PYTHON

Elementary

Assign a variable and print the square of it:

```
a=3+4
print(a**2)
```

Run a loop over an integer j ranging from 0 to 3. Note the indentation (by a tab or four spaces) to mark the body of the loop:

```
for j in range(4):
    k=j*3
    print("Value", k)
print("Loop finished")
```

Include all the routines from the numpy linear algebra package:

```
from numpy import *
```

Create a vector containing 400 numbers evenly spaced between 0 and 10*pi:

```
x=linspace(0,10*pi,400)
```

Print the help for "linspace":

```
help(linspace)
```

Print the third element of that array (index takes values 0,1,2,...):

```
print(x[2])
```

Apply some function to x (elementwise, to each element of x separately). Do not use loops!

```
y=sin(x)
```

Plotting

Include all the routines for plotting. The second line tells the jupyter notebook to display the plots directly inside the notebook:

```
from matplotlib import pyplot as plt
%matplotlib inline
```

Plot y vs. x:

```
plt.plot(x,y,linewidth=5,color="red")
plt.show()
```

Make a 50x50 grid of x and y values (coordinates of points in a rectangle, used for an image):

```
xmin=0; xmax=4; ymin=-2; ymax=2; N=50
x,y=meshgrid(linspace(xmin,xmax,N),li
nspace(ymin,ymax,N))
```

Make a 2D plot (values shown as colors):

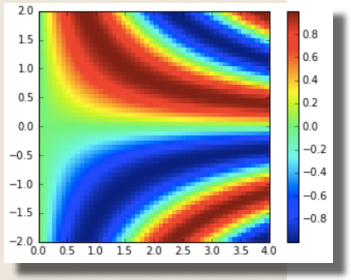
```
plt.imshow(sin(x*y),origin="lower",in
terpolation='none',
extent=[xmin,xmax,ymin,ymax])
plt.colorbar()
plt.show()
```

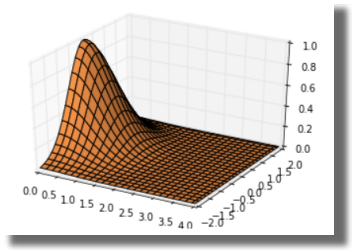
Make a 3D surface plot:

```
from mpl_toolkits.mplot3d import
Axes3D
fig = plt.figure()
ax = fig.add_subplot(111,
projection='3d')
ax.plot_surface(x,y,exp(-x**2-
y**2),rstride=2,cstride=2,color=[1,0.
6,0.3])
plt.show()
```

Save a figure into a pdf:

```
f = plt.figure()
plt.plot(x,cos(x))
plt.show()
f.savefig("test.pdf",
bbox_inches='tight')
```





Define a function. This one depends on a global parameter K (that has to be defined elsewhere):

```
def f(x):
    global K
    return(sin(K*x))
```

Note: One can have more than one return-value, in the style "`return(a,b)`"; and then "`A,B=f(x)`".

Some more linear algebra

Set up a 4x3 matrix (first with zeros, then set some elements):

```
A=zeros([4,3])
A[0,0]=1; A[2,1]=-2; A[3,2]=42
```

The same elements could have been set more efficiently by defining the locations and values:

```
index1=[0,2,3]; index2=[0,1,2]
values=[1,-2,42]
A[index1,index2]=values
```

Define a vector and perform matrix-vector multiplication:

```
b=[12,3.5,-7]
v=dot(A,b) # will be length 4 vector
```

This also works for matrix multiplication:

```
A=zeros([4,3]); B=zeros([3,7])
C=dot(A,B) # will be 4x7 matrix
```

Create random numbers. Gaussians in a 3x3 matrix:

```
M=random.randn(3,3)
```

Uniformly distributed, in a vector of length 3:

```
v=random.uniform(low=-5,high=5,size=3)
```

Solve the linear set of equations M*X=v:

```
X=linalg.solve(M,v)
```

Get all the eigenvalues and -vectors of M:

```
values,vectors=linalg.eig(M)
print(values)
```

Get the second column vector (index 1):

```
vec1=vectors[:,1]
```

Check eigenvalue equation is fulfilled (almost zero):

```
print(dot(M,vec1)-values[1]*vec1)
```

Some more array tricks

Find the dimensions of an arbitrary matrix:

```
shape(A)
```

Leaving indices free using the ":" means that the result will range over all possible values of that index.

```
A=zeros([3,7,5,4])
A[2,:,3,:] # will be a 7x4 matrix
```

If python encounters the sum of two arrays of unequal number of dimensions, it tries to "broadcast" (make compatible) the dimensions:

```
A=zeros([4,5])
b=zeros(5)
print(shape(A+b))
```

Sometimes, you have a 2D array (an image) and you want to apply a function elementwise. If that function expects a 1D array, what to do? Use flatten and reshape:

```
A=zeros([30,40])
Aflat=A.flatten() # makes a 1D array
result=f(Aflat) # your 1D function
B=reshape(result,[30,40]) # back to 2D
```


Generating a list. As opposed to a numpy array, a list can hold objects of different shape, e.g. matrices of different sizes. Here, the list is created using an 'iterator' over an array holding the dimensions:

```
dimensions=[2,7,4]
Matrices=[random.randn(n,n) for n in
dimensions]
print(shape(Matrices[1])) # is [7,7]
```

Similar example. This will create a 2x7 matrix and a 7x4 matrix:

```
dims=[2,7,4]
Weights=[random.randn(dims[j],dims[j
+1]) for j in range(2)]
```

Differential Equations

Solving an ordinary differential equation. In this example, we solve the equation of motion of a damped harmonic oscillator, by converting it into a set of two first-order differential equations. y[0] will be x(t) and y[1] will be the velocity, dx/dt. Therefore, dy[0]/dt=y[1] and dy[1]/dt=-(omega^2)*y[0]-gamma*y[1]. The initial conditions are stored in the vector y0, and ts is an array of time points at which the solution is desired:

```
from scipy import integrate
def f(y,t):
    global gamma, omega
    return([y[1],-(omega**2)*y[0]-
gamma*y[1]])
gamma=0.1; omega=1.0; y0=[0,1]
tmax=30
ts=linspace(0,tmax,100)
solution=scipy.integrate.odeint(f,y0,
ts)
plt.plot(ts,solution[:,0],)
plt.show()
```

PYTHON

Fourier transform

Calculate the fast Fourier transform of some array, using the numpy package fft. In this example, first set up a pulse of frequency 5, with some Gaussian envelope:

```
N=400
tmax=10.0
ts=linspace(-tmax,tmax,N)
y=sin(5*ts)*exp(-0.3*ts**2)
```

Now calculate and plot the Fourier transform. Note that zero frequency will sit at index 0. For plotting purposes, it is nicer to shift that to the middle of the interval, which is done using fftshift. Note that 'fftfreq' returns the frequencies, but to get 'omega' type frequencies (preferred by theoreticians), we need to multiply by 2pi:

```
yFourier=fft.fft(y)
dt=ts[1]-ts[0]
omegas=2*pi*fft.fftfreq(N,dt)
plt.plot(fft.fftshift(omegas),abs(fft
.fftshift(yFourier)))
```

WHERE TO LEARN MORE ABOUT PYTHON

Go to the website
https://www.python.org