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

- **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**
- Why NumPy?
- Data type objects
- NumPy arrays
- Indexing and slicing of arrays *project discussion*

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. *project examples,* HW2

Language specifics

- I/O interaction with files

    - when working with data it is generally more convenient to read it from a file containing it
    - in order to store some result the user have to be able to write to a file
    - in order to begin reading from or writing to a file, the user has to specify it

Example:

```python
file = open('files/lecture3/test.txt', 'r') # opens file for reading
sentences = file.readlines()
print(sentences)
print(len(sentences))
file.close()

file = open('files/lecture3/test.txt', 'w') # opens file for writing
file.write('We will overwrite the previous text \n and go to a new line as well')
file.close()

file = open('files/lecture3/test.txt', 'r') # opens file for reading
sentences = file.readlines()
print(sentences)
print(len(sentences))
file.close()
```

… the code above will result in:

```
['Hello all, I am a text file :)']
1
['We will overwrite the previous text \n', ' and go to a new line as well']
2
```

Language specifics

- I/O interaction with files
  - below are the possible file mode options for file I/O interaction:
    - r – open file to read-only
      - » Note: the file must exist, you can not write to it, but can only read from it
    - w – open the file to write-only
      - » Note: creates a new file or overwrites an existing one. can not read from it
    - a – open file to append
      - » Note: creates a new file or appends to an existing one, but does not delete any previous entries. can not read from it
    - r+ - open file to read and append (update)
      - » Note: the file must exist, and you can read from it and append at the end

Language specifics

- Standard library
    - Some of the top standard library modules in Python are:

    - Os – provides a selected list of operating system level functionality
    - Sys – provides access to some variables used by the interpreter
    - Io – deals with I/O functionality for the three main types of I/O: text, binary and raw
    - Math – it gives access to mathematical functions excluding complex numbers (->cmath)
    - Wave – part of Python core installation, provides interface to the WAV format
    - Audioop – consist of useful tools for operating on digital sound sampled data
    - Html – provides an utility to work with the html language
    - Time – provides functions related to time
    - Calendar – provides various calendar capability
    - Daytime – extended way of manipulating date and time

Language specifics

- Exceptions

  - exceptions in Python are raised when the interpreter finds a problem with executing a code
  - they can be used to notify the user that certain state is reached or a condition is met
  - exceptions can pass messages from one part of the code to another
  - there are different types of errors and some of them are shown below

Example:

Python

```
In [35]: a = [12,34,41,54]

In [36]: a[4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[4]

IndexError: list index out of range

In [37]: a[2]/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[2]/0

ZeroDivisionError: division by zero
```

Language specifics

- Exceptions
  - exceptions in Python are raised when the interpreter finds a problem with executing a code
  - they can be used to notify the user that certain state is reached or a condition is met
  - exceptions can pass messages from one part of the code to another
  - there are different types of errors and some of them are shown below

Example:

```
Python
In [38]: a.delete
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a.delete

AttributeError: 'list' object has no attribute 'delete'

In [39]: a[3] + 'Python'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[3] + 'Python'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Language specifics

- Exceptions
  - in order to handle exceptions, they have to be caught first

Example:

… running the code:

```
while True:
    try:
        a = int(input('Please enter a number: '))
        print('You entered the number ', a)
        print('I will now exit. Good bye!')
        break
    except ValueError:
        print('You entered an invalid number.  Please try again.')
```

… will produce the following result:

```
Please enter a number: q
You entered an invalid number.  Please try again.
Please enter a number: t
You entered an invalid number.  Please try again.
Please enter a number: 5
You entered the number  5
I will now exit. Good bye!
```

Why NumPy?

- Why NumPy?
  - Numpy is the main scientific open-source package for numerical computation in Python
  - Numpy provides:
    - functionality comparable to Matlab,
    - it allows for fast algorithm development and proof-of-concept scientific solutions
    - It provides logic manipulation functionality
    - large set of mathematical functions
    - linear algebra functionality
    - Fast Fourier transform
    - large multidimensional array objects
    - variety of routines for fast operations on arrays
    - different objects, like matrices and masked arrays
    - random simulation
    - sorting
    - statistical operations
    - … and much more

Why NumPy?

- Why NumPy?

  - NumPy's core functionality is the *ndarray*, which stands for *n*-dimensional array
  - NumPy arrays' data structure and standard sequences in Python have some important differences:

    - NumPy array elements <u>must</u> be of the same data type and take the same memory space
    - this gives NumPy the capability to make advanced mathematical calculations possible on large data sets
    - this kind of calculations are executed with higher efficiency and use more concise code as compared to the built-in sequences in Python
    - lists in Python can increase on the fly, while arrays in NumPy are fixed size once created
    - when the size of an ndarray is changed, a new array will be created and the reference (id) to the the original array will be released (lost and deleted)
    - in Python and NumPy, when having arrays of objects, arrays of different sized elements are possible

Why NumPy?

- Why NumPy?
  - The main differences between regular Python objects and NumPy objects are:
    - **Speed** – comparing the results from a simple test on performing addition over a regular Python list and over a NumPy array, reveals that the sum on the latter is faster
    - **Memory efficiency**:
      - NumPy's arrays are more compact than Python lists (example later in slides)
      - a list of lists in Python, would take at 3-5 times more space than a NumPy array using single-precision float type numbers
    - **Functionality** - FFT, convolution, statistics, linear algebra, histograms, etc.
    - **Convenience** – all vector and matrix operations come free with NumPy, while they are efficiently implemented and save unnecessary work

Why NumPy?

- Why NumPy?
  - The main differences between regular Python objects and NumPy objects are:
    - **Speed** – comparing the results from a simple test on performing addition over a regular Python list and over a NumPy array, reveals that the sum on the latter is faster for large calculations

Speed test example:

```python
from numpy import arange
import time

N = int(input('Please enter a number: '))

x = arange(N)
y = range(N)

tic = time.clock()
x.sum()
toc = time.clock()
t_numpy = toc - tic

tic = time.clock()
sum(y)
toc = time.clock()
t_list = toc - tic

print("numpy: %.3e sec" % (t_numpy))
print("list:  %.3e sec" % (t_list))
print("diff:  %.3e sec" % (t_list-t_numpy))
print("ratio: list is %.1i times slower than numpy ndarray" % (t_list/t_numpy))
```

Python

```
In [1]: run speed_test_list_ndarray
Please enter a number: 20000
numpy: 7.300e-05 sec
list:  8.330e-04 sec
diff:  7.600e-04 sec
ratio: list is 11 times slower than numpy ndarray
```

Data type objects

- Data type objects
  - there are five basic numerical types in NumPy:
    - bool – booleans
    - int – integers
    - uint – unsigned integers
    - float – floating point
    - complex – 2 double precision numbers
  - all numerical types in NumPy are instances of the dtype object and you can find them like this:

```
In [1]:  import numpy as np
In [2]:  np.<data type>
                 or
In [3]:  dir(np)
```

Data type objects

- Data type objects
  - NumPy 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
  - NumPy 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

  - To check how many bits each type occupies, use one of these notations:
    1) (np.dtype(np.<type>).itemsize)*8
    2) np.<type>().itemsize*8

Data type objects

- Data type objects
  - the difference between signed and unsigned integers and long type variables is:
    - the signed and unsigned types are of the same size
    - the signed can represent equal amount of values around the ‘0’ thus representing equal amount of positive and negative numbers
    - the unsigned can not represent any negative numbers, but can represent double the amount of total positive numbers as compared to the signed type
    - for 32-bit int we have:

      int: –2147483648 to 2147483647

      uint: 0 to 4294967295
    - for 64-bit long we have:

      long: -9223372036854775808 to 9223372036854775807

      ulong: 0 to 18446744073709551615

Data type objects

- Data type objects
  - some of the data types that contain numbers, explicitly specify the bit size of the particular type
  - this is an important thing to know when coding on a 32-bit or 64-bit platforms and a low-level languages are used (C or Fortran)
  - some NumPy data types can be used to:
    - convert python numbers to array scalars (used as functions)
    - convert python sequences of numbers to arrays
    - enter as arguments to the dtype keyword to call various NumPy methods

Data type objects

- Data type objects
  - Examples:
    - convert Python numbers to array scalars (used as functions)

```
In [3]: import numpy as np

In [4]: from sys import getsizeof

In [5]: x=float(2.5)

In [6]: type(x)
Out[6]: float

In [7]: getsizeof(x)
Out[7]: 24

In [8]: x = np.float128(2.5)

In [9]: type(x)
Out[9]: numpy.float128

In [10]: getsizeof(x)
Out[10]: 32
```

… try it in class

Data type objects

- Data type objects
  - Examples:
    - convert Python sequences (lists, tuples, etc.) of numbers to arrays

```
In [11]: a=([12,34,41,52])

In [12]: type(a)
Out[12]: list

In [13]: b=np.int32(a)

In [14]: type(b)
Out[14]: numpy.ndarray

In [15]: getsizeof(a)
Out[15]: 96

In [16]: getsizeof(b)
Out[16]: 80

In [17]: a
Out[17]: [12, 34, 41, 52]

In [18]: b
Out[18]: array([12, 34, 41, 52], dtype=int32)
```

Data type objects

- Data type objects
  - Examples:
    - enter as arguments to the dtype keyword to call various NumPy methods

```
In [19]: c = np.arange(5, dtype=np.float16)

In [20]: type(c)
Out[20]: numpy.ndarray

In [21]: c
Out[21]: array([ 0.,  1.,  2.,  3.,  4.], dtype=float16)

In [22]: getsizeof(c)
Out[22]: 80
```

... try it in class

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 */

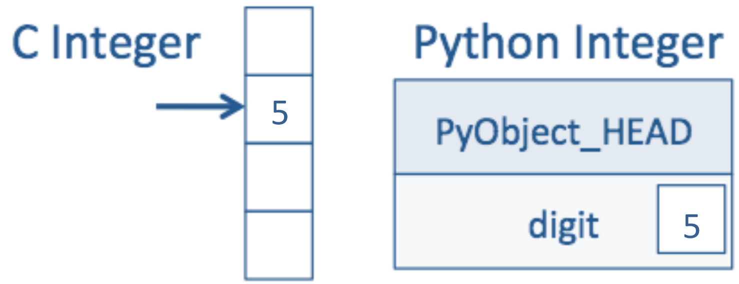
    Steps:

    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

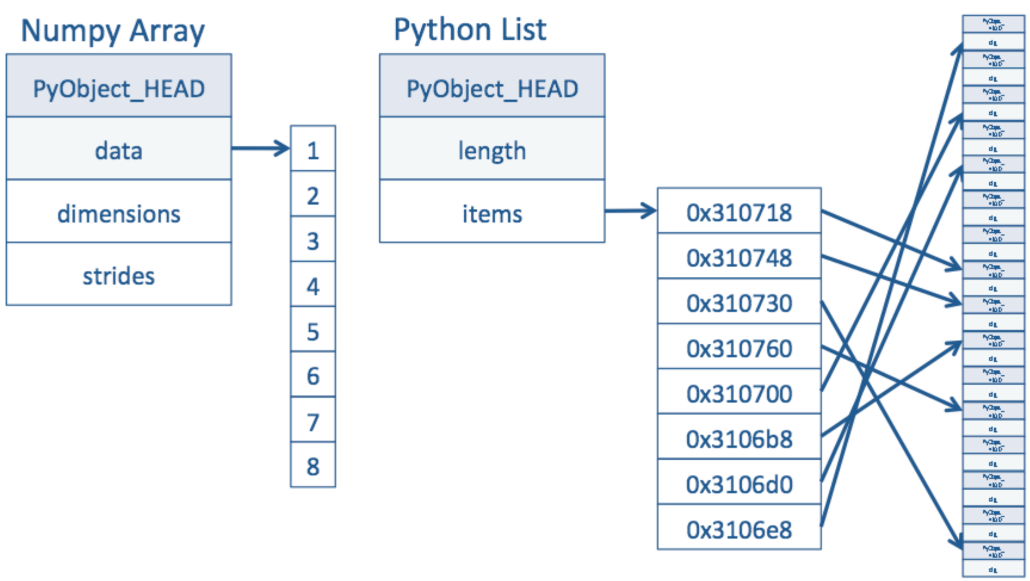
       find that ‘10’ is an integer obj.

       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:

```
Python

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)
```

… try it in class

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

... try it in class

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.        ])
```