかもしれない。世の中はiPadの登場で、日本でも電子書籍ブームが起きようとしている。米国では既にAmazonのKindleでブームとなっているのだそうだ。我らがTSNETスクリプト通信は2年前からPDFによる電子出版を試みている。そのようなことが、OpenOffice.orgのおかげで簡単にできるのである。ePubの形式に変換することもそれほど難しくはないだろうと思うが、今のところePubのフォーマットを選択する必然性がない。

いずれは著作者毎に編纂した著作集のようなものも簡単に作れるだろうし、作者の方々もそう考えているに違いない。

ePubについて言えば、むしろWebの表現として考えてみたいという感じはする。Webのページの制御をどのように行うのかというのは一つの課題なのである。DocBookのようなものは既にあるのだけど。

DocBook - Wikipedia
http://ja.wikipedia.org/wiki/DocBook

すべてはテキストを廻っている。僕らはテキストを自在に操るスクリプタである。と、宣言したいところだが、内実は足りていない。まだまだ修行が必要だ。さて、どこに向かうか。気の赴くままに・・・

(2010年3月28日投稿)

ＴＳＮＥＴスクリプト通信

ISSN 1884-2798　出版地：広島市

2010年7月3日	2.4.010版
2010年4月2日	2.4.008版(表紙調整、巻頭言改訂)
2010年4月1日	2.4.007版(エイプリルフール版)
2010年3月27日	2.4.003版

投稿規程

TSNETWiki ： 「投稿規程」のページを参照のこと

編集委員会(投稿順)

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

表紙・ロゴデザイン

mono966.org
http://mono966.org/

著作権

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

使用許諾・配布条件

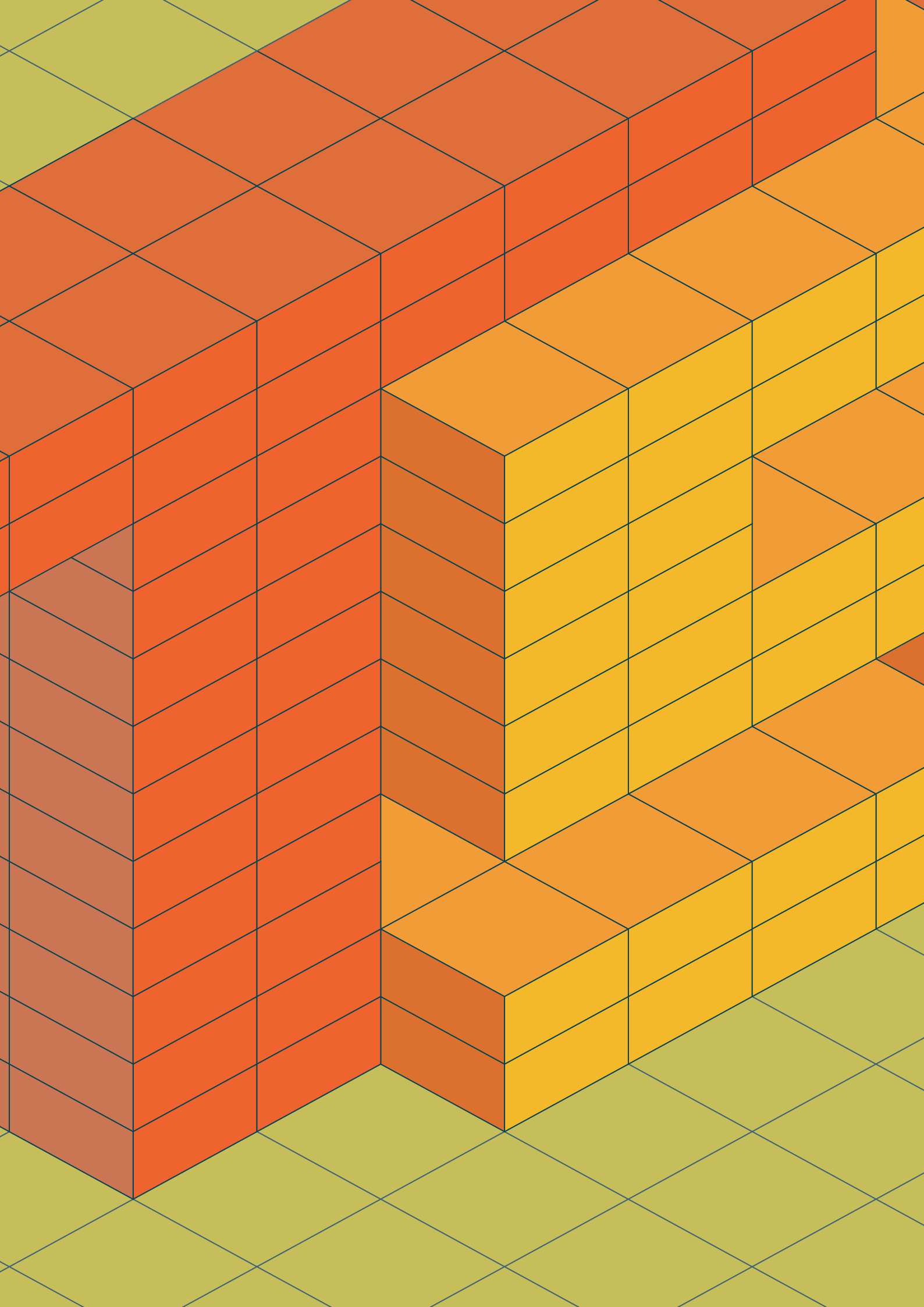
1. 編集委員会は「TSNETスクリプト通信 2.4.xxx版」を、ファイル名が「tsc_2.4.xxx.pdf」のPDFファイルとして無償で配布します。また、ファイル名、ファイル内容を一切改変しない状態での電子的再配布および印刷による再配布を無償で許諾します。
2. 関連するスクリプトファイルなどのプログラムについては、使用および再配布を無償で許諾しますが、改変後の再配布についてはオリジナルの著作権を併記することを条件に無償で許諾します。
3. 記事およびスクリプトファイルなどのプログラムに著作者の使用許諾・配布条件の記載がある場合は、著作権の項および上記2項に優先するものとします。

免責事項

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

編集ソフトウェア

OpenOffice.org 3.2.0 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