

# Python in Scientific Computing

Jussi Enkovaara

Olli Tourunen

CSC – Tieteen tietotekniikan keskus Oy  
CSC – IT Center for Science Ltd.

# What is Python?

- Modern, interpreted, object-oriented, full featured high level programming language
- Portable (Unix/Linux, Mac OS X, Windows)
- Open source, intellectual property rights held by the Python Software Foundation
- Python versions: 2.x and 3.x
  - 3.x is **not** backwards compatible with 2.x
  - This course uses 2.x version

# Why Python

- Fast program development
- Simple syntax
- Easy to write well readable code
- Large standard library
- Lots of third party libraries
  - Numpy, Scipy, Biopython
  - Matplotlib
  - ...

# Information about Python



- [www.python.org](http://www.python.org)
- H. P. Langtangen, “Python Scripting for Computational Science”, Springer
- [www.scipy.org](http://www.scipy.org)
- [matplotlib.sourceforge.net](http://matplotlib.sourceforge.net)
- [mpi4py.scipy.org](http://mpi4py.scipy.org)

# Course contents

- Basics of Python programming
- Numpy: fast array interface for Python
- Scipy: scientific tools for Python
- Matplotlib: visualization with Python
- Mpi4py: parallel programming with Python



# Python basics

# Basics of Python



- Syntax and code structure
- Data types and data structures
- Control structures
- Functions and modules
- Text processing and IO
- Classes

# Python program

- Typically, a `.py` ending is used for Python scripts, e.g. **hello.py**:

```
print "Hello"
```

- Scripts can be executed by the **python** executable:

```
[jenkova ~]$ python hello.py  
Hello
```

# Interactive python interpreter



- The interactive interpreter can be started by executing **python** without arguments:

```
[jenkova ~]$ python
Python 2.4.3 (#1, Jul 16 2009, 06:20:46)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)]
on linux2
Type "help", "copyright", "credits" or
"license" for more information.
>>> print "Hello"
Hello
>>>
```

- Useful for testing and learning

# Python syntax

- Variable and function names start with a letter and can contain also numbers and underscore, e.g “my\_var”, “my\_var2”
- Python is case sensitive
- Code blocks are defined by indentation
- Comments start by # sign

```
# example
if x > 0:
    x = x + 1 # increase x
    print "increasing x"
else:
    x = x - 1
    print "decreasing x"
print "x is processed"
```

# Data types

- Python is dynamically typed language
  - no type declarations for variables
- Variable does have a type
  - incompatible types cannot be combined

```
print "Starting example"  
x = 1.0  
for i in range(10):  
    x += 1  
y = 4 * x  
s = "Result"  
z = s + y # Error
```

# Numeric types

- Integers
- Floats
- Complex numbers
- Basic operations
  - + and -
  - \* , / and \*\*
  - implicit type conversions
  - be careful with integer division !

```
x = 2
```

```
x = 3.0
```

```
x = 4.0 + 5.0j
```

```
>>> 2.0 + 5 - 3
4.0
>>> 4.0**2 / 2.0 * (1.0 - 3j)
(8-24j)
>>> 1/2
0
>>> 1./2
0.5
```

# Strings

- Strings are enclosed by “ or ’
- Multiline strings can be defined with three double quotes

```
s1 = "very simple string"
s2 = 'same simple string'
s3 = "this isn't so simple string"
s4 = 'is this "complex" string?'
s5 = """This is a long string
expanding to multiple lines,
so it is enclosed by three "'s"""
```

- + and \* operators with strings:

```
>>> "Strings can be " + "combined"
'Strings can be combined'
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!'
```

# Data structures



- Lists and tuples
- Dictionaries

# List



- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place of the list

# Lists

- Defining lists

```
>>> l1 = [3, "egg", 6.2, 7]
>>> l2 = [12, [4, 5], 13, 1]
```

- Accessing list elements

```
>>> l1[0]
3
>>> l2[1]
[4, 5]
>>> l1[-1]
7
```

- Modifying list items

```
>>> l1[-2] = 4
>>> l1
[3, 'egg', 4, 7]
```

# Lists

- Adding items to list

```
>>> l1 = [9, 8, 7, 6]
>>> l1.append(11)
>>> l1
[9, 8, 7, 6, 11]
>>> l1.insert(1,16)
>>> l1
[9, 16, 8, 7, 6, 11]
>>> l2 = [5, 4]
>>> l1.extend(l2)
>>> l1
[9, 16, 8, 7, 6, 11, 5, 4]
```

- **+** and **\*** operators with lists:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

# Lists

- It is possible to access slices of lists

```
>>> l1 = [0, 1, 2, 3, 4, 5]
>>> l1[0:2]
[0, 1]
>>> l1[:2]
[0, 1]
>>> l1[3:]
[3, 4, 5]
>>> l1[0:6:2]
[0, 2, 4]
>>> l1[::-1]
[5, 4, 3, 2, 1, 0]
```

- Removing list items

```
>>> second = l1.pop(2)
>>> l1
[0, 1, 3, 4, 5]
>>> second
2
```

# Tuples

- Tuples are immutable lists
- Tuples are indexed and sliced like lists, but cannot be modified

```
>>> t1 = (1, 2, 3)
>>> t1[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
```

# Dictionaries

- Dictionaries are associative arrays
- Unordered list of key - value pairs
- Values are indexed by keys
- Keys can be strings or numbers
- Value can be any Python object

# Dictionaries

- Creating dictionaries

```
>>> grades = {'Alice' : 5, 'James' : 4, 'Carl' : 2}
>>> grades
{'James': 4, 'Alice': 5, 'Carl': 2}
```

- Accessing values

```
>>> grades['James']
4
```

- Adding items

```
>>> grades['Linda'] = 3
>>> grades
{'James': 4, 'Alice': 5, 'Carl': 2, 'Linda': 3}
>>> elements = {}
>>> elements['Fe'] = 26
>>> elements
{'Fe': 26}
```

# Variables



- Python variables are always references

```
>>> l1 = [1,2,3,4]
>>> l2 = l1
```

- **l1** and **l2** are references to the same list
- Modifying **l2** changes also **l1**!

```
>>> l2[0] = 0
>>> l1
[0, 2, 3, 4]
```

- Copy can be made by slicing the whole list

```
>>> l3 = l1[:]
>>> l3[-1] = 66
>>> l1
[0, 2, 3, 4]
>>> l3
[0, 2, 3, 66]
```

# Control structures

- **if – else** statements
- **while loops**
- **for** loops

# if statement

- **if** statement allows one to execute code block depending on condition
- Code blocks are defined by indentation, standard practice is to use four white spaces for indentation

```
if x > 0:  
    x += 1  
    y = 4 * x  
numbers[2] = x
```

- Boolean operators:

– **==**, **!=**, **>**, **<**, **>=**, **<=**

# if statement



- There can be multiple branches of conditions

```
if x == 0:
    print "x is zero"
elif x < 0:
    print "x is negative"
elif x > 100000:
    print "x is large"
else:
    print "x is something completely different"
```

# while loop

- **while** loop executes a code block as long as an expression is True

```
x = 0
cubes = {}
cube = 0
while cube < 100:
    cubes[x] = cube
    x += 1
    cube = x**3
```

# for loop

- **for** statement iterates over the items of any sequence (e.g. list)
- “Traditional” (C or Fortran) style

```
cars = ['Audi', 'BMW', 'Jaguar', 'Lada']  
n = len(cars)  
for i in range(n):  
    print "Car is", cars[i]
```

- Pythonic style

```
for car in cars:  
    print "Car is ", car
```

# for loop

- Several Python objects can be iterated over, e.g. dictionary keys

```
prices = {'Audi': 50, 'BMW' : 70, 'Lada' : 5}
for car in prices:
    print "Price of", car, "is", prices[car]
```

- iterating over many variables at once

```
coordinates = [[1.0, 0.0], [0.5, 0.5], [0.0, 1.0]]
for x, y in coordinates:
    print "X=", x, "Y=", y
```

```
prices = {'Audi': 50, 'BMW' : 70, 'Lada' : 5}
for car, price in prices.items():
    print "Price of", car, "is", price
```

# Functions and modules



- Defining functions
- Calling functions
- Importing modules

# Functions

- Function is block of code that can be referenced from other parts of the program
- Functions have arguments
- Functions can return values

# Function definition

```
def add(x, y):  
    result = x + y  
    return result  
  
x = 3.0  
y = 5.0  
sum = add(x, y)
```

- name of function is **add**
- **x** and **y** are arguments
- there can be any number of arguments and arguments can be any Python objects
- return value can be any Python object

# Modifying function arguments



- As Python variables are always references, function can modify the objects that arguments refer to

```
>>> def switch(mylist):
...     tmp = mylist[-1]
...     mylist[-1] = mylist[0]
...     mylist[0] = tmp
...
>>> l1 = [1,2,3,4,5]
>>> switch(l1)
>>> l1
[5, 2, 3, 4, 1]
```

- Side effects can be wanted or unwanted

# Modules

- Modules are extensions that can be imported to Python to provide additional functionality, e.g.
  - new data structures and data types
  - functions
- Python standard library includes several modules
- Several third party modules
- User defined modules

# Importing modules

- **import** statement

```
import math
x = math.exp(3.5)
```

```
import math as m
x = m.exp(3.5)
```

```
from math import exp, pi
x = exp(3.5) + pi
```

```
from math import *
x = exp(3.5) + sqrt(pi)
```

```
exp = 6.6
from math import *
x = exp(3.5) + sqrt(pi)
```

# Creating modules

- It is possible to make imports from own modules
- Define a function in file **mymodule.py**

```
def incx(x):  
    return x+1
```

- The function can now be imported in other .py files:

```
import mymodule # or from mymodule import inc  
  
y = mymodule.incx(1) # or y = incx(1)
```

# Summary

- Basic data structures
  - lists, tuples, dictionaries
- Variables are references to objects
- Control structures
  - **if - else**, **for**, **while**
- Functions help in reusing frequently used code blocks
- Additional functionality can be imported from modules



# Text processing and classes

# Text processing and file IO



- Working with files
- Reading and processing file contents
- String formatting and writing to files
- **pickle** module

# Opening and closing files

- `myfile = open(filename, mode)`
  - returns a handle to the file
- File can be opened for
  - reading: `mode='r'`  
(file has to exist)
  - writing: `mode='w'`  
(existing file is truncated)
  - appending: `mode='a'`
- Closing a file
  - `myfile.close()`

```
# open file for reading
infile = open('inp', 'r')

# open file for writing
outfile = open('out', 'w')

# open file for appending
appfile = open('cont', 'a')

#close files
infile.close()
...
```

# Reading from files

- A single line can be read from a file with the **readline()** -function

```
infile = open('inp', 'r')  
line = infile.readline()
```

- It is often convenient to iterate over all the lines in a file

```
infile = open('inp', 'r')  
for line in infile:  
    # process lines
```

# Processing lines

- Generally, a line read from a file is just a string
- A string can be split into a list of strings:

```
infile = open('inp', 'r')
for line in infile:
    line = line.split()
```

- Fields in a line can be assigned to variables and added to e.g. lists or dictionaries

```
for line in infile:
    line = line.split()
    x, y = float(line[1]), float(line[3])
    coords.append((x,y))
```

# Processing lines

- Sometimes one wants to process only files containing specific tags or substrings

```
for line in infile:
    if "Force" in line:
        line = line.split()
        x, y, z = float(line[1]), float(line[2]), float(line[3])
        forces.append((x,y,z))
```

- Other way to check for substrings:
  - `str.startswith()`, `str.endswith()`
- Python has also an extensive support for regular expressions in `re` -module

# String formatting

- Output is often wanted in certain format
- The string formatting operator `%` can be used to include in a string values with specific format

```
>>> x, y = 1.6666, 2.33333
>>> print "X is %4.2f and Y is %4.2f" % (x, y)
X is 1.67 and Y is 2.33
```

```
>>> x, y = 1.6666e4, 2.33333e6
>>> print "X is %4.2f and Y is %4.2f" % (x, y)
X is 16666.00 and Y is 2333330.00
>>> print "X is %4.2e and Y is %4.2e" % (x, y)
X is 1.67e+04 and Y is 2.33e+06
```

# String formatting

- Format specifier has the form `%[w][.p]t`
  - `w` is optional minimum width
  - `.p` gives optional precision (=number of decimals)
  - `t` is the conversion type
- Some conversion types
  - `s` : string
  - `d` : integer decimal
  - `f` : floating point decimal
  - `e` : floating point exponential

# Writing to file

- Data can be written to a file with **print** statements

```
outfile = open('out', 'w')
print >> outfile, "Header"
print >> outfile, "%6.3f %6.3f" % (x, y)
```

- File objects have also a **write()** function

```
outfile = open('out', 'w')
outfile.write("Header\n")
outfile.write("%6.3f %6.3f\n" % (x, y))
```

- The **write()** does not automatically add newline.
- File should be closed after writing is finished

# Pickle module

- Pickle allows one to dump arbitrary Python data structures to a file

```
import pickle

list1 = [5.0, [3, 4], 'monty']
my_dict1 = {'foo' : 4.0, 'bar' : list1}
f = open('mydata.pckl', 'w')
pickle.dump(my_dict1,f)
```

- Data can be retrieved later

```
import pickle

f = open('mydata.pckl', 'r')
my_dict1 = pickle.load(f)
```

# Summary

- Files are opened and closed with `open()` and `close()`
- Lines can be read by iterating over the file object
- Lines can be split into lists and check for existence of specific substrings
- String formatting operators can be used for obtaining specific output
- File output can be done with `print` or `write()`

# Classes



- Classes facilitate object oriented programming with Python
- Here, we consider classes only as data containers
- Data structures beyond lists and dictionaries

# Example: car

- Information about a car might contain
  - brand
  - model
  - horse power
  - maximum speed

# Class definition

- Before a class can be used, it has to be defined

```
class Car:  
    def __init__(self, brand, model, hp, speed):  
        self.brand = brand  
        self.model = model  
        self.hp = hp  
        self.speed = speed
```

- The class members (attributes, methods in general) are referred by *self.XXX*
- `__init__` is a special class method which is called when creating an instance of class
- Class attributes can be any Python objects

# Instance

- class definition does not create any objects
- Instances of class are created by “calling” the class

```
car1 = Car('Audi', 'A4', 150, 220)
```

- The attributes designated by *self.X* in the definition can now be referred by:

```
print "Brand and model are", car1.brand, car1.model
```

- Instances can be used as list or dictionary items

```
car1 = Car('Audi', 'A4', 150, 220)
car2 = Car('BMW', '250', 120, 230)
car_list = [car1, car2]
print car_list[0].hp
```



# Numpy

# Numpy – fast array interface



- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - some linear algebra, random numbers

# Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

# Numpy arrays

- Creating an array

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

# Numpy arrays

- More ways for creating arrays

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25,  0.   ,  2.25,  4.5  ])
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>> d = np.ones((2, 4))
>>> d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

# Numpy arrays

- Slicing is similar to lists

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

- Simple assignment creates references to arrays, use **copy()** for real copying of arrays

```
a = np.arange(10)
b = a # reference, changing values in b changes a
b = a.copy() # true copy
```

# Array operations

- Most operations for numpy arrays are done element-wise
  - $+$ ,  $-$ ,  $*$ ,  $/$ ,  $**$
- Numpy has special functions which can work with array arguments
  - $\sin$ ,  $\cos$ ,  $\exp$ ,  $\sqrt{\quad}$ ,  $\log$ , ...

```
>>> a = np.linspace(-pi, pi, 8)
>>> s = sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
>>> s = np.sin(a)
```

# Broadcasting

- Arithmetic operations in Numpy are usually done on element-by-element basis
- If array shapes are different, the smaller array may be **broadcasted** into a larger shape
- Broadcasting can help in vectorizing operations: loops in C instead of Python
- More efficient memory usage

# Broadcasting

- Broadcasting rules:
  - Numpy compares shapes element-wise starting from trailing dimension
  - Dimensions are compatible if they are:
    - equal
    - one of them is 1
  - When one dimension is 1, the larger is used
- Simplest example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.] )
```

b is treated as it was `np.array([2.0, 2.0, 2.0])`

# Broadcasting examples

- Number of dimensions can differ

```
A      (3d array): 256 x 256 x 3
B      (1d array):          3
Result (3d array): 256 x 256 x 3
```

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):          7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

- Non-matching shapes

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):          2 x 1
B      (3d array):  8 x 4 x 3 # second from last
                             dimensions mismatched
```

# Linear algebra

- Numpy can calculate matrix and vector products efficiently
  - `dot`, `vdot`, ...
- Eigenproblems
  - `linalg.eig`, `linalg.eigvals`, ...
- Linear systems and matrix inversion
  - `linalg.solve`, `linalg.inv`

```
>>> A = np.array(((2, 1), (1, 3)))
>>> B = np.array((-2, 4.2), (4.2, 6))
>>> C = np.dot(A, B)
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b) # solve C x = b
array([ 0.04453441,  0.06882591])
```

# Numpy performance

- Matrix multiplication
  - $C = A * B$
  - matrix dimension 200
  - pure python: 5.30 s
  - naive C: 0.09 s
  - numpy.dot: 0.01 s

# Summary



- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers



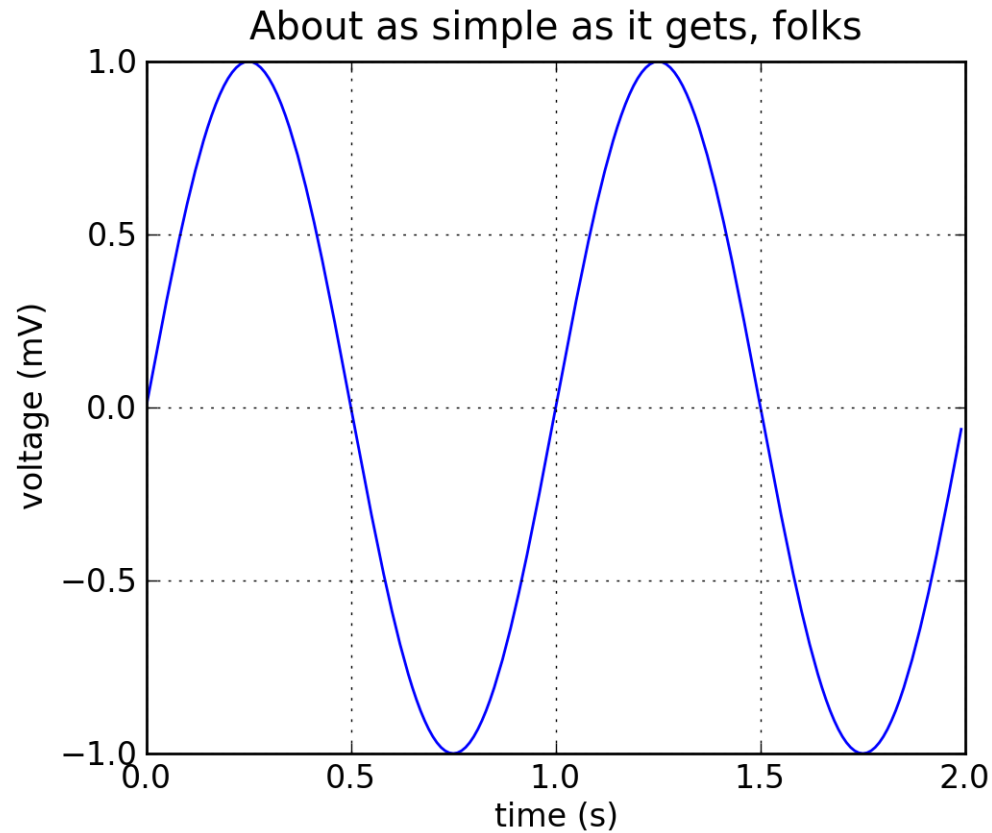
# Matplotlib

# Matplotlib

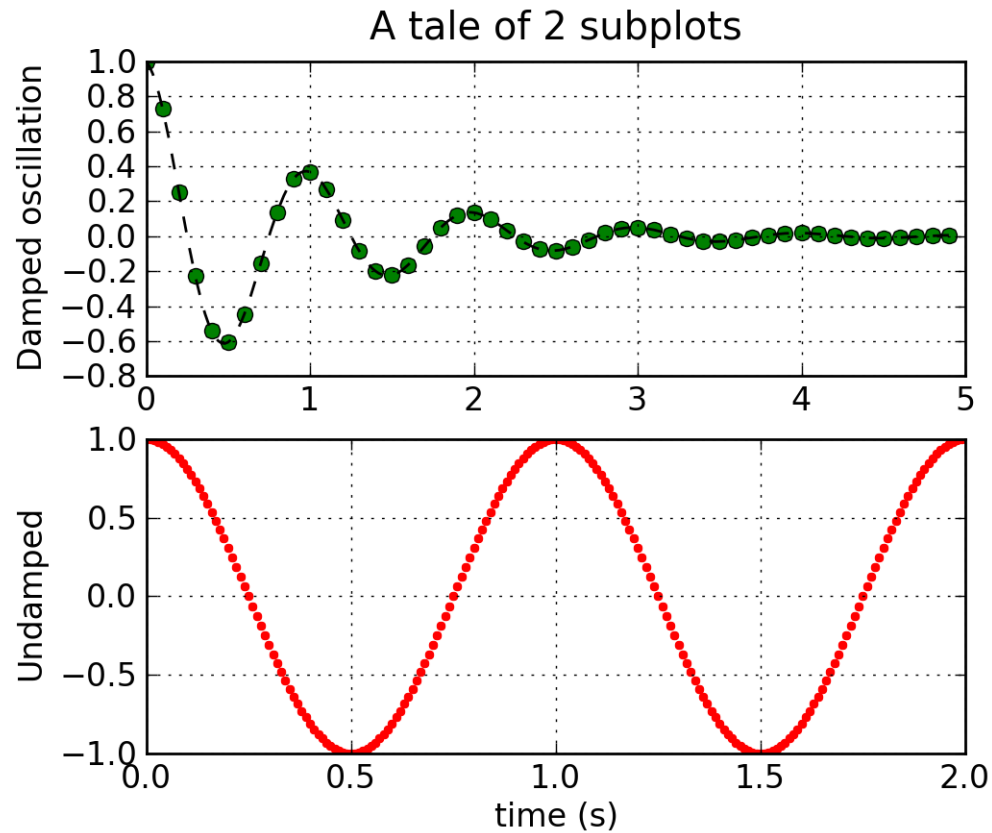


- 2D plotting library for python
- Can be used in scripts and in interactive shell
- Publication quality in various hardcopy formats
- “Easy things easy, hard things possible”
- Some 3D functionality

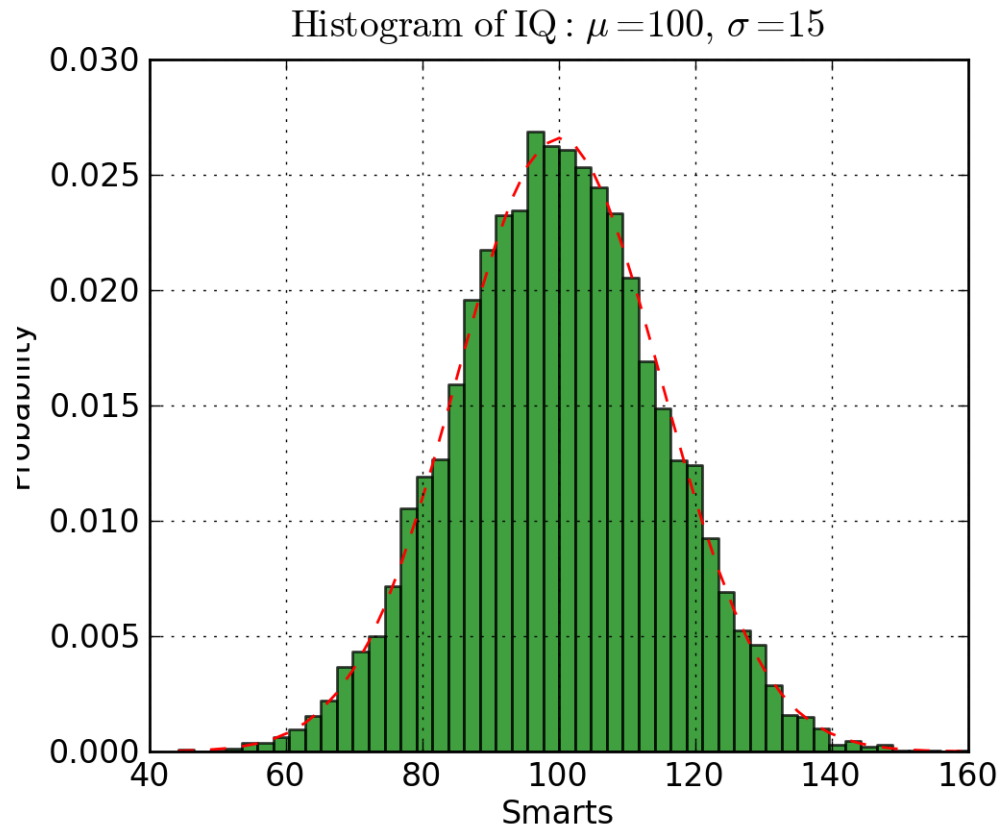
# Simple plot



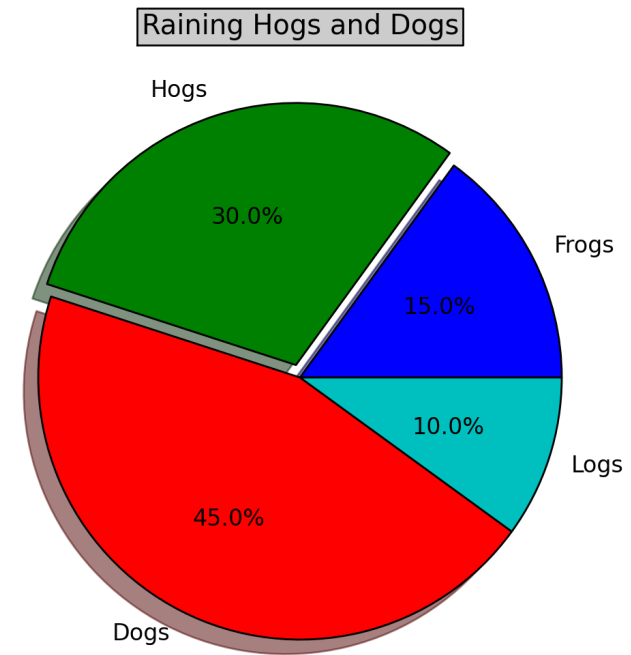
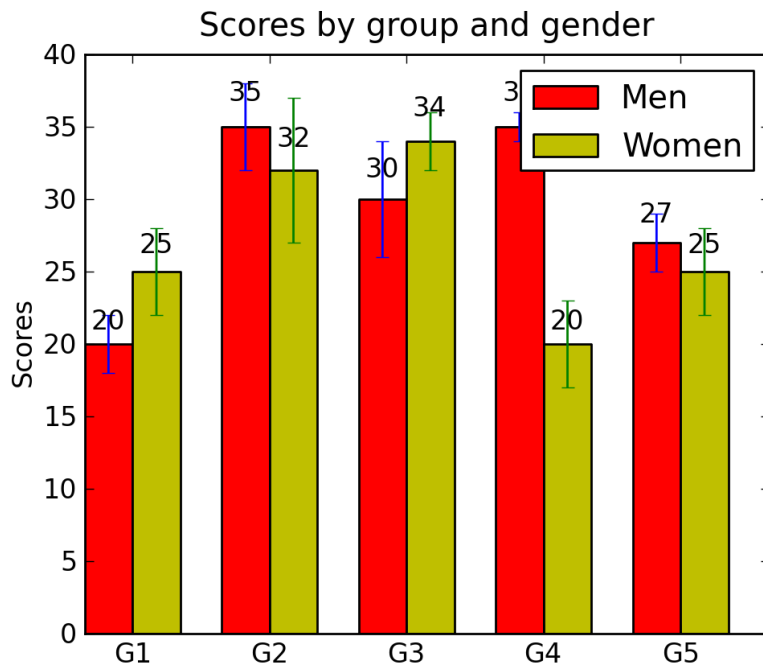
# Multiple subplots



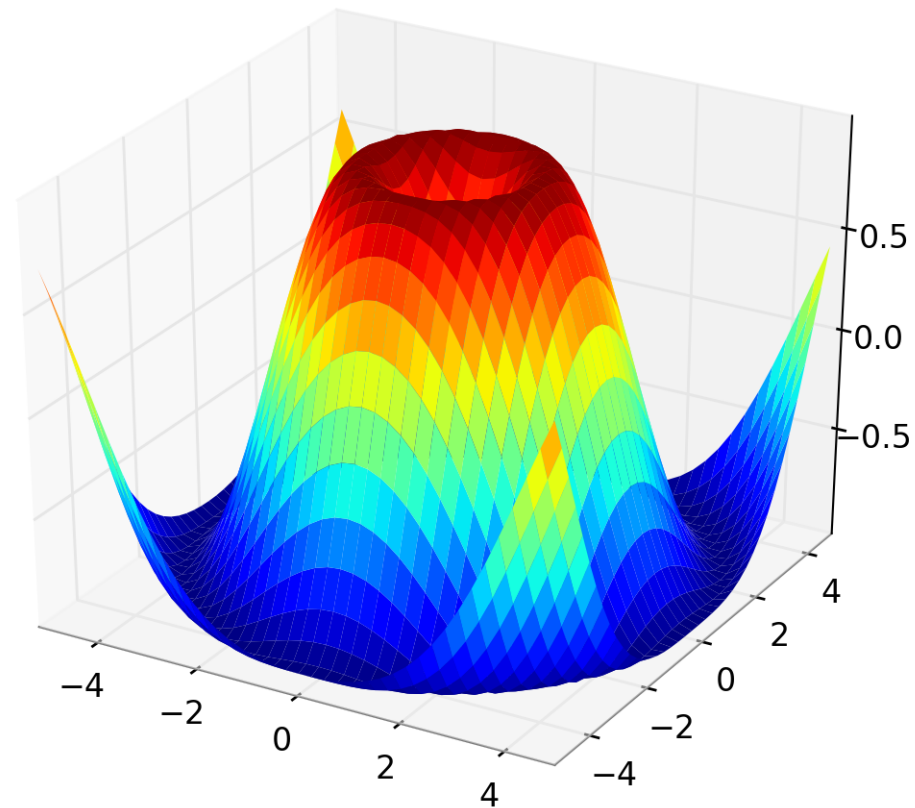
# Histograms



# Bar and pie charts



# 3D plots





# Scipy

# Scipy – Scientific tools for Python



- Scipy is a Python package containing several tools for scientific computing
- Modules for:
  - statistics, optimization, integration, interpolation
  - linear algebra, Fourier transforms, signal and image processing
  - ODE solvers, special functions
  - ...
- Vast package, reference guide is currently 632 pages
- Scipy is built on top of Numpy

# Library overview



- Clustering package (`scipy.cluster`)
- Constants (`scipy.constants`)
- Fourier transforms (`scipy.fftpack`)
- Integration and ODEs (`scipy.integrate`)
- Interpolation (`scipy.interpolate`)
- Input and output (`scipy.io`)
- Linear algebra (`scipy.linalg`)
- Maximum entropy models (`scipy.maxentropy`)
- Miscellaneous routines (`scipy.misc`)
- Multi-dimensional image processing (`scipy.ndimage`)
- Orthogonal distance regression (`scipy.odr`)
- Optimization and root finding (`scipy.optimize`)
- Signal processing (`scipy.signal`)
- Sparse matrices (`scipy.sparse`)
- Sparse linear algebra (`scipy.sparse.linalg`)
- Spatial algorithms and data structures (`scipy.spatial`)
- Special functions (`scipy.special`)
- Statistical functions (`scipy.stats`)
- Image Array Manipulation and Convolution (`scipy.stsci`)
- C/C++ integration (`scipy.weave`)

# Integration

- Routines for numerical integration
  - single, double and triple integrals
- Function to integrate can be given by function object or by fixed samples

```
from scipy.integrate import quad, simps, inf

def f(x):
    return exp(-x)

x = np.linspace(0, 1, 20)
y = np.exp(-x)
int1 = quad(f, 0, 1)      # integrate function object
int2 = simps(y, x)       # integrate function given by
samples
int3 = quad(f, 0, inf)   # integrate up to infinity
```

# Optimization

- Several classical optimization algorithms
  - Quasi-Newton type optimizations
  - Least squares fitting
  - Simulated annealing
  - General purpose root finding
  - ...

Rosenbrock function  $f(\mathbf{x}) = \sum_{i=1}^{N-1} (x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$

```
>>> from scipy.optimize import fmin
>>> def rosen(x):
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin(rosen, x0, xtol=1e-8)
```

# Special functions



- Scipy contains huge set of special functions
  - Bessel functions
  - Legendre functions
  - Gamma functions
  - ...

```
>>> from scipy.special import *  
>>> x = np.linspace(0, 5, 20)  
>>> plot(x, jv(1, x))  
>>> plot(x, jv(2, x))
```

# Linear algebra

- Wider set of linear algebra operations than in Numpy
  - decompositions, matrix exponentials
- Routines also for sparse matrices
  - storage formats
  - iterative algorithms

```
import numpy as np
from scipy.sparse.linalg import LinearOperator, cg

def mv(v):
    return np.array([ 2*v[0], 3*v[1]])

A = LinearOperator( (2,2), matvec=mv, dtype=float ) # define matrix-vector product
b = np.array((4.0, 1.0))
x = cg(A, b)      # Solve linear equation Ax = b with conjugate gradient
```



# Mpi4py

# Parallel programming with MPI



- Message passing paradigm:
  - independent processes
  - processes communicate by exchanging messages
  - processes have an id-number (rank)
  - communicator: group of processes
- MPI – message passing interface
  - standard defines C and Fortran interfaces
- Mpi4py provides Python interface

# Simple examples

- Parallel “hello”, no communication

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

print "I am rank", rank
```

- Communicating Python objects (pickle under hood)

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

# Simple examples

- Numpy arrays (nearly C speed)

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(100, dtype=numpy.float)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float)
    comm.Recv(data, source=0, tag=13)
```

- Note the difference between upper/lower case!
  - send/recv: general Python objects, slow
  - Send/Recv: continuous arrays, fast



# C - extensions

# C - extensions

- Some times there are time critical parts of code which would benefit from compiled language
- It is relatively straightforward to create a Python interface to C-functions
- Some tools can simplify the interfacing
  - SWIG
  - Cython, pyrex

# Passing a Numpy array to C



- Python

```
import myext

a = np.array(...)
myext.myfunc(a)
```

- C: myext.c

```
#include <Python.h>
#define NO_IMPORT_ARRAY
#include <numpy/arrayobject.h>

PyObject* my_C_func(PyObject *self, PyObject *args)
{
    PyArrayObject* a;
    if (!PyArg_ParseTuple(args, "O", &a))
        return NULL;
    ...
}
```

# Accessing array data



- myext.c

```
...
PyArrayObject* a;
int size = PyArray_SIZE(a);
double *data = (double *) a->data;
for (int i=0; i < size; i++)
{
    /* Process data */
}
Py_RETURN_NONE;
}
```

# Defining the Python interface



- myext.c

```
static PyMethodDef functions[] = {
    {"myfunc", my_C_func, METH_VARARGS, 0},
    {0, 0, 0, 0}
};

PyMODINIT_FUNC initemptyext(void)
{
    (void) Py_InitModule("myext", functions);
}
```

- Build as a shared library

```
gcc -shared -o myext.so -I/usr/include/python2.6 -fPIC myext.c
```

- Use in Python script

```
import myext

a = np.array(...)
myext.myfunc(a)
```