Lab 1 - Introduction to Python and numpy

- INFO 371 - Winter 2018
- Deadline: 2017-01-11 11:59 pm (tomorrow midnight)

In [6]:

```
## import the necessary tools, and check their corresponding versions
import numpy as np
print("numpy:", np.__version__)
import pandas as pd
print("pandas:", pd.__version__)
import matplotlib.pyplot as plt

%matplotlib inline
```

```
numpy: 1.13.3
pandas: 0.20.3
```

1. Basic numpy functionality. Solve these problems using numpy,

not loops! Note: consult McKinney Chapter 4.

a) create a 5x5 matrix of zeros x.

In [7]:

```
# insert your code here
x = np.zeros((5, 5))
print(x)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

b) create an empty matrix z, as large as x.

In [8]:

```
# your code here
z = np.zeros(x.shape)
# x = np.empty_like(x)
print(z)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

c) create a 8x7 matrix of random normals y. Note: use numpy

functions for this.

In [49]:

```
# do it here
y = np.random.normal(size = (8, 7))
print(y)
```

```
[[ 1.58447321 -1.00932178  1.21145518  0.38047708 -1.36602183 -0.90667125
  -0.09005057]
 [ 0.93550483  1.86140905  0.95406074  1.01716016 -0.26237948 -0.74250253
   1.63179732]
 [ 0.08371252  0.77327862 -3.67838223 -0.98768097 -0.16992509 -1.18467365
   0.42247558]
 [ 1.37288814 -0.76801236  0.17523301  0.45104012  1.20942442  0.79876247
```

```
 -1.38486043]
 [-0.40764573  0.22235688 -0.81601327  0.78833226 -0.52970218 -0.72773249
   1.20082965]
 [-0.01875194 -1.92104557 -1.99298457 -0.82428214  1.46102801 -1.30307117
  -1.70801866]
 [-0.71800869  0.57443468 -0.91581253  0.01487106 -1.62321758 -2.2934418
  -0.44989406]
 [ 0.33044001  0.5982405   0.38457563 -0.70436397  1.18446307  0.94093975
  -0.37113037]]
```

d) replace all negative values in y with their absolute values.

In [50]:

```
# do it here
y = np.abs(y)
print(y)
```

```
[[ 1.58447321  1.00932178  1.21145518  0.38047708  1.36602183  0.90667125
   0.09005057]
 [ 0.93550483  1.86140905  0.95406074  1.01716016  0.26237948  0.74250253
   1.63179732]
 [ 0.08371252  0.77327862  3.67838223  0.98768097  0.16992509  1.18467365
   0.42247558]
 [ 1.37288814  0.76801236  0.17523301  0.45104012  1.20942442  0.79876247
   1.38486043]
 [ 0.40764573  0.22235688  0.81601327  0.78833226  0.52970218  0.72773249
   1.20082965]
 [ 0.01875194  1.92104557  1.99298457  0.82428214  1.46102801  1.30307117
   1.70801866]
 [ 0.71800869  0.57443468  0.91581253  0.01487106  1.62321758  2.2934418
   0.44989406]
 [ 0.33044001  0.5982405   0.38457563  0.70436397  1.18446307  0.94093975
   0.37113037]]
```

e) extract from your y:

- rows 0,4,5
- elements 2,3 in rows 2,3 (4 elements in all)
- elements (0,0), (0,5), (5,0), (3,3)
- all elements that are larger than 1

In [51]:

```
# here is the place :-)
first_question = y[[0, 4, 5]]
print(first_question)
second_questions = y[1:3, 2:4]
print(second_questions)
thrid_qustions = y[[0,0,5,3], [0,5,0,3]]
print(thrid_qustions)
forth_questions = y[np.where(y > 1)]
print(forth_questions)
```

```
[[ 1.58447321  1.00932178  1.21145518  0.38047708  1.36602183  0.90667125
   0.09005057]
 [ 0.40764573  0.22235688  0.81601327  0.78833226  0.52970218  0.72773249
   1.20082965]
 [ 0.01875194  1.92104557  1.99298457  0.82428214  1.46102801  1.30307117
   1.70801866]]
[[ 0.95406074  1.01716016]
 [ 3.67838223  0.98768097]]
[ 1.58447321  0.90667125  0.01875194  0.45104012]
[ 1.58447321  1.00932178  1.21145518  1.36602183  1.86140905  1.01716016
  1.63179732  3.67838223  1.18467365  1.37288814  1.20942442  1.38486043
  1.20082965  1.92104557  1.99298457  1.46102801  1.30307117  1.70801866
  1.62321758  2.2934418   1.18446307]
```

2. Compare plain python and numpy

As there are no matrices in the plain python, let's use the numpy matrices for plain python too, just avoid the more efficient vectorized operations.

Hint: you may want to consult McKinney's "Python for Data Analysis" p 93+.

We will create two 2d matrices of data (x, y) and thereafter compute a value of z(x,y) for each of these data points.

a. create data.

- You will create n-by-n matrices. Pick an n. 100 is a good number.

In [52]:

```
# your code here:
n = 100
```

- pick a function you are going to calculate. $\sin(x^2 + y^2)$ is a good choice.

In [68]:

```
# define your function here
def func(x, y):
    return np.sin(x**2 + y**2)

print(func(1, 2))
...
```

```
-0.958924274663
```

Out[68]:

```
Ellipsis
```

- Your data will look like this: (x1,y1) (x2,y1) (x3,y1) ... (xn,y1)
  (x1,y2) (x2,y2) (x3,y2) ... (xn,y2)
  (x1,y3) (x2,y3) (x3,y3) ... (xn,y3)
  ...
  (x1,yn) (x2,yn) (x3,yn) ... (xn,yn)
  As a first step, you have to create the values (x1, x2, x3, ..., xn) and (y1, y2, y3, ..., yn). x and y values may or may not be equal.

  Hint: look up functions np.linspace and np.arange. You should create a range where the function behaves in an interesting way.

In [94]:

```
# compute the x and y ranges.
x = np.linspace(0, 100, n)
y = np.linspace(0, 100, n)
```

- create the matrices x and y

Note: check out np.meshgrid function.

In [95]:

```
# compute the matrices here
x_mesh, y_mesh = np.meshgrid(x, y)

print(x_mesh, y_mesh)
```

```
[[   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]
 [   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]
 [   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]
 ...,
 [   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]
 [   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]
 [   0.            1.01010101    2.02020202 ...,   97.97979798
    98.98989899  100.        ]] [[   0.            0.            0.         ...,    0.            0.
0.        ]
 [   1.01010101    1.01010101    1.01010101 ...,    1.01010101
    1.01010101    1.01010101]
```

```
 [   2.02020202    2.02020202    2.02020202 ...,    2.02020202
     2.02020202    2.02020202]
 ...,
 [  97.97979798   97.97979798   97.97979798 ...,   97.97979798
    97.97979798   97.97979798]
 [  98.98989899   98.98989899   98.98989899 ...,   98.98989899
    98.98989899   98.98989899]
 [ 100.          100.          100.         ...,  100.          100.          100.        ]]
```

b. now let's compute it in the plain python.

create empty results matrix z that can contain the results for all x and y values.

Hint: you may want to consult McKinney p 77+

In [96]:

```
# create the empty matrix here.
z = np.empty(shape = (n, n))
print(z)
```

```
[[  0.00000000e+000   2.34880409e-314   0.00000000e+000 ...,
    1.01660840e-080   3.23191374e+209   2.62685747e-051]
 [  3.09047438e-057   1.11859418e+185   1.10609456e+084 ...,
    2.66348258e+185   1.88948547e+025   9.39370462e-048]
 [  1.33203285e+059   8.40333904e+169   3.05147304e+059 ...,
    3.05146544e+035   4.06322282e-037   1.28525317e-056]
 ...,
 [  2.40710066e+069   1.09990897e+248   2.56585969e+102 ...,
    1.82154200e-052   1.13645322e+257   8.05422292e+169]
 [  7.92404327e+126   1.25786485e+233   5.62561654e+025 ...,
    2.21946071e+160   2.06520448e+079   6.10023054e-067]
 [  1.04833796e+069   3.47245573e+098   7.73255251e+030 ...,
    3.85702565e-047   4.23576593e+068   4.48035930e+021]]
```

loop through all rows and columns of z and compute the corresponding

z value based on the corresponding x and y values

In [97]:

```
# compute by loop here
for i in range(len(z)):
    for j in range(len(z[0])):
        z[i][j] = func(x_mesh[i, j], y_mesh[i, j])
```

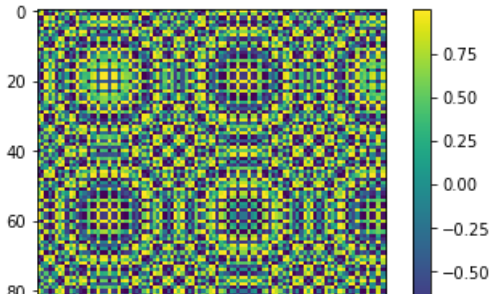
Now let's check if your calculations were correct. The best way (or at least the visually most pleasant way) to do it is by making an image of the result. You may use the command below, but feel free to modify this command to better suit your sense of beauty and your skills ;-)

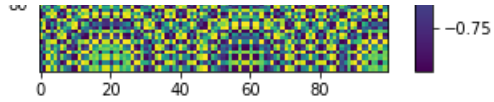
In [98]:

```
# feel free to use this code
plt.imshow(z, cmap=plt.cm.viridis)
plt.colorbar()
```

Out[98]:

```
<matplotlib.colorbar.Colorbar at 0x11b6ddb00>
```





- In order to measure the time of this calculation, you should wrap the computations above into a function:

In [145]:

```
# here you define a function that computes the z in the same
# element-wise way (and returns the result)
def computep(x, y):
    n = 100
    z = np.empty(shape = (n, n))
    x_mesh, y_mesh = np.meshgrid(x, y)
    for i in range(len(z)):
        for j in range(len(z[0])):
            z[i][j] = func(x_mesh[i, j], y_mesh[i, j])

    return z
```

and now measure how fast does it go. The best way to achieve it is the %timeit function i ipython.

Hint: consult McKinney p 63+

In [152]:

```
# measure the time here with %timeit
%timeit computep(x,  y)
```

```
32.5 ms ± 1.59 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

c. Now let's do this in numpy.

- compute the result in numpy vectorized way, and ensure the result looks correct. (you may want to plot it again)

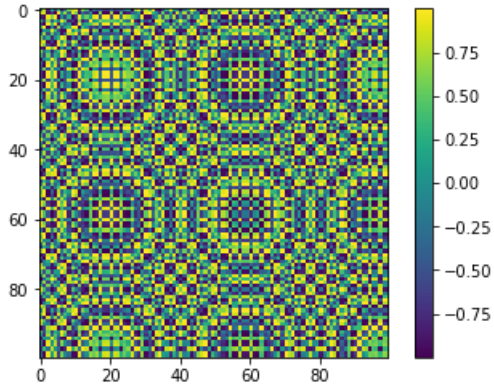
In [155]:

```
# do the vectorized computations and plot here
def tabulate(x, y):
    return np.vectorize(computep(x, y), otypes=[np.float64])

plt.imshow(z, cmap=plt.cm.viridis)
plt.colorbar()
```

Out[155]:

```
<matplotlib.colorbar.Colorbar at 0x11f2ebef0>
```



- finally, compute the speed of this approach. Use %timeit as above

In [156]:

```
%timeit tabulate(x, y)
```

```
31.4 ms ± 1.33 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```