Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Introduction to Python®**
- Python - pros and cons
- Installing the environment with core packages
- Python modules, packages and scientific blocks
- Working with the shell, IPyton and the editor HW1

- **Basic language specifics 1/2**
- Basic arithmetic operations, assignment operators, data types, containers
- Control flow (if/elif/else)
- Conditional expressions
- Iterative programming (for/continue/while/break)
- Functions: definition, return values, local vs. global variables

- **Basic language specifics 2/2**
- Classes / Functions (cont.): objects, methods, passing by value and reference
- Scripts, modules, packages
- I/O interaction with files
- Standard library
- Exceptions

- **NumPy 1/3** *project discussion*
- Why NumPy?
- Data type objects
- NumPy arrays
- Indexing and slicing of arrays HW2

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. HW3

NumPy arrays

- Difference between a C variable and a Python variable

  – For a C variable, the compiler already knows the type by its declaration:

    - int A = 5; /* C code */

    Steps:

    1. assign <int> to A

  – For a Python variable, is only known that the variable is some sort of Python object at the time of program execution:

    - A = 5 # python code

    Steps:

    1. Set A -> PyObject_HEAD -> typecode to integer
    2. Set A -> val = 5

NumPy arrays

- Difference between a C variable and a Python variable
  - For a C variable, the compiler already knows the type by its declaration:
    - int A = 5; /* C code */
    - int B = A + 10; /* C code */

    Steps:

    1. assign <int> to A
    2. call binary_add<int, int>(A, 10)
    3. assign the result to B
  - For a Python variable, is only known that the variable is some sort of Python object at the time of program execution:
    - A = 5 # python code
    - B = A + 10 # python code

    Steps:

    1. Set A -> PyObject_HEAD -> typecode to integer
    2. Set A -> val = 5
    3. call binary_add(A, 10):

       find typecode in A -> PyObject_HEAD

       A is an integer; The value is A -> val

       find that '10' is an integer obj.

       call binary_add<int, int>(A->val, int->val)

       result of this is an integer
    4. set B -> PyObject_HEAD -> typecode to integer
    5. set B -> val to ***result***

NumPy arrays

- Difference between NumPy arrays vs Python Lists
  - NumPy array:
    - A NumPy array is a Python object build around a C array
    - This means that it has a pointer to a contiguous data buffer of values
  - Python Lists:
    - A Python list has a pointer to a contiguous buffer of pointers
    - All of them point to different Python objects, which in turn has references to its data (in this case, integers)
  - Conclusion:
    - NumPy is much more efficient than Python, in the cost of storage and in speed of access

NumPy arrays

- NumPy arrays
  - NumPy provides an N-dimensional array type called – ndarray
  - an ndarray is a multidimensional container
  - it describes a collection of "items" of the same type
  - all items can be indexed using integer type notation
  - each item in an ndarray takes up the same size block of memory, hence they are called homogenous
  - all blocks are interpreted in exactly the same way
  - each item in an array is interpreted by a separate data-type object, one of which is associated with every array and is called dtype
  - besides basic types (booleans, integers, floats, etc.), the data type objects can represent data structures as well
  - each item from an array, is indexed, and is represented by a Python object whose type is one of the array scalar types provided in NumPy
  - these array scalars allow easy manipulation of even more complicated data organization
  - ndarrays can share similar data, so changes in one will reflect in the other
  - this is referred to as *'view'* and *'base'* of the ndarray (example later in slides)

NumPy arrays

- NumPy arrays

  Example:

```
In [23]: a = np.array([[12, 34, 41], [54, 62, 18], [72, 84, 96]], np.int16)

In [24]: a
Out[24]:
array([[12, 34, 41],
       [54, 62, 18],
       [72, 84, 96]], dtype=int16)

In [25]: a.size
Out[25]: 9

In [26]: a.shape
Out[26]: (3, 3)

In [27]: type(a)
Out[27]: numpy.ndarray

In [28]: a.dtype
Out[28]: dtype('int16')

In [29]: a[2,2]   # this is how we index a particular elemnt in the array (#9)
Out[29]: 96

In [30]: b = a[0,:]

In [31]: b
Out[31]: array([12, 34, 41], dtype=int16)

In [32]: b.shape
Out[32]: (3,)

In [33]: b[2] = 88   # this is how we reassign another value to a member in the array

In [34]: a[2,2] = 99 # the change above also affects the original array

In [35]: a
Out[35]:
array([[12, 34, 88],
       [54, 62, 18],
       [72, 84, 99]], dtype=int16)

In [36]: b
Out[36]: array([12, 34, 88], dtype=int16)
```

NumPy arrays

- NumPy arrays
  - arrays can be constructed using the following reserved words: array, zeros, ones or empty
    - array – will construct an array
    - zeros – will create an array filled with zeroes
    - ones – will create an array filled with ones
    - empty – will construct an empty array to be filled at a later point
  - NumPy array parameters:

    shape: tuple of ints – shape of created array

    dtype: data-type, optional – Any object that can be interpreted as a NumPy data type

    strides: tuple of ints, optional – Strides of data in memory

    buffer: object exposing buffer interface, optional – Used to fill the array with data

    offset: int, optional – Offset of array data in buffer

    order: {'C', 'F'}, optional – Row-major or column-major order

NumPy arrays

- NumPy arrays

Examples:

```
In [37]: c = np.zeros(shape=(4,5))   # the array contains zeroes for all elelements

In [38]: c
Out[38]:
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])

In [39]: d = np.empty(shape=(2,2))   # the array contains meaningless data

In [40]: d
Out[40]:
array([[ 0.,  0.],
       [ 0.,  0.]])

In [41]: e = np.ndarray(shape=(2,3), dtype=complex, offset=np.float_().itemsize, order='C')

In [42]: e
Out[42]:
array([[  0.00000000e+000 +1.72723382e-077j,
          2.12316144e-314 +2.14479474e-314j,
          2.12375379e-314 +2.24090241e-314j],
       [  2.12530167e-314 +2.12303539e-314j,
          2.24504872e-314 +3.27074300e+015j,
          3.28995843e-318 +8.34402697e-309j]])
```

check the 'sizeof' each

Numpy

- Zeros and Empty difference:
  - empty - returns an array of given type and shape, without initializing its entries
  - zeros - return a new array of given shape and type, initialized with zeros
  - empty is therefore be marginally faster, but requires the user to manually set all values in the array. Use with caution
  - Conclusion: there is a small optimization benefit when using empty: it is slightly faster as compared to other initialization of array to zeros or ones

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | NumPy v1.11 Manual | NumPy Reference | Routines | Array creation routines

numpy.empty

numpy.empty(*shape, dtype=float, order='C'*)

Return a new array of given shape and type, without initializing entries.

Parameters: **shape** : *int or tuple of int*
Shape of the empty array
dtype : *data-type, optional*
Desired output data-type.
order : *{'C', 'F'}, optional*
Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

Returns: **out** : *ndarray*
Array of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to None.

Examples

```
>>> np.empty([2, 2])
array([[ -9.74499359e+001,   6.69583040e-309],
       [  2.13182611e-314,   3.06959433e-309]])         #random
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
       [  496041986,    19249760]])                     #random
```

Numpy

Recap:

- \- negative index in Python lists: negative numbers mean that you count from the right instead of the left. So, in a[1,2,3,4], the reference a[3]=4 == a[-1]=4, a[2]=3 == a[-2]=3, etc.

- \- the 'endpoint' option: default = True and last element included, False – not included. Observe example:

```
In [10]: plb.linspace(1,5,10)
Out[10]:
array([ 1.        ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.        ])

In [11]: plb.linspace(1,5,10, endpoint=False)
Out[11]: array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6])

In [12]: plb.linspace(1,5,10, endpoint=True)
Out[12]:
array([ 1.        ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.        ])
```

NumPy arrays

- NumPy arrays
  - array attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array elements.
flags	Information about the memory layout of the array.
flatten	A 1-D iterator over the array.
imag	The imaginary part of the array.
real	The real part of the array.
size	Number of elements in the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
shape	Tuple of array dimensions.
strides	Tuple of bytes to step in each dimension when traversing an array.
ctypes	An object to simplify the interaction of the array with the ctypes module.
base	Base object if memory is from some other object.

*source – NumPy reference

NumPy arrays

- NumPy arrays

  Examples:

```
In [43]: f = np.ndarray(shape=(2,3,2), dtype=complex)

In [44]: f
Out[44]:
array([[[  0.00000000e+000 -2.00000013e+000j,
           2.12215769e-314 +9.88131292e-324j],
        [  0.00000000e+000 +0.00000000e+000j,
           0.00000000e+000 -9.84629069e+109j],
        [  0.00000000e+000 +0.00000000e+000j,
           2.25697366e-314 +0.00000000e+000j]],

       [[  0.00000000e+000 +2.25697468e-314j,
           0.00000000e+000 +0.00000000e+000j],
        [ -2.58861351e-056 +0.00000000e+000j,
           0.00000000e+000 -2.05241193e-191j],
        [  2.12381808e-314 +2.25685768e-314j,
          -4.57473710e+035 +2.24500133e-314j]]])

In [45]: f.real
Out[45]:
array([[[  0.00000000e+000,   2.12215769e-314],
        [  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,   2.25697366e-314]],

       [[  0.00000000e+000,   0.00000000e+000],
        [ -2.58861351e-056,   0.00000000e+000],
        [  2.12381808e-314,  -4.57473710e+035]]])

In [46]: f.real.T
Out[46]:
array([[[  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,  -2.58861351e-056],
        [  0.00000000e+000,   2.12381808e-314]],

       [[  2.12215769e-314,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000],
        [  2.25697366e-314,  -4.57473710e+035]]])
```

NumPy arrays

- NumPy arrays

Examples:

```
In [47]: f.imag.flags
Out[47]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [48]: f.imag.data
Out[48]: <memory at 0x110298ce0>

In [49]: f.real.dtype
Out[49]: dtype('float64')

In [50]: f.dtype
Out[50]: dtype('complex128')

In [51]: f.shape
Out[51]: (2, 3, 2)

In [52]: f.T.shape
Out[52]: (2, 3, 2)

In [53]: f.size
Out[53]: 12

In [54]: f.itemsize
Out[54]: 16

In [55]: f.nbytes
Out[55]: 192

In [56]: f.ndim
Out[56]: 3
```

Note - it can be seen that the attributes of ndarray can be used in a nested fashion

NumPy arrays

- NumPy arrays
  - flags – gives information about the memory layout of the array

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)	This array is a copy of some other array. When this array is de-allocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

*source – NumPy reference

NumPy arrays

- NumPy arrays
  - flatten – returns a copy of the same flattened array in one dimension

```
In [57]: g = np.arange(12, 24).reshape(3, 4)

In [58]: g
Out[58]:
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])

In [59]: g[:,:]
Out[59]:
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])

In [60]: g.flat[6]
Out[60]: 18

In [61]: g.flat[9]
Out[61]: 21

In [62]: g.flat[:]
Out[62]: array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])

In [63]: g.flatten()
Out[63]: array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])

In [64]: g.T.flat[6]
Out[64]: 14
```

NumPy arrays

- NumPy arrays
  - shape – besides checking or specifying the shape of an array, by using the shape command we can also re-shape an array so long that we do not change the number of elements in it

Example:

```
Python
In [65]: h = np.array([[12,34,41],[54,67,89],[102,13,45],[78,456,218]])

In [66]: h
Out[66]:
array([[ 12,  34,  41],
       [ 54,  67,  89],
       [102,  13,  45],
       [ 78, 456, 218]])

In [67]: h.shape
Out[67]: (4, 3)

In [68]: h.shape = (2,6)

In [69]: h
Out[69]:
array([[ 12,  34,  41,  54,  67,  89],
       [102,  13,  45,  78, 456, 218]])

In [70]: h.shape = (3,6)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-70-76a181f81944> in <module>()
----> 1 h.shape = (3,6)

ValueError: total size of new array must be unchanged
```

NumPy arrays

- NumPy arrays
  - strides – represents the number of bytes (8-bit each) needed to travel in each direction (in memory) in a multidimensional array in order to get to certain element in that array along a given axis

Example:

Note – the given array *i* is stored in a continuous block of memory of:

60 bytes (5*3*4)

```
In [71]: i = np.reshape(np.arange(3*4*5), (5,3,4))

In [72]: i
Out[72]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])

In [73]: np.shape(i)
Out[73]: (5, 3, 4)
```

NumPy arrays

- NumPy arrays
  - strides

Example:

```
In [74]: i[4][2][1]
Out[74]: 57

In [75]: i[4,2,1]
Out[75]: 57

In [76]: np.dtype(i[4,2,1])
Out[76]: dtype('int64')

In [77]: i
Out[77]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])

In [78]: i.strides
Out[78]: (96, 32, 8)
```

Note – you can easily refer to an element from the array, knowing its position as shown in lines Out[74]/[75] , or …

NumPy arrays

- NumPy arrays
  - strides

Example:

```
In [79]: np.array([4,2,1])
Out[79]: array([4, 2, 1])

In [80]: np.array([4,2,1]) * i.strides
Out[80]: array([384,  64,   8])

In [81]: sum(np.array([4,2,1]) * i.strides)
Out[81]: 456

In [82]: i.itemsize
Out[82]: 8

In [83]: sum(np.array([4,2,1]) * i.strides)/i.itemsize
Out[83]: 57.0

In [84]: i
Out[84]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[48, 49, 50, 51],
        [52, 53, 54, 55],
        [56, 57, 58, 59]]])
```

Note – … you can calculate it in an iterative way shown in line Out[83]

NumPy arrays

- NumPy arrays
  - transpose – transpose can easily be performed by used a specific attribute (command)

Example:

note:
.T and .transpose()
do the same job!

```
In [85]: i.transpose
Out[85]: <function ndarray.transpose>

In [86]: i.transpose()
Out[86]:
array([[[ 0, 12, 24, 36, 48],
        [ 4, 16, 28, 40, 52],
        [ 8, 20, 32, 44, 56]],

       [[ 1, 13, 25, 37, 49],
        [ 5, 17, 29, 41, 53],
        [ 9, 21, 33, 45, 57]],

       [[ 2, 14, 26, 38, 50],
        [ 6, 18, 30, 42, 54],
        [10, 22, 34, 46, 58]],

       [[ 3, 15, 27, 39, 51],
        [ 7, 19, 31, 43, 55],
        [11, 23, 35, 47, 59]]])

In [87]: np.shape(i.transpose())
Out[87]: (4, 3, 5)
```

NumPy arrays

- NumPy arrays
  - ctypes:
    - this module is part of the standard Python distribution package
    - it is used for shared C-libraries, in case you have some useful code written in C and would like to put a Python wrapper around it to incorporate a specific routine written in C in your code
    - this possibility opens up a great number of already well written and tested C routines
    - the problem when using this module however is that it can lead to nasty crashes because of poor type checking

Example:

a problem can occur when you pass an array as a pointer to a raw memory location and you forget to check if the subroutine may access memory outside of the array boundaries

NumPy arrays

- NumPy arrays
  - ctypes:
    - when using *ctypes* remember that this approach uses a raw memory location to a compiled code and it may not be error prone to user mistakes
    - good knowledge of the shared library and this module is a must
    - this approach most times requires extra Python code to handle errors of different kind to:
      - check for the data types
      - array boundaries of the passes objects
    - this however will slow down the interface because of all additional checking and type conversion (C to Python) that is necessary
    - this tool is for people with strong Python skills, but weak C knowledge

NumPy arrays

- NumPy arrays
  - ctypes:
    - to use *ctypes* you must have the following:
      - have a library to be shared
      - load the library to be shared
      - convert the Python objects to *ctypes* arguments that can be interpreted correctly
      - call the function from the library containing the *ctypes* arguments
    - when using *ctypes* some of the basic attributes that can be used are:
      - data, shape and strides (... for more attributes please refer to the NumPy guide)
    - one should be careful when using temporary arrays or arrays constructed on the fly, because they return a pointer to an invalid memory location since it has been de-allocated as soon as the next Python statement is reached

Examples:

a) (a+b).ctypes – wrong , because the array created as (a+b) is de-allocated before the next statement

b) c = (a+b).ctypes – correct, because *c* will have a reference to the array

NumPy arrays

- NumPy arrays
  - ctypes:

Examples:

```
In [88]: import numpy as np

In [89]: j = np.array([[12, 34, 99, 32],[41, 52, 45, 16],[64, 88, 67, 58]])

In [90]: j
Out[90]:
array([[12, 34, 99, 32],
       [41, 52, 45, 16],
       [64, 88, 67, 58]])

In [91]: j.ctypes
Out[91]: <numpy.core._internal._ctypes at 0x105999588>

In [92]: j.ctypes.data
Out[92]: 4466631472

In [93]: j.ctypes.shape
Out[93]: <numpy.core._internal.c_long_Array_2 at 0x105996840>

In [94]: j.ctypes.strides
Out[94]: <numpy.core._internal.c_long_Array_2 at 0x1059968c8>
```

NumPy arrays

- NumPy arrays

  – ctypes: Example:

1. begin with writing your C library and save the file ‘ctypes_lib.c’:

```c
#include <stdio.h>

void myprint(void);
void myprint()
{
    printf("This is ctypes example in Python\n");
}
```

2. install your gcc if you don’t have one (skip this step if you do):

   - PC: find a compiler and install using the .exe file. Try using Cygwin - a Unix-like environment on Win
   - Mac OS X in the terminal type: xcode-select –install

3. you need to compile the file as shared library using this notation:

   - PC: $ gcc -shared -Wl,-soname, ctypes_lib -o ctypes_lib.so -fPIC ctypes_lib.c
   - Mac OS X: $ gcc -shared -Wl,-install_name, ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c

```
Macintosh:lecture4 alex$ gcc -shared -Wl,-install_name,ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c
[Macintosh:lecture4 alex$
```

NumPy arrays

- NumPy arrays
  - ctypes: Example:

4. Create your ctypes Python wrapper module 'ctypes_lib_tester.py' and execute it:

```python
## Ctypes example of using a C file code:

import ctypes

c_test_lib = ctypes.CDLL('ctypes_lib.so')
c_test_lib.myprint()
```

5. The result should be:

```
Python
This is ctypes example in Python
```

6. If you run:

```
In [1]: c_test_lib.myprint()
Out[1]: 33
```

this only prints the number of characters in the 'c' library, so for the text:
"This is ctypes example in Python\n" there are 33, including the end of line character '\n'

NumPy arrays

- NumPy arrays
  - ctypes: Example:

7. If you compile and execute the library in the terminal here is the result:

lecture4 — -bash — 114×16

```
[Macintosh:lecture4 alex$ gcc -shared -Wl,-install_name,ctypes_lib.so -o ctypes_lib.so -fPIC ctypes_lib.c
[Macintosh:lecture4 alex$ ls -la
total 88
drwxr-xr-x  10 alex  501   340 Oct 24 13:55 .
drwxr-xr-x   9 alex  501   306 Nov 19  2015 ..
-rw-r--r--@  1 alex  501  6148 Oct 24 13:45 .DS_Store
-rw-r--r--   1 alex  501   613 Oct 22 13:52 class excercise.py
-rw-r--r--   1 alex  501   110 Oct 24 13:53 ctypes_lib.c
-rwxr-xr-x   1 alex  501  8376 Oct 24 13:55 ctypes_lib.so
-rw-r--r--   1 alex  501   122 Oct 24 13:55 ctypes_lib_tester.py
-rw-r--r--   1 alex  501  1361 Oct 24 12:55 lecture4 - inverting a non-square matrix using SVD - pseudoinverse.py
-rw-r--r--   1 alex  501    81 Oct 22 13:55 my_array
-rw-r--r--   1 alex  501   439 Oct 19 20:49 speed_test_list_ndarray.py
[Macintosh:lecture4 alex$ python ctypes_lib_tester.py
This is ctypes example in Python
Macintosh:lecture4 alex$
```

NumPy arrays

- NumPy arrays
  - base:
    - is an attribute used when we need to keep track of the memory reference of the original object owner in case two objects are referring to the same memory location
    - it is a way of NumPy to keep track of the data source in memory for any given array

Example:

```
In [95]: k = np.array([[12, 34, 99],[41, 52, 45],[64, 88, 67]])

In [96]: k
Out[96]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [97]: k.base is None
Out[97]: True

In [98]: l = k[0:2]

In [99]: l
Out[99]:
array([[12, 34, 99],
       [41, 52, 45]])

In [100]: l.base is k
Out[100]: True
```

Indexing and slicing of arrays

- Indexing and slicing of arrays (recap)
  - indexing:
    - so far we saw indexing of arrays in many occasions
    - indexing in NumPy arrays begins from '0' and is runs in an scalar/integer progression
  - slicing:
    - a slice is an object referring to a portion (slice) of an array
    - the newly created slice is an array generated by slicing the original array and it always represents a view of the latter

Indexing and slicing of arrays

- Indexing and slicing of arrays

Examples:

```
In [101]: m = np.array([[12, 34, 99],[41, 52, 45],[64, 88, 67]])

In [102]: m
Out[102]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [103]: m.shape
Out[103]: (3, 3)

In [104]: m[0:]       # slicing
Out[104]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [105]: m[1]        # indexing/slicing
Out[105]: array([41, 52, 45])

In [106]: m[1:2]      # slicing
Out[106]: array([[41, 52, 45]])

In [107]: m[1][0]     # indexing
Out[107]: 41

In [108]: m[:3]       # slicing
Out[108]:
array([[12, 34, 99],
       [41, 52, 45],
       [64, 88, 67]])

In [109]: m[2,1]      # indexing
Out[109]: 88

In [110]: m[-1,-3]    # slicing - representing element (-1+3=2, -3+3=0)
Out[110]: 64
```

Discussion

- Discussion

  matrix inversion:

  1. by definition a matrix is commutative with its inverse on multiplication:

     $A_{[mxn]}*A_{[nxm]}^{-1}=A_{[nxm]}^{-1}*A_{[mxn]}=I$

     so, it must be that m=n!

  2. A non-square matrix inverse is possible using SVD:

     There exists a left inverse U and a right inverse V that is defined for **all** matrices including non-square matrices

```python
# Using Singular Value Decomposition (SVD) for manually performing a pseudoinverse on a non-square matrix:
# It is not the actual inverse matrix, but the "best approximation" of such in the sense of least squares

from numpy import random, matrix, linalg, diag, allclose, dot

# Create Matrix A with size (3,5) containing random numbers:
A = random.random(15)
A = A.reshape(3,5)
A = matrix(A)

# 1-3. Using the SVD function will return:
# U - a matrix with columns = the eigenvectors (L) of the A*A.T
#     holds Left-singular vectors
# s - a diagonal matrix with diagonal = the singular values of matrix A:
#     the singular (diagonal) values in s are square roots of eigenvalues from U and V
#     Σ+ is the pseudoinverse of Σ, which is formed by replacing every non-zero
#     diagonal entry by its reciprocal and transposing the resulting matrix
# V - a matrix with columns = the eigenvectors (R) of the A.T*A
#     holds Right-singular vectors
# U and V - must preserve the properties of the original matrix A, so they are orthogonal
U,s,V = linalg.svd(A, full_matrices=False)

# Constrcut a giagonal matrix 'S', from the giagonal 's':
S = diag(s)

# 2-3. Invert the square diagonal matrix by inverting each diagonal element:
S[0,0], S[1,1], S[2,2] = 1/diag(S[0:3,0:3])

# 3-3. Now we use the SVD elements to obtain the pseudo-inverse of matrix A:
X = dot(U, dot(S, V))
X = X.T # Final step: we must transpose

# Check each matrix:
A.shape, U.shape, S.shape, V.shape

# Comparison test 1:
A.I-X

# Comparison test 2:
allclose(A.I, X)
```

HW assignment 2

1. Include a section with your name
2. Create matrix A with size (3,5) containing random numbers
3. Find the size and length of matrix A
4. Resize (crop) matrix A to size (3,4)
5. Find the transpose of matrix A and assign it to B
6. Find the minimum value in column 1 of matrix B
7. Find the minimum and maximum values for the entire matrix A
8. Create Vector X (an array) with 4 random numbers
9. Create a function and pass Vector X and matrix A in it
10. In the new function multiply Vector X with matrix A and assign the result to D
    (note: you may get an error! … think why and fix it. Recall matrix manipulation in class!)
11. Create a complex number Z with absolute and real parts != 0
12. Show its real and imaginary parts as well as it's absolute value
13. Multiply result D with the absolute value of Z and record it to C
14. Convert matrix B from a matrix to a string and overwrite B
15. Display a text on the screen: 'Name is done with HW2', but pass your 'Name' as a string variable
16. Organize your code: use each line from this assignment as a comment line before each step
17. Save all steps as a script in a .py file
18. Email me your .py file and screenshots of your running code before next class. I will run it!