Python essentials for the Computational Finance course

The aim of the document is to discuss a minimal number of Python commands that are needed for this course. Reading Python and Scipy tutorials is highly encouraged. This document is based on Python 3.x.

1 A warning

In Python, **numbering of elements of an object (array, string, list, tuple) starts from 0**; index of the element is given in square brackets

Example:

```
a=(1,2,3); print(a[1])
```

outputs the second element of `a`, that is 2.

2 Programming constructs

It is important to remember, that Phython groups commands by indentation.

The format of `for` cycle:

```
for i in Something:
  first command
  second command
  ....
  last command of the cycle
other commands (outside for)
```

"`Something`" is usually an object that has many elements; `i` takes the value of each element of the object `Something`

Example:

```
for x in (1,10,"blue"):
    print(x)
```

The format of the `while` cycle is as follows:

```
while condition:
   first command
   second command
   ...
   last command
commands outside while
```

Example:

```
i=1
while i<5:
   print(i,'squared is',i*i)
   i=i+1
```

The format of the `if` statement is as follows:

```
if condition:
  first command
  second command
  etc
elif condition:
  first command
  etc
else:
  first command
  etc
other commands
```

There may be many `elif` (else if) parts.

Example:

```
x=input('x=')
if x==0:
   print('zero')
elif x==1:
   print('one')
elif x==2:
   print('two')
else:
   print('a strange number')
```

3 Using modules and packages of Python. Defining new functions

Using modules and packages. Python functions are organized in modules or packages (collections of modules). There are two ways to use those functions. 1) import the module with the command

```
import module_name
```

Then it is possible to use a function `func()` from the module in the form `module_name.func()`.
2) import the needed functions from the module using the command

```
from module_name import fun1,fun2,etc
```

or, to import all functions,

```
from module_name import *
```

then it is possible to use function names directly. In the case of a package the command "`from package import *`" does not import all functions from all subpackages of the package; they should be imported separately.

Examples:

```
import time
time.ctime()
from scipy import sin
sin(0.5)
from scipy import *
cos(pi)
```

Defining your own functions: the `def` command. Example:

```
def f(x,y=1,z=0):
        tmp=x*y+z
        return tmp
```

If default value for a variable is given, then it is not necessary to specify it's value: valid uses of the function are for example

```
f(5)
f(2,3)
f(2,3,4)
f(1,z=2)
f(z=2,x=0,y=1)
```

invalid uses are: `f()`- `x` does not have a default value, `f(z=2,3)` - unnamed arguments have to be before named arguments.

4 Numerical computations in Python: the package SciPy

For numerical computations in Python there is the package **scipy** (which has to be installed together with the package **numpy**). The functions in the packages can operate on arrays, that speeds up computations a lot.

4.1 Creating arrays.

`arange(start,stop,step=1)` - creates array of the elements start, start+step, start+2*step, ... which are less than stop (stop is not included)

`linspace(start,stop, num=50)` - divides the interval [start,stop] into num-1 equal subintervals (ie returns num equally spaced points including start and stop)

`zeros(shape)` - returns an array filled with zeros; the dimensions of the matrix are in the variable shape Examples:
`zeros(10)` - one-dimensional array with 10 elements;
`zeros(shape=(3,4))` - two-dimensional array with 3*4 elements

`ones(shape)` - array filled with ones

`empty(shape)` - an array with given dimension with arbitrary values

`array([[1,2],[3,4],[5,6]])` - 3*2 matrix

4.2 Accessing array elements

Elements are numbered starting from 0.
For one-dimensional array `A`:
`A[1]` gives the second element of `A`
`A[1:3]` gives second and third elements (ie `A[1]`,`A[2]`, but not `A[3]`)
`A[2:]` gives all elements starting from the third (ie `A[2]`,`A[3]` etc)
`A[:3]` gives first three elements, ie `A[0]`,`A[1]`,`A[2]`.
`A[2:-1]` gives all elements except two first and 1 last; negative index after colon indicates how many elements to leave out from the end
If `b` is an array of integers, then `A[b]` returns the elements of A which have indeces in the array `b` in the same order as they are listed in `b`
for two-dimensional array:
`A[i,j]` gives the single element,
`A[i,:]` gives the (i+1)th row,
`A[:,j]` gives the (j+1)th column,
`A[A>0]` gives all elements that are greater than 0

Examples:

```
b=arange(1,11)
A=empty(shape=(2,10))
A[0,:]=b
A[1,0:3]=2
A[1,3:]=5
print(A)
print(A[:,3])
A[A<3]=0
print(A)
z=array([5,1,4,1])
print(A[1,z])
```

WARNING: assignments like `B=A` or `b=A[:,1]` DO NOT COPY values of `A` to new matrices; in this case `B` is just another name for the entries of `A` (ie they use exactly the same values, modifying one modifies the other, too) and `b` is just a name to use the second column of `A`; `b[0]=10` sets the element `A[0,1]` to be equal to 10. If copying of values is needed, then the commands of the form `B=A.copy()` and `b=A[:,1].copy()` should be used.
Arrays can be added or subtracted elementwise, also multiplication and division works elementwise. In order to multiply arrays as matrices, one has to use the command `dot(A,B)`

4.3 Other useful array functions:

sum(A) - the sum of elements of an array;

mean(A) - the average of the elements of `A`;

std(A) - standard deviation of elements of `A`;

amin(A) - minimal value of elements of `A`;

amax(A) - maximal value of elements of `A`;

minimum(A,B) - elementwise minimum of two arrays (or an array and a number)

maximum(A,B) - elementwise maximum of two arrays

log(A) - natural logarithm of elements of `A`;

4.4 Solving systems of linear equations and minimization

For solving systems of equations of the form $Ax = b$, where A is a square matrix, x is the vector of unknowns and b is the vector on the right hand side, we import the linalg subpackage by `from scipy import linalg` and use the commands `linalg.solve(A,b)` and `linalg.solve_banded()` with suitable arguments.
For minimizing a function of several variables we import the optimize subpackage by `from scipy import optimize` and use the commands `optimize.fmin()` and `optimize.fmin_cg`.

5 Graphics and file input

For graphics the package `matplotlib` should be installed. For simple plots the following commands work:

```
from pylab import plot, show
plot(x,y)
show()
```

The `show()` command should be the last command in a script, then all results of previous plot commands are shown together. Arguments `x,y` can be 1D arrays or 2d arrays. In the last case the plots corresponding to the columns of the arrays are created. If `x` has only one column or is 1D array, then it is used with every column of `y`.

Example:

```
from scipy import *
from pylab import plot,show
n=101
x=linspace(0,1,num=n)
y=empty(shape=(n,2))
y[:,0]=cos(x)
y[:,1]=exp(-x)
plot(x,y)
show()
```

Importing data from a csv file (assuming the decimal separator is . and field separator is , and that from the package `scipy` everything is imported)

```
x=loadtxt(filename,delimiter=',', \
  usecols=seq_of_columns, skiprows=n)
```

Sequence of columns numbers (starting from 0!) in the sequence `usecols` is of the form `(a,b,c,etc)`. If the argument `usecols=` is not given, all columns will be read in; otherwise only columns indicated in the sequence are read. The parameter `skiprows` specifies the number of rows to ignore at the beginning of the file.

Examples: create a file called data.csv with the content

```
"january",2.0,-2.0,3
"february",1,0,2
"march",5,1,3
```

The command

```
x=loadtxt("data.csv", delimiter=',', \
        usecols=(3,))
print(x)
```

reads in only the 4th column; the command

```
x=loadtxt("data.csv",separator=',', \
    usecols=(1,2),skiprows=1)
print(x)
```

reads in columns number 1,2 (ie the second and the third column) and skips the first line of the file.

6 Functions related to the standard normal distribution

Here it is assumed, that the commands `from scipy import *` and `from scipy import stats` have been entered previously.

`randn(d1,d2,...,dn)` - creates a n-dimensional array filled with normally distributed random numbers

`Phi=stats.norm.cdf` - defines `Phi` as cumulative distribution function of the standard normal distribution

`invPhi=stats.norm.ppf` - defines `invPhi` to be the inverse of the cumulative distribution function of the standard normal distribution

`stats.shapiro(x)` - Shapiro-Wilk test for checking if the data in the vector `x` may be normally distributed.

`stats.anderson(x)` - Anderson-Darling test for checking if the data in the vector `x` may be normally distributed.