An introduction to Python
and external packages for scientific computing

Phys 7682 / CIS 6229
Computational Methods for Nonlinear Systems
www.physics.cornell.edu/~myers/teaching/ComputationalMethods
Python (www.python.org)

- Python is dynamically typed
  - objects have types, but types of variables are not explicitly declared
  - variable names are attached to objects by assignment
    - \( x = 3 \)  \# \( x \) is the integer 3; name \( x \) added to namespace
    - \( x = 3.0 \)  \# \( x \) is the floating point 3.0
    - \( x = 'blah' \)  \# \( x \) is the string ‘blah’
    - \( x = [3, 3.0, 'blah'] \)  \# \( x \) is a list with 3 elements
    - \( x = 3 + 3.0 \)  \# \( x \) is the floating point 6.0
    - \( x = 3 + 'blah' \)  \# error: cannot add int and string
  - dynamic but strongly typed: only allowable operations can be executed
Dynamic typing (continued)

- arguments to functions are not specified or checked, but determined at runtime

```python
def f(x):       # at this point, x can formally be of any type
    y = x + 3    # x can only be a type that can be added to an int
    z = x[0]     # x can only be a type that can be indexed []

class foo:      # define some new datatype foo
    def __add__(self, other):
        # __add__ gets called when '+' operator used
        # write code to add a foo to something else

    def __getitem__(self, index):
        # __getitem__ gets called when '[]' operator used
        # write code to extract item at index from foo
```

- If it looks like a duck and quacks like a duck, it’s probably a duck
Python is interpreted

- individual statements are automatically compiled to bytecodes and executed within an interpreter
- interpreters can run full Python programs without human interaction, or execute individual commands in an interactive mode
- e.g., `python myfile.py`  # runs myfile.py in python interpreter
- e.g., in the ipython interpreter:

```
In [1]: x = 3
In [2]: print x
  3
In [3]: y = x + 4.0
In [4]: print y
  7.0
```

ipython adds to the standard interpreter:

- command history
- command completion (TAB)
- introspection and help
- “magic” functions (e.g., `%run`)
- streamlined access to operating system

See IPython paper on course web site for more details.
Python is object-oriented (OO)

- everything in Python is an object, i.e., a datatype with a defined set of allowable functions or operations (“methods”)
- methods (and other attributes) accessed via . (dot) operator
- e.g.,
  - x = 3        # x is an integer
  - y = x + 4    # addition def’d for ints by __add__ method
  - a = [1,2,3]  # a is a list
  - a.append(4)  # append is a method on list type
- but Python also allows for non-OO procedural code, e.g.,

```python
def factorial(n):
    # code to implement factorial
    # factorial is an object of function type
    x = factorial(3)       # equiv. to factorial.__call__(3)
y = x*x
```
scalars (numbers): integer, long, real (float), complex, bool, etc.

- \( x = 3 \)
- \( y = 3000000000 \)  \# long int: will print as \( 3000000000L \)
- \( w = 1.23 \)
- \( z = 3.1+4.2j \)  \# \( j = \sqrt{-1} \)
- \( s = \text{False} \)

hierarchy of conversions among scalar types: bool \( \rightarrow \) int \( \rightarrow \) long \( \rightarrow \) float \( \rightarrow \) complex

- \( a = w + z \)  \# \( a = 1.23 + (3.1+4.2j) = 4.33 + 4.2j \)
Python Basics: Built-in Types

• character strings
  - s = ‘home’
  - t = ‘running’
  - w = s + t[0:3]  # what is the value of w?
  - print ‘value of %s = %s’ % (name, val)  # formatted string

  - s = “home”      # strings via single or double quotes
  - doc = “””this string extends
    over multiple lines”””
  - # do not need to escape end-of-lines in triple-quoted strings
Python Basics: Built-in Container Types

- **lists**: ordered, dynamic, heterogeneous collections
  - `a = [1, [2.1, 'hello'], 'b', True]
    - indexing: `a[0]` is 1 ; `a[1]` is `[2.1, 'hello']`; 0-offset
    - slicing: `a[1:3]` is `[[2, 'hello'], 'b']`
  - for element in `a`:
    ```python
    # do something to every element
    ```
  - `b = [1,2,3] + [4,5,6]`    # list addition is concatenation
  - `a.reverse()`
  - `a.append(obj)`  # append obj to end of list in place
  - `a.sort(comparison_func)`  # sort according to func
  - `help(list)`     # get documentation, incl. defined methods
Python Basics: Built-in Container Types

• tuples: like lists, but immutable
  - a = (1, 2, 3) # cannot change elements
    # e.g., a[2]=4 raises an error
  - useful for returning multiple objects from function
    ‣ def func(a,b,c):
      # do something
      # return x,y,z
    ‣ r,s,t = func(a, b, c) # tuple unpack from function
  - useful in instances when immutable object not allowed (e.g., as key in dictionary)
Python Basics: Containers (cont’d)

- dictionaries: maps from keys to values (maps, associative arrays, hashes)
  - key can be any immutable type; value can be any type
  - d = {'a': 1, 'b': 2}
  - d['c'] = d['a'] + d['b']  # now d['c']=3
  - d[(1, (2,3), 12, 'x')] = some_object
  - d.keys()  # method returning list of keys (arbitrary order)
  - d.values() # method returning list of values (arb. order)
  - d.has_key(arg)  # is arg a key in d?
  - method/attribute lookup on objects done via dictionaries
    - e.g., a = [1,2,3]; a.append(4)
      - looks for ‘append’ as key in list.__dict__
      - a.append(4) calls list.__dict__['append'](a,4)
Python Basics: Containers, etc.

- **sets**: unordered collections of unique elements
  - `s1 = set([1, 2, 3]); s2 = set([3, 4, 5])`
  - `s3 = s1 & s2  # returns intersection: set([3])`
  - `s3 = s1 - s2  # returns difference set([1, 2])`

- **file objects**
  - `output = open('blah', 'w')`
  - `output.write('%6.3e\t%6.3e\n' % (x, y))`

- **function objects**
  - `def g(z):
    # code body and return statement for g(z)
 `
  - `f = lambda x: x + 3  # defines func f that returns arg+3`
  - `functions called with () operator  [  __call__() method  ]`
Python Basics: An interlude on ( ), [ ], { }

A plethora of punctuation

- **parentheses ( )**: defining tuples, calling functions, grouping expressions
  - `t = ('a', 'b', 'c')` # tuple definition
  - `z = func(x, y)` # function calling
  - `z = 2.*(x + 3) + 4./(y - 1.)` # grouping

- **square brackets [ ]**: indexing and slicing (lists, dictionaries, arrays)
  - `element = lst[i]` # list indexing: i’th element
  - `val = dct['k']` # dictionary indexing: value for key ‘k’
  - `y = a[i,j]` # numpy array indexing (later)
  - `sublist = lst[i:j]` # slicing: elements i,...,j-1

- **curly braces { }**: dictionary creation
  - `dct = {'a': 'apple', 'b': 'bear', 'c': cat}`
Python Basics: Built-in functions

- **built-in functions, e.g.,**
  - `help(obj)`: get help about an object
  - `dir(obj)`: get list of attributes and methods defined on an object
  - `range(N,M)`: return list of integers from N to M-1
  - `eval(string)`: evaluate a string as a Python expression
    - `eval('C*x**n', {'C':10.,'x':2.0, 'n':3})`
  - `str(object)`: convert obj to its string representation
  - `zip(seq1, seq2, ...)`: return “zipped” list of tuples
    - e.g., `zip([1,2,3],[4,5,6]) -> [(1,4), (2,5), (3,6)]`
  - `iter(collection)`: return iterator to traverse collection
Python Basics: Control flow
(note role of code indentation)

• for: iteration over a list (or any other iterable type)
  for element in list:
    # do something to every element in list
  for i in range(N):
    # i assumes values 0,1,2,...,N-1 (N elements in all)

• if - elif - else:
  if (x > 3) and (y < 4):
    # do something
  elif y >= 4:  # elif block not required
    # do something else
  else:  # else block not required
    # do something different still
Functions

• function definition & execution
  - note, code blocks are controlled by INDENTATION, not braces

```python
def factorial(n):
    """return factorial of an integer n, i.e.,
    n*(n-1)*(n-2)*...*1""

    if type(n) != type(0):
        raise TypeError, "integer required as input"
    if n==1:
        return 1
    else:
        return n * factorial(n-1)

x = factorial(5)     # sets x to 120
y = factorial(3.14)  # raise error with message
help(factorial)      # prints documentation string
```

• functions can return “multiple” values through a tuple
  - (actually just one value, i.e., the tuple)
Arguments to functions

```python
def g(x, y, z):
    return x + 2*y + 3*z

def f(x, y=3, z=10):
    return x + 2*y + 3*z

w = f(5)  # w = 5 + 2*3 + 3*10
w = f(5, 20)  # w = 5 + 2*20 + 3*10
w = f(3, 10, -2)  # w = 3 + 2*10 - 3*2

# Can specify arguments by keyword in any order:
w = f(z=8, y=0, x=2)  # w = 2 + 2*0 + 3*8

# Can bundle arguments into tuple and apply function to tuple
args = (10, 20, 30)
w = f(*args)  # w = f(args[0], args[1], args[2])
```
Functions & arguments (cont’d)

• references to objects are passed “by reference” and bound to local names in function scope
  - immutable arguments (e.g., numbers, strings) cannot be changed in function scope, so a local copy is made (passed “by value”)
  - mutable arguments (e.g., lists, dictionaries) can be changed within the function body since local variable and global variable can share the same reference

# Example from Lutz, Learning Python (3rd ed), p. 327

def changer(x,y):  # Function
    x = 2  # Changes local name’s value only
    y[0] = ‘spam’  # Changes shared object in place

X = 1
L = [1,2]
changer(X,L)  # Pass immutable and mutable
print X,L  # (1, [‘spam’, 2])
Object-oriented programming in Python

- class definition: new data types with associated behaviors

```python
class UndirectedGraph:       # pairs of nodes connected by edges
def __init__(self):      # “self” refers to this graph instance
    self.connections = {}              # map node to nodes

    def HasNode(self, node):
        # write code to determine if node is in graph

    def AddNode(self, node):
        # write code to add node to graph

    def AddEdge(self, node1, node2):
        # write code to add edge connecting node1 and node2
```
# make a ring graph with 10 nodes

g = UndirectedGraph()
for i in range(10):
    g.AddEdge(i, (i+1)%10)  # % is modulo operator

# read data from a file of node pairs and make a graph

g = UndirectedGraph()
for line in file('graphdata.txt'):
    nodes = line.split()  # split the line by whitespace
    g.AddEdge(nodes[0], nodes[1])  # nodes are strings in this graph

g.AddEdge((i,j), (m,n))  # tuples as nodes (edges in a 2D lattice)
• special methods can be defined for classes, e.g.,

- `__init__(self, ...)`: constructor / initialization
- `__repr__(self)`: how an object prints itself
- `__add__(self, other)`: add self to other: `a + b`
- `__sub__(self, other)`: subtract other from self: `a - b`
- `__getitem__(self, i)`: get ith element of object: `x[i]`
- `__call__(self, args)`: call object with args: `f(x, y, z)`
OOP: Inheritance

class EdgeLabeledUndirectedGraph (UndirectedGraph):
    def __init__(self):  # “self” refers to this graph instance
        UndirectedGraph.__init__(self)
        self.edgeLabels = {}

    def AddLabel(self, label, edge, value):
        edge = (min(edge), max(edge))  # sort by insure (i,j): i<j
        if labels not in self.edgeLabels:
            self.edgeLabels[label] = {}
            self.edgeLabels[label][edge] = value

    def GetLabel(self, label, edge):
        return self.edgeLabels[label][edge]

g = EdgeLabeledUndirectedGraph()
g.AddEdge(1, 2)
g.AddLabel('weight', (1,2), 100.)
g.AddEdge(2, 3)
g.AddLabel('weight', (2,3), 0.01)
g.AddLabel('name', (3,2), 'edge(2,3)')
Functional programming in Python

- emphasis on evaluation and composition of expressions, rather than control flow
- often useful for application of functions across lists

```python
def gt10(x):
    return x > 10

map(gt10, [1, 20, 3, 18])  ->  [False, True, False, True]

filter(gt10, [1, 20, 3, 18])  ->  [20, 18]

sum(filter(gt10, [1, 20, 3, 18]))  ->  38

sum(filter(lambda x: x>10, [1, 20, 3, 18]))  ->  38

# list comprehensions
[return_value for_statement <optional if_statement>]

[x*x for x in [1,20,3,18] if x>10]  ->  [400, 324]

[(i, lst.count(i)) for i in range(min(lst), max(lst)+1)]
```
Modules and imports

• have emphasized operations with built-in types and functions
• external modules can be imported if functionality is needed
• imported modules define their own namespace, accessed via the . (dot) operator

```python
import os  # operating system
os.remove('oldfile.txt')
os.system('ls -l > filelist.txt')

import math  # C math library
x = math.sin(0.1)
z = math.exp(10.3)

import pdb  # python debugger
pdb.run('myfunc()')

import profile  # profile performance of running code
profile.run('myfunc()')
```
import glob, os         # filename matching
for filename in glob.glob('*.f'):
    basename = filename.split('.')[0]    # strip off '*.f'
    os.rename(filename, basename+'.c')   # rename *.f to *.c

import pickle     # object persistence, e.g., storing to file
pickle.dump(some_complicated_object, output_file)
some_complicated_object = pickle.load(input_file)
Third-party packages for scientific computing

• scipy [www.scipy.org and links on course web page]
  - scipy is a collection of many useful numerical algorithms (numpy is the array core)

• scipy arrays
  - like built-in Python lists, except scipy arrays:
    ▸ are multidimensional and of rectangular shape (not lists of lists)
    ▸ have elements of homogeneous types, not arbitrary collections
    ▸ support “array syntax”, i.e., aggregate operations on arrays
    ▸ support slicing across all axes
    ▸ are more efficient to manipulate (looping in C, not Python)
  - more like arrays/matrices in Matlab
scipy arrays

Anatomy of an array

The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array `a` has an attribute `a.shape` which contains this tuple.

- all elements must be of the same dtype (datatype)
- the default dtype is float
- arrays constructed from list of mixed dtype will be upcast to the "greatest" common type
array syntax

- array syntax allows for concise expressions and compiled performance
  - looping in compiled C layer, not in python interpreter

```python
import scipy     # bring scipy namespace into current one

a = scipy.array([[1,2,3], [4,5,6], [7,8,9]])  # create array from list
b = a * 2         # multiply every element of a by 2

# add a and b element-wise

d = c[:,0]        # slice the 0'th column of array c

c_23 = c[2,3]     # extract [2,3] element of array

e = scipy.sin(c)  # compute element-wise sin of c

f = c > 10.       # create bool array: True where c>10., else False
```
import scipy

# various array constructor routines
p = scipy.arange(0.,1.,0.1)   # like Python range(), with float steps
q = scipy.zeros((N,M), float) # N x M array of float zeros
I = scipy.eye(10)             # 10x10 identity matrix (1. on diagonal)

# interconversion with Python lists
a_list = []
for line in file('filename'):
    vals = map(float, line.split())   # get list of floats per line
    a_list.append(vals)               # build up list incrementally
an_array = scipy.array(a_list)
original_list = an_array.tolist()
Numerical methods in scipy

- linear algebra & random arrays

```python
import scipy.linalg, scipy.random

# linear algebra routines in scipy.linalg module
eigvals, eigenvecs = scipy.linalg.eig(e)

# random array support in scipy.random module
r = scipy.random.random((L,M,N))  # L x N x M uniform random [0.,1)
s = scipy.random.randint(0, 3, (N,M))  # N x M unif. random [0,1,2])
```
Numerical methods (continued)

- ODE integration: arrays as the lingua franca of scipy

```python
import scipy, scipy.integrate  # import both the top-level scipy
# namespace, and the lower-level
# scipy.integrate module

def Lorenz(w,t,S,R,B):          # define a right-hand side function
    x,y,z = w
    return scipy.array([S*(y-x), R*x-y-x*z, x*y-B*z])

w_initial = scipy.array([0.,1.,0.])
timepoints = scipy.arange(0., 100., 0.01)
S = 10.; R = 28.; B = 8./3.
trajectory = scipy.integrate.odeint(Lorenz,w0,timepoints,args=(S,R,B))

# trajectory is a scipy array of shape 10000 x 3
```

- scipy provides functionality for integration, optimization, fitting, root-finding, special functions, FFTs, etc.
pylab (a.k.a. matplotlib)

- (mostly) 2D plotting package based largely on Matlab plotting syntax

```python
import pylab, scipy      # pylab can plot Python lists or scipy arrays

xvals = scipy.linspace(-10., 10., 100)    # equally spaced points in x
yvals = xvals**3                          # y = x**3 (x to power 3)
pylab.plot(xvals, yvals)                  # plot yvals vs. xvals
pylab.show()                              # display plot on screen
pylab.plot(xvals, yvals, 'r.')            # plot with red dots
pylab.hist(yvals)                         # histogram of yvals

# control of labels, legends, tickmarks, line width, etc.
```
Python summary

• Python as a general-purpose programming language
  - not strictly devoted to technical/scientific computing (sys admin, web tools, etc.)
  - object-oriented, but supports procedural and functional programming
  - useful as a calculator, for little scripts, for big packages, etc.

• Built-in types
  - scalars: int, long, float, complex, bool
  - containers: lists, tuples, dictionaries, sets, strings
  - functions, classes, files, modules, exceptions, iterators, etc.

• Python Standard Library
  - os interface, debugging/profiling, object copying & persistence, web programming, etc.

• Third party libraries for scientific computing
  - scipy/numpy, pylab, ipython
  - graphics and visualization: PIL (Python Imaging Library), VPython, VTK/Mayavi
  - NetworkX, Biopython, SloppyCell