UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **NumPy 2/3**
- Array operations
- Reductions
- Broadcasting
- Array: shaping, reshaping, flattening, resizing, dimension changing
- Data sorting — Midterm / Project proposal due

- **NumPy 3/3**
- Type casting
- Masking data
- Organizing arrays
- Loading data files
- Dealing with polynomials
- Good coding practices

- **Scipy 1/2**
- What is Scipy?
- Working with files
- Algebraic operations
- The Fast Fourier Transform
- Signal Processing — HW4

- **Scipy 2/2**
- Interpolation
- Statistics
- Optimization

- **Project**
- Project Presentation — Final Project

Good coding practices

- Good coding practices

- use short and concise comments to describe your code when needed
- use explicit variable names so that it is easy to understand what they are
- make variable names and comments in English
- avoid changing global variables in local scope functions
- use clean style such as:
  - spaces in for loop and if statements
  - spaces after commas
  - spaces before = and after
- use the same conventions as everybody else before you
- do not hard code your variables to increase portability
- make your code readable and beautiful to look
- write a simple code
- Import / load only what you need in your workspace
- test your code!

After following these recommendations you will increase the reliability of your code!

You can read more at: https://www.python.org/dev/peps/pep-0008/#class-names

Type casting

- Type casting – *recap*
  - type casting is needed when operation between scalars or arrays of different types has to be performed
  - different types take different amount of memory so it is essential to equalize the types before any operation is performed
  - it is more desirable to deal with larger types since the amount of rounding error (quantization error in signals) is smaller
  - since truncation of longer memory occupying types (such as float128, etc.) is not an option then it turns that when two types 'clash' the longer always wins and the shorter gets converted or *type casted* to the longer type
  - larger types represent larger numbers

Data type objects

- Data type objects *– lets recall*
  - NymPy supports much larger variety of types than what the standard Python implementation does:

Number type	Data type	Description
Booleans	bool, bool8, bool_	Boolean (True or False) stored as a byte – 8 bits
Integers	byte	compatible: C char – 8 bits
	short	compatible: C short – 16 bits
	int, int0, int_	Default integer type (same as C `long`; normally either `int32` or `int64`) – 64 bits
	longlong	compatible: C long long – 64 bits
	intc	Identical to C `int` – 32 bits
	intp	Integer used for indexing (same as C `size_t`) – 64 bits
	int8	Byte (-128 to 127) – 8 bits
	int16	Integer (-32768 to 32767) – 16 bits
	int32	Integer (-2147483648 to 2147483647) – 32 bits
	int64	Integer (-9223372036854775808 to 9223372036854775807) – 64 bits
Unsigned integers	uint, uint0	Python int compatible, unsigned – 64 bits
	ubyte	compatible: C unsigned char, unsigned – 8 bits
	ushort	compatible: C unsigned short, unsigned – 16 bits
	ulonglong	compatible: C long long, unsigned – 64 bits
	uintp	large enough to fit a pointer – 64 bits
	uintc	compatible: C unsigned int – 32 bits
	uint8	Unsigned integer (0 to 255) – 8 bits
	uint16	Unsigned integer (0 to 65535) – 16 bits
	uint32	Unsigned integer (0 to 4294967295) – 32 bits
	uint64	Unsigned integer (0 to 18446744073709551615) – 64 bits

Data type objects

- Data type objects *– lets recall*
  - NymPy supports much larger variety of types than what the standard Python implementation does:

Number type	Data type	Description
Floating-point numbers	half	compatible: C short – 16 bits
	single	compatible: C float – 32 bits
	double	compatible: C double – 64 bits
	longfloat	compatible: C long float – 128 bits
	float_	Shorthand for `float64` – 64 bits
	float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
	float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
	float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
	float128	128 bits
Complex floating-point numbers	csingle	64 bits
	complex, complex_	Shorthand for `complex128` – 128 bits
	complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
	complex128	Complex number, represented by two 64-bit floats (real and imaginary components)
	complex256	two 256 bit floats

Type casting

- Type casting – *recap*

  Examples:

  - a = 3 + 4 -> creates an integer
  - b = 6 + 7. -> creates a floating point number (higher precision)

```
In [1]: a = plb.arange(5)

In [2]: a
Out[2]: array([0, 1, 2, 3, 4])

In [3]: a.dtype
Out[3]: dtype('int64')

In [4]: a = a*1.    # we force type casting by multiplying 'int' to a 'float'

In [5]: a
Out[5]: array([ 0.,  1.,  2.,  3.,  4.])

In [6]: a.dtype
Out[6]: dtype('float64')
```

Type casting

- Type casting
  - when forcing to assign a member in an array object with a different type, the assignment is completed, but no type conversion is performed on the array
  - as a result the newly assigned value may be truncated if assigning from a float to an int for ex.

```
In [22]: g
Out[22]: array([ 1.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])

In [23]: g.dtype
Out[23]: dtype('float64')

In [24]: g[4] = 10    # lets try to asing an 'int' to position 5

In [25]: g     # the type is not changed regardless of the mis-type assignment
Out[25]: array([  1.,   2.,   2.,   2.,  10.,   2.,   2.,   2.,   2.,   2.])

In [26]: g = g.astype(int)   # lets type cast 'g' to 'int'

In [27]: g.dtype
Out[27]: dtype('int64')

In [28]: g
Out[28]: array([ 1,  2,  2,  2, 10,  2,  2,  2,  2,  2])

In [29]: g[4] = 10.5  # lets try to asing a 'float64' to position 5

In [30]: g     # the type is not changed regardless of the mis-type assignment
Out[30]: array([ 1,  2,  2,  2, 10,  2,  2,  2,  2,  2])
```

Rounding

- Rounding
  - rounding with around():

    all numbers having digits after decimal point from x.0-x.4 are rounded to the lower number
    all numbers having digits after decimal point from x.5-x.9 are rounded to the higher number

```
In [7]: b = plb.arange(1,2,0.1)

In [8]: b
Out[8]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])

In [9]: c = plb.around(b)   # round all in 'b' and assign it to 'c'

In [10]: c
Out[10]: array([ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.])

In [11]: c.dtype
Out[11]: dtype('float64')

In [12]: c = c.astype(int)   # type casting to 'int'

In [13]: c.dtype
Out[13]: dtype('int64')
```

Rounding

- Rounding

  – rounding with floor() and ceil():

    sometimes there is a need for an alternative way of rounding such as:

    - floor: rounds to the lower number
    - ceil: rounds to the higher number

```
In [14]: d = plb.arange(1,2,0.1)

In [15]: d
Out[15]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])

In [16]: e = plb.around(d)   # normal rounding around mid-point '0.5' as expected

In [17]: e
Out[17]: array([ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.])

In [18]: f = plb.floor(d)    # rounds all to the lower number

In [19]: f
Out[19]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

In [20]: g = plb.ceil(d)     # rounds all to the higher number

In [21]: g
Out[21]: array([ 1.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
```

Type info

- Type info – *for NumPy variables only*
  - to check the type of a scalar or an array we use:

    <scalar>.dtype   or   <array>.dtype

  - to check the type of a scalar or an array element we use:

    plb.dtype(<scalar>) or   plb.dtype(<array[element]>)

  - to check how many bytes a type takes we use:

    plb.<type>().itemsize

  - to check how many bits a type takes we use:

    (plb.dtype(plb.<type>).itemsize)*8

  - to check the minimum, maximum values and the int type of a scalar variable or array element we use:

    plb.iinfo(plb.<type>)

Type info

- Type info – *for NumPy variables only* – Examples:
  - to check the type of a scalar or an array we use:

```
h = plb.int32(12)

h.dtype
dtype('int32')
```

or

```
i = plb.arange(4)

i.dtype
dtype('int64')
```

  - to check the type of a scalar or an array element we use:

```
plb.dtype(h)
dtype('int32')
```

or

```
plb.dtype(i[2])
dtype('int64')
```

  - to check how many bytes a type takes we use:

```
plb.int64().itemsize
8
```

  - to check how many bits a type takes we use:

```
(plb.dtype(plb.int64()).itemsize)*8
64
```

  - to check the minimum, maximum values and the int type of a scalar variable or array element we use:

```
plb.iinfo(plb.int64())
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Type info

- Type info – *for NumPy variables only* – Examples:

  – to check the machine limits for floating point types namely resolution, minimum, maximum values, type we use:

```
plb.finfo(plb.float32())
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

  or we can use:

```
plb.finfo(plb.float32()).eps
1.1920929e-07
```

  where, the eps attribute shows the smallest representable positive number and is a float

  – we can also check for the speed of certain computations using different types:

```
In [31]: i = plb.arange(0,5,1)

In [32]: i
Out[32]: array([0, 1, 2, 3, 4])

In [33]: i.dtype
Out[33]: dtype('int64')

In [34]: %timeit i**6
100000 loops, best of 3: 2.81 µs per loop

In [35]: i = i.astype(plb.float64)

In [36]: i
Out[36]: array([ 0.,  1.,  2.,  3.,  4.])

In [37]: %timeit i**6
100000 loops, best of 3: 4.41 µs per loop
```

Type info

- Type info – *for NumPy variables only* – Examples:
  - we can also check for the speed of execution for larger pieces of code similar to the tic – toc functionality in Matlab:

```
Python

In [23]: # way to measure time laps in seconds:

In [24]: import time

In [25]: start_time = time.time()

In [26]: start_time
Out[26]: 1447704378.69137

In [27]: end_time = time.time()

In [28]: end_time
Out[28]: 1447704386.440553

In [29]: total_time = end_time - start_time

In [30]: total_time
Out[30]: 7.749182939529419

In [31]: print("start time is %s sec earlier than end time" % round(total_time,3))
start time is 7.749 sec earlier than end time
```

Masking data

- Masking data
  - it is sometimes necessary to only observe certain part of a large data set
  - in this way, one can exclude unwanted values like NaN or negative numbers
  - for that reason it is useful to mask the portion of the array, which is unwanted

```
In [38]: j = plb.array([12, 34, 41, 52]) # array

In [39]: k = plb.array([1, 0, 0, 1]) # creates a mask

In [40]: l = plb.ma.array(j, mask=k) # creates an array with masked values

In [41]: l[:]
Out[41]:
masked_array(data = [-- 34 41 --],
             mask = [ True False False  True],
       fill_value = 999999)

In [42]: l[0]
Out[42]: masked

In [43]: l[1]
Out[43]: 34

In [44]: l.mask[0]=False     # the mask for each array element can be changed

In [45]: l[0]
Out[45]: 12
```

Organizing arrays

- Organizing arrays
  - larger data has to be well organized
  - all fields need to have an appropriate description

```
Python

In [46]: m = plb.zeros((3,), dtype=[('Store:','S4'),('count',int),('location',float)])

In [47]: m
Out[47]:
array([(b'', 0, 0.0), (b'', 0, 0.0), (b'', 0, 0.0)],
      dtype=[('Store:', 'S4'), ('count', '<i8'), ('location', '<f8')])

In [48]: m.dtype.names # to see only the name fields of each element
Out[48]: ('Store:', 'count', 'location')

In [49]: m['Store:'] = ['East','West','North'] # assign an array of store names

In [50]: m['count']=plb.arange(1,4,1) # assign a sequence of numbers

In [51]: # individual field indexing will access and asign each individual location:

In [52]: m[0]['location']=43.7896;  m[1]['location']=64.4321;  m[2]['location']=87.2315

In [53]: m[1]
Out[53]: (b'West', 2, 64.4321)

In [54]: m['location']
Out[54]: array([ 43.7896,  64.4321,  87.2315])

In [55]: m[1]['location'] = 5   # asigning an 'int' to a 'float' it will be recorded

In [56]: m[1]['count'] = 2.345  # assigning a 'float' to an 'int' it will be truncated

In [57]: m
Out[57]:
array([(b'East', 1, 43.7896), (b'West', 2, 5.0), (b'Nort', 3, 87.2315)],
      dtype=[('Store:', 'S4'), ('count', '<i8'), ('location', '<f8')])
```

Loading data files

- Loading data files - text

  - most of the time it is necessary to load large sets of pre-calculated data for analysis
  - sometimes the data is provided through databases, but it is also common to provide data by using files
  - the most common data type that can be loaded is simple text data from large tables containing different calculations
  - here is a sample text file format we will load:

```
# sample  train  test  Total
1  480  319  799
2  584  389  973
3  572  394  966
4  590  392  982
5  585  390  975
```

Loading data files

- Loading data files - text

```
In [58]: n = plb.loadtxt('files/lecture7/lecture7_data1.txt') # lets load some data

In [59]: n
Out[59]:
array([[   1.,  480.,  319.,  799.],
       [   2.,  584.,  389.,  973.],
       [   3.,  572.,  394.,  966.],
       [   4.,  590.,  392.,  982.],
       [   5.,  585.,  390.,  975.]])

In [60]: n.shape     # now we need to check the shape of the two arrays
Out[60]: (5, 4)

In [61]: p = sum(n[:]) # lets take the sum of all columns

In [62]: p
Out[62]: array([   15.,  2811.,  1884.,  4695.])

In [63]: p[0] = 6      # we change element 1 to equal 6

In [64]: p
Out[64]: array([    6.,  2811.,  1884.,  4695.])

In [65]: p.shape
Out[65]: (4,)

In [66]: p = p.reshape(1,4)  # we need to change the shape of 'p' to match 'n'

In [67]: p.shape
Out[67]: (1, 4)
```

Loading data files

- Loading data files - text

```
In [68]: q = plb.append(n,p,axis=0)  # now we append array 'p' at the bottom of 'n'

In [69]: n
Out[69]:
array([[   1.,  480.,  319.,  799.],
       [   2.,  584.,  389.,  973.],
       [   3.,  572.,  394.,  966.],
       [   4.,  590.,  392.,  982.],
       [   5.,  585.,  390.,  975.]])

In [70]: p
Out[70]: array([[    6.,  2811.,  1884.,  4695.]])

In [71]: q
Out[71]:
array([[  1.00000000e+00,   4.80000000e+02,   3.19000000e+02,
          7.99000000e+02],
       [  2.00000000e+00,   5.84000000e+02,   3.89000000e+02,
          9.73000000e+02],
       [  3.00000000e+00,   5.72000000e+02,   3.94000000e+02,
          9.66000000e+02],
       [  4.00000000e+00,   5.90000000e+02,   3.92000000e+02,
          9.82000000e+02],
       [  5.00000000e+00,   5.85000000e+02,   3.90000000e+02,
          9.75000000e+02],
       [  6.00000000e+00,   2.81100000e+03,   1.88400000e+03,
          4.69500000e+03]])
```

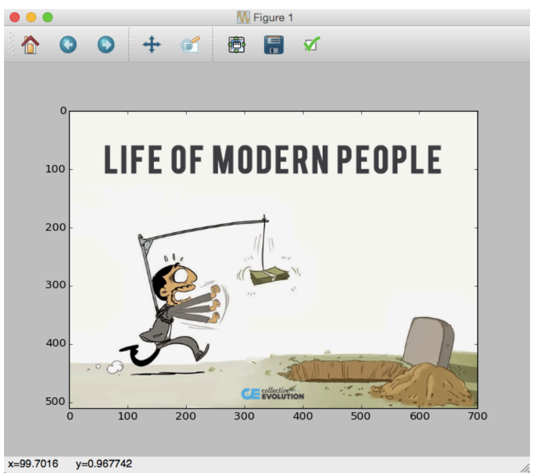
Loading data files

- Loading data files - text

```
In [72]: q = q.astype(int)   # before we write to a file we type cast to 'int'

In [73]: q
Out[73]:
array([[   1,  480,  319,  799],
       [   2,  584,  389,  973],
       [   3,  572,  394,  966],
       [   4,  590,  392,  982],
       [   5,  585,  390,  975],
       [   6, 2811, 1884, 4695]])

In [74]: plb.savetxt('files/lecture7/lecture7_data2.txt',q)    # save the new data

In [75]: r = plb.loadtxt('files/lecture7/lecture7_data2.txt')  # load the new data

In [76]: r.astype(int)
Out[76]:
array([[   1,  480,  319,  799],
       [   2,  584,  389,  973],
       [   3,  572,  394,  966],
       [   4,  590,  392,  982],
       [   5,  585,  390,  975],
       [   6, 2811, 1884, 4695]])
```

Loading data files

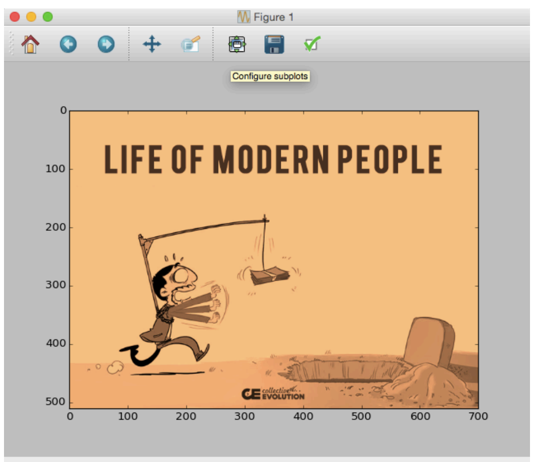
- Loading data files – images
  - you can load images, show, manipulate and save them back to files




```
Python

In [77]: s = plb.imread('files/lecture7/test1.png') # lets read a '.png' image

In [78]: plb.imshow(s) # now show it
Out[78]: <matplotlib.image.AxesImage at 0x113718780>

In [79]: s.dtype
Out[79]: dtype('float32')

In [80]: s.shape
Out[80]: (511, 700, 3)

In [81]: plb.imsave('files/lecture7/test1_retro.png', s[:,:,0], cmap=plb.cm.copper)

In [82]: plb.savefig('files/lecture7/test1_cp.png') # saves with x,y scales

In [83]: t = plb.imread('files/lecture7/test1_retro.png') # load the changed image

In [84]: plb.imshow(t) # .. and show it
Out[84]: <matplotlib.image.AxesImage at 0x1137149e8>
```

Loading data files

- Loading data files – images
  - you can load images, show, manipulate and save them back to files




```
In [77]: s = plb.imread('files/lecture7/test1.png') # lets read a '.png' image

In [78]: plb.imshow(s) # now show it
Out[78]: <matplotlib.image.AxesImage at 0x113718780>

In [79]: s.dtype
Out[79]: dtype('float32')

In [80]: s.shape
Out[80]: (511, 700, 3)

In [81]: plb.imsave('files/lecture7/test1_retro.png', s[:,:,0], cmap=plb.cm.copper)

In [82]: plb.savefig('files/lecture7/test1_cp.png') # saves with x,y scales

In [83]: t = plb.imread('files/lecture7/test1_retro.png') # load the changed image

In [84]: plb.imshow(t) # .. and show it
Out[84]: <matplotlib.image.AxesImage at 0x1137149e8>
```

Note: we only saved one RGB channel '0'

Loading data files

- Loading data files – NumPy data format
  - NumPy has its own data file writing format that efficiently stores and executed large data sets
  - … we will load sounds with SciPy later on

```
In [85]: u = plb.random((5,5)) # create an array of random numbers with size [5x5]

In [86]: u
Out[86]:
array([[ 0.99279631,  0.65777473,  0.02928297,  0.33453204,  0.71716816],
       [ 0.10242781,  0.81218654,  0.50409659,  0.1378117 ,  0.18494022],
       [ 0.29910582,  0.68431133,  0.80678923,  0.59830075,  0.25591226],
       [ 0.1488329 ,  0.89247252,  0.65925117,  0.8750453 ,  0.53301356],
       [ 0.65018547,  0.49480651,  0.78618737,  0.61346284,  0.25629997]])

In [87]: plb.save('files/lecture7/test_npdata.npy',u)

In [88]: v = plb.load('files/lecture7/test_npdata.npy')

In [89]: v
Out[89]:
array([[ 0.99279631,  0.65777473,  0.02928297,  0.33453204,  0.71716816],
       [ 0.10242781,  0.81218654,  0.50409659,  0.1378117 ,  0.18494022],
       [ 0.29910582,  0.68431133,  0.80678923,  0.59830075,  0.25591226],
       [ 0.1488329 ,  0.89247252,  0.65925117,  0.8750453 ,  0.53301356],
       [ 0.65018547,  0.49480651,  0.78618737,  0.61346284,  0.25629997]])
```

Loading data files

- Loading data – from databases

  – Python allows you to obtain data from an SQL Server

  – to install on your terminal do:

    >>> brew update

    >>> brew install unixodbc

  – to use type:

    >>> ipython   # -> on your terminal or use any IDE (Pyzo) directly:

```
import pyodbc
import pandas.io.sql as psql
```

```
cnxn = pyodbc.connect('DRIVER={SQL Server};SERVER=server_name;
        DATABASE=database_name;UID=USER;trusted_connection=true')
cursor = cnxn.cursor()
sql = "SELECT * from dbo.test_in_class”
df = psql.read_sql_query(sql, cnxn)
cnxn.close()
print(df)        -> it will contain the data read from the database ‘test_in_class’
```

Loading data files

- Loading data – from PDF files
  - Try using: pdfrw

```
## 4. Reading and writting PDFs:
from pdfrw import PdfReader, PdfWriter
# Reading pdfs:
x = PdfReader('files/lecture7/sample.pdf')
x.keys()
x.Info
x.Info.Producer
x.Root.Pages.Count
x.pages[0].Contents # returns a dictionary
x.pages[0].Contents.Length

# Writing pdfs:
y = PdfWriter()
y.addpage(x.pages[0])
y.write('files/lecture7/result1.pdf')
```

Loading data files

- Loading data – from PDF files
  - Try using: pdfrw

```
In [111]: x.keys()
Out[111]: ['/Size', '/Info', '/ID', '/Root']

In [112]: x.Info
Out[112]:
{'/AAPL:Keywords': [],
 '/CreationDate': "(D:20170508212347Z00'00')",
 '/Creator': '(Word)',
 '/Keywords': '()',
 '/ModDate': "(D:20170508212347Z00'00')",
 '/Producer': '(Mac OS X 10.11.6 Quartz PDFContext)',
 '/Title': '(Microsoft Word - Document1)'}

In [113]: x.Info.Producer
Out[113]: '(Mac OS X 10.11.6 Quartz PDFContext)'

In [114]: x.Root.Pages.Count
Out[114]: '1'

In [115]: x.pages[0].Contents
Out[115]: {'/Filter': '/FlateDecode', '/Length': '673'}

In [116]: x.pages[0].Contents.Length
Out[116]: '673'
```

Loading data files

- Loading data – from PDF files
  - Try using: pdfrw

    Example: merge / append files

```
## 4.1 Write and append paged to pdfs:
import os
from pdfrw import PdfReader, PdfWriter

writer = PdfWriter()

files = [x for x in os.listdir('files/lecture7/') if x.endswith('.pdf')]
for fname in sorted(files):
    writer.addpages(PdfReader(os.path.join('files/lecture7/', fname)).pages)

writer.write("files/lecture7/result2.pdf")
```

Loading data files

- Loading data – from Excel
  - Try using: pandas

```
## Reading .csv files:
import pandas as pd

test = pd.read_csv('Titanic.csv')
test.head(5)                   # reads the fiorst 5 lines
rows = len(test)               # counts the number of rows in the file
shape = test.shape             # shows the shape
columns = (test.columns)       # shows the column titles
survived = test.loc[test["Survived"] == 1]      # selction based on criteri
non_survived = test.loc[test["Survived"] == 0]

survived.count.__call__()[1]          # read information based on certain set criteria
non_survived.count.__call__()[1]      # read information based on certain set criteria
non_survived.count.__call__()[1] + survived.count.__call__()[1] # total number

female = test.loc[test["Sex"] == "female"]
female_survived = female.loc[female["Survived"] == 1]

female.count.__call__()[0]          # access information
female_survived.count.__call__()[0] # access information
```

Loading data files

- Loading data – from Excel
  - Try using: xlrd, xlwt, xlsxwriter

Xlwt - generate spreadsheet files compatible with Microsoft Excel versions 95 to 2003
Xlrd - This package is for reading data and formatting information from older Excel
Openpyxl - for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files

```
# begin reading '.xlsx' file
import openpyxl
import os

# begin reading '.xls' file
import xlrd      # for reading
import xlwt      # for writting

# read all fields of interest specified in file named 'list of indicators.xlsx':
wb = openpyxl.load_workbook('list of indicators.xlsx')
all_sheets = (wb.sheetnames)
sheet = wb.get_sheet_by_name(all_sheets[0]) # all_sheets[0] = "ФИН. ПОКАЗАТЕЛИ"
```

Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using roots:

will solve a polynomial with coefficients [n] represented in p[n]:
p[1] * x**(n-1) + ... + p[n-1]*x + p[n]

Example: find the roots of a polynomial with n = 4 coefficients [3, 5, -6, 8] :

$$3 * x^3 + 5 * x^2 - 6 * x + 8$$

solution:

```
In [90]: plb.roots([3,5,-6,8]) # find the roots with coefficients n = 4
Out[90]:
array([-2.74773668+0.j        ,  0.54053500+0.8236004j,
        0.54053500-0.8236004j])
```

Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using poly:

will find the coefficients of a polynomial with given roots:

c[0] * x**(n) + c[1] * x**(n-1) + ... + c[n-1] * x + c[n] where c[0] = 1 always

Example: find the coefficients of polynomial with n = 3 roots [5,-2,6] :

solution:

```
In [91]: plb.poly([5,-2,6]) # find the coefficients with roots N=3, [5,-2,6]
Out[91]: array([ 1, -9,  8, 60])
```

the polynomial looks like this:

$$1 * z^3 - 9 * z^2 + 8 * z + 60$$

Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using roots and poly:

Example:

```
In [92]: plb.poly([3,3,3]) # find the coefficients with roots N=3, [3,3,3]
Out[92]: array([  1,  -9,  27, -27])

In [93]: w = plb.roots([ 1, -9, 27, -27]) # find the roots with coefficients n = 4, [1,-9,27,-27]

In [94]: w
Out[94]:
array([ 3.00001941 +0.00000000e+00j,  2.99999029 +1.68133499e-05j,
        2.99999029 -1.68133499e-05j])

In [95]: w = plb.real(w) # select only the real part

In [96]: w
Out[96]: array([ 3.00001941,  2.99999029,  2.99999029])

In [97]: w = plb.round_(w)   # round the numbers

In [98]: w
Out[98]: array([ 3.,  3.,  3.])

In [99]: w = w.astype(int)   # type cast as 'int'

In [100]: w
Out[100]: array([3, 3, 3])
```

Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using polyfit: to find the least squares polynomial fit:

    p(x) = p[0] * x**deg + ... + p[deg]

    where *deg* is the degree of the fitting polynomial to points (x, y)
    the solution is a vector with coefficients p, which minimizes the squared error (Least Square Error)

  - The minimization of the square error in equations:

    x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
    x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1] -> more equations than unknowns
    ...
    x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]

    is given by:

    $$E_{LS} = \sum_{i=0}^{m} |p(x_i) - y_i|^2$$

    where E_{LS} is the Least Square Error and it minimizes the sum of all square residuals

Dealing with polynomials

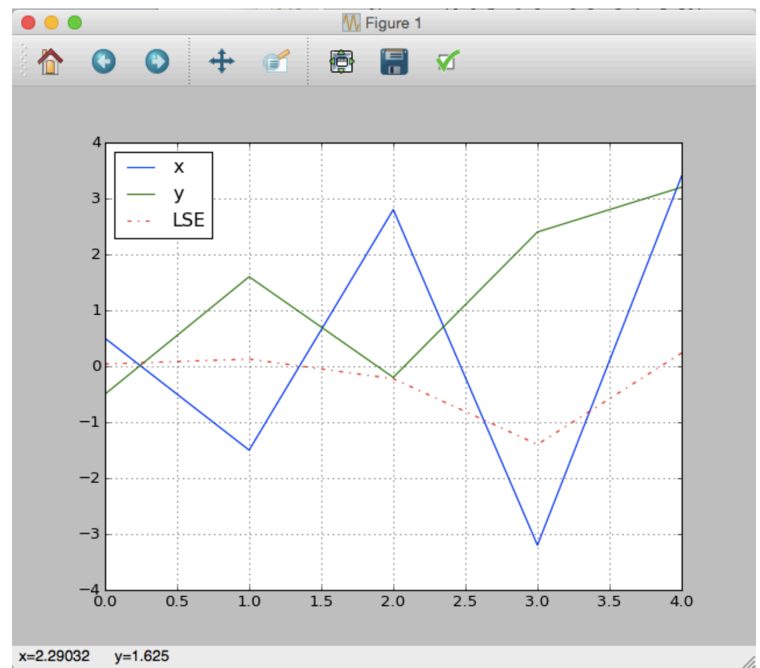
- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using polyfit for least squares polynomial fit:

p(x) = p[0] * x**n+ ... + p[n] where n is the degree of the fitting polynomial to points *(x, y)*

The solution is a vector of coefficients p that minimizes the squared error

Example:

```
# Using polyfit - least squares polynomial fit:
x = plb.array([0.5, -1.5, 2.8, -3.2,  3.4])
y = plb.array([-0.5, 1.6, -0.2, 2.4, 3.2])
z = plb.polyfit(x, y, 4)
z
plb.plot(x, color='b', label='x')
plb.grid(True)
plb.hold(True)
plb.plot(y, color='g', label='y')
plb.plot(z, linestyle='-.', color='r', label='LSE')
plb.legend(loc='upper left')
```



Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using polyval to evaluate polynomial at a specific point:

p[0]*x**(n-1) + p[1]*x**(n-2) + ... + p[n-2]*x + p[n-1]

where: - p is a 1D array of polynomial coefficients,
- p is of length n
- when x is a sequence, p(x) is returned for each element of x

Example:

$$2 * 9^3 - 5 * 9^2 + 6 * 9 - 4$$

```
In [101]: plb.polyval([2,-5,6,-4], 9)  # this represents: 2*9**3 - 5*9**2 + 6*9**1 - 4
Out[101]: 1103

In [102]: plb.polyval([2,-5,6,-4], plb.poly1d(9)) # poly1d is 1-dimenssional polynomial
Out[102]: poly1d([ 1103.])

In [103]: plb.polyval(plb.poly1d([2,-5,6,-4]), plb.poly1d(9))
Out[103]: poly1d([ 1103.])
```

Dealing with polynomials

- Dealing with polynomials
  - NumPy has several ways of solving polynomials
  - using poly1d is a one dimensional polynomial class. Two main usages:

    1/ poly1d([3,-4,7]) represents $3 * x^2 - 4 * x + 7$

    2/ poly1d([3,-4,7], True) represents $(x-3)(x+4)(x-7) = x^3 - 6x^2 - 19x + 84$

  - we can perform the following operations on polynomials:
    - » addition
    - » subtraction
    - » multiplication
    - » division
    - » gradation

Dealing with polynomials

- Dealing with polynomials

1/ poly1d([3,-4,7]) represents $3 * x^2 - 4 * x + 7$

2/ poly1d([3,-4,7], True) represents $(x-3)(x+4)(x-7) = x^3 - 6x^2 - 19x + 84$

Example:

```
In [103]: a = plb.poly1d([3,-4,7]) # represents: 3*x**2 - 4*x + 7

In [104]: print(a)    # construct the polynomial
   2
3 x - 4 x + 7

In [105]: b = plb.poly1d([3,-4,7], True)   # represents: (x-3)(x+4)(x-7)

In [106]: print(b)    # construct the polynomial
   3     2
1 x - 6 x - 19 x + 84

In [107]: b.r         # show the roots of the polynomial
Out[107]: array([-4.,  7.,  3.])

In [108]: b.c         # show the coefficients of the polynomial
Out[108]: array([  1,  -6, -19,  84])

In [109]: b.order     # show the order of the polynomial
Out[109]: 3

In [110]: a(8)        # evaluate the polynomial at point x=8
Out[110]: 167

In [111]: b(8)        # evaluate the polynomial at point x=8
Out[111]: 60
```

Dealing with polynomials

- Dealing with polynomials
  - we can change the variable of the polynomial
  - we can perform the following operations on polynomials:
    - » addition
    - » subtraction
    - » multiplication
    - » division
    - » gradation

Example:

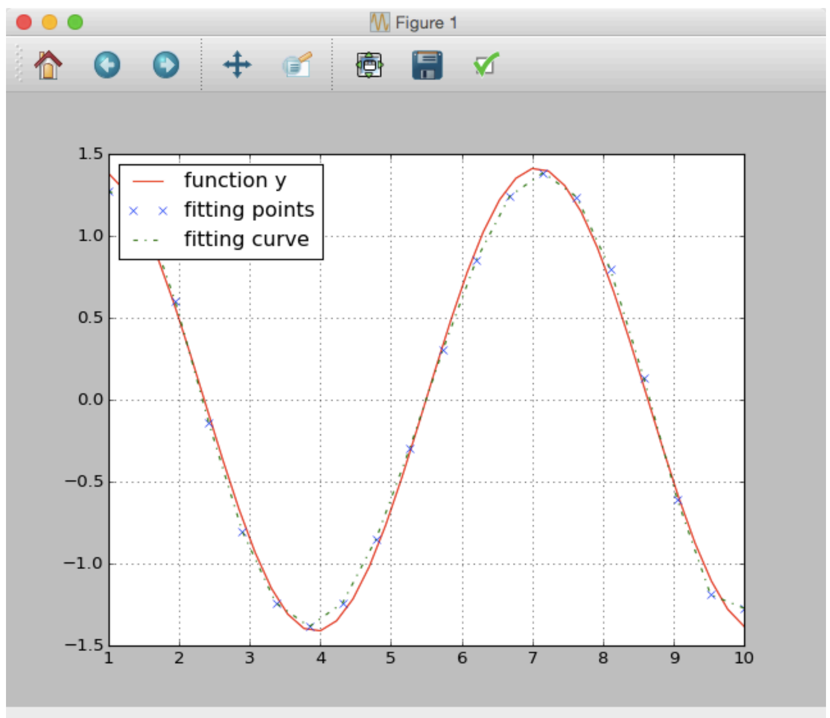
```
Python

In [112]: c = plb.poly1d([3,-4,7], variable='y') # we can select the variable to be 'y'

In [113]: print(c)
   2
3 y - 4 y + 7

In [114]: (c*c-3)/4+c**2  # addition, subtraction, multiplication, division and gradation
Out[114]: poly1d([ 11.25, -30.   ,  72.5 , -70.   ,  60.5 ])
```

Dealing with polynomials

- Dealing with polynomials

Example:

```python
x = plb.linspace(1, 10, 40) # create vector 'x' with 40 numbers between [1:10]
y = plb.cos(x) + plb.sin(x) # create and calculate function 'y'
p = plb.poly1d(plb.polyfit(x, y, 5))    # find the LSE with 5 degrees fitting
t = plb.linspace(1, 10, 20)
plb.plot(x, y, color='r', label='function y')   # plot with some specifics
plb.plot(t, p(t), 'x', label='fitting points')
plb.plot(t, p(t), '-.', label='fitting curve')
plb.grid(True)
plb.legend(loc='upper left')    # add the legend will labels
```



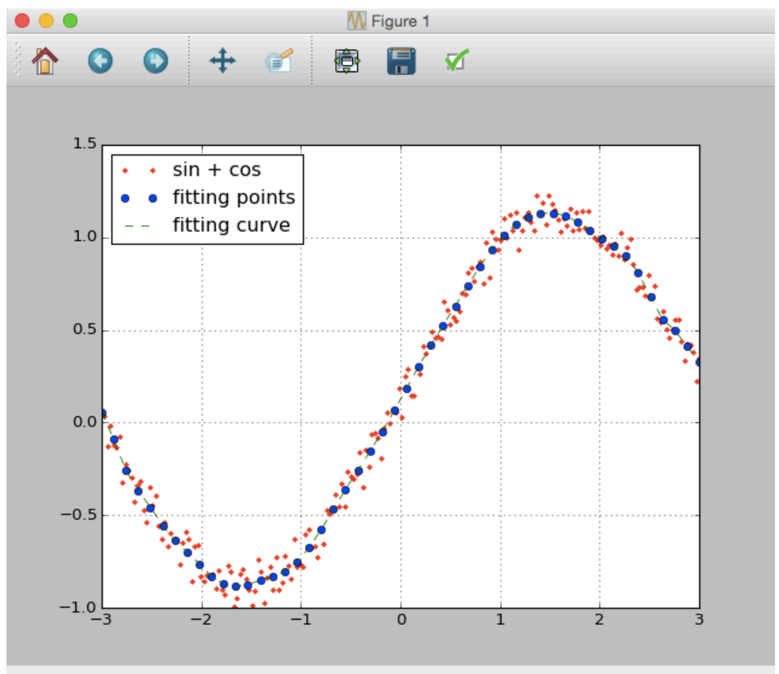
Dealing with polynomials

- Dealing with polynomials

Example:

```
import numpy as npy # we import numpy to use the 'polynomial' method
x = npy.linspace(-3, 3, 200) # create vector 'x' with 200 numbers between [-3:3]
y = npy.sin(x) + 0.25*plb.rand(200) # create and calculate function 'y'
p = npy.polynomial.Legendre.fit(x, y, 24)  # use a Legendre series class object
t = npy.linspace(-3, 3, 50)
plb.plot(x, y, 'r.', lw=1.75, label='sin + cos') # plot with some specifics
plb.plot(t, p(t), 'o', label='fitting points')
plb.plot(t, p(t), '--', linewidth=.75, label='fitting curve')
plb.grid(True)
plb.legend(loc='upper left')     # add the legend will labels
```

Note:
In order to take advantage of the ‘numpy.polynomial’ method, which contains different series such as Chebyshev, Hermite, Legendre and Laguerre we import numpy



Course Content Outline

- **NumPy 2/3**
- Array operations
- Reductions
- Broadcasting
- Array: shaping, reshaping, flattening, resizing, dimension changing
- Data sorting — Midterm / Project proposal due

- **NumPy 3/3**
- Type casting
- Masking data
- Organizing arrays
- Loading data files
- Dealing with polynomials
- Good coding practices

- **Scipy 1/2**
- What is Scipy?
- Working with files
- Algebraic operations
- The Fast Fourier Transform
- Signal Processing — HW4

- **Scipy 2/2**
- Interpolation
- Statistics
- Optimization

- **Project**
- Project Presentation — Final Project

What is Scipy?

- What is Scipy?
  - Scipy is Python's scientific core package designed for high level scientific computing
  - it contains many prebuild common mathematical functions, series, transforms that are easy to use
  - its toolboxes are well set to solve common scientific problems
  - good knowledge of the package allows for building complicated algorithms and proof-of concept solutions
  - Scipy has modules that can work on problems involving integration, optimization, interpolation, image and sound processing, Fourier transform, statistics, and many other special functions
  - Scipy's scientific libraries resemble the ones available in Matlab
  - Scipy integrates very well with NumPy and can operate on NumPy arrays very efficiently

What is Scipy?

- What is Scipy?
  - here is a list of some of the most common Scipy modules:
    - Scipy.fftpack – contains Fast Fourier Transforms (FFTs)
    - Scipy.integrate – contains a variety of integrating functions
    - Scipy.interpolate – contains a variety of interpolation classes
    - Scipy.io – functions for reading and writing data from/to a variety of file formats
    - Scipy.io.wavfile – read/write data from/to a variety of file formats 'wav','arff', etc.
    - Scipy.linalg – contains linear algebra routines
    - Scipy.ndimage – contains many functions for multi-dimensional image processing
    - Scipy.signal – rich filtering capabilities, wavlets, spectral analysis, and much more
    - Scipy.optimize – local optimization package and root finding
    - Scipy.spatia – nearest neighbor queries and distance functions
    - Scipy.stats – large number of probability distributions and statistical functions
    - Scipy.special – large variety of functions such as: elliptic, bessel, legendre, etc.
    - Scipy.misc – variety of other functions

What is Scipy?

- What is Scipy?
  - scipy.fftpack vs numpy.fft:
    - Some of the NumPy code is exported through Scipy, hence there are similarities between the two packages:

```
In [1]: import scipy.fftpack as s_fft

In [2]: import numpy.fft as n_fft

In [3]: # The two lines below will produce the same result:

In [4]: s_fft.fftfreq(8)
Out[4]: array([ 0.   ,  0.125,  0.25 ,  0.375, -0.5  , -0.375, -0.25 , -0.125])

In [5]: n_fft.fftfreq(8)
Out[5]: array([ 0.   ,  0.125,  0.25 ,  0.375, -0.5  , -0.375, -0.25 , -0.125])
```

    - scipy.sin = numpy.sin, etc. –

```
cpy.sin(90)
0.89399666360055785

npy.sin(90)
0.89399666360055785
```

What is Scipy?

- What is Scipy?
  - scipy.fftpack vs numpy.fft:
    - the scipy.fftpack does much more on top of what numpy.fft offers:
      - » fft and ifft - the Discrete Fourier Transform and its inverse of real or complex sequence of numbers
      - » fft2 and ifft2 - 2D discrete Fourier transform and its inverse
      - » fftn and ifftn - multidimensional discrete Fourier transform and its inverse
      - » dct and idct - Discrete Cosine Transform of arbitrary type sequence
      - » dst and idst - Discrete Sine Transform of arbitrary type sequence
      - » tilbert and itilbert - the h-Tilbert transform of a periodic sequence and its inverse
      - » hilbert and ihilbert - Hilbert Transform of a periodic sequence and its inverse
      - » fftfreq - the Discrete Fourier Transform sample frequencies
      - » convolve - performs convolution on a given signal
      - » … and more

What is Scipy?

- What is Scipy?
  - – scipy.fftpack vs numpy.fft:
    - Scipy runs faster

Scipy is much faster when arrays increases in size

```
Python

In [6]: size = 100000

In [7]: # measuring fft.scipy execution:

In [8]: %timeit s_fft.fftfreq(size); # real numbers
1000 loops, best of 3: 505 µs per loop

In [9]: # measuring fft.numpy execution:

In [10]: %timeit n_fft.fftfreq(size); # real numbers
1000 loops, best of 3: 576 µs per loop

In [11]: # measuring fft.scipy execution:

In [12]: %timeit s_fft.ifft(s_fft.fftfreq(size)); # complex numbers
1 loops, best of 3: 6.29 ms per loop

In [13]: # measuring fft.numpy execution:

In [14]: %timeit n_fft.ifft(n_fft.fftfreq(size)); # complex numbers
100 loops, best of 3: 14.9 ms per loop
```

What is Scipy?

- What is Scipy?
  - we usually import Scipy like this:

    ```
    [15] import numpy as np
    [16] from scipy import signal      # or choose any other Scipy module
    ```

  - since there are many modules in Scipy, we usually import only specific Scipy functionality and never need to import the complete Scipay package
  - Scipy depends on the NumPy library, while it provides convenient and fast N-dimensional array operation and manipulation
  - just like Python, NumPy and Matplotlib, Scipy is open-source

Working with files

- Working with files
  - most of the times it is necessary to load and save data files of different types such as:
    - text, sound or image
  - we already saw that we can import text files and images with NumPy
  - using Scipy, we can create and import more sophisticated file structures such as .csv, .mat, etc.
  - since Matlab .mat files are widely used in the scientific computing community, using Scipy we can read and write .mat files
  - using Python we can write directly to databases using a specifically designed non-SQL storage ways such as: PyTables and HDF5, but also in regular SQL tables

Working with files

- Working with files – text / .mat – 1/2

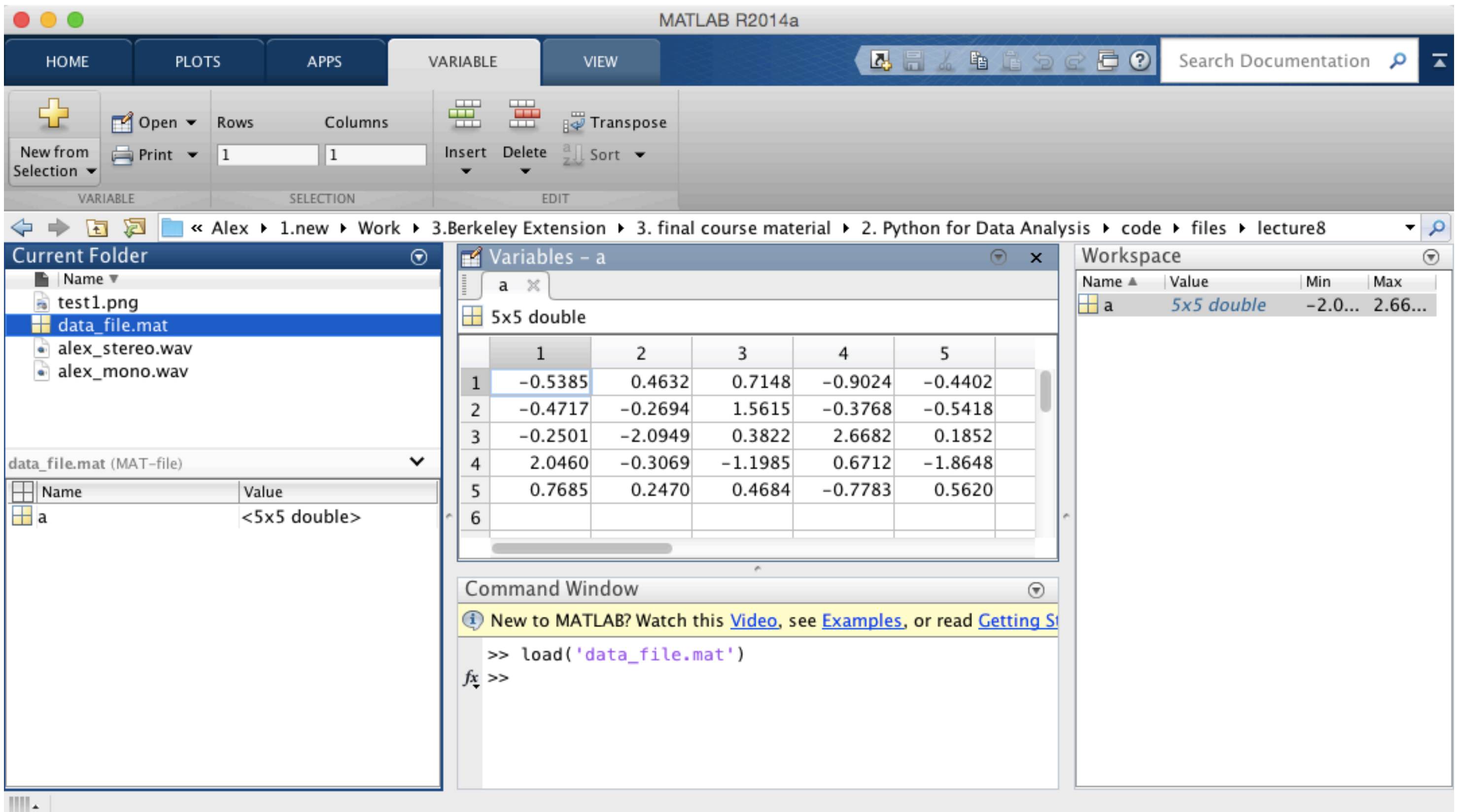
saving and loading .mat files

```
In [17]: from scipy import randn as rnd

In [18]: from scipy import io as sio

In [19]: a = rnd(5, 5) # create an array using "standard normal" distribution

In [20]: a
Out[20]:
array([[-0.53850611,  0.46316393,  0.71478529, -0.90239455, -0.44023213],
       [-0.47173271, -0.26944028,  1.56151908, -0.37684898, -0.5418205 ],
       [-0.2501384 , -2.0949024 ,  0.38220796,  2.66823633,  0.18517809],
       [ 2.04596859, -0.30689392, -1.19849106,  0.67119412, -1.86481571],
       [ 0.7685391 ,  0.24704472,  0.46837002, -0.77831832,  0.56196278]])

In [21]: sio.savemat('files/lecture8/data_file.mat', {'a':a}) # savemat expects a dictionary

In [22]: a_data = sio.loadmat('files/lecture8/data_file.mat', struct_as_record=True)

In [23]: a_data      # this call will bring the heather containing Matlab details
Out[23]:
{'__version__': '1.0',
 '__globals__': [],
 '__header__': b'MATLAB 5.0 MAT-file Platform: posix, Created on: Mon Jul 13 21:58:13 2015',
 'a': array([[-0.53850611,  0.46316393,  0.71478529, -0.90239455, -0.44023213],
        [-0.47173271, -0.26944028,  1.56151908, -0.37684898, -0.5418205 ],
        [-0.2501384 , -2.0949024 ,  0.38220796,  2.66823633,  0.18517809],
        [ 2.04596859, -0.30689392, -1.19849106,  0.67119412, -1.86481571],
        [ 0.7685391 ,  0.24704472,  0.46837002, -0.77831832,  0.56196278]])}

In [24]: a_data['a'] # this call will simply show the data - only
Out[24]:
array([[-0.53850611,  0.46316393,  0.71478529, -0.90239455, -0.44023213],
       [-0.47173271, -0.26944028,  1.56151908, -0.37684898, -0.5418205 ],
       [-0.2501384 , -2.0949024 ,  0.38220796,  2.66823633,  0.18517809],
       [ 2.04596859, -0.30689392, -1.19849106,  0.67119412, -1.86481571],
       [ 0.7685391 ,  0.24704472,  0.46837002, -0.77831832,  0.56196278]])
```

Working with files

- Working with files – text / .mat – 2/2

loading .mat files in Matlab, created in Scipy



Working with files

- Working with files – image 1/3
  - to be able to read images using Scipy, we may import Python Imaging Library (PIL)
  - PIL is for Python 2
  - PIL is not a dependency of SciPy
  - PIL needs to be installed separately, but it sometimes needs a relaxed installation to avoid rejection of package installation:
    - pip install pil --allow-external pil --allow-unverified pil

      --allow-external pil means to allow installation from an external source

      --allow-unverified pil means not to validate package using checksum

Working with files

- Working with files – image 2/3
  - Pillow is the “friendly PIL” package (easier to remember)
  - Pillow is based on PIL, but evolved to be better, friendlier and more modern than PIL
  - Pillow is for Python 2 and 3
  - the Pillow image module needs to be installed separately:

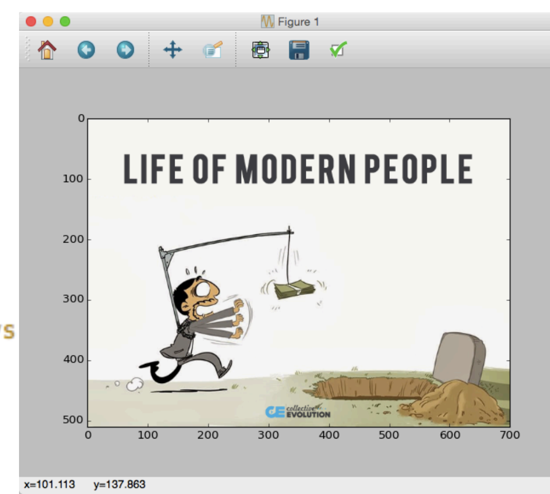
```
[25] pip install pillow
Collecting pillow
  Downloading Pillow-2.9.0.tar.gz (9.3MB)
Installing collected packages: pillow
  Running setup.py install for pillow
Successfully installed pillow-2.9.0
```

Working with files

- Working with files – image 3/3
  - ndimage can take advantage of the newly installed PIL (Pillow) functionality and is part of the Scipy main package

```
In [26]: # Scipy has this image functionality:

In [27]: from scipy import ndimage as simg

In [28]: img1 = simg.imread('files/lecture8/test1.png')

In [29]: img1.shape
Out[29]: (511, 700, 3)

In [30]: # Matplotlib has a similar functionality:

In [31]: import matplotlib.pyplot as mpt

In [32]: img2 = mpt.imread('files/lecture8/test1.png')

In [33]: img2.shape
Out[33]: (511, 700, 3)
```

```
mpt.imshow(img2) # unlike with Scipy, we can plot with Matplotlib
plt.pause(2)
mpt.imshow(img1) # we can also plot 'img1' since they are both NumPy arrays
plt.pause(1)
```



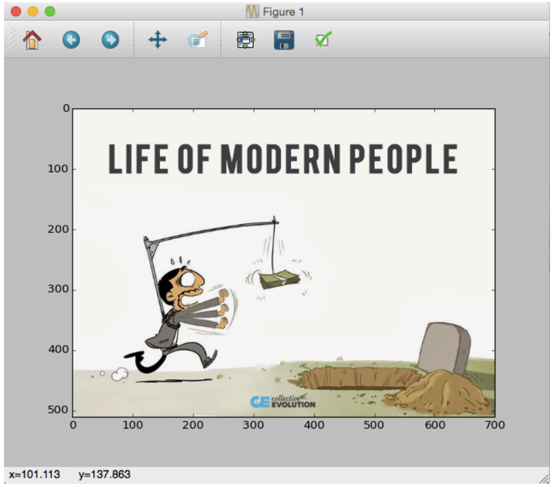
Working with files

- Working with files – image 3/3
  - ndimage using Pillow, can read images in different mode with different pixel resolutions

```python
img2 = simg.imread('files/lecture8/test1.png', flatten=False, mode='RGB')
```

  - same can be said for pyplot from Matplotlib, since it also uses Pillow to plot
  - Here are the options:

```
`mode` can be one of the following strings:

* 'L' (8-bit pixels, black and white)
* 'P' (8-bit pixels, mapped to any other mode using a color palette)
* 'RGB' (3x8-bit pixels, true color)
* 'RGBA' (4x8-bit pixels, true color with transparency mask)
* 'CMYK' (4x8-bit pixels, color separation)
* 'YCbCr' (3x8-bit pixels, color video format)
* 'I' (32-bit signed integer pixels)
* 'F' (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including
'LA' ('L' with alpha), 'RGBX' (true color with padding) and 'RGBa'
(true color with premultiplied alpha).
```



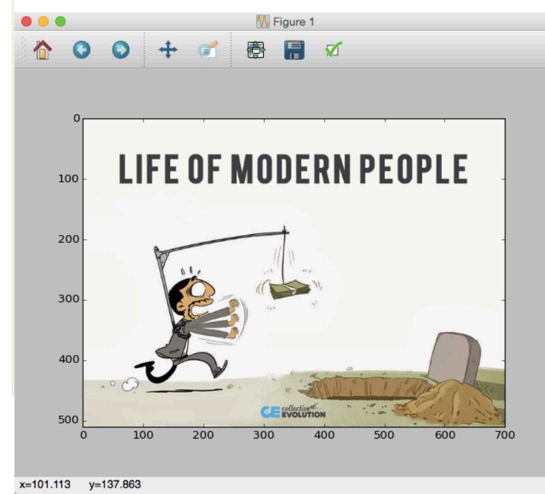
Working with files

- Working with files – image 3/3
  - be careful how you use the options:

Python

```
In [300]: img1 = simg.imread('files/lecture8/test1.png')

In [301]: img1.shape
img1.nbytes
img1.itemsize
Out[301]: (511, 700, 3)
Out[302]: 1073100
Out[303]: 1

In [304]: img2 = simg.imread('files/lecture8/test1.png', flatten=False, mode='RGB')

In [305]: img2.shape
img2.nbytes
img2.itemsize
Out[305]: (511, 700, 3)
Out[306]: 1073100
Out[307]: 1

In [308]: img3 = simg.imread('files/lecture8/test1.png', 'RGB')

In [309]: img3.shape
img3.nbytes
img3.itemsize
Out[309]: (511, 700)
Out[310]: 1430800
Out[311]: 4
```

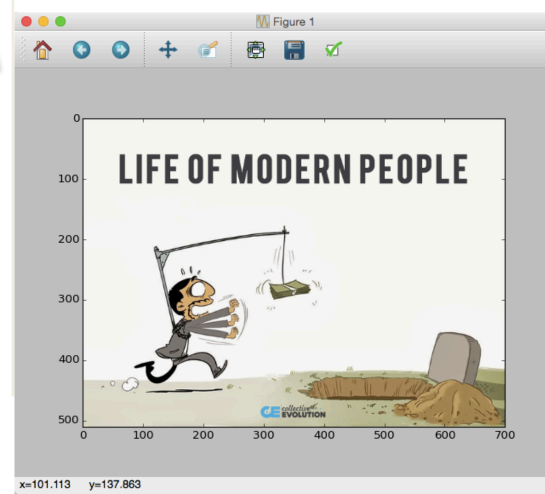
When `mode` is <u>not None</u> and `flatten` is True, the image is first converted according to `mode`, and the result is then <u>flattened</u> using mode 'F'



Working with files

- Working with files – image 3/3
  - be careful how you use the options:

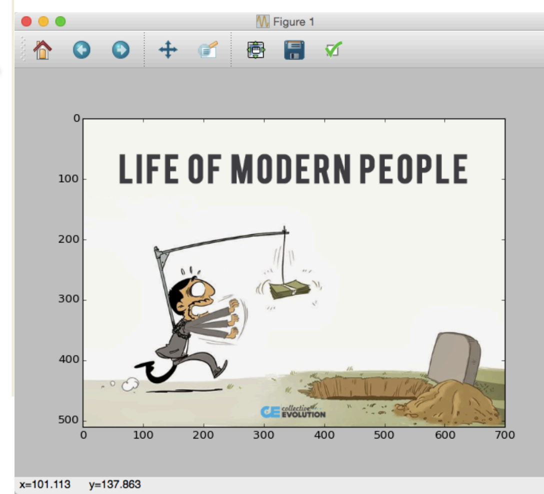
```
In [300]: img1 = simg.imread('files/lecture8/test1.png')

In [301]: img1.shape
img1.nbytes
img1.itemsize
Out[301]: (511, 700, 3)
Out[302]: 1073100
Out[303]: 1

In [304]: img2 = simg.imread('files/lecture8/test1.png', flatten=False, mode='RGB')

In [305]: img2.shape
img2.nbytes
img2.itemsize
Out[305]: (511, 700, 3)
Out[306]: 1073100
Out[307]: 1

In [308]: img3 = simg.imread('files/lecture8/test1.png', 'RGB')

In [309]: img3.shape
img3.nbytes
img3.itemsize
Out[309]: (511, 700)
Out[310]: 1430800
Out[311]: 4
```



Working with files

- Working with files – image 3/3
  - be careful how you use the options:

```
In [300]: img1 = simg.imread('files/lecture8/test1.png')

In [301]: img1.shape
img1.nbytes
img1.itemsize
Out[301]: (511, 700, 3)
Out[302]: 1073100
Out[303]: 1

In [304]: img2 = simg.imread('files/lecture8/test1.png', flatten=False, mode='RGB')

In [305]: img2.shape
img2.nbytes
img2.itemsize
Out[305]: (511, 700, 3)
Out[306]: 1073100
Out[307]: 1

In [308]: img3 = simg.imread('files/lecture8/test1.png', 'RGB')

In [309]: img3.shape
img3.nbytes
img3.itemsize
Out[309]: (511, 700)
Out[310]: 1430800
Out[311]: 4
```



Working with files

- Working with files – image 3/3
  - be careful how you use the options:

```
In [300]: img1 = simg.imread('files/lecture8/test1.png')

In [301]: img1.shape
img1.nbytes
img1.itemsize
Out[301]: (511, 700, 3)
Out[302]: 1073100
Out[303]: 1

In [304]: img2 = simg.imread('files/lecture8/test1.png', flatten=False, mode='RGB')

In [305]: img2.shape
img2.nbytes
img2.itemsize
Out[305]: (511, 700, 3)
Out[306]: 1073100
Out[307]: 1

In [308]: img3 = simg.imread('files/lecture8/test1.png', 'RGB')

In [309]: img3.shape
img3.nbytes
img3.itemsize
Out[309]: (511, 700)
Out[310]: 1430800
Out[311]: 4
```

Working with files

- Working with files – sound 1/5

reading .wav files

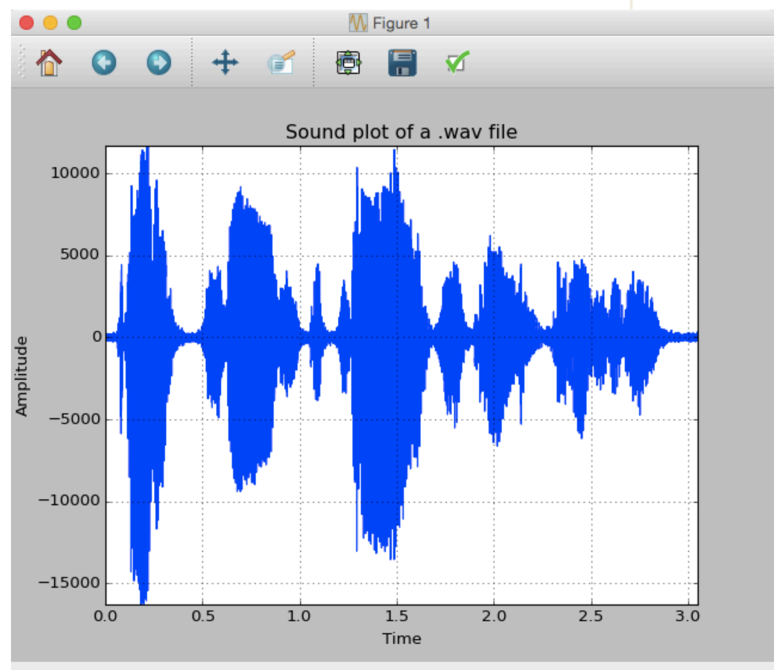
```
In [35]: from scipy.io.wavfile import read

In [36]: (fsx, x) = read('files/lecture8/alex_mono.wav')

In [37]: print(len(x.shape))   # '1' is mono
1

In [38]: print(x[:,])   # to access the channel no digit is used after ','
[-4 43 23 ..., 27 42 -3]

In [39]: x
Out[39]: array([-4, 43, 23, ..., 27, 42, -3], dtype=int16)

In [40]: (fsy, y) = read('files/lecture8/alex_stereo.wav')

In [41]: print(len(y.shape))   # '2' is stereo 2-dimensional array
2

In [42]: y
Out[42]:
array([[-4, -4],
       [44, 43],
       [20, 23],
       ...,
       [26, 29],
       [43, 41],
       [-4, -3]], dtype=int16)

In [43]: print(y[:,0])   # to access each channel separately use '0' or '1'
[-4 44 20 ..., 26 43 -4]

In [44]: print(y[:,1])
[-4 43 23 ..., 29 41 -3]
```

Working with files

- Working with files – sound 2/5

plotting .wav files

```python
# More on sound:
# Example 1:
from pylab import linspace, plot, title, xlabel, ylabel, grid, axis
from scipy.io.wavfile import read

(Fs, x) = read('files/lecture8/alex_mono.wav') # Fs - sampling frequency, x - signal
length = len(x)  # number of smaples in 'x'
time = length/Fs # calculate the length of the .wav file in secs
t = linspace(0,time,length) # create evenly spaced numbers between [0:time]
plot(t,x) # plot signal 'x'
title('Sound plot of a .wav file')
xlabel('Time')
ylabel('Amplitude')
axis('tight')
grid(True)
```



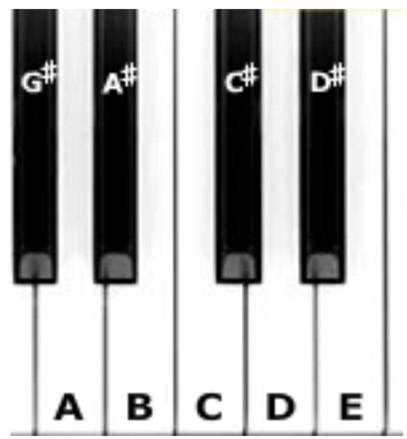
Working with files

- Working with files – sound 3/5
  - lets create a our own music by using the sin function alone

generating
writing
saving .wav files

Tone	Freq
A	440
B flat	466
B	494
C	523
C sharp	554
D	587
D sharp	622
E	659
F	698
F sharp	740
G	784
A flat	831
A	880

+



=

E C# C# D B B A B C# D E E E
E C# C# D B B A C# E E A A A

our melody

Working with files

- Working with files – sound 3/5
  - lets create our own music file by using the sin function alone

generating
writing
saving .wav files

```
# Example 2:
from pylab import plot, title, xlabel, ylabel, grid, axis
from numpy import linspace, int16, zeros, asarray, concatenate
from scipy import arange, sin, pi
from scipy.io.wavfile import read,write

Fs=8000 # Sampling frequency Fs

# Synthesis of the note:
def note(freq, Fs, length, amplitude=5):
 t = linspace(0,length,length*Fs)
 sig = sin(2*pi*freq*t)*amplitude
 return sig.astype(int16) # returns 2 byte integers

# Creating each tone with Fs samples per second and length 0.5 or 0.8 seconds:
Es = note(659,Fs,0.5,amplitude=5000)
El = note(659,Fs,0.8,amplitude=5000)
Css = note(554,Fs,0.5,amplitude=5000)
Csl = note(554,Fs,0.8,amplitude=5000)
D = note(587,Fs,0.5,amplitude=5000)
Bs = note(494,Fs,0.5,amplitude=5000)
Bl = note(494,Fs,0.8,amplitude=5000)
As = note(440,Fs,0.5,amplitude=5000)
Al = note(440,Fs,0.8,amplitude=5000)
Pau = zeros([200], dtype=int16) # this is the pause between the tones

# Create the melody:
All = concatenate((Es,Pau,Css,Pau,Csl,Pau,D,Pau,Bs,Pau,Bl,Pau,As,Pau,Bs,Pau,
                   Css,Pau,D,Pau,Es,Pau,Es,Pau,El,Pau,Pau,
                   Es,Pau,Css,Pau,Csl,Pau,D,Pau,Bs,Pau,Bl,Pau,As,Pau,Css,Pau,
                   Es,Pau,Es,Pau,As,Pau,As,Pau,Al))

write('files/lecture8/melody.wav',Fs,All) # writing our melody to a file
```

= E C# C# D B B A B C# D E E E
E C# C# D B B A C# E E A A A

our melody

… lets hear the melody

Working with files

- Working with files – sound 4/5
  - lets create a (pseudo) stereo music from a single (mono) channel

manipulating .wav files

Note: this is not a true stereo Signal

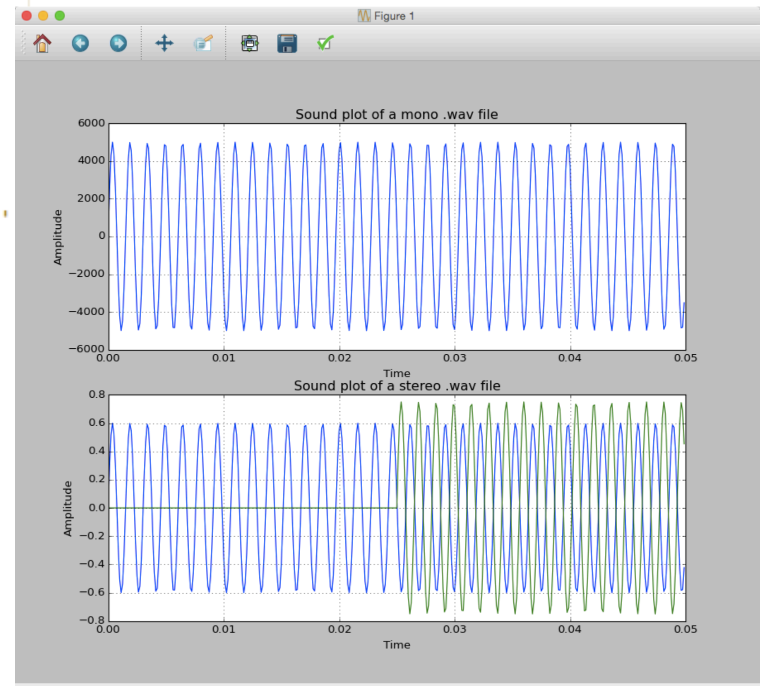
```
# Example 3 - manipulating sounds - creating stereo from mono:
from numpy import zeros, concatenate
from scipy import fft, arange, ifft, sin, pi
from scipy.io.wavfile import read,write

(Fs, x) = read('files/lecture8/melody.wav')
x      # contains all the samples
y = x  # we create the second channel
z=zeros([200])  # create an array of zeros
# 1. Time/Phase shift the two channel:
L=concatenate((x,z)) # we add zeros after the 'x' signal to create Left channel
R=concatenate((z,y)) # we add zeros before the 'y' signal to create Right channel
A=zeros([len(L),2])  # we now create the array to store 'x' and 'y' as L and R
A[:,0]=L  # we assign the Left channel
A[:,1]=R  # we assign the Right channel
# 2. Amplitude change:
A[:,0]=A[:,0]*1.2e-4 # we decrease the amplitude on the Left to avoid clipping
A[:,1]=A[:,1]*1.5e-4 # ampl. decrease on Right channel is more since it is delayed

# we write the file:
write('files/lecture8/melody_stereo.wav',Fs,A)
```

… lets hear the melody

Working with files

- Working with files – sound 5/5
  - we plot the mono and stereo music we just created

```
# Lets plot the mono and stereo sounds:
from pylab import linspace, plot, subplot, title, xlabel, ylabel, grid

length = len(L)  # number of smaples in either channel 'L' (they are equal)
time = length/Fs # calculate the length of the .wav file in seconds
t = linspace(0,time,length) # create evenly spaced numbers between [0:time]

subplot(2,1,1)
plot(t[0:400],L[0:400]) # plot the first 400 samples from the mono signal 'L'
title('Sound plot of a mono .wav file')
xlabel('Time'); ylabel('Amplitude'); grid(True)

subplot(2,1,2)
plot(t[0:400],A[0:400]) # plot the first 400 samples from the stereo signal 'A'
title('Sound plot of a stereo .wav file')
xlabel('Time'); ylabel('Amplitude'); grid(True)
```



manipulating .wav files