Introduction to Python

Lecture 3

Arul Lakshminarayan, 6/10/17

Loop constructs

- **For** and **While**.
- **For** iterates in iterables such as containers
- **While** checks Boolean values/satisfiability

```
for <iterator> in <iterable>:
    <block>
```

```
c = 4
for c in "Python":
    print c

c
```

```
for (x,y) in Z:
    <block>
```

```
>>> for (x,y) in (1,2),(3,4),(5,6):
        print(x**2,y**2)
```

range function: Warning: slightly different from ver. 2.x (xrange in ver. 2.x)

range(start,end,step) "creates a list" start,start+step,start+2*step, …
till it is less than end.
Not accessible as a list object, till
list(range(start,end,step))
default start=0, default step=1

```
>>> x=range(2,10,3)
>>> x[0]
2
>>> x[1]
5
>>> x[2]
8
>>> x
range(2, 10, 3)
>>> type(x)
<class 'range'>
```

```
>>> y=list(range(2,10,3))
>>> y
[2, 5, 8]
>>> y=list(range(2,10,2))
>>> y
[2, 4, 6, 8]
>>> type(y)
<class 'list'>
>>>
```

```
TRY
>>> l=[]
>>> for y in range(10):
        l.append(y)
```

Example: 1-d maps

Logistic map: x -> f(x)=r x(1-x)

```
def logistic_map(r,x,n):
        for i in range(n+1):
                print(x,end=', ')
                x=r*x*(1-x)
```

Exercise: code a function for the ``doubling map” x -> f(x)=2 x modulo 1

see iterates for 100 times and notice that for arbitrary initial conditions (in (0,1)) they go to 0. When do they do that and why?

```
for <iterator> in <iterable>:
    <block1>
    if <test1>:
        continue
    <block2>
<block5>
```

```
for <iterator> in <iterable>:
    <block1>
    if <test2>:
        break
    <block2>
else:
    <block4>
<block5>
```

If test2 is False, block2 is iterated till last iteration step.
then control passes to else: and block4 is performed before block5.
If test2 is True, iterator is escaped and block5 is evaluated.

Example: Primality

```
def primeq(x):
    for i in range(2,int(x**.5)+1):
        if x%i==0:
            print('False')
            break
    else:
            print('True')
```

Note the indent in the “else” statement.
Test what happens if it is aligned with “if”.

Example: Primality

```
def primeq(x):
    for i in range(2,int(x**.5)+1):
        if x%i==0:
            print('False')
            break
    else:
            print('True')
```

```
>>> primeq(10)
False
>>> primeq(25)
False
>>> primeq(27)
False
>>> primeq(29)
True
>>>
```

Note the indent in the “else” statement.
Test what happens if it is aligned with “if”.

```
>>> primeq(10)
False
>>> primeq(25)
True
True
True
False
>>>
```

List comprehensions

```
>>> L1=list(range(10))
>>> L1
>>> L2=[x*x for x in L1]
>>> L2
>> L3=[x*x for x in L1 if x%2==0]
>>> L3
```

```
>>> lpoints=[(x,x/2) for x in L1]
>>> lpoints
>>> ldist=[(x*x+y*x)**.5 for (x,y) in lpoints]
>>> ldist
>>>
```

TRY:

lpoints1=[(x,y) for x in L1 for y in L2]

While

```
while <test>:
    <block1>
<block2>
```

```
while True :
    print "Type Control-C to stop this!"
```

As for “for”, “while” can be interrupted by continue, break, if …

Understand the output of

```
for i in range(10):
    while i in range(5):
        print(i,i**2)
        i+=1
    else:
        print(i,i)
```

Sieve of Eratosthenes: List of prime numbers

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2	3		5		7		9		11		13		15		17	
2	**3**		5		7				11		13				17	
2	3		**5**		7				11		13				17	

```
# Sieve of Eratosthenes: primes
import time

def sieve(n):
    start_time=time.time()
    """
    Use Sieve of Eratosthenes to compute list of primes <=n.
    Version 1 Different from Version in Stewart. Version uses timer"""
    Lprimes=list(range(2,n+1))
    for i in Lprimes:
        if  i*i<= n:
            for k in range(i*i,n+1,i):
                if k%i==0 and k in Lprimes:
                    Lprimes.remove(k)
    end_time=time.time()
    return len(Lprimes),end_time-start_time
```

Numpy

Numerical Python: Tools specific to computation

Main actor: ARRAYS: arrays or ndarrays

Like lists, except they are homogeneous and cannot be added to

```
>>> import numpy as np
>>>
>>> a=np.array([1,2,5,7])
>>> type(a)
<class 'numpy.ndarray'>
>>> a[0],a[1],a[2]
(1, 2, 5)
>>> a.shape
(4,)
>>> a[3]=9
>>> a
array([1, 2, 5, 9])
>>>
```

```
>>> a2d=np.array([[1,2],[3,4]])
>>> a2d
array([[1, 2],
       [3, 4]])
>>> print(a2d)
[[1 2]
 [3 4]]
>>> a2d.shape
(2, 2)
>>>
```

https://www.python-course.eu/index.php

```
>>> ## Ways of creating arrays
>>> a=np.zeros((2,3))
>>> print(a)
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
>>> a=np.empty((2,2))
>>> print(a)
[[  4.94065646e-324   9.88131292e-324]
 [  2.47032823e-323   4.44659081e-323]]
>>> a=np.ones((2,2))
>>> print(a)
[[ 1.  1.]
 [ 1.  1.]]
>>> a=np.eye(3)
>>> print(a)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

```
>>> help(np.random)
Help on package numpy.random in numpy:

NAME
    numpy.random

DESCRIPTION
    ========================
    Random Number Generation
    ========================

    ====================
=========================================================
    Utility functions

============================================================================
==
    random_sample        Uniformly distributed floats over ``[0, 1)``.
    random               Alias for `random_sample`.
    bytes                Uniformly distributed random bytes.
    random_integers      Uniformly distributed integers in a given range.
    permutation          Randomly permute a sequence / generate a random sequence.
    shuffle              Randomly permute a sequence in place.
    seed                 Seed the random number generator.
    choice               Random sample from 1-D array.

    ====================
```

Compatibility functions	
rand	Uniformly distributed values.
randn	Normally distributed values.
ranf	Uniformly distributed floating point numbers.
randint	Uniformly distributed integers in a given range.

Univariate distributions	
beta	Beta distribution over ``[0, 1]``.
binomial	Binomial distribution.
chisquare	:math:`\chi^2` distribution.
exponential	Exponential distribution.
f	F (Fisher-Snedecor) distribution.
gamma	Gamma distribution.
geometric	Geometric distribution.
gumbel	Gumbel distribution.
hypergeometric	Hypergeometric distribution.
laplace	Laplace distribution.
logistic	Logistic distribution.
lognormal	Log-normal distribution.
logseries	Logarithmic series distribution.
negative_binomial	Negative binomial distribution.
noncentral_chisquare	Non-central chi-square distribution.
noncentral_f	Non-central F distribution.
normal	Normal / Gaussian distribution.
pareto	Pareto distribution.
poisson	Poisson distribution.
power	Power distribution.
rayleigh	Rayleigh distribution.
triangular	Triangular distribution.
uniform	Uniform distribution.
vonmises	Von Mises circular distribution.
wald	Wald (inverse Gaussian) distribution.
weibull	Weibull distribution.
zipf	Zipf's distribution over ranked data.

```
>>> np.random.random()
0.702300623632869
>>> np.random.random(10)
array([ 0.29665841,  0.62870005,  0.1881841 ,  0.96082173,  0.92515552,
        0.59382921,  0.97462479,  0.15204944,  0.51245089,  0.47862482])
>>> np.random.randn(10)
array([-0.09606186,  0.46646645,  1.14411964,  0.78557085,  1.44135266,
       -0.07429623, -0.43208525,  1.44862351,  0.62171587,  0.94636639])
```

```
>>>##Generating arrays of random numbers
>>>
>>> a=np.random.standard_normal((2,2))
>>> a
array([[ 0.02519554,  0.74814784],
       [ 0.82917378,  0.76525869]])
>>> a=np.random.random((2,2))
>>> a
array([[ 0.55789999,  0.96194553],
       [ 0.38743052,  0.98357223]])
>>> a=np.random.standard_normal((2,2))
>>> a
array([[ 1.10176981, -0.03261667],
       [-0.54790951,  0.6602611 ]])
>>> a=np.random.randn(2,2)
>>> a
array([[ 0.86772907,  1.50899631],
       [-0.93854074,  1.13951124]])
>>>
```

Caution: “randn” does not take tuples as argument. But “standard_normal” does!

Slicing, mutability

```
>>> a=np.random.randn(3,3)
>>> a
array([[-1.03207779, -0.2740379 , -1.40255791],
       [-0.20209728, -0.4141725 , -0.64277807],
       [-0.11530382, -0.72801668,  0.42105809]])
>>> b=a[:2,1:3]
>>> b
array([[-0.2740379 , -1.40255791],
       [-0.4141725 , -0.64277807]])
>>> b[0,0]
-0.27403789804242484
>>> b[0,0]=1
>>> b
array([[ 1.        , -1.40255791],
       [-0.4141725 , -0.64277807]])
>>> a
array([[-1.03207779,  1.        , -1.40255791],
       [-0.20209728, -0.4141725 , -0.64277807],
       [-0.11530382, -0.72801668,  0.42105809]])
>>>
```

Use a.copy() to copy a

```
>>> a
array([[ 0.85899144,  0.56035937],
       [ 0.8344387 ,  0.22246127]])
>>> rw1=a[0,:]
>>> rw1
array([ 0.85899144,  0.56035937])
>>> c1=a[:,0]
>>> c1
array([ 0.85899144,  0.8344387 ])
>>>
```

np.arange

`numpy.arange`([*start*,]*stop*, [*step*,]*dtype=None*)¶

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1,10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(10.0)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> np.arange(10.3)
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
>>> np.arange(1,10.0,.5)
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,
        6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
>>> np.arange(1.2,5.3,.2)
array([ 1.2,  1.4,  1.6,  1.8,  2. ,  2.2,  2.4,  2.6,  2.8,  3. ,  3.2,
        3.4,  3.6,  3.8,  4. ,  4.2,  4.4,  4.6,  4.8,  5. ,  5.2])
```

numpy.linspace

numpy.linspace

numpy.**linspace**(*start, stop, num=50, endpoint=True, retstep=False, dtype=None*) [source]

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

```
>>> np.linspace(0,10,10)
array([  0.        ,   1.11111111,   2.22222222,   3.33333333,
         4.44444444,   5.55555556,   6.66666667,   7.77777778,
         8.88888889,  10.        ])
>>> np.linspace(0,10,10,0)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> np.linspace(1.3,10.2,10)
array([  1.3       ,   2.28888889,   3.27777778,   4.26666667,
         5.25555556,   6.24444444,   7.23333333,   8.22222222,
         9.21111111,  10.2       ])
>>> (10.2-1.3)/9
0.9888888888888887
>>> 1.3+_
2.2888888888888888
>>>
```

Basic array math

```
>>> a=np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b=a+.5
>>> b
array([[ 0.5,  1.5,  2.5],
       [ 3.5,  4.5,  5.5],
       [ 6.5,  7.5,  8.5]])
>>> b*a
array([[  0. ,   1.5,   5. ],
       [ 10.5,  18. ,  27.5],
       [ 39. ,  52.5,  68. ]])
>>> b+a
array([[  0.5,   2.5,   4.5],
       [  6.5,   8.5,  10.5],
       [ 12.5,  14.5,  16.5]])
>>> a/b
array([[ 0.        ,  0.66666667,  0.8       ],
       [ 0.85714286,  0.88888889,  0.90909091],
       [ 0.92307692,  0.93333333,  0.94117647]])
>>>
```

array mult. not matrix mult.

```
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> a*b
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.dot(a,b)
array([[  3.,   3.,   3.],
       [ 12.,  12.,  12.],
       [ 21.,  21.,  21.]])
>>>
```

```
>> np.multiply(a,b)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.add(a,b)
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
>>>
```

```
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> v
array([ 1. ,  0.5,  0. ])
>>> np.dot(a,v)
array([ 0.5,  5. ,  9.5])
>>> np.dot(v,a)
array([ 1.5,  3. ,  4.5])
>>>
```

Transpose, conjugate, Adjoint

```
>>> a=np.arange(9).reshape(3,3)+np.linspace(2,10,9).reshape(3,3)*1j
>>>
>>> a
array([[ 0. +2.j,  1. +3.j,  2. +4.j],
       [ 3. +5.j,  4. +6.j,  5. +7.j],
       [ 6. +8.j,  7. +9.j,  8.+10.j]])
>>> a.T
array([[ 0. +2.j,  3. +5.j,  6. +8.j],
       [ 1. +3.j,  4. +6.j,  7. +9.j],
       [ 2. +4.j,  5. +7.j,  8.+10.j]])
```

```
>> a.conj()
array([[ 0. -2.j,  1. -3.j,  2. -4.j],
       [ 3. -5.j,  4. -6.j,  5. -7.j],
       [ 6. -8.j,  7. -9.j,  8.-10.j]])

>>> a.conj().T
array([[ 0. -2.j,  3. -5.j,  6. -8.j],
       [ 1. -3.j,  4. -6.j,  7. -9.j],
       [ 2. -4.j,  5. -7.j,  8.-10.j]])
```

Linear algebra

numpy.linalg

help(numpy.linalg)

NAME
    numpy.linalg

DESCRIPTION
    Core Linear Algebra Tools

    Linear algebra basics:

    - norm   Vector or matrix norm
    - inv   Inverse of a square matrix
    - solve   Solve a linear system of equations
    - det   Determinant of a square matrix
    - lstsq   Solve linear least-squares problem
    - pinv   Pseudo-inverse (Moore-Penrose) calculated using a singular
                      value decomposition
    - matrix_power   Integer power of a square matrix

    Eigenvalues and decompositions:

    - eig   Eigenvalues and vectors of a square matrix
    - eigh   Eigenvalues and eigenvectors of a Hermitian matrix
    - eigvals   Eigenvalues of a square matrix
    - eigvalsh   Eigenvalues of a Hermitian matrix
    - qr   QR decomposition of a matrix
    - svd   Singular value decomposition of a matrix
    - cholesky   Cholesky decomposition of a matrix

```
>>> a=np.array([[1,2],[2,1]])
>>> a
array([[1, 2],
       [2, 1]])
>>> np.linalg.eig(a)
(array([ 3., -1.]), array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]]))
>>> la=np.linalg
>>> la.eigh(a)
(array([-1.,  3.]), array([[-0.70710678,  0.70710678],
       [ 0.70710678,  0.70710678]]))
>>> la.eigvals(a)
array([ 3., -1.])
>>> la.eigvalsh(a)
array([-1.,  3.])
>>>
```

Basic plotting

```
>>> import matplotlib.pylab as plt
>>> import numpy as np
>>> x=np.linspace(0,2*np.pi,50)
>>> fig1=plt.plot(x,np.sin(x))
>>> fig2=plt.plot(x,np.cos(x))
>>> plt.show()
>>>
```

Example: A single spin in a magnetic field

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$H = \sigma_z + h\,\sigma_x \quad = \begin{pmatrix} 1 & h \\ h & -1 \end{pmatrix}$$

The EIGENVALUEs of H are the energy levels

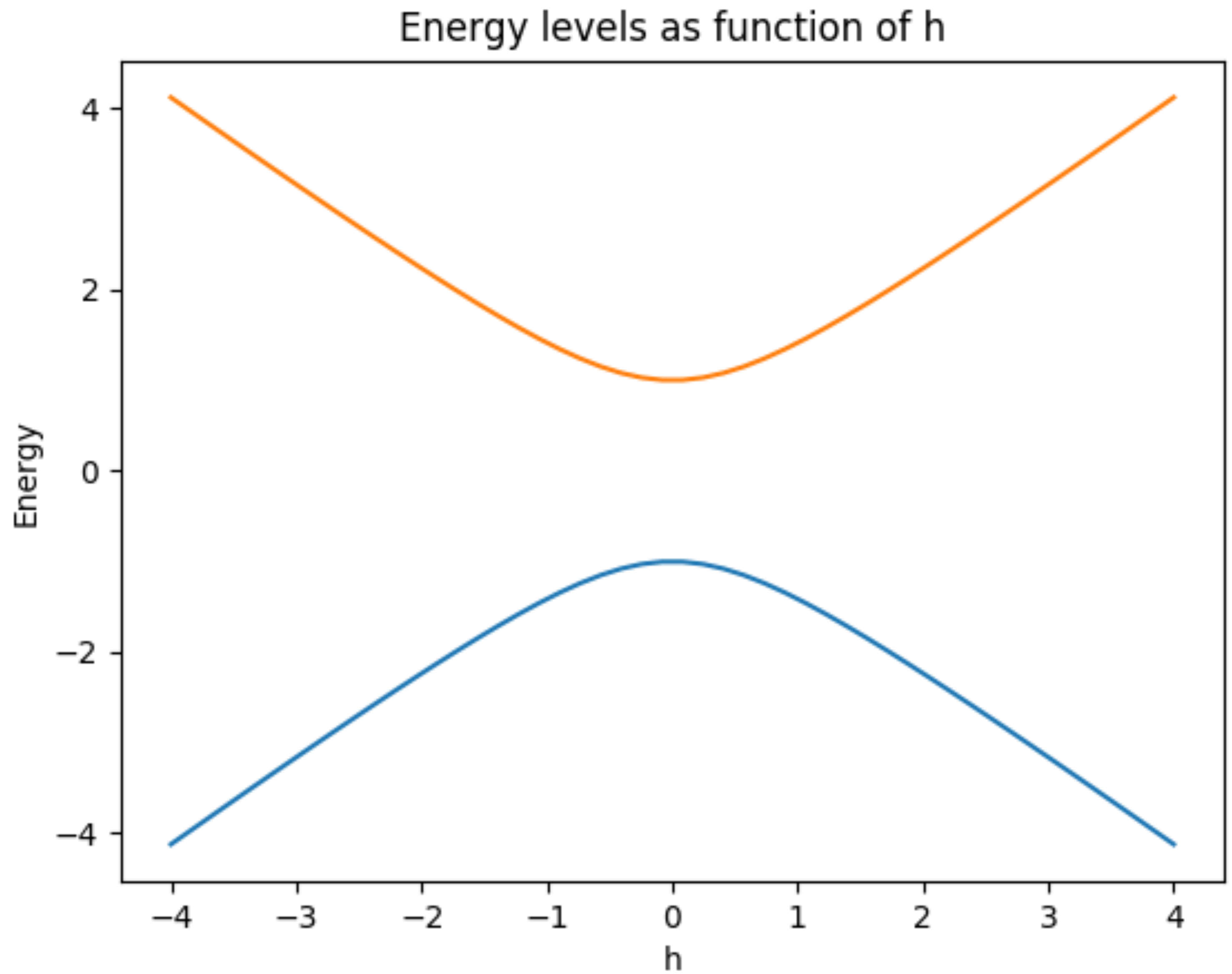
```python
import numpy as np
import pylab as plt

sigmax=np.array([[0,1],[1,0]])
sigmaz=np.array([[1,0],[0,-1]])
evallist=[]
x=[]
y=[]
z=[]

for h in np.linspace(-4,4,50):
    hamil=sigmaz+h*sigmax
    eval1=np.linalg.eigvalsh(hamil)
    x.append(h)
    y.append(eval1[0])
    z.append(eval1[1])
    evallist.append([h,eval1[0],eval1[1]])
np.savetxt('evalszsx.dat',evallist,fmt="%2.5f")
plt.plot(x,y)
plt.plot(x,z)
plt.xlabel('h')
plt.ylabel('Energy')
plt.title('Energy levels as function of h')
plt.show()
```

Python — more evalszsx.dat — 80×24

```
-4.00000 -4.12311 4.12311
-3.83673 -3.96491 3.96491
-3.67347 -3.80715 3.80715
-3.51020 -3.64987 3.64987
-3.34694 -3.49314 3.49314
-3.18367 -3.33703 3.33703
-3.02041 -3.18165 3.18165
-2.85714 -3.02709 3.02709
-2.69388 -2.87350 2.87350
-2.53061 -2.72103 2.72103
-2.36735 -2.56989 2.56989
-2.20408 -2.42033 2.42033
-2.04082 -2.27265 2.27265
-1.87755 -2.12725 2.12725
-1.71429 -1.98463 1.98463
-1.55102 -1.84544 1.84544
-1.38776 -1.71052 1.71052
-1.22449 -1.58094 1.58094
-1.06122 -1.45815 1.45815
-0.89796 -1.34400 1.34400
-0.73469 -1.24088 1.24088
-0.57143 -1.15175 1.15175
-0.40816 -1.08009 1.08009
evalszsx.dat
```


Energy levels as function of h
Energy
h
4
2
0
−2
−4
−4
−3
−2
−1
0
1
2
3
4