Python+Numpy+Matplotlib

講習会資料

2014/6/20

Python言語とは

- 1990年頃Guido van Rossumによって作られる。
- 現在、広汎な分野で利用されている平易かつ強力なスクリプト言語。
- インタプリ型言語(コンパイル不要)

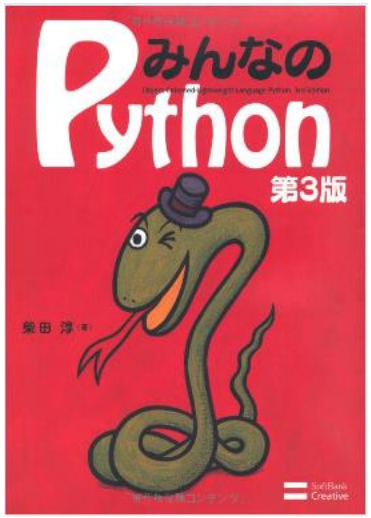
[コメント]

MATLAB既修者が比較的多いと思われるため

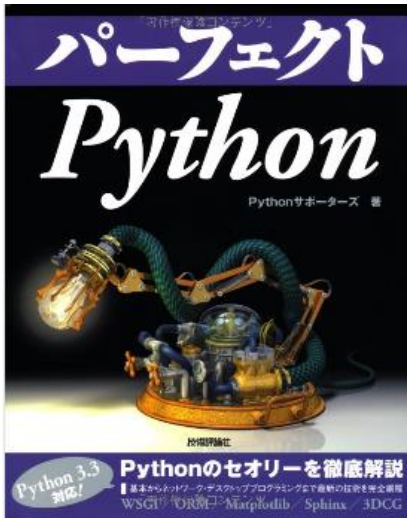
本講習会では随時MATLABと比較しながら説明する。

□

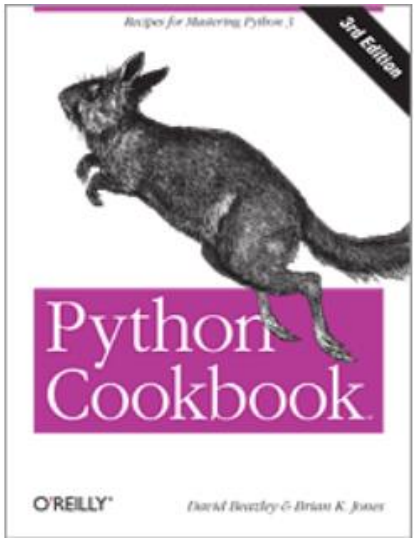
参考書



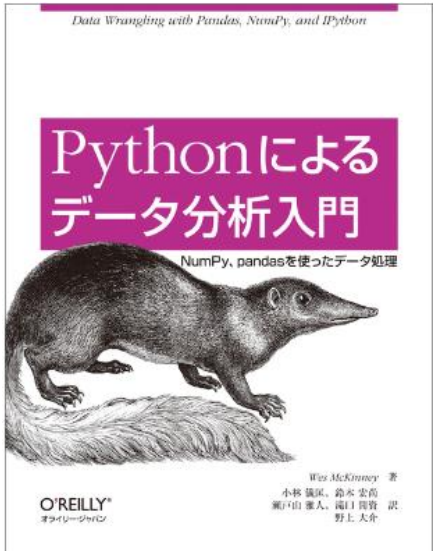
第3版
ソフトバンク
クリエイティブ
2012
(Pyhton3対応)



技術評論社
2013
(Python3対応)

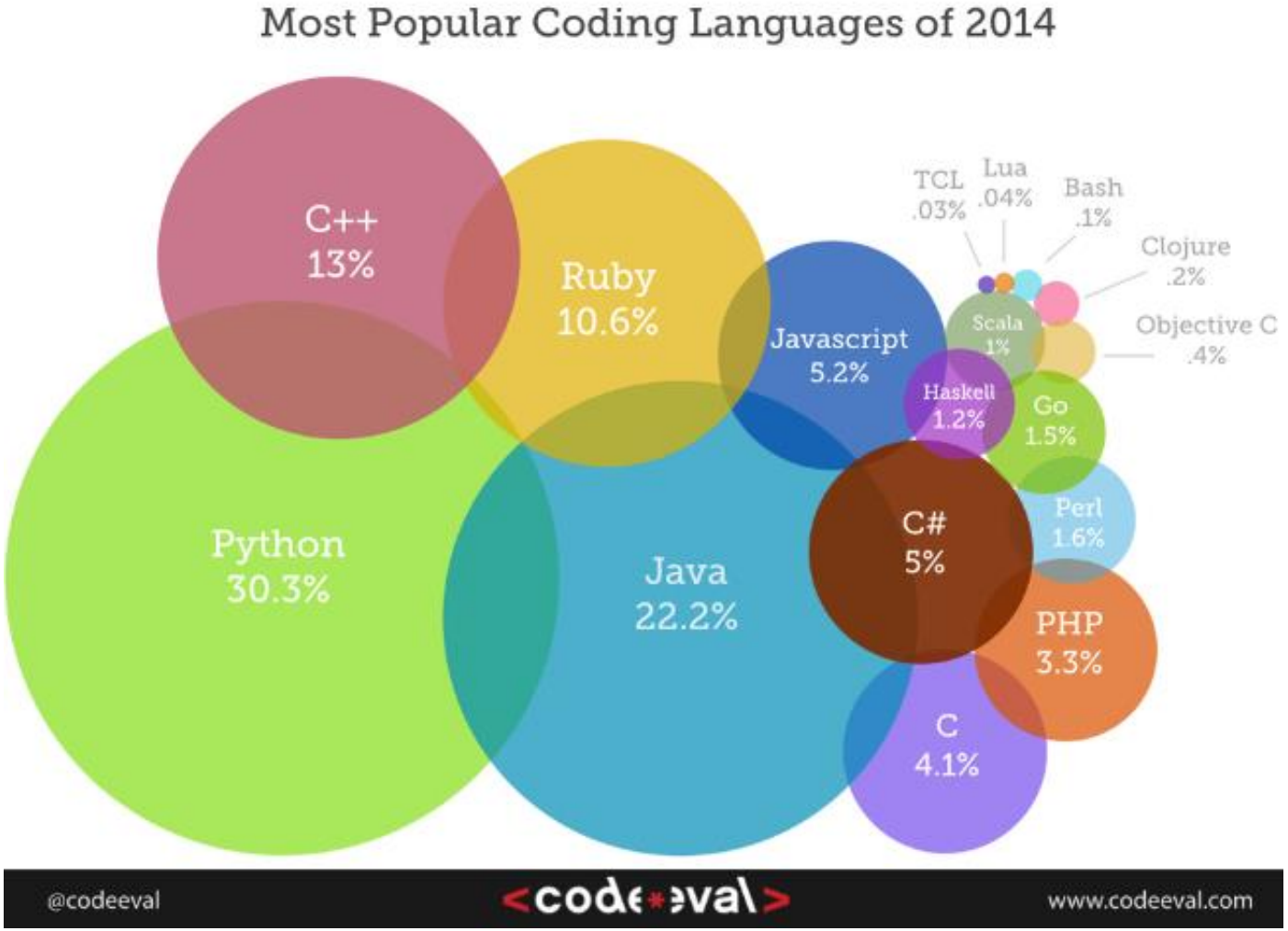


3rd Edition
O'REILLLY
2013
(Python3対応)



オライリー
ジャパン
2013
(ipython,
numpy,
matplotlib,
pandas)

PythonのPopularity



Pythonの特徴

- 動的script言語
- compile不要。変数の型宣言、関数のprototype宣言は不要。当然header fileも不要。
- 学習が容易。
- 可読性が高い。
- 強制indent rule　(最初は違和感があるかもしれないが、これは大発明である。）
- 予約語や面倒で覚えにくいruleが少ない。
- 自然で常識的な文法や記号
- selfは省略不可(C++やJavaではthisが省略可能なので可読性が大きく劣化する）
- Debugが容易。
- 豊富な標準ライブラリ、サードパーティライブラリ、世界中に大量のコード資源

Python2とpython3

- Python2.xの系列とpython3.xの系列がある。
- Python3では「よりシンプルで一貫性を持たせる」ために、いくつかの点でPython 2との後方互換性が犠牲にされた。
- 本講習会では、Python3で説明する。
（Python3.0が2008年にリリースされてから、すでに５年以上経過し、主要LibraryがPytrhon3に対応するようになっている。Python2はすでにメンテナンスモードになっている。）

Python関係のinstallation(Win)

windows7(64bit)の例、(本講習会で使用、言及するpackage)

(1)python_3.4.1.msiを取得して実行
(2)numpy-1.8.1-win32-superpack-python3.4.exeを取得して実行
(3)sympy-0.7.5-win32.exeを取得して実行
(4)scipy-0.14.0.win32-py3.4.exeを取得して実行
(5)matplotlib-1.3.1.win32-py3.4.exeを取得して実行
(6)環境変数設定(PATHにC:¥Python34¥Scriptsを、PATHEXTにPYC;PYを含める。)
(7)pip3 install python-dateutil　pyparsingを実行
(8)pip3 install ipythonを実行
ipythonをQt modeでも起動可能にしたい場合にはさらに以下を実行
(8-a)PyQt4-4.10.4-gpl-Py3.4-Qt4.8.6-x32.exeを取得して実行
(8-b)pip3 install pygments pyzmqを実行
[コメント]
scipy, matplotlib, pyqtについては現時点でPython3.4対応のWindows用正規版installerが公開されてなかったので、Unofficial Windows Binaries for Python Extension Packages　から取得した。
□

Python関係のinstallation(Ubuntu)

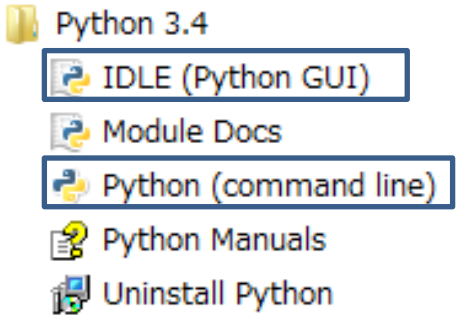
ubuntu(14.04 desktop 64bit)の例、(本講習会で使用、言及するpackage)

(1)sudo apt-get install python3-pip
(2)sudo apt-get install python3-numpy
(3)sudo apt-get install python3-matplotlib
(4)sudo apt-get install python3-scipy
(5)sudo pip3 install sympy
(6)sudo pip3 install ipython

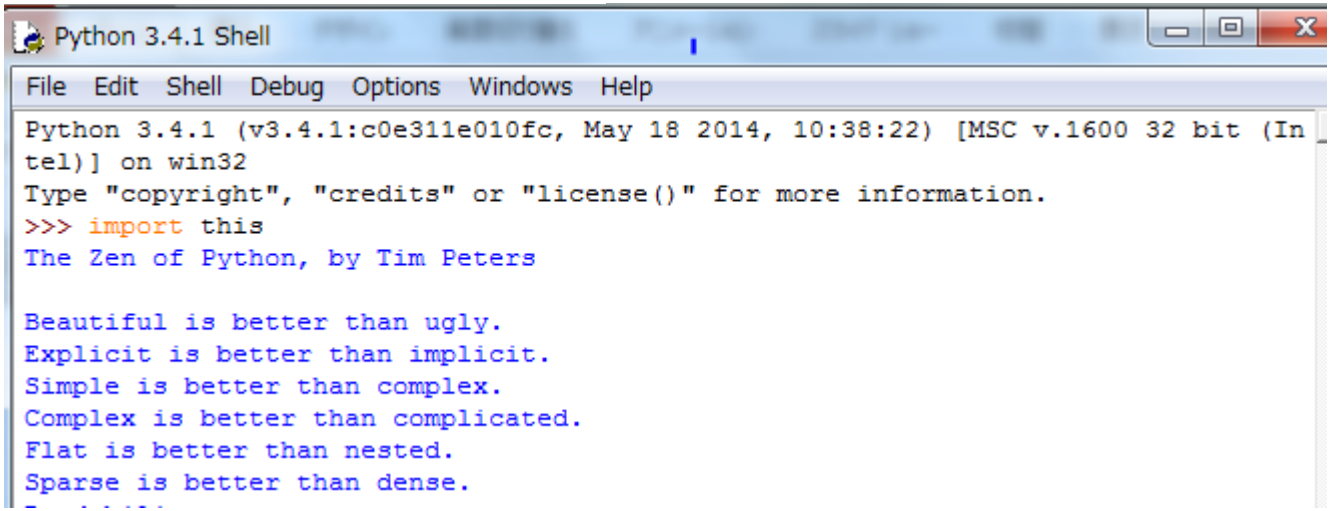
[コメント]
ubuntu(14.04)には最初からpython(2.7.6)とpython(3.4.0)が共存でinstallされているので、上の手順ではpython(3.4.0)の方にinstallされるようにしている。
(すなわち、python3-*, pip3というように3を付けること。）

aptはubuntu(debian系)のpackage管理システムである。
pipはpythonのpackage管理システムである。
□

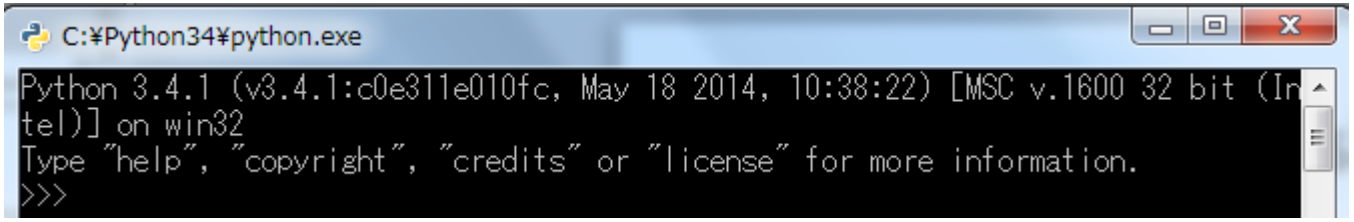
Python shell



Windowsにpython3をインストールすると2種類のPython shellが用意される。



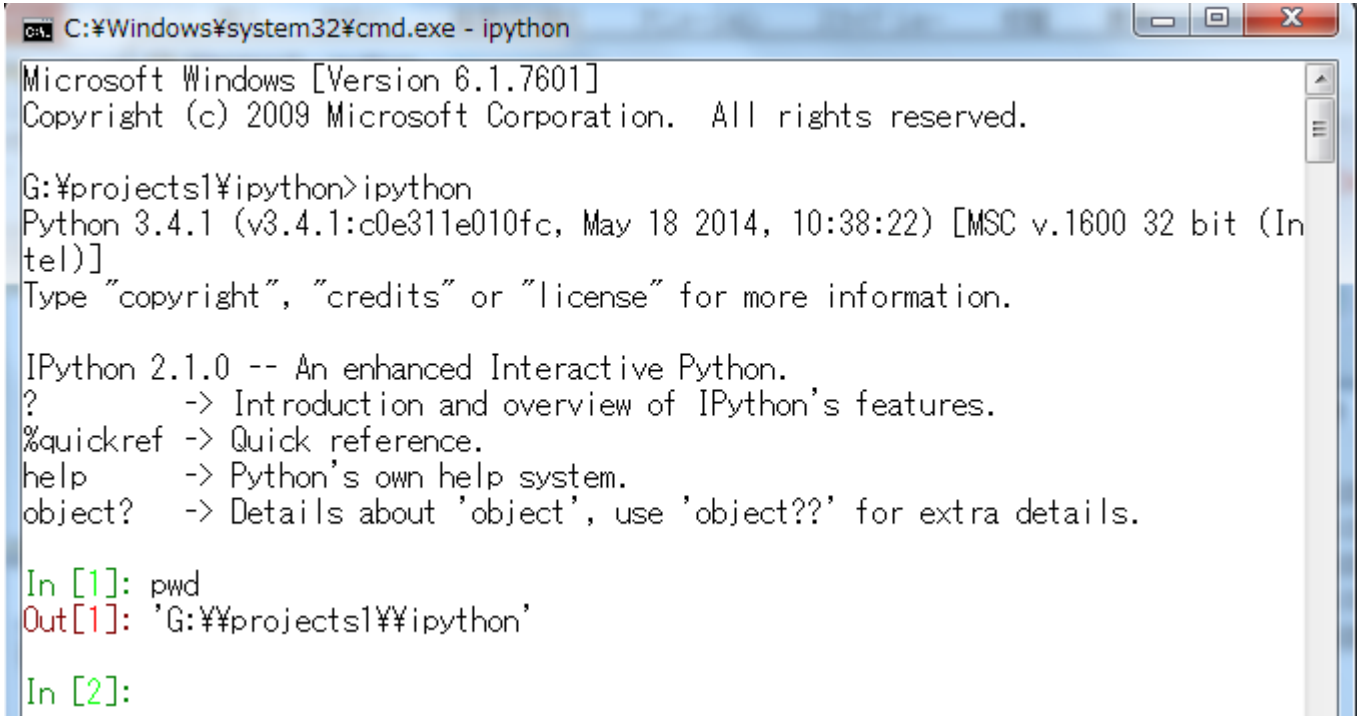
IDLE
(Python GUI)



Python
Command line

ipython (1)

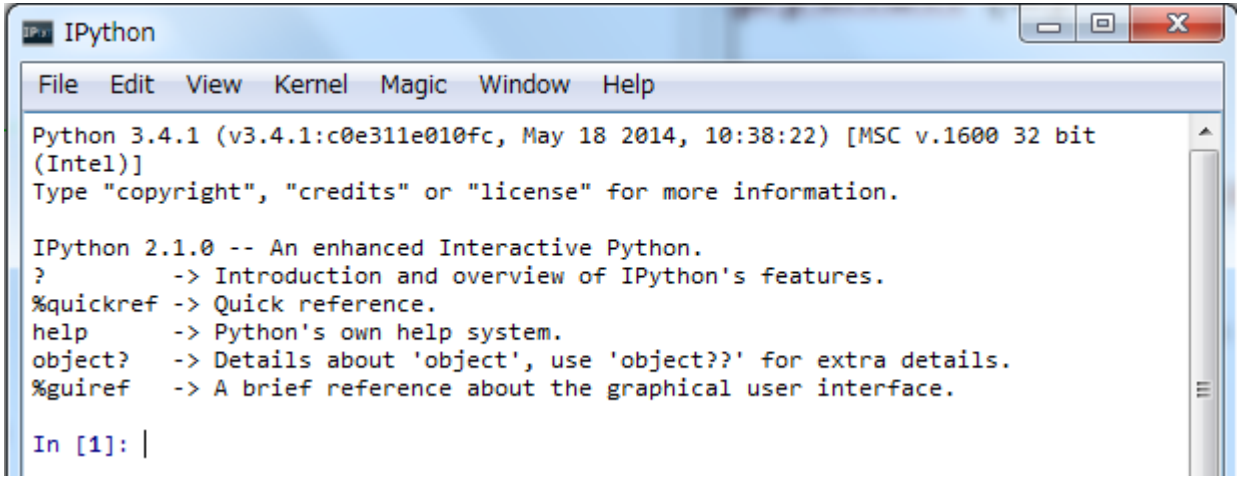
本講習会ではIDLEよりもさらに高機能な
Python shellであるipythonを使用する。



Windowsのコマンドプロムプトから'ipython'を起動する。

ipython (2)

ipythonを**Qt modeで起動するのも一つの選択**である。(若干動作が異なるようだ)
例えば、Windowsのipythonは前ページのdefault modeで起動させると、文字列のcopy & pasteが標準的やり方で出来ない。(copyの場合は、まず右クリックメニューから矩形範囲指定モードに切り替え、左ドラッグで矩形範囲指定してから右クリックでcopyする。pasteは右クリックメニューからpasteコマンドを実行する。)
これに対して、Qt modeで起動すると、標準的やり方でcopy & pasteができる。



Windowsのコマンドプロムプトから'ipython qtconsole'を入力する。

ipythonの便利な機能 (1)

(1)オートコンプリート機能(TABで関数名やファイル名を補完してくれる)
(ちなみに、関数名のオートコンプリートはPython3の標準shellでも可能)

(2)ヘルプ機能

In [**1**]: import os #os moduleをimport しておく。
In [**2**]: **help(os.path.join)** #関数os.path.joinを調べる。(これは標準shellでも可能)
Help on function join in module ntpath:
join(path, *paths)
# Join two (or more) paths.

In [**3**]: **os.path.join?**
Type: function
String form: <function join at 0x021734B0>
File: c:¥python34¥lib¥ntpath.py
Definition: os.path.join(path, *paths)
Docstring: <no docstring>

さらに、**os.path.join??** とするとコードの中身まで表示してくれる。(Viewerが起動)

ipythonの便利な機能 (2)

(3)!に続けて入力すればOS自身のshellコマンドを実行可能。

In [**8**]: !ren a5.py a6.py #WindowsのCmdのren(名前変更)コマンドを実行

(4) 多くの便利な組み込みコマンドが用意してある。

In [**11**]: lsmagic #magic コマンド一覧表示

Out[**11**]:

Available line magics:

%alias %alias_magic %autocall %autoindent %automagic **%bookmark** **%cd** %cls %colors %config %copy %cpaste %ddir %debug %dhist **%dirs** %doctest_mode %echo **%ed** %edit %env %gui **%hist** %history %install_default_config %install_ext %install_profiles %killbgscripts %ldir %load %load_ext %loadpy %logoff %logon %logstart %logstate %logstop **%ls** **%lsmagic** %macro %magic %matplotlib %mkdir %notebook %page **%paste** %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 **%popd** %pprint %precision %profile %prun %psearch %psource **%pushd** **%pwd** **%pycat** %pylab %quickref %recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir **%run** %save %sc %store %sx %system %tb %time %timeit %unalias %unload_ext %who %who_ls %whos %xdel %xmode

以下省略

ipythonの便利な機能 (3)

(5) コマンドhistory

上下矢印キーで、コマンドhistory buffer上で移動できる。
histで全コマンドhistoryを表示可能
exec(In[番号]),eval(In[番号]) などで再実行、再評価が可能。(_i番号でもOK)

(6) directory関係のコマンド

pwd: 現在のdirectoryを表示
ls: 現在のdirectoryに存在するfile/directoryを表示
cd DIR: DIRに移動する。(pushd ,popdも利用可能。dirsでこのstackを確認可能)
cd -: 直前のdirecotryに移動する。
bookmark MARK : 現在のdirectory(絶対パス)をMARKと別名定義する。
cd –b MARK: 別名定義されたdirectory(MARK)に移動する。(-bは省略可能)

(7)スクリプトファイル編集・確認

ed SCRIPT : スクリプトファイル(SCRIPT)を編集する。(拡張子pyは省略可能)
default editorはWindowsではnotepad, Linuxではviとなる。
別のeditorを使いたい場合には環境変数EDITORに登録しておく。
(ただし、Qt modeで起動すると、環境変数EDITORが効かないようだ。)
編集せずに、確認だけなら、pycatを使っても良い。

ipythonの便利な機能 (4)

(8) スクリプトファイルの実行

run SCRIPT: スクリプトファイル(SCRIPT)を実行する。(拡張子pyは省略可能)

run –d SCRIPT: debugger(PDB)を介してSCRIPTを実行する。

簡単なpython program例

```
In [1]: ed a1.py    #スクリプトファイルを編集

#-*- coding:shift_jis -*-
print('べき乗のテスト')
while 1:
    imax=input('imax? ')
    try:
        imax=int(imax)  #文字列を整数に変換
        break  #while loopから抜ける
    except:
        print('整数を入力してください')
for i in range(imax):
    print(i,i**2,i**3)
```

```
In [2]: run a1.py   #スクリプトファイルを実行

べき乗のテスト

imax? 5
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

- スクリプトファイルの文字コードがasciiまたはutf-8以外の場合は先頭行に文字コードを記載すること。
- while, for, try, exceptなどのcode blockはindentを行うこと。
- indexを行う直前行の最後にはコロンをつけること。
- #から行末まではコメントである。

Pythonの主要基本型

型(class)	説明	分類	instance例
object	object型	object型	object()
bool	論理型	論理型	True, False
int	整数型	数値型	26, 0b11010,0o32,0x1a
float	浮動小数点型	数値型	3.1415, 1.5e-3, 1/2
complex	複素数型	数値型	1j, 1.1+2.3j
tuple	タプル型	シーケンス型	(), (1,), (True,False)
list	リスト型	シーケンス型	[],[1], [True,False]
str	文字列型	シーケンス型	'man',"man",'人'," 人"
bytes	バイト列型	シーケンス型	b'man', b'¥xe4¥xba¥xba'
bytearray	バイト配列型	シーケンス型	bytearray(b'man')
range	範囲型	イタレータ型	range(1,10,2)
dict	辞書型	マッピング型	{},{'y':2014,'m':12,'d':25},dict()
set	集合型	集合型	{1,3,5,7,9},set()
function	関数型	関数型	lambda x:x**2

型(class)の調べ方

```
In [31]: type('man')
Out[31]: str

In [32]: 'man'.__class__
Out[32]: str

In [33]: isinstance('man',str)
Out[33]: True

In [34]: isinstance('man',bytes)
Out[34]: False

In [35]: isinstance('man',object)
Out[35]: True

in[36]: issubclass(str,object)
Out[36]: True
```

instance　aの型を調べるには、

type(a)を調べるか、

または

a.__class__を調べればよい。

（3.__class__など、リテラル（即値）に対しては__class__が機能しない場合があるが、type()は常に機能する。）

isinstance(a,A)は、

type(a)がAのsubclass

である場合でもTrueとなる。

(すべてのclassはobject classのsubclassである！）

型変換(cast)の方法

```
In [44]: float(1) #int to float
Out[44]: 1.0
In [46]: int(-1.9) #float to int
Out[46]: -1
In [47]: complex(0)  #int to complex
Out[47]: 0j

In [48]: list((1,2,3))  #tuple to list
Out[48]: [1, 2, 3]
In [49]: tuple([1,2,3]) #list to tuple
Out[49]: (1, 2, 3)
In [50]: list(range(5))  #range to list
Out[50]: [0, 1, 2, 3, 4]
In [51]: list('abc')  #str to list
Out[51]: ['a', 'b', 'c']
```

多くの場合、第一引数に変換元変数を入れて、変換先変数を生成させればよい。

[コメント]

Python3では、range(...)はlistでなくiteratorを戻すようになった。

従って、Python2のようにlistを得るにはさらにlist(...)でcastする必要がある。

□

bool

```
In [10]: not True, True and True, True or False
Out[10]: (False, True, True)
#bool型(論理型)の真、偽はTrue,Falseで表現される。
#r論理否定、論理積、論理和に対する演算子は'not', 'and','or'である。

In [11]: bool(0), bool(0.1), bool(()), bool([]), bool({}), bool({0})
#全てのobjectがboolにcast可能である。
#この例では、整数、実数、空tuple, 空list, 空dict, setをcastしてみた。
Out[11]: (False, True, False, False, True)
#条件文など、さまざまな状況で暗黙の論理型への変換が行われる。

In [12]: 3 and 6, 3 or 6, 1 and 0     #整数の論理積と論理和
Out[12]: (6, 3, 0)  #評価手続きにおける最後の整数が戻される。

In [13]: 3&6, 3|6  #これは整数のbit毎の論理積、論理和の意味である。
Out[13]: (2, 7)
```

[参考] MATLABでの論理型

```
--> a=(2==2);b=(2==3);
--> disp(a);disp(b);     %MATLABでは真、偽は1,0で表現される。
 1
 0
--> whos a b
  Variable Name       Type   Flags          Size       Bytes
          a   logical                [1x1]         1
          b   logical                [1x1]         1
%MATLABでは行列を基本量と考えるので、scalar変数を設定したつもりでも
%1x1行列と解釈されているのが分かる。
```

	論理否定	論理積	論理和	bit毎論理積	bit毎論理和
python	not True -->False	3 and 6 -->6	3 or 6 -->3	3 & 6 -->2	3|6 -->7
MATLAB	~1 -->0	3 && 6 -->1	3 || 6 -->1	bitand(3,6) -->2	bitor(3,6) -->7

pythonとMATLABの論理演算の比較

int, float,complex

型(class)	説明	分類	instance例
int	整数型	数値型	26, 0b11010,0o32,0x1a
float	浮動小数点型	数値型	3.1415, 1.5e-3, 1/2
complex	複素数型	数値型	1j, 1.1+2.3j

In [**26**]: 10**30 #Python3ではint型に上限はない
Out[**26**]: 1000000000000000000000000000000

In [**27**]: 10/5, 10//5 #Python3では10/5はfloat, 10//5はintになる。
Out[**27**]: (2.0, 2)

In [**28**]: 10//3, 10%3 #商と余りを求める。
Out[**28**]: (3, 1)

In [**29**]: divmod(10,3) #商と余りはdivmodを使っても求まる。
Out[**29**]: (3, 1)

[参考] MATLABでの数値型

```
--> a=1;b=1+i;     %MATLABでは虚数単位はjでなくiを使用する。
--> whos a b       %NATLABではwho またはたwhosで変数情報が見れる。
  Variable Name      Type   Flags          Size      Bytes
          a    double                [1x1]         8
          b    double                [1x1]         16
%MATLABでは行列を基本量と考えるので、scalar変数を設定したつもりでも
%1x1行列と解釈されているのが分かる。
%整数を設定したつもりでもdoubleと解釈される。
%複素数であっても型はdoubleと表示される。
%(Sizeが16であることから複素数であることが分かる。)
```

	和	差	積	商	余り	べき乗
python	a+b	a-b	a*b	a/b, a//b	a%b	a**b
MATLAB	a+b	a-b	a*b	a/b, b¥a	rem(a,b)	a^b
	加算		等号	不等号	小なり	大なり
python	a+=b		==	!=	<, <=	>, >=
MATLAB	なし		==	~=	<, <=	>, >=

主な算術演算子、比較演算子 (MATLABでは要素毎演算では.*のように.を付ける)

listとtuple(1)

型(class)	説明	分類	instance例
tuple	タプル型	シーケンス型	(), (1,), (True,False)
list	リスト型	シーケンス型	[],[1], [True,False]

In [**41**]: a=[10,20,30,40,50] #生成

In [**42**]: a[0],a[-1],a[-2],a[1:4] #GET
Out[**42**]: (10, 50, 40, [20, 30, 40])

In [**43**]: a[0],a[-1]=11,51 #SET

In [**44**]: a
Out[**44**]: [11, 20, 30, 40, 51]

In [**45**]: a[1:4]=[1,2]

In [**46**]: a
Out[**46**]: [11, 1, 2, 51]

In [**47**]: a=(10,20,30,40,50) #生成

In [**48**]: a[0],a[-1],a[-2],a[1:4] #GET
Out[**48**]: (10, 50, 40, (20, 30, 40))

In [**49**]: a[0]=11 #SET
TypeError: 'tuple' object does not support item assignment

listは要素、部分列の変更が可能であるが、tupleでは禁止されている。
(list: mutable, tuple: immutable)

listとtuple(2)

```
In [52]: a,b=[1,2],[True,'Japan',[None,'dog']]
```
#listには何でも入れることができる。

```
In [53]: a+b  #listの結合
Out[53]: [1, 2, True, 'Japan', [None, 'dog']]

In [55]: a*3     #listの3回繰り返し
Out[55]: [1, 2, 1, 2, 1, 2]

In [58]: b.index('apan')   #indexを求める
Out[58]: 1

In [59]: a.append(10)  #追加
In [60]: a
Out[60]: [10, 20, 30, 40, 50, 10]
```

```
In [63]: a,b=(1,2),(True,'Japan',(None,'dog'))
```
#tupleには何でも入れることができる。

```
In [64]: a+b   #2つのtupleの結合
Out[64]: (1, 2, True, 'Japan', (None, 'dog'))

In [65]: a*3  #tupleの3回繰り返し
Out[65]: (1, 2, 1, 2, 1, 2)

In [66]: b.index('Japan')  #indexを求める。
Out[66]: 1

In [67]: a.append(10)  #追加
AttributeError: 'tuple' object has no attribute
'append'
```
tupleには追加ができない。
もちろん、a+(10,)として別のtuple objectを生成することは可能。

listとtuple(3)

In [**78**]: a=1,2 #tupleの丸括弧は誤解釈されない限り省略可能
In [**79**]: a,type(a)
Out[**79**]: ((1, 2), tuple)

In [**81**]: b=1, #要素数1個のtupleは誤解釈されないように最後のカンマが必要。
In [**82**]: b,type(b)
Out[**82**]: ((1,), tuple)

In [**89**]: c,d=(1,2,),[1,2,] #一般のtupleやlistでも最後のカンマをつけても良い。
#最後の要素を特別扱いしないので時として有用な仕様である。(code自動生成など)
In [**90**]: c,d
Out[**90**]: ((1, 2), [1, 2])

そもそもlistに機能制限を付けただけに思われるtupleの存在意義は何か？
(1)tupleはdict(辞書)のkeyになれるが、listはそうではない。
(2)tupleはmutableでない分、listより高速に処理ができる。
(3)tupleは元々、複数のobjectを「単純に」1個にまとめるという意味がある。
（丸括弧が省略できるので、あたかも複数個の関数戻り値などが可能に見せられる）

str, bytes, bytearray(1)

型(class)	説明	分類	instance例
str	文字列型	シーケンス型	'man',"man",'人',"人"
bytes	バイト列型	シーケンス型	b'man', b'¥xe4¥xba¥xba'
bytearray	バイト配列型	シーケンス型	bytearray(b'man')

python3ではstr(文字列)は常にUnicodeを意味する。これに対して、bytes, bytearrayは単純なbyte列の意味であり、それ以上でも以下でもない。

bytesとbytearrayの違いは、tupleとlistの違いと同様である。
bytes,tupleはimmutable(要素、部分列の変更不能)であるが、dictのkeyになりえる。

```
In [56]: a,b='ABCDE','FG'

In [57]: a[0],a[-1],a[-2],a[1:4]   #文字列のindexing方法はlistやtupleと同じである。
Out[57]: ('A', 'E', 'D', 'BCD')

In [58]: a+b,b*3   #文字列の連結や繰り返しもlistやtupleと同じである。
Out[58]: ('ABCDEFG', 'FGFGFG')
```

str, bytes, bytearray(2)

```
In [105]: a,b,c,d='dog',"dog",'''dog''',"""dog"""
In [106]: a==b==c==d
Out[106]: True
#Pythonではsingle quoteとdouble quoteは全く同じ意味である。
#途中に改行が含まれない限り、triple quoteとも同じ意味になる。
In [107]: a,b='this ¥'dog¥'', "this 'dog'“
#single quoteとdouble quote quoteを組合わせると、Escape記号を回避可能なので便利。
In [108]: a,b,a==b
Out[108]: ("this 'dog'", "this 'dog'", True)
#triple quoteの中には以下のように生の改行を含めることができる。
In [118]: s='''¥
   .....: #include <stdio.h>
   .....: int main(){
   .....:     printf("hello world¥¥n");
   .....:     return 0;
   .....: }
   .....: '''
In [119]: open('hello.c','w').write(s)  #上のC言語のコードをファイルに書き込む。
```

str, bytes, bytearray(3)

文字列formattingの方法には2種類があり、どちらも有用である。

```
In [132]: folder, name, num= 'test','data',12

In [135]: filename='%s/%s_%04d.csv' % (folder, name, num)   #古い方法 (%を使用)
In [136]: filename  #古いといっても、便利過ぎるので将来消えることは在り得ない。
Out[136]: 'test/data_0012.csv'

In [137]: filename='{folder:}/{name:}_{num:04d}.csv' ¥
   .....: format(folder=folder,name=name,num=num)   #新しい方法 (中括弧を使用)
In [138: filename
Out[138]: 'test/data_0012.csv'
```

以下の空白、TAB, 改行などを削除するstirp(), rstrip(),lstrip(), 文字列を分解、合成するsplit(), join()も非常に良く使用する関数である。

```
In [165]: a=' abc d e¥n'
In [166]: a.strip(), a.lstrip(), a.rstrip()
Out[166]: ('abc d e', 'abc d e¥n', ' abc d e')
In [169]: '@'.join(a.strip().split())
Out[169]: 'abc@d@e'
```

str, bytes, bytearray(4)

```
In [59]: c='日本'   #Python3ではstr(文字列)は常にUnicodeを意味する。
#従って、一度strに変換されたら、文字コードのことは一切考えなくてよい。

In [60]: b1,b2,b3 = c.encode('utf-8'), c.encode('utf-16'), c.encode('shift_jis')
In [61]: b1,b2,b3  #strを文字コードを指定してbytesに変換する。(encodeする)
#実際には、'utf-8'がencode/decodeのdefaultであるため、'utf-8'は省略してよい。
Out[61]: (b'¥xe6¥x97¥xa5¥xe6¥x9c¥xac', b'¥xff¥xfe¥xe5e,g', b'¥x93¥xfa¥x96{')
#各byteは原則として、16進表示されるが、ascii文字についてはそのまま表示される。

In [62]: c1,c2,c3 = b1.decode('utf-8'),b2.decode('utf-16'),b3.decode('shift_jis')
In [63]: c1,c2,c3   #bytesを文字コードを指定して、strに変換する。(decodeする。)
Out[63]: ('日本', '日本', '日本')  #すべて同じstrに戻ったことが確認できる。

#ファイルとはbyte列であるから、ファイルへの読み書きの考え方も上と同じである。
In [72]: c='日本'   #これはstr(すなわちUnicode)である。
In [73]: open('c1.txt','w',encoding='shift_jis').write(c)   #shift_jisでencodeして書き込む。
In [74]: d=open('c1.txt',encoding='shift_jis').read()  #shift_jisでdecodeして読みだす。
In [75]: d
Out[75]: '日本'   #これはstr(すなわちUnicode)である。
```

[参考]MATLABでの文字列型

```
--> a='python';b=['ruby';'perl'];c=char('python','ruby');
--> disp(c);
python
ruby
--> whos a b c
  Variable Name      Type   Flags          Size      Bytes
          a      char                [1x6]        12
          b      char                [2x4]        16
          c      char                [2x6]        24
```

%MATLABではN文字の文字列は、1行N列のchar型行列と解釈される。
%M個の文字列からなるリストは、M行のchar型行列と解釈される。
%上のb=['ruby';'perl']のような設定方法では、文字列の長さが揃っていないと
%Errorがでる。char()関数を使うと、自動的に空白文字を付加して揃えてくれる。

```
-->c2=c(2,:);    %行列の第2行を取得する。
--> disp(size(c2));
 1 6    %1行6列
--> disp(size(deblank(c2)));
 1 4   %1行4列
```

%逆に付加された空白文字を削除するには、このようにdeblank()関数を利用する。

range, map, filter

```
In [155]: list(range(5))   #0以上5未満の整数
Out[155]: [0, 1, 2, 3, 4]

In [156]: list(range(2,10,2))  #1よりstep=2ずつ増やしていき、10未満の整数
Out[156]: [2, 4, 6, 8]

In [157]: list(range(9,0,-3))  #9より、step=3ずつ減らしていき、0より大きい整数
Out[157]: [9, 6, 3]

In [160]: list(map(int, '1 4 5 9'.split()))  #str型['1','4','5','9']を全てint型に変換する。
Out[160]: [1, 4, 5, 9]

In [161]: list(map(lambda x:x**2, range(5)))  #[0,1,2,3,4]の全てについて2乗を求める。
Out[161]: [0, 1, 4, 9, 16]

In [164]: list(filter(lambda x:x%3==0, range(10)))   #3で割り切れる10未満の整数
Out[164]: [0, 3, 6, 9]
```

#組み込み関数range(), map(), filter()はiterator型を戻す。
#上の例では具体的な結果を表示させたいので、これらをlistにcastしている。

listの内包表記

mapやfilterと似た処理ができる「listの内包表記」も非常によく利用される。
前スライドの例を「listの内包表記」を使って書き直すと、以下のようになる。

```
In [23]: [int(x) for x in '1 4 5 9'.split()]   #str型['1','4','5','9']を全てint型に変換する。
Out[23]: [1, 4, 5, 9] #結果はlist
In [24]: [x**2 for x in range(5)]  #[0,1,2,3,4]の全てについて2乗を求める。
Out[24]: [0, 1, 4, 9, 16]
In [25]: [x for x in range(10) if x%3==0]  #3で割り切れる10未満の整数
Out[25]: [0, 3, 6, 9]

#以下のように多重loopであっても構わない。
In [26]: [(i,j,k) for i in range(20) for j in range(i) for k in range(j)  ¥
   ...: if i**2==j**2+k**2]   #20以下のピタゴラス数
Out[26]: [(5, 4, 3), (10, 8, 6), (13, 12, 5), (15, 12, 9), (17, 15, 8)]
#やや邪道であるが、for loopでの実行を1行で書きたい場合にも使うことがある。
In [41]: [print(',' if i else '',i**2,end='') for i in range(5)]
#Y if X else Zは3項演算子   (C言語のX ? Y: Zと同じ意味)
Out[41]: [None, None, None, None, None]  #ここでの評価値(list)は何の意味もない。
 0, 1, 4, 9, 16
```

[参考]MATLABでのrange, map, filter

[range相当]

```
--> disp(0:6)    %pythonのrange(0,7)と等価 (pythonでは7未満の意味)
 0 1 2 3 4 5 6
--> disp(0:2:6)  %pythonのrange(0,7,2)と等価
 0 2 4 6
--> disp(6:-2:0)  %pythonのrange(6,-1,-2)と等価 (range(6,0,-2)とすると0が含まれない)
 6 4 2 0
```

[map相当] (arrayfunc)

```
--> a=arrayfun(@(x) x^2+1, 0:3); disp(a);   #@はpythonのlambdaに相当する。
%pythonのa=map(lambda x: x**2+1, range(4)))と等価
 1 2  5 10
```

[filter相当] (MATLABにはfilterと完全等価なものは存在しないが、findなどで代用できる。)

```
--> a=[1,3,4,2,5]; disp(a);
 1 3 4 2 5
--> b=a(find(rem(a,2)==0)); disp(b);  #偶数のみ。find(...)とは...が真になるindex listを求める。
%pythonのb=filter(lambda x:x%2==0, a)と等価。
 4 2
```

dict(1)

list型では、indexは0から始まる整数であるが、dict型(辞書型)では、このindexに相当するkeyとして整数、文字列、tupleなど自由に使用できる。(listはkeyとして使用できない)

```
In [30]: exts={'py':'Python','rb':'Ruby','pl':'Perl'}
In [31]: for key in exts:
   ....:     value=exts[key]   #keyからvalueを取り出すにはlistやtupleと同様に大括弧を使用
   ....:     print(key,value)
py Python
pl Perl
rb Ruby
In [33]: for key,value in exts.items():   #items()を使えば、keyとvalueを同時に取り出せる。
   ....:     print(key,value)
py Python
pl Perl
rb Ruby

In [35]: 'py' in exts, 'c' in exts  #keyがdictに含まれるかどうかはinを使用すればよい。
Out[35]: (True, False)
```

dict(2)

```
In [39]: files=['foo.py','bar.pl','hoge.rb'] #言語名を調査したいファイル名のlistを与える。

In [40]: [(f,exts[f.split('.')[-1]]) for f in files]  #この書き方はlistの内包表現である。
Out[40]: [('foo.py', 'Python'), ('bar.pl', 'Perl'), ('hoge.rb', 'Ruby')]

In [42]: exts2={'c':'C','f':'Fortran'} #別のdictを定義する。
In [43]: exts.update(exts2) #updateメソッドで2つのdictを合成することができる。
In [44]: exts
Out[44]: {'py': 'Python', 'pl': 'Perl', 'c': 'C', 'rb': 'Ruby', 'f': 'Fortran'}
In [45]: del exts['pl'] #delでdictの要素を削除可能
In [46]: exts
Out[46]: {'py': 'Python', 'c': 'C', 'rb': 'Ruby', 'f': 'Fortran'}
```

dictの生成方法は中括弧を使う方法以外にもいくつかある。(以下は全て同じ意味。)

```
In [47]: exts=dict(py='Python',rb='Ruby',pl='Perl')  #keyが文字列の場合のみ利用可能。
In [49]: exts=dict([('py','Python'),('rb','Ruby'),('pl','Perl')])
In [51]: exts=dict(zip(['py','rb','pl'],['Python','Ruby','Perl']))  #さらにzipを併用
ちなみに、以下のようにzipは引数に3個以上のlistがある場合も許容される。
In [55]: list(zip([1,2,3],[4,5,6],[7,8,9]))  #zip自体はiteratorを戻すのでlistにcastする。
Out[55]: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

set

```
In [62]: langs={'Python','Ruby','Perl'} #set(集合)は要素を中括弧で囲む
In [63]: 'Python' in langs, 'C' in langs  #集合の要素かどうか判定するにはinを使う。
Out[63]: (True, False)
In [64]: s1=set(range(10))  #10未満の整数の集合(iteratorをsetにcastしている）
In [65]: s1
Out[65]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
In [66]: s2=set(filter(lambda x: x%2==0, s1))  #s1のうち2で割り切れるものの集合
In [67]: s2
Out[67]: {0, 2, 4, 6, 8}
In [68]: s3=set(filter(lambda x: x%3==0, s1))  #s1のうち3で割り切れるものの集合
In [69]: s3
Out[69]: {0, 3, 6, 9}
In [70]: s2.intersection(s3)  #積集合(積集合) (2でも3でも割り切れるもの)
Out[70]: {0, 6}
In [71]: s2.union(s3)  #和集合 (2または3で割り切れるもの)
Out[71]: {0, 2, 3, 4, 6, 8, 9}
In [72]: s2.difference(s3) #差集合(2では割り切れるが、3では割り切れないもの)
Out[72]: {2, 4,  8}
In [73]: s2.issubset(s1), s1.isuperset(s2) #s2はs1の部分集合か？s1はs2を含むか？
Out[73]: (True, True)
```

function(1)

Pythonの関数はdefで定義開始する。

```
In [79]: def mul4(x,y,z=3,u=4):   #4個の引数を取る。(z,uの２個はdefault値を持つ)
   .....:     return x,y,z,u,x*y*z*u  #5個の戻り値を持つ。(本当は1つのtupleであるが。)

In [89]: mul4(1,2,3,5)   #4個とも順番通りに引数を与える。(positional arguments)
Out[89]: (1, 2, 3, 5, 30)
In [90]: mul4(u=5,z=3,y=2,x=1)  #このように書けば順序は不問。(keyword arguments)
Out[90]: (1, 2, 3, 5, 30)

In [91]: mul4(1,2,u=5)  #zは与えていないので、default値(3)が使われる。
Out[91]: (1, 2, 3, 5, 30)

In [93]: mul4(1,2,u=5,z=4)  #2個のpositional argumentsと2個のkeyword argument
Out[93]: (1, 2, 4, 5, 40)
In [94]: t,d=(1,2),dict(u=5,z=4)  #tuple(t)(listでも可)とdict(d)を設定しておく。
In [95]: mul4(*t,**d)  #これはmul4(1,2,u=5,z=4)で与えるのと等価である。
Out[95]: (1, 2, 4, 5, 40)
```

一般に、呼び出し側の関数の引数の中で使われる*t, **dとはそれぞれtuple(t)(listでも可) , dict (d)を展開するという意味である。

function(2)

関数定義側の引数の中でも、*やを使うことができる。**
この書き方は、可変長引数や、Optionの数が多い場合等で非常に有用である。

```
In [2]: def mul4(*pargs,**kargs):
   ...:     assert len(pargs)==2, pargs    #positional引数の数は2個であること。
   ...:     x,y=pargs
   ...:     z=kargs.get('z',3)
   ...:     u=kargs.get('u',4)
   ...:     return x,y,z,u,x*y*z*u
In [3]: mul4(1,2,u=5)  #pargs=(1,2), kargs=dict(u=5)が入る。
Out[3]: (1, 2, 3, 5, 30)
```

#もちろん、f(x,y,*pargs), f(x,y,kargs)といように、通常の引数と併用することも自由。**

```
In [18]: def mul3(w:'width'=3,d:'depth'=2,h:'height'=1)->'volume':
   ....:     '''箱の体積を求める'''
   ....:     return w*d*h
In [19]: mul3(4,3)
Out[19]: 12
```

#関数の説明はdef直後のtriple quate領域に書く。(これをdocstringと呼ぶ)
#python3からは、引数部分にも直接説明文字列を入れられるようになった。

[参考] MATLABでのfunction

以下の内容のmul4.mと名前のファイルを作成しておく。
関数名とファイル名を一致させておくこと。

```
function [x1,y1,z1,u1,p]=mul4(x,y,z,u)
   x1=x;y1=y;z1=z;u1=u;p=x*y*z*u;
end
```

#戻り値が1個の関数では、括弧[ ]は不要である。

caller側では以下のように利用する。

```
--> [x,y,z,u,p]=mul(1,2,3,5);
--> disp([x,y,z,u,p])
  1  2  3  5 30
```

lambda

In [161]: list(**map**(l**ambda** x:x**2, range(4))) #[0,1,2,3]の全てについて2乗+1を求める。
Out[161]: [1, 5, 10, 17]

In [164]: list(**filter**(l**ambda** x:x%3==0, range(10))) #3で割り切れる10未満の整数
Out[164]: [0, 3, 6, 9]

すでに、mapやfilterの説明で述べたが、lambdaとは、one-linerで定義可能な無名関数である。(一般には、「lambda 引数列: 戻り値」という形をとる。）

実際、上のような例で、defを使って、関数を複数行で定義し、さらにその関数を参照するのは面倒だし美しくない。

[参考] MATLABでのlambda

MATLABではpythonのlambdaに相当する無名関数は
@(引数) 式
というように書く。

mapの説明のところで出てきた例と同じであるが、

```
--> a=arrayfun(@(x) x^2+1, 0:3); disp(a);
%pythonのa=map(lambda x: x**2+1, range(4)))と等価
1 2  5 10
```

class

```
class Quaternion:
    def __init__(self,a=0.0,b=0.0,c=0.0,d=0.0):
        self.a, self.b,self.c, self.d=a,b,c,d
    def __add__(self,other):
        return Quaternion(
            self.a+other.a, self.b+other.b,
            self.c+other.c, self.d+other.d)
    def __mul__(self,other):
        a,b,c,d=self.a, self.b, self.c,self.d
        A,B,C,D=other.a,other.b,other.c,other.d
        return Quaternion(
            a*A-b*B-c*C-D*D, a*B+b*A+c*D-D*c,
            a*C+c*A+d*B-b*D, a*D+d*A+b*C-c*B)
    def __str__(self):
        return '%1.2f+%1.2fI+%1.2fJ+%1.2fK' % ¥
            (self.a,self.b,self.c,self.d)
q1=Quaternion(1,1,1,1)
q2=Quaternion(1,-1,-1,-1)
print('q1+q2=',q1+q2)
print('q1*q2=',q1*q2)
```

quaternion.pyを作成し、
Quaternion classを定義する。
(Quaternionとは4元数のこと)
ここでは簡単のため、
「和」と「積」と「文字列変換」のみ定義

__init__ : instance生成関数
__add__: +演算子定義関数
__mul__: *演算子定義関数
__str__: 文字列変換関数。
(printでは自動的に__str__がcallされる)

[実行結果]

```
In [10]: run quaternion
q1+q2= 2.00+0.00I+0.00J+0.00K
q1*q2= 2.00+0.00I+0.00J+0.00K
```

この場合は、和と積が同じ値になる。

PythonとMATLABの制御構造

[Pythonでのfor文とif文]

```
for i in range(3):
    if i<1:
        print(i,'less')
    elif i==1:  #pythonではelif
        print(i,'equal')
    else:
        print(i,'more')
```

[Pythonでのwhile文]

```
i,j=0,1  #pythonでは複数代入可能
while i+j<10:
    i,j=j,i+j   #pythonでは複数代入可能
    print(j)   #Fibonaccii数列(1,2,3,5,8...)
```

#loop式、条件式の最後にコロンを
#つけ、indentさせる。
#(コロンの後は必ずindentさせる)

[MATLABでのfor文とif文]

```
for i=0:2
    if i<1
        disp({i,'less'}); #中括弧はcell array
    elseif i==1  #MATLABではelseif
        disp({i,'equal'});
    else
        disp({i,'more'});
    end  #MATLABではendが必要
end  #MATLABではendが必要
```

[MATLABでのwhile文]

```
i=0;j=1;
while i+j<10
    i_old=i;i=j;j=i_old+j;
    disp(j);
end   #MATLABではendが必要
```

module, package ,library

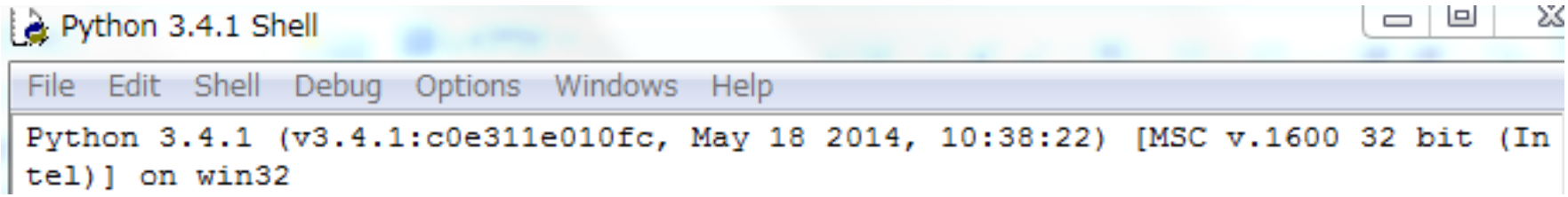
pythonでは、一つのscriptファイルのことをmoduleという。
一つのmoduleから、別のmoduleで定義された変数、関数、classなどを利用する場合は、
このmoduleをimportして利用する。
また、複数のmoduleを集めたものをpackageという。(packageの厳密な定義や構成法は
若干複雑であるので省略するが、利用するだけなら、難しく考える必要はない。)
moduleやpackageとは一般的な意味ではlibraryであると考えて良い。

moduleのimportの仕方は大雑把に2通りある。
(1a) import モジュール名
(1b) import モジュール名 as モジュール別名
(2a) from モジュール名 import オブジェクト名
(2b)　from モジュール名　import オブジェクト名 as オブジェクト別名
(複数モジュールあるいは複数オブジェクトの場合はカンマで区切って一文でimportできる。)
(packageの場合は、「パッケージ名.モジュール名」というようにピリオドで結合して記述する。)

(1a)(1b)の形式では、importする側では、「モジュール(別)名.オブジェクト名」として参照する。
すなわち、**モジュール(別)名がそのまま名前空間の役割を果たす**。
(2a)(2b)の形式では、オブジェクト(別)名を修飾なしでそのまま利用可能である。
どちらも一長一短があるので状況に応じて使い分ければよい。

拡張module

pythonではmoduleをC言語で作成するための、インターフェースが定義されている。
C言語で作成されたmoduleのことを拡張moduleという。
[コメント]
C言語インターフェースをそのまま使って、拡張moduleを作成するのは、かなり面倒な作業であり、これを軽減してくれるツールがいくつか存在する。
Cythonはそのうちの代表的なツールである。**CythonはPythonの上位互換言語であり、変数に型をつけることができる**。この型情報などを利用して、C言語のソースファイルを自動生成してくれる。
□
[コメント]
拡張moduleを自分でビルドする場合は、C コンパイラーのバージョンに注意する必要がある。Python shell起動時に表示されるMSC v.1600とはMSVC 2010のことであり、このPython自体をビルドしたコンパイラと合わせる必要がある。



標準library

python言語と共に標準で提供されるライブラリである。以下はほんの一例である。

math: 数学関数
cmath: 複素数学関数
Tkinter： tcl/tkへのpython インターフェイス (pythonの標準GUIツールキット)
sqlite3: データベース
pickle: オブジェクトのシリアライゼーション
zipfile : ZIPアーカイブ
re: 正規表現
xml: XML操作
threading: スレッドベースの並列処理
multiprocessing: プロセスベースの並列処理

主なthird party library(1)

(1)Numpy : 配列処理、線形代数
(2)matplotlib: グラフ作成(2D,3D)
(3)scipy: 各種科学技術計算ライブラリ
(信号処理、数値積分、微分方程式、最適化問題、疎行列、wavelet...)
(4)sympy: 数式処理
(sympyにはmpmath(任意精度計算)も内蔵されており、特殊関数なども利用可能)

科学技術計算では以上の4つが特に重要であると思われる。

http://www.scipy.org/ より

主なthird party library(2)

これ以外にも、多くの便利なサードパーティライブラリが存在する。以下一例である。

pandas: データ解析
python for windows extension: MS-Windows利用のための拡張機能(COM制御など)
xlrd, xlwt, xlutils: Excelファイルの読み書き
Vpython: 3Dグラフィックス
pydot : Graphviz's DOT(グラフ記述言語)のpython interface
pysvg : SVG(Scalable Vector Graphics)生成
pymol: 分子グラフィックス
pyserial: シリアル通信
pyusb: USB制御
django: WEB フレームワーク
mpi4py: MPIのpython interface
pyCUDA: CUDAのpython interface
pyQuante: 量子化学計算
pythtb: 電子状態のtight binding model

Numpy (配列の生成1)

Numpyでは任意次元の配列を取り扱うことができる。
(Excelの代用としても十分利用価値がある。しかも多次元版Excelである。
Excelの代用という意味ではpandasを使うとさらに高度な処理が可能。）

```
In [1]: import numpy as np   #npという別名でimportするのが標準的である。

In [2]: a1=np.array([1,2,3],int)  #1次元int型arrayの生成
In [3]: a2=np.array([[1,2,3],[4,5,6]])  #2次元arrayの生成(この場合自動でint型になる)
In [5]: a3=np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])  #3次元arrayの生成

In [6]: a2.dtype, a1.shape, a2.shape, a3.shape  #data型(dtype)と形状(shape)を調べる
Out[6]: (dtype('int32'), (3,), (2, 3), (2, 2, 3))
#shapeの定義では、[]の入れ子の外側が左側軸、内側が右側軸と解釈される。

In [7]: a1.astype(float),a1.astype(complex)     #astype()で型変換したarrayが得られる。
Out[7]: (array([ 1.,  2.,  3.]), array([ 1.+0.j,  2.+0.j,  3.+0.j]))

In [8]: a4=np.array(['日本',[1,2], {'a':10,'b':20}],object)
#object型のarrayには何でも入れることができる。
```

Numpy (配列の生成2)

```
In [16]: np.zeros((2,3),int)
 #all 0のarrayを生成
Out[16]:
array([[0, 0, 0],
       [0, 0, 0]])

In [17]: np.ones((2,3),int)
#all 1のarrayを生成
Out[17]:
array([[1, 1, 1],
       [1, 1, 1]])

In [18]: np.identity(3,int)
#3x3単位行列を生成
Out[18]:
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

```
In [19]: a5=np.diag([1,2,3]);a5
#対角成分から対角行列を生成
Out[19]:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

In [20]: np.diag(a5)
#正方行列から対角成分を取り出す。
Out[20]: array([1, 2, 3])
```

[参考] MATLABでの配列の生成

```
--> a1=[1,2];a2=[1;2];a3=[1,2;3,4];
--> disp(a1)   %行ベクトル
 1 2
--> disp(a2)  %列ベクトル
 1
 2
 --> disp(a3)  %2x2行列
 1 2
 3 4
-->
disp(size(a1));disp(size(a2));disp(size(a3));
 1 3   %1行3列
 3 1   %3行1列
 2 2   %2行2列
%MATLABでは行列を基本量と考える
%ので、行ベクトル、列ベクトルも
%行列の特別な場合と考える。
%MATLABでは3次元以上の配列は
%あまり統一的な設定方法は存在しない。
```

```
--> a1=zeros(1,2);a2=ones(1,2);a3=eye(2);
--> disp(a1)
 0 0
--> disp(a2)
 1 1
--> disp(a3)
 1 0
 0 1
%zeros(), ones(), eye()はMALAB,Numpy
%とも同じ関数名である。
%Numpyではeye()よりidentity()が正式だが。

--> a1={1,'dog';[2,3],{4,5}}
--> disp(a1)
 [1] [dog]
 [1x2 double array] [1x2 cell array]
%MATLABには何でも入れられるcell arrayと
%いうものがある。これはNumpyのobject型
%(np.array(...,object))に相当する。
```

Numpy(形状変更)

```
In [10]: a2=np.array([[1,2,3],[4,5,6]]);a2
#shape=(3,2)のarrayを生成する。
Out[10]:
array([[1, 2, 3],
       [4, 5, 6]])

In [11]: a2.reshape(6,)
#shape=(6,)の一次元形状に変更
Out[11]: array([1, 2, 3, 4, 5, 6])

In [12]: a2.reshape(-1,)
#(6,)の代わりに(-1,)でも同じ。
#-1は現在の形状を元に自動計算する。
Out[12]: array([1, 2, 3, 4, 5, 6])
```

```
In [13]: a2.reshape(3,2)
#shape=(3,2)の２次元形状に変更
Out[13]:
array([[1, 2],
       [3, 4],
       [5, 6]])

In [14]: a2.reshape(-1,2)
#(3,2)の代わりに(-1,2)でも同じ。
#-1は現在の形状を元に自動計算する。
Out[14]:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

形状変更の際には全要素数は
不変であることに注意

[参考] MATLABでの(形状変更)

```
--> a2=[1,2,3;4,5,6];
%shape=(3,2)のarrayを生成する。
--> disp(a2)
 1 2 3
 4 5 6
```

```
--> disp(size(a2));disp(size(a3));disp(size(a4));
 2 3  %2行3列
 1 6  %1行6列
 3 2  %3行2列
```

%size()はpythonのshapeに相当

```
--> a3=reshape(a2,1,6);
%shape=(1,6)の一次元形状に変更
--> disp(a3)
 1 4 2 5 3 6
```

%Numpyとは要素の並びが異なることに注意。
%後で述べるようにNumpyはC　言語的並び
%であり、MATLABはFortran的並びである。

```
--> a4=reshape(a2,3,2);
%shape=(3,2)の２次元形状に変更
--> disp(a4)
 1 5
 4 3
 2 6
```

Numpy(部分配列,slice)

#sliceとは等差数列型index list
#のことであり、これを[...]に与えて
#部分配列を取得する。

```
In [17]: a1=np.array(range(1,10)).¥
   ....: reshape(3,3);a1
Out[17]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
In [18]: a1[1:3,1:3]
#1以上3未満の行番号、
#1以上3未満の列番号を取り出す。
Out[18]:
array([[5, 6],
       [8, 9]])
In [19]: a1[1:,1:]
#sliceの第二parameterを省略したら
#最後までの意味
Out[19]:
array([[5, 6],
       [8, 9]])
In [20]: a1[:-2,:-2]
#-2は最後から2番目の意味
#sliceの第一parameterを省略したら
#最初から(すなわち0)の意味
Out[20]: array([[1]])

In [21]: a1[::-1,::-1]
#sliceの第3parameterはstepの意味。
#これが-1ということは逆順を意味する。
Out[21]:
array([[9, 8, 7],
       [6, 5, 4],
       [3, 2, 1]])

In [22]: slice1=slice(0,3,2)     #0:3:2の意味
In [23]: a1[slice1,slice1]
#このようにslice objectを利用してもよい。
Out[23]:
array([[1, 3],
       [7, 9]])
```

Numpy(部分配列, fancy indexing)

#fancy indexingとは任意のindex列
#を[...]に与えて部分配列
#を取得する方式である。

```
In [24]: a1[[0,2],:]
Out[24]:  #第0行と第2行と取り出す。
array([[1, 2, 3],
       [7, 8, 9]])

In [25]: a1[:,[0,2]]
Out[25]:  #第0列と第2列と取り出す。
array([[1, 3],
       [4, 6],
       [7, 9]])
In [26]: a1.take([0,2],axis=0)
Out[26]:
#以前はfancy indexingの代わりに
#take()が利用されていた。
array([[1, 2, 3],
       [7, 8, 9]])
```

```
In [27]: a1[[0,2],[0,2]]
Out[27]: array([1, 9])
#両軸ともに、fancy indexingを使用。
#これは、[0,0]成分と[2,2]成分を取り出す。
#結果は1D arrayであり、以下とは別物
#であるので注意。
In [28]: a1[[0,2],:][:,[0,2]]
Out[28]:
array([[1, 3],
       [7, 9]])

In [29]: a1[[2,0,1],:]
Out[29]:
array([[7, 8, 9],
       [1, 2, 3],
       [4, 5, 6]])
#fancy indexingはこのように
#データの並べ替え(permutation)
#の目的でも利用される。
```

Numpy(部分配列, boolean indexing)

#boolean indexingとはboolean array
#を[...]に与えてTrueのindexだけの
#部分配列を取得する方式である。

```
In [30]: a2=(a1%2==0);a2
Out[30]:
array([[False,  True, False],
       [ True, False,  True],
       [False,  True, False]], dtype=bool)
```
#値が偶数の場合のみTrueとなる
#boolean arrayを作成

```
In [31]: a1[a2]   #boolean indexing
Out[31]: array([2, 4, 6, 8])
```
#indexがTrueに対応する成分だけ
#を1D arrayとして取り出す。

```
In [32]: a3=np.where(a2);a3
Out[32]: (array([0, 1, 1, 2], dtype=int32),
 array([1, 0, 2, 1], dtype=int32))
```
#np.where()はboolean arrayを
#fancy indexing用の「arrayのtuple」に
#変換してくれる。
#np.where()はMATLABのfind()に対応する。

```
In [33]: a1[a3]  #fancy indexing
Out[33]: array([2, 4, 6, 8])
```
#先ほどのa1[a2]と同じ結果が得られる。

[参考]MATLABでの部分配列

```
--> a1=reshape(1:9,3,3);disp(a1)
 1 4 7
 2 5 8
 3 6 9
--> a2=a1(1:2:end,1:2:end);disp(a2)
 1 7
 3 9
```

%slice型部分配列を取得。

%Numpyでのa1[0::2,0::2]と等価

%Numpyでは行、列のindexは
%0から始まるが、MATLABでは
%1から始まる。
%Numpyではindexingは[....]を
%使うが、MATLABでは(...)を使う。

%このように至るところで、
%NumpyはC言語的であり、
%MATLABはFortran的である。

```
--> disp(a1([1,3],:)) %fancy indexingも可能
 1 4 7
 3 6 9
--> disp(a1([1,3],[1,3]))
```

%両軸fancy indexingの仕様はNumpyと違う。

```
 1 7
 3 9
--> a2=find(rem(a1,2)==0);disp(a2)
 2    %find(...)は...がTrueとなるindexの
 4    %1次元化された列ベクトルを戻す。
 6    %Numpyのwhere()と違い常に1次元化する。
 8
--> a3=a1(a2); disp(a3)
 2
 4
 6
 8
```

%MATLABではboolean indexingは直接
%サポートしてないが、上のようにfind()を
%使ってfancy indexingに変換すればよい。

Numpy(部分配列への代入)

```
n [34]: a1[1:,1:]=¥
np.array([[50,60],[80,90]]);a1
#slice型部分配列への代入も可能
Out[34]:
array([[ 1,  2,  3],
       [ 4, 50, 60],
       [ 7, 80, 90]])

In [35]: a1[[0,2],:]=¥
 ....: np.array([[11,22,33],[77,88,99]]);a1
#fancy indexingの場合でも代入可能
Out[35]:
array([[11, 22, 33],
       [ 4,  50,  60],
       [77, 88, 99]])
```

```
In [36]: a1[a1>70]=[-1,-2,-3];a1
#boolean indexingの場合でも代入可能
Out[36]:
array([[11, 22, 33],
       [ 4, 50, 60],
       [-1, -2, -3]])

In [37]: a1[a1>70]=-10;a1
#broadcast（一つの値を複数個所にcopy)
#も可能。
Out[37]:
array([[11, 22, 33],
       [ 4, 50, 60],
       [-1, -2, -3]])
```

Numpy(部分配列、参照とコピー)

```
In [84]: a1=np.array([1,2,3]);a1
Out[84]: array([1, 2, 3])

In [85]: a2=a1[1:];a2  #slice型部分配列
Out[85]: array([2, 3])

In [86]: a2[0]=20;a2
Out[86]: array([20,  3])

In [87]: a1
Out[87]: array([ 1, 20,  3])
#a2を変更したら、a1まで変更された。
#すなわち、a2はa1の部分配列
#への参照である。
#もし、参照でなくコピーにしたければ
#a2=a1[1:].copy()と明示的にコピー
#させる必要がある。
```

```
In [91]: a1=np.array([1,2,3]);a1
Out[91a1]: array([1, 2, 3])

In [92]: a2=a1[[1,2]];a2   #fancy indexing
Out[92]: array([2, 3])

In [93]: a2[0]=20;a2
Out[93]: array([20,  3])

In [94]: a1
Out[94]: array([1, 2, 3])
#a2を変更しても、a1は変化しない。
```

[一般規則]
Numpyの部分配列は、indexがsliceの場合は参照である。その他の場合(fancy index等)はコピーである。
a1[[0,2],:]のようにsliceとfancy indexingが混在している場合もコピーである。

Numpy(配列の結合)

#まず2x2の2次元配列を2個用意する。

In [24]: a1=np.array(range(1,5)).¥
  .....: reshape(2,2);a1
Out[24]:
array([[1, 2],
[3, 4]])

In [25]: a2=np.array(range(5,9)).¥
  .....: reshape(2,2);a2
Out[45]:
array([[5, 6],
[7, 8]])

In [25]: **np.concatenate**((a1,a2),axis=0)
#第0軸(=行)方向に結合する。
#np.vstack((a1,a2))を使っても同じ。
Out[25]: #shape=(4,2)となる。
array([[1, 2],
[3, 4],
[5, 6],
[7, 8]])

In [26]: **np.concatenate**((a1,a2),axis=1)
#第1軸(=列)方向に結合する。
#np.hstack((a1,a2))を使っても同じ。

Out[26]: #shape=(2,4)となる。
array([[1, 2, 5, 6],
[3, 4, 7, 8]])

[参考] MATLABでの配列の結合

```
--> a1=reshape(1:4,2,2);disp(a1)
 1 3
 2 4
--> a2=reshape(5:8,2,2);disp(a2)
 5 7
 6 8

--> disp([a1;a2])  %cat(1,a1,a2)でも可
%Numpyのnp.vstack((a1,a2))と等価
 1 3
 2 4
 5 7
 6 8

--> disp([a1,a2]) %cat(2,a1,a2)でも可
%Numpyのnp.hstack((a1,a2))と等価
 1 3 5 7
 2 4 6 8
```

```
--> disp(cat(3,a1,a2))  %第3軸での結合
%Numpyのnp.dstack((a1,a2))と等価
(:,:,1) =
 1 3
 2 4
(:,:,2) =
 5 7
 6 8
--> disp(size(cat(3,a1,a2)))
 2 2 2   %shape=(2,2,2)のarrayが得られる。
```

Numpy(軸の入替)

```
In [19]: a1=np.array(range(2*3)).¥
   ....: reshape(2,3);a1
Out[19]:
array([[0, 1, 2],
       [3, 4, 5]])

In [20]: a1.T   #転置
#第0軸(行)と第1軸(列)の入れ替え
Out[20]:
array([[0, 3],
       [1, 4],
       [2, 5]])
```

$$B_{ji} = A_{ij}$$

```
In [23]: a2=np.array(range(2*3*4*5)).¥
   ....: reshape(2,3,4,5)
#shape=(2,3,4,5)の4次元arrayを生成

In [25]: a2.shape
Out[25]: (2, 3, 4, 5)

In [26]: a2.swapaxes(1,3).shape
#第1軸と第3軸を入れ替える。
Out[26]: (2, 5, 4, 3)
```

$$B_{ilkj} = A_{ijkl}$$

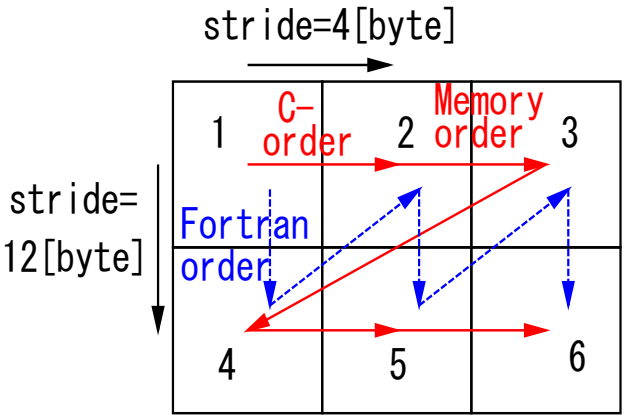
```
In [27]: a3=a2.transpose(1,2,3,0)
#第1,2,3,0軸を新しい第0,1,2,3軸とする。
In [28]: a4=np.einsum('ijkl->jkli',a2)
#einsumを使うと同じ処理を視覚的に
#分かり易く表現できる。
In [29]: a3.shape,a4.shape,(a3==a4).all()
Out[29]: ((3, 4, 5, 2), (3, 4, 5, 2), True)
```

$$B_{jkli} = A_{ijkl}$$

Numpy(1次元化とメモリ上の並び)

```
In [13]: a1=array([[1,2,3],
   ...: [4,5,6]])  #2次元array
In [14]: a1
Out[14]:
array([[1, 2, 3],
       [4, 5, 6]])
In [15]: a1.T #転置（軸の入替）
Out[15]:
array([[1, 4],
       [2, 5],
       [3, 6]])
In [16]: a1.shape, a1.T.shape
Out[16]: ((2, 3), (3, 2))
In [17]: a1.strides, a1.T.strides
Out[17]: ((12, 4), (4, 12))
#Numpyは軸の入れ替えの際
#memory上のdata並びを
#書き換えるわけではない。
#各軸のstride情報を
#書き換えているだけである。
```

```
In [18]: a1.ravel(order='C'), a1.T.ravel(order='C')
#C言語並びを仮定して1次元化する(default)
#a1.ravel(order='C')とa1.reshape(-1)は等価
Out[18]: (array([1, 2, 3, 4, 5, 6]), array([1, 4, 2, 5, 3, 6]))

In [19]: a1.ravel(order='F'), a1.T.ravel(order='F')
#Fortran並びを仮定して1次元化する。
Out[19]: (array([1, 4, 2, 5, 3, 6]), array([1, 2, 3, 4, 5, 6]))

In [20]: a1.ravel(order='K'), a1.T.ravel(order='K')
#Memory上の並びに従って1次元化する。
Out[20]: (array([1, 2, 3, 4, 5, 6]), array([1, 2, 3, 4, 5, 6]))
```



Numpy(軸の縮約)

```
In [30]: a1=np.array(range(2*3*4*5)).¥
   ....: reshape(2,3,4,5)
In [31]: a2=np.array(range(6*4*3*2)).¥
   ....: reshape(6,4,3,2)
In [32]: a3=np.tensordot(a1,a2,
   ....: ([0,1,2],[3,2,1]))
#a1の第0,1,2軸とa2の第3,2,1軸とを
#縮約する。
In [33]: a4=np.einsum('ijkl,mkji->lm',
   ....: a1,a2)
#einsumを使うと同じ処理を視覚的に
#分かり易く表現できる。
In [34]: a3.shape,a4.shape,(a3==a4).all()
Out[35]: ((5, 6), (5, 6), True)
```

$$C_{lm} = \sum_{ijk} A_{ijkl} B_{mkji}$$

shape=$(5,6) \leftarrow (2,3,4,5)(6,4,3,2)$

```
In [36]: a5=np.array(range(6*4)).¥
   ....: reshape(6,4)
In [37]: a6=a5.dot(a1)
In [38]: a7=np.tensordot(a5,a1,[-1,-2])

In [39]: a6.shape,a7.shape,(a6==a7).all()
Out[39]: ((6, 2, 3, 5), (6, 2, 3, 5), True)
```

A.dot(B)とは、
(1)Bが2次元以上の配列の場合、
np.tensorsot(A,B,[-1,-2])と等価である。
(2)Bが1次元の配列の場合、
np.tensordot(A,B,[-1,-1])と等価である。
(-1は右端軸、-2は右から2番目の軸の意味)

従って特にA,Bが2次元配列の場合は行列行列積を意味し、Aが2次元配列でBが1次元配列の場合は行列ベクトル積を意味する。

np.inner(A,B)はnp.tensordot(A,B,[-1,-1])と等価

[参考] MATLABでの軸の入替と縮約

ここでは、MATLABの多次元配列についての記述は省略する。
2次元以下の配列に限定すれば、
軸の入替と縮約とは、単に、
転置と、行列積に他ならない。
(列ベクトル、行ベクトルも行列とみなす。)
に他ならない。

```
--> a1=[1,2;3,4];a2=[5,6;7,8];
--> disp(a1)
 1 2
 3 4
--> disp(a1')  %転置行列
%Numpyでは転置はa1.Tと書かれる。
 1 3
 2 4
--> disp(a1*a2)     %行列行列積
%Numpyではa1.dot(a2)と書かれる。
 19 22
 43 50
```

[注意]
MATLABでは行列積はa1*a2であり、
要素毎の積はa1.*a2である。
これに対して、Numpyでは、
a*bは要素毎の積を意味する。

実は、Numpyにはこれまで説明してきた任意次元配列を表すndarray class以外にmatrix classというものが存在する。
matrix classを使う場合は行列積はMATLAB同様にa1*a2と書ける。
□

Numpy(outerとkron)

```
In [77]: A=np.array([[1,2],[3,4]])
In [78]: I=np.identity(2) #2x2単位行列

In [79]: C=np.outer(I,A);C
#まず、I,Aを1次元化したのち、
#2つのvectorのtensor積を計算
Out[79]:
array([[ 1.,  2.,  3.,  4.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 1.,  2.,  3.,  4.]])

In [80]: D=np.kron(I,A);D
#Kroneckerのtensor積(いわゆる
#演算子としてのtensor積)を計算
Out[81]:
array([[ 1.,  2.,  0.,  0.],
       [ 3.,  4.,  0.,  0.],
       [ 0.,  0.,  1.,  2.],
       [ 0.,  0.,  3.,  4.]])
```

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

$$\mathrm{D} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 3 & 4 \end{pmatrix}$$

```
In [82]: C.reshape(2,2,2,2).¥
  ....: swapaxes(1,2).reshape(4,4)
Out[82]:
array([[ 1.,  2.,  0.,  0.],
       [ 3.,  4.,  0.,  0.],
       [ 0.,  0.,  1.,  2.],
       [ 0.,  0.,  3.,  4.]])
#一旦,shape=(2,2,2,2)にして、第1軸と第2軸
#を入れ替えれば、CとDは同じものである。
```

[参考] MATLABでのouterとkron

```
--> A=[1,2;3,4]; I=eye(2);
-->disp(reshape(I,4,1)*reshape(A',1,4))
#MATLABにはouter相当の関数は
#見当たらないようだが、
#1次元化して、行列積を計算すれば
#十分である。
 1 2 3 4
 0 0 0 0
 0 0 0 0
 1 2 3 4

-->disp( kron(I,A))
 1 2 3 4
 3 4 0 0
 0 0 1 2
 0 0 3 4
```

Numpy(ufunc) (1)

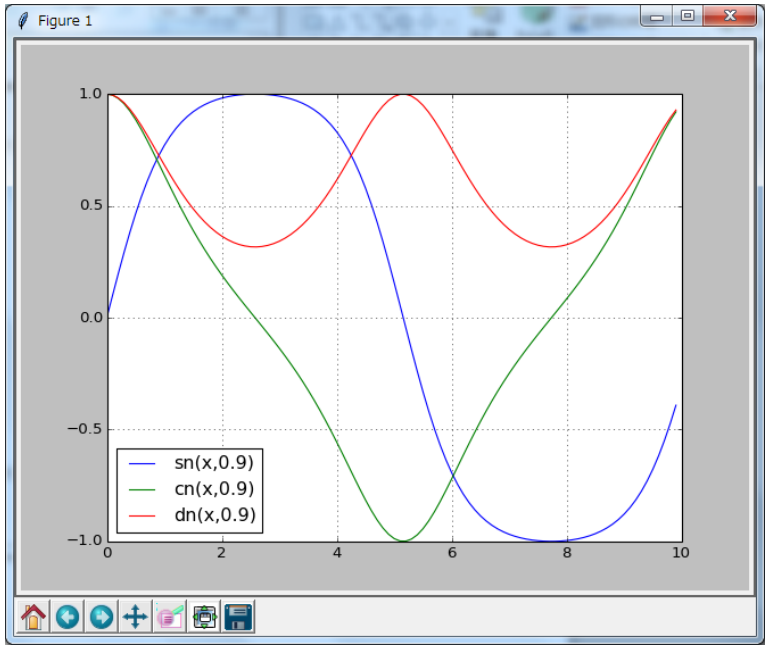
ufuncとは引数、戻り値ともにarrayである関数であり、arrayの要素毎に関数が適用されるものである。numyではsin,cos,exp等の代表的な関数や多項式(poly1d)に対してufuncが用意されている。

```
In [6]: a1=np.array([1,2,3,4],float)

In [7]: np.exp(a)    #np.exeはufuncである。(要素毎のexpを計算する。)
Out[7]: array([  2.71828183,   7.3890561 ,  20.08553692,  54.59815003])

In [8]: a**0.5    #べき乗もufuncである。(要素毎のべき乗を計算)
Out[8]: array([ 1.        ,  1.41421356,  1.73205081,  2.        ])

In [9]: a2=np.array([5,6,7,8],float)

In [10]: a1+a2    #和もufuncである。(要素ごとの和)
Out[10]: array([  6.,   8.,  10.,  12.])

In [11]: a1*a2    #積もufuncである。(要素ごとの積)
Out[11]: array([  5.,  12.,  21.,  32.])
```

Numpy(ufunc) (2)

np.vectorize()を利用することでscalar関数をufunc化(vector化)することが可能である。
以下は、scalar関数しか用意されてないJacobiの楕円関数をvector化する例である。

```
In [138]: from sympy.mpmath import ellipfun
In [139]: jsn,jcn,jdn=map(ellipfun, 'sn cn dn'.split())  #ellipfunは関数generator
In [140]: jsnv=np.vectorize(lambda x:float(jsn(x,0.9).real))
In [141]: jcnv=np.vectorize(lambda x:float(jcn(x,0.9).real))
In [142]: jdnv=np.vectorize(lambda x:float(jdn(x,0.9).real))
```



後で述べるが、ufunc化された関数
を使うと、例えばmatplotlibのplot関数で
以下のように、簡潔に記述することができる。
plot(X, jsnv(X), X, jcnv(X), X, jdnv(X))

**さらに、一般にvector化された関数を
使うと、高速で処理することが可能となる。
(ただし、C言語でvector化が実装されている
ことが前提)**

Numpy(線形代数1)

```
In [63]: import numpy.linalg as LA
In [64]: A=np.array([[1,2],[2,5]])
#対称行列Aを定義
In [65]: b=np.array([3,4])
#vector bを定義

In [66]: x=LA.solve(A,b);x
#線形方程式Ax=bを解く。
Out[66]: array([7., -2.])

In [67]: Ainv=LA.inv(A);Ainv
#Aの逆行列を求める。
Out[67]:
array([[ 5., -2.],
       [-2.,  1.]])
In [68]: Ainv.dot(b)
#この逆行列を使って、Ax=bを解く。
Out[68]: array([ 7., -2.])
#当然、さっきと同じ結果が得られる。
```

```
In [69]: E,C=LA.eigh(A) #固有値問題を解く。
In [70]: E #Aの固有値のlist
Out[70]: array([ 0.17157288,  5.82842712])
In [71]: C #Aの固有vectorのlist
Out[71]: #各列が固有vectorである。
array([[-0.92387953,  0.38268343],
       [ 0.38268343,  0.92387953]])
In [72]: np.diag(E)
#Eを対角成分とする行列
Out[72]:
array([[ 0.17157288,  0.        ],
       [ 0.        ,  5.82842712]])
In [73]: LA.norm(A.dot(C)-C.dot(np.diag(E)))
#AC=CEが成立していることを確かめる。
Out[73]: 1.3668035872266426e-16  #OK
In [74]: L=LA.cholesky(A);L #Cholesky分解
Out[74]: #左下三角行列
array([[ 1.,  0.],
       [ 2.,  1.]])
In [75]: LA.norm(L.dot(L.T)-A) #LL'=Aを確かめる。
Out[75]: 0.0    #OK
```

Numpy(線形代数2)

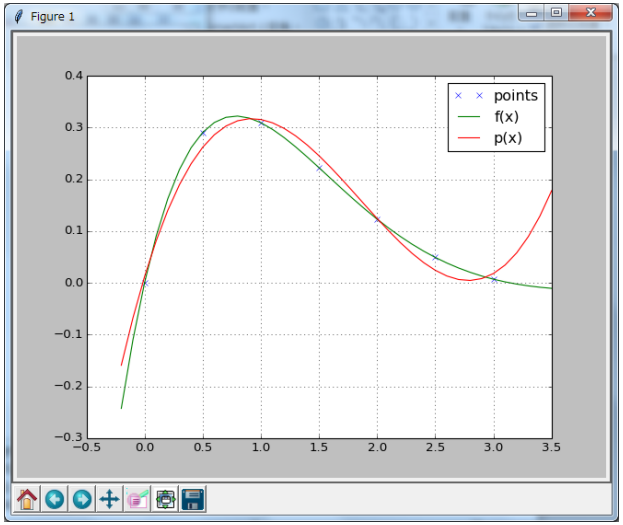
```
In [76]: B=outer([1,2],[1,3]);B
Out[76]:
array([[1, 3],
       [2, 6]])
In [77]: LA.matrix_rank(B,tol=1e-10)
#行列のrankを調べる。
Out[77]: 1  #outerで作ったので当然rank=1

In [78]: E,C=eig(B) #固有値問題を解く。
#非Hermite行列の対角化はeigh()でなく
#eig()を使用すること。
In [79]: E  #Bの固有値のlist
Out[79]: array([ 0.,  7.])
#rank=1なので非ゼロ固有値は1個のみ。
In [80]: LA.norm(B.dot(C)-C.dot(np.diag(E)))
#BC=CEが成立していることを確かめる。
Out[80]: 4.4408920985006262e-16  #OK
```

```
In [81]: U,S,V=LA.svd(B)  #特異値分解
In [82]: S   #Bの特異値のlist
Out[82]: array([  7.07106781e+00,
  5.61733355e-16])
#rank=1なので非ゼロ特異値は1個のみ。
In [83]: LA.norm(U.dot(np.diag(S)).dot(V)-B)
#USV=Bを確かめる。
Out[83]: 3.20429675896556 61e-15  #OK
In [84]: I=np.identity(2)  #2次元単位行列
In [85]: LA.norm(U.dot(U.T)-I) #Uは直交行列
Out[85]: 4.6443961262081255e-16 #OK
In [86]: LA.norm(V.dot(V.T)-I)  #Vは直交行列
Out[86]: 0.0   #OK
In [87]: Q,R=LA.qr(B); R   #QR分解
Out[87]:   #Rは右上三角行列
array([[ -2.23606798e+00,  -6.70820393e+00],
       [  0.00000000e+00,   1.77635684e-15]])
In [88]: LA.norm(Q.dot(Q.T)-I) #Qは直交行列
Out[88]: 4.7102773760513248e-16 #OK
```

Numpy(多項式)

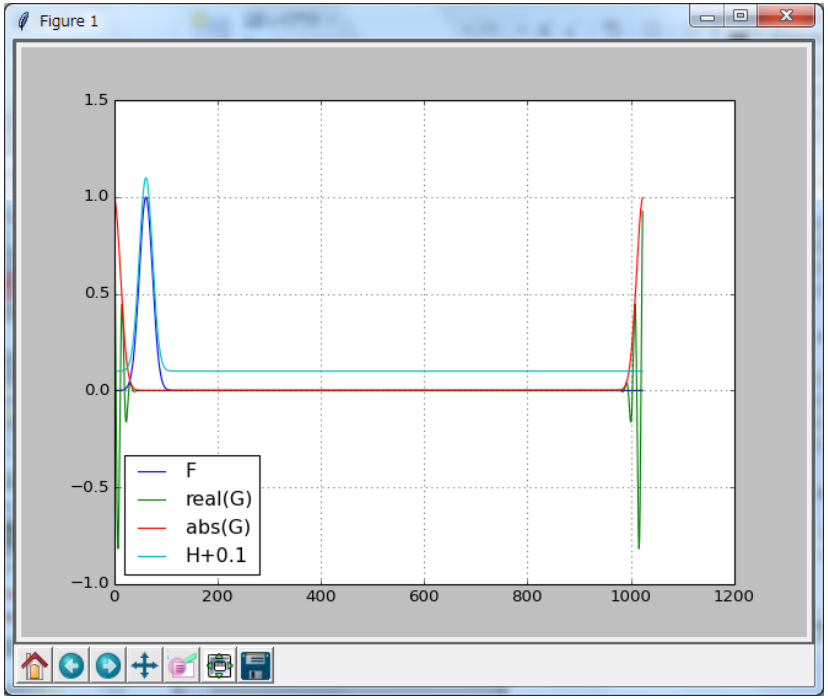
```
In [26]: p1=np.poly1d([1,2]) #x+2
In [27]: p2=np.poly1d([1,3,2]) #x**2+3*x+2
In [28]: p3=poly1d([1,1],True)  #根で定義
In [29]: p1,p3
Out[29]: (poly1d([1, 2]), poly1d([ 1, -2,  1]))
In [30]: print(p2)  #printで見やすく表示
   2
1 x + 3 x + 2
In [31]: p1+p2  #多項式の和
Out[31]: poly1d([1, 4, 4])
In [32]: p1*p2  #多項式の積
Out[32]: poly1d([1, 5, 8, 4])
In [33]: p1**3  #多項式の冪乗
Out[33]: poly1d([ 1,  6, 12,  8])
In [34]: p2/p1   #多項式の割算（商と余）
Out[34]: (poly1d([ 1.,  1.]), poly1d([ 0.]))
In [35]: p2(1),p2(2),p2([1,2])  #多項式の値
Out[35]: (6, 12, array([ 6, 12]))
In [36]: p2(p1)  #多項式に多項式を代入
Out[36]: poly1d([  1.,   7.,  12.])
```

```
In [37]: p2.c   #多項式の係数
Out[37]: array([1, 3, 2])
In [38]: p2.r   #多項式の根
Out[38]: array([-2., -1.])
In [39]: f=lambda x:np.exp(-x)*np.sin(x)
In [40]: x=np.arange(0,3.5,0.5) #samping点
In [41]: xi=np.arange(-0.2,3.6,0.1) #plot点
In [42]: p=np.poly1d(np.polyfit(x,f(x),3))
#sampling点を3次多項式でfittingする。
In [43]: plt.plot(x,f(x),'x',xi,f(xi),xi,p(xi))
```



Numpy(FFT)

```
In [16::  X=np.arange(1024)  #1024点を用意
In [17]: F=np.roll(np.exp(-np.pi*(((X-512)/32)**2)),-450)  #Gaussian波形を用意。
In [18]: G=np.fft.fft(F)/32  #FFT
In [21]: H=np.fft.ifft(G)*32  #逆FFT
In [22]: plt.plot(X,F,X,G.real,X,abs(G),X,H+0.1)
In [25]: plt.grid()
In [26]: plt.legend(['F','real(G)','abs(G)','H+0.1'],loc='best')
```

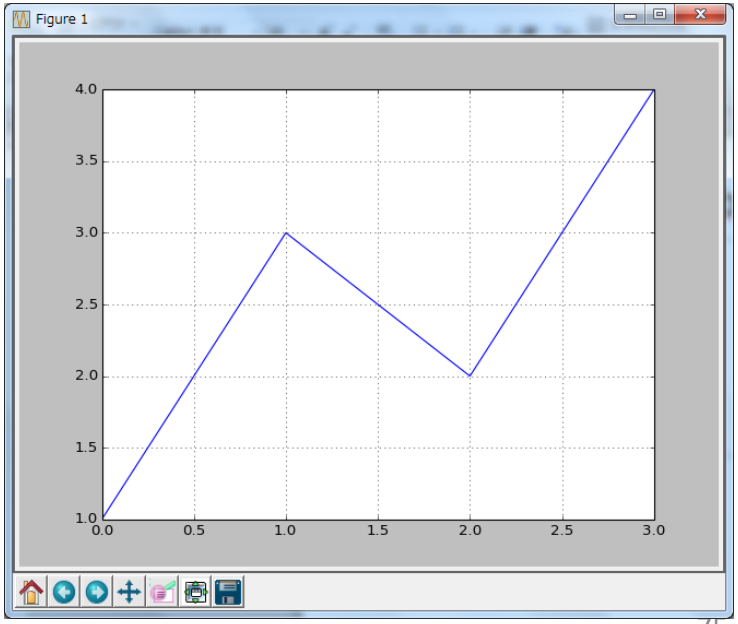


F-->(FFT)-->G-->(逆FFT)-->H
で元の波形に戻ることを確認
(Hはoffsetをつけてplotしている)

Matplotlib (非interactive mode)

optionなしで、ipythonを起動して、そこからMatplotlibを利用する場合は、Matplotlibは非interactive modeで起動する。

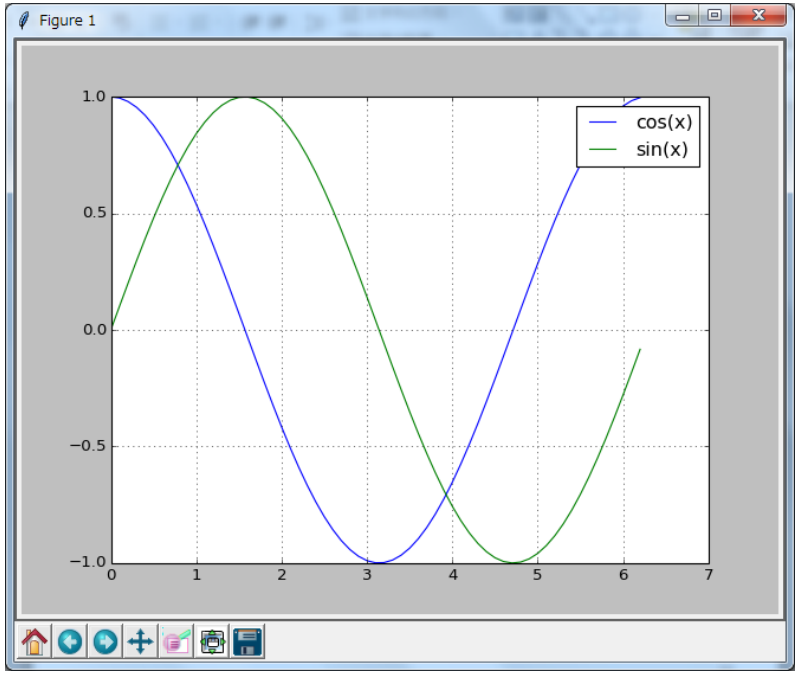
```
In [12]: import matplotlib.pyplot as plt   #これが定形的なimportの仕方。

In [16]: plt.plot([1,3,2,4])
Out[16]: [<matplotlib.lines.Line2D at 0x490c030>]

In [17]: plt.grid()

In [18]: plt.show()
#非interactive modeでは
#showを実行して初めてグラフが書かれる。

#この時点で無限Loopに入り、
#ipythonからは制御できなくなるが、
#グラフ下のツールバーからいくつかの
#操作は可能である。
```



Matplotlib (interactive mode)

OSのshell上から
ipython --pylab
で起動する。
このmodeでは、numpy, matplotlibなどはすでにimportされた状態になっている。
(np. やplt.はつけてもつけなくても利用可能になっている。)
さらに、matplotlibはinteractive modeとなっている。

```
In [18]: X=linspace(0,2*pi,100)
In [19]: Y1=cos(X)
In [20]: Y2=sin(X)
In [21]: plot(X,Y1,X,Y2)
Out[21]:  省略
In [22]: grid()
In [23]: legend(['cos(x)','sin(x)'])
Out[23]:  省略
```



#interactive modeではグラフ関連
#コマンドを入力するごとに
#グラフが変化する。

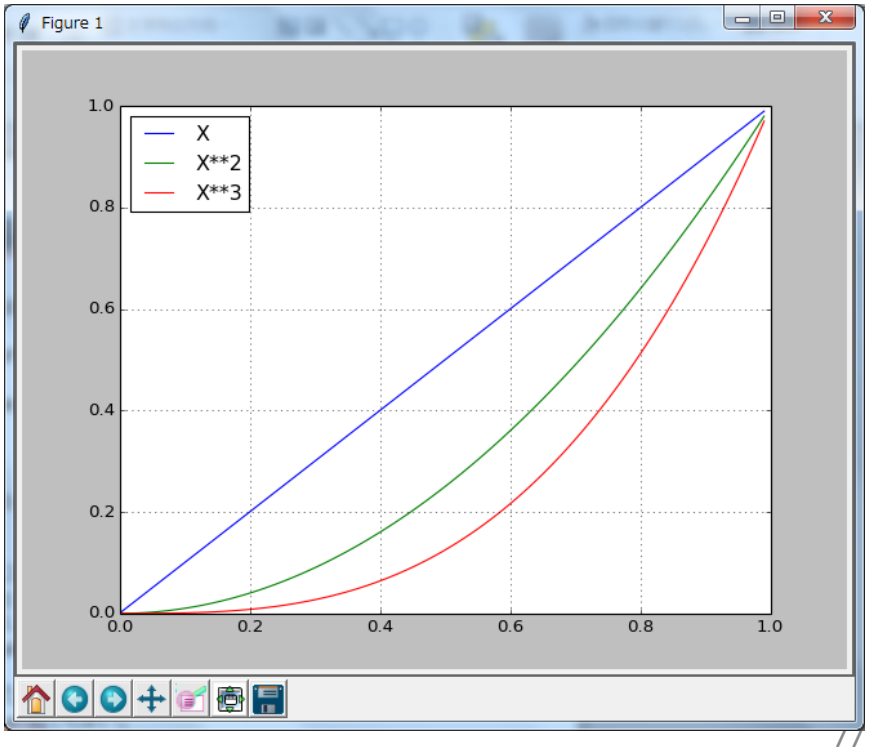
Matplotlib　(複数のLine)

In [30]: X=arange(0,1,0.01)　#1D np.array型である。

In [32]: a1=vstack((X,X**2,X**3))　#3つの1D arrayを縦に積んで2D arrayを作る。

In [34]: plot(X,a1.T)　#.T(転置)を忘れないこと。
Out[34]:　省略

In [35]: grid()
In [36]: legend('X X**2 X**3'.split(), ¥
   ....: loc='best')
Out[36]: 省略



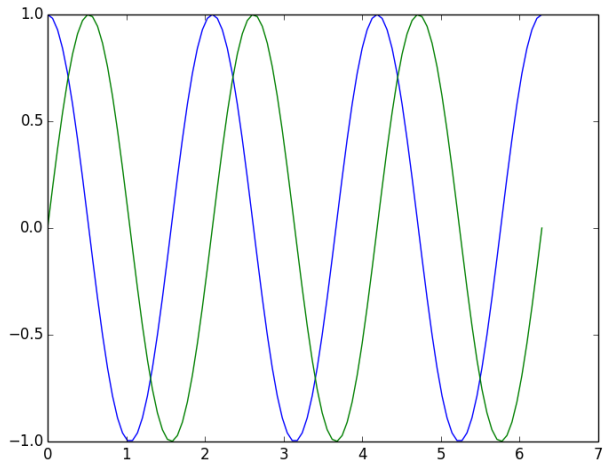
複数のLineを書くには、
plot(X1,Y1);plot(X2,Y2)
と2-plotに分けてもよいし、
plot(X1,Y1,X2,Y2)と1-plotでもよい。

さらに、ここで紹介したように
Xが共通なら、Yはnp.arrayでもよい。

Matplotlib (ファイルへの出力)

```
In [52]: X=linspace(0,2*pi,100)
In [53]: for n in range(8):
   ....:     Y=exp(1.0j*X*n)
   ....:     plot(X,Y.real,X,Y.imag)
   ....:     savefig('exp_%d.png' % n)
   ....:     clf()
   ....:
```

#ファイル出力はsavefig()を利用。
#重ね書きされないように
#毎回clf()(=clear figure)を実行。



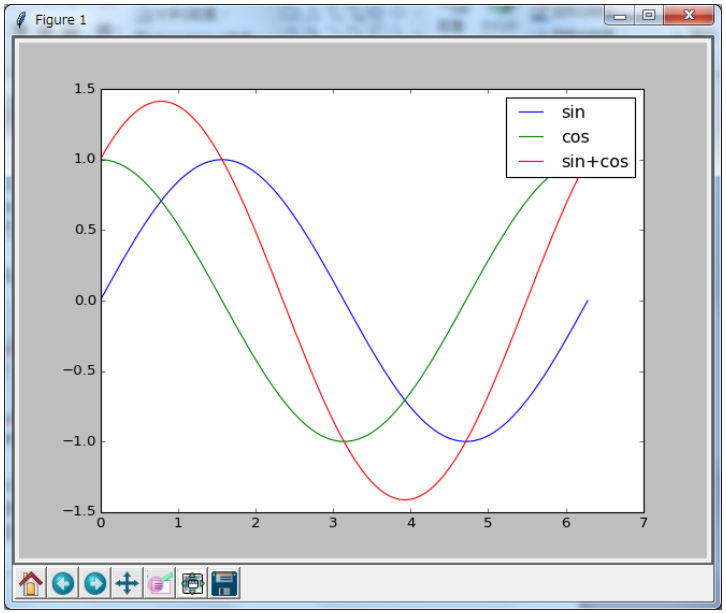

```
In [54]: ls *.png
 ドライブ G のボリューム ラベルがありません。
 ボリューム シリアル番号は 8496-2681 です

 G:¥projects1¥ipython のディレクトリ

2014/06/14  19:41            10,791 exp_0.png
2014/06/14  19:41            37,644 exp_1.png
2014/06/14  19:41            47,821 exp_2.png
2014/06/14  19:41            49,227 exp_3.png
2014/06/14  19:41            58,843 exp_4.png
2014/06/14  19:41            61,702 exp_5.png
2014/06/14  19:41            57,757 exp_6.png
2014/06/14  19:41            68,488 exp_7.png
               8 個のファイル             392,273 バイト
               0 個のディレクトリ  52,120,539,136 バイトの空き領域
```

#確かに8個のPNGファイルが出来ている。

Matplotlib　(Object明示方式)

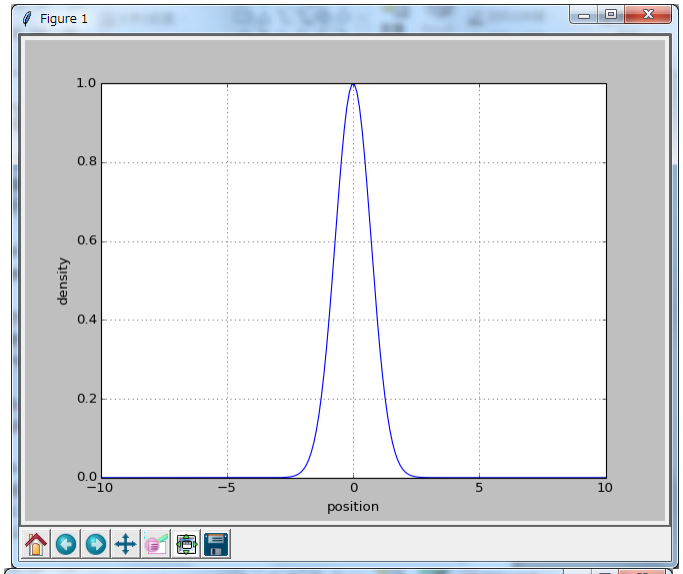
Maptplotlibはこれまで説明したようにObjectを明示しなくても利用可能であるが、
複数のグラフやLineを操作する場合は、これらのObjectを明示的に
操作する方式の方が好ましい場合も多い。

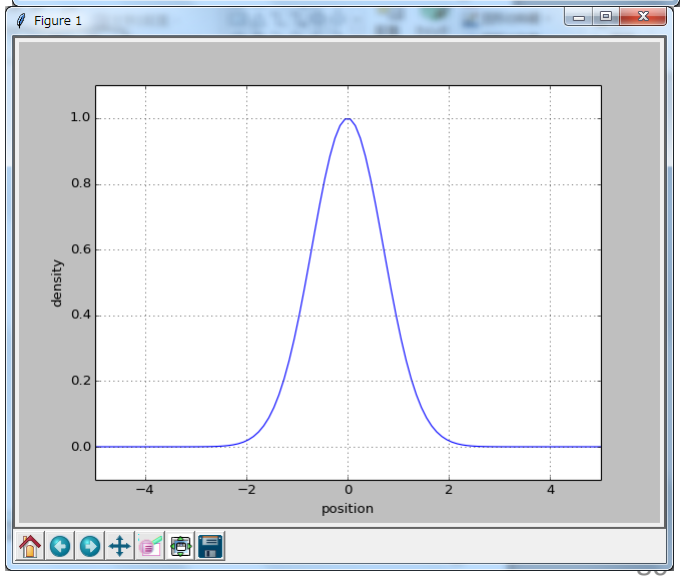
```
In [1]: X=linspace(0,2*pi,100)
In [2]: F1=gcf()   #gcf='get current figure'
In [3]: type(F1)
Out[3]: matplotlib.figure.Figure
In [4]: A1=F1.gca() #gca='get current axis'
#上を省略して単にA1=gca()でもOK
In [5]: type(A1)
Out[5]: matplotlib.axes.AxesSubplot
In [6]: L1,L2=A1.plot(X,sin(X),X,cos(X))
In [7]: type(L1)
Out[7]: matplotlib.lines.Line2D
In [9]: L3,=A1.plot(X,sin(X)+cos(X))
#カンマをつけないと、L3が右辺の戻り値そのもの(すなわちlist)となる。
In [10]: A1.legend([L1,L2,L3],['sin','cos','sin+cos'])
Out[10]: show()
```



Matplotlib (軸の設定)

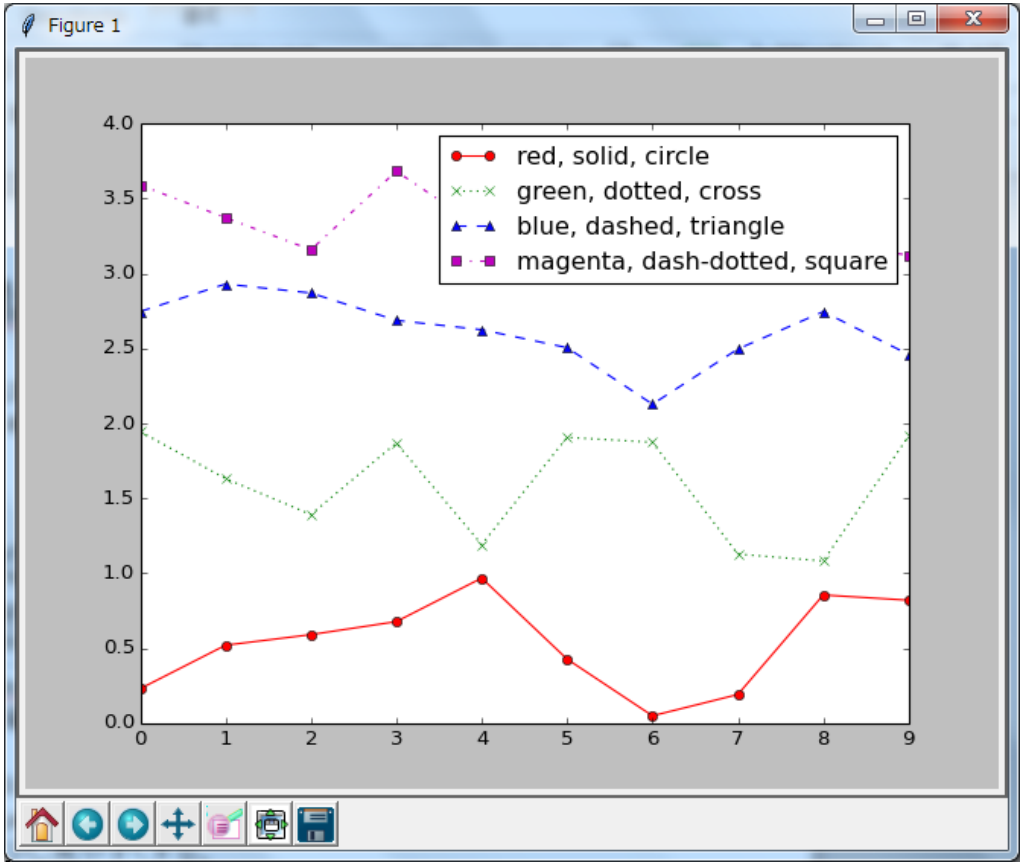
```
In [23]: X=linspace(-10,10,200)
In [24]: Y=exp(-X**2)  #Gausssian関数
In [27]: plot(X,Y)
Out[27]: 省略
In [28]: grid()
In [29]: xlabel('position')  #x軸のラベルを書く
Out[29]: 省略
In [30]: ylabel('density')  #y軸のラベルと書く
Out[30]: 省略
In [31]: xmin,xmax,ymin,ymax=axis1=axis()
#自動設定された軸の範囲を調べる
In [32]: axis1
Out[32]: (-10.0, 10.0, 0.0, 1.0)
In [33]: axis([xmin*0.5,xmax*0.5,ymin-0.1,ymax+0.1])
#軸の範囲を調整して少し見やすくする。
Out[33]: [-5.0, 5.0, -0.1, 1.1]
```





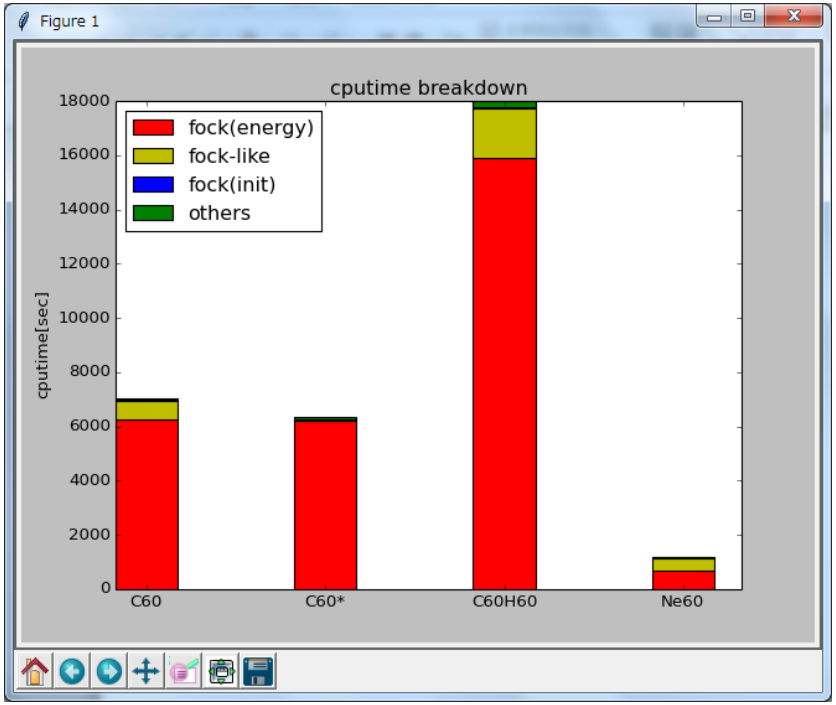
Matplotlib　(line style)

```
In [27]: from numpy.random import random
In [28]: for i,style in enumerate(['r-o','g:x','b--^','m-.s']):
   ....:     plot(random(10)+i,style)
   ....:
In [29]: legend([
   ....:'red, solid, circle',  #実線、○
   ....: 'green, dotted, cross',  #点線,X
   ....: 'blue, dashed, triangle', #破線、△
   ....: 'magenta, dash-dotted, square'])
                                  #一点鎖線、□
Out[29]: 省略
```



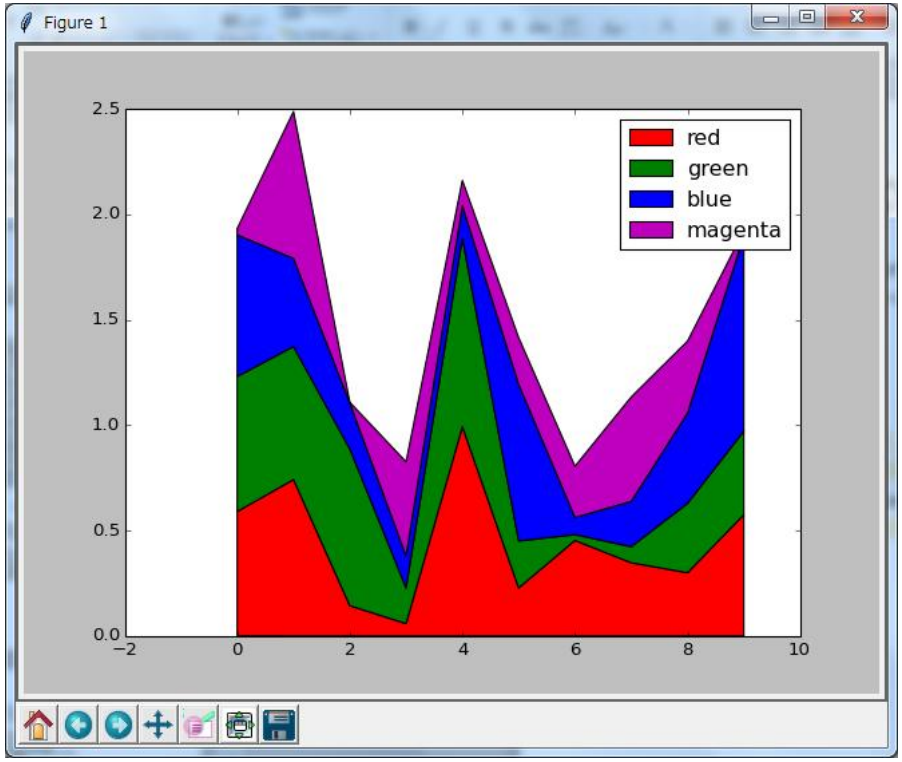
Matplotlib （棒グラフ)

```
In [25]: a1=array([
   ....: [6.24e+03, 6.23e+03, 1.59e+04, 6.82e+02],
   ....: [6.97e+02, 3.40e+00, 1.82e+03, 4.71e+02],
   ....: [4.30e+01, 3.66e+01, 8.00e+01, 7.00e+00],
   ....: [7.00e+01, 7.00e+01, 2.00e+02, 4.00e+01]])
In [26]: ind ,width= arange(4), 0.35 #x位置と幅
In [27]: a1_cum=a1.cumsum(axis=0) #累積和
In [28]: colors='r y b g'.split()  #色['r','y','b','g']
In [29]: bars = [bar(ind,a1[i], width, ¥
   ....: color=colors[i], ¥
   ....: bottom= (a1_cum[i-1] if i else None)) ¥
   ....: for i in range(4)]  #基点(=bottom)をずらした
In [30]: ylabel('cputime[sec]')  #4つの(=4色の)棒グラフを書く。(積み上げ棒グラフ)
Out[30]: 省略
In [31]: title('cputime breakdown')
Out[31]: 省略
In [32]: xticks(ind+width/2., ('C60', 'C60*', 'C60H60', 'Ne60'))
Out[32]: 省略
In [33]: legend( bars, ['fock(energy)','fock-like', 'fock(init)', 'others'],loc='best')
Out[33]: 省略
```



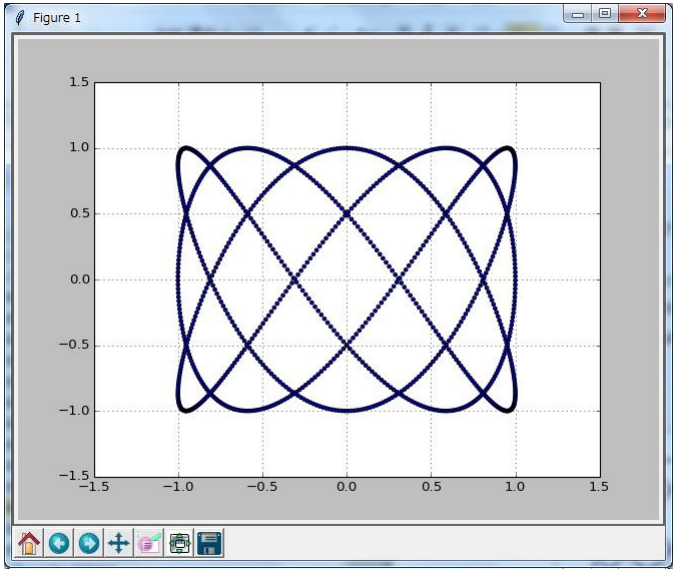
Matplotlib　(積み上げグラフ)

```
In [14]: a1=random((4,10))
In [15]: X=range(10)
In [15]: colors=list('rgbm')
In [16]: stackplot(X,a1,colors=colors)
Out[16]:  省略
In [17]: bars=[bar([0],[0],color=color) ¥
   ....: for color in colors]
#現状、stackplotに対して、
#legendがうまく機能しないようであり、
#仕方ないので、ダミーの棒グラフを書き
#棒グラフに対してlegendを付ける。
In [18]: legend(bars, ¥
'red green blue magenta'.split())
Out[18]: 省略
```

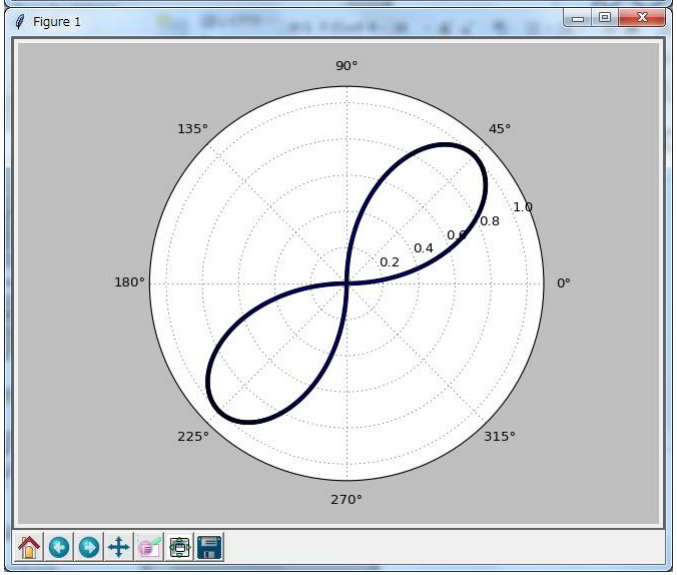


Matplotlib (散布図、極グラフ)

#直交座標での散布図(scatter graph)
In [23]: T=linspace(0,2*pi,1000) #角度
In [24]: X=sin(3*T)
In [25]: Y=cos(5*T)
In [26]: scatter(X,Y,s=10) #sは点の大きさ
Out[26]: 省略
In [27]: grid()



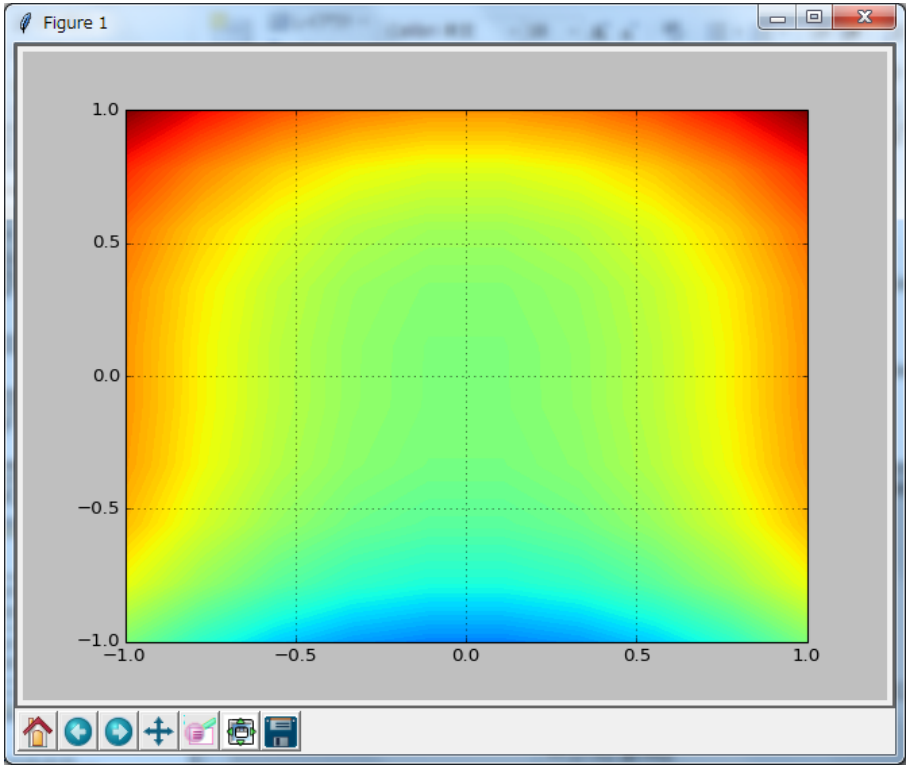
#極座標でのグラフ
In [58]: clf() #clear figure
In [60]: R=sin(2*T)
In [61]: polar(T,R)
Out[61]: 省略



Matplotlib (等高線)

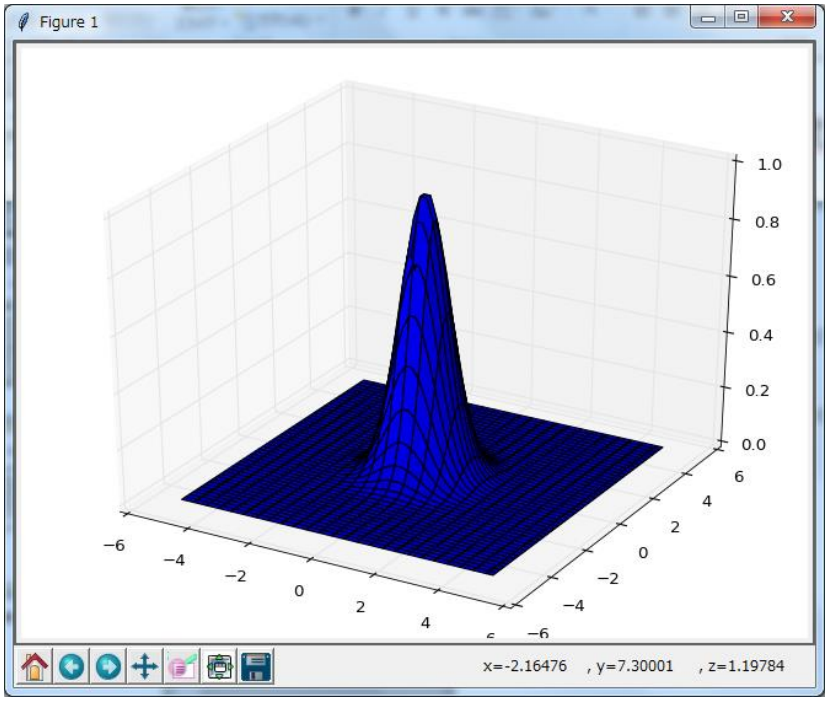
#関数 z(x,y)=x2+y**3**
#の色分けした等高線plotを行う。

In [64]: X1=Y1=linspace(-1,1,10)
In [66]: One=ones(10)
In [67]: X,Y=outer(X1,One),outer(One,Y1)
#X,Y=meshgrid(X1,Y1)としても同じ。
In [68]: Z=X**2+Y**3
#Zは2次元arrayであるから、右辺の
#X,Yも2次元arrayにしておく必要がある。
In [69]: N=linspace(-2,2,100) #等高線密度
In [70]: contourf(X,Y,Z,N)
#2個のparameter (i,j)を使った
#曲面(X[i,j], Y[i,j], Z[i,j])がplotされる
#と考える。(Z軸は色で表現される)
Out[70]: 省略
In [71]: grid()



Matplotlib　(3Dグラフ)

```
In [11]: from mpl_toolkits.mplot3d import Axes3D
#3D用に別のmoduleをimportする必要がある。
In [12]: F1=gcf()
In [13]: A1=Axes3D(F1)
In [14]: X1=Y1=linspace(-5,5,50)
In [15]: One=ones(50)
In [16]: X,Y=outer(X1,One),outer(One,Y1)
#X,Y=meshgrid(X1,Y1)としても同じ。
In [17]: Z=exp(-(X**2+Y**2))
In [19]: A1.plot_surface(X,Y,Z,rstride=1,cstride=2)
Out[19]: 省略
In [20]: A1.set_zlim(0.0,1.1)
Out[20]: (0.0, 1.1)
In [21]: draw()
```



[参考] MATLABでのplot(1)

MTLAB plotとMatplotlibの共通点、相違点を整理する。

〇Lineのプロット
plot(X1,Y1), plot(X1,Y1,X2,Y2...),plot(X1,A) は両者とも同様に動作する。
ここで、Xi,Yiは１次元配列、Aは２次元配列であり、Aの各列ベクトルが
各Lineを表す。

〇重ね書き
Matplotlib: 基本的に重ね書きされる。
　現在のグラフを消したければ、clf(), cla()を実行する。
MATLAB: hold on/off コマンドで制御する。(起動時はhold offモード)
　hold on モードでは重ね書きされる。

〇grid(), xlabel(), ylabel(), title()
　両者とも同様に動作する。

〇軸範囲の設定 : axis()は両者とも同様に動作する。
　axis() #現在の軸範囲(xmin,xmax,ymin,ymax)を取得
　axis([xmin,xmax,ymin,ymax]) #軸範囲を設定 ([...]は必要)

[参考] MATLABでのplot(2)

〇legend
Matplotlib : legend(['python','ruby'], loc='best')　#[...]が必要
MATLAB : legend('python, 'ruby') % [...]は不要

〇グラフのファイルへ出力
Matplotlib: savefig('test.png')
MATLAB:　 print(h,'-dpng','test.png')
　handle (h)が不明なら、その前に、h=gcf()で取得しておく。

〇line style
plot()の引数で、'r-o', 'g:x', 'b--^', 'm-.s' など両者とも同様に動作する。

〇棒グラフ
Matplotlib: bar(pos, heights, width, color, ...)　#pos, heights, bottomはlist
　積み上げグラフにするには、bottomを設定して、複数回plotが必要
MATLAB: bar(pos, heights, width, color, ...)
　heightsを行列にして、引数に'stackded'を入れれば、積み上げグラフになる。

[参考] MATLABでのplot(3)

〇積み上げ折れ線グラフ
Matplotlib : stackplot()を利用
MATLAB : 不明

〇散布図
scatter(X,Y,s)が両者とも同様に動作する。(sは点のサイズ)

〇極グラフ
polar(T,R)が両者とも同様に動作する。

〇等高線
contourf(X,Y,Z,N)が両者とも同様に動作する。
NumpyのX,Y=meshgrid(X1,Y1)はMATLABでは[X,Y]=meshgrid(X1,Y1)となる。
NumpyのZ=X**2+Y**3 はMATLABではをZ=X.^2+Y.^2となる。(.を忘れないこと)

[参考] pandas(1)

pandasとはnumpyの1D array, 2D arrayを元にして、さらに
data解析用に便利な機能を追加したpackageである。
numpyの1D array -->pandasのSeries型
numpyの2D array --> pandasのDataFrame型
という対応関係になっている。
pandasとはExcelの代用として 利用するものであると考えると理解しやすい。

```
In [38]: import pandas as pd  #これが定形的なimportの仕方。
In [48]: a1=np.array(range(12)).reshape(4,3)**0.5   #平方根
In [50]: df1=pd.DataFrame(a1,index=list('abcd'),columns=list('ABC'))
In [54]: df1
Out[54]: A         B         C
a  0.000000  1.000000  1.414214
b  1.732051  2.000000  2.236068
c  2.449490  2.645751  2.828427
d  3.000000  3.162278  3.316625
```

#pandasではindexとして文字列が利用できるのが大きな利点である。
#もちろん、numpy同様、整数のindexやsliceも同時に利用可能である。

[参考] pandas(2)

In [55]: df1['A']　#列Aを取得する。
Out[55]:
a　0.000000
b　1.732051
c　2.449490
d　3.000000
Name: A, dtype: float64

In [57]: df1.ix['a']　#行aを取得する。(今度は、.ixが必要である)
Out[57]:
A　0.000000
B　1.000000
C　1.414214
Name: a, dtype: float64

In [58]: type(df1),type(df1['A']),type(df1.ix['a'])　#型を調べる。
Out[58]:　#それぞれDataFrame型、Series型、Series型となる。
(pandas.core.frame.DataFrame,
 pandas.core.series.Series,
 pandas.core.series.Series)

[参考] pandas(3)

In [62]: df1.ix[['a','c'],['A','C']]　#部分配列(a,c行とA,C列)を取得する。
#一個の列を取得する場合以外は必ず、.ixが必要である。
Out[62]:

	A	C
a	0.00000	1.414214
c	2.44949	2.828427

In [63]: df1.plot()
Out[63]: 省略
#plot methodを使うと、
#文字列indexが自動的に反映された
#グラフが作成される。

#ここでは、pandasの機能のうち
#ごく一部の機能のみを紹介した。
#これ以外にも便利な機能が沢山
#あるので、詳細はマニュアル参照のこと。