Python and Statistics for Matlab Users

Session 1

Nicolas Barrier

OT-Med Labex

September 19, 2016

Why use Python instead of Matlab?

Pros

- Python is free and can be used on multiple platforms
- All the usefull things in Matlab have their Python counterpart
- Reads .mat files!
- `http://journals.ametsoc.org/doi/abs/10.1175/BAMS-D-12-00148.1`
- `http://abandonmatlab.wordpress.com/`

Cons

- No "official" IDE as in Matlab
- The set-up can be excruciating (package dependency)

The aim of the present class is . . .

To provide you the knowledge to move on to Python by the end of the week!

- Python basics (list, arrays, dictionaries, loops)
- Input/Output of files (text, netcdf and .mat)
- Graphical representations (time series, contours, maps)
- Statistics (correlation, EOFs, etc.)

Discussed from case studies!

Table of Contents

Python scripts

Python scripts are written in `.py` files. To run the script from a terminal (Linux, Mac Os X or Windows):

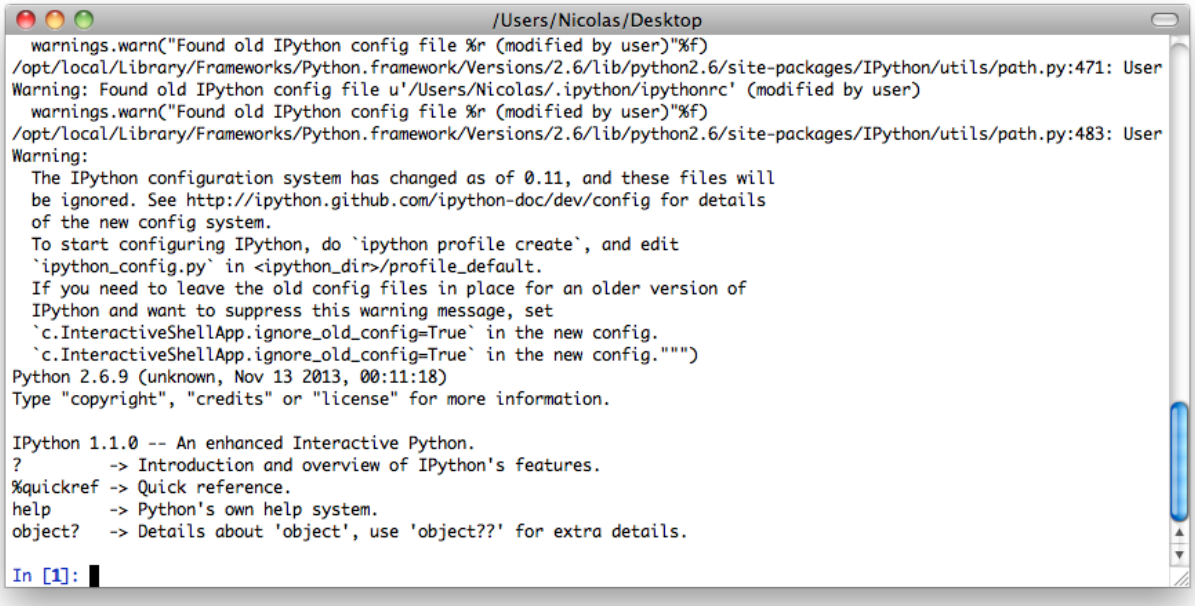
```
> python script.py arg1 arg2 arg3
```

The `python` statement is followed by the script name and the list of arguments.

Interactive Python

In a Terminal or Command Prompt:

```
> ipython  # (equivalent to matlab -nojvm)
```



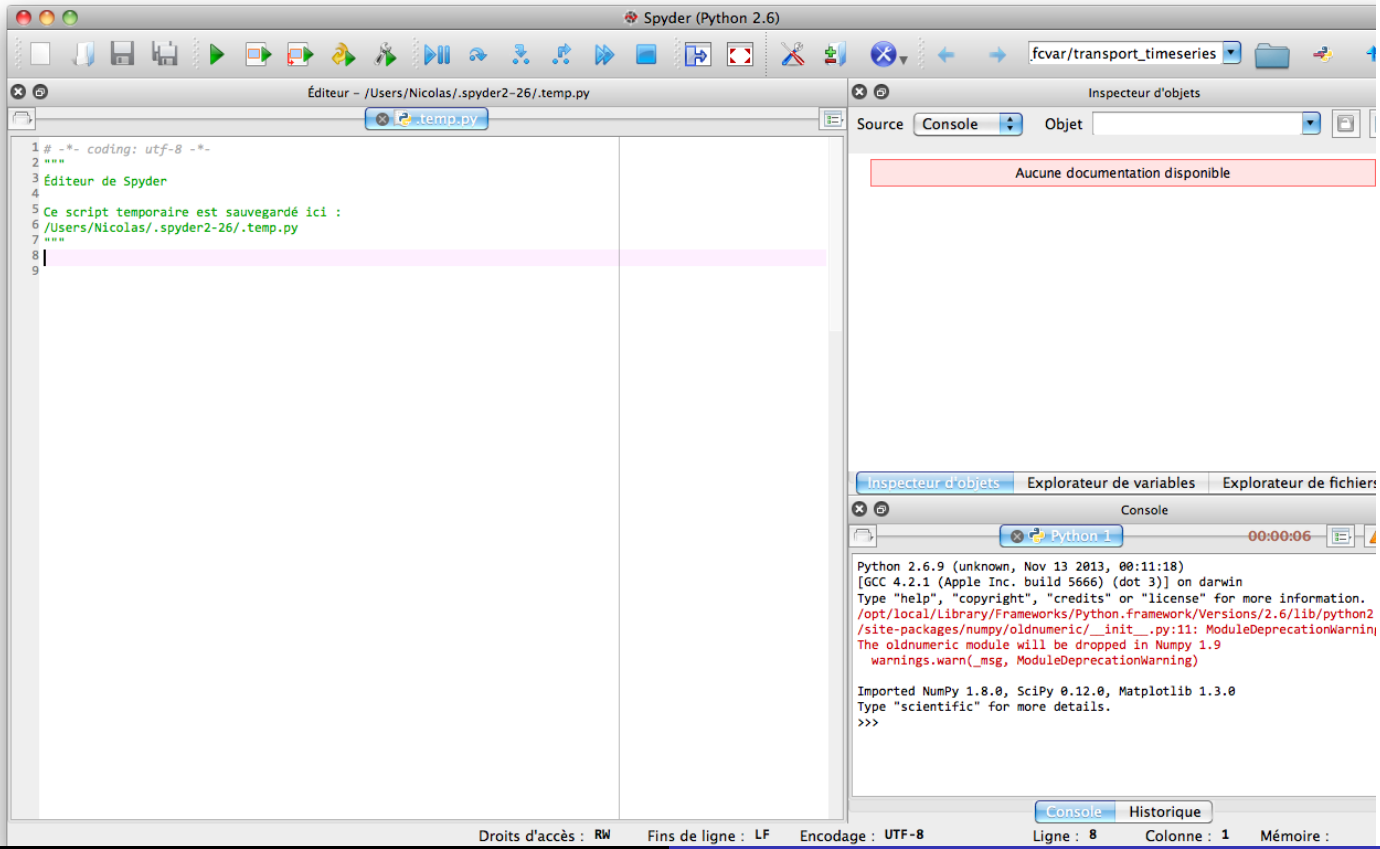
To run a script: `run script.py arg1 arg2 arg3`.

To check your variables: `whos`

IDE (Matlab like)

In a Terminal or Command Prompt:

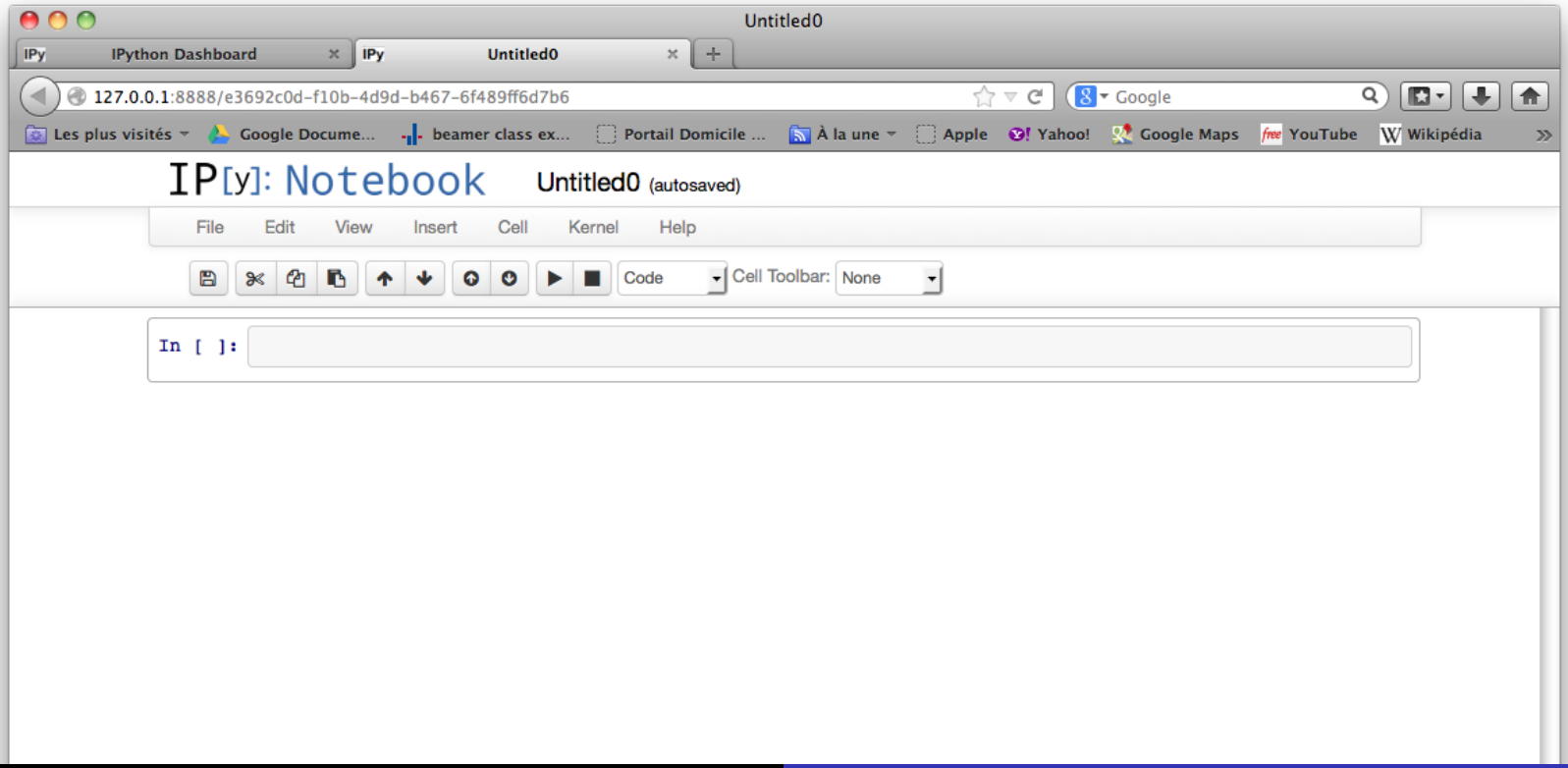
```
> spyder &
```



IDE (HTML)

In a Terminal or Command Prompt:

```
> ipython notebook &
```



Installing libraries

If you are using Anaconda:

```
> conda install library # install
> conda update library # update
```

If your library is on PyPi:

```
> pip install library # install
> pip install library --upgrade # update
```

To install from source, go to the package directory and type:

```
> python setup.py install --home=/my/directory/
```

The `setup.py` file is the library's setup file, the `install` option means that you want to install the package, and the `--home` option provides the installation directory.

Loading libraries

The first lines in a .py file are generally devoted to the loading of the libraries that are used in the program:

```
# loading the numpy library
import numpy
# loading matplotlib with the shortname mp
import matplotlib as mp
# loading the Dataset function of the netCDF4 lib.
from netCDF4 import Dataset
# loading all the functions of the scipy.stats lib.
# UNADVISED!!!!!
from scipy.stats import *

numpy.fonction(arguments) # calling numpy functions
mp.fonction(arguments)  # calling matplotlib functions
Dataset(arguments) # no netCDF4.Dataset in this case
pearson3(arguments) # no scipy.stats.pearson3 in this case
```

Loading your libraries

If you have built your own functions or changed the location of installed libraries, you must first add the directory to the path:

```
import sys
sys.path.append('/add/other/directory/')
import mylib
```

Hint

You can create the PYTHONPATH environment variable. With Mac Os X/Linux, edit your .bashrc or .cshrc file and add:

```
# bashrc
export PYTHONPATH=${PYTHONPATH}:/add/other/directory
# cshrc
setenv PYTHONPATH /add/other/directory:${PYTHONPATH}
```

Getting some help

To obtain the list of functions/methods/attributes associated with a variable or a package:

```
x=1; dir(x) # [ 'conjugate', 'denominator',...]
import sys; dir(sys) #['chartostring', 'date2index',...]
```

To get some help about a function/method:

```
help(x.conjugate)

#Help on built-in function conjugate:
#
#conjugate(...)
#    Returns self, the complex conjugate of any int.
#(END)
```

Table of Contents

List manipulation

```
x = [1] # initialisation
x.append([1, 2, 3, 4]) # add list in list [1, [1, 2, 3, 4]]
x.append('String') # add str in list [1, [1, 2, 3, 4], 'String']
len(x) # 3

x = [1]
x.extend([1, 2, 3, 4]) # add list elmt. in list [1, 1, 2, 3, 4]
x.extend('String')  # add list elmt. in list
#[1, 1, 2, 3, 4, 'S', 't', 'r', 'i', 'n', 'g']
# strings are considered as a list of characters!
len(x) # 11
```

Warning!

No algebraic operations with list!

```
x = [0, 1, 2]; y = [3, 4, 5]
x + y # [0, 1, 2, 3, 4, 5] concatenation
2*x # [0, 1, 2, 0, 1, 2]  concatenation
```

List manipulation

```
# removing list element
x = range(15, 20) # [15, 16, 17, 18, 19]
x.pop(2) # 2 = index of the elmt. to remove
x # [15, 16, 18, 19]
```

Caution when copying a list!

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = x
x[4] = 30
print y #[ 1  2  3  4 30  6  7  8  9 10]

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = []
y[:] = x
x[4] = 30
print y #[ 1  2  3  4  5  6  7  8  9 10]
```

List indexing

Different from Matlab and R (can be confusing)

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print list[0] # 1, equivalent to list(1)
print list[2:5] # [3,4,5] equivalent to list(3:5)
print list[-1] # 10 equivalent to list(end)
print list[-5:-3] # [6,7] equivalent to list(end-4:end-3)
print list[6:] # [7, 8, 9, 10] eq. to list(7:end)
print list[:3] # [1,2,3] eq. to list(1:3)
print list[2:-2:2] # [3, 5, 7] eq. to list(3:2:end-2)
print list[::4] # [1, 5, 9] eq. to list(1:4:end)
```

Table of Contents

Array creation

To do mathematics: Numpy library

```
import numpy as np

# initialisation
x = np.arange(0, 10, 1) # 0 to 9 by 1 (ints)
x = np.linspace(10, 20, 41) # 10 to 20 with 41 elements (floats)
x = np.array([0, 1, 2, 3, 4, 5]) # initialisation from list
x = np.ones((2, 3, 8), dtype=np.int) # init with 1 integers
x = np.zeros((2, 3, 8)) # init with 0 floats
x = np.empty((2, 3, 8), dtype=np.float) # init with random floats
x = x.astype(np.int) # convert into integers

type(x) # <type 'numpy.ndarray'>
x.ndim # 3: number of dimensions
x.shape # (2,3,8): size of the array
x.dtype # int64: array type
```

Mathematics with Numpy

```
# initialisation of (3,10) array
x = np.array([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
              [7, 1, 3, 5, 6, 7, 8, 10, 4, 6],
              [5, 1, 0, 3, 5, 8, 12, 5, 8, 1]])

np.mean(x,axis=0) # mean over 1st dim. (10 val)
np.std(x, axis=1) # std. deviation over 2nd dim. (3 val)
np.sum(x, axis=-1) # sum over last dim. (3 val)
np.cumsum(x, axis=-2) # cumu. sum over 2nd to last dim. (3x10)
np.prod(x, axis=0) # product along the 1st dim. (10 val)
```

Mathematics with Numpy

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([6, 7, 8, 9, 10])

z = x*y  # equivalent to matlab: x.*y
z = x/y  # (almost) equivalent to matlab: x./y
# [0 0 0 0 0]
# Python does integer division

# to overcome, change one of the variable into float
z = x/y.astype(np.float)
# [ 0.16666667 0.28571429 0.375 0.44444444 0.5]
```

Array manipulation

```
x = np.empty((2, 3, 8), dtype=np.float)
x[:, :, 0].shape, x[:, -1, ::2].shape # (2,3) (2,4) subarrays

y = np.copy(x)  # or y = x.copy(): copy x into y

# convert from 1D (10 elmt) to 2D (2 by 5)
x = np.arange(0, 10)
xresc=np.reshape(x, (2,5)) # [0. 1. 2. 3. 4.] C
xresf=np.reshape(x, (2,5), order="F") # [0. 2. 4. 6. 8.] Fortran

# convert from ND to 1D. Caution with the choice of order
xbisc=np.ravel(xresc) # [0 1 2 3 4 5 6 7 8 9]
xbisf=np.ravel(xresc, order="F") # [0 5 1 6 2 7 3 8 4 9]

xtile = np.tile(x, (2, 1)) # rep. matrix
# (2, 10): [[0 1 2 3 4 5 6 7 8 9] [0 1 2 3 4 5 6 7 8 9]]
xtile2 = np.tile(x, (1, 2)) # rep. matrix: caution with dim.!
# (1, 20): [[0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]]
```

Array manipulation

```
# changing dimension orders
x = np.ones((5, 10, 15))
xt = np.transpose(x)  # (15, 10, 5)
xt2 = np.transpose(x, (2, 0, 1)) # (15, 5, 10)

# find indexes where xresc is between 3 and 6
index = np.nonzero((xresc <= 6) & (xresc >= 3))
index # (array([0, 0, 1, 1]), array([3, 4, 0, 1]))
# cond. is met for xresc[0,3],xresc[0,4],xresc[1,0],xresc[1,1]

xresc[index] # [ 3.  4.  5.  6.]: 1D array
# eq. to xresc[[0, 0, 1, 1], [3, 4, 0, 1]]

# extraction of unique values
x = np.array([0, 0, 0, 1, 2, 2, 3, 3, 3, 3])
np.unique(x) # [0 1 2 3]
```

Array manipulation

```
x = np.zeros((2, 3, 5))
y = np.ones((2, 7, 5))
z = np.concatenate((x, y), axis=1)
# concatenation along 2nd dim. the input arrays must
# have the same dimensions, except along the cont. dimension
z.shape # (2, 10, 5)
z[0, :, 0] # [0. 0. 0. 1. 1. 1. 1. 1. 1. 1.]

x = np.arange(0, 5) # 1D array
x2d = np.atleast_2d(x) # 2D array
x2d.shape # (1, 5)

# saving of numpy objects
np.savez('save.npz', xdata=xt2, xbis=x)
# loading of numpy objects as a dict
fin = np.load('save.npz')
xt2 = fin['xdata']
x = fin['xbis']
```

Not a number (numpy.nan)

```
x = np.arange(1, 10).astype(np.float)
x[0] = np.nan
# WARNING: doesi not work for integer array

i = np.nonzero(np.isnan(x))[0] # find index of NaNs
i = np.nonzero(x!=x)[0] # same thing but using the def. of NaN
print i # [0]

print np.nanmean(x)  # 5.5
print np.nansum(x)   # 44.0
print np.nanstd(x)   # 2.29
# no np.nancumsum function !
```

Masked numpy arrays (numpy.ma)

```
x = np.ma.masked_where(np.isnan(x),x) # mask data where nan
x = np.ma.masked_equal(x,5)     # mask x when equal to 5
x = np.ma.masked_greater(x,7)   # mask x when greater than 7
x = np.ma.masked_less(x,3)      # mask x when less than 3
x[2] = np.ma.masked             # mask the 3rd element of x

type(x) # <class 'numpy.ma.core.MaskedArray'>
x  #[-- -- -- 4.0 -- 6.0 7.0 -- --]
x.mask  # [ True  True True False  True False False  True  True]
np.mean(x) # 5.66666666667 no need of np.nanmean
np.sum(x) # 17.0
np.std(x) # 1.24721912892
np.cumsum(x) # [-- -- -- 4.0 -- 10.0 17.0 -- --]
```

Masked numpy arrays

Warning!

To extract unmasked values, it is unadvised to use the `mask` attribute. It won't work if you don't have any masked data! Use the `np.ma.getmaskarray` function instead (also works for simple `numpy.arrays`)

```
x = np.ma.arange(10, 15) # [10 11 12 13 14]: no masked value

x.mask  # False
iok = np.nonzero(x.mask==0) # (array([0]),)
x[iok] # [10]

np.ma.getmaskarray(x) # [False False False False False]
iok = np.nonzero(np.ma.getmaskarray(x)==0) # (array([0,1,2,3,4]))
x[iok] # [10 11 12 13 14] GOOD!
```

Masked numpy arrays

Warning!

Masked arrays cannot be saved using the `np.savez` function. Tip: convert masked values into `np.nan` and convert into `np.array`.

```
# conversion of masked values into NaN
x[np.ma.getmaskarray(x)] = np.nan

# conversion from np.ma.array to np.array
x = np.array(x)

# saving into file
np.savez("zarray.npz", x=x)
```

Table of Contents

Loops

```
x = range(1, 11) # array from 1 to 10

for p in range(0, len(x)): # note the colon!
    print x[p] # p: index of the element
    # inside a loop, the indent must change!
# no end statement, just left indent

for temp in x:
    print temp # temp: element itself

cpt = 0 # initialisation of counter
while cpt < len(x):
    print x[cpt]
    cpt += 1 # iterate counter
```

Loops

```
# loop over multiple lists
x = range(1, 11); y = range(11, 21); z = range(21, 31)
zip(x,y,z) # [(1,11,21), (2,12,22) ... ]: list of x,y,z tuples
for xtemp, ytemp, ztemp in zip(x, y, z):
    print xtemp, ytemp, ztemp # 1 11 21; 2 12 22; ...

# Loop on 2D arrays
x = np.empty((50, 50))
for i in xrange(0, x.shape[0]):
    for j in xrange(0, x.shape[1]):
        temp = x[i, j]
```

Special loops (List Comprehensions)

Very efficient loops, which output a list:

```
combs1 = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
# [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

# equivalent to:
combs2 = []
for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if x != y:
            combs2.append((x, y))

x = range(1, 10)
y = [3, 5, 6]
# extract x values that are not in y
z = [temp for temp in x if temp not in y]
# [1, 2, 4, 7, 8, 9]
```

Table of Contents

If statements

```
x = 10
y = 9
z = 10

if ((x==y) & (x==z)):
    print 'Equality'
elif ((x<=y) & (y<=z)):
    print 'Increasing order'
elif ((x>=y) & (y>=z)):
    print 'Decreasing order'
else:
    print 'No order'

x = np.ones((10, 10))
y = np.ones((10, 10))
np.all(x == y)   # True
np.any(x != y)   # False
```

Logical expressions

Python expression	Meaning
`not(a)`	not a
`a == b`	a equal b
`a != b`	a not equal b
`a & b`	a and b
`a | b`	a or b
`a >= b`	a greater equal b
`a > b`	a greater b
`a <= b`	a less equal b
`a < b`	a less b

Table of Contents

Dictionary

Can be viewed as a list with *named* variables.

```
# dictionary creation
data = {} # empty dict.
data = {'datar':np.arange(0,10), 'datstr':'This is a dictionary'}
data['datstr'] # retrieve the data from a dict.
data['datlist'] = [0, 1, 2] # add a new element to the dict.

data.keys() # list of var names ['datar', 'datlist', 'datstr']
data.values() # list of var values
data.items() # list of (key, val) tuples

# print key, val couples
for k, v in data.items():
    print 'key, val = ', k, v
# eq. to for k, v in zip(data.keys(), data.values()):

# removing dict. entry
data.pop("datstr")
```

Dictionary concatenation

```
data2 = {'datfunc':np.mean, 'datflt':0.5}
data.update(data2)  # add the elmts of data2 into data
data.keys() # ['datfunc', 'datar', 'datflt', 'datlist']

# interesting dictionary: globals()
# contains all the global variables
globals().update(dict)
print(datar) # [0 1 2 3 4 5 6 7 8 9]
# no more dict['datar']!
```

Dictionary

Hint!

One can create something like Matlab's structure in Python using dictionaries. Structures in Python are in fact `class` objects (*detailled description on session 6*).

```
class struct(object):
    def __init__(self, data): # data = dictionary
        self.__dict__.update(data)

obj = struct({})
obj.x = np.arange(-5, 6)
obj.name = 'Structure'
print obj.name, obj.x
```

Dictionaries Input/Output

Achived using the `pickle` library.

```
import pickle

output = {'obj1':obj, 'obj2':data}

# save outputs dict
fout = open('output.pic', 'w') # open file
pickle.dump(output, fout) # put dict into file
fout.close() # close file

# open output file
pfile = open('output.pic', 'r')
dataf = pickle.load(pfile) # extract dict from file
pfile.close()
dataf['obj1']
dataf['obj2']
```