Intro to the NumPy Module

COT 4305 Python Programming

Dr. Ionut Cardei

Overview

- NumPy
- NumPy Examples
- SciPy overview

Reading List

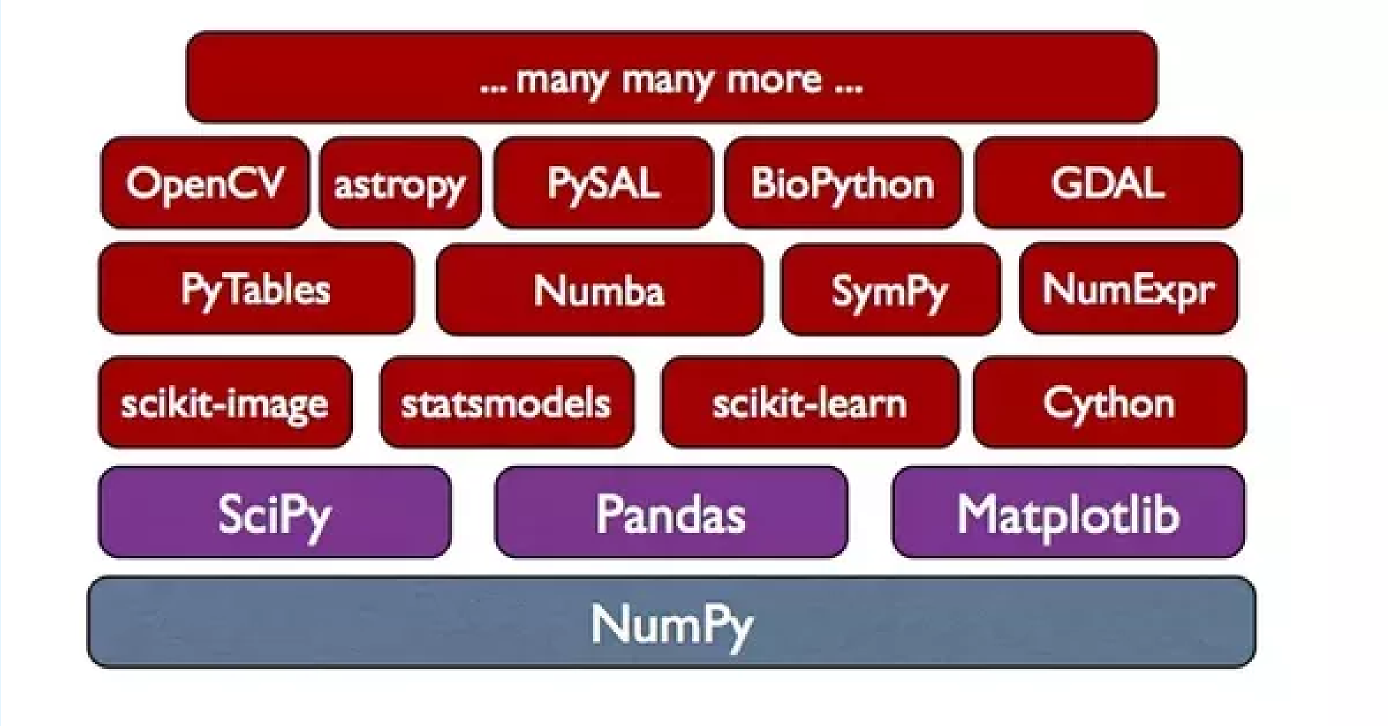
- The NumPy Tutorial at:

  https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
- Source files posted on the Canvas NumPy module

NumPy

- Open Source project since 2005.
- Numpy module offers:
  - Multidimensional homogeneous array/matrix of arbitrary types implemented efficiently in C
  - Efficient memory use
  - Wide variety of functions for array manipulation, matrix operations
  - Programs with numpy are much faster than using straight Python if most operations work on arrays instead of scalars.
  - NumPy programming style comparable with Matlab / Octave

Overview



- Numpy is at the core of a large module ecosystem for scientific computation for Python

Importing NumPy

- Standard style:
  - importing numpy as “np” keeps everyone in sync
  - style: np alias commonly used

```
>>> import numpy as np
>>>
```

The NumPy **ndarray** Object

- Implements a multidimensional array (matrix)
- Operations implemented in C/C++, fast
- A dimension is called “axis”
- Supports:
  - Slicing like Python lists
  - Indexing with int arrays
  - Indexing with boolean arrays
  - Reshaping
  - Transpose
  - Many np ‘universal’ functions: operating in element-by-element fashion: sin, cos, exp, log,...

ndarray Properties

- ndarray.ndim: the number of axes (dimensions) of the array (i.e. ‘rank’)
- ndarray.shape: the dimensions of the array, a tuple (m,n) if the array has m rows and n columns
- ndarray.size: the total number of elements of the array.
- ndarray.dtype: an object describing the type of the elements in the array. E.g. int32, int64, complex128
- ndarray.itemsize: the size in bytes of each element of the array. E.g. float64 has itemsize 8 (=64/8)
- ndarray.data: the buffer containing the actual elements of the array. Not recommended to access directly.

ndarray Example

- All examples are from file numpy-ex.py on Canvas.

```
>>> r = np.arange(12)
>>> r
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a = r.reshape(3, 4) # a is a 3 x 4 matrix
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.dtype
dtype('int64')
>>> a.shape
(3, 4)
>>> a.dtype
dtype('int64')
>>> a.ndim
2
>>> a.itemsize
8
>>> type(a)
<class 'numpy.ndarray'>
```

Creating an ndarray

- Can indicate data type as an optional parameter. Element data type inferred from argument.
- From a Python single (nested) list or tuple:

```
>>> ia = np.array([1, 2, 3])              # 1x3 int64 array
>>> ia
array([1, 2, 3])
>>> fa = np.array([1.0, 2.5, 3.14])       # 1x3 int64 array
>>> fa
array([ 1.  ,  2.5 ,  3.14])
>>> squares = np.array([i**2 for i in range(0, 4)])
>>> squares
array([0, 1, 4, 9])
>>> mat1 = np.array([[1,2,3], [4,5,6]])    # 2x3 matrix
>>> mat1
array([[1, 2, 3],
       [4, 5, 6]])
>>> mat2 = np.array([[1], [2], [3]])       # 3x1 matrix
>>> mat2
array([[1],
       [2],
       [3]])
```

Creating an ndarray

- Works with complex numbers, too.

```
>>> # create array with complex elements:
z = np.array([[1-1j, 2+1j], [-1j, 1+2j]], dtype=np.complex64)

>>> z
array([[ 1.-1.j,  2.+1.j],
       [-0.-1.j,  1.+2.j]], dtype=complex64)
```

- Indexing: using the [ ] operator, with one index per dimension:

```
>>> # the array indexing operator:
>>> squares[2]    # element at index 2
4
>>> squares[-1]   # element at last index, same rules as for Python lists
9
>>> # indexing in arrays with more than one dimension:
>>> z[0,1]        # first row, second column. Prints '2+1j'
(2+1j)
```

Creating an ndarray

- Create unit maxtrix of rank n: np.eye(n)
- Create array of 0s, 1s, or a non-initialized array:

```
>>> print(a)
[[0 0 0]
 [0 0 0]]
>>> a = np.zeros([2, 3], dtype=np.int16)   # create arrays of 0s:
>>> print(a)
[[0 0 0]
 [0 0 0]]
>>> b = np.ones([3, 3], dtype=np.float32)  # create array if zeros:
>>> print(b)
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
>>> # create an array with elements not initialized (garbage data):
>>> c = np.empty((3,5), dtype=np.int32)
>>> print(c)
[[-1061860360       32592    45519136           0           0]
 [          0           0           0           0           0]
 [          0           0           0           0           0]]
```

Creating an ndarray with Number Sequences

- np.arange(start, stop, step)
  - Similar to Python standard class/function range().
  - Number of elements not clear from parameters due to float rounding error

```
>>> a = np.arange(-3, 20, 4, dtype=np.int16)
>>> print(a)
[-3  1  5  9 13 17]
```

- np.linspace(first, last, number_of_elements) creates an array with the desired number of elements:

```
>>> a = np.linspace(2, 21, 10)
>>> print(a)
[  2.           4.11111111   6.22222222   8.33333333  10.44444444
  12.55555556  14.66666667  16.77777778  18.88888889  21.        ]
```

Creating an ndarray with Random Numbers

- numpy.random.rand(d0, d1, ..., dn):
  - Creates an array with random values U(0,1) in a given shape

```
>>> # create 2 x 3 array with random numbers in U(0,1):

>>> arand = np.random.rand(2, 3)

>>> print(arand)
[[ 0.51603197  0.67677556  0.37845946]
 [ 0.93328694  0.10663002  0.48977789]]
```

Creating an ndarray from a Function Over Coordinates

- numpy.fromfunction(function, shape, **kwargs)
  - Construct an array by executing a function over each coordinate.
  - *function* is a callable taking # of arguments equal to the number of dimensions in shape
  - Each parameter is an ndarray with the value of the coordinate for the corresponding axis
  - The resulting array therefore has a value fn(x, y, z) at coordinate (x, y, z).
  - dtype is the desired data type (default=float)

Creating an ndarray from a Function Over Coordinates

- numpy.fromfunction(function, shape, **kwargs)

```
>>> def fun(x, y):      # x, y  are arrays with coordinates for each element
        return x + y

>>> fa = np.fromfunction(fun, (2, 3), dtype=np.float32)
>>> print(fa)
[[ 0.  1.  2.]
 [ 1.  2.  3.]]
>>> # same, with lambda expression:
>>> fa = np.fromfunction(lambda x, y: x + y, (2, 3))
>>> print(fa)
[[ 0.  1.  2.]
 [ 1.  2.  3.]]
>>> # Look at the parameters passed to the function.
>>> # Each one is an ndarray with the values of the corresponding coordinate:
>>> dummy = np.fromfunction(lambda i, j: print("i={}\nj={}".format(i,j)), (2, 3), dtype=np.float32)
i=[[ 0.  0.  0.]
 [ 1.  1.  1.]]
j=[[ 0.  1.  2.]
 [ 0.  1.  2.]]
```

Reshaping an ndarray

- Change shape from m x n to p x q if the number of elements is preserved and m, n, p, q are positive integers.

```
>>> a = np.arange(0, 12)     # start with 1 x 12 array
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> b = a.reshape((3, 4))    # convert to 3 x 4
>>> print(b)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> c = b.reshape((2, 6))    # convert to 2 x 6
>>> print(c)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
>>> d = c.reshape((3, 5))    # shape mismatch: error
Traceback (most recent call last):
  File "<pyshell#237>", line 1, in <module>
    d = c.reshape((3, 5))    # shape mismatch: error
ValueError: total size of new array must be unchanged
```

Basic Operations

- Arithmetic operations (including *) and np functions are called element-wise.
- Relational operators return arrays with bool elements, like in Matlab.

```
>>> a = np.arange(0, 5)   # [0, 1, 2, 3, 4]
>>> b = np.arange(-2, 3)  # [-2, -1, 0, 1, 2]
>>> print(a + b)
[-2  0  2  4  6]
>>> print(a - b)
[2 2 2 2 2]
>>> print(a * b)
[ 0 -1  0  3  8]
>>>
>>> print(a**2)
[ 0  1  4  9 16]
>>> print(np.power(a, 3))
[ 0  1  8 27 64]
>>> print(b**a)
[ 1 -1  0  1 16]
>>> print(b > 0)
[False False False  True  True]
```

Matrix Multiplication in numpy

- Use the dot method and function:

```
>>> a = np.arange(0, 6).reshape((2, 3))
>>> b = np.arange(5, -1, -1).reshape((3, 2))
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> print(b)
[[5 4]
 [3 2]
 [1 0]]
>>> print(a.dot(b))
[[ 5  2]
 [32 20]]
>>> print(np.dot(a, b))
[[ 5  2]
 [32 20]]
```

In-place operators

- To modify the elements of a matrix using arithmetic operators use the in-place operators +=, -=, *=, /=, etc.

```
>>> a = np.arange(0, 6).reshape((2, 3))
>>> b = np.arange(5, -1, -1).reshape((2, 3))
>>> print("a={}\nb={}".format(a,b))
a=[[0 1 2]
 [3 4 5]]
b=[[5 4 3]
 [2 1 0]]
>>> a *= b      # call in-place multiplication operator: modifies a directly
>>> print(a)
[[0 4 6]
 [6 4 0]]
```

- The two operands for in-place operators should be of the same type.

ndarray unary operations

- Min, max, sum, cumulative sum
- Methods take an *axis* keyword parameter to indicate the dimension on which to apply the operation.
- Notice that the return from a.max(axis=1) should be a column array, with shape (2,1). Numpy instead returns a shape (2,), a tuple with one axis, i.e. one dimension.

```
>>> a = np.array([[3, 2, 4], [1, 5, 0]])
>>> a.sum()     # sum over all elements: 15
15
>>> a.min()     # returns 0
0
>>> a.max(axis=0)   # max of each column: [3, 5, 4]
array([3, 5, 4])
>>> a.max(axis=1)   # max of each row: returns [4, 5]
array([4, 5])
>>> a.sum(axis=0)   # sum of each column: returns [4, 7, 4]
array([4, 7, 4])
>>> a.sum(axis=1)   # sum of each row: returns [9, 6]
array([9, 6])
```

argmin and argmax

- numpy.argmax(a, axis=None, out=None)[source]
  - Returns the indices of the maximum values along an axis.
- Parameters:
  - a : array_like, input array.
  - axis : int, optional. By default, the index is into the flattened array, otherwise along the specified axis.
  - out : array, optional. If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.
- Returns:
  - index_array : ndarray of ints. Array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.

argmin and argmax

- If no axis argument given, argmin/max functions return the index in a flattened array.
- Examples.

```
>>> a = np.array([[3, 2, 4], [1, 5, 0]])
>>> print(a.argmax())          # 4, index in a flatten array
4
>>> print(a.argmin(axis=0))    # [1, 0, 1], indices for min on columns
[1 0 1]
>>> print(a.argmin(axis=1))    # [1, 2], indices for min on rows
[1 2]
```

Indexing – Single Axis Arrays

- Just like Python lists: indexing slicing, and iteration

```
>>> a = np.arange(20)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
>>> print(a[10])
10
>>> print(a[3:10])
[3 4 5 6 7 8 9]
>>> print(a[:10])
[0 1 2 3 4 5 6 7 8 9]
>>> print(a[-1])    # last element
19
>>> print(a[::-1])  # reverse order
[19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]
>>> print(a[10:3:-2])
[10  8  6  4]
>>> print(a[:4:-3])
[19 16 13 10  7]
>>> a[1::3] = -1    # assign new value to a[1],a[4],...,a[19]
>>> print(a)
[ 0 -1  2  3 -1  5  6 -1  8  9 -1 11 12 -1 14 15 -1 17 18 -1]
```

Indexing – Multi Axis Arrays

- With one index per axis. The parameter to [ ] is actually one tuple.
- Slicing works on each axis, as expected.
- **Important**: slicing does NOT create a new copy, but returns a 'view' of the original data and the returned object shares the data with the original array.
- When fewer indices are provided than the number of axes, the missing indices are considered complete slices.
- The expression within brackets in a[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes.
  - NumPy also allows you to write this using dots as a[i, ...].
  - e.g. a[1] is the same as a[1,:] and a[1,...]
- The dots (...) represent as many colons as needed to produce a complete indexing tuple.
- Notice that column vectors with n elements are represented by shape tuple (n,), and not (n,1). E.g. the shape of a[:, 1] is a one-element-tuple (3,) and not (3,1).

Indexing – Multi Axis Arrays

```
>>> a = np.arange(0, 12).reshape((3, 4))
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(a[1,2])       # 1,2 is passed as a tuple; second row, third column
6
>>> print(a[(1,2)])     # equivalent with passing a tuple; second row, third column
6
>>> b = a[:,:]          # returns an array that shares the data with a
>>> b[1, 2] = -1        # changes a[1,2]
>>> print(a[1,2])       # -1 instead of 6
-1
>>>
>>> # slicing on multiple axes:
>>> print(a[:1,:1])
[[0]]
>>> print(a[:2,:2])
[[0 1]
 [4 5]]
>>> print(a[::-1, ::-1])    # reverse order rows and columns in each row
[[11 10  9  8]
 [ 7 -1  5  4]
 [ 3  2  1  0]]
```

Indexing – Multi Axis Arrays

```
>>> # a[1] is the same as a[1,:] and a[1,...]
>>> print(a[1])
[ 4  5 -1  7]
>>> print(a[1,...])
[ 4  5 -1  7]
>>> # Notice that column vectors with n elements are represented by
>>> # shape tuple (n,), and not (n,1).
>>> c = a[:,1]
>>> print(c)
[1 5 9]
>>> print(c.shape)     # we would expect (3,1), but we get:
(3,)
```

Indexing – Multi Axis Arrays

- Indexing with … in more than 2D: example with 5 axes:
  - x[1,2,...] is equivalent to x[1,2,:,:,:],
  - x[...,3] to x[:,:,:,:,3] and
  - x[4,...,5,:] to x[4,:,:,5,:].

```
>>> a3 = np.arange(24).reshape((2,3,4))   # define array with 3 axes, 2x3x4,
>>> print(a3)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
>>> print(a3[1,:,:])     # same as a3[1]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
>>> print(a3[:,:,1])     # same as a3[...,1]:
[[ 1  5  9]
 [13 17 21]]
>>> print(a3[...,1])
[[ 1  5  9]
 [13 17 21]]
```

Iteration on Arrays

- Iteration on multi-axes arrays is done with respect to the first axis.
- To iterate over all array elements, use the flat array attribute.

```
>>> a = np.arange(12).reshape((3,4))   # define array with 2 axes, 3x4
>>> for b in a:
        print("b={}".format(b))


b=[0 1 2 3]
b=[4 5 6 7]
b=[ 8  9 10 11]
>>> # To iterate over all array elements, use the flat array attribute:
>>> for x in a.flat:
        print(x, end=' ')


0 1 2 3 4 5 6 7 8 9 10 11
```

Modifying the Shape

- ndarray.reshape(newshape):
  - Function that returns a new array with the desired shape without modifying the shape of the original array.
  - If new element count is different, ValueError is raised.
  - IMPORTANT: the elements are shared among the original and the reshaped array.
- ndarray.resize(newshape):
  - Modifies the array shape and size. If new size is larger, the new elements are set to 0. If smaller, array is truncated.
  - Cannot resize an array that references or is referenced by another array. ValueError will be raised.

Modifying the Shape

```
>>> a = np.arange(12)
>>> b = a.reshape((3,4))
>>> print(b)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> b[1, ::2] = -1        # show that b shares a's data
>>> print(a)
[ 0  1  2  3 -1  5 -1  7  8  9 10 11]
>>>
>>> c = np.arange(7)        # new array
>>> c.resize((2, 5))        # 3 new elements, set to 0:
>>> print(c)
[[0 1 2 3 4]
 [5 6 0 0 0]]
>>> # create a 'view' of c with a different shape:
>>> d = c.reshape((5,2))
>>> # attempt to resize c will cause ValueError:
>>> c.resize((3,2))
Traceback (most recent call last):
  File "<pyshell#598>", line 1, in <module>
    c.resize((3,2))
ValueError: cannot resize an array that references or is referenced
by another array in this way.  Use the resize function
```

Array Stacking Along Axes

- Vertical stacking: np.vstack() – stacks along first axis
- Horizontal stacking: np.hstack() – stacks along second axis
- np.r_ and np.c_ are useful objects with [ ] operator for stacking numbers along one axis. r_ for horizontal, c_ for vertical, using range literals (”:”).

```
>>> a = np.arange(4).reshape(2,2)
>>> print(a)
[[0 1]
 [2 3]]
>>> b = np.arange(4,8).reshape(2,2)
>>> print(b)
[[4 5]
 [6 7]]
>>> print(np.vstack((a,b)))      # vertical stacking
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
>>> print(np.hstack((a,b)))      # horizontal stacking
[[0 1 4 5]
 [2 3 6 7]]
>>> r = np.r [-1:3, 5, 7:10]        # horizontal stack numbers
>>> print(r)
[-1  0  1  2  5  7  8  9]
```

Array Splitting

- np.hsplit() / vsplit(): split array on horiz/vertical axis by specifying the number of equally shaped arrays to return, or by specifying the columns / rows after which the division should occur. Functions return list of new ndarray objects.
- **Caution:** new arrays **share data** with the original array.

```
>>> a = np.arange(8).reshape(2,4)
>>> print(a)
[[0 1 2 3]
 [4 5 6 7]]
>>> (b,c) = np.hsplit(a, 2)          # split a in 2 equal 2x2 subarrays
>>> print("b={} \nc={}".format(b,c))
b=[[0 1]
 [4 5]]
c=[[2 3]
 [6 7]]
>>> b[1,1] = 100                      # change b --> a[1,1] becomes also 100
>>> print(b)
[[  0   1]
 [  4 100]]
>>> print(a)
[[  0   1   2   3]
 [  4 100   6   7]]
```

Data Sharing and Copying

- When calling a function with an array parameter, it gets a mutable object reference. The actual and formal parameters refer to **the same object**. No copying/sharing are involved.
- Slicing and many functions (as seen in previous slides) return new array objects that **share array elements with the original array**. E.g. a.hsplit(), a.reshape(). These functions return **a view** to the same data. This is called a shallow copy.
  - One can create a view with the view() method:

    b = a.view()
- To create a complete copy of an array (including its data), do a deep copy:

  c = a.copy()
- Useful attributes:
  - a.flags.owndata is True if a owns its data (it's not a view)
  - b.base is set to the original array's data if b is a view of some other array
- IMPORTANT: when sharing is not desired, call copy() !

Data Sharing and Copying

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> print(a.flags.owndata) # True, a "owns" its data, it's not shared
True
>>> b = a.view()            # create a shallow copy (a view into a)
>>> print(a is b)           # False --> these are different objects
False
>>> print(a.flags.owndata) # True, a still "owns" its data
True
>>> print(b.flags.owndata) # False, b is a view of a (shallow copy)
False
>>> b[1] = -1               # change b, but also a[1]
>>> print(a)
[ 0 -1  2  3  4  5]
>>> # a.base is None because a owns its data. Not so for b:
>>> print("a's data buffer: {} and its base: {}".format(a.data, a.base))
a's data buffer: <memory at 0x7f50aa931d08> and its base: None
>>> print("b's data buffer: {} and its base: {}".format(b.data, b.base))
b's data buffer: <memory at 0x7f50aa931d08> and its base: [ 0 -1  2  3  4  5]
```

Indexing with Arrays of Indices

- If **a** is a an arbitrary array and **b** is an array of indices (int), then **a**[**b**] is an array with elements from **a** with the same shape as **b,** so that (**a**[**b**])[*index*] == **a**[**b**[*index*]], where *index* is a valid index tuple from **b**.

```
>>> a = np.arange(10)**2   # square numbers from 0 to 81
>>> print(a)
[ 0  1  4  9 16 25 36 49 64 81]
>>> b = [4, 1, 9]        # b.shape is (3,)
>>> print(a[b])          # a[b] shape is same as b's shape
[16  1 81]
>>> a = np.arange(10)**2        # square numbers from 0 to 81
>>> b = np.array([4, 1, 9])     # b.shape is (3,)
>>> print(a[b])                 # a[b] shape is same as b's shape
[16  1 81]
>>> c = np.array([[4,2,9], [8,1,3]])    # c.shape is (2,3)
>>> print(c)
[[4 2 9]
 [8 1 3]]
>>> print(a[c])                 # a[c] shape is same as c's shape
[[16  4 81]
 [64  1  9]]
```

Indexing with Arrays of Indices

- The index can have more than one axis.
- The arrays of indices for each dimension must have the same shape.

```
>>> a = np.arange(6).reshape(2,3)
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> b = np.array([[0, 1, 1, 0], [1, 0, 1, 0]])
>>> c = np.array([[0, 1, 0, 2], [0, 2, 1, 1]])
>>> print(a[b,c])    # (a[b,c])[i,j] == a[b[i],c[j]]
[[0 4 3 2]
 [3 2 4 1]]
>>> # The arrays of indices for each dimension must have the same shape.
>>> print(a[(b,c)])  # works if index arrays are in a list/tuple sequence
[[0 4 3 2]
 [3 2 4 1]]
```

Example with Array Indexing

- Problem:

  a) find maximum values and the corresponding times in a matrix with two time series: sin and cos

  b) replace in the data series the max values with -1

Example with Array Indexing

```
>>> time = np.linspace(0, 10, 5)  # create 5 time points
>>> print(time)
[  0.    2.5   5.    7.5  10. ]
>>> # create array with two time series:
>>> d = np.vstack([np.sin(time), np.cos(time)])
>>> print(d)
[[ 0.          0.59847214 -0.95892427  0.93799998 -0.54402111]
 [ 1.         -0.80114362  0.28366219  0.34663532 -0.83907153]]
>>> # find index for max value per-column, i.e. for each time point:
>>> maxindex = np.argmax(d, axis=0)    # this is a 2 x 5 array
>>> print(maxindex)
[1 0 1 0 0]
>>> maxvals = d[maxindex, np.arange(d.shape[1])]  # d.shape[1] is # of time points
>>> print(maxvals)
[ 1.          0.59847214  0.28366219  0.93799998 -0.54402111]
>>>
>>> # double-check that the found coordinates correspond to the max values:
>>> np.all(maxvals == d.max(axis=0))
True
>>> # part b):
>>> # replace max values (among sine and cos) for each time point with 1000.0:
>>> d[maxindex, np.arange(d.shape[1])] = -1
>>> print(d)
[[ 0.         -1.         -0.95892427 -1.         -1.        ]
 [-1.         -0.80114362 -1.          0.34663532 -0.83907153]]
```

Indexing with Boolean Arrays

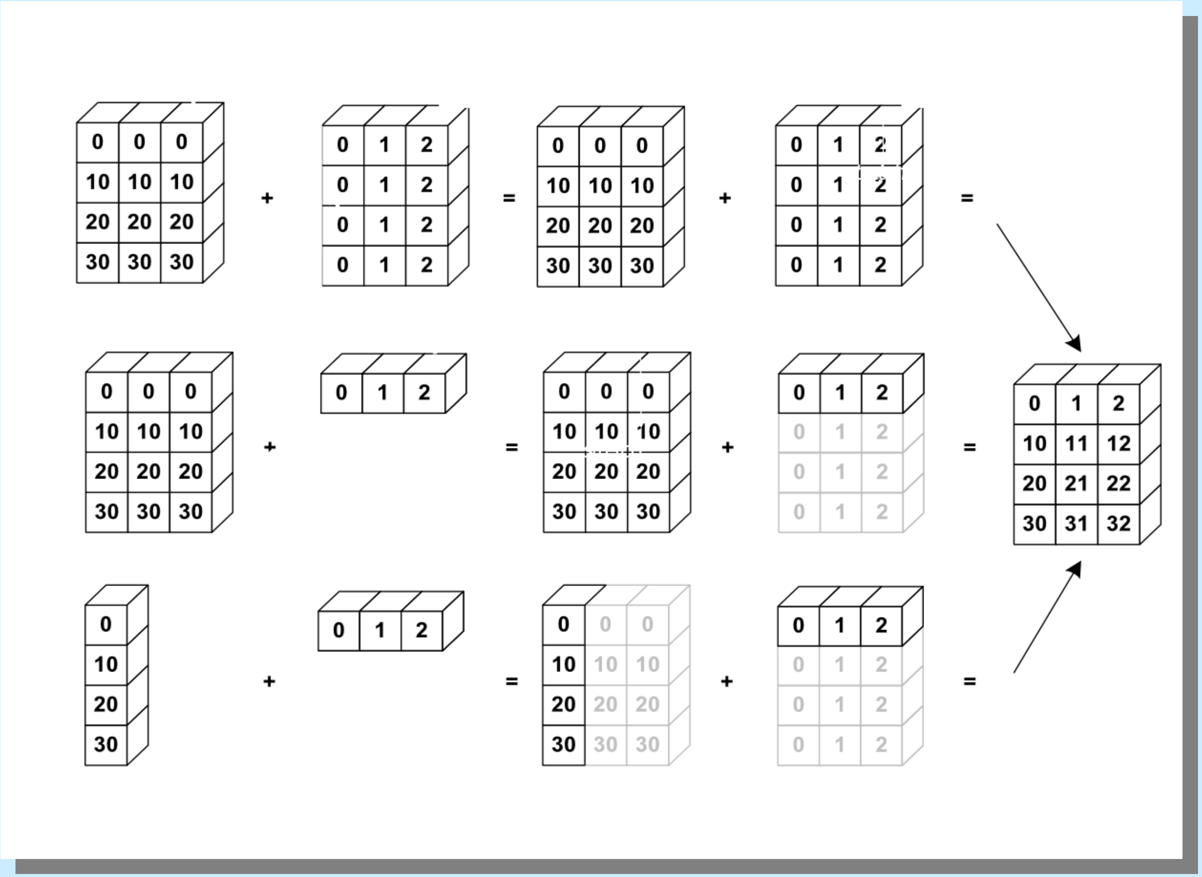
- For an arbitrary array **a** and a bool array **b** with the same shape, **a**[**b**] is a new array with elements from **a** for which **b**[index]==True
  - The bool index array works like a filter
  - Can reassign those values to something else:

    a[b] = new_value

```
>>> a=np.arange(12).reshape(3,4)
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> # create boolean index array: b[i,j]==True if a[i,j] is even:
>>> b = (a % 2) == 0
>>> print(b)
[[ True False  True False]
 [ True False  True False]
 [ True False  True False]]
>>> print(a[b])   # get all elements from a that are even:
[ 0  2  4  6  8 10]
>>> a[b] = -1     # reassign all even values from a to -1
>>> print(a)
[[-1  1 -1  3]
 [-1  5 -1  7]
 [-1  9 -1 11]]
```

Broadcasting

- Basic operations on numpy arrays (addition, etc.) are elementwise.
  - On arrays of the same size.
- But one can do operations on arrays of **different** sizes if numpy can transform arrays so they all have the same sizes.
  - Transformation is called **broadcasting .**
- Examples:



Broadcasting

- Numpy starts at the trailing dimension and goes forward.
- Rule #1: if all input arrays do not have the same number of dimensions, a “1” will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.
  - e.g. shapes (4,3) and (3,) become: (4,3) and (**1,** 3)
- Rule #2: ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the “broadcast” array.
  - e.g. now (4,3) and (**1,** 3) becomes: (4,3) and (**4**, 3), and the row is copied 3 more times for the second operand.
- After application of the broadcasting rules, the sizes of all arrays must match. More details can be found at https://docs.scipy.org/doc/numpy-dev/user/basics.broadcasting.html
- Why broadcasting ? So that array operations are done by the C language library (fast!) instead of Python code.

Broadcasting

- Two **dimensions are compatible** when
  - they are equal, or
  - one of them is 1
- If these conditions are not met, a ValueError: frames are not aligned exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.
- When either of the dimensions compared is 1, the other is used. In other words, dimensions with size 1 **are stretched or "copied"** to match the other.

Broadcasting

- Examples (just shapes are shown, colored axis copied):
  - (1,) and (2,3) → (1,1) and (2,3) → (2,3) and (2,3)
  - (10,10,3) and (10,3) → (10,10,3) and (1,10,3) → (10,10,3) and (10,10,3)
  - (100, 100, 3) and (3,) → (100, 100, 3) and (1, 3) → (100, 100, 3) and (1, 1, 3) → (100, 100, 3) and (100, 100, 3)
- Where broadcasting fails:
  - (4,2) and (3,1): not compatible: 4 != 3
  - (4,3,1) and (2,2): not compatible: 3 != 2

Broadcasting

```
>>> # (1,) and (2,3) → (1,1) and (2,3) → (2,3) and (2,3)
>>> np.array([1]) + np.array([[1,2,3], [4,5,6]])
array([[2, 3, 4],
       [5, 6, 7]])
>>> # (2,2,2) and (2,) → (2,2,2) and (1,2) → (2,2,2) and (1,1,2) → (2,2,2) and (2,2,2)
>>> np.arange(8).reshape(2,2,2) + np.array([10,20])
array([[[10, 21],
        [12, 23]],

       [[14, 25],
        [16, 27]]])
>>> # (3,1) and (2,) → (3,1) and (1,2) → (3,1) and (1,2) → (3,2) and (1,2) → (3,2) and (3,2)
>>> np.array([[1], [2], [3]]) + np.array([10, 20])
array([[11, 21],
       [12, 22],
       [13, 23]])
>>> # (2,2,3) and (3,) → (2,2,3) and (1,1,3) → (2,2,3) and (1,2,3) → (2,2,3) and (2,2,3)
>>> np.arange(12).reshape(2,2,3) + np.array([100,200,300])
array([[[100, 201, 302],
        [103, 204, 305]],

       [[106, 207, 308],
        [109, 210, 311]]])
```

Elements of Linear Algebra

- In numpy.linalg:
  - Matrix inverse: a.inv()

```
>>> a = np.array([[1, 3], [3, 4]])
>>> ainv = np.linalg.inv(a)
>>> print(ainv)
[[-0.8  0.6]
 [ 0.6 -0.2]]
>>> print(a.dot(ainv))   # we should get np.eye(2), with some roundoff error
[[  1.00000000e+00  -1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00]]
```

  - Solve linear equation: A*x = b: np.linalg.solve() :

```
>>> A = np.array([[2, 1], [3, -2]])
>>> b = np.array([4, -1])
>>> x = np.linalg.solve(A, b)
>>> print(x)
[ 1.  2.]
>>> print(A.dot(x) - b)      # Should be [0,0] with some roundoff error.
[  8.88178420e-16   0.00000000e+00]
```

Elements of Linear Algebra

- Matrix trace (sum of diagonal elements): a.trace()
- Matrix transpose: a.T
- Matrix determinant: np.linalg.det(a)
- Matrix rank: np.linalg.rank(a)
- Matrix norm: np.linalg.norm(a)   (for default norm-2, Euclidean)
- Eigenvalues & vectors: (eigvals, eigvects) = np.linalg.eig(a)

NumPy Examples

- Represent images with NumPy and display with matplotlib.pyplot
- Compute and display the Mandelbrot fractal
- Animation with smooth image transition

Representing Images

- An image is a 2D matrix W x H of pixels.
- Each pixel could be:
  - 3 int values for red, green, blue (RGB) – most common.
  - 4 int values for red, green, blue, transparency – α (RGBA)
  - A value (int or float) that maps to an index into a palette, which is an array of RBG values.
- Matplotlib defines a rich set of colormaps. Check them out at https://matplotlib.org/examples/color/colormaps_reference.html
  - Advantage of using colormaps: no need to select custom RGB values when displaying matrix data of form W x H x range, where range is int or float, all scaled to the [0,1] interval
  - A photograph does not need a colormap – it includes RGB values as a W x T x 3 array.

Display Images

- We can use matplotlib.pyplot (similar to pylab).
- matplotlib.pyplot.subplots() creates grid of subfigures
- matplotlib.pyplot.imshow(imagemaxtrix,…) displays image
- fig.colorbar() creates a colorbar legend
- Example image 2D matrix, 50 x 50:

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [ 1  2  3  4  5  6  7  8  9 10]
 [ 2  3  4  5  6  7  8  9 10 11]
 [ 3  4  5  6  7  8  9 10 11 12]
 [ 4  5  6  7  8  9 10 11 12 13]
 [ 5  6  7  8  9 10 11 12 13 14]
 [ 6  7  8  9 10 11 12 13 14 15]]
```

Image with Colormap

- Plot the same 50x50 matrix with values from 0 to 98, with 6 different colormaps. The colorbar is at the right of each subplot:

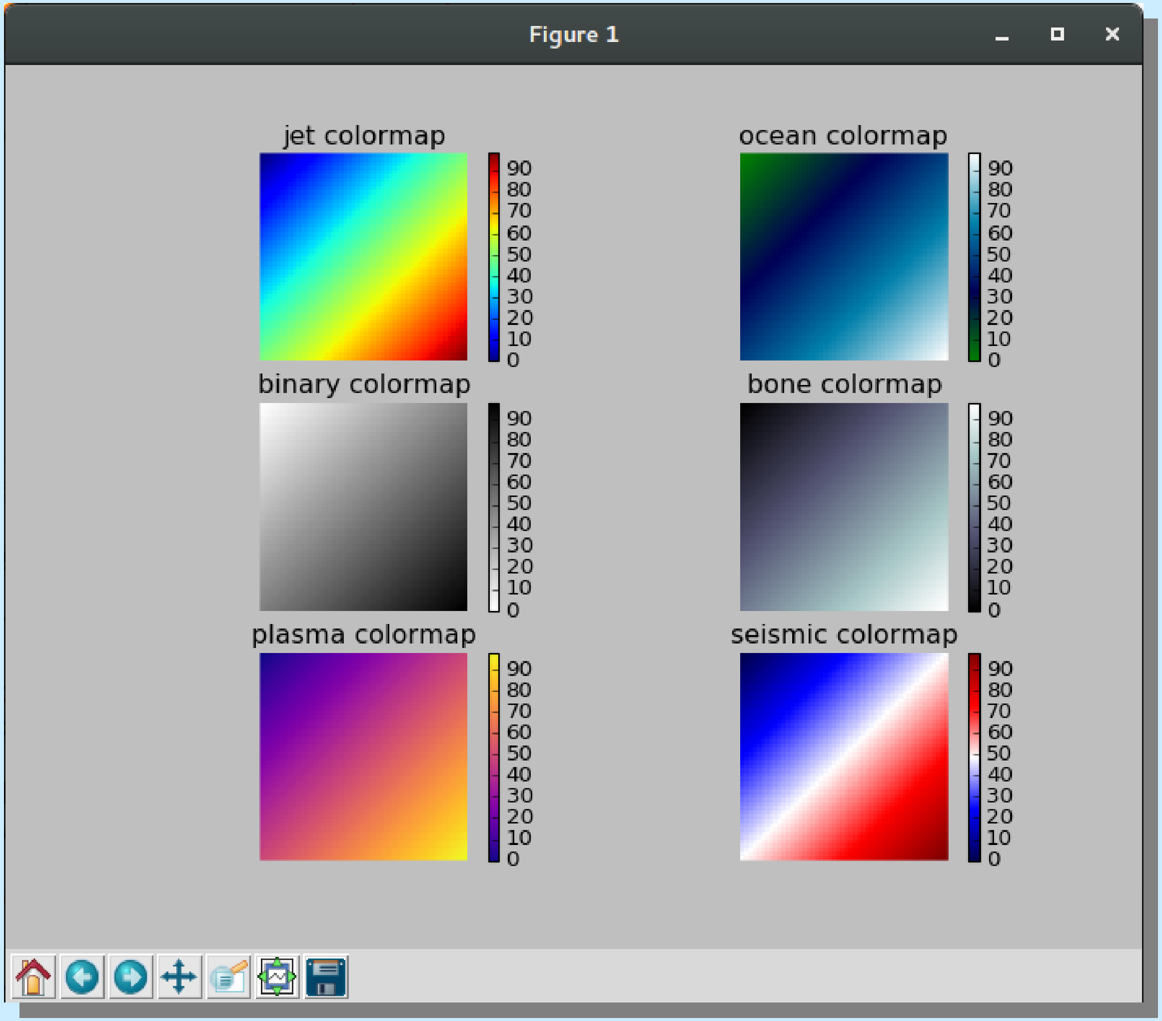


Image with Colormap

- Read source file basic-image.py on Canvas.
- Image matrix: mkimg(50, 50), where:

```
import numpy as np
import matplotlib.pyplot as plt

def mkimg(w, h):
    """Returns a gradient w x h matrix that looks like:
[[ 0  1  2  3  4  5  6  7  8  9]
 [ 1  2  3  4  5  6  7  8  9 10]
 [ 2  3  4  5  6  7  8  9 10 11]
 [ 3  4  5  6  7  8  9 10 11 12]
 [ 4  5  6  7  8  9 10 11 12 13]
 [ 5  6  7  8  9 10 11 12 13 14]
 [ 6  7  8  9 10 11 12 13 14 15]]
 """
    a = np.fromfunction(lambda i,j: i + j, (w,h), dtype=int)
    return a

image = mkimg(50, 50)
```

Image with Colormap

- Display the same image with different colormaps in a 3 x 2 grid

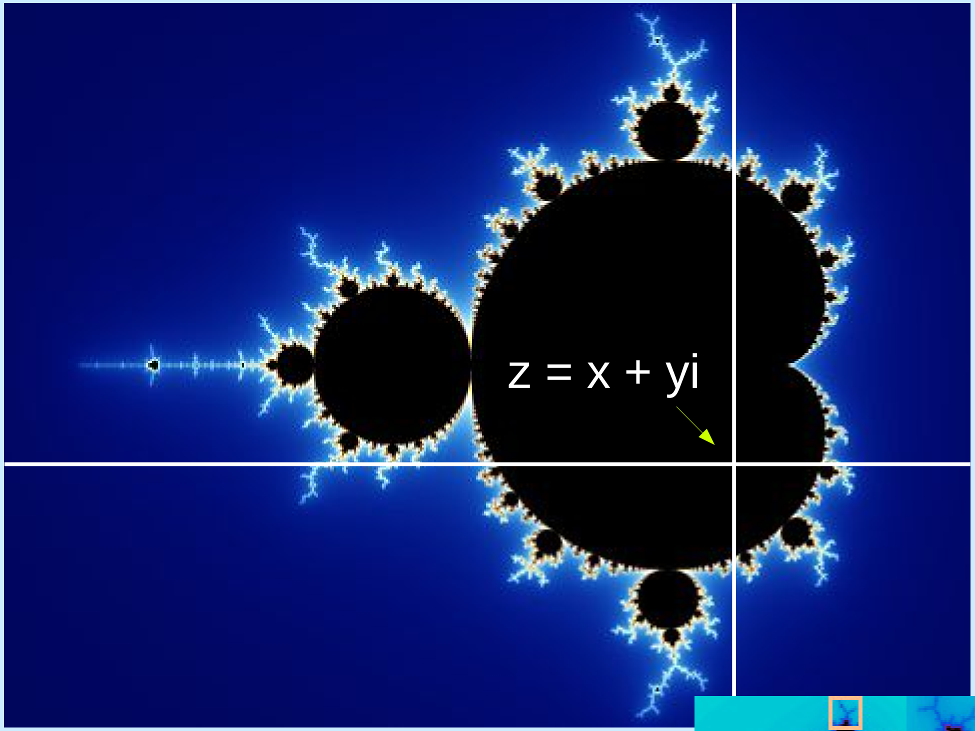
```
# colormap names
colormaps = ["jet", "ocean", "binary", "bone", "plasma", "seismic"]

ncols = 2
nrows = round(len(colormaps) / ncols + 0.4999)

# create a figure with nrows x ncols subplots.
(fig, axes) = plt.subplots(nrows=nrows, ncols=ncols)    # axes is  nrows x ncols axis ndarray

for (axis, cm) in zip(axes.flat, colormaps):
    img = axis.imshow(image, interpolation='none', cmap=plt.get_cmap(cm))    # create subplot image
    axis.set_title(cm + " colormap")
    cbar = fig.colorbar(img, ax=axis, orientation="vertical")   # vertical colorbar for subplot img, attached to axis
    axis.set_axis_off()                                         # don't show ticks or labels for this subplot

plt.show()
```

The Mandelbrot Fractal

- Reference: https://en.wikipedia.org/wiki/Mandelbrot_set
- The Mandelbrot set is the set of **complex** numbers **c** for which the function $f_c(z)=z^2+c$ does not diverge when iterated from z=0, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, $f_c(f_c(f_c(0)))$, etc., remains bounded in absolute value.
- To image the Mandelbrot set, a complex number z=x+yi corresponds to a pixel at coordinate (x,y). Scaling may be necessary.
- Why is it a fascinating topic ?
  - It has a self-similar structure (shape). Zooming in (magnifying) reveals shapes seen at a larger scale. This makes this set a fractal.

The Mandelbrot Fractal

- Complex points **in** the Mandelbrot set are colored black:



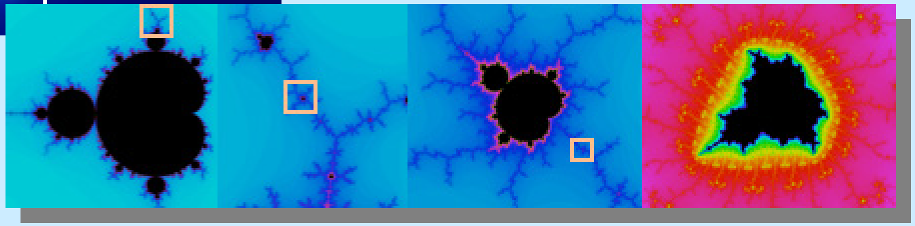
How do we get the extra colors?

The function $f_c(z)$ iteration is repeated *n* (n=20) times for **each point c** on the complex plane corresponding a pixel.

Element (i,j) of the result matrix is the number of iterations in point ***c*** needed for the function iteration to diverge ($|f_c(...)| > 2$).

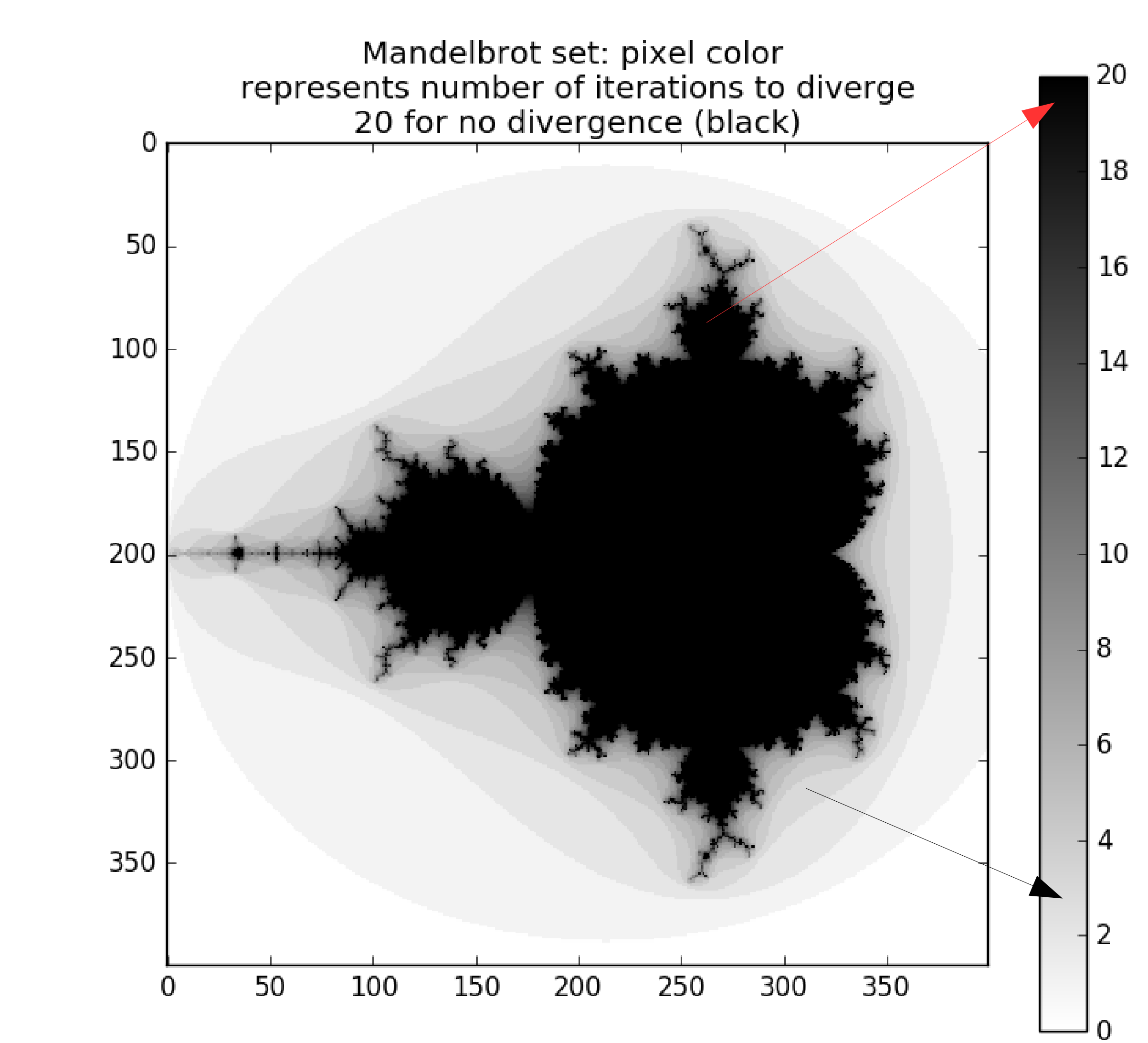
Black color pixel for no diverging.

Some quasi-self-similar features when zooming in:



The Mandelbrot Fractal

- mandelbrot.py program on Canvas produces this 400x400 figure:



Pixel color maps to the number of iterations needed for the iterative application of $f_c(.)$ to diverge.

This matrix goes up to 20 iterations .

Divergence is considered for a complex number $z=x + yi$ if $|z|>2$.

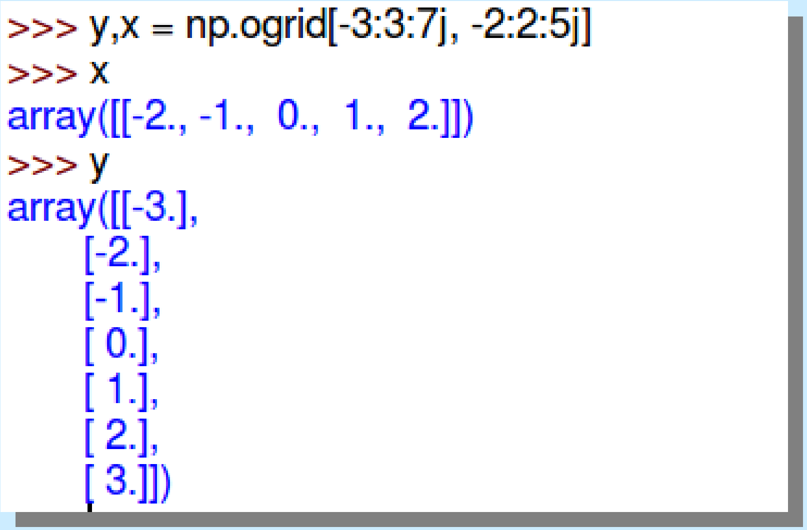
The absolute value of z is

$$|z|=\sqrt{x^2+y^2}$$

This point diverged after 3 iterations.

The Mandelbrot Fractal

- Additional NumPy utilities and functions needed so we can understand the code:
  - (y,x) = np.ogrid[-1.4:1.4:**h***1j, -2:0.8:**w***1j]: y is a h x 1 array with h equally spaced floats between -1.4 and 1.4. x is a similar 1 x w array for the horizontal grid. The grid width (w) and height (h) are passed as a complex number step param.
    - np.ogrid is not a function – it's an object that is 'sliced'.
  - e.g.



```
>>> y,x = np.ogrid[-3:3:7j, -2:2:5j]
>>> x
array([[-2., -1.,  0.,  1.,  2.]])
>>> y
array([[-3.],
       [-2.],
       [-1.],
       [ 0.],
       [ 1.],
       [ 2.],
       [ 3.]])
```

The Mandelbrot Fractal

- Additional NumPy utilities and functions needed so we can understand the code:
  - np.conj(z): returns the complex conjugates for numbers in array z
  - We know that for any complex number *w*, $w * w^* == |w|^2$
  - |w| is the absolute value of w and always $|w| \in \mathbb{R}^+ \cup \{0\}$
  - Examples:

```
>>> z = np.array([0, 4, 2-3j, 4+5j, -2j])
>>> z
array([ 0.+0.j,  4.+0.j,  2.-3.j,  4.+5.j, -0.-2.j])
>>> np.conj(z)
array([ 0.-0.j,  4.-0.j,  2.+3.j,  4.-5.j, -0.+2.j])
>>> z * np.conj(z)     # this is |z|**2
array([  0.+0.j,  16.+0.j,  13.+0.j,  41.+0.j,   4.+0.j])
```

The Mandelbrot Fractal

- The function producing the Mandelbrot matrix:

```
def mandelbrot( h,w, maxit=20 ):
    """Returns an image matrix of the Mandelbrot fractal of size (h,w).
       Element (i,j) is the number of iterations needed for repeated f_c(.)
       iteration for the matching complex number c to diverge."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]      # y is a h x 1 column, x is w x 1 row.
    c = x+y*1j            # c is a 2D grid with unifform spaced complex numbers. Broadcasting used.
    z = c                 # initial point for function iteration: z is a h x w complex matrix.

    # divtime is the return value. divtime[i,j] will be the number of iterations needed for the
    # repeated iteration of f_c(.) to diverge. Initialized with maxit to simplify the code.
    # If divtime[i,j] == maxit at the end, then f_c(.) (for corresponding c) does NOT diverge.
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
       z = z**2 + c                          # This is the repeated function iteration: z = f_c(z)

       diverge = (z * np.conj(z) > 2**2      # z*np.conj(z) is |z|**2. If |z|>2, f_c(z) will diverge.
       # the divergence test is done for ALL points c: diverge[i,j]==True if |z|>2.

       # We need to store in divtime the iteration count (i) where f_c started to diverge:
       div_now = diverge & (divtime==maxit)  # who is diverging right in this iteration
       divtime[div_now] = i                  # assignment with indexing with bool 2D array
       z[diverge] = 2                        # Limit z growth to avoid arithmetic overflow.

    return divtime
```

The Mandelbrot Fractal

- The main program:

```
side = 400             # number of pixels per image side
maxiterations = 20     # the max number of iterations to compute divergence

mandel_matrix = mandelbrot(side, side, maxiterations)

colormap = plt.get_cmap('binary')     # grayscale, where white==0 and black=max.

# create the image
fig = plt.imshow(mandel_matrix, interpolation='none', cmap=colormap)
# interpolation='none' to avoid mixing colors

cbar = plt.colorbar(fig, orientation="vertical")   # vertical colorbar for subplot img, attached to axis

title = "Mandelbrot set: pixel color \nrepresents number of iterations to diverge\n\
{} for no divergence (black)".format(maxiterations)
plt.title(title)

plt.show()      # display the image
```

Example: Animation with Image Transition

- Animate images with matplotlib.
- Example: transition back and forth smoothly between image1 and image2.
- Principle: generate periodically a new H x W x 3 array and display it as an image. Its shape is (h,w,3).
  - The image generated is a mix between image 1 and image 2:

    img = img1 * s + img2 * (1.0 – s), where s varies in [0,1]
  - When s is 0<s<1, so is img1<img<img2
- Load an image (jpg or png) into an ndarray: plt.imread(filename)
- Class *FuncAnimation* from the matplotlib.animation module takes care of the animation part. Implement a function executed periodically that generates the image array, called *updatefig.*
- Screenshots on the next slide...

Example: Animation with Image Transition

Example: Animation with Image Transition

- File animation-transition.py on Canvas.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import sys

def image_load(filename):
    return plt.imread(filename)
```

Example: Animation with Image Transition

- The image generator: pre-computes a list of images and then yields successive images from the list:

```
def image_gen(file1, file2, steps=30):
    """Generator for image arrays."""
    img1 = image_load(file1)     # load the two image files into ndarrays
    img2 = image_load(file2)
    if img1.shape != img2.shape:
        print("Error: the two images have different shapes.", file=sys.stderr)
        exit(2)

    # go from img1 to img2 than back to img1. s varies from 0 to 1 and then back to 0:
    svalues = np.hstack([np.linspace(0.0, 1.0, steps), np.linspace(1.0, 0, steps)])

    # construct now the list of images, so that we don't have to repeat that later:
    images = [np.uint8(img1 * (1.0 - s) + img2 * s) for s in svalues]

    # get a  new image as a combination of img1 and img2
    while True:             # repeat all images in a loop
        for img in images:
            yield img
```

Example: Animation with Image Transition

- Setting up the figure; create the generator *imggen*; define the timer even handler *updatefig*; create the animation object. Go.

```
fig = plt.figure()
# create image plot and indicate this is animated. Start with an image.
im = plt.imshow(image_load("florida-keys-800-480.jpg"), interpolation='none', animated=True)

# the two images must have the same shape:
imggen = image_gen("florida-keys-800-480.jpg", "Grand_Teton-800-480.jpg", steps=30)

# updatefig is called for each frame, each update interval:
def updatefig(*args):
    global imggen
    img_array = next(imggen)     # get next image animation frame
    im.set_array(img_array)      # set it. FuncAnimation will display it
    return (im,)

# create animation object that will call function updatefig every 60 ms
ani = animation.FuncAnimation(fig, updatefig, interval=60, blit=False)
plt.title("Image transformation")
plt.show()
```

SciPy Overview

- Main reference: https://docs.scipy.org/doc/scipy/reference/tutorial/
- SciPy is a set of modules with a range of useful algorithms with scientific applications based on the NumPy library.
- Has high-level functions very easy to use interactively, with Spyder, for instance:
  - Data processing and visualization.
  - Similar to Matlab, Octave, R-Lab, SciLab.
- Functions easy to integrate in Python programs, together with many other modules.

SciPy Overview

- SciPy subpackages:

Subpackage	***Description***
Cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions