

Introduction to Scientific Computing

Adjusted from:

<http://www.nanohub.org/resources/?id=99>

Original Authors are: Eric Jones and Travis Oliphant

Many excellent resources on the web

>> google: "learn python"

some good example:

<http://www.diveintopython.org/toc/index.html>

<http://www.scipy.org/Documentation>

Topics

- Introduction to Python
- Numeric Computing
- SciPy and its libraries

What Is Python?

ONE LINER

Python is an interpreted programming language that allows you to do almost anything possible with a compiled language (C/C++/Fortran) without requiring all the complexity.

PYTHON HIGHLIGHTS

- Automatic garbage collection
- Dynamic typing
- Interpreted and interactive
- Object-oriented
- “Batteries Included”
- Free
- Portable
- Easy to Learn and Use
- Truly Modular

Who is using Python?

NATIONAL SPACE TELESCOPE LABORATORY

Data processing and calibration for instruments on the Hubble Space Telescope.

INDUSTRIAL LIGHT AND MAGIC

Digital Animation

PAINT SHOP PRO 8

Scripting Engine for JASC
PaintShop Pro 8 photo-editing software

CONOCOPHILLIPS

Oil exploration tool suite

LAWRENCE LIVERMORE NATIONAL LABORATORIES

Scripting and extending parallel physics codes. pyMPI is their doing.

WALT DISNEY

Digital animation development environment.

REDHAT

Anaconda, the Redhat Linux installer program, is written in Python.

ENTHOUGHT

Geophysics and Electromagnetics engine scripting, algorithm development, and visualization

Language Introduction

Interactive Calculator

```
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variables type
>>> type(a) ↑
<type 'int'>
# an arbitrarily long integer
>>> a = 1203405503201
>>> a
1203405503201L
>>> type(a) ↑
<type 'long'>
>>>> type(a).__name__=='long'
True
>>>> print type.__doc__
type(name, bases, dict) ↑
```

```
# real numbers
>>> b = 1.2 + 3.1
>>> b
4.2999999999999998
>>> type(b) ↑
<type 'float'>
# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j) ↑
```

The four numeric types in Python on 32-bit architectures are:

integer (4 byte)

long integer (any precision) ↑

float (8 byte like C's double) ↑

complex (16 byte) ↑

The Numeric module, which we will see later, supports a larger number of numeric types.



Complex Numbers

CREATING COMPLEX NUMBERS

```
# Use "j" or "J" for imaginary
# part. Create by "(real+imagj)",
# or "complex(real, imag)".
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> (1+2j) / (1+1j)
(1.5+0.5j)
```

EXTRACTING COMPONENTS

```
# to extract real and im
# component
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

ABSOLUTE VALUE

```
>>> a=1.5+0.5j
>>> abs(a)
1.5811388
```

Strings

CREATING STRINGS

```
# using double quotes
>>> s = "hello world"
>>> print s
hello world
# single quotes also work
>>> s = 'hello world'
>>> print s
hello world
```

STRING OPERATIONS

```
# concatenating two strings
>>> "hello " + "world"
'hello world'

# repeating a string
>>> "hello " * 3
'hello hello hello '
```

STRING LENGTH

```
>>> s = "12345"
>>> len(s)
5
```

FORMAT STRINGS

```
# the % operator allows you
# to supply values to a
# format string. The format
# string follows
# C conventions.
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> s = "%s %f, %d" %
(s,x,y) ↑
>>> print s
some numbers: 1.34, 2
```


The strings

```

>>> s = "hello world"
>>> s.split() ↑
['hello', 'world']

>>> ' '.join(s.split()) ↑
hello world

>>>
s.replace('world', 'Mars') ↑
'hello Mars'

# strip whitespace
>>> s = "\t    hello    \n"
>>> s.strip() ↑
'hello'

```

Regular expressions:

```

re.match(regex, subject)
re.search(regex, subject) ↑
re.group() ↑
re.groups() ↑
re.sub(regex, replacement, sub) ↑

>>import re
>>s="The time is 12:30pm!"
>>m=re.match(".*time is (.*)pm", s)
>>m.group(1) ↑
'12:30'
>>m.groups() ↑
↑), '12:30' (
>>m=re.search(r'time.*(\d+:\d+)pm', s)
>>m.group(1) ↑
'12:30'
>>re.sub(r'\d+:\d+', '2:10', s) ↑
'The time is 2:10pm!'

```

Multi-line Strings

```
# triple quotes are used
# for mutli-line strings
>>> a = """hello
... world"""
>>> print a
hello
world
```

```
# multi-line strings using
# "\" to indicate
continuation
>>> a = "hello " \
...     "world"
>>> print a
hello world
```

```
# including the new line
>>> a = "hello\n" \
...     "world"
>>> print a
hello
world
```

List objects

LIST CREATION WITH BRACKETS

```
>>> l = [10,11,12,13,14]
>>> print l
[10, 11, 12, 13, 14]
```

CONCATENATING LIST

```
# simply use the + operator
>>> [10, 11] + [12,13]
[10, 11, 12, 13]
```

REPEATING ELEMENTS IN LISTS

```
# the multiply operator
# does the trick.
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

range(start, stop, step)

```
# the range method is helpful
# for creating a sequence
```

```
>>> range(5) ↵
[0, 1, 2, 3, 4]
```

```
>>> range(2,7) ↵
[2, 3, 4, 5, 6]
```

```
>>> range(2,7,2) ↵
[2, 4, 6]
```

Indexing

RETRIEVING AN ELEMENT

```
# list
# indices: 0 1 2 3 4
>>> l = [10,11,12,13,14]
>>> l[0]
10
```

SETTING AN ELEMENT

```
>>> l[1] = 21
>>> print l
[10, 21, 12, 13, 14]
```

OUT OF BOUNDS

```
>>> l[10]
Traceback (innermost last):
File "<interactive input>",line 1,in ?
IndexError: list index out of range
```

NEGATIVE INDICES

```
# negative indices count
# backward from the end of
# the list.
#
# indices: -5 -4 -3 -2 -1
>>> l = [10,11,12,13,14]

>>> l[-1]
14
>>> l[-2]
13
```



The first element in an array has index=0 as in C. Take note Fortran programmers!

More on list objects

LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,
# string, and another list.
>>> l = [10, 'eleven', [12, 13]]
>>> l[1]
'eleven'
>>> l[2]
[12, 13]

# use multiple indices to
# retrieve elements from
# nested lists.
>>> l[2][0]
12
```

LENGTH OF A LIST

```
>>> len(l)
3
```

DELETING OBJECT FROM LIST

```
# use the del keyword
>>> del l[2]
>>> l
[10, 'eleven']
```

DOES THE LIST CONTAIN x ?

```
# use in or not in
>>> l = [10, 11, 12, 13, 14]
>>> 13 in l
1
>>> 13 not in l
0
```

Slicing

var [lower : upper]

Slices extract a portion of a sequence by specifying a lower and upper bound. The extracted elements start at lower and go up to, *but do not include*, the upper element. Mathematically the range is [lower,upper).

SLICING LISTS

```
# indices: 0  1  2  3  4
>>> l = [10,11,12,13,14]
# [10,11,12,13,14]
>>> l[1:3]
[11, 12]

# negative indices work also
>>> l[1:-2]
[11, 12]
>>> l[-4:3]
[11, 12]
```

OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list.

# grab first three elements
>>> l[:3]
[10,11,12]
# grab last two elements
>>> l[-2:]
[13,14]
```

A few methods for list objects

`some_list.append(x)`

Add the element `x` to the end of the list, `some_list`.

`some_list.count(x)`

Count the number of times `x` occurs in the list.

`some_list.index(x)`

Return the index of the first occurrence of `x` in the list.

`some_list.remove(x)`

Delete the first occurrence of `x` from the list.

`some_list.reverse()`

Reverse the order of elements in the list.

`some_list.sort(cmp)`

By default, sort the elements in ascending order. If a compare function is given, use it to sort the list.

List methods in action

```
>>> l = [10,21,23,11,24]

# add an element to the list
>>> l.append(11) ↵
>>> print l
[10,21,23,11,24,11]
```

```
# how many 11s are there?
>>> l.count(11) ↵
2
```

```
# where does 11 first occur?
>>> l.index(11) ↵
3
```

```
# remove the first 11
>>> l.remove(11) ↵
>>> print l
[10,21,23,24,11]
```

```
# sort the list
>>> l.sort() ↵
>>> print l
[10,11,21,23,24]
```

```
# reverse the list
>>> l.reverse() ↵
>>> print l
[24,23,21,11,10]
```


Mutable vs. Immutable

MUTABLE OBJECTS

```
# Mutable objects, such as
# lists, can be changed
# in-place.

# insert new values into list
>>> l = [10,11,12,13,14]
>>> l[1:3] = [5,6]
>>> print l
[10, 5, 6, 13, 14]
```



The `cStringIO` module treats strings like a file buffer and allows insertions. It's useful when working with large strings or when speed is paramount.

IMMUTABLE OBJECTS

```
# Immutable objects, such as
# strings, cannot be changed
# in-place.

# try inserting values into
# a string
>>> s = 'abcde'
>>> s[1:3] = 'xy'
Traceback (innermost last):
  File "<interactive input>",line 1,in ?
TypeError: object doesn't support
      slice assignment

# here's how to do it
>>> s = s[:1] + 'xy' + s[3:]
>>> print s
'axyde'
```

Dictionaries

Dictionaries store *key/value* pairs. Indexing a dictionary by a *key* returns the *value* associated with it.

DICTIONARY EXAMPLE

```
# create an empty dictionary using curly brackets
>>> record = {}
>>> record['first'] = 'Jmes'
>>> record['last'] = 'Maxwell'
>>> record['born'] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
# create another dictionary with initial entries
>>> new_record = {'first': 'James', 'middle': 'Clerk'}
# now update the first dictionary with values from the new one
>>> record.update(new_record)
>>> print record
{'first': 'James', 'middle': 'Clerk', 'last': 'Maxwell', 'born': 1831}
```

A few dictionary methods

`some_dict.clear()`

Remove all key/value pairs from the dictionary, `some_dict`.

`some_dict.copy()`

Create a copy of the dictionary

`some_dict.has_key(x)`

Test whether the dictionary contains the key `x`.

`some_dict.keys()`

Return a list of all the keys in the dictionary.

`some_dict.values()`

Return a list of all the values in the dictionary.

`some_dict.items()`

Return a list of all the key/value pairs in the dictionary.

Dictionary methods in action

```
>>> d = {'cows': 1, 'dogs':
5, ...     'cats': 3}

# create a copy.
>>> dd = d.copy() ↑
>>> print dd
{'dogs': 5, 'cats': 3, 'cows': 1}

# test for chickens.
>>> d.has_key('chickens') ↑
0

# get a list of all keys
>>> d.keys() ↑
['cats', 'dogs', 'cows']
```

```
# get a list of all values
>>> d.values() ↑
[3, 5, 1]

# return the key/value pairs
>>> d.items() ↑
[('cats', 3), ('dogs', 5),
 ('cows', 1)]

# clear the dictionary
>>> d.clear() ↑
>>> print d
{}
```

Tuples

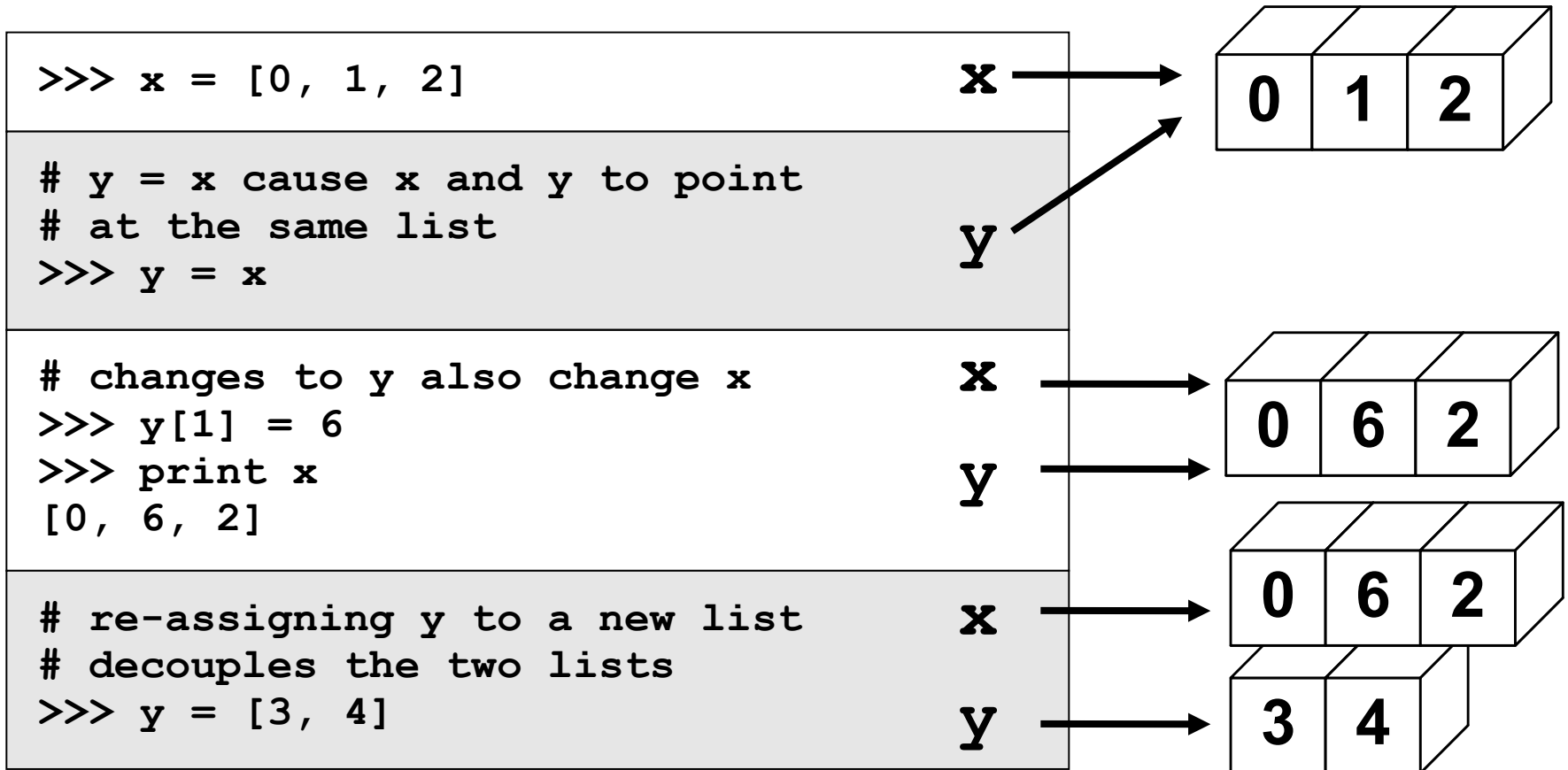
Tuples are a sequence of objects just like lists. Unlike lists, tuples are immutable objects. While there are some functions and statements that require tuples, they are rare. A good rule of thumb is to use lists whenever you need a generic sequence.

TUPLE EXAMPLE

```
# tuples are built from a comma separated list enclosed by ( )
>>> t = (1, 'two')
>>> print t
(1, 'two')
>>> t[0]
1
# assignments to tuples fail
>>> t[0] = 2
Traceback (innermost last):
File "<interactive input>", line 1, in ?
TypeError: object doesn't support item assignment
```

Assignment

Assignment creates object references.



Multiple assignments

```
# creating a tuple without
() ↑
>>> d = 1,2,3
>>> d
↑)3 ,2 ,1(
```

```
# multiple assignments
>>> a,b,c = 1,2,3
>>> print b
2
```

```
# multiple assignments from a
# tuple
>>> a,b,c = d
>>> print b
2
```

```
# also works for lists
>>> a,b,c = [1,2,3]
>>> print b
2
```

If statements

`if/elif/else` provide conditional execution of code blocks.

IF STATEMENT FORMAT

```

if <condition>:
    <statements>
elif <condition>:
    <statements>
else:
    <statements>
    
```

IF EXAMPLE

```

# a simple if statement
>>> x = 10
>>> if x > 0:
...     print 1
... elif x == 0:
...     print 0
... else:
...     print -1
... < hit return >
1
    
```


Test Values

- True means any non-zero number or non-empty object
- False means not true: zero, empty object, or **None**

EMPTY OBJECTS

```
# empty objects evaluate false
>>> x = []
>>> if x:
...     print 1
... else:
...     print 0
... < hit return >
0
```

For loops

For loops iterate over a sequence of objects.

```
for <loop_var> in <sequence>:  
    <statements>
```

TYPICAL SCENARIO

```
>>> for i in range(5):  
...     print i,  
... < hit return >  
0 1 2 3 4
```

LOOPING OVER A STRING

```
>>> for i in 'abcde':  
...     print i,  
... < hit return >  
a b c d e
```

LOOPING OVER A LIST

```
>>> l=['dogs', 'cats', 'bears']  
>>> accum = ''  
>>> for item in l:  
...     accum = accum + item  
...     accum = accum + ' '  
... < hit return >  
>>> print accum  
dogs cats bears
```

While loops

While loops iterate until a condition is met.

```
while <condition>:
    <statements>
```

WHILE LOOP

```
# the condition tested is
# whether lst is empty.
>>> lst = range(3)
>>> while lst:
... print lst
... lst = lst[1:]
... < hit return >
[0, 1, 2]
[1, 2]
[2]
```

BREAKING OUT OF A LOOP

```
# breaking from an infinite
# loop.
>>> i = 0
>>> while 1:
... if i < 3:
...     print i,
...     else:
...         break
...     i = i + 1
... < hit return >
0 1 2
```

Anatomy of a function

The keyword **def** indicates the start of a function.

Function arguments are listed separated by commas. They are passed by *assignment*. More on this later.

```
def add(arg0, arg1):
    a = arg0 + arg1
    return a
```

Indentation is used to indicate the contents of the function. It is *not* optional, but a part of the syntax.

A colon (**:**) terminates the function definition.

An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value **None**.

Our new function in action

```
# We'll create our function
# on the fly in the
# interpreter.
>>> def add(x,y):
...     a = x + y
...     return a
```

```
# test it out with numbers
>>> x = 2
>>> y = 3
>>> add(x,y)
5
```

```
# how about strings?
>>> x = 'foo'
>>> y = 'bar'
>>> add(x,y)
'foobar'
```

```
# functions can be assigned
# to variables
>>> func = add
>>> func(x,y)
'foobar'
```

```
# how about numbers and strings?
>>> add('abc',1)
Traceback (innermost last):
File "<interactive input>", line 1, in ?
File "<interactive input>", line 2, in add
TypeError: cannot add type "int" to string
```

More about functions

```
# Every function returns
# a value (or NONE) ↴
# but you don't need to
# specify returned type!

# Function documentation
>>> def add(x,y):
...     """this function
...     adds two numbers"""
...     a = x + y
...     return a
```

```
# You can always retrieve
# function documentation
>>> print add.__doc__
```

```
this function
adds two numbers
```

```
# FUNCTIONAL PROGRAMMING:
# "map(function, sequence)"
>>> def cube(x): return
x*x*x ...
>>> map(cube, range(1, 6))
[1, 8, 27, 64, 125]

# "reduce (function,
sequence)"
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55

# "filter (function,
sequence)"
>>> def f(x): return x % 2 !=
0
...
>>> filter(f, range(2, 10))
[3, 5, 7, 9]
```

Even more on functions

```
# buld-in function "dir" is
# used to list all
# definitions in a module
>>> import scipy
>>> dir(scipy) ↑
.....
...<a lot of stuf>...
.....
```

```
# Lambda function:
# Python supports one-line mini-
# functions on the fly.
# Borrowed from Lisp, lambda
# functions can be used anywhere
# a function is required.
>>> def f(x): return x*x
>>> map(f, range(5)) ↑
[0, 1, 4, 9, 16]
>>> map(lambda x: x*x, range(5)) ↑
[0, 1, 4, 9, 16]
```

```
# more on lambda function:
>>> a=range(10)
>>> a.sort(lambda x,y: cmp(y,x)) ↑
>>> print a
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> map(lambda x: x*2+10, range(5))
[10, 12, 14, 16, 18]
>>> print reduce(lambda x,y: x+y, range(5))
10
```

Modules

EX1.PY

```
# ex1.py

PI = 3.1416

def sum(lst):
    tot = lst[0]
    for value in lst[1:]:
        tot = tot + value
    return tot

l = [0,1,2,3]
print sum(l), PI
```

FROM SHELL

```
[ej@bull ej]$ python ex1.py
6, 3.1416
```

FROM INTERPRETER

```
# load and execute the module
>>> import ex1
6, 3.1416
# get/set a module variable.
>>> ex1.PI
3.1415999999999999
>>> ex1.PI = 3.14159
>>> ex1.PI
3.1415899999999999
# call a module variable.
>>> t = [2,3,4]
>>> ex1.sum(t)
9
```


Modules cont.

INTERPRETER

```
# load and execute the module
>>> import ex1
6, 3.1416
< edit file >
# import module again
>>> import ex1
# nothing happens!!!

# use reload to force a
# previously imported library
# to be reloaded.
>>> reload(ex1) ↑
10, 3.14159
```

EDITED EX1.PY

```
# ex1.py version 2

PI = 3.14159

def sum(lst):
    tot = 0
    for value in lst:
        tot = tot + value
    return tot

l = [0,1,2,3,4]
print sum(l), PI
```

Modules cont. 2

Modules can be executable scripts or libraries or both.

EX2.PY

```

" An example module "

PI = 3.1416

def sum(lst):
    """ Sum the values in a
        list.
    """
    tot = 0
    for value in lst:
        tot = tot + value
    return tot

```

EX2.PY CONTINUED

```

def add(x,y):
    " Add two values."
    a = x + y
    return a

def test():
    l = [0,1,2,3]
    assert( sum(l) == 6)
    print `test passed`

# this code runs only if this
# module is the main program
if __name__ == `__main__`:
    test()

```

Classes

SIMPLE PARTICLE CLASS

```
>>> class particle:
...     # Constructor method
...     def __init__(self, mass, velocity):
...         # assign attribute values of new object
...         self.mass = mass
...         self.velocity = velocity
...     # method for calculating object momentum
...     def momentum(self):
...         return self.mass * self.velocity
...     # a "magic" method defines object's string representation
...     def __repr__(self):
...         msg = "(m:%2.1f, v:%2.1f)" % (self.mass, self.velocity)
...         return msg
```

EXAMPLE

```
>>> a = particle(3.2, 4.1)
>>> a
(m:3.2, v:4.1)
>>> a.momentum()
13.119999999999999
```

Classes

We want to solve equation $(x-x_0)^2=0$

First try:

```
..x0=3
..def fx(x):
..    return (x-x0)**2
..
..print optimize.fsolve(fx,1.)
```

We used global variable `x0`! How to eliminate global variable?

Second try:

```
..class Fx:
..    def __init__(self, x0):
..        self.x0 = x0
..    def __call__(self, x):
..        return (x-self.x0)**2
..
..fx = Fx(3.)
..print optimize.fsolve(fx,1.)
```

Classes

More elegant solution provided by scipy:

```
..def fx(x,x0):  
..    return (x-x0)**2  
..  
..print optimize.fsolve(fx,1.,args=(3.,))
```

Reading files

FILE INPUT EXAMPLE

```
>>> results = []
>>> f = open('rcs.txt', 'r')

# read lines and discard header
>>> lines = f.readlines()[1:]
>>> f.close()

>>> for l in lines:
...     # split line into fields
...     fields = line.split()
...     # convert text to numbers
...     freq = float(fields[0])
...     vv = float(fields[1])
...     hh = float(fields[2])
...     # group & append to results
...     all = [freq, vv, hh]
...     results.append(all)
... < hit return >
```

PRINTING THE RESULTS

```
>>> for i in results: print i
[100.0, -20.30..., -31.20...]
[200.0, -22.70..., -33.60...]
```

EXAMPLE FILE: RCS.TXT

```
#freq (MHz)   vv (dB)   hh (dB)
100           -20.3    -31.2
200           -22.7    -33.6
```

More compact version

ITERATING ON A FILE AND LIST COMPREHENSIONS

```

>>> results = []
>>> f = open('rcs.txt', 'r')
>>> f.readline()
'#freq (MHz)  vv (dB)  hh (dB)\n'
>>> for l in f:
...     all = [float(val) for val in l.split()]
...     results.append(all)
... < hit return >
>>> for i in results:
...     print i
... < hit return >

```

EXAMPLE FILE: RCS.TXT

```

#freq (MHz)  vv (dB)  hh (dB)
100          -20.3    -31.2
200          -22.7    -33.6

```

Same thing, one line

OBFUSCATED PYTHON CONTEST...

```
>>> print [[float(val) for val in l.split()] for
...         l in open("rcs.txt","r")
...         if l[0] !="#"]
```

EXAMPLE FILE: RCS.TXT

#freq (MHz)	vv (dB)	hh (dB) ↑
100	-20.3	-31.2
200	-22.7	-33.6

Scipy version

numerical multicolumn data...

```
>>>from scipy import *  
...data = loadtxt('rcs.txt')  
...print data
```

EXAMPLE FILE: RCS.TXT

#freq (MHz)	vv (dB)	hh (dB) ↑
100	-20.3	-31.2
200	-22.7	-33.6

Sorting

THE CMP METHOD

```
# The builtin cmp(x,y)
# function compares two
# elements and returns
# -1, 0, 1
# x < y --> -1
# x == y --> 0
# x > y --> 1
>>> cmp(0,1)
-1

# By default, sorting uses
# the builtin cmp() method
>>> x = [1,4,2,3,0]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4]
```

CUSTOM CMP METHODS

```
# define a custom sorting
# function to reverse the
# sort ordering
>>> def descending(x,y):
...     return -cmp(x,y)

# Try it out
>>> x.sort(descending)
>>> x
[4, 3, 2, 1, 0]
```

Sorting

SORTING CLASS INSTANCES

```

# Comparison functions for a variety of particle values
>>> def by_mass(x,y):
...     return cmp(x.mass,y.mass) ^
>>> def by_velocity(x,y):
...     return cmp(x.velocity,y.velocity) ^
>>> def by_momentum(x,y):
...     return cmp(x.momentum(),y.momentum()) ^

# Sorting particles in a list by their various properties
>>> x = [particle(1.2,3.4),particle(2.1,2.3),particle(4.6,.7)]
>>> x.sort(by_mass) ^
>>> x
[(m:1.2, v:3.4), (m:2.1, v:2.3), (m:4.6, v:0.7)]
>>> x.sort(by_velocity) ^
>>> x
[(m:4.6, v:0.7), (m:2.1, v:2.3), (m:1.2, v:3.4)]
>>> x.sort(by_momentum) ^
>>> x
[(m:4.6, v:0.7), (m:1.2, v:3.4), (m:2.1, v:2.3)]

```

Criticism of Python

FUNCTION ARGUMENTS

All function arguments are called by reference. Changing data in
subroutine effects global data!

```
>>> def sum(lst):
...     tot=0
...     for i in range(0,len(lst)):
...         lst[i]+=1
...         tot += lst[i]
...     return tot
>>> a=range(1,4)
>>> sum(a)
9
```

```
>>> a
[2, 3, 4]
# Can be fixed by
>>> a=range(1,4)
>>> a_copy = a[:] # be careful: a_copy = a would not work
>>> sum(a_copy)
9
>>> a
[1, 2, 3]
```

Criticism of Python

FUNCTION ARGUMENTS

Python does not support something like "const" in C++. Good Python codes SHOULD NOT change any function arguments. No “in-place” modification of data. New data should be returned by return statement!

COPYING DATA

User has "no direct contact" with data structures. User might not be aware of data handling. Python is optimized for speed -> references.

```
>>> a=[1,2,3,[4,5]]
>>> b=a[:]
>>> a[0]=2
>>> b
[1,2,3,[4,5]]
>>> a[3][0]=0
>>> b
[1,2,3,[0,5]]
```

```
# Can be fixed by
>>> import copy
>>> a=[1,2,3,[4,5]]
>>> b = copy.deepcopy(a)
>>> a[3][0]=0
>>> b
[1,2,3,[4,5]]
```

Criticism of Python

CLASS DATA

In C++ class declaration uncovers all important information about the class - class members (data and methods). In Python, data comes into existence when used. User needs to read implementation of the class (much more code) to find class data and understand the logic of the class.

This is particularly important in large scale codes.

Python classes are most convenient for small to intermediate size projects.

RELOADING MODULES

If you import a module in command-line interpreter, but the module was later changed on disc, you can reload the module by typing

```
reload modulexxx
```

This reloads the particular modulexxx, but does not recursively reload modules that might also be changed on disc and are imported by the modulexxx.

NumPy

NumPy and SciPy

In 2005 Numarray and Numeric were merged into common project called "NumPy". On top of it, SciPy was build recently and spread very fast in scientific community.

Home: <http://www.scipy.org/SciPy>

IMPORT NUMPY AND SCIPY

```
>>> from numpy import *
>>> import numpy
>>> numpy.__version__
'1.0.1'

    or better
>>> from scipy import *
>>> import scipy
>>> scipty.__version__
'0.5.2'
```


Array Operations

SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
```

MATH FUNCTIONS

```
# Create array from 0 to 10
>>> x = arange(11.)

# multiply entire array by
# scalar value
>>> a = (2*pi)/10.
>>> a
0.628318530718
>>> a*x
array([ 0., 0.628, ..., 6.283])

# apply functions to array.
>>> y = sin(a*x)
```

NumPy defines the following constants:



```
pi = 3.14159265359
e = 2.71828182846
```

Introducing Numeric Arrays

SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
>>a=array([0,1,2],dtype=float) ↵
```

CHECKING THE TYPE

```
>>> type(a)
<type 'numpy.ndarray'>
```

NUMERIC TYPE OF ELEMENTS

```
>>> a.dtype
dtype('int32') ↵
```

BYTES IN AN ARRAY ELEMENT

```
>>>a.itemsize
4
```

ARRAY SHAPE

```
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

CONVERT TO PYTHON LIST

```
>>> a.tolist() ↵
[0, 1, 2, 3]
```

ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
[10, 1, 2, 3]
```

Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS


```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

(ROWS,COLUMNS)

```
>>> shape(a)
(2, 4)
```

GET/SET ELEMENTS

```
>>> a[1,3]
13
```



```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

ADDRESS FIRST ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, 13])
```

FLATTEN TO 1D ARRAY

```
>>>a.ravel()
array([0, 1, 2, 3, 10, 11, 12, 13])
```



A.FLAT AND RAVEL() REFERENCE ORIGINAL MEMORY

```
>>> a.ravel()[7]=-1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0,3:5]
array([3, 4])
```

```
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```

```
>>> a[:,2]
array([2, 12, 22, 32, 42, 52])
```

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Slices Are References

Slices are references to memory in original array. Changing values in a slice also changes the original array.

```

>>> a = array([0,1,2])

# create a slice containing only the
# last element of a
>>> b = a[2:3]
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 1,  2, 10]) ↑

```

Array Constructor

```
array(object, dtype=None, copy=1, order=None, subok=0, ndmin=0) ↵
```

- object** -any type of Python sequence. Nested list create multi-dimensional arrays.
- dtype** -character (string). Specifies the numerical type of the array. If it is None, the constructor makes its best guess at the numeric type.
dtype: integer, float, complex,...
- copy** -if **copy=0** and sequence is an array object, the returned array is a reference that data. Otherwise, a copy of the data in **sequence** is made.
- Order** -'C': C-contiguous order, 'FORTRAN': fortran contiguous, None: any order or maybe even non-contiguous

Array Constructor Examples

FLOATING POINT ARRAYS DEFAULT TO DOUBLE PRECISION

```
>>> a = array([0,1.,2,3])
>>> a.dtype
dtype('float64')
```

↑
notice decimal

USE TYPECODE TO REDUCE PRECISION

```
>>> a = array([0,1.,2,3], 'f')
>>> a.dtype
dtype('float32')
>>> len(ravel(a))*a.itemsize
16
```



BYTES FOR MAIN ARRAY STORAGE

```
# flat assures that
# multidimensional arrays
# work
>>> len(a.flat)*a.itemsize
32
```

ARRAYS REFERENCING SAME DATA

```
>>> a = array([1,2,3,4]) ↑
>>> b = array(a, copy=0) ↑
>>> b[1] = 10
>>> a
array([ 1, 10,  3,  4]) ↑
```

32-bit Typecodes

Character	Bits (Bytes)	Identifier
D	128 (16)	Complex , Complex64
F	64 (8)	Complex0, Complex8, Complex16, Complex32
d	64 (8)	Float , Float64
f	32 (4)	Float0, Float8, Float16, Float32
l	32 (4)	Int
i	32 (4)	Int32
s	16 (2)	Int16
1 (one)	8 (1)	Int8
u	32 (4)	UnsignedInt32
w	16 (2)	UnsignedInt16
b	8 (1)	UnsignedInt8
O	4 (1)	PyObject



Highlighted typecodes correspond to Python's standard Numeric types.

Array Creation Functions

arange ([start,] stop [, step=1] , dtype=None)

Nearly identical to Python's `range ()`. Creates an array of values in the range `[start,stop)` with the specified **step** value. Allows non-integer values for **start**, **stop**, and **step**. When not specified, **dtype** is derived from the **start**, **stop**, and **step** values.

```
>>> arange(0,2*pi,pi/4)
array([ 0.000,  0.785,  1.571,  2.356,  3.142,
        3.927,  4.712,  5.497])
```

ones (shape , dtype=None , order='C')

zeros (shape , dtype=float , order='C')

shape is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to **float** (in old versions **Int**)!!!

```
>>> ones((2,3),dtype=float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], 'float64')
```

Array Creation Functions (cont.)

`identity(n, dtype='l')`

Generates an n by n identity matrix with `dtype=Int`.

```
>>> identity(4)
```

```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

```
>>> identity(4, 'f')
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]], 'f')
```

Mathematic Binary Operators

`a + b` → `add(a,b)`
`a - b` → `subtract(a,b)`
`a % b` → `remainder(a,b)`

`a * b` → `multiply(a,b)` ↑
`a / b` → `divide(a,b)` ↑
`a ** b` → `power(a,b)` ↑

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.] )
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead
>>> add(a,b,a) # a += b
array([4, 6]) ↑
>>> a
array([4, 6]) ↑
```

Comparison and Logical Operators

```

equal          (==)
greater_equal (>=)
logical_and    (and)
logical_not    (not)

```

```

not_equal      (!=)
less           (<)
logical_or     (or)

```

```

greater        (>)
less_equal     (<=)
logical_xor

```

2D EXAMPLE

```

>>> a = array(((1,2,3,4), (2,3,4,5)))
>>> b = array(((1,2,5,4), (1,3,4,5)))
>>> a == b
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])
# functional equivalent
>>> equal(a,b)
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])

```

Bitwise Operators

<code>bitwise_and (&)</code>	<code>invert (~)</code>	<code>right_shift(a, shifts)</code>
<code>bitwise_or ()</code>	<code>bitwise_xor</code>	<code>left_shift (a, shifts)</code>

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_and(a,b)
array([ 17,  34,  68, 136])
```

```
# bit inversion
```

```
>>> a = array((1,2,3,4), dtype=uint8)
```

```
>>> invert(a)
array([254, 253, 252, 251], 'b')
```

```
# surprising type conversion
```

```
>>> left_shift(a,3)
array([ 8, 16, 24, 32], 'i')
```

Changed from
UnsignedInt8
to Int32

Trig and Other Functions

TRIGONOMETRIC

<code>sin (x)</code>	<code>sinh (x)</code>
<code>cos (x)</code>	<code>cosh (x)</code>
<code>arccos (x)</code>	
<code>arccosh (x)</code>	
<code>arctan (x)</code>	<code>arctanh (x)</code>
<code>arcsin (x)</code>	<code>arcsinh (x)</code>
<code>arctan2 (x, y)</code>	

OTHERS

<code>exp (x)</code>	<code>log (x)</code>
<code>log10 (x)</code>	<code>sqrt (x)</code>
<code>absolute (x)</code>	<code>conjugate (x)</code>
<code>negative (x)</code>	<code>ceil (x)</code>
<code>floor (x)</code>	<code>fabs (x)</code>
<code>hypot (x, y)</code>	<code>fmod (x, y)</code>
<code>maximum (x, y)</code>	<code>minimum (x, y)</code>

`hypot (x, y)`

Element by element distance
calculation using $\sqrt{x^2 + y^2}$

SciPy

CURRENT PACKAGES

- Special Functions (`scipy.special`)
- Signal Processing (`scipy.signal`)
- Fourier Transforms (`scipy.fftpack`)
- Optimization (`scipy.optimize`)
- General plotting (`scipy.[plt, xplt, gplt]`)
- Numerical Integration (`scipy.integrate`)
- Linear Algebra (`scipy.linalg`)
- Input/Output (`scipy.io`)
- Genetic Algorithms (`scipy.ga`)
- Statistics (`scipy.stats`)
- Distributed Computing (`scipy.cow`)
- Fast Execution (`weave`)
- Clustering Algorithms (`scipy.cluster`)
- Sparse Matrices* (`scipy.sparse`)

Basic Environment

CONVENIENCE FUNCTIONS

```
>>> info(linspace)
```

```
linspace(start, stop, num=50, endpoint=1, retstep=0)
```

Evenly spaced samples.

Return num evenly spaced samples from start to stop. If endpoint=1 then last sample is stop. If retstep is 1 then return the step value used.

```
>>> linspace(-1,1,5)
```

```
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

```
>>> logspace(0,3,4)
```

```
array([  1.,  10., 100., 1000.])
```

```
>>> info(logspace)
```

```
logspace(start, stop, num=50, endpoint=1)
```

Evenly spaced samples on a logarithmic scale.

Return num evenly spaced samples from 10**start to 10**stop. If endpoint=1 then last sample is 10**stop.

info help system for scipy
similar to dir for the rest of python

linspace get equally spaced points.
r_[] also does this (shorthand)

logspace get equally spaced points in log10 domain

Basic Environment

CONVENIENT MATRIX GENERATION AND MANIPULATION

```
>>> A = mat('1,2,4;4,5,6;7,8,9')  
>>> A=mat([[1,2,4],[4,5,6],[7,8,9]])
```

Simple creation of matrix with “;” meaning row separation

```
>>> print A  
Matrix([[1, 2, 4],  
        [2, 5, 3],  
        [7, 8, 9]])
```

Matrix Power

```
>>> print A**4  
Matrix([[ 6497,  9580,  9836],  
        [ 7138, 10561, 10818],  
        [18434, 27220, 27945]])
```

Matrix Multiplication and Matrix Inverse

```
>>> print A*A.I  
Matrix([[ 1.,  0.,  0.],  
        [ 0.,  1.,  0.],  
        [ 0.,  0.,  1.]])
```

```
>>> print A.T  
Matrix([[1, 2, 7],  
        [2, 5, 8],  
        [4, 3, 9]])
```

Matrix Transpose

More Basic Functions

TYPE HANDLING

<code>iscomplexobj</code>	<code>real_if_close</code>	<code>isnan</code>
<code>iscomplex</code>	<code>isscalar</code>	<code>nan_to_num</code>
<code>isrealobj</code>	<code>isneginf</code>	<code>common_type</code>
<code>isreal</code>	<code>isposinf</code>	<code>cast</code>
<code>imag</code>	<code>isinf</code>	<code>typename</code>
<code>real</code>	<code>isfinite</code>	

SHAPE MANIPULATION

<code>squeeze</code>	<code>vstack</code>	<code>split</code>
<code>atleast_1d</code>	<code>hstack</code>	<code>hsplit</code>
<code>atleast_2d</code>	<code>column_stack</code>	<code>vsplit</code>
<code>atleast_3d</code>	<code>dstack</code>	<code>dsplit</code>
<code>apply_over_axes</code>	<code>expand_dims</code>	<code>apply_along_axis</code>

OTHER USEFUL FUNCTIONS

<code>select</code>	<code>unwrap</code>	<code>roots</code>
<code>extract</code>	<code>sort_complex</code>	<code>poly</code>
<code>insert</code>	<code>trim_zeros</code>	<code>any</code>
<code>fix</code>	<code>fliplr</code>	<code>all</code>
<code>mod</code>	<code>flipud</code>	<code>disp</code>
<code>amax</code>	<code>rot90</code>	<code>unique</code>
<code>amin</code>	<code>eye</code>	<code>extract</code>
<code>ptp</code>	<code>diag</code>	<code>insert</code>
<code>sum</code>	<code>factorial</code>	<code>nansum</code>
<code>cumsum</code>	<code>factorial2</code>	<code>nanmax</code>
<code>prod</code>	<code>comb</code>	<code>nanargmax</code>
<code>cumprod</code>	<code>pade</code>	<code>nanargmin</code>
<code>diff</code>	<code>derivative</code>	<code>nanmin</code>
<code>angle</code>	<code>limits.XXXX</code>	

Input and Output

scipy.io replaced by loadtxt and savetxt --- Reading and writing ASCII files

textfile.txt

Student	Test1	Test2	Test3	Test4
Jane	98.3	94.2	95.3	91.3
Jon	47.2	49.1	54.2	34.7
Jim	84.2	85.3	94.1	76.4

Read columns 1 & 3

Read from line 3 to the end

```
loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None,
        converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)
```

```
>>> a = loadtxt('textfile.txt', skiprows=2, usecols=(1,3))
```

```
>>> print a
```

```
[[ 98.3  94.2  95.3  91.3]
 [ 47.2  49.1  54.2  34.7]
 [ 84.2  85.3  94.1  76.4]]
```

Reads columns 1, and 3

Read from line 3 to the end of the file

Input and Output

scipy.io --- Reading and writing raw binary files

```
fid = open(file_name[, mode='rb'[, buffering]])
```

Class for reading and writing binary files into Numeric arrays.

Methods

- **file_name** The complete path name to the file to open.
- **mode** Open the file with given permissions: ('r', 'w', 'a') for reading, writing, or appending. This is the same as the mode argument in the builtin open command.
- **format** b stands for binary file
+would allow simultaneous reading & writing

- read** read data from file and return Numeric array
- write** write to file from Numeric array
- fort_read** read Fortran-formatted binary data from the file.
- fort_write** write Fortran-formatted binary data to the file.
- rewind** rewind to beginning of file
- size** get size of file
- seek** seek to some position in the file
- tell** return current position in file
- close** close the file

Few examples

Examples of SciPy use

Integration

Suppose we want to integrate Bessel function

$$\int_0^x dt J_1(t) / t$$

```
>>>from scipy import integrate, special
>>>info(integrate)
.....<documentation of integrate module>.....
>>> integrate.quad(lambda t: special.j1(t)/t,
0,pi)
(1.062910971494,1.18e-14) ↑
```

j1int.py module:

```
from scipy import *
from scipy import integrate,special
def fun(x):
    return integrate.quad(lambda t: special.j1(t)/t,0,x)

x=linspace(0,30.,300)
for tx in x:
    print tx, fun(tx)[0]
```

Minimization

Suppose we want to minimize the function

$$(x - a)^2 + (y - b)^2 = \min$$

```

>>> from scipy import *
>>> import scipy
>>> info(scipy)
.... <documentation of all available modules>
>>> from scipy import optimize
>>> info(optimize)
>>> info(optimize.fmin_powell)

>>> def func((x,y),(a,b)): return (x-a)**2+(y-b)**2

```

Starting guess

```

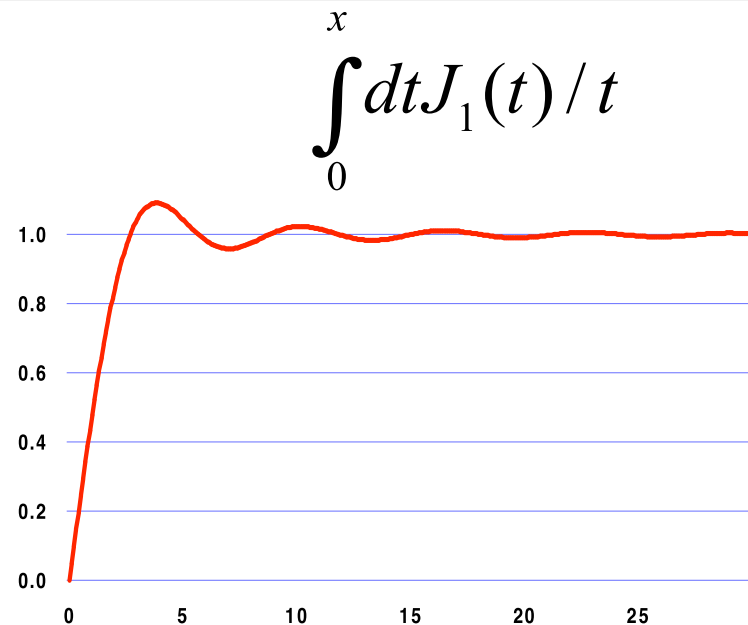
>>> optimize.fmin_powell(func, (0,0), args=((5,6),))
Optimization terminated successfully,
Current function value: 0.00000
Iterations: 2
Function evaluations: 38
array([5., 6.])

```

additional arguments

Root finding and integration

The function $\int_0^x dt J_1(t) / t == 1$



has many solutions. Suppose we want to find all solution in the range [0:100]

Put it all together

```

from scipy import *
"""
    Finds all solutions of the equation Integrate[j1(t)/t,{t,0,x}] == 1
    in the range x=[0,100]
"""

def func(x,a):
    " Computes Integrate[j1(t)/t,{t,0,x}] - a"
    return integrate.quad(lambda t: special.j1(t)/t, 0, x)[0] - a

# Finds approximate solutions of the equation in the range [0:100]
x = r_[0:100:0.2]          # creates an equally spaced array
b = map(lambda t: func(t,1), x) # evaluates function on this array

z = [];                  # approximate solutions of the equation
for i in range(1,len(b)): # if the function changes sign,
    if (b[i-1]*b[i]<0): z.append(x[i]) # the solution is bracketed

print "Zeros of the equation in the interval [0:100] are"
j=0
for zt in z:
    print j, optimize.fsolve(func,zt,(1,)) # calling root finding
    routine, finds all zeros.
    j+=1

```

It takes around 2 seconds to get

Zeros of the equation in the interval [0:100] are

- 0 2.65748482457
- 1 5.67254740317
- 2 8.75990144967
- 3 11.872242395
- 4 14.9957675329
- 5 18.1251662422
- 6 21.2580027553
- 7 24.3930147628
- 8 27.5294866728
- 9 30.666984016
- 10 33.8052283484
- 11 36.9440332549
- 12 40.0832693606
- 13 43.2228441315
- 14 46.362689668
- 15 49.5027550388
- 16 52.6430013038
- 17 55.7833981883
- 18 58.9239218038
- 19 62.0645530515
- 20 65.2052764808
- 21 68.3460794592
- 22 71.4869515584
- 23 74.6278840946
- 24 77.7688697786
- 25 80.9099024466
- 26 84.0509768519
- 27 87.1920884999
- 28 90.3332335188
- 29 93.4744085549
- 30 96.615610689
- 31 99.7568373684

Linear Algebra

scipy.linalg --- FAST LINEAR ALGEBRA

- Uses vendor optimized BLAS, or ATLAS.
- Low-level access to BLAS and LAPACK routines in modules `linalg.fblas`, and `linalg.flapack` (FORTRAN order)
- High level matrix routines
 - Linear Algebra Basics: `inv`, `solve`, `det`, `norm`, `lstsq`, `pinv`
 - Decompositions: `eig`, `lu`, `svd`, `orth`, `cholesky`, `qr`, `schur`
 - Matrix Functions: `expm`, `logm`, `sqrtn`, `cosm`, `coshm`, `funm` (general matrix functions)

Some simple examples

```

>>> A=matrix(random.rand(5,5)) # creates random matrix
>>> A.I
<inverse of the random matrix>
>>> linalg.det(A)
<determinant of the matrix>
>>> linalg.eigvals(A)
<eigenvalues only>
>>> linalg.eig(A)
<eigenvalues and eigenvectors>
>>> linalg.svd(A)
<SVD decomposition>
>>> linalg.cholesky(A)
<Cholesky decomposition for positive definite A>
>>> B=matrix(random.rand(5,5))
>>> linalg.solve(A,B)
<Solution of the equation A.X=B>

```

Special Functions

scipy.special

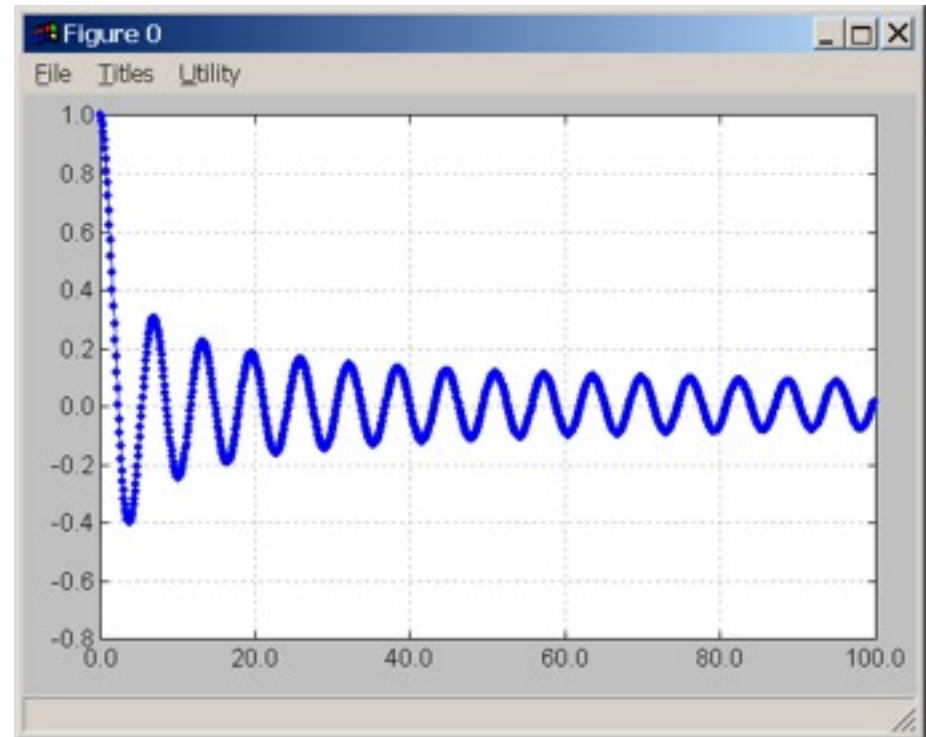
Includes over 200 functions:

Airy, Elliptic, Bessel, Gamma, HyperGeometric, Struve, Error, Orthogonal Polynomials, Parabolic Cylinder, Mathieu, Spheroidal Wave, Kelvin

FIRST ORDER BESSEL EXAMPLE

```
#environment setup
>>> import gui_thread
>>> gui_thread.start()
>>> from scipy import *
>>> import scipy.pyplot as plt

>>> x = r_[0:100:0.1]
>>> j0x = special.j0(x) ↑
>>> plt.plot(x, j0x) ↑
```



Interpolation

scipy.interpolate --- General purpose Interpolation

- **1-d linear Interpolating Class**

- Constructs callable function from data points
- Function takes vector of inputs and returns linear interpolants

- **1-d and 2-d spline interpolation (FITPACK)**

- Splines up to order 5
- Parametric splines

Integration

scipy.integrate --- General purpose Integration

- **Ordinary Differential Equations (ODE)**

`integrate.odeint`, `integrate.ode`

- **Samples of a 1-d function**

`integrate.trapz` (trapezoidal Method), `integrate.simps` (Simpson Method), `integrate.romb` (Romberg Method)

- **Arbitrary callable function**

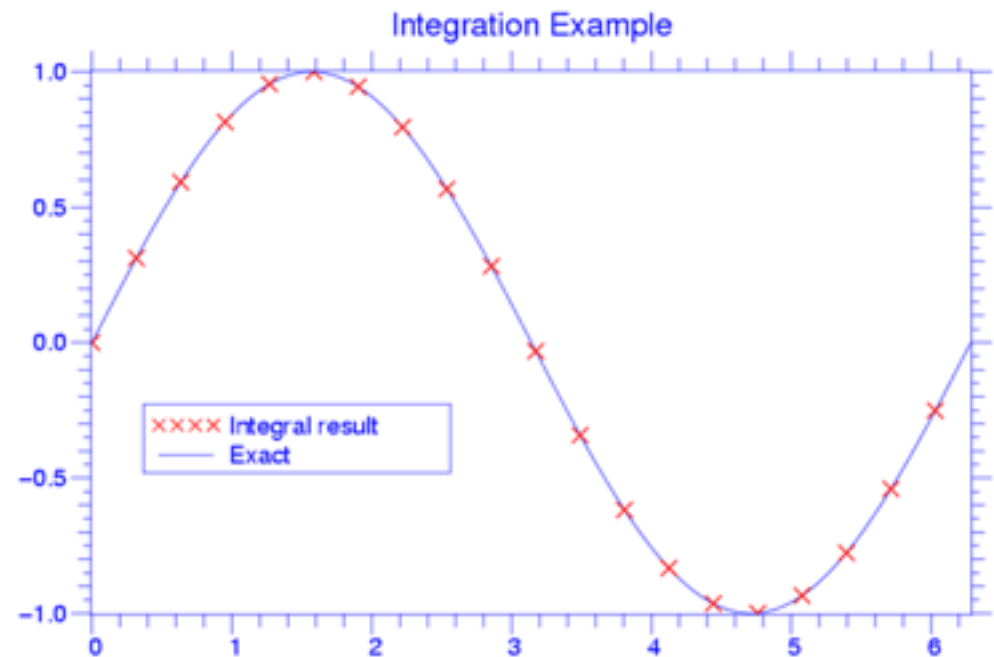
`integrate.quad` (general purpose), `integrate.dblquad` (double integration), `integrate.tplquad` (triple integration), `integrate.fixed_quad` (fixed order Gaussian integration), `integrate.quadrature` (Gaussian quadrature to tolerance), `integrate.romberg` (Romberg)

Integration

scipy.integrate --- Example

```
>>> def func(x):
    return integrate.quad(cos,0,x)[0]
>>> vecfunc = vectorize(func)

>>> x = r_[0:2*pi:100j]
>>> x2 = x[::5]
>>> y = sin(x)
>>> y2 = vecfunc(x2)
>>> xplt.plot(x,y,x2,y2,'rx')
```



scipy.optimize --- unconstrained minimization and root finding

- **Unconstrained Optimization**

`fmin` (Nelder-Mead simplex), `fmin_powell` (Powell's method), `fmin_bfgs` (BFGS quasi-Newton method), `fmin_ncg` (Newton conjugate gradient), `leastsq` (Levenberg-Marquardt), `anneal` (simulated annealing global minimizer), `brute` (brute force global minimizer), `brent` (excellent 1-D minimizer), `golden`, `bracket`

- **Constrained Optimization**

`fmin_l_bfgs_b`, `fmin_tnc` (truncated newton code), `fmin_cobyla` (constrained optimization by linear approximation), `fminbound` (interval constrained 1-d minimizer)

- **Root finding**

`fsolve` (using MINPACK), `brentq`, `brenth`, `ridder`, `newton`, `bisect`, `fixed_point` (fixed point equation solver)

Solving Hydrogen Atom

Schroedinger equation

$$\left(-\frac{\hbar^2}{2m} \nabla^2 - \frac{Ze^2}{4\pi\epsilon_0 r} \right) \psi(\mathbf{r}) = E\psi(\mathbf{r})$$

Ansatz for the solution:

$$\psi(\mathbf{r}) = Y_{lm}(\hat{\mathbf{r}}) \frac{u(r)}{r}$$

Dimensionless variables:

$$x = \frac{r}{r_B} \quad \varepsilon = \frac{E}{E_0}$$

$$r_B = \frac{4\pi\epsilon_0 \hbar^2}{me^2}$$

$$E_0 = \frac{\hbar^2}{2mr_B^2}$$

Ordinary differential Equation:

$$u''(x) - \left(\frac{l(l+1)}{x^2} - \frac{2Z}{x} - \varepsilon \right) u(x) = 0$$

Hydrogen Atom

First order system of equations
 Can be solved by “integrate.odeint”

$$y = (u(x), u'(x))$$

$$\frac{dy}{dx} = (u'(x), u''(x))$$

Boundary conditions

$$u(0) = 0 \rightarrow \psi(0) < \infty$$

$$u(\infty) = 0 \rightarrow \int |\psi(r)|^2 r^2 dr \propto \int u^2(r) dr < \infty$$

Boundary conditions given at the two ends → need shooting method

Shooting

- Suppose the two boundary conditions are given at a and b . Choose $u(a)=0$ and $u'(a)=c$ with c some small constant.
- Solve for $u(x)$ to the other end and check if $u(b)=0$.
- Using root finding routine find energy E for which $u(b)=0$. This is a bound state.
- Continue with increasing E until sufficient number of bound states is found.

Solving radial Hydrogen atom

- It turns out that forward integration of radial Schroedinger equation is unstable. It is better to start integrating at infinity, and then continue down to zero.
- It is better to use logarithmic mesh for radial variable rather than linear. Radial functions need smaller number of points in logarithmic mesh.