Python 入門 (初級編)

リリース ***0.1***

Noboru Yamamoto

2017 年 **12** 月 **15** 日

目次

Date 2008 年 12 月 31 日 水曜日/converted Sphinx 2017 年 12 月 15 日

Version 0.1 / 0.1

Author Noboru Yamamoto 山本 昇

Organization Accelerator Control Group, Accelerator Laboratory, KEK, JAPAN and Graduate University and 高エネルギー加速器研究機構 加速器第二研究系

第 1 章

Python Programming をはじめよう。

1.1 Python を使ってみよう

Python を使う方法には幾つかの方法があります。大きく分けると、対話的利用と非対話的利用のふたつです。非対話的利用では Python プログラム (Script としばしば呼ばれる）をファイルの形で用意して、それを Python 本体に実行させるという 2 段階を経由します。

`/usr/lib/python2.x/site-packages`

1.2 Python プログラム (Script) ファイルの作成

Python プログラムを含むファイルを作るのには、多くの方法があります。どの方法を用いても出来上がるのは、text 形式の Python プログラムを含むファイルです。ここではこのファイルを **MyHelloWorld.py** `Python` **見本プログラム**と呼びましょう。

Python プログラムファイルを作成するには、text 形式ファイルを作成する事ができるどのようなアプリケーションも利用可能です[*1]。 しかしながら、Emacs など Python の文法を理解して支援してくれるアプリケーションを使う方がいいでしょう。Eclipse などの開発環境にも Python のプログラム作成を支援してくれる拡張機能が用意されています。これらを使うのも良い方法です。Python 向けの開発環境として Python IDLE などのアプリケーションも存在しています[*2]。

お好きな方法を使って、次の Python プログラム:

[*1] 例えば、Windows であれば』『メモ帳』『、MacOSX であれば』『テキストエディット』『,Linux では』『Text Edit/gedit』『などがあります。

[*2] 筆者 (NY) は Emacs に pyhton-mode.el を設定することで、Python プログラム作成も Emacs 環境で行っています。Emacs はプログラム作成に適した環境を提供してくれる素晴らしいアプリケーションではありますが、コンピュータプログラム作成の初心者にはちょっと最初の敷居が高いかも知れません。IDLE は Python の標準的な配布パッケージに含まれています。Python 専用の開発環境であり、プログラムの開発からテストまでを一つのアプリケーションの中で行う事ができます。Eclipse+PyDev は汎用の (Java base) 開発環境に Python 開発用の拡張機能 PyDev を追加した物です。PyCharm は Free 版 (Community Edition) と有償版 (Professional version) が存在します。Editra はフリーなプログラムエディタであり、Python のプログラム開発支援環境を備えています。Komodo は ActiveState 社が販売している Python 開発環境です。筆者は Komodo を使った経験はありませんが、Active State 社は Python のパッケージの提供など Python コミュニティではよく知られた会社です。Python(やその他の言語の) 開発の有償のサポートなども販売しています。

```
.. code-block:: python
```

#!/usr/bin/env python #-- *coding:utf-8* --

import sys,math

r=float(sys.argv[1]) s=math.sin(r)

print u」こんにちは」.encode(「utf-8」) print 「Hello World!」,」sin(「,r,」) = 「,s print u」さようなら」.encode(「utf-8」)

を内容とするテキストファイル MyHelloWorld.py を作ってみましょう。[*3] このプログラムの各行の意味はすぐ後で、詳しく説明します。いまはただ、このプログラムを入力して、保存してください。

注釈: フリーウェアとして入手可能な **IDLE** と **PyCharm** のプログラム編集中の画面を *IDLE のファイル編集画面* と *PyCharm のファイル編集画面 + デバッグ画面* に挙げておきます。 (追記：Python の開発環境としては、 IPython, spyder なども出てきています。)

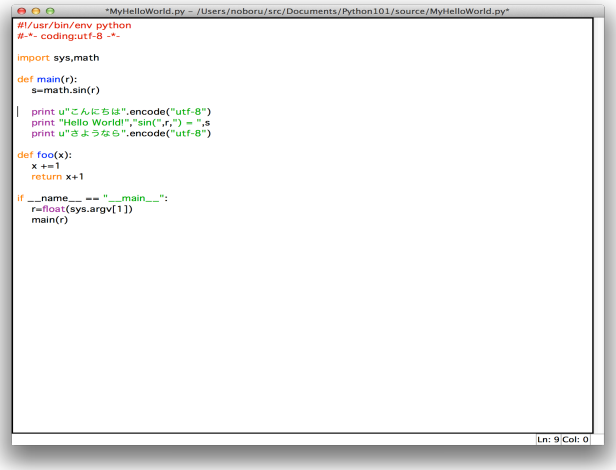

図-1.1　IDLE のファイル編集画面

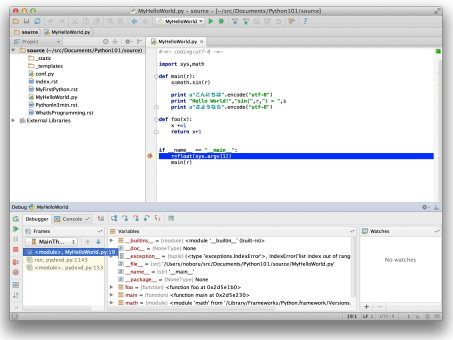

図-1.2　PyCharm のファイル編集画面 + デバッグ画面

[*3] このプログラムは、」Python Scripting for Computational Science」からお借りしました。

1.3 プログラムの実行

こうやって作成した Python プログラムを含むファイル MyHelloWorld.py を実行するにも幾つかの方法があります。もっとも基本的な方法は端末プロンプトで python コマンドを使って実行する方法です。:

```
% python MyHelloWorld.py 1.4
こんにちは
Hello World! sin( 1.4 ) =  0.985449729988
さようなら
```

python コマンドの引数として、作成した Python プログラムファイルの名前』『』MyHelloWorld.py』『』とこのプログラムが必要とする数値の引数 (ここでは 1.4 とした）を与えて実行しています。

此処で一つ注意があります。Pythoh がその他の program 言語と大きく異なっている事の一つに、」空白文字」[*4]が意味をもつと言う事があります。上のプログラムでも各行の最初に空白文字を入れてはいけません。

例えば:

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import sys,math

r=float(sys.argv[1])
 s=math.sin(r)

print u"こんにちは".encode("utf-8")
print "Hello World!","sin(",r,") = ",s
print u"さようなら".encode("utf-8")
```

のように余分な空白文字があると、:

```
%python MyHelloWorld.py 1
  File "MyHelloWorld.py", line 7
    s=math.sin(r)
    ^
IndentationError: unexpected indent
```

というように、実行時のエラーとなってしまいます。

また、ファイル名に誤りがあると、:

```
%python MyGoodbyWorld.py 1.4
/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/
↪MacOS/Python: can't open file 'MyGoodbyWorld.py': [Errno 2] No such file or␣
↪directory
```

と言うようなエラーメッセージ (『』No such file or directory』『) が表示されます。このエラーメッセージはお使いの OS (Windows/Linux/MacOSX) では、別の表現になっているかも知れません。あるいは、実行時のコマ

[*4] 空白およびタブ

ンド行の入力で最後の引数を忘れると、:

```
%python MyHelloWorld.py
Traceback (most recent call last):
  File "MyHelloWorld.py", line 6, in <module>
    r=float(sys.argv[1])
IndexError: list index out of range
```

というように、別のエラーメッセージ (『』 IndexError: list index out of range』『) が表示されます。プログラム本体に、綴りの間違いがあると:

```
Traceback (most recent call last):
  File "MyHelloWorld.py", line 7, in <module>
    s=math.sim(r)
AttributeError: 'module' object has no attribute 'sim'
```

と言うようなエラーメッセージが表示されます。エラーメッセージが示している様に、』『sin』『とあるべきところが』『sim』『となっているためです。プログラム初心者にとってエラーメッセージがでることは嬉しいことではありませんが、エラーメッセージをよく読んで、プログラムファイルの内容やコマンド入力行が正しくなるまで、修正を根気よく加えて行きましょう。[*5]

注釈: **IDLE** や **PyCharm** などの Python プログラム開発環境では、ファイルを編集中の画面から、直接編集中のファイルを実行するなどの機能が用意されています。また、エラーが発生した場合には、ソースコード中のエラーの場所を表示あるいはその場所への移動がサポートされていることもあります。プログラム作成の方法が一通りわかったところで、これらの機能の使い方を調べてみるといいでしょう。

脚注

[*5] このようなプログラムの修正作業をデバッグ (debug)、あるいがバグ取り、虫取り、虫出しなどとよびます。ようするに』『』しらみつぶし』『』と言う事ですね。一つずつプログラムの誤り（バグ=虫) を潰して行きましょう。

第 2 章

Python　入門

2.1 Python プログラムの実行

さあ Python プログラムを動かしてみましょう。[*6]

2.1.1 Python 処理系の起動

Python 処理系は、Linux や MacOSX では標準でインストールされています。[*7]ターミナルでコマンド 『』『python』『』 を入力することで、Python 処理系を起動します。:

```
%: python
Python 2.5 (r25:51908, Mar  6 2007, 14:03:40)
[GCC 3.2.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

python が起動されると、上図のように」version」等が表示されます。>>> は標準的な Python の入力プロンプトで、ユーザに次の入力を求めています。

2.1.2 python の文の直接実行

>>>は標準的な Python の入力プロンプトです[*8]。ここに python の一文を入力することで直接 python の文を実行／評価できます。

```
>>> print "123 +456 is ", 123+456
123 +456 is  579
>>>
```

[*6] この文書は当初メールで配布されました。一日一通一セクションで作成されていました。章立てはその影響をのこしています。

[*7] Windows ではご自分で Python 処理系をインストールする必要があるかもしれません。http://www.python.org/ などからインストーラーをダウンロードすることで簡単にインストール出来ます。また、Linux, MacOSX でも、これらのサイトからソースコードやインストーラーをダウンロードしてインストールすることで、最新版の Python 処理系を使う事が出来る様になります。

[*8] Python 処理系はインタプリタと呼ばれる形式のプログラム言語処理系です。Python プログラムの入ったファイルを与えてプログラムを実行する他に、端末/入力用ウィンドウから直接キーボードで Python プログラムを入力して、その結果を直ちに表示させることができます。これを Python 処理系の対話型処理と呼びます。

2.1.3 スクリプトの実行。

ファイル Hello.pyPython 見本プログラム を作成します。[*9]

```
%cat Hello.py
#!python
print "Hello, World."

% python Hello.py
Hello, World.
%
```

cat コマンドで作成したプログラムファイルの中身を表示しています。ファイルの第一行目はコメント行で、Unix 式にはこのファイルが **python** コマンドで実行される事を宣言しています。ファイルの二行目 ' print「Hello, World.」は *Python* プログラムの文で、端末に *'Hello, World.* と表示することを指示しています。次のプロンプトでは **python** コマンドにプログラムファイルの名前を与えて、実行しています。プログラムは「Hello, World.」と印刷して終了します。

このままでも立派な Python プログラムの一つではあるのですが、これから長くプログラムを書き続けていく最初の一歩としては、次の形のプログラム Python 見本プログラムにしておくのが良いでしょう。

```
%: cat Hello.py
#!/bin/env python
# -*- coding: utf-8 -*-

def main():
    print "Hello World."

 if __name__ == "__main__":
     main()
```

この python プログラムを **python** コマンドの引数として与えることで、プログラムが実行されます。実行結果はこちら

```
%: python Hello2.py
Hello World.
```

プログラムの実行結果は変わらないのになぜこの形をお奨めするのでしょうか。計算機プログラムを長く続けて行く上では、プログラムの再利用が重要なポイントとなります。Python ではプログラムの再利用を支援する仕組みとして **import** 文が用意されています。作成したプログラムをライブラリ (Python ではむしろ***モジュール***と呼びます) として別の応用プログラムで利用することができます。此処でお見せしたパターンはモジュールでよく見られるパターンです。将来に備えて、いまからこのパターンに慣れておこうと言う訳です。

ファイルの 2 行目-*- *coding:utf-8* -*-はこのプログラムファイルが UTF-8 エンコーディングをつかって書かれていることを宣言しています。特に、日本語などの on-Ascii な文字列を取り扱う場合には、この宣言を行っておくことがエンコーディングの違いによる問題を避けるためには望まれます。

[*9] でもどうやって？　いい質問です。後の開発ツールをご覧ください}。

次に、**def** 文を使って *main()* 関数を定義します。*:*以降のインデントされたブロックの中に *main()* 関数の本体、つまり *main()* 関数が呼出されたときに実行される Python の文が記述されています。

次の文 *if __name__ ==*「*__main__*」*:*はモジュールファイルの典型的な一文で、このモジュールファイルが主プログラムとして実行された場合には、次のインデントされたブロック内の文を実行することを指示しています。インデントされたブロック内には *main()* 関数の呼び出しが指示されていますので、main 関数の本体が実行され端末に」Hello, World.」が表示されることになります。

2.2 プログラムの構成要素

Python ではその ‘‘C‘‘ やその他のプログラミング言語と同様に関数を定義し、それらを組み合わせることでプログラムが構成されます。名前は関数ですが、値を返さない関数もあります[*10]。

Python ではプログラムを構成する際の要素として、クラスを用いることも出来ます。クラスを使ったオブジェクト指向プログラミングについては後ほど詳しく説明します。

よく使う関数やクラスの定義を別ファイルに保存しておいて、複数のプログラムでこれらの定義を使う事もよく行われます。これらの定義ファイルをモジュールと呼びます。モジュールの使い方についても、後ほど説明します。

2.2.1 関数の定義

まず、簡単な関数の定義をご覧下さい。:

```
def add(x,y):
  z=x+y
  return z
```

add(*x*, *y*)

def add(x,y): z=x+y return z

ここでは二つの引数 `x,y` の和を戻り値とする関数 `add(x,y)` が定義されています。

`def` は関数定義のための Python 言語のキーワードです。`def` の後に定義する関数名と、引数の並びを記述します。

C 言語においては、 複数の文からなるブロックを *{* および *}* で囲むことで、プログラムのブロック構造を記述しました。:

```
double add(double x, double y){
 double z;
 z=x+y;
 return z;
}
```

[*10] 値を返すものを関数、値を返さないものを手続きと呼び分けることも可能ですが、0 も数の内とおもえば、関数と呼ぶことに不思議はありません。

Python では同様のブロック構造を指定するのに：と行のインデントを使います。つまり、最初のプログラムにおいて:

```
z=x+y
return z
```

が一つのブロック構造になっています。ブロック構造の終了は行のインデントがなくなることで示されます。

同一のブロック構造中の文は emph{同じインデント}を持っていなければなりません。具体的には、同じ数の空白文字が含まれます。インデントにタブを使うことも出来ますが、おすすめできません。

2.2.1.1 開発ツール

このような Python の独特なブロック構造の指定法に従ったプログラム開発には、emacs(+python-mod.el), Python IDE(idle), eclipse(+PyDev) , PyCharm など Python 言語の文法に対応したエディタを使うことが望ましいでしょう。

emacs+python-mode.el では tab を指定した数の空白に置き換えてくれますし、DEL キーを使うことでインデントレベルを減らすことも簡単です。またこれらのツールを使うと、予約語の文字色をかえたり、括弧のバランスをチェックしてくれたりと役立つ機能を提供してくれます。

IPython は端末上の実行環境で、syntax に基づいたヒントを表示してくれるなどの機能を持ちます。IPytohn-Notebook から発展した jupyter は Web をインタフェースにした Notebook 型の開発環境を提供します。Notebook は数式処理プログラムとよく知られる Mathematica で使われるような、ドキュメントとプログラムコードが混在したファイルとなっています。Notebook の中に、プログラムコードとその結果（Graphical な出力を含む）を埋め込んで保存することができます。

申し訳ないことに、この文書の著者は emacs を開発環境として使っており、これらの最新の開発環境にはいささか疎くなっています。近くでこれらのツールをお使いの方を見つけて、質問してみましょう。

jupyter は複数の言語を Notebook 形式でサポートすることができます。この機能は拡張可能で、各言語向けの IPython Kernel をインストールすることで拡張が実現できます。詳細は jupyter のドキュメントをご覧ください。

2.2.2 関数の実行

定義した関数は、:

```
ans=add(2,3)
print "sum of 100 and 200 is ",add(100,200)
add(400,500)
```

といったように Python プログラム中で式が許される場所で自由に呼び出せます。最後の形式では、戻り値はもしあったとしても無視されます。（対話モードでの実行では画面に印刷されます。）引数がない関数を定義／実行するには、

```
def hello():
    print "Hello! Welcome to the Python world."

hello()
```

の様に空の引数リストをつけて呼び出します。

2.2.3 引数の既定値の指定。

関数定義では引数に省略時の既定値を関数定義時に与えることが出来ます。

```
def add1(x,y=1):
  return (x+y)
```

は第二引数 y が既定値 1 を持つ関数 add1 を定義しています。この関数の実行には幾つかの形が許されます。

```
print add1(2)
print add1(2,3)
print add1(x=4)
print add1(y=5,x=6)
```

最初の行の実行では add1 の第 2 引数が省略されていますので、既定値 y=1 が使われます。第 2 行の形式では、二つの実引数が関数定義の仮引数のリストの順番に割当られ、add1 の実行では x=2,y=3 として評価されます。第 3 の形式の様に既定値のない仮引数にも関数実行時に名前を指定して値を渡すことが出来ます。第 4 の形式では、仮引数 x、y の名前を指定して値を割り当てています。この場合には、実引数の現れる順序は関数定義時の順序と無関係です。このプログラムの実行結果は、

```
>>> print add1(2)
3
>>> print add1(2,3)
5
>>> print add1(x=4)
5
>>> print add1(y=5,x=6)
11
```

となります。

既定値を持つ関数定義時に既定値を持つ仮引数は既定値をもたない仮引数より前にあっては行けません。つまり、

```
def badd(x,y=1,z):  # これはエラーになる。
   return (x+y+z)
```

のようなプログラムは許されません。対話的実行時にこれを入力すると:

```
>>> def badd(x,y=1,z):
...   return (x+y+z)
```

```
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

と文法エラー (SyntaxError) が表示されます。

この他に Python では不定個の引数を持つ関数などを定義することも出来ますが、これについては回を改めて説明いたします。

2.3 文と制御構造

計算機プログラムが力を発揮するためには、動作の繰り返しや、条件判断がプログラム中に記述できることが重要となります。

これらの繰り返しや条件判断の仕組みはプログラムの**制御構造**と呼ばれます。前節で説明した関数や言葉だけがでてきたクラスも**制御構造**の一部と言う事が出来ます。この節では、最も基本的な順次実行、繰り返し、条件判断の制御構造について説明します。

2.3.1 Python プログラムの行

Python などのプログラミング言語では、プログラムは**文**が一列に並べられたものと捉えられます。一列にならんだ**文**は他の制御構造が無ければ、最初から順番に実行されていきます。これを**順次実行**と呼ぶ事もあります。

例えば、先ほど示したプログラムの実行例:

```
>>> print add1(2)
3
>>> print add1(2,3)
5
>>> print add1(x=4)
5
>>> print add1(y=5,x=6)
11
```

では、`print add1(2)` が最初に実行され、その結果として `3` が端末に表示 (`print`) され、次に `print add1(2,3)` が実行され、端末には `5` が表示されると言う具合です。

文はプログラム処理系にとって意味のプログラムの一纏まりと言う事が出来ます。これに対して、**行**はエディタなどのプログラムファイル作成アプリケーションがデータを取り扱う際の一つの纏まりです。通常は行の始めから改行記号までが一つの行として取り扱われます。この**行**はエディタなどの画面に表示されているプログラムの一行 (物理的な行) とは一致していないかも知れません。行頭から改行記号までの文字列が長ければ、エディタプログラムは画面上でそれを折り返して表示することがあるからです。[*11]

[*11] 論理的な一行が物理的な一行を超えると、プログラムの構成が読みにくくなってしまいますので、避ける方が良いでしょう。プログラムの**文**も適切な長さに押さえる事が望まれます。

C 言語では単文の文末は文字「;」で指示されます。

```
main(){
 printf("Hello World\n");
}
```

それに対して、Python プログラムでは原則的に行の終わりが一つ文の終わりとなります。[*12]

```
def main():
  print "Hello World"

if __name__ == "__main__":
  main()
```

一行に収まらない場合には行の終わりを\とすることで、文を継続することが出来ます。:

```
def main():
  print \
    "Hello World"

if __name__ == "__main__":
   main()
```

また、括弧 「()」の中の式を複数行にまたがって書く事も可能です。:

```
x=(1+2+
   3)
print x
```

物理的に長い行はプログラムの読み易さを損ないます。一つの文は一行に収まるように考える事が望まれます。[*13]

2.3.2 制御ブロック

Python の大きな特徴は制御ブロックの指定方法です。

C 言語等では、

```
if(cond){
  action_for_true();
  return 1;
}
else{
  action_for_false();
  return 0;
}
```

[*12] 行 と 文 の使い分けは計算機プログラム言語以外ではあまり気にして居ないかもしれない。意味的に (プログラム慣れした人は論理的に等というかも知れない) 一つながりが 文 , 見た目の (物理的な) 一塊が 行 であろうか。

[*13] Python 禅では、読み易さには意味がある と考えます。

と言った様に、{}を用いて、制御ブロック（複文）を表します。これに対して、Python ではこれと同等の文は、:

```
if (cond):
  action_for_true()
  return 1
else:
  action_for_false()
  return 0
```

となります。つまり、制御ブロックの開始が行末のコロン」:」で指定され、ブロック内の行は同じ長さのインデント (空白文字の並び）を持ちます。ブロックの終了はインデントの長さ（深さ）が短く（浅く）なったことで検知されます。とくに、最外周の制御ブロックは長さ（深さ）０のインデントを持つこと、言い換えれば行の始めの文字は空白文字以外の文字でなければならないことに注意してください。

これは対話的に Python プログラムを実行する際に、特に注意しなければならないでしょう。

すなわち、インデントがない以下の行は、正常に実行されますが、

```
>>> print "Hello World"
Hello World
```

行頭に空白文字をつい追加してしまった、以下の入力はエラーを引き起こしています。

```
>>>  print "Hello World"
  File "<stdin>", line 1
    print "Hello World"
    ^
IndentationError: unexpected indent
>>>
```

最初このルールは奇妙なものに思え、エラーを引き起こすことも多いでしょう。しかしながら、C 言語のプログラムと Python プログラムを比べてみると C 言語でも（言語の文法としては不要な) インデントを導入することが、プログラムの視認性を高めるために推奨されているため、大きな違いは無いとも言えます。むしろ Python では言語仕様が人間がプログラムを読む際の視認性を高める事を**強制**しているとも言えます。Python でのプログラムを書き進める事で、すぐにこのルールに抵抗はなくなり、このルールによって Python では簡潔で、視認性のよいプログラムを書き易いことがわかるでしょう。

emacs の python_mode.el や Python の配布物に含まれる IDE, eclipse 環境の python plugin 等をつかうことで、このルールに従った Python プログラムを書くことはより簡単になるでしょう。

注釈: Python を対話的に利用している時には、`help()` 関数が利用出来ます。help の引数はモジュール、関数、データ型、データなどのオブジェクトで、オブジェクトに応じたヘルプメッセージが表示されます。 開発環境に依っては、プログラムファイル作成ウィンドウでこれらの情報をみることもできます。

2.4 制御構造

2.4.1 構造化プログラミング

Python は近代的なプログラム言語のひとつとして、構造化プログラムのための制御構造を持っています。構造化プログラムでは、一連の処理の固まりである**ブロック構造**と、条件によりそれらのブロックを選択する**分岐**、またブロック構造をある条件のもとで繰り返す**ループ（繰り返し)** が基本となります。

2.4.1.1 ブロック構造

先日のメールに書きました様に、Python ではブロック構造を行のインデントを使って表現します。ブロック構造をネストする（入れ子構造にする）こともちろん可能で、この場合にはインデントの深さがネストの深さに対応します。

```
a=3
while a>0:
  if a > 2:
    a=a/2
  else:
    a=3*a+1
    print a
```

2.4.1.2 条件判断 (分岐)

条件判断には if 文が用意されています。<条件式>が成立する場合に statements を実行するプログラムはつぎの様になります。

```
if <条件式>:
  statements
```

条件式は C 言語と同様の算術・比較演算子や論理演算子（and　および　or , not) などを使って記述します。

論理演算子は C 言語の論理演算子 (&&や ||等) は「使えない」ことに気をつけてください。[*14]

if 文と対応する else 節ももちろん用意されています。

```
if <条件式>:
  <statements>
else:
  <other statements>
```

Python には変数の値によって実行するべき文を選択する C 言語の Switch 文に相当する制御構造は文としては**存在しません**。幾つかの条件から成立する条件を選んで、実行する文を選択するためには、elif 文が用意されています。

[*14] その他の算術/比較演算子は C 言語と共通なので、ついつい Python で and と書くべきところで&&と書いてしまいがちです。もっとも Python でのプログラミングをしばらく続けた後 C 言語に移ると、ついつい&&のつもりで and と書いてしまうのですが。

```
if <cond1>:
  <statements1>
elif <cond2>:
  <statements2>
elif <cond3>:
  <statements3>
....
else:
  <staementsN>
```

2.4.2 繰り返し (ループ)

Python は繰り返しの構造を記述するための文として for 文と while 文を提供します。

2.4.2.1 while 文, break, continue

while 文は次の形式を持ちます。

```
while <条件式>:
  <実行文の並び>
```

つまり条件<cond>が成立する間、<実行文の並び>を逐次実行します。

while 文からの脱出に break 文を使うことも出来ます。また、<実行文の並び>の途中でループの頭に戻るための continue 文も用意されています。

```
while b >a:
  q=b/a
  r=b-q*a
  if r == 0:
    break
  b=a
  a=r
<next statements after while>
```

2.4.2.2 for 文

以上の制御構造があれば、原理的にはどんな構造化プログラムの構造でも表現できるはずですが、よく使われる繰り返しを表現するために Python にも `for` 文が用意されています。

Python の `for` 文は

```
for <ターゲット> in <式> :
  <実行文の並び>
```

の形式を持ちます。たとえば、1 から 10 までの整数の和を求める for 文は

```
sum=0
for x in (1,2,3,4,5,6,7,8,9,10):
  sum=sum+x

print sum
```

となります。(1,2,3,4,5,6,7,8,9,10) は python の tuple と呼ばれるデータ型です（これについては次回説明します。）

あるいは、range() 関数を使って、

```
sum=0
for x in range(1,11):
  sum=sum+x
print sum
```

と書くことも出来ます。Python の for 文は else 節を持つことができますので、上記のプログラムは、

```
sum=0
for x in range(1,11):
 sum=sum+x
else:
  print sum
```

と等価です。(print sum のインデントの違いに注意。)

2.4.3 Python キーワード

さて、プログラムの構造を作るために if,for,while などのキーワードを使ってきました。この他にも、Python ではいくつかのキーワードが用意されています。これらのキーワードは Python で予約されており、変数名、クラス名、関数名などの idenfifier として使うことはできません。

Python では以下の語がキーワードとして予約されています。

:: and del from not while as elif global or with assert else if pass yield break except import print class exec in raise continue finally is return def for lambda try

ここに挙げた 2 7 個のキーワードの内、すでに 13 個のキーワードについては既にこれまでの説明で使われています。

2.5 データ型、式

ython は整数、浮動小数点、文字列といったその他のプログラミング言語でもおなじみのデータ型の他に、辞書型、リスト型、タプル（tuple) 型といったデータ型を用意しています。数値計算のパッケージではアレイ（配列）型データも利用可能です。

2.5.1 数値データ

Python では数値データ型として整数型（符号付き 32 ビット、-2147483648　～2147483647)、浮動小数点型、複素数型が用意されています。また整数型には無限精度（と言ってももちろんメモりの制約で無限と言う訳には行かないが、Python のシステムとしては最大値の制限がない）の長整数型も用意されています。整数の計算式中で途中結果が整数型の範囲を超えると自動的に長整数型のデータに変換されます。

長整数型の定数データを入力するには入力する整数の最後に」L」を付けます。また、複素数型データでは虚部を文字」j」をつけて指定します。型変換を使って明示的にしていすることも可能である。(純虚数を定数として入力する場合には、」1j」とする必要です。」j」だけでは名前 「j」と解釈されます。）

```
18889465931478580854784L #長整数型の定数
2+3j #複素数型定数

long(18889465931478580854784) #型変換による明示的な指定
complex(2,3)
```

行中の文字」#」はコメントの開始を示しています。この文字から行の最後 (\n）まではコメントとして取り扱われ、Python プログラムの実行には影響しません。

2.5.2 式

数値計算の式はほぼ C 言語と同様です。

```
1+2-3*4/5.**(6^7)
```

に項演算子, +-*/ **(べき乗) \%(剰余）<< >> (シフト）&| ^ (ビット毎の論理演算）などは C 言語と共通です。その他に//(floor division) 等の演算子もあります。

式の中で、sqrt, sine, cosine などの数値関数を用いるには、

```
import math
math.sin(math.pi/3)
```

などとします。import 文とモジュールについては後日詳しく説明します。ここでは、数値関数を使う場合には、」import math」が必要ということと、関数名の前にも」math.」が必要と言うことに注意しておいて下さい。

Python には数値型や文字列型のデータ型の他に、リスト、タプル、辞書型と言った非常に協力なデータ型が提供されています。これらのデータ型を使いこなすことで、Python プログラムがより楽しくなるでしょう。Python ではさらにクラスを定義することで、これらのデータ型の拡張も可能となっています。

2.5.3 リストとタプル

Python には複数のデータの並びを取り扱うためのデータ構造としてリストおよびタプル (tuple) が用意されています。C 言語や Fortran の一次元配列と同じ様に複数のデータが並んでおり、それらの要素は番号を指定す

ることで取り出すことが出来ます。Python の言い方で言えば、list と tuple はいずれも sequence 型のデータと呼ばれます。

```
l=[3, 2, 7, 6, 5, 6, 3, 2, 6, 9]
for i in range(10):
  print l[i], l[i]**2
```

は変数 l に長さ 10 のリスト [3, 2, 7, 6, 5, 6, 3, 2, 6, 9] を割当、それぞれの数とその自乗を印刷します。

ところで、上記のプログラムは、

```
l=[3, 2, 7, 6, 5, 6, 3, 2, 6, 9]
for i in l:
  print i, i**2
```

と書くことも出来ます。range(10) は 0 から 10 までの整数を表すリスト (sequence) を返す関数です。また:

```
for i in range(10):
```

はこのシークエンスから順番に要素を取り出して、i にその値を割当、ブロック内の文を実行することになっています。最初のプログラムではこの添字を使ってリストの要素を取り出していた訳ですが、後者のプログラムでは直接リスト（シークエンス）の要素を for 文で取り出している訳です。

同じプログラムを tuple を使って書くと、

```
t=(3, 2, 7, 6, 5, 6, 3, 2, 6, 9)
for i in t:
  print i, i**2
```

となります。

上記の二つのプログラムを比べると、list ではデータの並びを [] で囲み、tuple では () で囲んでいるだけです。これではなぜ list と tuple と言う二つの異なるデータタイプが存在する理由がわからないと思います。

それではリスト (list) とタプル (tuple) の違いは何でしょうか？ Python の用語では List は mutable(変更可能), tuple は immutable(変更不能) なデータの並びと言われます。リストはデータを作成した後に要素を取り出す（sequence 型）の他に、要素を追加する、挿入する、並べ替える、要素を変更すると行った操作が可能です。たとえば、

```
l=[3, 2, 7, 6, 5, 6, 3, 2, 6, 9]
for i in range(10):
  if ( l[i]%2 == 0):
    l[i]=(l[i],l[i]**2)

print l
```

はリスト l を単純なリスト [3, 2, 7, 6, 5, 6, 3, 2, 6, 9] からタプルあるいは整数を要素とするリスト、

[3, (2, 4), 7, (6, 36), 5, (6, 36), 3, (2, 4), (6, 36), 9]

に変更します。一方 tuple を変更することは出来ません。

```
t=(3, 2, 7, 6, 5, 6, 3, 2, 6, 9)
for i in range(10):
  if ( t[i]%2 == 0):
    t[i]=(t[i],t[i]**2)

print t
```

はエラー (型エラー例外):

```
TypeError: 'tuple' object does not support item assignment
```

を発生します。タプルは後で述べる辞書型データのキー（キーは変更不能でなければならない）として使うことが出来ます。また、データ構造を変更不能 (immutable) とすることで、不用意なデータの変更から引き起こされるプログラムの不正動作を防ぎます。

リストとタプルはお互いに型変換関数 list() および tuple() によってそれぞれのデータ型のデータを作成することが出来ます。

2.5.4 リストおよびタプル（シークエンス型）データの添字

Python のリストおよびタプル（より正確には sequence 型のデータ) は一次元に番号付けられており、添字をしていすることで、その中の要素を取り出せます。Python では添字は 0 から始まります。

```
l=[1,2,3]
print l[0],l[1],l[2]
```

さらに負の添字も許されており、添字 i が負の場合には、リスト（シークエンス）の最後から-i 番目のデータを取り出します。

```
l=[1,2,3]
print l[-3],l[-2],l[-1]
```

組み込み関数 ‘‘len()‘‘を使うことで、シークエンス型の要素の数がわかります。

```
l=[1,2,3]
print "Size of list l is ",len(l)
```

2.5.5 スライス

シークエンス型の要素を一つ取り出すだけではなく、シークエンス型データの一部を同じデータ型のシークエンスとしてとり出せます。 この場合シークエンスの添字に代えてスライスを指定します。スライスの最も一般的な形では、取り出す最初の要素の添字、取り出す最後の要素の次の要素の添字，増分の三つを指定します。

```
l=range(10)
s=l[0:3:1]
```

これはまた、

```
l=range(10)
s=l[0:3]
```

あるいは

```
l=range(10)
s=l[:3]
```

と書くことも出来ます。添字と同様に負の数をスライスに指定することが出来ます。たとえば、

```
l=range(10)
s=l[-3:]
```

を実行すると s には リスト l の最後の三つの要素からなるリスト [7,8,9] が割当られます。

2.6 リスト操作

Python のデータは実はオブジェクトであって、それ自体が様々な固有のメソッドを持っています。mutable(変更可能）なリストデータはリストを変更イ操作）するためのメソッドを持っています。

2.6.1 リストの操作

リストは要素を添字あるいはスライスをしてして取り出すだけではなく、要素を追加、挿入, 拡張あるいは削除したり、要素の順序を整列したり反転することができます。

2.6.1.1 要素の追加

append メソッドを使います。

```
l=[1,2,3]
l.append(5)
print l
```

2.6.1.2 リストの拡張

extend メソッドを使います。

```
l=[1,2,3]
l.extend([2,3])
print l

l.append([4,5])
print l
```

extend メソッドはリストに引数のリストの要素を追加します。それにたいして、append にリストを指定した場合にはリストそのものが要素になります。ここで、Python のリストは (Tuple も）要素が同じ型のデータである必要はありません。この点はその他の言語の配列（通常同じ方のデータとなる）と大きく異なりますので、注意が必要です。

2.6.1.3 リスト要素の挿入

insert メソッドを使います。

```
l=[1,2,3]
l.insert(1,10)
print l
l.insert(0,20)
print l
l.insert(-1,30)
print l
```

insert は第一引数で与えた添字が示す場所の後に第二引数に指定された新要素を挿入します。

2.6.1.4 リスト要素の削除

remove メソッド remove は insert のほぼ逆向きのメソッドです。添字に指定した要素を削除します。

```
l=[1,2,3,4,5]

l.remove(3)
print l
```

del 文をつかうことも可能です。

```
l=[1,2,3,4,5]
del l[2]
```

2.6.1.5 リストの整列および反転

sort および　reverse

リストを整列あるいは反転するのに、sort および reverse のメソッドがそれぞれ用意されています。

```
l=[1,5,3,2,4]
l.sort()
print l
l.reverse()
print l
```

sort,reverse はいずれもメソッド呼び出しですが、値を返さないこと注意してください。これらの関数を呼出しますと、リストの内容そのものの順序が変わってしまいます。

組み込み関数 sorted() を使うと、リストそのものを変更すること無く、整列されたリストが返されます。

%reversed() もグローバル名として予約されていますが、こちらは関数ではなく type となっています。%reversed() を実行すると、逆順に要素を返す ‘‘iterator‘‘が返されます。

```
l=[1, 5, 3, 2, 4]
print sorted(l) # 整列されたリスト
print l #元のリストはそのまま
for e in reversed(l):#逆順にリストの内容が取り出される。
  print e
l.sort() #リストを整列。値は返さない。
print l #リストの中身が整列されている。
l.reverse() #リストの中身を逆順にする。
print l
l.sort(reverse=True) #リストの中身を逆順に整列する。
print l
for e in reversed(l):
  print e
```

2.7 データ構造：文字列型と辞書型

有名な Pascal 教科書の題名は」データ構造＋アルゴリズム=プログラム」ですが、Python のデータ構造は、数値型、sequence(list, tuple, string)、辞書、および class で定義された object ということになります。

またアルゴリズムは関数、class（のメソッド）としてカプセル化されます。import 文によって導入されるモジュールもアルゴリズム（とそれに付随するデータ構造）をパッケージ化したものということになります。

まずは基本データ構造の一つである文字列型と辞書型について説明します。

2.7.1 文字列型

Python でも文字列は基本データ型として取り扱われます。C 言語と異なるのは、文字列型はあるけれど文字型はないと言うことです一文字の文字も文字列となります。

文字列定数は引用符 (『) あるいは二重引用符 (「) で文字の並びで囲むことで表現します。

```
s="abc"
t='ABC'
```

Python は Unicode をサポートしていますので、日本語文字も取り扱うことは出来ます。日本語の出力にはプログラムだけでなく、出力端末の文字コードの設定などが関係します。機会があれば別途まとめたいと思います。ともあれ、unicode の定数は

```
u=u"abc"
```

と引用符の前に u を付けることで指定します。

Python の文字列型はリストやタプルと同じくシークエンス型データの一種となっています。ということは文

字列の一部は添字やスライスで取り出すことができると言うことになります。また、文字列をリストやタプルに変換することも可能です。

```
s="abcdefghijklmn"
print  s[::2]
for c in s:
print c
print list(s)
print tuple(s)
```

リストには’insert』,』remove’などのメッソッドが存在したように、文字列データにも固有のメソッドが存在します。

```
print dir("")
```

```
help(type(""))
```

を実行することで、それらのメソッドの名前や説明を端末に表示させてみましょう。type() 関数は引数のデータ型を値として返します。また、dir() は引数の属性をリストとして返します。

2.7.1.1 文字列の連結

文字列は + 演算子を使うことで、連結した文字列を作成できます。

```
print "abc"+"123"
```

2.7.1.2 書式指定

C 言語では printf, sprintf などで書式を与えることで、データを印刷する方法を指定しました。 Python では文字列に % 演算子を指定することで、同様の書式変換を行います。

```
import math
th=math.pi/3
s=math.sin(th)

print "sin(%f) = %f"%(th,s)
```

を実行しますと、

```
sin(1.047198) = 0.866025
```

が端末に表示されます。

書式指定は C 言語と同じく %d,%f,%g,%s などが使えます。桁数や小数点以下の精度指定も同様です。

書式変換の演算子 % のあとに続くデータがタプルの場合には、書式指定中の書式指定子とタプルの要素が順次対応づけられます。

2.8 データ型：辞書型

辞書型データはいわゆるハッシュド・リストを提供します。キーと値のペアの集まりと考えることができます。細かく言うとこのようなキーと値の関連づけをおこなうデータはマップ型と呼ばれる抽象データ型であって、辞書型 (dictionary) はこのマップ型データの一つの実装と言うことになります。Python の組み込みデータ型としては 辞書型データが唯一のマップ型データです。

2.8.1 辞書型データの生成

辞書型データを作成する一つの方法はリテラルに辞書型データを定義する方法です。

```
d={"a":1}
```

これはキー」a」に値 1 を対応させる辞書型データ d を定義します。複数のキー一値ペアを持つ辞書は単に、

```
d={"a":1,"b:2,"c":3}
```

とします。dict() 関数を使うとこの辞書は、

```
d=dict(a=1,b=2,c=3)
```

と書くことができます。この方法ではキーに引用符をつける必要がないことに注意してください。

2.8.1.1 辞書型データの key として許されるオブジェクト

辞書型データで定義される (key, value) のペアで、値 (value) はどのような Python データ型が許されますが、key となる data には制限があります。key は Python でいうところの immutable なオブジェクトだけが許されます。関係を定義した後に key が変わってしまうのでは、後の高速な検索が不可能のなってしまうからだと思われます。immutable なオブジェクトとは事後に変更することが不可能なオブジェクトです。具体的には、数値、文字列データ、タプルなどがそれです。

リストや辞書型データは mutable すなわち変更可能オブジェクトであるため、辞書型データの key となれないことに注意してください。

2.8.2 辞書型データの利用

辞書型データの持つ key に対応する値を取り出すには、

```
d=dict(a=1,b=2,c=3)
print d["a"]
```

と key を」[]」で囲んで指定します。文字列が key の場合、引用符が必要なことに注意してください。

```
d={"a":1,"b":2,"c":3}
k="a"
print d[k]
```

としても同様です。

2.8.3 オブジェクトとしての辞書型データ

すべての Python のデータ型と同じく、辞書型データもオブジェクトとなっています。辞書型データはオブジェクトとしていくつかのメソッドが定義されています。それらのメソッドのうち辞書型データに特有ないくつかのメソッドを説明します。

2.8.3.1 items/values/keys

辞書型データは Key と値 (value) の対応表を持っています。items メソッドはこの Key とその値からなる長さ２の tuple から構成されるリストを値として返します。

```
>>> d=dict(a=1,b=2,c=3)
>>> d.items()
[('a', 1), ('c', 3), ('b', 2)]
```

このメソッドを使って、別の辞書型データを更新することも可能です。

```
>>> e=dict(x=0,y=-1,z=1)
>>> e.update(d.items())
>>> e
{'a': 1, 'c': 3, 'b': 2, 'y': -1, 'x': 0, 'z': 1}
```

辞書データの Key のリストあるいは値のリストは、keys および values メソッドで取り出すことが可能です。

```
>>> e.keys()
['a', 'c', 'b', 'y', 'x', 'z']
>>> e.values()
[1, 3, 2, -1, 0, 1]
```

2.8.3.2 has_key メソッド

has_key() メソッドは辞書型データが引数を key としてもっていれば True さもなければ False を返します。[*15]

```
d={}
for word in line:
   if d.has_key(word):
      d[word] +=1
   else:
      d[word]=1
```

[*15] そういえば、True と False は論理型の真と偽を表すためにあらかじめ定義されているということはお話しましたっけ？ Python にはこの他にも None と言う定数が定義されています。奇妙なことに True と False は定数ではありません。でも混乱をさけるために,True と False を再定義することは避けた方が無難でしょう。

```
for k in d:
   print k, d[k]
```

これは単語のリスト line から単語の出現頻度をカウントするプログラムになっています。

2.8.3.3 辞書型データの get() メソッド

```
for key in myList:
   print "value for %s is "%key, d.get(key,"N/A")
```

辞書型データ d の get メソッドは引数 key を辞書型データ d が持つ場合には d[k] を返しますが、key が d のキーで無い場合には、 2 番目の引数を既定値として返します。 2 番目の引数を省略した場合、None が指定されたものとして取り扱います。

2.8.4 辞書型データと繰り返し

items/keys/valuse メソッドには、それぞれに対応した,iteritems/iterkeys/itervalues メソッドが存在します。これらはリストではなく for-loop での使用に適したイテレータを返します。

```
for k, v in dictionary.iteritems():
    print k,v
```

2.8.5 文字列 Format と辞書型データ

パイソンでは C 言語の sprintf と同様の書式変換を文字列データに対する % 演算子で指定します。

```
x=1
y=2
print "%f + %f makes %f"%(x,y,x+y)
```

この場合、% 演算子にはタプルが与えられ、書式指定の書式指定子 (%f など）は現れる順序でタプルの要素の値と関連づけられて行きます。

文字列の % 演算子にタプルに代えて辞書型データを引き渡すことも可能です。この場合には、次の例に見る様に辞書型データの key で書式指定子に対応させるデータを直接指定することが可能です。

```
x=1;y=2
"%(a)f plus %(b)f makes %(ans)f\n"%dict(ans=x+y,a=x,b=y)
```

このプログラムの実行結果は、

```
>>> '1.000000 plus 2.000000 makes 3.000000\n'
```

となります。

2.8.5.1 プログラム実行環境としての辞書型データ

Python プログラム中で定義する変数名、関数名、クラス名などの名前は global および local という二つの名前空間でコントロールされています。プログラム実行中に利用できる変数名などの名前はこの二つの名前空間のいずれかに登録されていることが必要です。

global および local に登録されている名前を調べるために、二つの組み込み関数, globals() と locals(), が用意されています。globals() と locals() は実行されたコンテキストでの global 名前空間あるいは local 名前空間のそれぞれの名前とその値からなる辞書型データを返します。

```
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'y': 2,
 '__doc__': None, 'x': 1}
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'y': 2,
 '__doc__': None, 'x': 1}
```

組み込みの関数 eval() を使うと Python の式 (expression) を、global 環境と local 環境を指定して評価することが出来ます。

Python の変数名、関数名などの名前 (identifier) はこの様に辞書型データを使って管理されていると考えることが出来ます。(実装上もそうなっているかどうかは確かめていません)。つまり、名前とその値（となるオブジェクト）の対応表でそれらは結び付けられているだけです. C 言語等のコンパイラ言語では変数名はそのデータ型の記憶領域に付けられた名前であって、その関係を動的に変更することは出来ません。一方 Python では変数名はオブジェクトを検索するためのキーであって、その割当はプログラム実行時に動的に変更することが可能です。C 言語であえてその対応物を考えると、汎用のポインタ変数ということになります。どんなオブジェクト (のポインタ）でもその変数に割り当てることが出来ます。

2.9 内包表現とジェネレータ式/map, apply, eval

Python にはごく初期から、‘ map, apply, lambda, filter, eval,execfile‘ といった Lisp ではおなじみの関数が用意されていた。Python の創始者である Guido さんはこれらの Lisp 風関数はお気に召さなかったらしい（実のところこの小文の筆者はこれらの関数を好んで使っていたのであるが)。Python2.x では内包表現およびジェネレータ式が導入され, これらの Lisp 風関数の多くはこれらの新しい表現で置き換え可能となった。eval および execfile についてはまだ置き換えれらるものはない。

2.9.1 内包表現 (map, filter)

Python では、配列などのシークエンス型データの各要素について演算を施した結果を再びシークエンスとする場合などに内包表現 ((list comprehension) を用います。例えば、0 以上 2N 未満の偶数からなるリストは、

```
[2*i for i in range(N)]
```

で生成されます。　あるいは、

```
[i for i in range(2*N)  if (i%2 ==0)]
```

とすることもできます。これらが内包表現の例です。内包表現が導入される以前は、

```
quad=map(lambda x:x**4,range(100))
```

などとしていたましたが、内包表現をつかうことで、シンプルに

```
quad=[x**4 for x in range(100)]
```

と書ける様になりました。

2.9.2 generator 式

generator 式は内包表現と似ています。まずは generator 式の例をお見せします。in の後の式が generator 式です。

```
sum=0
for x in ( 2*i for i in range(100)):
   sum=sum+x
```

目につく違いは内包表現はリストを生成するので、外側に鍵括弧 (『[『と』]』) を持つが、generatro 式は丸括弧 (『(『および』)』) で囲まれていることです。それでは generator 式は Tuple を生成するのかというと、そういう訳ではなく、generator 式は generator オブジェクトを生成します。generator オブジェクトは iterable つまり next() メソッドを持ち、for 文の範囲指定等に洗われることができます。

内包表現はリストを作成し、List は iterrable ですから、generator 式の代わりに内包表現を使うことも原則として可能です。(その逆は真ならずなので、注意が必要)

それではなぜ generator 式が存在するのか？

内包表現は必ずリストを作成しますので、呼び出し時にリストが作成されます。非常に大量の要素について、繰り返しを実行する際にはこのことはメモリの使用量においても、実行時間の観点からも効率的とは言えません。一方、generator 式は iterrator オブジェクトを生成し、要素となるオブジェクトは next() メソッドが呼ばれる毎に生成されます。これにより長大なリストを生成する必要がなくなり、メモリの必要量もそれに応じて少なくてすみます。また、iteration のサイズが不定の場合にも generator 式を使う事が出来ます。

generator 式は (return の代わりに) yield 文をもつ generator 関数の特別な形とも言えます。

2.9.3 内包表現および generator 式中での条件 (filter の置き換え)

内包表現と generator 式に if で条件をつけることができます。例えば、

```
even=[i for i in range(100) if i % 2 == 0]

for e in ([i for i in range(100) if i % 2 == 0):e
    ... do something for e ...
```

など。filter 関数を使うとこれは、

```
even=filter( lambda x: x%2 == 0, range(100))
```

などとなります。内包表現および generator 式を使うことで、filter() 関数および lambda 式の双方が不要となりました。[*16]

2.9.4 exec 文, eval, execfile

2.9.4.1 exec 文

exec 文は動的な python プログラムの実行の為に用意されています。exec 文の一般的な形式は、

```
"exec" expression ["in" expression ["," expression]]
```

となっています。

最初の expression は文字列、ファイルオブジェクト、あるいはコードオブジェクトで、exec はこれらを Python プログラムとして実行します。」in」の後の一つあるいは二つの dictionary は評価の際に gobal および local の環境として使われます。

```
exec "print `Hello`"
```

2.9.4.2 eval

Lisp においては apply,eval は map 関数と同じくよく使われる関数です。Python における eval 関数は Python の式を表す文字列あるいは compile() 関数が返すコードオブジェクトを引数とします。eval は与えられた文字列あるいはコードオブジェクトを式として評価し、その値を返します。

```
eval(...)
    eval(source[, globals[, locals]]) -> value
```

eval では exec 文とは異なり、文字列に与えることのできるのは式だけです（文は不可）。

```
import sys
eval("sys.stdout.write('Hello\\n')")
```

2.9.4.3 execfile

execfile 関数は eval の file 版です。filename から Python プログラムを読み込みそれを評価します。exec 文と同様に globals と locals の辞書をオプションとして与えることが可能です。

[*16] そういえば、True と False は論理型の真と偽を表すためにあらかじめ定義されているということはお話しましたっけ？ Python にはこの他にも None と言う定数が定義されています。奇妙なことに True と False は定数ではありません。でも混乱をさけるために,True と False を再定義することは避けた方が無難でしょう。

```
execfile(filename[, globals[, locals]])
```

import 文ではファイル名はプログラム中に固定されていますが、execfile あるいは exec 文をつかうことで、読み込むファイルの名前を動的に、つまりプログラムの実行中に、変更することが可能となります。

2.9.5 apply

apply は動的なプログラムを作成するときに活躍する。callable なオブジェクト（関数オブジェクト, __call__ メソッドを持つクラスのインスタンス等）とパラメータからなる tuple, 関数評価の環境となるオプションの辞書型データ kwargs を引数としてとる。

```
apply(...)
    apply(object[, args[, kwargs]]) -> value
```

結果は kwargs(もしあれば）を環境とし、args の各要素を引数として object を評価したものとなる。Python 2.3 以降では apply は obsolete となり、同等の表現は、

```
object(*args,**kwargs)
```

となった。(Lisp を少しかじった私としては, apply を使った表現の方が意味がはっきりしていて好ましいと感じるのであるが）

2.9.6 reduce

reduce() 関数は関数 func とシークエンス (Tuple, list など）を引数とする。

```
reduce(...)
    reduce(function, sequence[, initial]) -> value
```

引数の function は二つの引数をとるもので、まず reduce は initial がある場合には、function(initial, sequence[0]) を、initial が省略された場合には、function(sequence[0],sequence[1]) を計算する。以下順次 function の結果と sequence の次の要素に function を繰り返し適用する。例えば、

```
reduce(lambda x,y:x+y,(1,2,3,4,5))
15
```

は 1+2+3+4+5 を計算して、その結果 15 を値として返す。reduce の仕組みとしては、((((1+2)+3)+4)+5) の方が適当かも知れない。

```
reduce(lambda x,y:(x,y), (1,2,3,4,5))
((((1, 2), 3), 4), 5)
```

2.9.6.1 zip

Mathematica/SAD script の Thread 関数に似ている。あるいは行列の転置と同様の効果をもつ。zip は一つ以上の sequence を引数として持つ。その戻り値は tuple のリストであって、リストの i-番目の成分は zip の引数に与えられた sequence の i 番目の成分からなる tuple となる。

```
zip(...)
    zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]
```

```
print zip((1,2,3),(2,4,6),(5,15,20))
[(1, 2, 5), (2, 4, 15), (3, 6, 20)]
```

zip は None を最初の引数とする map 関数とほぼ同等である。

```
print map(None, [1,2,3],[2,4,6],[5,15,20])
[(1, 2, 5), (2, 4, 15), (3, 6, 20)]
```

ただし、引数とする sequence が一つだけの場合には結果が異なるので注意が必要である。

```
>>print map(None, [1,2,3])
[1, 2, 3]
>>print zip( [1,2,3])
[(1,), (2,), (3,)]
```

逆に map 関数を利用していたような表現は zip と内包表現で記述できる。

```
>>> def func(i,c):
...     return dict( ((c,i),) )
...
>>> print map(func,(0,1,2),("a","b","c"))
[{'a': 0}, {'b': 1}, {'c': 2}]
>>> print [func(i,c) for i,c in zip((0,1,2),("a","b","c"))]
[{'a': 0}, {'b': 1}, {'c': 2}]
```

zip 関数と関連して enumeratet type についても述べておく。あるシークエンス l があったとき、

```
e=enumberate(l)
```

で生成される enumerate 型のデータ e は method next が呼ばれるたびに、

```
(0,l[0]), (1,l[1]), (2,l[2]),...
```

を次々と返す。l[i] が存在しないときには StopIteration の例外を発生する。

```
for p in enumerate(l):
  print "%d-th elment of l is %s"%p
```

2.10 モジュール (module)

Python を利用する利点の一つに豊富な Python プログラムのライブラリがあります。Python とともに通常配布される標準的なライブラリや、問題に応じて開発／提供されているライブラリがあります。Python ではこれらのライブラリを**モジュール:module**と呼びます。

2.10.1 標準モジュール

Python の配布では通常 (組み込み用などの特殊な場合を除いて) 数多くの標準モジュールが付属しています。これらのモジュールは Python を立ち上げてそれらのモジュールをインポートすることで、すぐに使い始めることが出来ます。例えば、Python を使用しているプラットフォームについての情報を os モジュール中の os.uname() 関数を使って入手することが出来ます。:

```
import os
print os.uname()
```

ここでは import os によって、os モジュールを Python にインポート (ロードと言った方がピンとくる方もいらっしゃるでしょう) しています。次の print 文で os モジュール中の uname() 関数を実行しその結果を端末に表示しています。実行結果を次に示します。:

```
% python
python
Python 2.7.6 (v2.7.6:3a1db0d2747e, Nov 10 2013, 00:42:54)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
import os
>>> print os.uname()
print os.uname()
('Darwin', 'MacBook-Pro.local', '13.0.0', 'Darwin Kernel Version 13.0.0: Thu Sep␣
↪19 22:22:27 PDT 2013; root:xnu-2422.1.72~6/RELEASE_X86_64', 'x86_64')
>>>
```

import 文によってインポートされたモジュール内のオブジェクト (サブモジュール、関数、クラス、データなど) を取り出すためには、モジュールの名前とそのオブジェクトを指定する名前を合わせて (os.uname の様に) 指定します。このオブジェクト自体がモジュールオブジェクトである場合も可能です。その場合には、このサブモジュール内のオブジェクトは、親のモジュール名、サブモジュール名、オブジェクト名を合わせて指定する事で、指定する事が出来ます。

深い階層のモジュールの中のオブジェクトを何度も呼出す場合には、from import ``\ 文や\ ``import as 文を使う事が出来ます。例えば::

import os print os.uname()

from os import uname print uname()

from os import os.uname as un print un()

はいずれも同じ結果を与えます。:

```
>>> from os import uname as un
>>> print un()
('Darwin', 'MacBook-Pro.local', '13.0.0', 'Darwin Kernel Version 13.0.0: Thu Sep␣
↪19 22:22:27 PDT 2013; root:xnu-2422.1.72~6/RELEASE_X86_64', 'x86_64')
```

from import の特別な場合として from import *があります。*はワイルドカードであって、指定されたモジュール内の名前空間中の名前を全てメインの名前空間に持ってくる事を指定います。

第 3 章

Python プログラム例

ここでちょっとおもむきをかえて、実戦的（？）な例をとって python でプログラムを作成する過程を逐次追ってみることにする。

3.1 プログラム例 1

ファイルにある数値等のデータをプログラムで処理する例です。

3.1.1 問題の提示

ここで考えるのは、:

```
http://tide.gsi.go.jp/furnish.html
```

で提供されている潮位のデータを Python を使って解析しようということです。まずは提供されるファイルのデータを Python で読み込み、適当な形に加工して（例えば、統計をとる、変動周波数を調べる、グラフ表示する）みようとおもいます。発展問題として、上記 URL から指定されたデータをダウンロードするプログラムを作成することも考えられます。

このサイトから、全部で 25 ある国土地理院管理の験潮所の潮位データをダウンロードすることができます。データファイルはアスキーフォーマットで、次に示すようにヘッダ部分とそれに続くデータ部分からなっています。また、このファイルの EOL(End of Line) は DOS 形式 (0x0d0x0a つまり』\r\n』) になっています。

```
#Record of Sealevel(mm)
#Location No 11
#Location Katsuura
#TimeZone +09:00(JST)
#Broken or No transmit data "-999"
#Datum constant 5000 mm
#Height of the fixed point  2002 mm
#date       time      mm
2008/07/08 04:00:00  2959
2008/07/08 04:00:30  2963
```

3.1.2 ファイルの読み込み

とりあえず、このファイルを Python プログラムで開いて読んでみます。

```
fn="11_20080708-20080713_sec30_n.txt"
f=file(fn,"rU") #f=open(fn,"rU") でも同じ。
print f.next()

#. \ ``fn``\ に文字列としてファイル名を割当。
#. ファイルを読み出しモード (`r`) でオープンします。\ `U`\ はファイルの行末を"Universal newline
↪support"を使って処理する事を指定しています。
#. ファイルから一行目を取り出して、端末に表示します。
```

実行結果は、

```
>>> fn="11_20080708-20080713_sec30_n.txt"
>>> f=file(fn,"rU")

>>> print f.next()
#Record of Sealevel(mm)
```

となって、ちゃんと一行分が読めているようです。open 関数の第 2 引数はファイルのモードを示します。r は読み込みモードを示します。U は改行文字が (『\r』,『\r\n』,『\n』) のいずれであっても受けるモードであることを指定します。

ヘッダはとりあえず、読み飛ばしてしまいましょう。

```
while 1:
      l=f.next()
      if not(l[0] == "#"):
        break
```

これを実行すると 変数 l には

```
>>> print l
2008/07/08 04:00:00  2959
```

と最初のデータが文字列として読み込まれています。

```
date, t, d=l.split()
```

とすることで、変数 date, t, d にそれぞれ日付、時間、データが収められます。`split()` は文字列型のオブジェクトが持つメソッドで、引数に指定された文字で文字列を部分文字列に分解します。引数に指定が無い場合には、空白文字で切り分けられます。

```
>>> print date
2008/07/08
>>> print t
04:00:00
>>> print d
```

```
2959
>>> type(d)
<type 'str'>
>>> d=int(d)
>>> print type(d),d
<type 'int'> 2959
```

最後のデータ d は split した直後は文字列になっていますので、整数として取り扱うために int 関数で強制的な型変換を行います。

3.1.3 時間データの変換

時間データを python で取り扱うためにここでは datetime モジュールを使ってみます。:

```
import datetime
dt=datetime.datetime(1,1,1)
```

この datetime オブジェクト dt を読み込んだ文字列 date と t から設定してみます。

```
dir(dt)
```

を実行すると

```
['__add__', '__class__', '__delattr__', '__doc__', '__eq__', '__ge__', '__
↪getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__',
↪'__new__', '__radd__', '__reduce__', '__reduce_ex__', '__repr__', '__rsub__', '__
↪setattr__', '__str__', '__sub__', 'astimezone', 'combine', 'ctime', 'date', 'day
↪', 'dst', 'fromordinal', 'fromtimestamp', 'hour', 'isocalendar', 'isoformat',
↪'isoweekday', 'max', 'microsecond', 'min', 'minute', 'month', 'now', 'replace',
↪'resolution', 'second', 'strftime', 'strptime', 'time', 'timetuple', 'timetz',
 'today', 'toordinal', 'tzinfo', 'tzname', 'utcfromtimestamp', 'utcnow', 'utcoffset
↪', 'utctimetuple', 'weekday', 'year']
```

と datetime オブジェクトのメソッドが表示されます。あるいは、

```
help(dt)
```

を実行すればより詳細な説明が表示されます。いずれにせよ、datetime オブジェクトの (Unix のシステム関数と同じ名前を持つ)strptime メソッドを使うことでなんとかなりそうです。:

```
dt=datetime.datetime(1,1,1)                                    # datetime オブジェクトをとりあえず作成。
dt=dt.strptime(" ".join((date,t)),"%Y/%m/%d %H:%M:%S")   # 文字列データから datetime オブジェクトの値を設定。
print dt
```

で確かに、思った通りの変換が行われていることがわかります。strptime は同名の Unix システム関数とおなじく、strftime 関数と同じフォーマット文字を受け付けます。フォーマット文字は」man strftime」をみることで調べることができます。

3.1.4 関数にまとめてみる。

ここまでの結果を関数にまとめると、

```
import datetime

def data(l):
  date,t,d=l.strip()
  d=int(d)
  dt=datetime.datetime(1,1,1)
  dt=dt.strptime(" ".join((date,t)),"%Y/%m/%d %H:%M:%S")
  return (dt,d)
```

となります。

ヘッダ部分についても

```
def header(l):
     h=l[1:].split()
     key=h[0]
     val=" ".join(h[1:])
     return dict(((key,v),))
```

ととりあえず定義しておくと、ファイル全体を読み込む関数として、

```
def readTide(fn):
    myhd=dict()
    myData=[]
    f=file(fn,"rU")
    for l in f:
       if l[0]=="#":
          myhd.update(header(l))
       else:
          myData.append(data(l))
    return (myhd,myData)
```

を定義することができそうです。

3.2 クラス化

ここまでで、潮位データを読み込んで Python で取り扱えるデータとするための関数を作成しました。関数作成までには、dir() や help() の助けを借りながら、システムの関数の動作を確認しながら、関数を作成していきました。

複数のデータセットを取り扱うさいには、さらに一歩すすめて、これらの関数とそれらの関数が返す値をクラスにまとめておくと便利なことがあります。次に、前節で作成した関数群をクラスにまとめてみます。

3.2.1 クラスの作成

前回作成した関数群は三つの関数 , header, data, readTide がありました。これらの関数を以下に示します。

```
import time

def header(l):
   h=l[1:].split()
   key=h[0]
   val=" ".join(h[1:])
   return dict(((key,val),))

def data(l):
   date,t,d=l.split()
   d=int(d)
   dt=time.strptime(" ".join((date,t)),"\%Y/\%m/\%d \%H:\%M:\%S")
   return (dt,d)

def readTide(fn):
   myhd=dict()
   myData=[]
   f=file(fn,"rU")
   for l in f:
       if l[0]=="#":
          myhd.update(header(l))
       else:
          myData.append(data(l))
   return (myhd,myData)
```

readTide 関数が主要な関数で、与えられたファイル名からデータを読み込んで、ヘッダの辞書型データと時刻毎のデータのリストを値として返します。header 関数と data 関数は一行毎の文字列をその行のタイプ（header かデータか）によって解釈し、しかるべきデータを値として返します。クラス化にはこれらの (file 名、ヘッダ、データリスト）をインスタンスの内部データとし、上記の関数をメソッドとすれば良さそうです。クラス化の一例としてこんなクラスを作ってみました。

```
#TideData.py
import time

class TideData:
   def __init__(self, fn):
       self.fn=fn
       self.hd=dict()
       self.data=[]
       self.ReadFile()

   def ReadFile(self,fn=""):
       if fn:
           self.fn=fn
       f=file(self.fn,"rU")
       for l in f:
           if l[0]=="#":
               self.HeaderLine(l)
```

```
            else:
                self.DataLine(l)

    def HeaderLine(self, l):
        h=l[1:].split()
        key=h[0]
        val=" ".join(h[1:])
        self.hd[key]=val

    def DataLine(self, l):
        date,t,d=l.split()
        d=int(d)
        dt=time.strptime(" ".join((date,t)),"%Y/%m/%d %H:%M:%S")
        self.data.append((dt,d))
```

クラスの初期化関数__init__では、一組のデータを表現する (file 名、ヘッダ、時間とその時の潮位のデータのリスト）を確保し、初期化しています。その後、ReadFile メソッドを呼び出すことで、ファイルからこれらの中身を読み込んでいます。ReadFile の中身は前回の readTide 関数とほぼ同じです。次に見るように、行毎の処理関数で、インスタンスのプロパティに直接新しいデータを追加していますので、return で値を返す必要はありません。前回の heder(), data() 関数は HeaderLine(), DataLine() メソッドになりました。Python ではプロパティ名とメソッド名は同じ名前空間に属するため、同じ名前のプロパティとメソッドが共存することはできません。ということで、メソッド名を変更しました。ここでもプロパティを各メソッドで直接変更することで、値をリターンする必要はなくなっています。

3.2.2 クラスの利用

このクラスの使い方の一例を示します。

```
import TideData,Gnuplot,time

td=TideData.TideData("11_20080706-20080710_sec30.txt")
gp=Gnuplot.Gnuplot()
gp.plot([(time.mktime(x[0]),x[1]) for x in td.data])
```

などとします（Gnuplot モジュールとそれが呼び出す gnuplot コマンドがシステムにインストルされていなければなりません)。この例では出力されるグラフの横軸は UTC 時間の秒数そのままになってしまいます。横軸ラベルを時間表示にする方法については、別の機会にご説明できればと思います。

複数のデータファイルを読み込む際は、

```
import TideData,Gnuplot,time


td0=TideData.TideData("11_20080701-20080705_sec30.txt")
td1=TideData.TideData("11_20080706-20080710_sec30.txt")
td2=TideData.TideData("11_20080711-20080715_sec30.txt")
td3=TideData.TideData("11_20080716-20080720_sec30.txt")
gp=Gnuplot.Gnuplot()
```

```
gp('set data style lines')
gp.plot([(time.mktime(x[0]),x[1]) for x in (td0.data+td1.data+td2.data+td3.data)])
```

以上で取得した潮位データをグラフ化して表示するための Python プログラムが出来上がりました。

3.3 EPICS CA monitor example

次の Python プログラムの一例として EPICS CA ライブラリを Python から利用してその値を画面に表示するためのプログラムを示します。EPICS CA ライブラリを Python から利用するために ca モジュールを利用します。ca モジュールは標準ライブラリではありませんので、通常はそれぞれインストールが必要です。KEK や J-PARC の制御システム計算機ではすでにインストール済みですので、ユーザはそれぞれが ca モジュールをインストールする必要はありません。

```
#!/bin/env python
"""
Simple EPICS monitor callback example
"""
import ca

class myCh(ca.channel):
    def __init__(self,name):
        ca.channel.__init__(self,name)

    def cb(self,valstat):
        self.update_val(valstat)
        print self.name, valstat[:3],ca.TS2Ascii(valstat[3])

def test():
   ch=myCh("fred")
   try:
       ch.wait_conn()
       try:
           ch.monitor(ch.cb)
           ca.pend_event(0)
       except:
           raise
   except:
       print "channel not connected"

if __name__ =="__main__":
    test()
```

cas モジュールを使った場合には、:

```
#!/bin/env python
"""
Simple example for monitor callback using cas.
"""
```

```
import cas

class myObj:
    def __init__(self,chname):
        self.name=chname
        self.ch=cas.caopen(chname)

    def cb(self,valstat):
        val,sevr,stat,ts=valstat
        print self.name,val,sevr,stat,cas.ca.TS2Ascii(ts)

    def startmonitor(self):
        cas.camonitor(self.name,self.cb)

def test(name="fred"):
    obj=myObj(name)
    obj.startmonitor()
    try:
        cas.ca.pend_event(0)
    finally:
        cas.caclear_monitor("fred",ch)

if __name__ == "__main__":
    test("jane")
```

3.4 RDB の利用

次の例は relational databse 中のデータを Python から取り出して、利用すると言う物です。RDB を Python から取り扱うためのモジュールは幾つか存在しますが、ここでは PostgreSQL をサポートするモジュールの一つである `psycopg2` を使う例を示します。

この例では PostgreSQL に納められた EPICS record についての設定データから 350BT のアラーム状態を表すレコードに関する情報をとりだした後、KEKBAlarm プログラムのための設定データファイルを作成します。まずは PostgreSQL から必要な EPICS record に関する情報をとりだすためのモジュール `JK350BTAlarm.py` を定義します。:

```
#!/bin/env python
# -*- encoding:utf-8 -*-
# file:JK350BTAlarm.py
import psycopg2
import cStringIO

class epics_records:
    _dbname="jk"
    _tablename="epics_records_test1"
    _desc_tablename= "epics_records_desc_test1"
    _user="readonly"
    _passwd="readonly"
    _host="jkjrdb01.ccr.jkcont"
```

```
    _con=None

    def __init__(self):
        if not self._con:
            self._con=psycopg2.connect(database=self._dbname,
                                       host=self._host,
                                       user=self._user,
                                       password=self._passwd)
        self.cursor=self._con.cursor()

    def select_fault_record(self):
        self.cursor.execute("select * from epics_records_test1 where name ~ '350BT.
↪*STAT\:FAULT.*' and (type = 'bi' or type = 'calc')")

    def select_off_record(self):
        self.cursor.execute("select * from epics_records_test1 where name ~ '350BT.
↪*STAT\:OFF.*' and (type = 'bi' or type = 'calc')")

    def select_interlock_record(self):
        self.cursor.execute("select * from epics_records_test1 where name ~ '350BT.
↪*ILK\:.*' and (type = 'bi' or type = 'calc')")
```

次にこの JK350BTAlarm.py モジュールを利用して KBKAlarm 設定ファイルを作成するプログラム genKEKBAlarmConfig.py を用意します。:

```
#!/bin/env python
#-*- encoding utf-8 -*-
#file:genKEKBAlarmConfig.py
import cStringIO

class KEKBAlarmConfigElement:
    def __init__(self, channel, message,
                 connectMask=-1, alarmMask=-1, alarmXor=0, alarmValue=None):
        self.channel=channel
        self.message=message
        self.connectMask=connectMask
        self.alarmMask=alarmMask
        self.alarmXor=alarmXor
        self.alarmValue=alarmValue

    def __str__(self):
        fmt0="{\"%s\",\"%s\",%s, %s, %s}"
        fmt1="{\"%s\",\"%s\",%s, %s, %s, %s}"
        if self.alarmValue:
            return fmt1%(self.channel,
                         self.message,
                         self.connectMask,
                         self.alarmMask,
                         self.alarmXor,
                         self.alarmValue)
        else:
            return fmt0%(self.channel,
```

```
                        self.message,
                        self.connectMask,
                        self.alarmMask,
                        self.alarmXor,)

class KEKBAlarmConfigFile:
    def __init__(self, name="AlarmConf"):
        self.name=name
        self.ch_f=open(name+"Channels.sad","w")
        self.check_f=open(name+"Check.sad","w")
        self.elements=[]

    def __del__(self):
        self.ch_f.close()
        self.check_f.close()

    def add(self,channel, message, connectMask=-1, alarmMask=-1, alarmXor=0,␣
↪alarmValue=None):
        self.elements.append(KEKBAlarmConfigElement( channel, message, connectMask,␣
↪alarmMask, alarmXor, alarmValue ) )

    def __str__(self):
        fmt="""
        SetEnv["EPICS_CA_ADDR_LIST",GetEnv["EPICS_CA_ADDR_LIST"]//" 10.64.31.16"];
        %sChannels={
        %s};
        """
        strIO=cStringIO.StringIO()
        for e in self.elements:
            strIO.write(str(e))
            strIO.write(",\n")
        return fmt%(self.name,strIO.getvalue())


def test():
    cf=KEKBAlarmConfigFile("JKJ350BT")

    import JK350BTAlarm
    db=JK350BTAlarm.epics_records()
    db.select_fault_record()
    for e in db.cursor:
        cf.add(e[0],e[3])

    db.select_off_record()
    for e in db.cursor:
        cf.add(e[0],e[3])

    db.select_interlock_record()
    for e in db.cursor:
        cf.add(e[0],e[3])

    print cf
    cf.ch_f.write(str(cf))
```

```
if __name__ == "__main__":
  test()
```

注釈: このプログラムの最後の二行は Python プログラムのいわば定跡で、このプログラムファイルがモジュールとして `import` された場合には、`test()` は実行されないが、**python** コマンドの引数として指定される等メインプログラムとして起動された際には `test()` 関数を実行するという意味になっています。プログラムの再利用性を意識したプログラム作成に慣れておくことは、将来特に有用です。

第 4 章

Python への招待

Date　2017 年 12 月 15 日,

Version　0.1 / 0.1

Author　Noboru Yamamoto

Organization　Accelerator Control Group, Accelerator Laboratory, KEK, JAPAN and Graduate University

この文書は、これからプログラミングについて学ぼうという初心者に向けて書かれる。Python プログラミング言語を使いながら、コンピュータ　プログラミングについて学んでいく。コンピュータプログラムに関する知識を仮定しない。

Contents:

4.1 コンピュータプログラミングとは何か

4.2 Excel ファイルの取り扱い

Microsoft 社の Excel はいわゆる表計算プログラムのなかである種の標準としての位置を持っている。その他の表計算プログラムでもこの Excel の取り扱う形式のファイルは入出力のファイル形式に利用可能であることが多い。

Excel の入出力ファイル形式もその歴史の長さからいくつかの形式が今でも使われている。一つは長く使われてきた xls 形式であり、最近の主流である xlsx 形式である。xls 形式はバイナリフォーマットであるが、xlsx 形式はその中身は xml を基本としたファイル形式となっている。

これらの Excel のサポートするファイル形式を Python から入出力’するためのモジュールとして、いくつかのモジュールが開発／提供されている。

openpyxl　xlsx をサポート。xls 形式はサポートしない。Book/Worksheet オブジェクトをサポート。

xlrd, xlwt　xls 形式ファイルの入力 (xlrd) および出力 (xlwt). xlsx 形式もサポート。Excel ファイルを読み込んで、データにアクセスする基本的な機能が実装されている。pandas などの高度なデータ形式を直接サポートしているわけではない。

pandas　pandas も excel ファイルを読み込む機能を持っている。読み込んだデータは pandas の dataframe オブジェクトになっている。内部的には xlrd/xlwt を使っているようである。

4.2.1 xlrd Example

https://blogs.harvard.edu/rprasad/2014/06/16/reading-excel-with-python-xlrd/ から

```
from __future__ import print_function
from os.path import join, dirname, abspath
import xlrd

fname = join(dirname(dirname(abspath(__file__))), 'test_data', 'Cad Data Mar 2014.
↪xlsx')

# Open the workbook
xl_workbook = xlrd.open_workbook(fname)

# List sheet names, and pull a sheet by name
#
sheet_names = xl_workbook.sheet_names()
print('Sheet Names', sheet_names)

xl_sheet = xl_workbook.sheet_by_name(sheet_names[0])

# Or grab the first sheet by index
#  (sheets are zero-indexed)
#
xl_sheet = xl_workbook.sheet_by_index(0)
print ('Sheet name: %s' % xl_sheet.name)

# Pull the first row by index
#  (rows/columns are also zero-indexed)
#
row = xl_sheet.row(0)  # 1st row

# Print 1st row values and types
#
from xlrd.sheet import ctype_text

print('(Column #) type:value')
for idx, cell_obj in enumerate(row):
   cell_type_str = ctype_text.get(cell_obj.ctype, 'unknown type')
   print('(%s) %s %s' % (idx, cell_type_str, cell_obj.value))

# Print all values, iterating through rows and columns
#
num_cols = xl_sheet.ncols   # Number of columns
for row_idx in range(0, xl_sheet.nrows):    # Iterate through rows
    print ('-'*40)
    print ('Row: %s' % row_idx)   # Print row number
    for col_idx in range(0, num_cols):  # Iterate through columns
```

```
        cell_obj = xl_sheet.cell(row_idx, col_idx)  # Get cell object by row, col
        print ('Column: [%s] cell_obj: [%s]' % (col_idx, cell_obj))
```

4.2.2 比較

4.2.2.1 excel ファイルの読み込み

いずれの場合も Excel ファイルを読み込んで、Workbook オブジェクトを作成する。xlrd は xls,xlsx のどうちらのファイル形式にも対応している。openpyxl は xlsx 形式のみ。xls を開こうとすると xlrd を使えとのメッセージが出た。

xlrd `wb=xlrd.open_workbook(u"./data/MRshift_Run#72(20170301-0331)all.xls")`

openpyxl `wb=openpyxl.reader.excel.load_workbook(u"./data/MRshift_Run#72(20170301-0331)all.xlsx")`

4.2.2.2 worksheet の選択

workbook オブジェクトから work シートを選択する。worksheet はシートの名前あるいは、ブック中のシートの順番を指定して取り出す。

xlrd

```
sheet_names=wb.sheet_names()

sheet=wb.sheet_by_index(0)
sheet=wb.sheet_by_name(sheet_names[0])
```

openpyxl

```
ws0=wbk.worksheets[0]
```

4.2.2.3 データのアクセス方法

ワークシートオブジェクトは基本的には 2 次元のテーブルである。このテーブルの中の要素にアクセスする方法について示す。

xlrd は基本的に 2 次元のテーブルを要素番号 (0 始まり）を指定して、要素を取り出します。一方、openpyxl は Excel などでよく使われる」A1」型の要素指定と要素番号による指定のどちらも使うことができます。このとき行 (row), 列 (column) の要素番号が 1 始まりであることに注意します。

xlrd

```
th=ws0.nrows
tw=ws0.ncols
r=ws0.row(0)
```

```
c=ws0.col(0)
v=ws0.cell(0,0).value # ws0.cell_value(0,0)
```

openpyxl

```
ws0=wbk.worksheets[0]
th=ws0.max_row
tw=ws0.max_column
cell_a1=ws0["A1"]
crange=ws0["A1:B2"]
v=ws0.cell(row=1,column=1).value
```

4.3 Python の八つの特徴

Python を学ぶ上で、(特に既にプログラム言語の知識がある方が）知っておくべき八つの Python の特徴を上げてみます。

1. Python ではインデントが意味をもつ。
2. python は構造化プログラムをサポートする。
3. Python はオブジェクト指向プログラムをサポートする。
4. python は関数型プログラムをサポートする。
5. Python の文字列は immutable(変更不能な) 文字のリストである。
6. Python にはポインタ型のデータは存在しない。
7. Python のライブラリはモジュールと呼ばれる。
8. Python には実用的な哲学 (Python 禅,Zen of Python) がある。

Python は簡単なことをとても簡単にし、むずかしいことが 多すぎることのないようにするという、賢明な妥協の哲学をもっている

Python 禅の全文は Python インタプリタの中でよむことができる。

```
% python
```

Python 2.7.4 (v2.7.4:026ee0057e2d, Apr 6 2013, 11:43:10) [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin

Type 「help」, 「copyright」, 「credits」 or 「license」 for more information.

```
>>> import this
```

The Zen of Python, by Tim Peters

*Beautiful is better than ugly.

*Explicit is better than implicit.

*Simple is better than complex.

-Complex is better than complicated. -Flat is better than nested. -Sparse is better than dense. -Readability counts. -Special cases aren’t special enough to break the rules. -Although practicality beats purity. -Errors should never pass silently. -Unless explicitly silenced. -In the face of ambiguity, refuse the temptation to guess. -There should be one– and preferably only one –obvious way to do it. -Although that way may not be obvious at first unless you’re Dutch. -Now is better than never. -Although never is often better than *right* now. -If the implementation is hard to explain, it’s a bad idea. *If the implementation is easy to explain, it may be a good idea. *Namespaces are one honking great idea – let’s do more of those! >>>

4.4 Indices and tables

- genindex
- modindex
- search

索引