python-tutorial-copy1

April 19, 2018

This very first exercise is supposed to give you a brief introduction into Python for scientific computing and image processing, because we are going to use Python in the remainder of this lecture.

1 Material availability

We will make the exercise sheets and required files available on our website: `https://hci.iwr.uni-heidelberg.de/ial/mlcv`

2 Python installation

Please install the Anaconda python distribution, python version 3.6 from `https://www.continuum.io/downloads`. Anaconda is a distribution of Python that ships most packages that are used in scientific computing and image processing. Note that we **will not give support** for other Python distributions and will **always run your code in the default Anaconda environment**

Python can be used in three ways: * either interactively in the console (`python` / `ipython`), or * you can write scripts (e.g. save it as `mycode.py`), and execute them by calling `python mycode.py`, or * interactively in a Jupyter Notebook.

We recommend to use the interactive Jupyter Notebooks. Jupyter is included in Anaconda and you can start it from the command line by just calling `jupyter notebook`.

3 Python introduction

3.1 The Scientific Python Ecosystem

In this course, we are going to use the Python programming language. There exist many highly usable and well maintained scientific libraries for Python. Packages that are useful for image analysis are:

- numpy, provides multi-dimensional arrays and fast numeric routines that work with these arrays.
- scipy, a collection of many scientific algorithms for areas such as optimization, linear algebra, integration, interpolation, Fourier transforms, signal and image processing. Makes use of numpy arrays.
- matplotlib, a plotting library which provides a MATLAB like interface. Check out the great gallery with many examples.
- scikit-learn, a quickly growing collection of machine learning algorithms, such as Support Vector Machines, Decision Trees, Nearest Neighbor Methods and many more. Their website offers good overview documentation and great examples, too.
- scikit-image, a collection of image processing algorithms.
- vigranumpy, a C++ library for multidimensional ar- rays, image processing and machine learning, developed in our group. It exposes most functions to Python via the vigranumpy module.
- pytorch a deep learning framework

Below is an introduction to Python with a focus on the elements needed in this lecture.

3.2 Python basics: interactive calculator, variables

Python is a powerful pocket calculator:

```
In [235]: 1 / 4 # WARNING: in python 2 this gives you integer division!

Out[235]: 0.25

In [236]: 3 + 7 - 5

Out[236]: 5

In [237]: # use ** for exponentiation: base ** exponent
          2 ** 3

Out[237]: 8
```

Store any kind of result in a variable and compute with variables. Variable names can be any name (no spaces!) that starts with a letter or underscore

```
In [238]: a = 3
          b = 8

          a * b

Out[238]: 24
```

3.3 Python basics: strings, lists, accessing ranges of values

Variables can store different things than numbers! For instance strings (single or double quotes)

```
In [240]: name = 'Carsten'
          name

Out[240]: 'Carsten'

In [241]: # lists are comma separated values enclosed in []-brackets
          listOfNumbers = [1, 2, 3]
          listOfStrings = ['cat', 'dog', 'apple', 'chair']

In [242]: # access element in list - zero-based indexing!
          listOfStrings[0]

Out[242]: 'cat'

In [243]: listOfNumbers[1]

Out[243]: 2

In [244]: # access elements in list from behind by using negative indices!
          listOfStrings[-1]

Out[244]: 'chair'

In [245]: listOfNumbers[-3]

Out[245]: 1
```

```
In [246]: # get a range of elements from a list: "startIndex:endIndex",
          # where endIndex is not included any more
          listOfNumbers[0:2]

Out[246]: [1, 2]

In [247]: listOfStrings[1:-1]

Out[247]: ['dog', 'apple']

In [248]: # if one of the bounds of a range is not specified,
          # the range will start at the beginning, or end at the end of the list
          listOfNumbers[:2]

Out[248]: [1, 2]

In [249]: listOfStrings[2:]

Out[249]: ['apple', 'chair']

In [250]: # omitting start AND end index gives you the full list as well
          bla = [1,2,3]
          blupp = bla
          blupp[1] = 3
          bla[:] = [5,6,7]
          blupp

Out[250]: [5, 6, 7]

In [251]: # strings are essentially lists of characters, can be indexed the same way
          name[3:]

Out[251]: 'sten'
```

3.4 Python basics: printing results a bit more structured

```
In [252]: print("Hello World")
          name = 'Carsten'
          number = 42
          number2 = 0
          print("Hello", name, ", the answer (", number2, ") to everything is", number)

Hello World
Hello Carsten , the answer ( 0 ) to everything is 42

In [253]: print(f"New in pytohn 3.6: Hello {name}, the answer ({number2}) to everything is {number}")

New in pytohn 3.6: Hello Carsten, the answer (0) to everything is 42
```

3.5 Python basics: comparisons, if / else, for-loops

Conditions e.g. using math relations (<, >, <=, >=, ==) can be checked with "`if`". An if-clause ends with a colon "`:`", and all lines of code that should be executed if the condition is true are indented (standard: 4 spaces, do **not** mix tabs and spaces!)

```
In [254]: if 1 > 3:
                  print("In what dimension do you live in???")
```

```
In [255]: # one can also specify what to do if the condition is not true
          if 1 > 3:
                  print("In what dimension do you live in???")
          else:
                  print("1 is not greater than 3!")
```

```
1 is not greater than 3!
```

```
In [256]: # several options can be concatenated (elif = else if)
          if 1 > 3:
                  print("In what dimension do you live in???")
          elif 1 < 3:
                  print("That's right!")
          else:
                  print("should never get here...")
```

```
That's right!
```

```
In [257]: # python knows the two basic truth types: True and False (booleans)
          truth = True
          truth
```

```
Out[257]: True
```

```
In [258]: # conditions evaluate to True or False
          truth = 1 > 3
          truth
```

```
Out[258]: False
```

```
In [259]: # conditions can use variables
          number = 32
          if (number / 8) > 3:
                  print("yay")
```

```
yay
```

```
In [260]: # conditions can be concatenated using "and" and "or"
          if 1 < 3 and 2 > 5:
                  print("aaargh")
```

```
In [261]: # conditions can be negated using "not"
          if 1 < 3 and not 2 > 5:
                  print("yay")
```

```
yay
```

```
In [262]: # for-loops can execute some code for each element of a list
          listOfNumbers = [1, 2, 3]
          for n in listOfNumbers:
                  print("Found", n)
```

```
Found 1
Found 2
Found 3
```

```
In [263]: # If you want to do something for all natural numbers in a range, use range([start],end).
          # Only specifying the end index starts at 0. The end index is again not included!
          for n in range(5):
              if n == 3:
                  continue
              print("Found", n)
```

```
Found 0
Found 1
Found 2
Found 4
```

```
In [264]: # this time we give a lower bound as well
          for n in range(2, 5):
                  print("Found", n, "with square:", n**2)
```

```
Found 2 with square: 4
Found 3 with square: 9
Found 4 with square: 16
```

3.6 Python basics: defining and calling functions

Let's define our own absolute function. Function definitions start with "`def`", follwed by the function name, and then in parentheses the function arguments. The end is again marked by a colon, and the following code is indented. Functions can return values using the "`return`" statement, which makes the function exit immediately.

```
In [265]: def absolute(x):
                  if x >= 0:
                          return x, 'a'
                  else:
                          return -x, 'b'
                  print("This is never reached")

          # To call a function, write its name, and then give arguments in parentheses.
          absolute(5)

Out[265]: (5, 'a')
```

```
In [266]: a,b = absolute(-17)
          b

Out[266]: 'b'
```

```
In [267]: # the returned value of a function can again be stored in a variable
          number = -12.34
          result = absolute(number)
          print("The absolute of", number, "is", result)
```

```
The absolute of -12.34 is (12.34, 'b')
```

3.7 Python basics: importing modules

We can use code from other "modules" in python by "importing" them. Numpy stands for "numerical python" and gives you most math features we will need in this lecture.

```
In [268]: import numpy

          # use a method and a variable defined in that module:
          numpy.cos(0.5 * numpy.pi)

Out[268]: 6.123233995736766e-17

In [269]: # The "as" statement allows you to specify an alias
          # to access functions from within that module with less typing.
          import numpy as np
          np.cos(0.5 * np.pi)

Out[269]: 6.123233995736766e-17
```

3.8 Numpy basics: multi dimensional arrays

```
In [270]: # define a 1D array from a list
          a = np.array([1, 2, 3, 4, 5])

          # indexing works like for lists
          a[2:-1]

Out[270]: array([3, 4])

In [271]: # numpy arrays have a "shape", which is the size in each dimension
          a.shape

Out[271]: (5,)

In [272]: # you can find the datatype of the values inside that array
          a.dtype

Out[272]: dtype('int64')

In [273]: # arrays can tell you their min and max value
          print(a.min(), a.max())

1 5

In [274]: # 2-dimensional arrays can be created as a list of lists (each inner list is a row!)
          b = np.array([[1,2], [3,4], [5,6]])
          b

Out[274]: array([[1, 2],
                 [3, 4],
                 [5, 6]])

In [275]: # the shape has now two entries (rows, columns)
          b.shape

Out[275]: (3, 2)

In [276]: # element acces now needs two indices or ranges
          b[0, 0]

Out[276]: 1

In [277]: b[2, 1]
```

```
Out[277]: 6

In [278]: # get the entries from ALL rows (= colon), second column (=index 1)
          b[:, 1]

Out[278]: array([2, 4, 6])

In [279]: # we can now work with the elements that we've adressed
          b[:, 1] = b[:, 1] + 4
          b[:, 1] += 4
          b

Out[279]: array([[ 1, 10],
                 [ 3, 12],
                 [ 5, 14]])

In [280]: # Arrays can also be created of a fixed size filled with zeros.
          # The shape must be specified as list if there are more than one dimension.
          c = np.zeros( [5,3] )
          c.shape
          c

Out[280]: array([[ 0.,  0.,  0.],
                 [ 0.,  0.,  0.],
                 [ 0.,  0.,  0.],
                 [ 0.,  0.,  0.],
                 [ 0.,  0.,  0.]])

In [281]: # or initialize an array with random numbers between 0 and 1
          d = np.random.random( [5,3] )
          d

Out[281]: array([[ 0.04027621,  0.40523663,  0.30725967],
                 [ 0.43931523,  0.17370666,  0.48747864],
                 [ 0.68915072,  0.24115091,  0.55294229],
                 [ 0.58314351,  0.69151291,  0.02111766],
                 [ 0.32546903,  0.59241179,  0.72848449]])

In [282]: # or initialize an array with a range of numbers: params are (start, stop, step)
          # where the stop value is not included any more
          e = np.arange(0.0, 1.0, 0.01)
          e.shape
          e

Out[282]: array([ 0.  ,  0.01,  0.02,  0.03,  0.04,  0.05,  0.06,  0.07,  0.08,
                  0.09,  0.1 ,  0.11,  0.12,  0.13,  0.14,  0.15,  0.16,  0.17,
                  0.18,  0.19,  0.2 ,  0.21,  0.22,  0.23,  0.24,  0.25,  0.26,
                  0.27,  0.28,  0.29,  0.3 ,  0.31,  0.32,  0.33,  0.34,  0.35,
                  0.36,  0.37,  0.38,  0.39,  0.4 ,  0.41,  0.42,  0.43,  0.44,
                  0.45,  0.46,  0.47,  0.48,  0.49,  0.5 ,  0.51,  0.52,  0.53,
                  0.54,  0.55,  0.56,  0.57,  0.58,  0.59,  0.6 ,  0.61,  0.62,
                  0.63,  0.64,  0.65,  0.66,  0.67,  0.68,  0.69,  0.7 ,  0.71,
                  0.72,  0.73,  0.74,  0.75,  0.76,  0.77,  0.78,  0.79,  0.8 ,
                  0.81,  0.82,  0.83,  0.84,  0.85,  0.86,  0.87,  0.88,  0.89,
                  0.9 ,  0.91,  0.92,  0.93,  0.94,  0.95,  0.96,  0.97,  0.98,  0.99])
```

```
In [283]: # Similar to the one above, might be more familiar to matlab users:
          # (start, stop, numberOfSteps), where stop IS included
          e = np.linspace(0.0, 1.0, 101)
          e.shape
          e
```

```
Out[283]: array([ 0.  ,  0.01,  0.02,  0.03,  0.04,  0.05,  0.06,  0.07,  0.08,
                  0.09,  0.1 ,  0.11,  0.12,  0.13,  0.14,  0.15,  0.16,  0.17,
                  0.18,  0.19,  0.2 ,  0.21,  0.22,  0.23,  0.24,  0.25,  0.26,
                  0.27,  0.28,  0.29,  0.3 ,  0.31,  0.32,  0.33,  0.34,  0.35,
                  0.36,  0.37,  0.38,  0.39,  0.4 ,  0.41,  0.42,  0.43,  0.44,
                  0.45,  0.46,  0.47,  0.48,  0.49,  0.5 ,  0.51,  0.52,  0.53,
                  0.54,  0.55,  0.56,  0.57,  0.58,  0.59,  0.6 ,  0.61,  0.62,
                  0.63,  0.64,  0.65,  0.66,  0.67,  0.68,  0.69,  0.7 ,  0.71,
                  0.72,  0.73,  0.74,  0.75,  0.76,  0.77,  0.78,  0.79,  0.8 ,
                  0.81,  0.82,  0.83,  0.84,  0.85,  0.86,  0.87,  0.88,  0.89,
                  0.9 ,  0.91,  0.92,  0.93,  0.94,  0.95,  0.96,  0.97,  0.98,
                  0.99,  1.  ])
```

3.9 Numpy basics: matrix and vector algebra

Most numpy math operations can be applied to matrices as well as scalars, then they are applied per element.

```
In [284]: a = np.array([1,2,3,4,5]) * np.pi
          a
```

```
Out[284]: array([  3.14159265,   6.28318531,   9.42477796,  12.56637061,  15.70796327])
```

```
In [285]: np.cos(a)
```

```
Out[285]: array([-1.,  1., -1.,  1., -1.])
```

Numpy can perform matrix multiplication and matrix-vector, but you have to use `np.dot`. $A * B$ does not give you real matrix multiplication!

```
In [286]: a = np.eye(3) * 2 # create identity matrix of shape 3x3, then multiply by 2
          a
```

```
Out[286]: array([[ 2.,  0.,  0.],
                 [ 0.,  2.,  0.],
                 [ 0.,  0.,  2.]])
```

```
In [287]: b = np.array([1,2,3]) # vector of shape 3(x1)
          b.shape
```

```
Out[287]: (3,)
```

```
In [288]: # one would think you could perform a matrix-vector multiplication
          # as follows:
          a * b
```

```
Out[288]: array([[ 2.,  0.,  0.],
                 [ 0.,  4.,  0.],
                 [ 0.,  0.,  6.]])
```

```
In [289]: # but if you want to get back a vector:
          np.dot(b, a)
```

```
Out[289]: array([ 2.,  4.,  6.])
```

```
In [290]: # let's try this for matrix-matrix multiplication
          c = np.array([[1,2,3], [4,5,6], [7,8,9]])
          print(c.shape)
          a * c

(3, 3)

Out[290]: array([[  2.,   0.,   0.],
                 [  0.,  10.,   0.],
                 [  0.,   0.,  18.]])
```

```
In [291]: np.dot(a, c)

Out[291]: array([[  2.,   4.,   6.],
                 [  8.,  10.,  12.],
                 [ 14.,  16.,  18.]])
```

You can access the transpose of a matrix by appending `.T`:

```
In [292]: # get the transpose
          c.T

Out[292]: array([[1, 4, 7],
                 [2, 5, 8],
                 [3, 6, 9]])
```

```
In [293]: # now we could also do c^T * a
          np.dot(c.T,a)

Out[293]: array([[  2.,   8.,  14.],
                 [  4.,  10.,  16.],
                 [  6.,  12.,  18.]])
```

3.10 Numpy basics: array access by a truth array

```
In [294]: # we can also store boolean values inside an array and use it as mask
          mask = np.array([True, False, True, False, True])
          a = np.array([1, 2, 3, 4, 5])
          a[mask]

Out[294]: array([1, 3, 5])
```

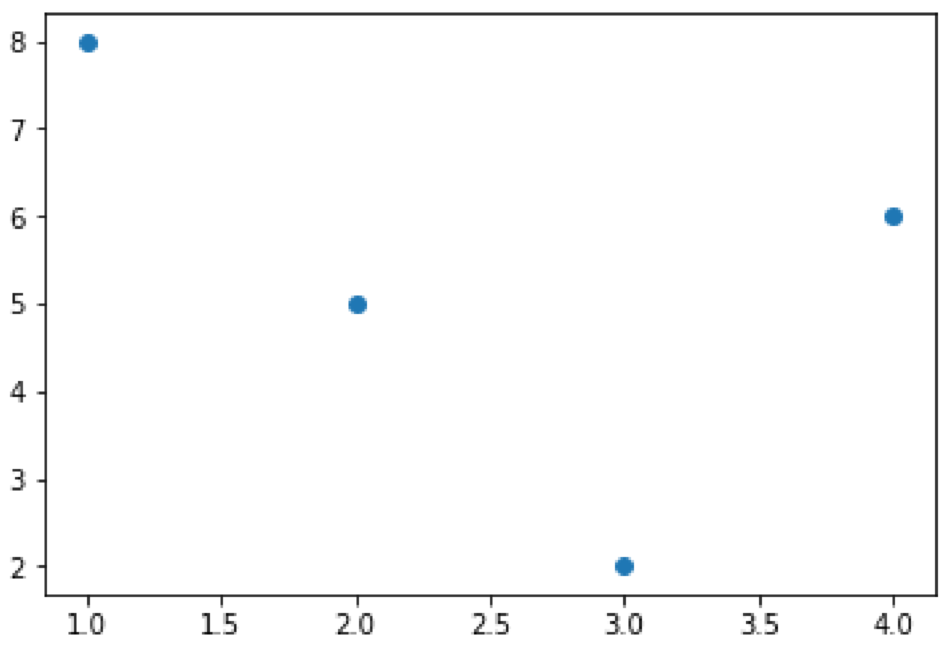
```
In [295]: # conditions on the full array can create these truth arrays
          # (by checking for each element)
          mask2 = a > 3
          mask2

Out[295]: array([False, False, False,  True,  True], dtype=bool)
```

```
In [296]: a[mask2]

Out[296]: array([4, 5])
```

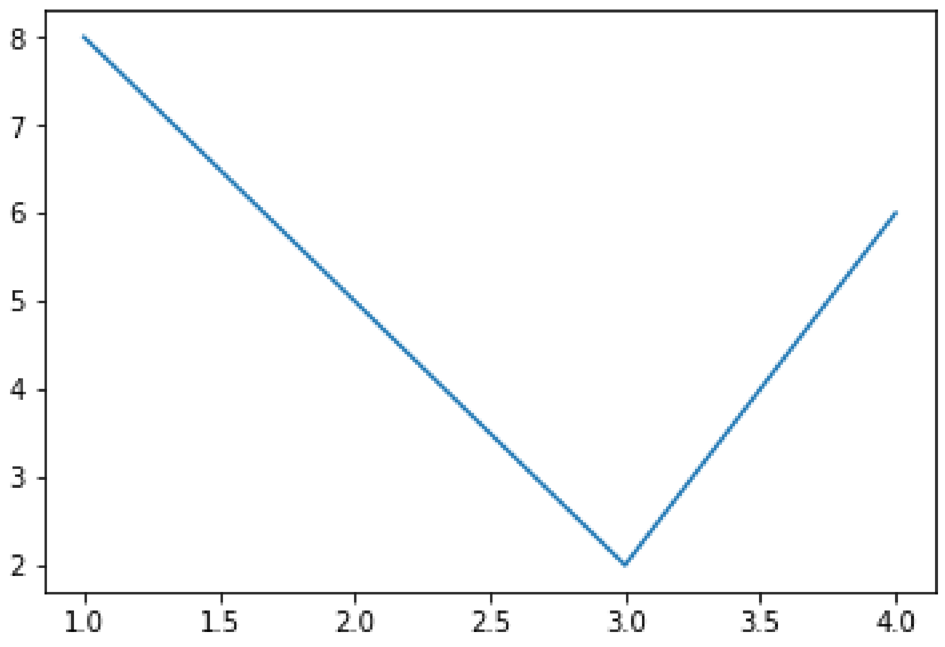
```
In [297]: # We could have written the condition inside the []-brackets.
          # Again, the returned values can be modified directly, e.g. using +=, ...
          print(a)
          a[a > 3] = 0
          a

[1 2 3 4 5]

Out[297]: array([1, 2, 3, 0, 0])
```

3.11 Plotting graphs and 2D-arrays

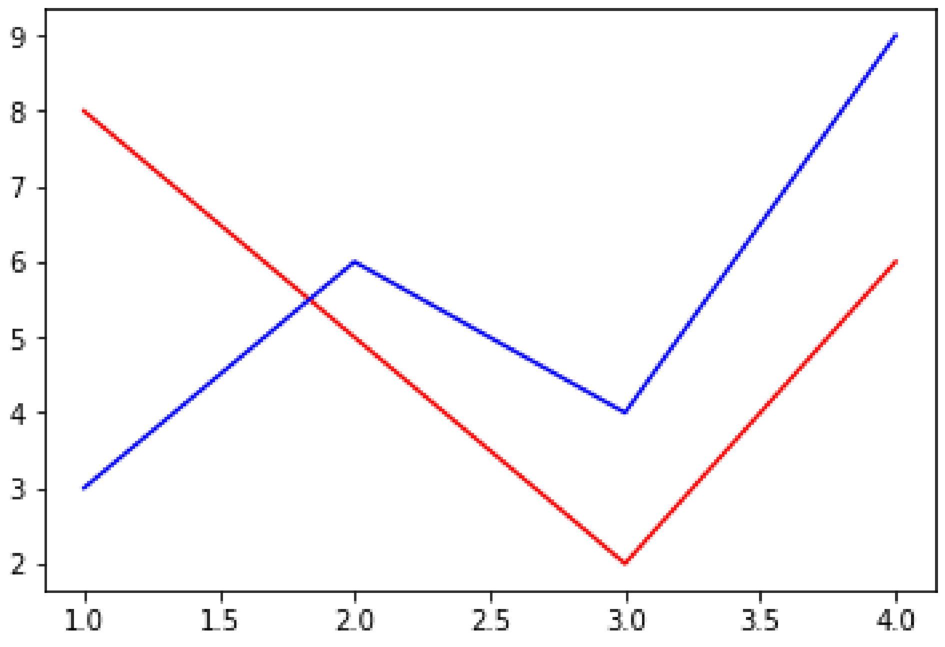
```
In [298]: # get plotting functionality with pretty much the same syntax as matlab plotting
          import matplotlib.pyplot as plt
          # the following line is only needed in ipython notebooks.
          # Do not put it into your python scripts
          %matplotlib inline

          # plot a set of points
          plt.figure()
          plt.scatter([1,2,3,4], [8,5,2,6])
          plt.show()
```



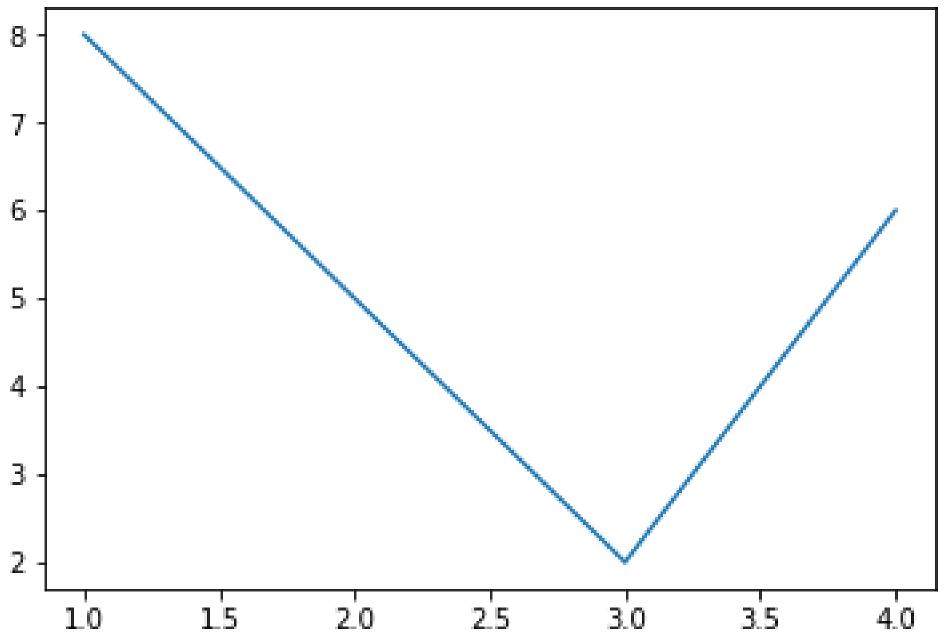
```
In [299]: # plot a line
          plt.figure()
          plt.plot([1,2,3,4], [8,5,2,6])
          plt.show()
```



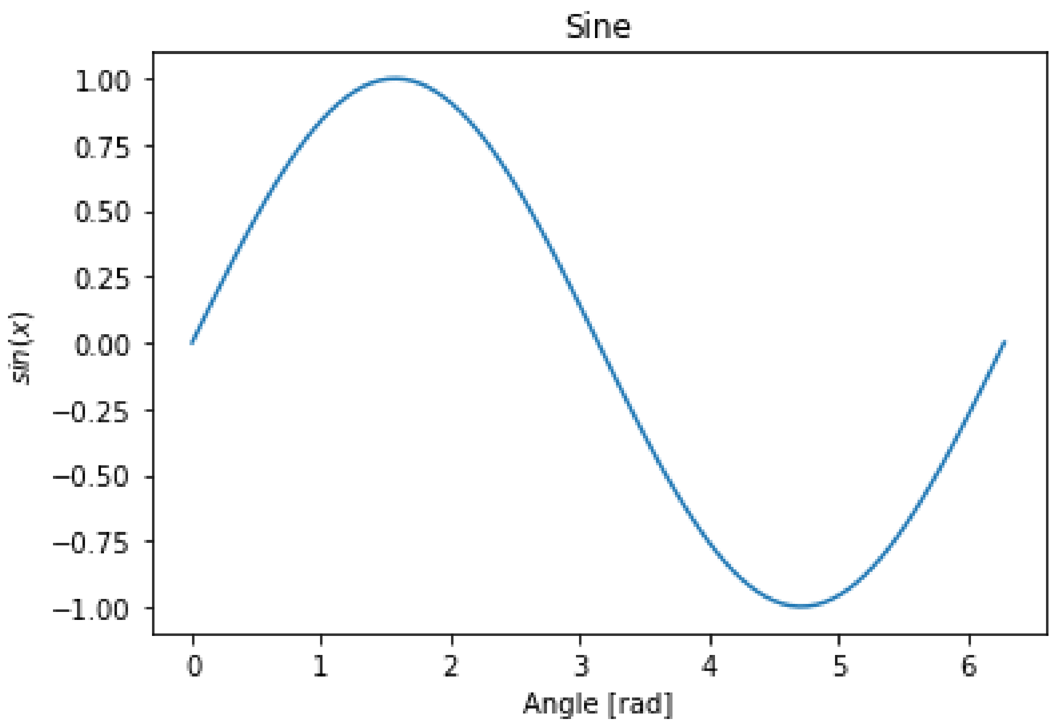
```
In [300]: # plot two lines with different colors
          plt.figure()
          plt.plot([1,2,3,4], [8,5,2,6], 'r') # r for red
          plt.plot([1,2,3,4], [3,6,4,9], 'b') # b for blue
          plt.show()
```



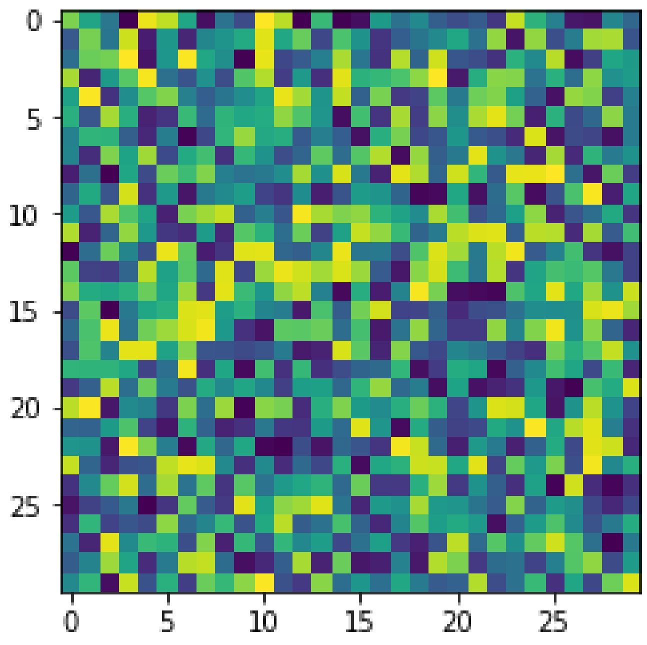
```
In [301]: # save a plot to a file
          plt.figure()
          plt.plot([1,2,3,4], [8,5,2,6])
          plt.savefig('test.png')
```



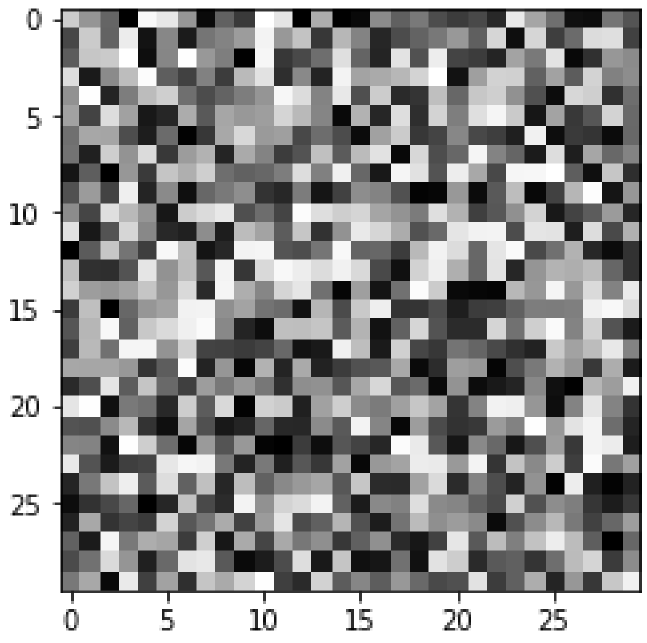
```
In [302]: # plot a sine function, give axis labeling
          plt.figure()
          x = np.linspace(0.0, 2.0*np.pi, num=100)
          plt.plot(x, np.sin(x))
          plt.xlabel('Angle [rad]')
          plt.ylabel('$sin(x)$')  # latex math is allowed in matplotlib!
          plt.title('Sine')
          plt.show()
```



```
In [303]: # plot a random 2D array as image
          # (by default it uses a heatmap-colorscheme:
          #  lowest value in array is blue, highest=red)
          randomImage = np.random.random( [30,30] )
          plt.figure()
          plt.imshow(randomImage, interpolation='nearest')
          plt.show()
```



```
In [304]: # plot it in grayscale
          plt.figure()
          plt.imshow(randomImage, cmap='gray', interpolation='nearest')
          plt.show()
```



3.12 Loading and displaying images

```
In [305]: # get documentation of the imread function:
          plt.imread?
```

```
Signature: plt.imread(*args, **kwargs)
Docstring:
Read an image from a file into an array.

*fname* may be a string path or a Python file-like object.  If
using a file object, it must be opened in binary mode.

If *format* is provided, will try to read file of that type,
otherwise the format is deduced from the filename.  If nothing can
be deduced, PNG is tried.

Return value is a :class:‘numpy.array‘.  For grayscale images, the
return array is MxN.  For RGB images, the return value is MxNx3.
For RGBA images the return value is MxNx4.

matplotlib can only read PNGs natively, but if ‘PIL
<http://www.pythonware.com/products/pil/>‘_ is installed, it will
use it to load the image and return an array (if possible) which
can be used with :func:‘~matplotlib.pyplot.imshow‘.
```

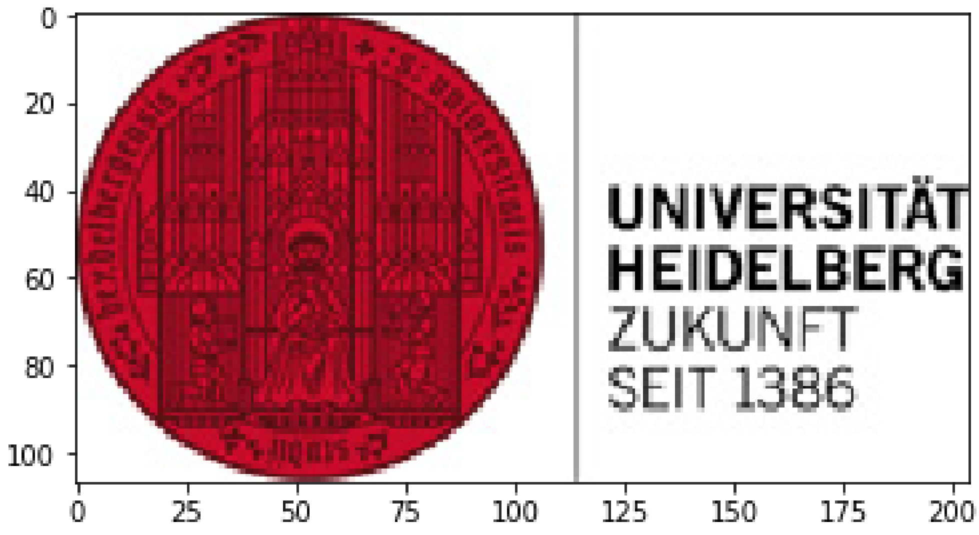
```
In [306]: # load an image. This one has 3 color channels, for Red Green Blue (=RGB).
          # you can get it e.g. in the ex 01 folder at http://tiny.cc/bmia16
          image = plt.imread('unihd.jpg')
          image.shape
```

```
Out[306]: (107, 204, 3)
```

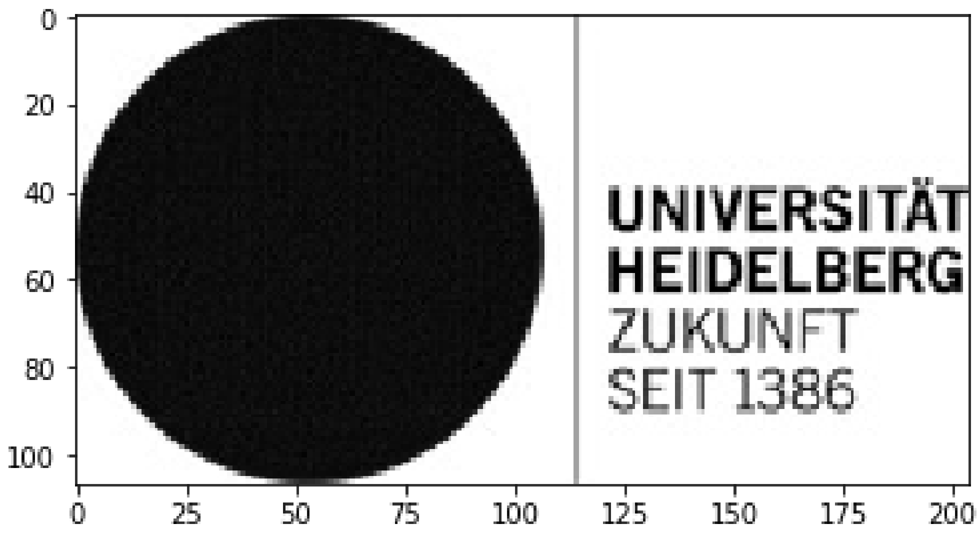
```
In [307]: image.dtype
```

```
Out[307]: dtype('uint8')
```

```
In [308]: # Show image, using the three color channels
          plt.imshow(image)
          plt.show()
```

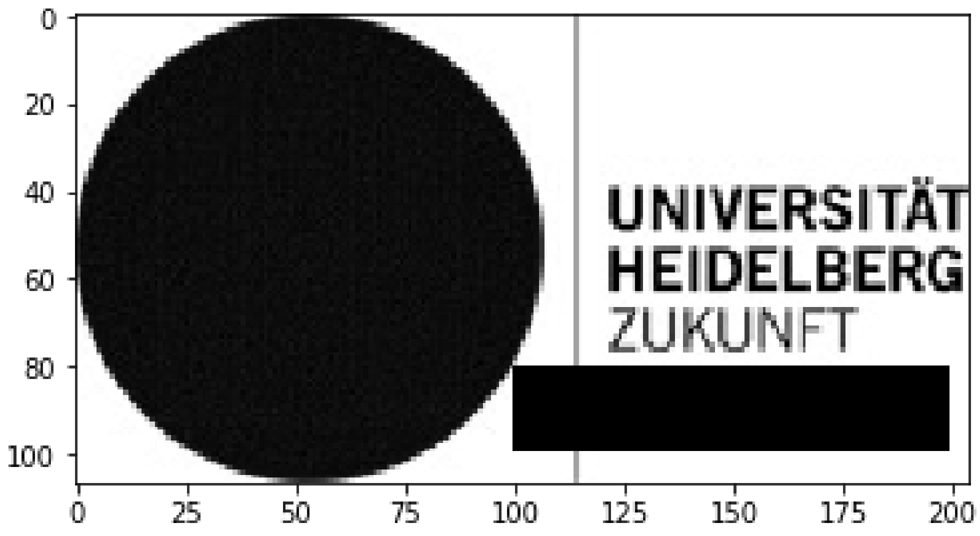


```
In [309]: # show only the green channel (closely related to grayscale intensity)
          grayImage = image[:,:,1]
          plt.imshow(grayImage, cmap='gray')
          plt.show()
```

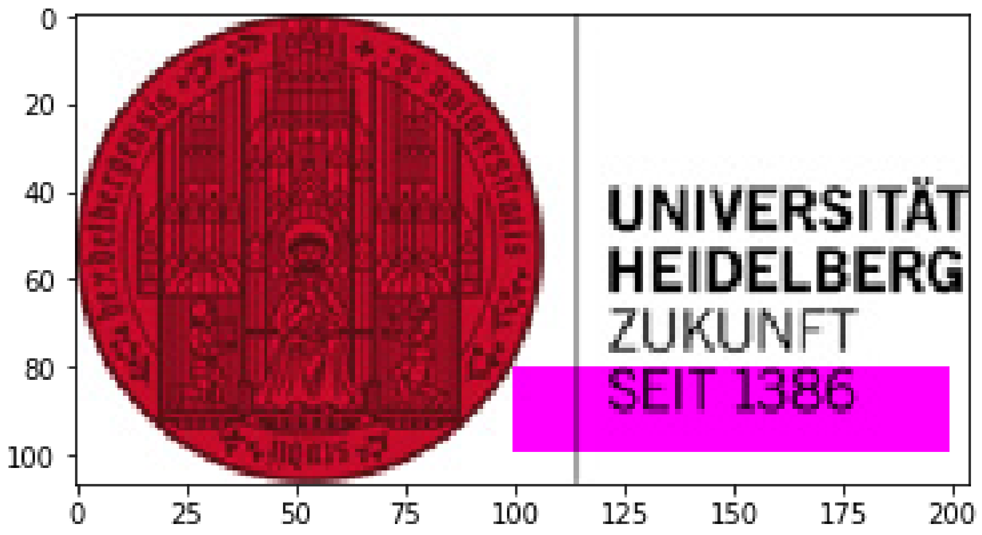


numpy slicing (e.g. taking out the green channel) gives you a view into the real data, it does not copy the values! So if we alter values in `grayImage`, `image` will also change! Let's set a block of the green channel to zero:

```
In [310]: grayImage[80:100, 100:200] = 0

          # see the change:
          plt.imshow(grayImage, cmap='gray')
          plt.show()
```



```
In [311]: plt.imshow(image)
          plt.show()
```



3.13 Complex numbers: 1 + 3j

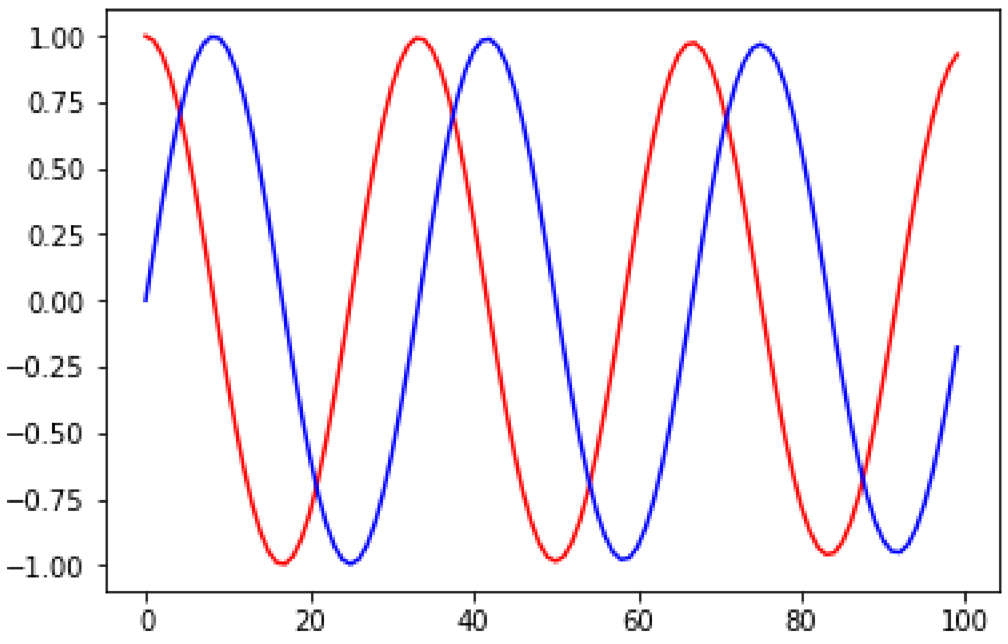
```
In [312]: # python has built-in support for complex numbers, where "j" = sqrt(-1).
          a = 3
          x = 1 + 3j
          y = 4 + 6j

In [313]: # supports operations
          x+y

Out[313]: (5+9j)

In [314]: x*y

Out[314]: (-14+18j)

In [315]: # .imag gives the imaginary component, .real the real
          x.real

Out[315]: 1.0

In [316]: (x+y).real

Out[316]: 5.0

In [317]: # set up a complex sinusoid function using cosine, sine, and exp (e ** something)
          x = np.arange(0.0, 1.0, 0.01) # range from 0 to 1 in 0.01 steps
          k = 3
          sigma = 3.0
          complexSinusoid = (np.cos(2 * np.pi * k * x) + 1j
                             * np.sin(2*np.pi*k*x)) * np.exp(-(x ** 2) / (2 * sigma**2))
          # this gives us 100 elements of a complex datatype!
```

```
In [318]: complexSinusoid

Out[318]: array([  1.00000000e+00 +0.00000000e+00j,
                   9.82281794e-01 +1.87380274e-01j,
                   9.29755824e-01 +3.68116372e-01j,
                   8.44285710e-01 +5.35800004e-01j,
                   7.28903833e-01 +6.84486260e-01j,
                   5.87703621e-01 +8.08904639e-01j,
                   4.25694144e-01 +9.04646105e-01j,
                   2.48622197e-01 +9.68319527e-01j,
                   6.27681980e-02 +9.97671938e-01j,
                  -1.25276846e-01 +9.91668350e-01j,
                  -3.08845366e-01 +9.50528298e-01j,
                  -4.81429937e-01 +8.75717805e-01j,
                  -6.36914254e-01 +7.69897079e-01j,
                  -7.69790156e-01 +6.36825800e-01j,
                  -8.75352999e-01 +4.81229383e-01j,
                  -9.49868438e-01 +3.08630964e-01j,
                  -9.90704697e-01 +1.25155109e-01j,
                  -9.96425627e-01 -6.26897867e-02j,
                  -9.66841280e-01 -2.48242648e-01j,
                  -9.03014190e-01 -4.24926223e-01j,
                  -8.07221175e-01 -5.86480513e-01j,
                  -6.82872018e-01 -7.27184840e-01j,
                  -5.34387952e-01 -8.42060671e-01j,
                  -3.67044264e-01 -9.27047987e-01j,
                  -1.86782653e-01 -9.79148955e-01j,
                  -1.83060289e-16 -9.96533799e-01j,
                   1.86678913e-01 -9.78605135e-01j,
                   3.66636663e-01 -9.26018506e-01j,
                   5.33498047e-01 -8.40658406e-01j,
                   6.81356210e-01 -7.25570668e-01j,
                   8.04982005e-01 -5.84853661e-01j,
                   9.00009154e-01 -4.23512161e-01j,
                   9.63088643e-01 -2.47279135e-01j,
                   9.92006895e-01 -6.24117837e-02j,
                   9.85763536e-01 +1.24530895e-01j,
                   9.44606023e-01 +3.06921102e-01j,
                   8.70019931e-01 +4.78297505e-01j,
                   7.64675290e-01 +6.32594415e-01j,
                   6.32330889e-01 +7.64356742e-01j,
                   4.77700006e-01 +8.68933086e-01j,
                   3.06282349e-01 +9.42640142e-01j,
                   1.24168209e-01 +9.82892582e-01j,
                  -6.21781778e-02 +9.88293836e-01j,
                  -2.46148365e-01 +9.58684587e-01j,
                  -4.21224338e-01 +8.95147284e-01j,
                  -5.81209725e-01 +7.99966557e-01j,
                  -7.20449369e-01 +6.76546989e-01j,
                  -8.34029467e-01 +5.29291195e-01j,
                  -9.17951190e-01 +3.63442587e-01j,
                  -9.69271630e-01 +1.84898452e-01j,
                  -9.86207117e-01 +3.62326617e-16j,
                  -9.68195260e-01 -1.84693124e-01j,
                  -9.15913564e-01 -3.62635833e-01j,
                  -8.31253997e-01 -5.27529828e-01j,
                  -7.17254476e-01 -6.73546786e-01j,
                  -5.77989735e-01 -7.95534621e-01j,
                  -4.18425516e-01 -8.89199483e-01j,
                  -2.44241304e-01 -9.51257073e-01j,
                  -6.16279321e-02 -9.79547930e-01j,
                   1.22932715e-01 -9.73112637e-01j,
                   3.02898048e-01 -9.32224336e-01j,
                   4.71896986e-01 -8.58377431e-01j,
                   6.23955769e-01 -7.54232960e-01j,
                   7.53709369e-01 -6.23522617e-01j,
                   8.56591006e-01 -4.70914891e-01j,
                   9.28993059e-01 -3.01848142e-01j,
                   9.68393707e-01 -1.22336575e-01j,
                   9.73444847e-01 +6.12439587e-02j,
                   9.44018210e-01 +2.42382680e-01j,
                   8.81208114e-01 +4.14665063e-01j,
                   7.87290813e-01 +5.72000258e-01j,
                   6.65641944e-01 +7.08836675e-01j,
                   5.20615083e-01 +8.20358103e-01j,
                   3.57385769e-01 +9.02653414e-01j,
                   1.81766598e-01 +9.52853874e-01j,
                   5.34135770e-16 +9.69233234e-01j,
                  -1.81463906e-01 +9.51267107e-01j,
                  -3.56196466e-01 +8.99649579e-01j,
                  -5.18018505e-01 +8.16266550e-01j,
                  -6.61219091e-01 +7.04126814e-01j,
                  -7.80757317e-01 +5.67253395e-01j,
                  -8.72439947e-01 +4.10539077e-01j,
                  -9.33068661e-01 +2.39571313e-01j,
                  -9.60551728e-01 +6.04327923e-02j,
                  -9.53976203e-01 -1.20515221e-01j,
                  -9.13638154e-01 -2.96859031e-01j,
                  -8.41029917e-01 -4.62360109e-01j,
                  -7.38784924e-01 -6.11176042e-01j,
                  -6.10582131e-01 -7.38067010e-01j,
                  -4.61013524e-01 -8.38580487e-01j,
                  -2.95419468e-01 -9.09207635e-01j,
                  -1.19697832e-01 -9.47505907e-01j,
                   5.99063121e-02 -9.52183564e-01j,
                   2.37022883e-01 -9.23143178e-01j,
                   4.05382971e-01 -8.61482665e-01j,
                   5.59040977e-01 -7.69453894e-01j,
                   6.92584807e-01 -6.50380425e-01j,
                   8.01326704e-01 -5.08537390e-01j,
                   8.81467969e-01 -3.48997858e-01j,   9.30231781e-01 -1.77451203e-01j])
```

```
In [319]: complexSinusoid.dtype
```

```
Out[319]: dtype('complex128')
```

```
In [320]: # plot a figure that shows the complex and the real parts in different colors
          plt.figure()
          plt.plot(complexSinusoid.real, 'r')
          plt.plot(complexSinusoid.imag, 'b')
          plt.show()
```



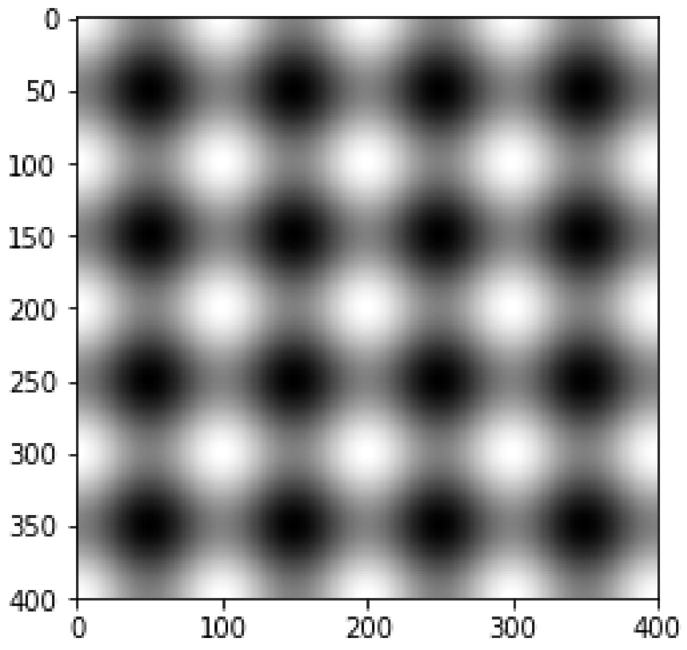
3.14 Create a 2D array of complex numbers

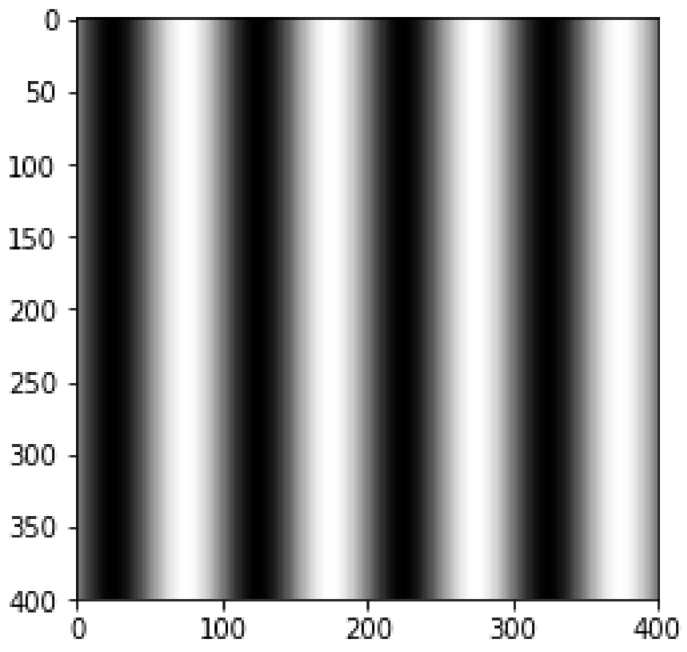
```
In [321]: # creating a 2D array filled with complex zeros:
          a = np.zeros([256,256], dtype=np.complex128)

In [322]: # set up a 2D coordinate grid:
          x = np.linspace(-4*np.pi, 4*np.pi, 400)
          y = np.linspace(-4*np.pi, 4*np.pi, 400)
          meshX, meshY = np.meshgrid(x, y)

          # evaluate some complex function for all coordinates on the grid
          pattern2D = np.cos(meshY) + np.exp(-1j * meshX)
          pattern2D.shape

Out[322]: (400, 400)

In [323]: meshY

Out[323]: array([[-12.56637061, -12.56637061, -12.56637061, ..., -12.56637061,
                  -12.56637061, -12.56637061],
                 [-12.50338129, -12.50338129, -12.50338129, ..., -12.50338129,
                  -12.50338129, -12.50338129],
                 [-12.44039196, -12.44039196, -12.44039196, ..., -12.44039196,
                  -12.44039196, -12.44039196],
                 ...,
                 [ 12.44039196,  12.44039196,  12.44039196, ...,  12.44039196,
                   12.44039196,  12.44039196],
                 [ 12.50338129,  12.50338129,  12.50338129, ...,  12.50338129,
                   12.50338129,  12.50338129],
                 [ 12.56637061,  12.56637061,  12.56637061, ...,  12.56637061,
                   12.56637061,  12.56637061]])
```

```
In [324]: pattern2D.dtype
```

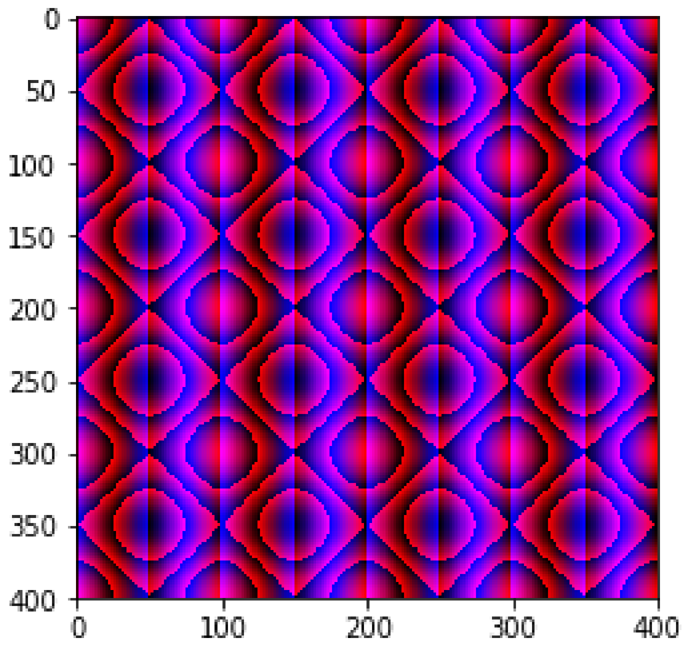
```
Out[324]: dtype('complex128')
```

```
In [325]: # show the real and imaginary part independently
          plt.figure()
          plt.imshow(pattern2D.real, cmap='gray')
          plt.show()

          plt.figure()
          plt.imshow(pattern2D.imag, cmap='gray')
          plt.show()
```





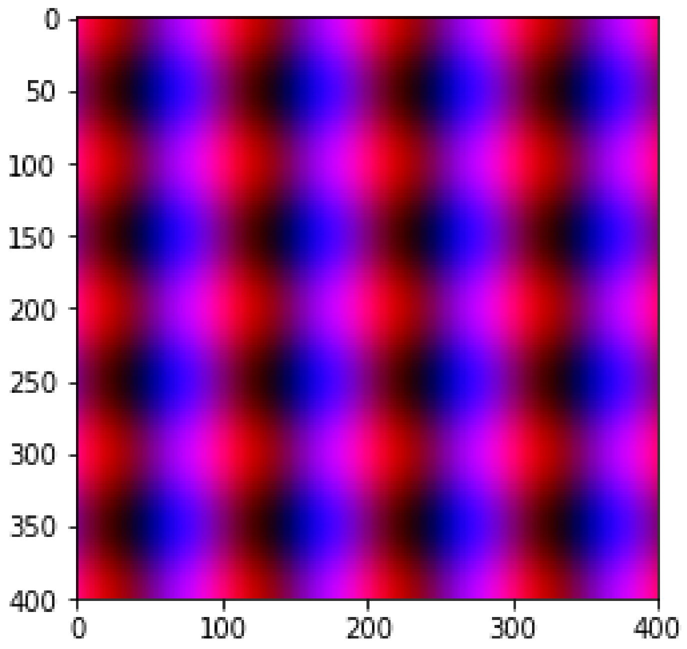
```
In [326]: # showing both channels together requires creating a RGB image of real numbers
          patternRGB = np.zeros([400,400,3], dtype=np.float64)
          patternRGB[:,:,0] = pattern2D.real # red channel for real part
          patternRGB[:,:,2] = pattern2D.imag # blue channel for imaginary part
          plt.figure()
          plt.imshow(patternRGB)
          plt.show()
```



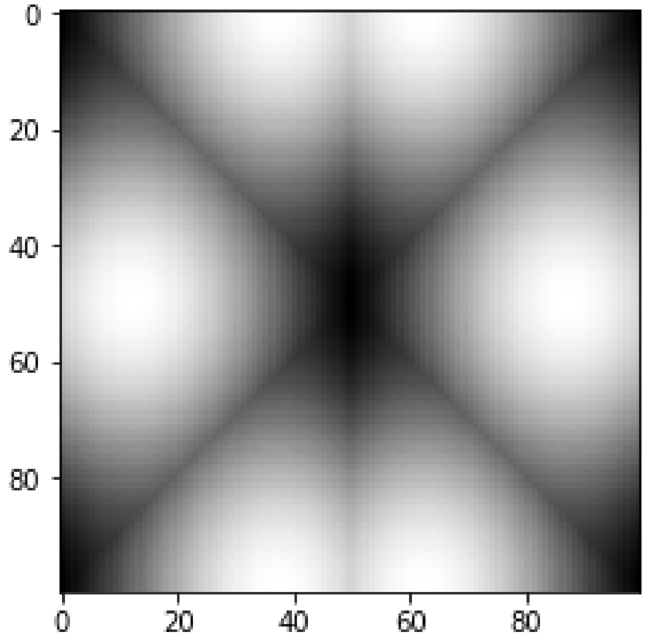
What happened above? The values in the red and blue channel now range from -1 to 1. Color images are expected to be in the range 0 to 1 (float), or 0 to 255 (integral), for each color channel. For grayscale images, `plt.imshow` can automatically normalize the data to the range $[0, 1]$. Unfortunately, for color images we have to perform this normalization on our own. For any matrix M, denote the min and max values by M_{min} and M_{max} respectively. Then the 0-1-normalization can be performed as

$$\hat{M} = \frac{M - M_{min}}{M_{max} - M_{min}}$$

```
In [327]: # Let's try again but scale the data
          patternRGB = np.zeros([400,400,3], dtype=np.float64)
          minR = pattern2D.real.min()
          maxR = pattern2D.real.max()
          minI = pattern2D.imag.min()
          maxI = pattern2D.imag.max()
          patternRGB[:,:,0] = (pattern2D.real - minR) / (maxR - minR)
          patternRGB[:,:,2] = (pattern2D.imag - minI) / (maxI - minI)
          plt.figure()
          plt.imshow(patternRGB)
          plt.show()
```



```
In [328]: # crop out an area and show the sum of the absolute values of real and imaginary channel:
          crop2D = pattern2D[200:300, 150:250]
          cropAbsolute = np.abs(crop2D.real) + np.abs(crop2D.imag)
          plt.figure()
          plt.imshow(cropAbsolute, cmap='gray')
          plt.show()
```



3.15 Further reading on python:

- full course held two weeks ago at Uni Heidelberg by Ullrich Koethe in German: `https://hackpad.com/Python-Kurs-fxJpABLeY8L`
- Spyder intro: `https://www.youtube.com/watch?v=_pIQEh2H11s`
- Python 3 basics: `http://askpython.com/`
- Extensive Python 3 tutorial: `http://www.python-course.eu/python3_course.php`
- Numpy course: `http://www.python-course.eu/numpy.php`
- In IPython, type `?` after a function or class to get more documentation! IPyhon also has tab-completion.