初心者用
Python 講座

第 4 版 2008 年 5 月 16 日

文責：斎藤輪太郎

Pythonの勧め

Python が他のプログラミング言語と比較して優れているのは、考えをプログラムとして表現したり、プログラムを書いた人の意図を読み取る時の障害が少なく人間にやさしいという点にあります。プログラミングを職業としない人も、もうプログラミング言語をいくつか習得した人も、一つだけあるいはもう一つだけ言語を学ぶとしたら Python を強くお勧めします。細胞シミュレーションプラットフォーム E-Cell System も Python をベースに作られています。

ぜひこの機会に本テキストで勉強することをお勧めします。

高橋恒一
E-Cell System アーキテクト
The Molecular Science Institute

※この序文は第 3 版 2007 年 6 月 12 日に対して寄せられたものです。

0. はじめに

Python はリスト構造やハッシュ構造を扱える強力なスクリプト言語であり、他のスクリプト言語(Perl など)と比べたときの主な利点は以下の２つです。

(1) TAB による字下げを文法の中に採り入れており、括弧の少ない非常に見やすいプログラムを書くことが可能です。
(2) Python の言語設計思想に初めからオブジェクト指向が取り入れられており(C や Perl は後からオブジェクト指向を取り入れました)、オブジェクト指向プログラミングがしやすくなっています。

これまで他のスクリプト言語でオブジェクト指向によらないプログラミングをしていたユーザが Python によるプログラミングに慣れると、以下のような印象を持つのではないでしょうか。

【プログラムの見やすさ、分かりやすさ】
・プログラムが大変見やすい。プログラムのコメント・ドキュメンテーションが自然にできる。従って時間が経過しても書いたプログラムの解読が容易。
・文法が単純なので、極めて短期間で実用なプログラムを書く技術を習得できる。
・初心者と上級者でコーディングがそれほど変わらない。他人のコードを読みやすい。
・プログラムを小さな単位(モジュール)に分解しやすい。おかげで保守が楽になった。

【オブジェクト指向】
・オブジェクト指向プログラミングがしやすい。
・プログラムの再利用が促進される。使い捨てのプログラムを書く機会が減った。
・複数のプログラムを次々と動かし途中の結果を記録した中間ファイルを渡してゆく、という面倒な処理をすることが少なくなった。特に面倒な中間ファイルを作成する機会が減った。
・プログラムの修正が容易。プログラムを修正するときに、見なければならない箇所が減った。プログラムの動作を変更したい場合、修正箇所が少なくて済む。
・大規模なシステムを作成することが可能。

本講座では Python のエッセンスを手短に伝え、皆さんができるだけ早く実用的な Python プログラムが書けるようになることを目的とします。

1. 文字列の表示

まずは、なにか文字列を出力してみるところから始めましょう。一番簡単なのはコマンドプロンプトで単純に python と打ち込んで対話モードに入り、print “Hello”と入力してみることです。

```
[user@local ~]$ python   ← Python 対話モードの起動
Python 2.3.4 (#1, Oct 11 2006, 06:18:43)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello"   ← 打ち込む
Hello   ← 文字列が表示された
>>>   ← Ctrl+d で抜ける
[user@local ~]$
```

対話モードからは Ctrl+d で抜けます。対話モードでは打ち込んだ命令が使い捨てになってしまいます。そこでちゃんとプログラムファイルを作って何度も使えるようにします。まず以下のようなファイルを作成して下さい。ファイル名を hello.py とします。

```
#!/usr/bin/env python

print “Hello, world!”
```

上記の Python のスクリプト(プログラム)を作成したら、以下のようにして実行権を与えましょう。

```
chmod +x hello.py
```

そして以下のようにして hello.py を実行します。

```
./hello.py
```

Hello, world!という文字列が出力されましたか？このように print は次に続く文字列を出力する機能があります。次に print の行を、

```
print 1+2
```

に変えて実行してみましょう。どうなりましたか？print に数式が続く場合、その数式が計算されて答えが出力されます。

課題 1-1：123 x 456 を計算させてみましょう。掛け算には * を使います。

2. 変数とその演算

変数は数値や文字列を一時的に入れておく入れ物のような存在です。

```
val = 150                  # 1
name = “Watson Crick”      # 2
```

#1 は 150 を val という変数に代入します。

#2 は”Watson Crick”という文字列を name に代入します。

変数で演算を行うことも可能です。

```
val1 = 10 + 20             #3
val2 = 20 / 2              #4
val = val1 + val2          #5
val = val + 1              #6
```

#3 は 30 を val1 に代入します。

#4 は 10 を val2 に代入します。

#5 は変数 val1 と val2 を足した結果を val に代入します。

#6 は val に 1 を加え、それを新しい val の値とします。

シャープ記号(#)の後は、コメントとして扱われ、プログラムの実行には影響を与えません。

２つの文字列をつなげることも可能です。

```
str1 = “Yukichi”
str2 = “Fukuzawa”
str3 = str1 + “ “ + str2
```

str3 には str1 と一文字空白と str2 をつなげた”Yukichi Fukuzawa”が入ります。

文字列と数値をつなげるには、バッククオートを用い、`数値を表す変数名`で数値を文字列に変換します

```
num = 7
str4 = “Number #” + `num`
```

課題 2-1： 変数 val1 に 123、val2 に 456 を代入し、この二つを足した数を val3 に代入してみましょう。また答えを print val3 で表示しましょう。

3. ユーザ入力

プログラム実行中にユーザが処理の流れを決めたいときがしばしばあります。そのような時にユーザからの入力を受け付けられるようにすると便利です。

```
input_str = raw_input(“Input something: ”)
```

とすると、“Input something: ”の文字列が表示された後ユーザの入力を受け付け、ユーザ入力の結果が input_str に代入されます。以下のプログラムを打ち込んでみましょう。

```
#!/usr/bin/env python

user_input = raw_input(“Input any words or sentence: “)  #1
print “You typed: “, user_input #2
```

#1 でユーザの入力を待ちます。ここで何か文字列を入力してみましょう。
#2 で”You typed: “とともにその入力された文字列を表示します。

raw_input は文字列の入力を受け付けますが、数値を受け付けるためには input を使います。

```
input_number = input(“Input any number: “)
```

課題 3-1：ユーザから数値の入力を受け付け、その２倍の数を表示するプログラムを書きましょう。

出力例：

```
Input number：12    ← ユーザが 12 と入力
24                  ← 出力
```

課題 3-2：ユーザから２つの数値の入力を受け付け、その和を表示するプログラムを書きましょう。

出力例：

```
Input first number：12      ← ユーザが 12 と入力
Input second number：13     ← ユーザが 13 と入力
25                          ← 出力
```

4. 基本的な条件分岐：if 文

ある条件によって処理の内容を変えたいということはよくありますね。多くの他の言語と同様、Python でも if 文を使って条件分岐を行います。以下のプログラムを打ち込んで動作を確認しましょう。

```
#!/usr/bin/env python

input_number = input(“Input any natural number: “)
if input_number % 2 == 0:
        print “That’s even number.”
else:
        print “That’s odd number.”
```

偶数を入力すると、”That’s even number”、奇数を入力すると”That’s odd number”と出力されます。

・`input_number = input(“Input any natural number: “)`でユーザからの自然数の入力を促します。

・input_number % 2 は input_number を 2 で割ったときの余りです。これが 0 なら、その下のタブで 1 つ右にシフトした文すなわち、`print “That’s even number.”`を実行します。

・そうでなければ、else 以下のタブで 1 つ右にシフトした文すなわち、`print “That’s odd number.”`が実行されます。

---

if の構文は以下の通りです。

---

if 条件:
        条件に合ったときの処理
elif 2 番目の条件:
        2 番目の条件に合ったときの処理
else:
        どの条件にも合わなかったときの処理

Pythonではタブが重要な意味を持ちます。if,elseの後の処理はスペースではなく、必ずタブを使用して１つ右にシフトさせるようにします。

---

if 条件：

< *タブで右へ* > 処理

また処理系によってはタブの前後に余計なスペースが入っているとエラーが出る事がありますので、注意しましょう。

課題 4-1：ユーザより２つの数の入力を受け付け、１番目の数 < ２番目の数のときは”Smaller”、１番目の数 > ２番目の数のときは”Larger”、１番目の数 = ２番目の数のときは”Equal”と表示するプログラムを作成しましょう。

5. while 文によるループ

これまでのプログラムでは、実行の流れがかならず上から下でした。ここでは、プログラムの流れを下から上に戻す手法（ループ）を学びます。一番簡単な while 文から学びましょう。

while 文はある条件を満たしている間は一定範囲のスクリプトを実行し続ける文で、以下のような構造をしています。

---

while 条件:
　　処理 1
　　処理 2
　　処理 3

条件を満たしている間、タブで下げられた部分が繰り返し実行されます。以下のプログラムを入力して実行してみましょう。

```
#!/usr/bin/env python

num = 0
while num <= 5:
    print num, “ “,
    num = num + 1
```

実行すると以下のような結果が得られるはずです。

```
0 1 2 3 4 5
```

- num = 0 で最初に num を 0 にセットします。
- num が 5 以下の間、print num, “ “,(数と空白の出力)と num = num + 1(num の値を 1 つ増やす)が繰り返されます。
- num が 5 になると条件に合わなくなり、そこで while 文は終了です。
- 結果的に num が 0 から 5 まで増え、それが出力されます。


while 文や後述する for 文のループを強制的に抜け出す命令として、break、ループの次のサイクルを強制的に開始する命令として continue などがあります。

課題 5-1：while 文を使って 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 を計算しましょう。

課題 5-2：ユーザが入力した数の合計を求めるプログラムを書きましょう。ユーザは最初に数の個数を入力し、次に合計を求めたい数を入力するようにします。

Input number of values: *3*　　← ユーザが 3 と入力
Input value #1: *13*　← ユーザが 13 と入力
Input value #2: *5*
Input value #3: *9*
27　　← 出力

6. リストとタプル

似たようなデータの集合を同じ変数名で一元的に管理したいときにリストはとても便利です。例えば、

```
name = [ “Thomas”, “John”, “Angela”, “Joanna” ]
```

とすれば、name[0]は”Thomas”,name[1]は”John”を表します。従って、

```
name = [ “Thomas”, “John”, “Angela”, “Joanna” ]
print name[1], name[2]
```

とすれば、John Angela と出力されます。

この例で分かるように使い方は、初期設定は、

変数名 =[要素 0,要素 1,要素 2, …]

で行い、その後リストの要素を参照・変更するには

変数名[添字]

とします。例えば var1[10]は var1 という配列の 10 番目(0 から数えるなら 11 番目)の要素を表します。

```
name[2] = “Angie”
```

とすれば、２番目の要素は”Angela”から”Angie”へ変更されます。また len(name)は name に入っている要素の個数を表します。

今まで使ってきたような配列変数でない変数のことをスカラ変数と呼びます。

次のスクリプトをみてみましょう。

```
#!/usr/bin/env python

name = [“Thomas”, “John”, “Angela”, “Joanna”]
i = 0
while i < len(name):
   print name[i], “ “,
   i = i + 1
print
```

以下のような出力が得られるはずです。

```
Thomas John Angela Joanna
```

- name = [“Thomas”, “John”, “Angela”, “Joanna”]で name にリストを設定します。
- i = 0 で i の値を 0 にセットします。
- while i < len(name)で i が name の要素数(この場合、4)を下回っている間、以下のタブ区切りの部分が実行されます。
- print name[i], “ “,でリストの i 番目が空白とともに出力されます。最後のカンマは改行を防ぎます。
- i = i + 1 で i の値が 1 つ増えます。
- 結果的に i が 0 から 3 まで増え、リストの 0 から 3 番目までが出力されます。
- while 文の外で print 文が実行され、改行が行われます。

リストと非常に似たものにタプルがあります。これは[…]ではなく、(…)の中に要素を記述します。

```
name = (“Thomas”, “John”, “Angela”, “Joanna”)
```

リストなら、name[2] = “Angie”のように後から一部を変更できますが、タプルでは変更ができません。従って後から中味を変更したいものをリストに、変更したくないものをタプルにすると便利でしょう。

リスト名.append(要素)でリストの後方に要素を追加することができます。例えば、

```
name = []
```

として name がリストであることを宣言しておいてから、

```
name.append(“Thomas”)
name.append(“John”)
name.append(“Angela”)
```

とすれば、[“Thomas”, “John”, “Angela”]のようなリストができます。

リストの変数の代入操作(b=a)は a という中味のコピーを b に作るわけではなくて、b が指し示すものは a が指し示すものと同一ということを意味します。例えば以下の文を実行してみましょう。

```
a = ["Hop", "Step", "Jump"]
b = a
b[1] = "Skip"
```

```
print a
```

リスト b の操作によってリスト a の中味も変わっているのが確認できると思います。リストや後述のディクショナリを用いるときは注意が必要です。a の内容のコピーを b に作成したいときは、

```
b = a[:]
```

とします。

課題 6-1 : ユーザが入力した文字列を逆順に表示するプログラムを書きましょう。ユーザは最初に入力する文字列の個数を入力し、次に順次文字列を入力します。

```
Input number of strings: 4        ← ユーザが 4 と入力
Input string #1: Adenine     ← ユーザが Adenine と入力
Input string #2: Cytosine
Input string #3: Guanine
Input string #4: Thymine
Thymine                           ← 出力
Guanine
Cytosine
Adenine
```

7. for 文によるループ

for 文も while 文と同じようなループを制御する文です。以下のような構造をしています。

---

for 変数 in リスト:

　　処理 1

　　処理 2

　　処理 3

while 文のときと同じように、処理はタブで 1 つ右にずらしておきます。

for 文ではリストやタプルの中の要素が毎回変数に代入されてから、処理が実行されます。リストの中の要素を使い切るとそこで for 文は終了となります。以下のプログラムを打ち込んで実行してみましょう。

```
#!/usr/bin/env python

names = (“Thomas”, “John”, “Angela”, “Joanna”)
for person in names:
   print person, “ “,
print
```

names の中の要素が先頭から 1 つずつ変数 person に代入されます。そして代入されるたびに処理(この場合、print person, “ “,)が行われます。

課題 7-1：下の例に示すように、ユーザから入力された複数の項目に対して、先頭に”You typed :”を付加して出力しましょう。このとき、出力時に for 文を用いましょう。

```
Input number of items: 3        ← ユーザが 3 と入力
Input string #1: Adenine     ← ユーザが Adenine と入力
Input string #2: Cytosine
Input string #3: Guanine
You typed: Adenine          ← 出力
You typed: Cytosine
You typed: Guanine
```

8. ディクショナリ

Python ではキーと値を対応させるような体系を持った変数があります。それがディクショナリです。例えば文字通り、簡単な英和辞書を作ってみましょう。

```
E_to_J = { “Apple”: “Ringo”, “Orange”: “Mikan”, “Grape”: “Budo” }
print E_to_J[ “Orange” ]
```

ディクショナリの中味は{}の中に記述します。そして、

キー：値

の対応関係を並べてゆきます。後からキーに対応する値を参照するには、

ディクショナリ名[キー]

とします。E_to_J[“Orange”]は”Mikan”を表す事になります。上記は、

```
E_to_J = {}
E_to_J[“Apple”] = “Ringo”
E_to_J[“Orange”] = “Mikan”
E_to_J[“Grape”] = “Budo”
print E_to_J[ “Orange” ]
```

のように書き換えることができます。最初に E_to_J = {}で E_to_J がディクショナリであることを宣言しています。

ディクショナリ名.keys()

でディクショナリに含まれる全てのキーがリストとして返されます。試しに、上記プログラムの後に

```
print E_to_J.keys()
```

を入れて実行してみましょう。以下が出力されるはずです(但しリストの中の順番は同じとは限りません)。

```
[‘Apple’, ‘Orange’, ‘Grape’]
```

課題 8-1: ディクショナリの全てのキーと値を一行ずつ出力するプログラムを for 文などを用いて作成しましょう。ディクショナリに対する for 文で keys()が省略できることも確かめて下さい。

プログラム例：

```
E_to_J = { “Apple”: “Ringo”, “Orange”: “Mikan”, “Grape”: “Budo” }
```

```
# ここに for 文などを用いたプログラムを作成
#
```

実行結果例（順番が同じとは限らない）：

```
Apple Ringo
Grape Budo
Orange Mikan
```

9. ファイルのオープンと行単位の読み込み

まず以下のようなファイルを作ってみましょう。ファイル名を food.txt とします。

```
I like an apple.
He ate a banana.
I cooked some corn.
She has some donuts.
```

次に以下のような Python スクリプトを書いてみましょう。ファイル名を eat.py とします。

```
#!/usr/bin/env python

fh = open(“food.txt”, “r”)
for line in fh:
       print line
fh.close()
```

そして eat.py を実行してみましょう。food.txt の内容がそのまま出力されましたか？ではスクリプトの説明をしましょう。

- fh = open(“food.txt”, “r”)は”food.txt”というファイルを扱う（オープンする）ことを宣言します。そして fh という変数（ファイルハンドル）でこのファイルを管理します。ファイルの読み込みは全てこの fh というファイルハンドルを使って行います。fh でなくても、handle など任意の名前をつけることができます。
- for line in fh:で一行ずつが line に読み込まれ、直後にある TAB で右に一段シフトした処理(この場合、print line)が行われます。読み込みはファイルが終わるまで行われます。つまり通常は、food.txt の行数分だけ直後の処理が実行されます。
- print line で読み込んだ一行を表示します。
- fh.close()でファイルを閉じます。

課題 9-1：ファイル中の行数を数える Python スクリプトを書きましょう。

課題 9-2：テキストファイルを行末から行頭へ向かって表示するようなスクリプトを書きましょう。

例：

1. Albatross 2. Eagle 3. Birdie 4. Par 5. Bogey	→
	5. Bogey 4. Par 3. Birdie 2. Eagle 1. Albatross

10. 関数

10.1 簡単な関数の作り方

関数は特定の処理を行うための機能単位です。例えば、以下の文字列を出力する処理を考えてみましょう。

```
print “****************************************”
print “ This work was done by Yukichi Fukuzawa”
print “****************************************”
print “ Last update: 2006/08/11”
```

上記のような処理がプログラムの至るところで必要なとき、上記４行をその至るところで書くのは大変です。そこで例えばプログラムの先頭の方で、

```
#!/usr/bin/env python

def print_author():
       print “****************************************”
       print “ This work was done by Yukichi Fukuzawa”
       print “****************************************”
       print “ Last update: 2003/10/5”
```

としておくと、上記処理が必要な箇所で、

```
print_author()
```

とするだけで処理が実行されます。４行が１行に圧縮されたことになります。なお、`def print_author()`の直下に

```
def print_author():
       “””This function prints the author.”””
                      :
                      :
```

と関数の使い方の注釈を入れておくと、後で `print print_author.__doc__`で関数の使い方を表示することができて便利です。

独自の関数を作ることは、プログラミング作業の効率化に大きな役割を果たします。

課題 10-1 “Hello, world!”と表示する関数を作成しましょう。

10.2 関数の引数

「10.1 簡単な関数の作り方」で作成した関数は毎回必ず同じ文字列を出力するものでした。しかし、場合によって関数の処理内容を微妙に変えたいケースが頻繁にあります。例えば、先ほどの関数は"Yukichi Fukuzawa"と出力していましたが、それを場合によって"Shigenobu Okuma"に変更したい場合はどうすればいいでしょうか？

そこで関数には、「引数」と呼ばれる値を与えることができます。引数は関数が受け取る変数のことです。関数では引数によって処理の内容を調整する事ができます。例えば"Yukichi Fukuzawa"のところを場合によって"Shigenobu Okuma"に変更したい場合、

```
def print_author(author):
        print "****************************************"
        print " This work was done by ", author
        print "****************************************"
        print " Last update: 2003/10/5"
```

とします。そして print_author(author = "Yukichi Fukuzawa")と呼び出すと、author という変数に"Yukichi Fukuzawa"が代入され、print " This work was done by ", author_name のところで"Yukuchi Fukuzawa"と出力されます。

ここで、print_author(author = "Shigenobu Okuma")とすれば、"Yukichi Fukuzawa"が"Shigenobu Okuma"に変わって実行されます。この場合の括弧の中の"Yukichi Fukuzawa"や、"Shigenobu Okuma"が引数になります。

関数は、

```
def function(引数を受け取る変数１, 変数２, …):
        処理１
        処理２
        処理３
```

のように、関数名の前に「def」を付加し、その処理内容をタブのブロックで記述します。呼び出すときは、

```
function(引数を受け取る変数１ = 引数１, 引数を受け取る変数２ = 引数２)
```

とします。なお、上記は

```
function(引数１, 引数２)
```

と省略可能です。

課題 10-2 与えられた２つの整数の和を表示する関数 sum_of_pair を作成しましょう。

10.3 関数の返り値

関数は数学的には $f(x) = 2x$ のように、入力値に対して高々 1 つの出力値を対応させるものとして定義されます。Python の関数でも最終的に決められた値「返り値」を return 文を用いて決めることができます。

例えば、 $f(x) = 2x$ を実装したい場合、

```
def f(x):
      return x * 2
```

とします。そして、

```
print f(10)
```

とすれば、f(10)が 20 を返すので、20 という値が表示されます。

課題 10-3 与えられた２つの整数の和を「返す」関数 `sum_of_pair` を作成しましょう。

10.4 ローカル変数

Python では関数の中で新規に作成した変数はその関数の中だけで通用するものとなります。これをローカル変数と呼びます。例えば、

```
#!/usr/bin/env python

def diff(x, y):
        z = x – y # 関数内で定義されたローカル変数 z
        return z

z = 10 # 関数の外で変数 z を定義
print diff(10,3)
print z
```

のようにした場合、関数内で定義されたローカル変数 z は関数の中だけで通用する名前となります。このため、print diff(10,3)としてローカル変数 z が 7 になっても、関数の外で定義された z は 10 のままです。

ローカル変数は非常に重要な意味を持ちます。例えば複数の人が書いたプログラムを統合したとき、複数の人が同じ名前の変数を別の意味で使っていた、ということがよく起こります。このときローカル変数を使用していれば、表記上の名前は同じでもそれぞれの関数内で独立した変数として扱われるので、名前の衝突によってプログラムがうまく動かなくなるという心配がいりません。

課題 10-4：2 つの数値を受け取り、その 2 つの変数とその間の整数をすべて足し合わせて結果を返す関数 total を定義しましょう。引き数は正の整数であるとし、最初の引数が 2 番目の引数より常に小さいと仮定します。例えば total(3,7)は 25 となります。

10.5 組み込み関数

Python にはあらかじめ組み込まれている関数がいくつかあります。例えば既に学んだファイルの入出力を開始する open、ユーザから入力を受け付ける input、リストやタプルの長さを返す len などは組み込み関数です。必要に応じて組み込み関数を使用すれば、自分でプログラムを作成する労力を減らす事ができます。

よく使う関数に range がありますこれは range(n)で 0 から n−1 までの整数を順番に持つリストができあがります。例えば、

```
print range(10)
```

とすると、

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

が得られます。2 つの整数n_1, n_2を指定すると、リスト[n_1, n_1+1, n_1+2, …, n_2-1]が得られます。例えば、

```
print range(3,10)
```

は

```
[3, 4, 5, 6, 7, 8, 9]
```

となります。

他にリストやタプルに含まれる数の合計を求める関数 sum など様々な関数が組み込まれています。

課題 10-5：range と len を用いて長さの同じ 2 つのリストの同じ位置にある要素の積の総和を求める関数 inner_prod を作成しましょう。例えば、

```
print inner_prod((1, 2, 3), (4, 5, 6))
```

の出力は 1×4 + 2×5 + 3×6 = 32 となります。

11. モジュール

モジュールはプログラムの構成単位です。より端的に言えば、Python では 1 つのプログラムファイルが 1 つのモジュールに該当します。皆さんが今まで作成した Python のプログラムファイルは全てモジュールということになります。

モジュールはプログラムの再利用や、共同で行うプログラミング作業を容易にします。import 文を用いれば、他のモジュールを簡単に利用できるのです。以下のプログラム calc.py を打ち込んで、このモジュールを実行してみましょう。

calc.py

```
#!/usr/bin/env python

def sum(nums):
	total = 0
	for num in nums:
		total += num
	return total

if __name__ == “__main__”:
	sample_list = [ 2,5,7,10 ]
	print “List is: “, sample_list
	print “Total is: “, sum(sample_list)
```

if __name__ == “__main__”はモジュールがどのように実行されているのかを判定しています。

```
./calc.py
```

で実行すると、__name__は自動的に“__main__”となり、if __name__ == “__main__”:以下が実行されます。

さてこのモジュールを他のモジュールから利用してみましょう。今度は以下の imp_test.py を作成しましょう。

imp_test.py

```
#!/usr/bin/env python

```

```
import calc
test_list = [1, 3, 9, 10 ]
print “List is: “, test_list
print “Total is: “, calc.sum(test_list)
```

この例で分かるように、calc.py をモジュールとして利用するには、プログラムの先頭で import calc と打ち込みます。また calc.py の中の sum という関数を利用するには、calc.sum(引数)とするのです。

calc モジュールを取り込んだ場合、calc モジュールの中で__name__という変数は”calc”になるため、if __name__ == “__main__”:以下は実行されません。

if __name__ == “__main__”:はそのモジュールが他のモジュールに取り込まれることを前提に、そこに含まれる各関数のテストを行うときによく用いられます。また **if __name__ == “__main__”:以下を見れば、各関数をどのように動かせばいいか分かるので、注釈代わりにもなり、大変便利です**。

課題 11-1：与えられたリスト中に含まれる数の平均値を求める関数 mean を含むモジュール calc2 を作成しましょう。次に、imp_test2 の中で calc2 モジュールを取り込み、ユーザから入力された複数の数の平均値を求めるプログラムを作成しましょう。

12. クラス

12.1 クラスとは？

クラスは一言で言えば、変数とそれを操作する関数をまとめたものです。例えば今「人」というものを考えます。人には必ず名前があります。従って、名前を格納する変数を考えることができます。また、生まれた年を表す変数があってもいいでしょう。そして人に対しては、必ず「歳」を定義することができます。これは現在の年から生まれた年を引く関数として実装できるでしょう。

専門用語では「人」のようなモデル化の対象となるものをオブジェクトと呼びます。

12.2 クラスの定義

それでは「人」というオブジェクトを Person というクラスとして実装してみましょう。

```
class Person:
        def __init__(self, nm):  # ここはタブで一段右へ
                self.name = nm     # ここはタブで二段右へ
```

これで「人」を抽象化できたことになります。さらに「人」の具体的な対象(例えば福澤諭吉など)を考えることができます。例えば、

```
person1 = Person(“Yukichi Fukuzawa”)
```

とすれば、person1 は”Yukichi Fukuzawa”を表すようになると考えていいでしょう。さらに、

```
person2 = Person(“Shigenobu Okuma”)
```

とすれば、person2 は”Shigenobu Okuma”を表すようになります。

person1 や person2 のように Person というクラスから具体化された対象をインスタンスと呼びます。

- `class Person` は、以下のタブで右に寄せられたブロック内でクラスの定義をすることを示します。
- `def __init__(self, nm)`はインスタンスを作成する関数です。具体的には、`Person(`引数`)`が呼ばれたときの処理を関数として定義しています。`person1 = Person(“Yukichi Fukuzawa”)`と呼び出したときには、変数 `self` には `Person` の

インスタンスである person1、nm には”Yukuchi Fukuzawa”という引数が入ります。

・ self.name は**インスタンスに属する変数** name を表します。self.name = nm で person1 というインスタンスに属する変数 name に nm がセットされます。

クラスの中で定義された関数をメソッドと呼びます。またクラスの中に定義された固有のメソッドや変数をそのクラスの属性と呼び、インスタンスが持つメソッドや変数もそのインスタンスの属性と呼びます。

これだけだとインスタンスの属性に代入するだけで面白くないので、名前を表示するメソッド show_name を作成してみましょう。

```
class Person:
        def __init__(self, nm):
                self.name = nm
        def show_name(self):
                print self.name
```

こうしておくと、

```
person1 = Person(“Yukichi Fukuzawa”)
```

で person1 に属する変数 name に”Yukuchi Fukuzawa”が入り、

```
person1.show_name()
```

で、person1 をインスタンスとしてメソッド show_name が呼び出され、”Yukuchi Fukuzawa”と出力されます。

クラスの一般的な定義方法は、

---

class クラス名：
def __init__(self, 引数 1, 引数 2、 …):
処理
def メソッド名(self, 引数 1, 引数 2、…):
処理

で、インスタンスの作成は、

インスタンス名 = クラス名(引数 1, 引数 2, …)

とします。するとメソッド__init__が呼ばれます。さらに各インスタンスに対してメソッドを呼び出すには、

インスタンス名.メソッド名(引数 1, 引数 2, …)

とします。self は個々のインスタンスを表します。person1.メソッド()とすれば、self は person1 を表し、person2.メソッド()とすれば、self は person2 を表します。メ

ソッド中の処理の中でインスタンスの属性となる変数には

self.変数名

でアクセスします。

さて人には必ず誕生日があるので、ここで誕生日という属性をさらに加えることを考えましょう。インスタンスを作成するときに、誕生日を引数として受け取るように変更します。

```
class Person:
	def __init__(self, nm, birthy):
		self.name = nm
		self.birthyear = birthy
	def show_name(self):
		print self.name
```

こうすることで

```
person1 = Person(“Yukichi Fukuzawa”, 1835)
```

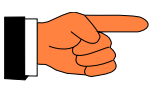
とすれば、person1 に属する変数 birthyear に 1835 が入ります。但し、今まで Person(“名前”)と呼んでいたものを全て Person(“名前”, 誕生年)に変更しなければなりません。そこで、メソッド__init__の引数は変更せずに、

```
class Person:
	def __init__(self, nm):
		self.name = nm
	def set_birthyear(self, birthy):
		self.birthyear = birthy
	def show_name(self):
		print self.name
```

とすると、

```
person1 = Person(“Yukichi Fukuzawa”)
person1.set_birthyear(1835)
```

でこれまでの__init__の呼び出し方法を変更することなく、birthyear という変数を追加できます。



クラス定義の外からでも person1.name で直接 person1 というインスタンスが持つ変数 name にアクセスできますが、変数間の整合性をメソッドで保つことができなくなるので、これはお勧めできません。クラス定義の外からインスタンスの変数にアクセスするときは、必ずメソッドを通すようにします。

課題 12-1：これまでの説明に従って「人」をモデル化したクラスを実装しましょう。次に入力された今年の西暦から、その人が今年何歳になるかを計算するメソッド get_age を作成しましょう。

12.3 組み込みメソッド

実は Python ではこれまで使ってきた数値、文字列、リスト、タプル、ディクショナリなどは全てオブジェクトなのです。従ってこれらのオブジェクトにはメソッドが存在します。例えば、文字列には大文字に変換して返すメソッド upper がありますので、

```
print “Hello”.upper()
```

とすると、”HELLO”という文字列が返ってきます。各オブジェクトにどのようなメソッドが実装されているかは、関数 dir を用いて調べることができます。またその使い方は各メソッド属性__doc__に記述されていることがあります。試しに対話モードで、

```
dir(“Hello”)
```

としてどのようなメソッドが文字列に実装されているか、見てみましょう。また、

```
print “Hello”.upper.__doc__
```

で upper の使い方を見ることができるでしょう。

課題 12-2：文字列中の特定の文字を他の文字に変換するメソッド replace の使い方を__doc__を見て調べ、replace を用いて”aattctg”を t→u に変換し、”aauucug”を出力してみましょう。

12.4 クラスの継承

ここまで人のモデル化およびクラスとしての実装を考えてきました。しかし一口に「人」と言ってもいろんなカテゴリーがあります。男性、女性、大人、子供、日本人、カナダ人、学生、従業員、…。

学生であれば、通っている学校があるでしょう。しかしこれは一般の人には必ずしも当てはまりません。逆に、人に当てはまることは学生にも当てはまります。例えば学生には名前もありますし、誕生日もあります。

そこで「学生」というクラスを作成するとき、「人」というクラスを改良して「学生」にするのではなく、「人」の属性を全て引き継いだ上で、「学生」に特有の属性を定義するのが便利です。これを**継承**といいます。

「学生」というクラスを「人」の継承として実装するためには、まず今までと同じように

```
class Person:
	def __init__(self, nm):
		self.name = nm
	def set_birthyear(self, birthy):
		self.birthyear = birthy
	def show_name(self):
		print self.name
```

を打ち込んだ後、

```
class Student(Person):
```

として、Student という学生を表すクラスが Person の属性を全て引き継ぐことを表し、続けて Student 特有のメソッドを実装します。

```
	def set_school_name(self, sc):
		self.school_name = sc
	def get_school_name(self):
		return self.school_name
```

あとは、

```
student1 = Student(“Heikichi Muto”)
```

で Person クラスのメソッドである__init__が呼び出され、Student1 の属性 name に”Heikichi Muto”という名前が書き込まれます。同様に、

```
student1.set_birthyear(1941)
```

で誕生年を記録できます。そして、Student クラス特有のメソッドである

set_school_name および get_school_name を

```
student1.set_school_name(“Keio”)
print student1.get_school_name()
```

のように呼び出すことができます。

クラスの継承は以前に作成したクラスの再利用を促進する重要なメカニズムです。以前に作成したクラスを少し変更して使用したい場合、継承を行って新たなクラスを作成し、変更したい部分に関してのみ新たに変数およびメソッドを追加すればいいことになります。以前に作成したクラスの動作概要を知っていれば、そのクラスのメソッドの中味のプログラムを見て理解しなくても、継承によってクラスを改良できるのです。

課題 12-2：クラス Person を継承してクラス BaseBall_Player を作りましょう。BaseBall_Player 特有の変数としてプレーするポジションを表す position を作成し、また同時にそのポジションを返すメソッド get_position を実装しましょう。

13. 例外処理

プログラム実行中に「存在すると期待されていたファイルがない」、「ユーザ入力が異常」など、予想外の事態が発生し、以降の処理を打ち切りたいときがあります。そんなとき、例外処理が活躍します。例外処理とは簡単に言うと、完了が期待される処理を打ち切って行う処理のことです。以下のスクリプトを打ち込んで実行してみましょう。

```
#!/usr/bin/env python

try:
    num_of_input = input("Input number of values: ")
    i = 0
    vals = []
    while i < num_of_input:
        val = input("Input value #" + `i` + ": ")
        if val >= 100:
            raise "TooBig"
        vals.append(val)
        i = i + 1
    print "The following values are user inputs:"
    for val in vals:
        print val

except "TooBig":
    print "The value is too big!!"
```

このスクリプトはユーザから数値の入力を受け付け、それらを後から全て表示するだけのものです。但し、入力された数値が 100 以上だった場合は処理を打ち切り、”The value is too big”と表示して終了します。

重要なポイントは以下の通りです。

- try 文によって例外を捕捉するブロックを定義します。
- raise 文によって例外処理に移行することを示します。
- except 文によって例外処理を定義します。

例外処理のおおまかな形式は、

```
try:
        処理
        … raise “eName” …
        処理
except “eName”:
        “eName”という例外が発生したときの処理
```

です。これで try ブロックの中で状況に応じて raise eName によって eName という例外を発生させ、処理を一気に except eName ブロックに移行することができます。

課題 13-1：ユーザが入力した数の合計を求めるプログラムを書きましょう。ユーザは最初に数の個数を入力し、次に合計を求めたい数を入力するようにします。但し、入力途中で合計が 100 を超えた場合、処理を打ち切って”Total value exceeded 100.”と表示します。