CHAPTER 1

NumPy: creating and manipulating numerical data

Authors: *Emmanuelle Gouillart, Didrik Pinte, Gaël Varoquaux, and Pauli Virtanen*

This chapter gives an overview of NumPy, the core tool for performant numerical computing with Python.

1.1 The NumPy array object

Section contents

1.1.1 What are NumPy and NumPy arrays?

NumPy arrays

Python objects

- high-level number objects: integers, floating point
- containers: lists (costless insertion and append), dictionaries (fast lookup)

NumPy provides

- extension package to Python for multi-dimensional arrays
- closer to hardware (efficiency)
- designed for scientific computation (convenience)
- Also known as *array oriented computing*

```
>>> import numpy as np
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

Tip: For example, An array containing:

- values of an experiment/simulation at discrete time steps
- signal recorded by a measurement device, e.g. sound wave
- pixels of an image, grey-level or colour
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan
- ...

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
In [1]: L = range(1000)

In [2]: %timeit [i**2 for i in L]
1000 loops, best of 3: 403 us per loop

In [3]: a = np.arange(1000)

In [4]: %timeit a**2
100000 loops, best of 3: 12.7 us per loop
```

NumPy Reference documentation

- On the web: http://docs.scipy.org/
- Interactive help:

```
In [5]: np.array?
String Form:<built-in function array>
Docstring:
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0, ...
```

- Looking for something:

```
>>> np.lookfor('create array')
Search results for 'create array'
---------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
```

```
In [6]: np.con*?
np.concatenate
np.conj
np.conjugate
np.convolve
```

Import conventions

The recommended convention to import numpy is:

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

1.1.2 Creating arrays

Manual construction of arrays

- **1-D**:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

- **2-D, 3-D, . . .**:

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)     # returns the size of the first dimension
```

```
2

>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)
```

Exercise: Simple arrays

- Create a simple two dimensional array. First, redo the examples from above. And then create your own: how about odd numbers counting backwards on the first row, and even numbers on the second?
- Use the functions `len()`, `numpy.shape()` on these arrays. How do they relate to each other? And to the `ndim` attribute of the arrays?

Functions for creating arrays

Tip: In practice, we rarely enter items one by one...

- Evenly spaced:

```
>>> a = np.arange(10) # 0 .. n-1  (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- or by number of points:

```
>>> c = np.linspace(0, 1, 6)   # start, end, num-points
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

- Common arrays:

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

- `np.random`: random numbers (Mersenne Twister PRNG):

```
>>> a = np.random.rand(4)       # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])

>>> b = np.random.randn(4)      # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

>>> np.random.seed(1234)        # Setting the random seed
```

Exercise: Creating arrays using functions

- Experiment with `arange`, `linspace`, `ones`, `zeros`, `eye` and `diag`.
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function `np.empty`. What does it do? When might this be useful?

1.1.3 Basic data types

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. `2.` vs `2`). This is due to a difference in the data-type used:

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

Tip: Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

The **default** data type is floating point:

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

There are also other types:

Complex

```
>>> d = np.array([1+2j, 3+4j, 5+6*1j])
>>> d.dtype
dtype('complex128')
```

Bool

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

Strings

```
>>> f = np.array(['Bonjour', 'Hello', 'Hallo',])
>>> f.dtype     # <--- strings containing max. 7 letters
dtype('S7')
```

Much more

- `int32`
- `int64`
- `uint32`
- `uint64`

1.1.4 Basic visualization

Now that we have our first data arrays, we are going to visualize them.

Start by launching IPython:

```
$ ipython
```

Or the notebook:

```
$ ipython notebook
```

Once IPython has started, enable interactive plots:

```
>>> %matplotlib
```

Or, from the notebook, enable plots in the notebook:

```
>>> %matplotlib inline
```

The `inline` is important for the notebook, so that plots are displayed in the notebook and not in a new window.

Matplotlib is a 2D plotting package. We can import its functions as below:

```
>>> import matplotlib.pyplot as plt  # the tidy way
```

And then use (note that you have to use `show` explicitly if you have not enabled interactive plots with `%matplotlib`):

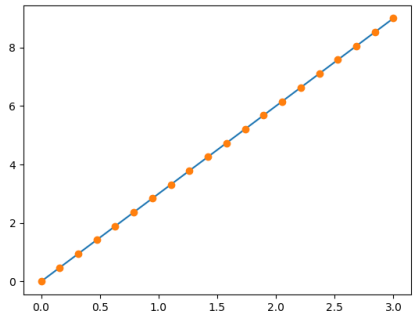
```
>>> plt.plot(x, y)       # line plot
>>> plt.show()           # <-- shows the plot (not needed with interactive plots)
```

Or, if you have enabled interactive plots with `%matplotlib`:

```
>>> plt.plot(x, y)       # line plot
```

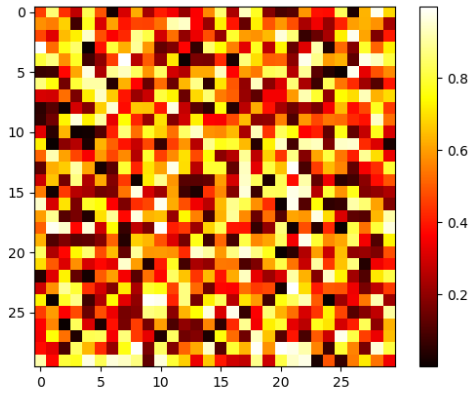
- **1D plotting**:

```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
```



- **2D arrays** (such as images):

```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
```



See also:

More in the: *matplotlib chapter*

Exercise: Simple visualizations

- Plot some simple arrays: a cosine as a function of time and a 2D matrix.
- Try using the `gray` colormap on the 2D matrix.

1.1.5 Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

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

Warning: Indices begin at 0, like other Python sequences (and C/C++). In contrast, in Fortran or Matlab, indices begin at 1.

The usual python idiom for reversing a sequence is supported:

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

For multidimensional arrays, indexes are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
```

```
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

Note:

- In 2D, the first dimension corresponds to **rows**, the second to **columns**.
- for multidimensional `a`, `a[0]` is interpreted by taking all elements in the unspecified dimensions.

Slicing: Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

Note that the last index is not included! :

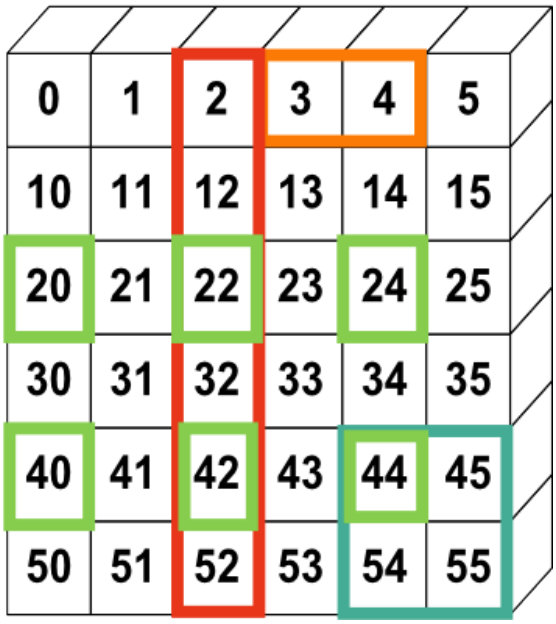
```
>>> a[:4]
array([0, 1, 2, 3])
```

All three slice components are not required: by default, *start* is 0, *end* is the last and *step* is 1:

```
>>> a[1:3]
array([1, 2])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
```

A small illustrated summary of NumPy indexing and slicing...

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



You can also combine assignment and slicing:

```
>>> a = np.arange(10)
>>> a[5:] = 10
>>> a
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
>>> b = np.arange(5)
>>> a[5:] = b[::-1]
>>> a
array([0, 1, 2, 3, 4, 4, 3, 2, 1, 0])
```

Exercise: Indexing and slicing

- Try the different flavours of slicing, using `start`, `end` and `step`: starting from a linspace, try to obtain odd numbers counting backwards, and even numbers counting forwards.
- Reproduce the slices in the diagram above. You may use the following expression to create the array:

```
>>> np.arange(6) + np.arange(0, 51, 10)[:, np.newaxis]
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

Exercise: Array creation

Create the following arrays (with correct data types):

```
[[1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 6, 1, 1]]

[[0., 0., 0., 0., 0.],
 [2., 0., 0., 0., 0.],
 [0., 3., 0., 0., 0.],
 [0., 0., 4., 0., 0.],
 [0., 0., 0., 5., 0.],
 [0., 0., 0., 0., 6.]]
```

Par on course: 3 statements for each

Hint: Individual array elements can be accessed similarly to a list, e.g. `a[1]` or `a[1, 2]`.

Hint: Examine the docstring for `diag`.

Exercise: Tiling for array creation

Skim through the documentation for `np.tile`, and use this function to construct the array:

```
[[4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1],
 [4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1]]
```

1.1.6 Copies and views

A slicing operation creates a **view** on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use `np.may_share_memory()` to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives.

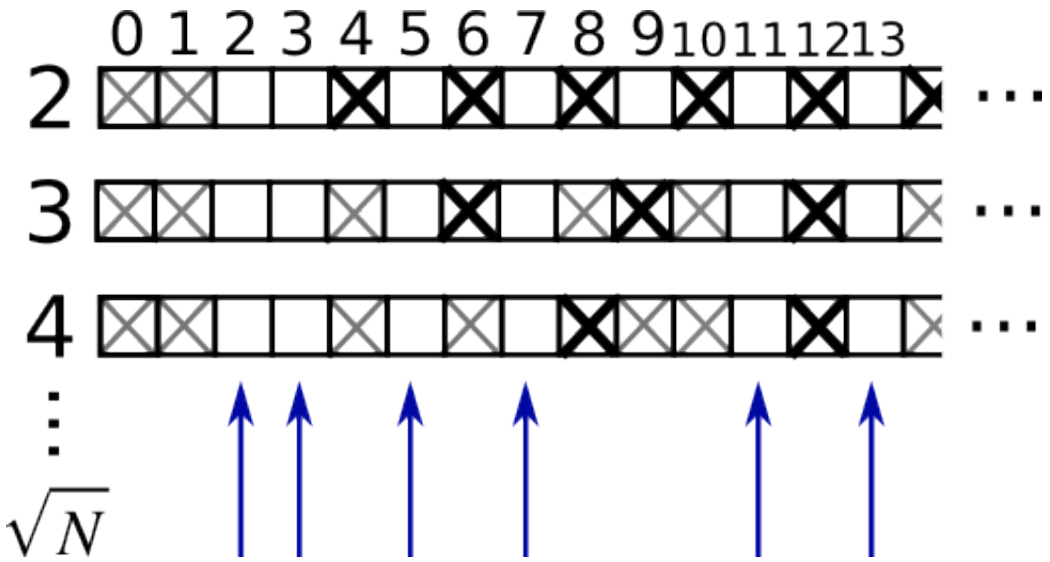
When modifying the view, the original array is modified as well:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
>>> np.may_share_memory(a, b)
True
>>> b[0] = 12
>>> b
array([12,  2,  4,  6,  8])
>>> a   # (!)
array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])

>>> a = np.arange(10)
>>> c = a[::2].copy()  # force a copy
>>> c[0] = 12
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.may_share_memory(a, c)
False
```

This behavior can be surprising at first sight... but it allows to save both memory and time.

Worked example: Prime number sieve



Compute prime numbers in 0–99, with a sieve

- Construct a shape (100,) boolean array `is_prime`, filled with True in the beginning:

```
>>> is_prime = np.ones((100,), dtype=bool)
```

- Cross out 0 and 1 which are not primes:

```
>>> is_prime[:2] = 0
```

- For each integer `j` starting from 2, cross out its higher multiples:

```
>>> N_max = int(np.sqrt(len(is_prime) - 1))
>>> for j in range(2, N_max + 1):
...     is_prime[2*j::j] = False
```

- Skim through `help(np.nonzero)`, and print the prime numbers
- Follow-up:
  - Move the above code into a script file named `prime_sieve.py`
  - Run it to check it works
  - Use the optimization suggested in the sieve of Eratosthenes:
  1. Skip `j` which are already known to not be primes
  2. The first number to cross out is j^2

1.1.7 Fancy indexing

Tip: NumPy arrays can be indexed with slices, but also with boolean or integer arrays (**masks**). This method is called *fancy indexing*. It creates **copies not views**.

Using boolean masks

```
>>> np.random.seed(3)
>>> a = np.random.randint(0, 21, 15)
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,  0, 20, 12,  7, 14])
>>> (a % 3 == 0)
array([False,  True, False,  True, False, False, False,  True, False,
        True,  True, False,  True, False, False], dtype=bool)
>>> mask = (a % 3 == 0)
>>> extract_from_a = a[mask] # or,  a[a%3==0]
>>> extract_from_a           # extract a sub-array with the mask
array([ 3,  0,  9,  6,  0, 12])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
>>> a[a % 3 == 0] = -1
>>> a
array([10, -1,  8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1,  7, 14])
```

Indexing with an array of integers

```
>>> a = np.arange(0, 100, 10)
>>> a
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Indexing can be done with an array of integers, where the same index is repeated several time:

```
>>> a[[2, 3, 2, 4, 2]]  # note: [2, 3, 2, 4, 2] is a Python list
array([20, 30, 20, 40, 20])
```

New values can be assigned with this kind of indexing:

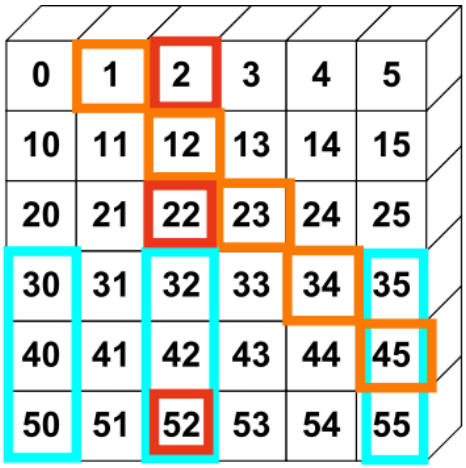
```
>>> a[[9, 7]] = -100
>>> a
array([   0,   10,   20,   30,   40,   50,   60, -100,   80, -100])
```

Tip: When a new array is created by indexing with an array of integers, the new array has the same shape as the array of integers:

```
>>> a = np.arange(10)
>>> idx = np.array([[3, 4], [9, 7]])
>>> idx.shape
(2, 2)
>>> a[idx]
array([[3, 4],
       [9, 7]])
```

The image below illustrates various fancy indexing applications

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Exercise: Fancy indexing

- Again, reproduce the fancy indexing shown in the diagram above.
- Use fancy indexing on the left and array creation on the right to assign values into an array, for instance by setting parts of the array in the diagram above to zero.

1.2 Numerical operations on arrays

Section contents

1.2.1 Elementwise operations

Basic operations

With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
```

All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])

>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2,  3,  6, 13, 28])
```

These operations are of course much faster than if you did them in pure python:

```
>>> a = np.arange(10000)
>>> %timeit a + 1
10000 loops, best of 3: 24.3 us per loop
>>> l = range(10000)
>>> %timeit [i+1 for i in l]
1000 loops, best of 3: 861 us per loop
```

Warning: Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c                   # NOT matrix multiplication!
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Note: Matrix multiplication:

```
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Exercise: Elementwise operations

- Try simple arithmetic elementwise operations: add even elements with odd elements
- Time them against their pure python counterparts using `%timeit`.
- Generate:
  - `[2**0, 2**1, 2**2, 2**3, 2**4]`
  - `a_j = 2^(3*j) - j`

Other operations

Comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
```

Tip: Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

Transcendental functions:

```
>>> a = np.arange(5)
>>> np.sin(a)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
>>> np.log(a)
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])
>>> np.exp(a)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

Shape mismatches

```
>>> a = np.arange(4)
>>> a + np.array([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4) (2)
```

Broadcasting? We'll return to that *later*.

Transposition:

```
>>> a = np.triu(np.ones((3, 3)), 1)   # see help(np.triu)
>>> a
array([[ 0.,  1.,  1.],
       [ 0.,  0.,  1.],
```

```
       [ 0.,  0.,  0.]])
>>> a.T
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  1.,  0.]])
```

Warning: The transposition is a view

As a results, the following code **is wrong** and will **not make a matrix symmetric**:

```
>>> a += a.T
```

It will work for small arrays (because of buffering) but fail for large one, in unpredictable ways.

Note: Linear algebra

The sub-module `numpy.linalg` implements basic linear algebra, such as solving linear systems, singular value decomposition, etc. However, it is not guaranteed to be compiled using efficient routines, and thus we recommend the use of `scipy.linalg`, as detailed in section *Linear algebra operations: scipy.linalg*

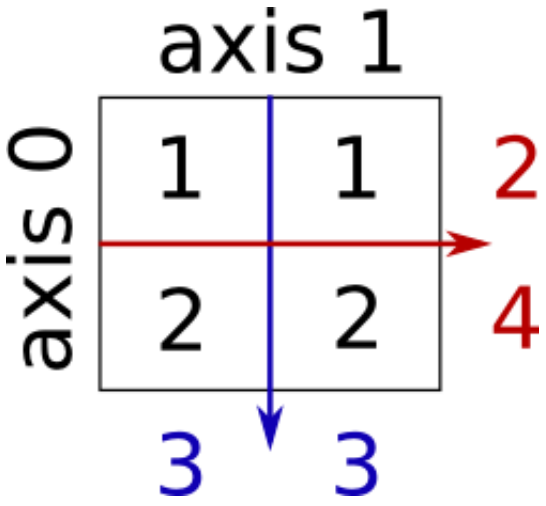
Exercise other operations

- Look at the help for `np.allclose`. When might this be useful?
- Look at the help for `np.triu` and `np.tril`.

1.2.2 Basic reductions

Computing sums

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```



Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0)   # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1)   # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```

Tip: Same idea in higher dimensions:

```
>>> x = np.random.rand(2, 2, 2)
>>> x.sum(axis=2)[0, 1]
1.14764...
>>> x[0, 1, :].sum()
1.14764...
```

Other reductions

— works the same way (and take `axis=`)

Extrema:

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3

>>> x.argmin()  # index of minimum
0
```

```
>>> x.argmax()  # index of maximum
1
```

Logical operations:

```
>>> np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

Note: Can be used for array comparisons:

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True

>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

Statistics:

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2.,  5.])

>>> x.std()          # full population standard dev.
0.82915619758884995
```

... and many more (best to learn as you go).

Exercise: Reductions

- Given there is a `sum`, what other function might you expect to see?
- What is the difference between `sum` and `cumsum`?

Worked Example: data statistics

Data in `populations.txt` describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

You can view the data in an editor, or alternatively in IPython (both shell and notebook):

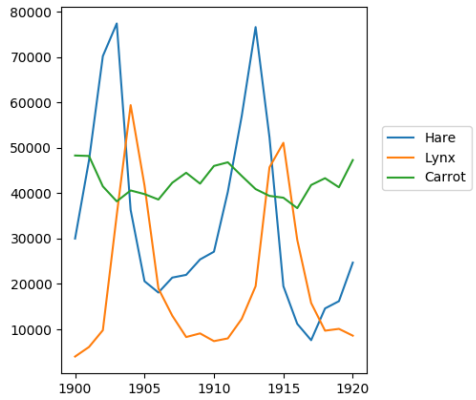
```
In [1]: !cat data/populations.txt
```

First, load the data into a NumPy array:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables
```

Then plot it:

```
>>> from matplotlib import pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
>>> plt.plot(year, hares, year, lynxes, year, carrots)
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
```



The mean populations over time:

```
>>> populations = data[:, 1:]
>>> populations.mean(axis=0)
array([ 34080.95238095,  20166.66666667,  42400.        ])
```

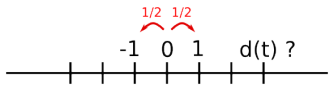
The sample standard deviations:

```
>>> populations.std(axis=0)
array([ 20897.90645809,  16254.59153691,   3322.50622558])
```

Which species has the highest population each year?:

```
>>> np.argmax(populations, axis=1)
array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2])
```

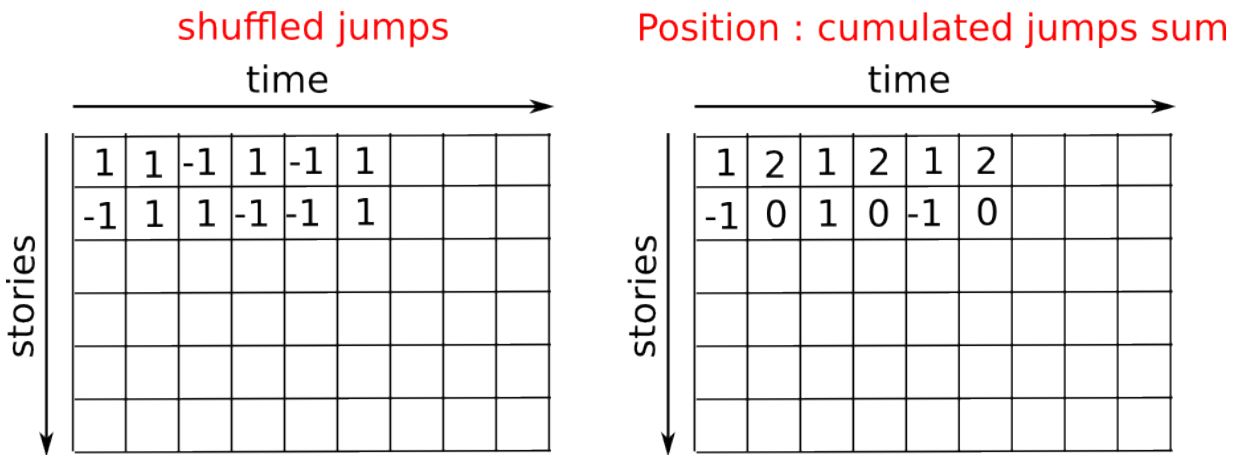
Worked Example: diffusion using a random walk algorithm



Tip: Let us consider a simple 1D random walk process: at each time step a walker jumps right or left with equal probability.

We are interested in finding the typical distance from the origin of a random walker after `t` left or right jumps? We are going to simulate many "walkers" to find this law, and we are going to do so using array computing tricks: we are going to create a 2D array with the "stories" (each walker has a story) in one direction, and the time in the

other:




```
>>> n_stories = 1000 # number of walkers
>>> t_max = 200      # time during which we follow the walker
```

We randomly choose all the steps 1 or -1 of the walk:

```
>>> t = np.arange(t_max)
>>> steps = 2 * np.random.randint(0, 1 + 1, (n_stories, t_max)) - 1 # +1 because the high value␣
↪is exclusive
>>> np.unique(steps) # Verification: all steps are 1 or -1
array([-1,  1])
```

We build the walks by summing steps along the time:

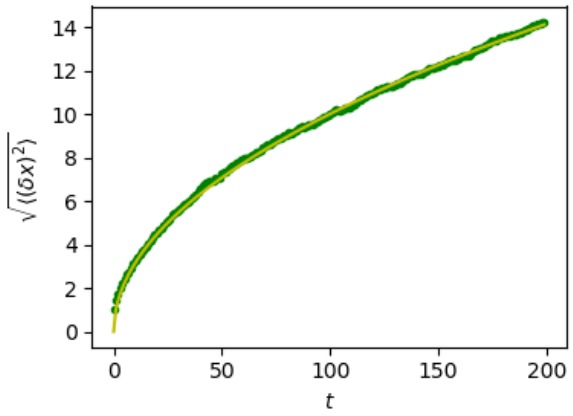
```
>>> positions = np.cumsum(steps, axis=1) # axis = 1: dimension of time
>>> sq_distance = positions**2
```

We get the mean in the axis of the stories:

```
>>> mean_sq_distance = np.mean(sq_distance, axis=0)
```

Plot the results:

```
>>> plt.figure(figsize=(4, 3))
<matplotlib.figure.Figure object at ...>
>>> plt.plot(t, np.sqrt(mean_sq_distance), 'g.', t, np.sqrt(t), 'y-')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
>>> plt.xlabel(r"$t$")
<matplotlib.text.Text object at ...>
>>> plt.ylabel(r"$\sqrt{\langle (\delta x)^2 \rangle}$")
<matplotlib.text.Text object at ...>
>>> plt.tight_layout() # provide sufficient space for labels
```



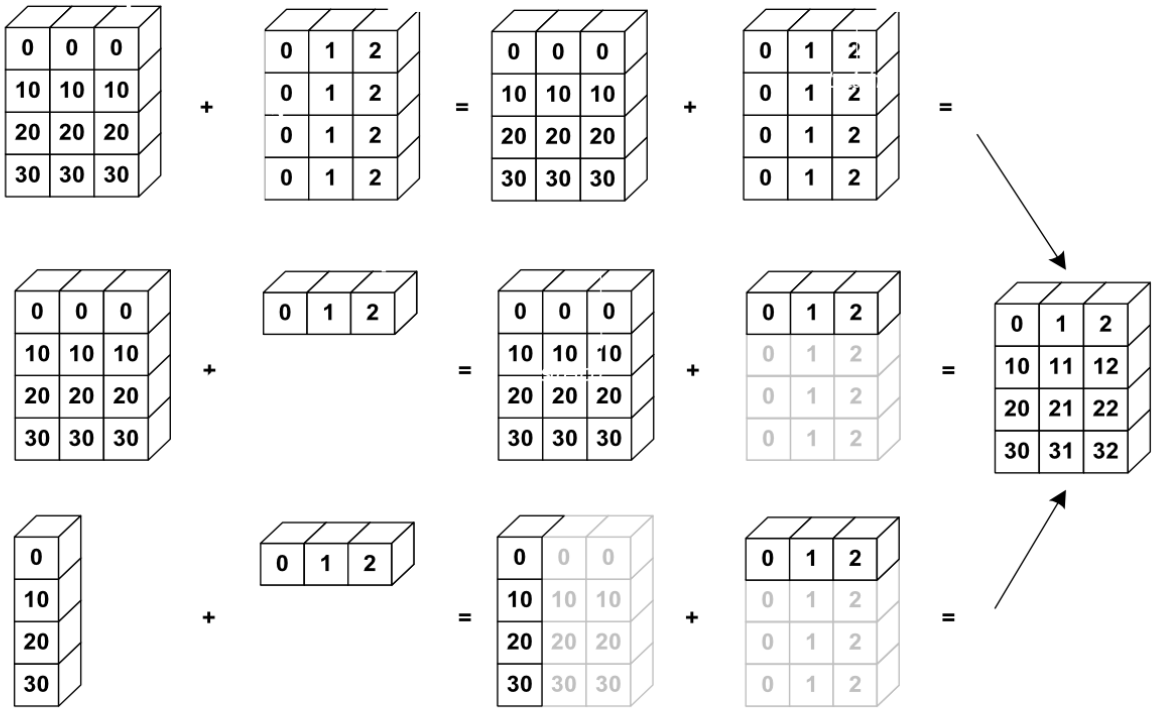
We find a well-known result in physics: the RMS distance grows as the square root of the time!

1.2.3 Broadcasting

- Basic operations on `numpy` arrays (addition, etc.) are elementwise
- This works on arrays of the same size.

  Nevertheless, It's also possible to do operations on arrays of different sizes if *NumPy* can transform these arrays so that they all have the same size: this conversion is called **broadcasting**.

The image below gives an example of broadcasting:



Let's verify:

```
>>> a = np.tile(np.arange(0, 40, 10), (3, 1)).T
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
>>> b = np.array([0, 1, 2])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

We have already used broadcasting without knowing it!:

```
>>> a = np.ones((4, 5))
>>> a[0] = 2  # we assign an array of dimension 0 to an array of dimension 1
>>> a
array([[ 2.,  2.,  2.,  2.,  2.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

An useful trick:

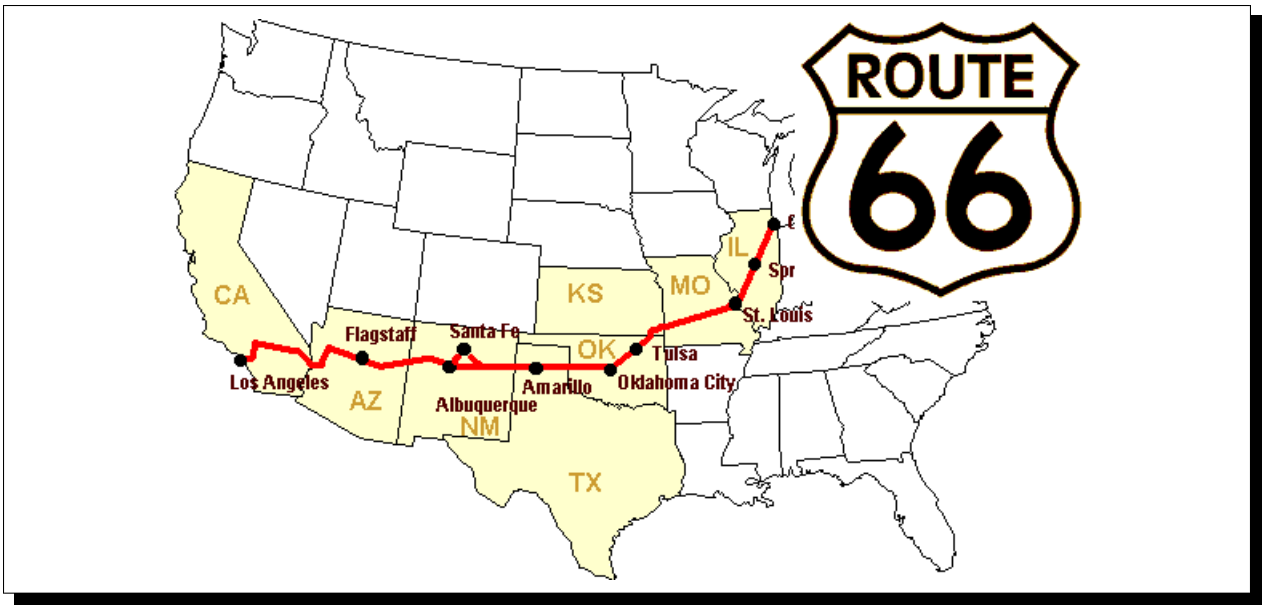
```
>>> a = np.arange(0, 40, 10)
>>> a.shape
(4,)
>>> a = a[:, np.newaxis]  # adds a new axis -> 2D array
>>> a.shape
(4, 1)
>>> a
array([[ 0],
       [10],
       [20],
       [30]])
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Tip: Broadcasting seems a bit magical, but it is actually quite natural to use it when we want to solve a problem whose output data is an array with more dimensions than input data.

Worked Example: Broadcasting

Let's construct an array of distances (in miles) between cities of Route 66: Chicago, Springfield, Saint-Louis, Tulsa, Oklahoma City, Amarillo, Santa Fe, Albuquerque, Flagstaff and Los Angeles.

```
>>> mileposts = np.array([0, 198, 303, 736, 871, 1175, 1475, 1544,
...        1913, 2448])
>>> distance_array = np.abs(mileposts - mileposts[:, np.newaxis])
>>> distance_array
array([[   0,  198,  303,  736,  871, 1175, 1475, 1544, 1913, 2448],
       [ 198,    0,  105,  538,  673,  977, 1277, 1346, 1715, 2250],
       [ 303,  105,    0,  433,  568,  872, 1172, 1241, 1610, 2145],
       [ 736,  538,  433,    0,  135,  439,  739,  808, 1177, 1712],
       [ 871,  673,  568,  135,    0,  304,  604,  673, 1042, 1577],
       [1175,  977,  872,  439,  304,    0,  300,  369,  738, 1273],
       [1475, 1277, 1172,  739,  604,  300,    0,   69,  438,  973],
       [1544, 1346, 1241,  808,  673,  369,   69,    0,  369,  904],
       [1913, 1715, 1610, 1177, 1042,  738,  438,  369,    0,  535],
       [2448, 2250, 2145, 1712, 1577, 1273,  973,  904,  535,    0]])
```

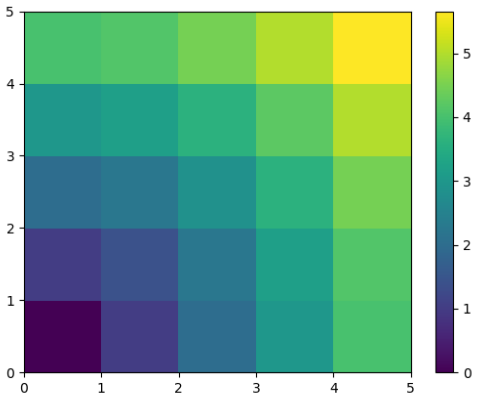


A lot of grid-based or network-based problems can also use broadcasting. For instance, if we want to compute the distance from the origin of points on a 10x10 grid, we can do

```
>>> x, y = np.arange(5), np.arange(5)[:, np.newaxis]
>>> distance = np.sqrt(x ** 2 + y ** 2)
>>> distance
array([[ 0.        ,  1.        ,  2.        ,  3.        ,  4.        ],
       [ 1.        ,  1.41421356,  2.23606798,  3.16227766,  4.12310563],
       [ 2.        ,  2.23606798,  2.82842712,  3.60555128,  4.47213595],
       [ 3.        ,  3.16227766,  3.60555128,  4.24264069,  5.        ],
       [ 4.        ,  4.12310563,  4.47213595,  5.        ,  5.65685425]])
```

Or in color:

```
>>> plt.pcolor(distance)
>>> plt.colorbar()
```



Remark : the `numpy.ogrid()` function allows to directly create vectors x and y of the previous example, with two "significant dimensions":

```
>>> x, y = np.ogrid[0:5, 0:5]
>>> x, y
```

```
(array([[0],
       [1],
       [2],
       [3],
       [4]]), array([[0, 1, 2, 3, 4]]))
>>> x.shape, y.shape
((5, 1), (1, 5))
>>> distance = np.sqrt(x ** 2 + y ** 2)
```

Tip: So, `np.ogrid` is very useful as soon as we have to handle computations on a grid. On the other hand, `np.mgrid` directly provides matrices full of indices for cases where we can't (or don't want to) benefit from broadcasting:

```
>>> x, y = np.mgrid[0:4, 0:4]
>>> x
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])
>>> y
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])
```

See also:

Broadcasting: discussion of broadcasting in the *Advanced NumPy* chapter.

1.2.4 Array shape manipulation

Flattening

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

Higher dimensions: last dimensions ravel out "first".

Reshaping

The inverse operation to flattening:

```
>>> a.shape
(2, 3)
>>> b = a.ravel()
>>> b = b.reshape((2, 3))
```

```
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
```

Or,

```
>>> a.reshape((2, -1))     # unspecified (-1) value is inferred
array([[1, 2, 3],
       [4, 5, 6]])
```

Warning: `ndarray.reshape` **may** return a view (cf `help(np.reshape)`), or copy

Tip:

```
>>> b[0, 0] = 99
>>> a
array([[99,  2,  3],
       [ 4,  5,  6]])
```

Beware: reshape may also return a copy!:

```
>>> a = np.zeros((3, 2))
>>> b = a.T.reshape(3*2)
>>> b[0] = 9
>>> a
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

To understand this you need to learn more about the memory layout of a numpy array.

Adding a dimension

Indexing with the `np.newaxis` object allows us to add an axis to an array (you have seen this already above in the broadcasting section):

```
>>> z = np.array([1, 2, 3])
>>> z
array([1, 2, 3])

>>> z[:, np.newaxis]
array([[1],
       [2],
       [3]])

>>> z[np.newaxis, :]
array([[1, 2, 3]])
```

Dimension shuffling

```
>>> a = np.arange(4*3*2).reshape(4, 3, 2)
>>> a.shape
(4, 3, 2)
>>> a[0, 2, 1]
5
>>> b = a.transpose(1, 2, 0)
>>> b.shape
(3, 2, 4)
>>> b[2, 1, 0]
5
```

Also creates a view:

```
>>> b[2, 1, 0] = -1
>>> a[0, 2, 1]
-1
```

Resizing

Size of an array can be changed with `ndarray.resize`:

```
>>> a = np.arange(4)
>>> a.resize((8,))
>>> a
array([0, 1, 2, 3, 0, 0, 0, 0])
```

However, it must not be referred to somewhere else:

```
>>> b = a
>>> a.resize((4,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that has been referenced or is
referencing another array in this way.  Use the resize function
```

Exercise: Shape manipulations

- Look at the docstring for `reshape`, especially the notes section which has some more information about copies and views.
- Use `flatten` as an alternative to `ravel`. What is the difference? (Hint: check which one returns a view and which a copy)
- Experiment with `transpose` for dimension shuffling.

1.2.5 Sorting data

Sorting along an axis:

```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = np.sort(a, axis=1)
```

```
>>> b
array([[3, 4, 5],
       [1, 1, 2]])
```

Note: Sorts each row separately!

In-place sort:

```
>>> a.sort(axis=1)
>>> a
array([[3, 4, 5],
       [1, 1, 2]])
```

Sorting with fancy indexing:

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a)
>>> j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```

Finding minima and maxima:

```
>>> a = np.array([4, 3, 1, 2])
>>> j_max = np.argmax(a)
>>> j_min = np.argmin(a)
>>> j_max, j_min
(0, 2)
```

Exercise: Sorting

- Try both in-place and out-of-place sorting.
- Try creating arrays with different dtypes and sorting them.
- Use `all` or `array_equal` to check the results.
- Look at `np.random.shuffle` for a way to create sortable input quicker.
- Combine `ravel`, `sort` and `reshape`.
- Look at the `axis` keyword for `sort` and rewrite the previous exercise.

1.2.6 Summary

What do you need to know to get started?

- Know how to create arrays : `array`, `arange`, `ones`, `zeros`.
- Know the shape of the array with `array.shape`, then use slicing to obtain different views of the array: `array[::2]`, etc. Adjust the shape of the array using `reshape` or flatten it with `ravel`.
- Obtain a subset of the elements of an array and/or modify their values with masks

```
>>> a[a < 0] = 0
```

- Know miscellaneous operations on arrays, such as finding the mean or max (`array.max()`, `array.mean()`). No need to retain everything, but have the reflex to search in the documentation (online docs, `help()`, `lookfor()`)!!
- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more NumPy functions to handle various array operations.

Quick read

If you want to do a first quick pass through the Scipy lectures to learn the ecosystem, you can directly skip to the next chapter: *Matplotlib: plotting*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter, as well as to do some more *exercices*.

1.2.7 `maskedarray`: dealing with (propagation of) missing data

- For floats one could use NaN's, but masks work for all types:

```
>>> x = np.ma.array([1, 2, 3, 4], mask=[0, 1, 0, 1])
>>> x
masked_array(data = [1 -- 3 --],
             mask = [False  True False  True],
       fill_value = 999999)


>>> y = np.ma.array([1, 2, 3, 4], mask=[0, 1, 1, 1])
>>> x + y
masked_array(data = [2 -- -- --],
             mask = [False  True  True  True],
       fill_value = 999999)
```

- Masking versions of common functions:

```
>>> np.ma.sqrt([1, -1, 2, -2])
masked_array(data = [1.0 -- 1.41421356237... --],
             mask = [False  True False  True],
       fill_value = 1e+20)
```

Note: There are other useful *array siblings*

While it is off topic in a chapter on numpy, let's take a moment to recall good coding practice, which really do pay off in the long run:

Good practices

- Explicit variable names (no need of a comment to explain what is in the variable)

- Style: spaces after commas, around =, etc.

  A certain number of rules for writing "beautiful" code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).

- Except some rare cases, variable names and comments in English.

1.2.8 Loading data files

Text files

Example: `populations.txt`:

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
1902    70.2e3  9.8e3   41500
1903    77.4e3  35.2e3  38200
```

```
>>> data = np.loadtxt('data/populations.txt')
>>> data
array([[  1900.,  30000.,   4000.,  48300.],
       [  1901.,  47200.,   6100.,  48200.],
       [  1902.,  70200.,   9800.,  41500.],
...
```

```
>>> np.savetxt('pop2.txt', data)
>>> data2 = np.loadtxt('pop2.txt')
```

Note: If you have a complicated text file, what you can try are:

- `np.genfromtxt`
- Using Python's I/O functions and e.g. regexps for parsing (Python is quite well suited for this)

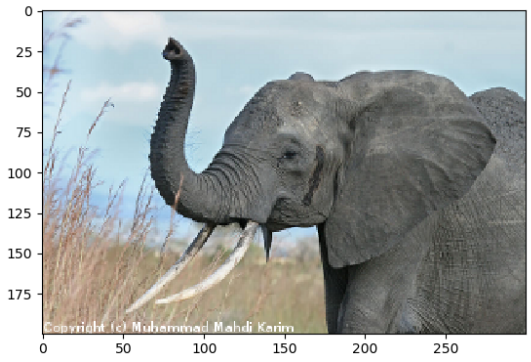
Reminder: Navigating the filesystem with IPython

```
In [1]: pwd      # show current directory
'/home/user/stuff/2011-numpy-tutorial'
In [2]: cd ex
'/home/user/stuff/2011-numpy-tutorial/ex'
In [3]: ls
populations.txt  species.txt
```

Images

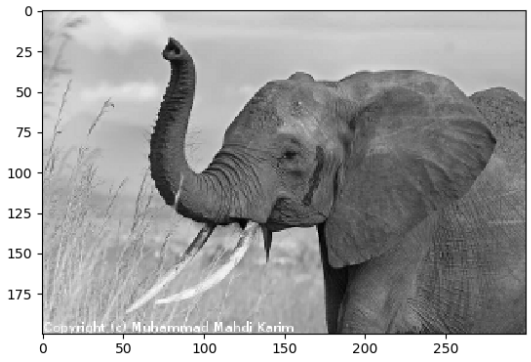
Using Matplotlib:

```
>>> img = plt.imread('data/elephant.png')
>>> img.shape, img.dtype
((200, 300, 3), dtype('float32'))
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at ...>
>>> plt.savefig('plot.png')

>>> plt.imsave('red_elephant.png', img[:,:,0], cmap=plt.cm.gray)
```



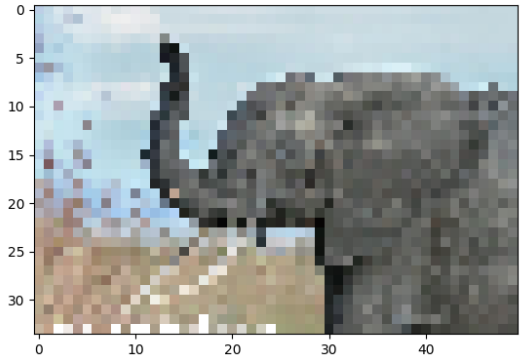
This saved only one channel (of RGB):

```
>>> plt.imshow(plt.imread('red_elephant.png'))
<matplotlib.image.AxesImage object at ...>
```



Other libraries:

```
>>> from scipy.misc import imsave
>>> imsave('tiny_elephant.png', img[::6,::6])
>>> plt.imshow(plt.imread('tiny_elephant.png'), interpolation='nearest')
<matplotlib.image.AxesImage object at ...>
```



NumPy's own format

NumPy has its own binary format, not portable but with efficient I/O:

```
>>> data = np.ones((3, 3))
>>> np.save('pop.npy', data)
>>> data3 = np.load('pop.npy')
```

1.3 Some exercises

1.3.1 Array manipulations

1. Form the 2-D array (without typing it in explicitly):

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

   and generate a new array containing its 2nd and 4th rows.

2. Divide each column of the array:

   ```
   >>> import numpy as np
   >>> a = np.arange(25).reshape(5, 5)
   ```

   elementwise with the array `b = np.array([1., 5, 10, 15, 20])`. (Hint: `np.newaxis`).

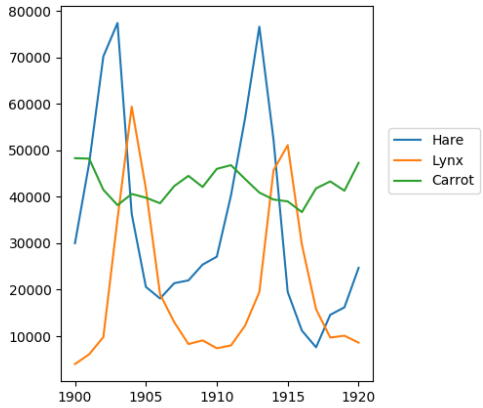
3. Harder one: Generate a 10 x 3 array of random numbers (in range [0,1]). For each row, pick the number closest to 0.5.

   - Use `abs` and `argsort` to find the column `j` closest for each row.
   - Use fancy indexing to extract the numbers. (Hint: `a[i,j]` – the array `i` must contain the row numbers corresponding to stuff in `j`.)

1.3.2 Data statistics

The data in `populations.txt` describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T  # trick: columns to variables

>>> import matplotlib.pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
<matplotlib.axes...Axes object at ...>
>>> plt.plot(year, hares, year, lynxes, year, carrots)
[<matplotlib.lines.Line2D object at ...>, ...]
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
<matplotlib.legend.Legend object at ...>
```



Computes and print, based on the data in `populations.txt`...

1. The mean and std of the populations of each species for the years in the period.
2. Which year each species had the largest population.
3. Which species has the largest population for each year. (Hint: `argsort` & fancy indexing of `np.array(['H', 'L', 'C'])`)
4. Which years any of the populations is above 50000. (Hint: comparisons and `np.any`)
5. The top 2 years for each species when they had the lowest populations. (Hint: `argsort`, fancy indexing)
6. Compare (plot) the change in hare population (see `help(np.gradient)`) and the number of lynxes. Check correlation (see `help(np.corrcoef)`).

... all without for-loops.

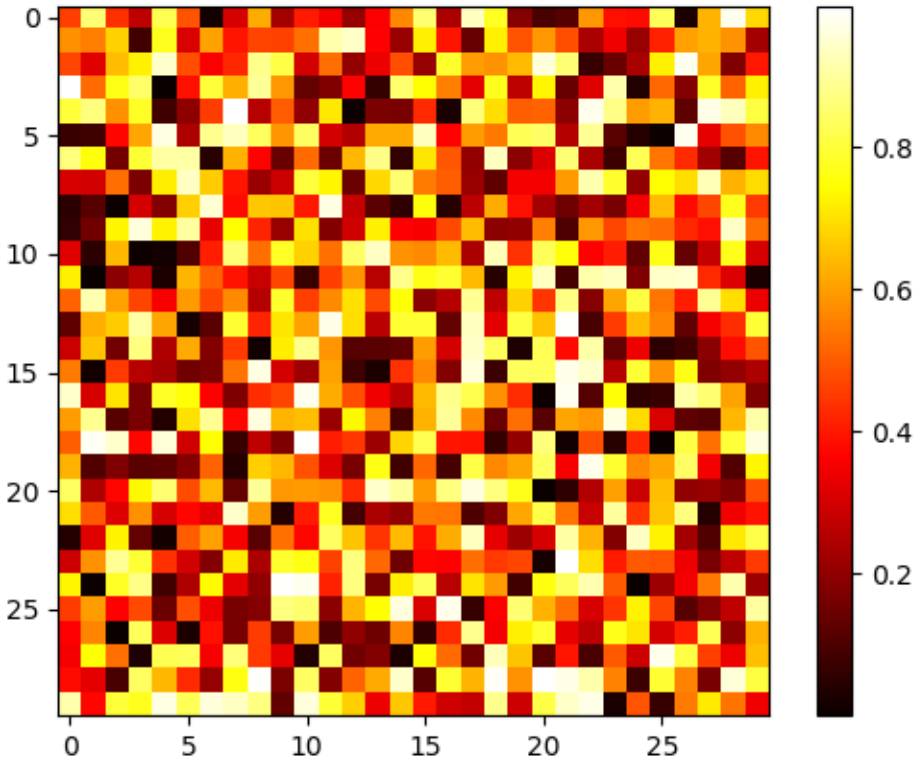
Solution: `Python source file`

1.4 Full code examples

1.4.1 Full code examples for the numpy chapter

2D plotting

Plot a basic 2D figure

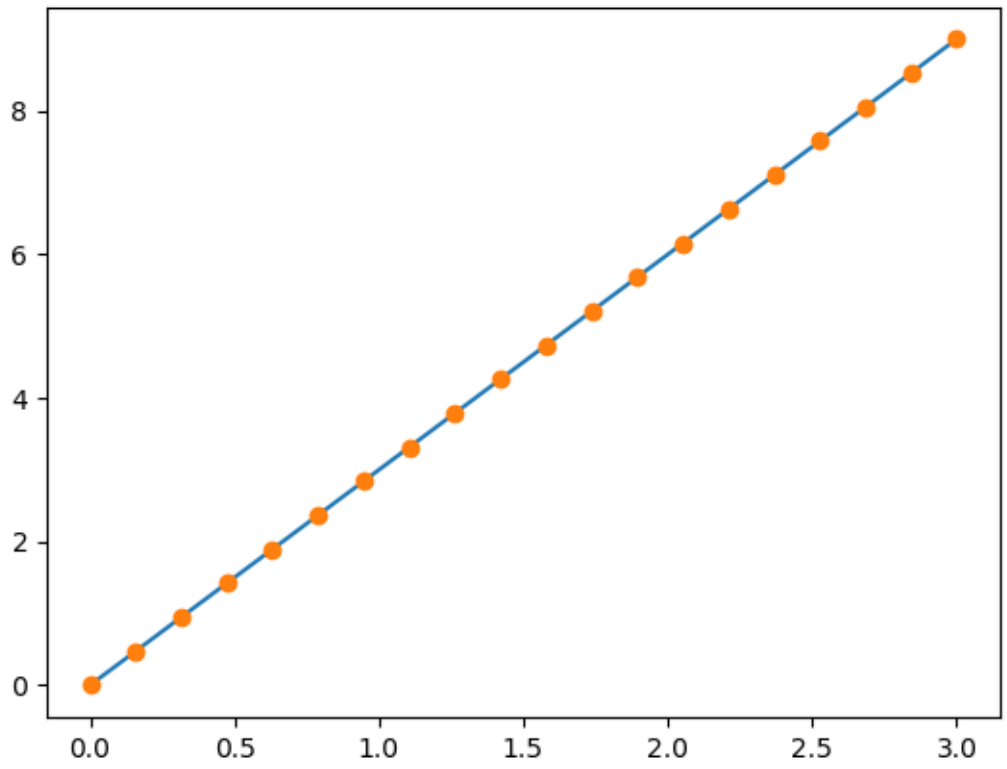



```
import numpy as np
import matplotlib.pyplot as plt

image = np.random.rand(30, 30)
plt.imshow(image, cmap=plt.cm.hot)
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.110 seconds)

1D plotting

Plot a basic 1D figure

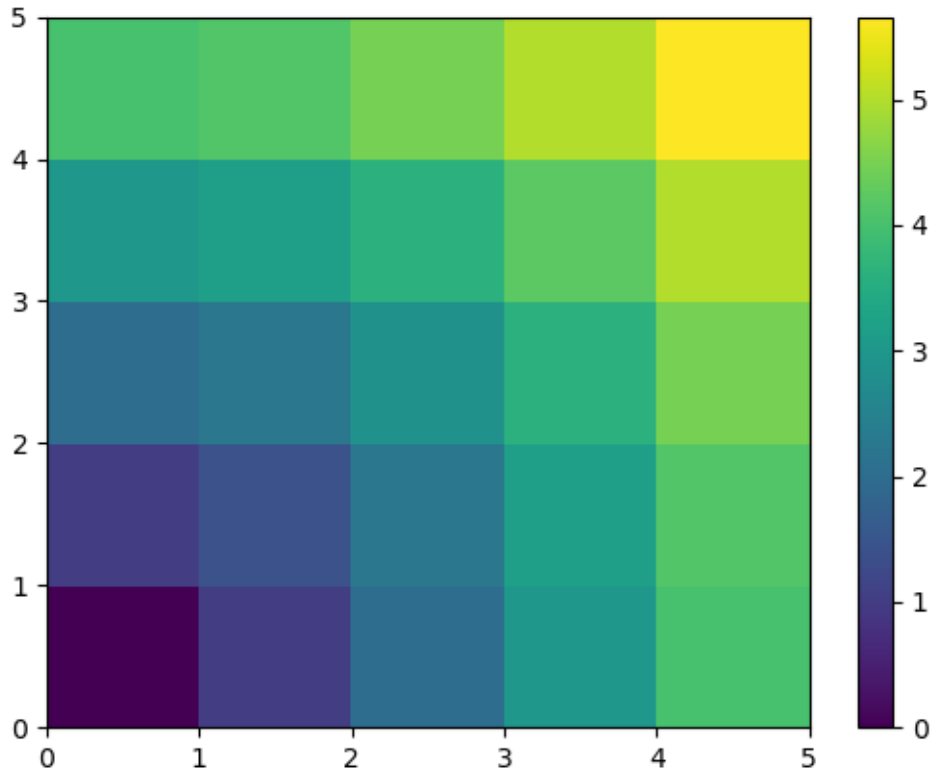



```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 3, 20)
y = np.linspace(0, 9, 20)
plt.plot(x, y)
plt.plot(x, y, 'o')
plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

Distances exercise

Plot distances in a grid

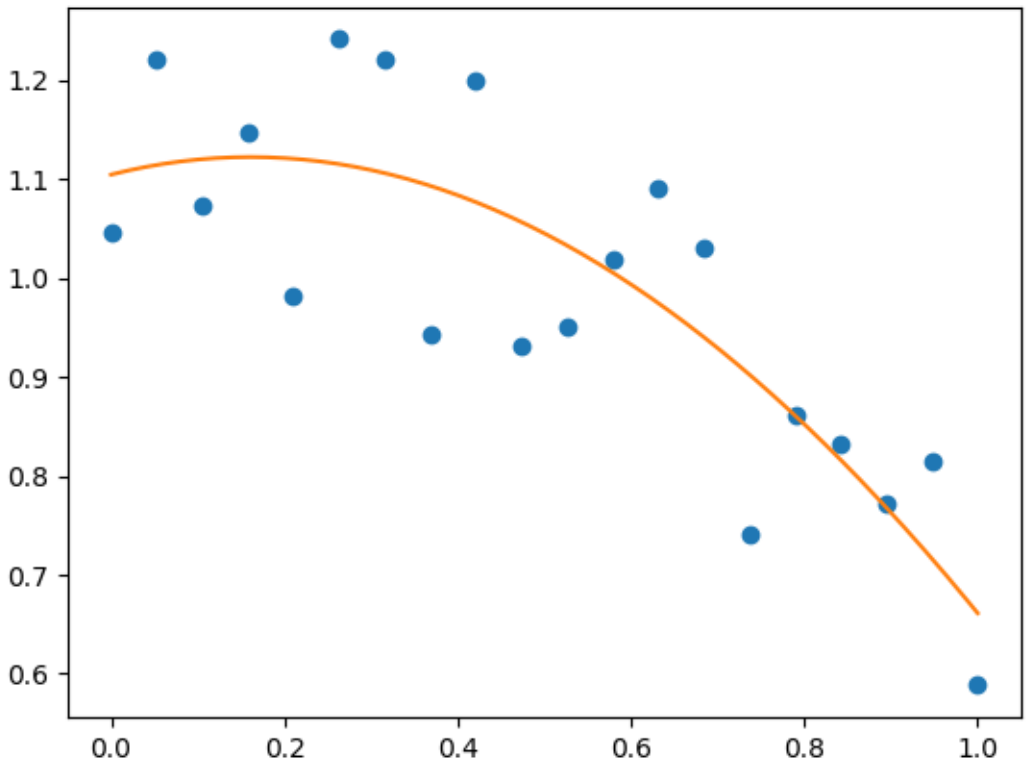



```
import numpy as np
import matplotlib.pyplot as plt

x, y = np.arange(5), np.arange(5)[:, np.newaxis]
distance = np.sqrt(x ** 2 + y ** 2)
plt.pcolor(distance)
plt.colorbar()
plt.show()
```

Total running time of the script: (0 minutes 0.116 seconds)

Fitting to polynomial

Plot noisy data and their polynomial fit




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

np.random.seed(12)

x = np.linspace(0, 1, 20)
y = np.cos(x) + 0.3*np.random.rand(20)
p = np.poly1d(np.polyfit(x, y, 3))

t = np.linspace(0, 1, 200)
plt.plot(x, y, 'o', t, p(t), '-')
plt.show()
```

Total running time of the script: (0 minutes 0.054 seconds)

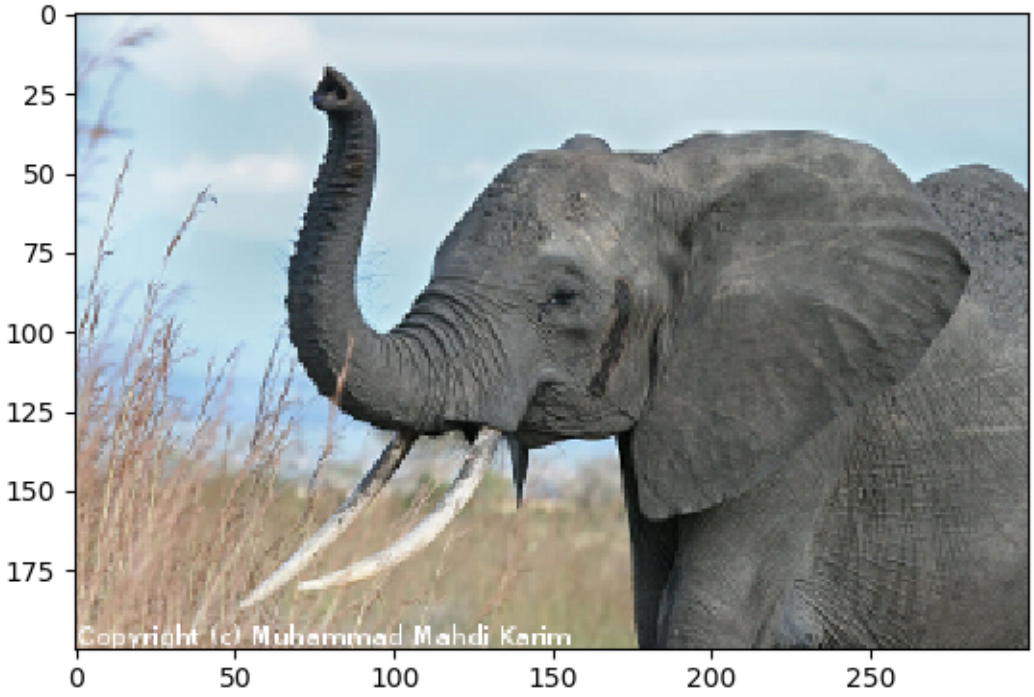
Reading and writing an elephant

Read and write images

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

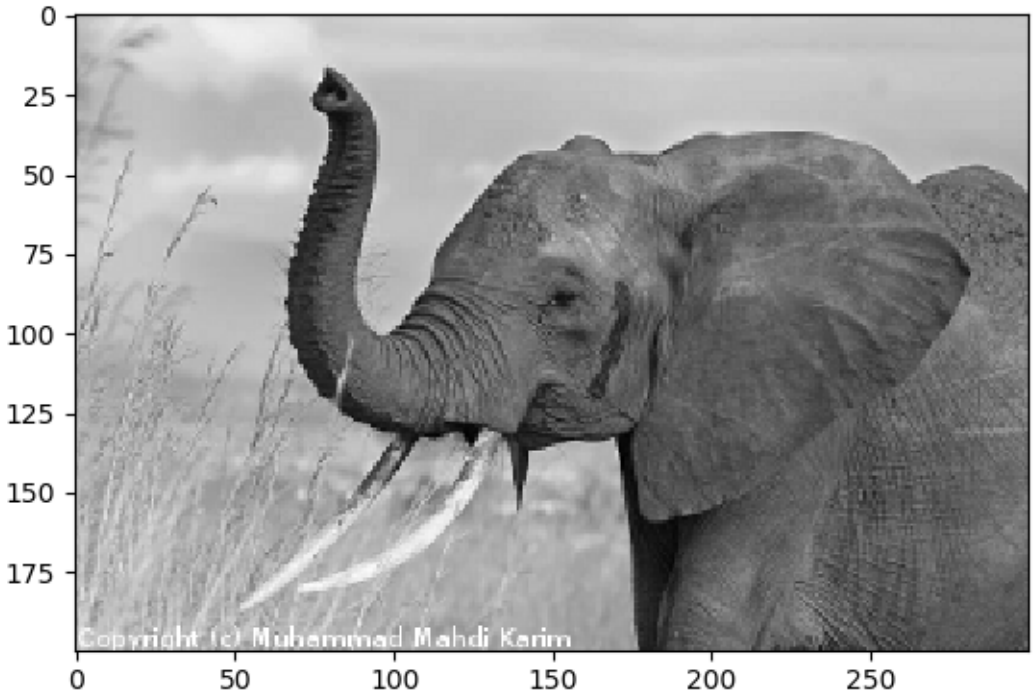
original figure

```
plt.figure()
img = plt.imread('../data/elephant.png')
plt.imshow(img)
```



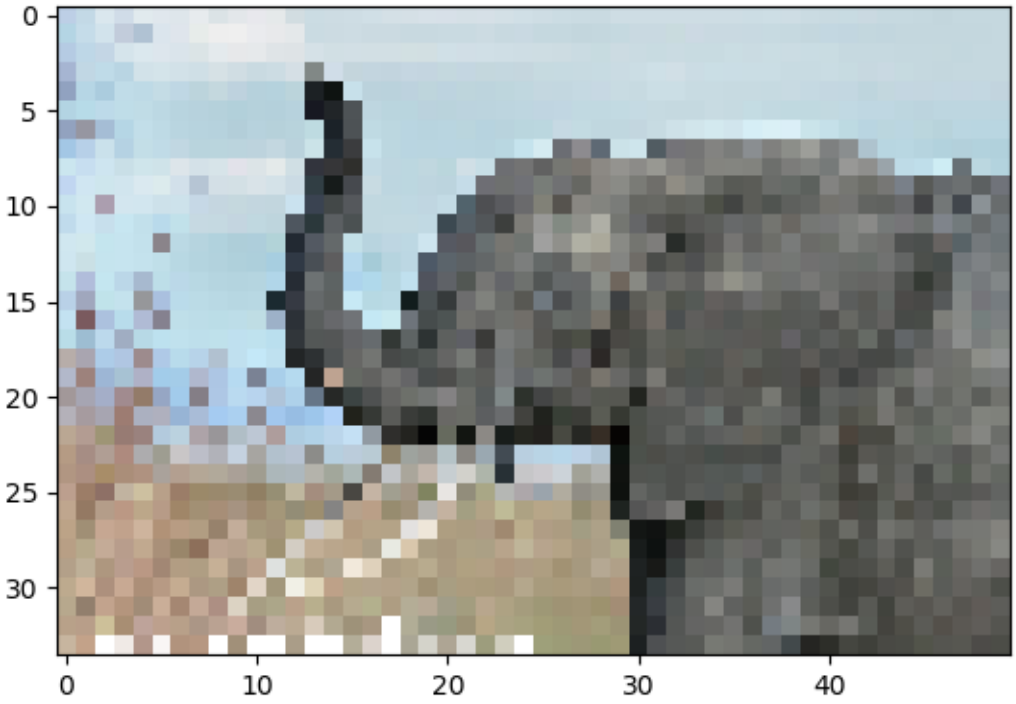
red channel displayed in grey

```
plt.figure()
img_red = img[:, :, 0]
plt.imshow(img_red, cmap=plt.cm.gray)
```



lower resolution

```
plt.figure()
img_tiny = img[::6, ::6]
plt.imshow(img_tiny, interpolation='nearest')

plt.show()
```



Total running time of the script: (0 minutes 0.217 seconds)

CHAPTER 2

Matplotlib: plotting

Thanks

Many thanks to **Bill Wing** and **Christoph Deil** for review and corrections.

Authors: *Nicolas Rougier, Mike Müller, Gaël Varoquaux*

Chapter contents

2.1 Introduction

Tip: Matplotlib is probably the most used Python package for 2D-graphics. It provides both a quick way to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib in interactive mode covering most common cases.

2.1.1 IPython, Jupyter, and matplotlib modes

Tip: The Jupyter notebook and the IPython enhanced interactive Python, are tuned for the scientific-computing workflow in Python, in combination with Matplotlib:

For interactive matplotlib sessions, turn on the **matplotlib mode**

IPython console When using the IPython console, use:

```
In [1]: %matplotlib
```

Jupyter notebook In the notebook, insert, **at the beginning of the notebook** the following magic:

```
%matplotlib inline
```

2.1.2 pyplot

Tip: *pyplot* provides a procedural interface to the matplotlib object-oriented plotting library. It is modeled closely after Matlab™. Therefore, the majority of plotting commands in pyplot have Matlab™ analogs with similar arguments. Important commands are explained with interactive examples.

```
from matplotlib import pyplot as plt
```

2.2 Simple plot

Tip: In this section, we want to draw the cosine and sine functions on the same plot. Starting from the default settings, we'll enrich the figure step by step to make it nicer.

First step is to get the data for the sine and cosine functions:

```
import numpy as np

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)
```

`X` is now a numpy array with 256 values ranging from $-\pi$ to $+\pi$ (included). `C` is the cosine (256 values) and `S` is the sine (256 values).

To run the example, you can type them in an IPython interactive session:

```
$ ipython --pylab
```

This brings us to the IPython prompt:

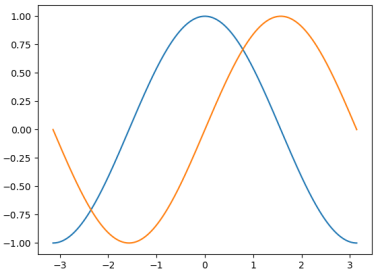
```
IPython 0.13 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

Tip: You can also download each of the examples and run it using regular python, but you will lose interactive data manipulation:

```
$ python plot_exercise_1.py
```

You can get source for each step by clicking on the corresponding figure.

2.2.1 Plotting with default settings



Hint: Documentation

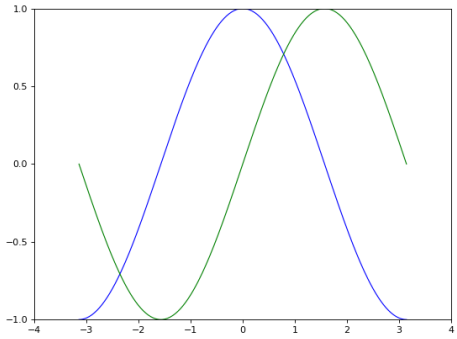
- plot tutorial
- plot() command

Tip: Matplotlib comes with a set of default settings that allow customizing all kinds of properties. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on.

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

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C)
```

```
plt.plot(X, S)

plt.show()
```

2.2.2 Instantiating defaults



Hint: Documentation

- Customizing matplotlib

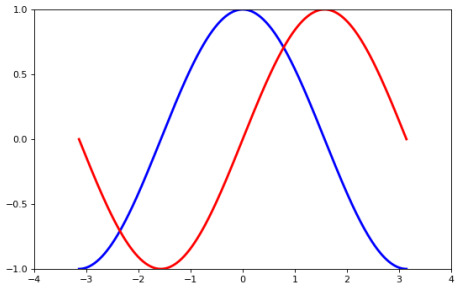
In the script below, we've instantiated (and commented) all the figure settings that influence the appearance of the plot.

Tip: The settings have been explicitly set to their default values, but now you can interactively play with the values to explore their affect (see *Line properties* and *Line styles* below).

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

# Create a figure of size 8x6 inches, 80 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
plt.xlim(-4.0, 4.0)

# Set x ticks
```

```
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))

# Set y limits
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Save figure using 72 dots per inch
# plt.savefig("exercise_2.png", dpi=72)

# Show result on screen
plt.show()
```

2.2.3 Changing colors and line widths



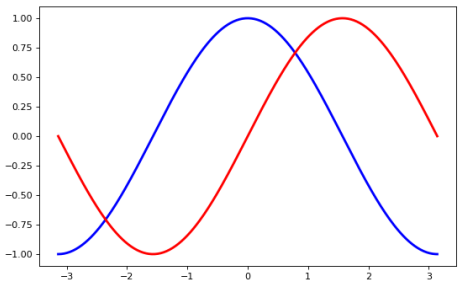
Hint: Documentation

- Controlling line properties
- Line API

Tip: First step, we want to have the cosine in blue and the sine in red and a slighty thicker line for both of them. We'll also slightly alter the figure size to make it more horizontal.

```
...
plt.figure(figsize=(10, 6), dpi=80)
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
...
```

2.2.4 Setting limits



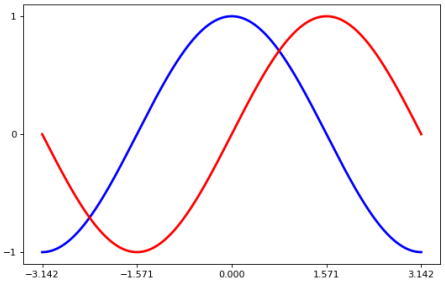
Hint: Documentation

- xlim() command
- ylim() command

Tip: Current limits of the figure are a bit too tight and we want to make some space in order to clearly see all data points.

```
...
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
...
```

2.2.5 Setting ticks



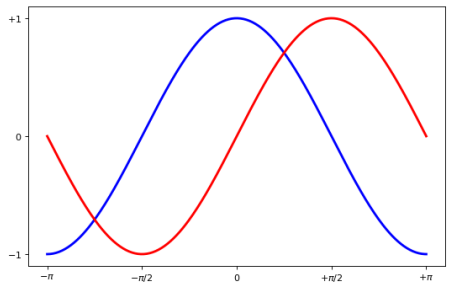
Hint: Documentation

- xticks() command
- yticks() command
- Tick container
- Tick locating and formatting

Tip: Current ticks are not ideal because they do not show the interesting values (+/-π,+/-π/2) for sine and cosine. We'll change them such that they show only these values.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi])
plt.yticks([-1, 0, +1])
...
```

2.2.6 Setting tick labels



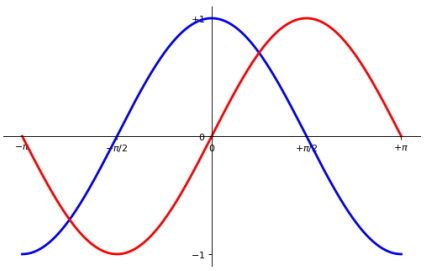
Hint: Documentation

- Working with text
- xticks() command
- yticks() command
- set_xticklabels()
- set_yticklabels()

Tip: Ticks are now properly placed but their label is not very explicit. We could guess that 3.142 is π but it would be better to make it explicit. When we set tick values, we can also provide a corresponding label in the second argument list. Note that we'll use latex to allow for nice rendering of the label.

```
...
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
           [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.yticks([-1, 0, +1],
           [r'$-1$', r'$0$', r'$+1$'])
...
```

2.2.7 Moving spines



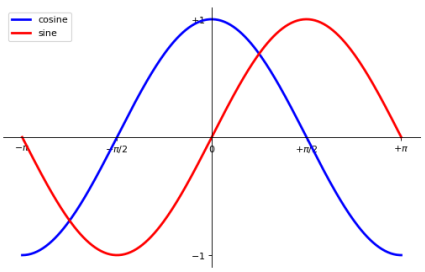
Hint: Documentation

- Spines
- Axis container
- Transformations tutorial

Tip: Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions and until now, they were on the border of the axis. We'll change that since we want to have them in the middle. Since there are four of them (top/bottom/left/right), we'll discard the top and right by setting their color to none and we'll move the bottom and left ones to coordinate 0 in data space coordinates.

```
...
ax = plt.gca()  # gca stands for 'get current axis'
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
...
```

2.2.8 Adding a legend



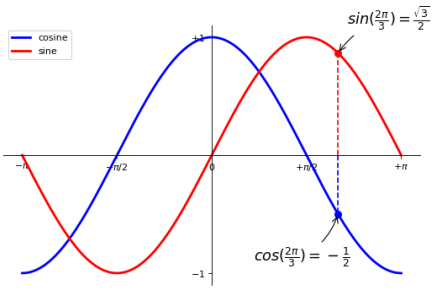
Hint: Documentation

- Legend guide
- legend() command
- Legend API

Tip: Let's add a legend in the upper left corner. This only requires adding the keyword argument label (that will be used in the legend box) to the plot commands.

```
...
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-", label="sine")

plt.legend(loc='upper left')
...
```

2.2.9 Annotate some points



Hint: Documentation

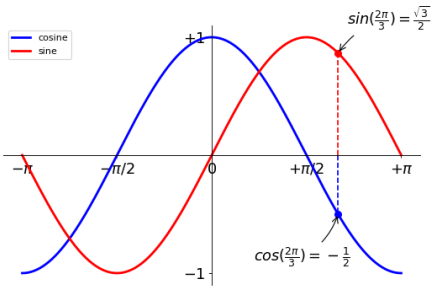
- Annotating axis
- annotate() command

Tip: Let's annotate some interesting points using the annotate command. We chose the $2\pi/3$ value and we want to annotate both the sine and the cosine. We'll first draw a marker on the curve as well as a straight dotted line. Then, we'll use the annotate command to display some text with an arrow.

```
...

t = 2 * np.pi / 3
plt.plot([t, t], [0, np.cos(t)], color='blue', linewidth=2.5, linestyle="--")
plt.scatter([t, ], [np.cos(t), ], 50, color='blue')

plt.annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',
             xy=(t, np.cos(t)), xycoords='data',
             xytext=(-90, -50), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
```

```
plt.plot([t, t],[0, np.sin(t)], color='red', linewidth=2.5, linestyle="--")
plt.scatter([t, ],[np.sin(t), ], 50, color='red')

plt.annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
             xy=(t, np.sin(t)), xycoords='data',
             xytext=(+10, +30), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
...
```

2.2.10 Devil is in the details



Hint: Documentation

- Artists
- BBox

Tip: The tick labels are now hardly visible because of the blue and red lines. We can make them bigger and we can also adjust their properties such that they'll be rendered on a semi-transparent white background. This will allow us to see both the data and the labels.

```
...
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))
...
```

2.3 Figures, Subplots, Axes and Ticks

A **"figure"** in matplotlib means the whole window in the user interface. Within this figure there can be **"subplots"**.

Tip: So far we have used implicit figure and axes creation. This is handy for fast plots. We can have more control over the display using figure, subplot, and axes explicitly. While subplot positions the plots in a regular grid, axes allows free placement within the figure. Both can be useful depending on your intention. We've already worked with figures and subplots without explicitly calling them. When we call plot, matplotlib calls `gca()` to get the

current axes and gca in turn calls `gcf()` to get the current figure. If there is none it calls `figure()` to make one, strictly speaking, to make a `subplot(111)`. Let's look at the details.

2.3.1 Figures

Tip: A figure is the windows in the GUI that has "Figure #" as title. Figures are numbered starting from 1 as opposed to the normal Python way starting from 0. This is clearly MATLAB-style. There are several parameters that determine what the figure looks like:

Argument	Default	Description
`num`	`1`	number of figure
`figsize`	`figure.figsize`	figure size in inches (width, height)
`dpi`	`figure.dpi`	resolution in dots per inch
`facecolor`	`figure.facecolor`	color of the drawing background
`edgecolor`	`figure.edgecolor`	color of edge around the drawing background
`frameon`	`True`	draw figure frame or not

Tip: The defaults can be specified in the resource file and will be used most of the time. Only the number of the figure is frequently changed.

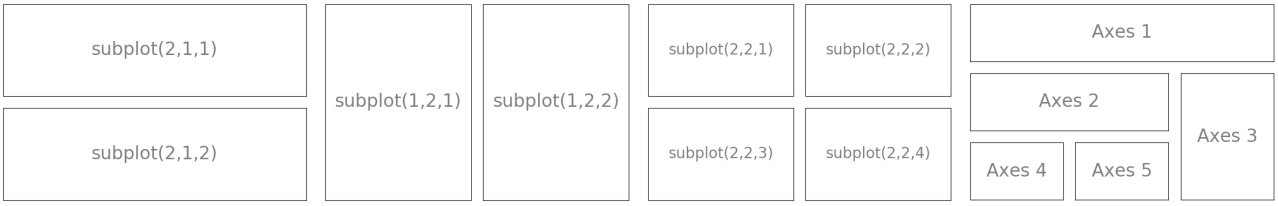
As with other objects, you can set figure properties also setp or with the set_something methods.

When you work with the GUI you can close a figure by clicking on the x in the upper right corner. But you can close a figure programmatically by calling close. Depending on the argument it closes (1) the current figure (no argument), (2) a specific figure (figure number or figure instance as argument), or (3) all figures (`"all"` as argument).

```
plt.close(1)     # Closes figure 1
```

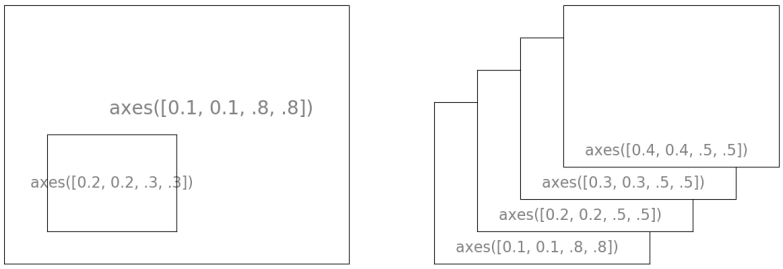
2.3.2 Subplots

Tip: With subplot you can arrange plots in a regular grid. You need to specify the number of rows and columns and the number of the plot. Note that the gridspec command is a more powerful alternative.



2.3.3 Axes

Axes are very similar to subplots but allow placement of plots at any location in the figure. So if we want to put a smaller plot inside a bigger one we do so with axes.



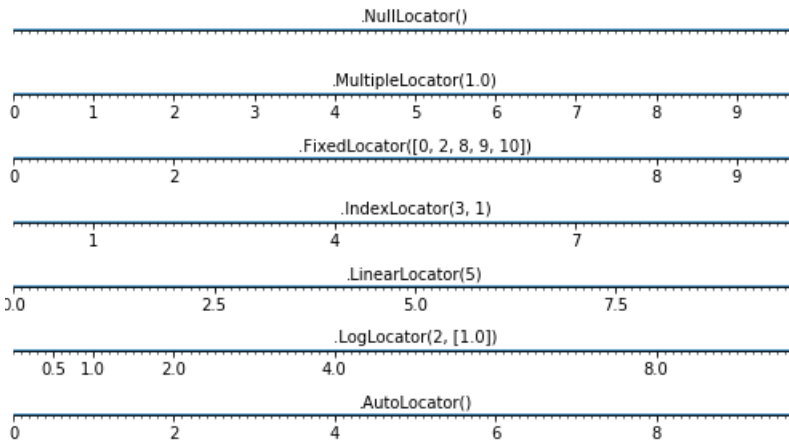
2.3.4 Ticks

Well formatted ticks are an important part of publishing-ready figures. Matplotlib provides a totally configurable system for ticks. There are tick locators to specify where ticks should appear and tick formatters to give ticks the appearance you want. Major and minor ticks can be located and formatted independently from each other. Per default minor ticks are not shown, i.e. there is only an empty list for them because it is as `NullLocator` (see below).

Tick Locators

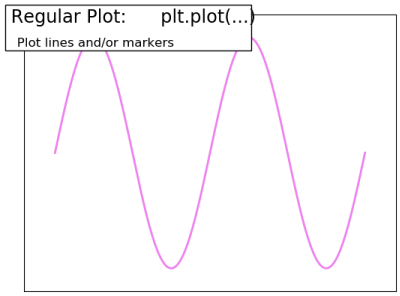
Tick locators control the positions of the ticks. They are set as follows:

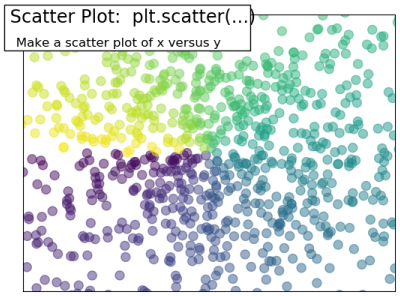
```
ax = plt.gca()
ax.xaxis.set_major_locator(eval(locator))
```

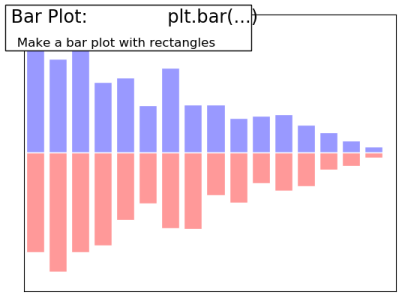


There are several locators for different kind of requirements:
All of these locators derive from the base class `matplotlib.ticker.Locator`. You can make your own locator deriving from it. Handling dates as ticks can be especially tricky. Therefore, matplotlib provides special locators in matplotlib.dates.

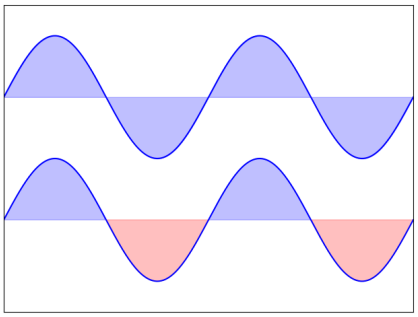
2.4 Other Types of Plots: examples and exercises







2.4.1 Regular Plots



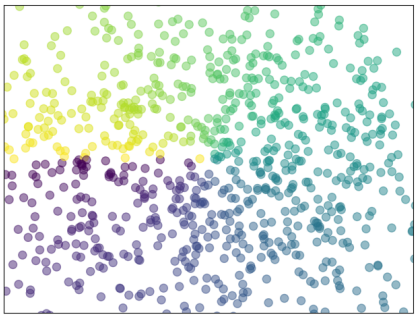
Starting from the code below, try to reproduce the graphic taking care of filled areas:

Hint: You need to use the fill_between command.

```
n = 256
X = np.linspace(-np.pi, np.pi, n, endpoint=True)
Y = np.sin(2 * X)

plt.plot(X, Y + 1, color='blue', alpha=1.00)
plt.plot(X, Y - 1, color='blue', alpha=1.00)
```

Click on the figure for solution.

2.4.2 Scatter Plots



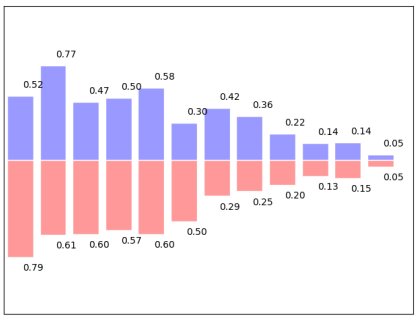
Starting from the code below, try to reproduce the graphic taking care of marker size, color and transparency.

Hint: Color is given by angle of (X,Y).

```
n = 1024
X = np.random.normal(0,1,n)
Y = np.random.normal(0,1,n)

plt.scatter(X,Y)
```

Click on figure for solution.

2.4.3 Bar Plots



Starting from the code below, try to reproduce the graphic by adding labels for red bars.

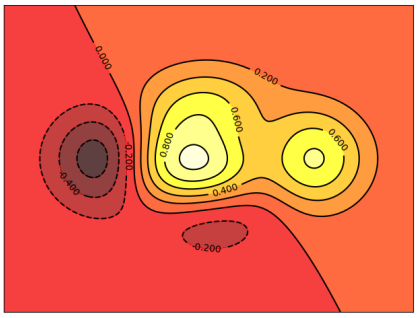
Hint: You need to take care of text alignment.

```
n = 12
X = np.arange(n)
Y1 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)
Y2 = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)

plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')
plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):
    plt.text(x + 0.4, y + 0.05, '%.2f' % y, ha='center', va='bottom')
```

```
plt.ylim(-1.25, +1.25)
```

Click on figure for solution.

2.4.4 Contour Plots



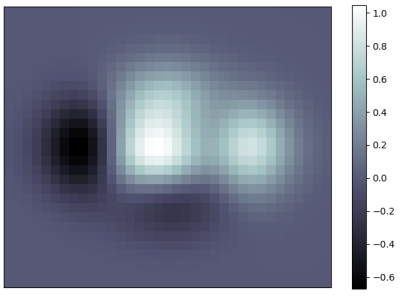
Starting from the code below, try to reproduce the graphic taking care of the colormap (see *Colormaps* below).

Hint: You need to use the clabel command.

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 -y ** 2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)

plt.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='jet')
C = plt.contour(X, Y, f(X, Y), 8, colors='black', linewidth=.5)
```

Click on figure for solution.

2.4.5 Imshow



Starting from the code below, try to reproduce the graphic taking care of colormap, image interpolation and origin.

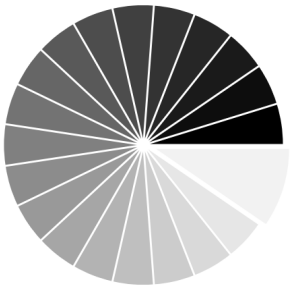
Hint: You need to take care of the `origin` of the image in the imshow command and use a colorbar

---

```
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)

n = 10
x = np.linspace(-3, 3, 4 * n)
y = np.linspace(-3, 3, 3 * n)
X, Y = np.meshgrid(x, y)
plt.imshow(f(X, Y))
```

Click on the figure for the solution.

2.4.6 Pie Charts



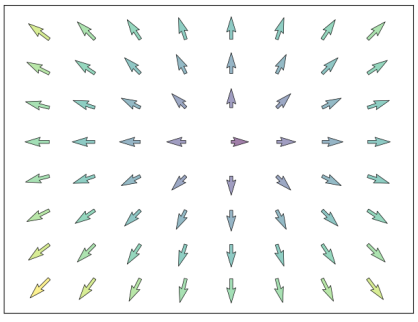
Starting from the code below, try to reproduce the graphic taking care of colors and slices size.

Hint: You need to modify Z.

```
Z = np.random.uniform(0, 1, 20)
plt.pie(Z)
```

Click on the figure for the solution.

2.4.7 Quiver Plots



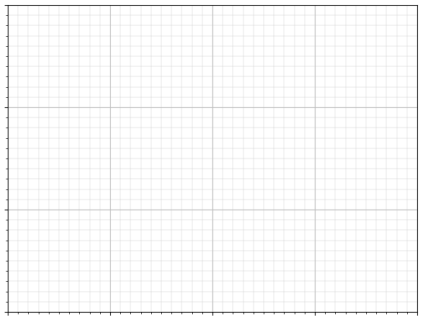
Starting from the code above, try to reproduce the graphic taking care of colors and orientations.

Hint: You need to draw arrows twice.

```
n = 8
X, Y = np.mgrid[0:n, 0:n]
plt.quiver(X, Y)
```

Click on figure for solution.

2.4.8 Grids



Starting from the code below, try to reproduce the graphic taking care of line styles.

```
axes = plt.gca()
axes.set_xlim(0, 4)
axes.set_ylim(0, 3)
axes.set_xticklabels([])
axes.set_yticklabels([])
```

Click on figure for solution.

2.4.9 Multi Plots

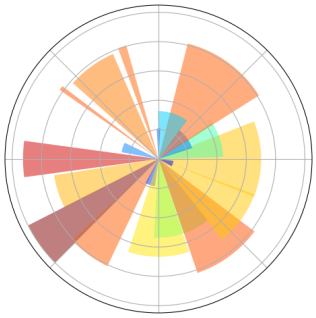
Starting from the code below, try to reproduce the graphic.

Hint: You can use several subplots with different partition.

```
plt.subplot(2, 2, 1)
plt.subplot(2, 2, 3)
plt.subplot(2, 2, 4)
```

Click on figure for solution.

2.4.10 Polar Axis



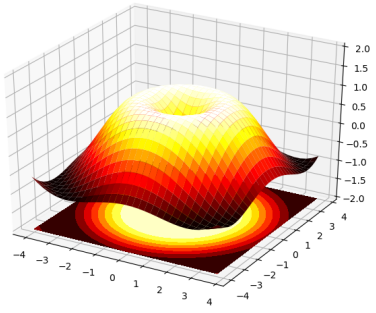
Hint: You only need to modify the `axes` line

Starting from the code below, try to reproduce the graphic.

```
plt.axes([0, 0, 1, 1])

N = 20
theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
bars = plt.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_alpha(0.5)
```

Click on figure for solution.

2.4.11 3D Plots



Starting from the code below, try to reproduce the graphic.

Hint: You need to use contourf

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
```

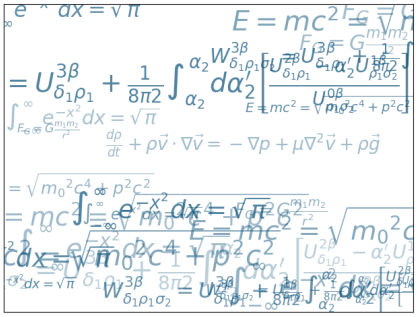
```
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

Click on figure for solution.

See also:

3D plotting with Mayavi

2.4.12 Text



Try to do the same from scratch !

Hint: Have a look at the matplotlib logo.

Click on figure for solution.

Quick read

If you want to do a first quick pass through the Scipy lectures to learn the ecosystem, you can directly skip to the next chapter: *Scipy : high-level scientific computing*.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter later.

CHAPTER 3

Scipy : high-level scientific computing

Authors: *Gaël Varoquaux, Adrien Chauve, Andre Espaze, Emmanuelle Gouillart, Ralf Gommers*

Scipy

The `scipy` package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

Tip: `scipy` can be compared to other standard scientific-computing libraries, such as the GSL (GNU Scientific Library for C and C++), or Matlab's toolboxes. `scipy` is the core package for scientific routines in Python; it is meant to operate efficiently on `numpy` arrays, so that numpy and scipy work hand in hand.

Before implementing a routine, it is worth checking if the desired data processing is not already implemented in Scipy. As non-professional programmers, scientists often tend to **re-invent the wheel**, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code. By contrast, `Scipy`'s routines are optimized and tested, and should therefore be used when possible.

Chapters contents

- *Linear algebra operations:* `scipy.linalg`
- *Interpolation:* `scipy.interpolate`
- *Optimization and fit:* `scipy.optimize`
- *Statistics and random numbers:* `scipy.stats`
- *Numerical integration:* `scipy.integrate`
- *Fast Fourier transforms:* `scipy.fftpack`
- *Signal processing:* `scipy.signal`
- *Image manipulation:* `scipy.ndimage`
- *Summary exercises on scientific computing*
- *Full code examples for the scipy chapter*

Warning: This tutorial is far from an introduction to numerical computing. As enumerating the different submodules and functions in scipy would be very boring, we concentrate instead on a few examples to give a general idea of how to use `scipy` for scientific computing.

`scipy` is composed of task-specific sub-modules:

`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

Tip: They all depend on `numpy`, but are mostly independent of each other. The standard way of importing Numpy and these Scipy modules is:

```
>>> import numpy as np
>>> from scipy import stats  # same for other sub-modules
```

The main `scipy` namespace mostly contains functions that are really numpy functions (try `scipy.cos is np.cos`). Those are exposed for historical reasons; there's no reason to use `import scipy` in your code.

3.1 File input/output: `scipy.io`

Matlab files: Loading and saving:

```
>>> from scipy import io as spio
>>> a = np.ones((3, 3))
>>> spio.savemat('file.mat', {'a': a}) # savemat expects a dictionary
>>> data = spio.loadmat('file.mat')
>>> data['a']
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Warning: Python / Matlab mismatches, *eg* matlab does not represent 1D arrays

```
>>> a = np.ones(3)
>>> a
array([ 1.,  1.,  1.])
>>> spio.savemat('file.mat', {'a': a})
>>> spio.loadmat('file.mat')['a']
array([[ 1.,  1.,  1.]])
```

Notice the difference?

Image files: Reading images:

```
>>> from scipy import misc
>>> misc.imread('fname.png')
array(...)
>>> # Matplotlib also has a similar function
>>> import matplotlib.pyplot as plt
>>> plt.imread('fname.png')
array(...)
```

See also:

- Load text files: `numpy.loadtxt()`/`numpy.savetxt()`
- Clever loading of text/csv files: `numpy.genfromtxt()`/`numpy.recfromcsv()`
- Fast and efficient, but numpy-specific, binary format: `numpy.save()`/`numpy.load()`
- More advanced input/output of images in scikit-image: `skimage.io`

3.2 Linear algebra operations: `scipy.linalg`

Tip: The `scipy.linalg` module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK).

- The `scipy.linalg.det()` function computes the determinant of a square matrix:

```
>>> from scipy import linalg
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> linalg.det(arr)
-2.0
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.det(arr)
0.0
>>> linalg.det(np.ones((3, 4)))
Traceback (most recent call last):
...
ValueError: expected square matrix
```

- The `scipy.linalg.inv()` function computes the inverse of a square matrix:

```
>>> arr = np.array([[1, 2],
...                 [3, 4]])
>>> iarr = linalg.inv(arr)
>>> iarr
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.allclose(np.dot(arr, iarr), np.eye(2))
True
```

Finally computing the inverse of a singular matrix (its determinant is zero) will raise `LinAlgError`:

```
>>> arr = np.array([[3, 2],
...                 [6, 4]])
>>> linalg.inv(arr)
Traceback (most recent call last):
...
...LinAlgError: singular matrix
```

- More advanced operations are available, for example singular-value decomposition (SVD):

```
>>> arr = np.arange(9).reshape((3, 3)) + np.diag([1, 0, 1])
>>> uarr, spec, vharr = linalg.svd(arr)
```

The resulting array spectrum is:

```
>>> spec
array([ 14.88982544,   0.45294236,   0.29654967])
```

The original matrix can be re-composed by matrix multiplication of the outputs of `svd` with `np.dot`:

```
>>> sarr = np.diag(spec)
>>> svd_mat = uarr.dot(sarr).dot(vharr)
>>> np.allclose(svd_mat, arr)
True
```

SVD is commonly used in statistics and signal processing. Many other standard decompositions (QR, LU, Cholesky, Schur), as well as solvers for linear systems, are available in `scipy.linalg`.

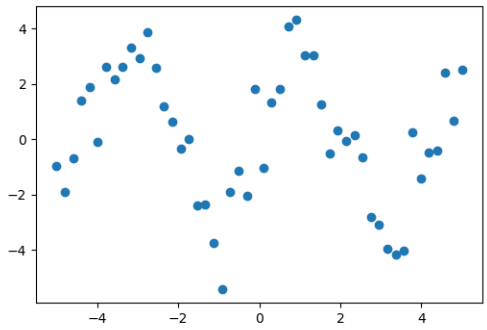
3.3 Optimization and fit: `scipy.optimize`

Optimization is the problem of finding a numerical solution to a minimization or equality.

Tip: The `scipy.optimize` module provides algorithms for function minimization (scalar or multi-dimensional), curve fitting and root finding.

```
>>> from scipy import optimize
```

3.3.1 Curve fitting

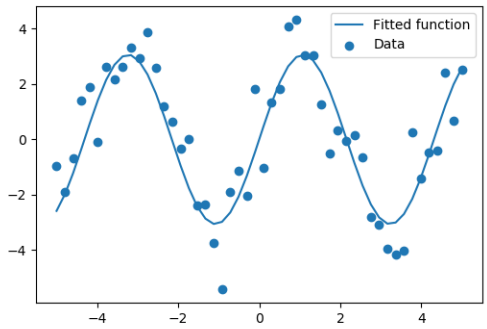


Suppose we have data on a sine wave, with some noise:

```
>>> x_data = np.linspace(-5, 5, num=50)
>>> y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)
```

If we know that the data lies on a sine wave, but not the amplitudes or the period, we can find those by least squares curve fitting. First we have to define the test function to fit, here a sine with unknown amplitude and period:

```
>>> def test_func(x, a, b):
...     return a * np.sin(b * x)
```



We then use `scipy.optimize.curve_fit()` to find a and b:

```
>>> params, params_covariance = optimize.curve_fit(test_func, x_data, y_data, p0=[2, 2])
>>> print(params)
[ 3.05931973  1.45754553]
```

Exercise: Curve fitting of temperature data

The temperature extremes in Alaska for each month, starting in January, are given by (in degrees Celcius):

```
max:  17,  19,  21,  28,  33,  38, 37,  37,  31,  23,  19,  18
min: -62, -59, -56, -46, -32, -18, -9, -13, -25, -46, -52, -58
```

1. Plot these temperature extremes.
2. Define a function that can describe min and max temperatures. Hint: this function has to have a period of 1 year. Hint: include a time offset.
3. Fit this function to the data with `scipy.optimize.curve_fit()`.
4. Plot the result. Is the fit reasonable? If not, why?
5. Is the time offset for min and max temperatures the same within the fit accuracy?

solution

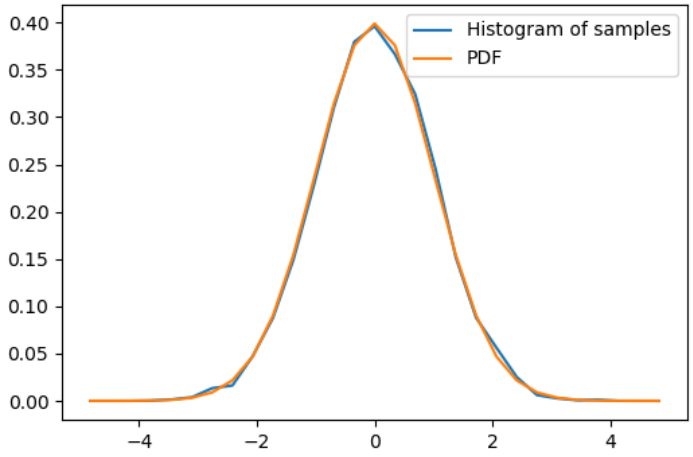
3.4 Statistics and random numbers: `scipy.stats`

The module `scipy.stats` contains statistical tools and probabilistic descriptions of random processes. Random number generators for various random process can be found in `numpy.random`.

3.4.1 Distributions: histogram and probability density function

Given observations of a random process, their histogram is an estimator of the random process's PDF (probability density function):

```
>>> samples = np.random.normal(size=1000)
>>> bins = np.arange(-4, 5)
>>> bins
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> histogram = np.histogram(samples, bins=bins, normed=True)[0]
>>> bins = 0.5*(bins[1:] + bins[:-1])
>>> bins
array([-3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5])
>>> from scipy import stats
>>> pdf = stats.norm.pdf(bins)  # norm is a distribution object

>>> plt.plot(bins, histogram)
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(bins, pdf)
[<matplotlib.lines.Line2D object at ...>]
```



The distribution objects

`scipy.stats.norm` is a distribution object: each distribution in `scipy.stats` is represented as an object. Here it's the normal distribution, and it comes with a PDF, a CDF, and much more.

If we know that the random process belongs to a given family of random processes, such as normal processes, we can do a maximum-likelihood fit of the observations to estimate the parameters of the underlying distribution. Here we fit a normal process to the observed data:

```
>>> loc, std = stats.norm.fit(samples)
>>> loc
-0.045256707...
>>> std
0.9870331586...
```

Exercise: Probability distributions

Generate 1000 random variates from a gamma distribution with a shape parameter of 1, then plot a histogram from those samples. Can you plot the pdf on top (it should match)?

Extra: the distributions have many useful methods. Explore them by reading the docstring or by using tab completion. Can you recover the shape parameter 1 by using the `fit` method on your random variates?

3.4.2 Mean, median and percentiles

The mean is an estimator of the center of the distribution:

```
>>> np.mean(samples)
-0.0452567074...
```

The median another estimator of the center. It is the value with half of the observations below, and half above:

```
>>> np.median(samples)
-0.0580280347...
```

Tip: Unlike the mean, the median is not sensitive to the tails of the distribution. It is "robust".

Exercise: Compare mean and median on samples of a Gamma distribution

Which one seems to be the best estimator of the center for the Gamma distribution?

The median is also the percentile 50, because 50% of the observation are below it:

```
>>> stats.scoreatpercentile(samples, 50)
-0.0580280347...
```

Similarly, we can calculate the percentile 90:

```
>>> stats.scoreatpercentile(samples, 90)
1.2315935511...
```

Tip: The percentile is an estimator of the CDF: cumulative distribution function.

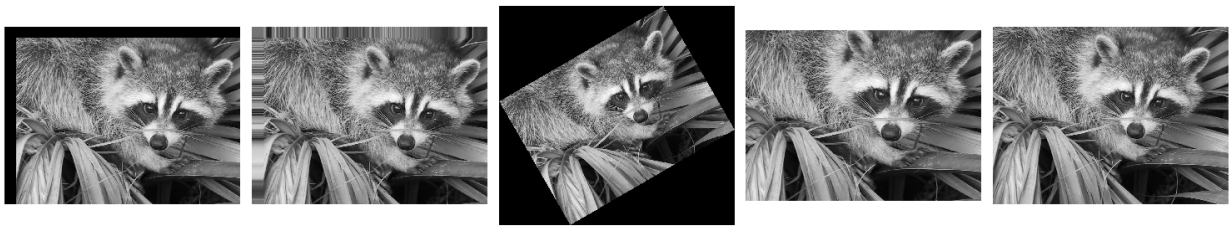
3.5 Image manipulation: `scipy.ndimage`

`scipy.ndimage` provides manipulation of n-dimensional arrays as images.

3.5.1 Geometrical transformations on images

Changing orientation, resolution, ..

```
>>> from scipy import misc  # Load an image
>>> face = misc.face(gray=True)

>>> from scipy import ndimage # Shift, roate and zoom it
>>> shifted_face = ndimage.shift(face, (50, 50))
>>> shifted_face2 = ndimage.shift(face, (50, 50), mode='nearest')
>>> rotated_face = ndimage.rotate(face, 30)
>>> cropped_face = face[50:-50, 50:-50]
>>> zoomed_face = ndimage.zoom(face, 2)
>>> zoomed_face.shape
(1536, 2048)
```



```
>>> plt.subplot(151)
<matplotlib.axes._subplots.AxesSubplot object at 0x...>

>>> plt.imshow(shifted_face, cmap=plt.cm.gray)
<matplotlib.image.AxesImage object at 0x...>

>>> plt.axis('off')
(-0.5, 1023.5, 767.5, -0.5)

>>> # etc.
```

3.5.2 Image filtering

Generate a noisy face:

```
>>> from scipy import misc
>>> face = misc.face(gray=True)
>>> face = face[:512, -512:]  # crop out square on right
>>> import numpy as np
>>> noisy_face = np.copy(face).astype(np.float)
>>> noisy_face += face.std() * 0.5 * np.random.standard_normal(face.shape)
```

Apply a variety of filters on it:

```
>>> blurred_face = ndimage.gaussian_filter(noisy_face, sigma=3)
>>> median_face = ndimage.median_filter(noisy_face, size=5)
>>> from scipy import signal
>>> wiener_face = signal.wiener(noisy_face, (5, 5))
```

Other filters in `scipy.ndimage.filters` and `scipy.signal` can be applied to images.

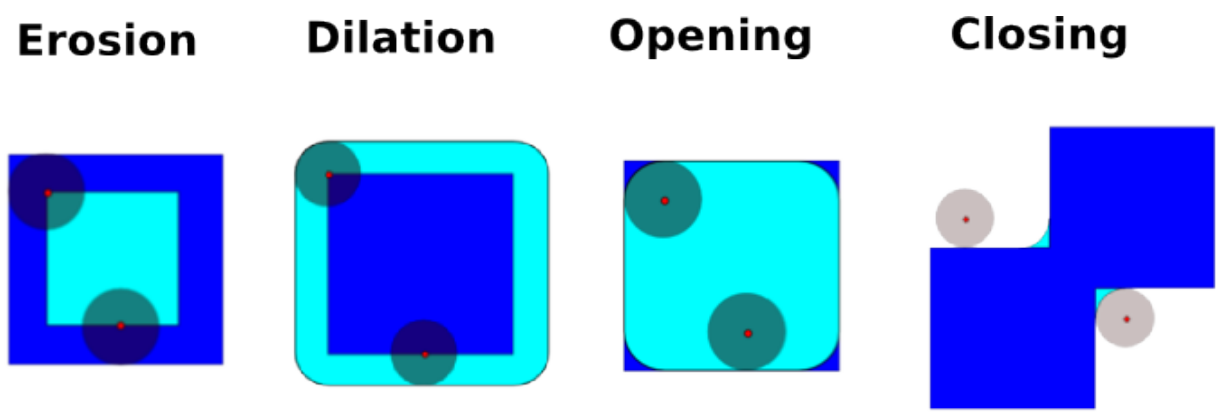
Exercise

Compare histograms for the different filtered images.

3.5.3 Mathematical morphology

Tip: Mathematical morphology stems from set theory. It characterizes and transforms geometrical structures. Binary (black and white) images, in particular, can be transformed using this theory: the sets to be transformed

are the sets of neighboring non-zero-valued pixels. The theory was also extended to gray-valued images.



Mathematical-morphology operations use a *structuring element* in order to modify geometrical structures.

Let us first generate a structuring element:

```
>>> el = ndimage.generate_binary_structure(2, 1)
>>> el
array([[False, True, False],
       [...True, True, True],
       [False, True, False]], dtype=bool)
>>> el.astype(np.int)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

- **Erosion** `scipy.ndimage.binary_erosion()`

```
>>> a = np.zeros((7, 7), dtype=np.int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> #Erosion removes objects smaller than the structure
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[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, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
```

```
       [0, 0, 0, 0, 0, 0, 0]])
```

- **Dilation** `scipy.ndimage.binary_dilation()`

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

- **Opening** `scipy.ndimage.binary_opening()`

```
>>> a = np.zeros((5, 5), dtype=np.int)
>>> a[1:4, 1:4] = 1
>>> a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> ndimage.binary_opening(a, structure=np.ones((3, 3))).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(np.int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

- **Closing:** `scipy.ndimage.binary_closing()`

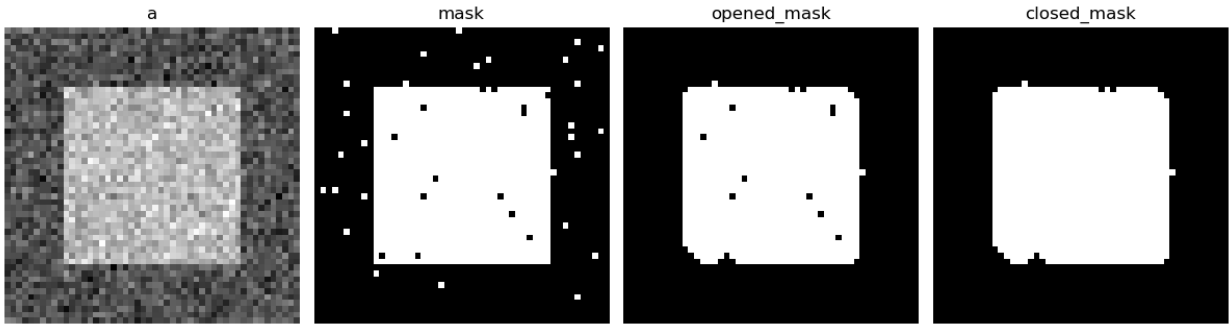
Exercise

Check that opening amounts to eroding, then dilating.

An opening operation removes small structures, while a closing operation fills small holes. Such operations can therefore be used to "clean" an image.

```
>>> a = np.zeros((50, 50))
>>> a[10:-10, 10:-10] = 1
```

```
>>> a += 0.25 * np.random.standard_normal(a.shape)
>>> mask = a>=0.5
>>> opened_mask = ndimage.binary_opening(mask)
>>> closed_mask = ndimage.binary_closing(opened_mask)
```



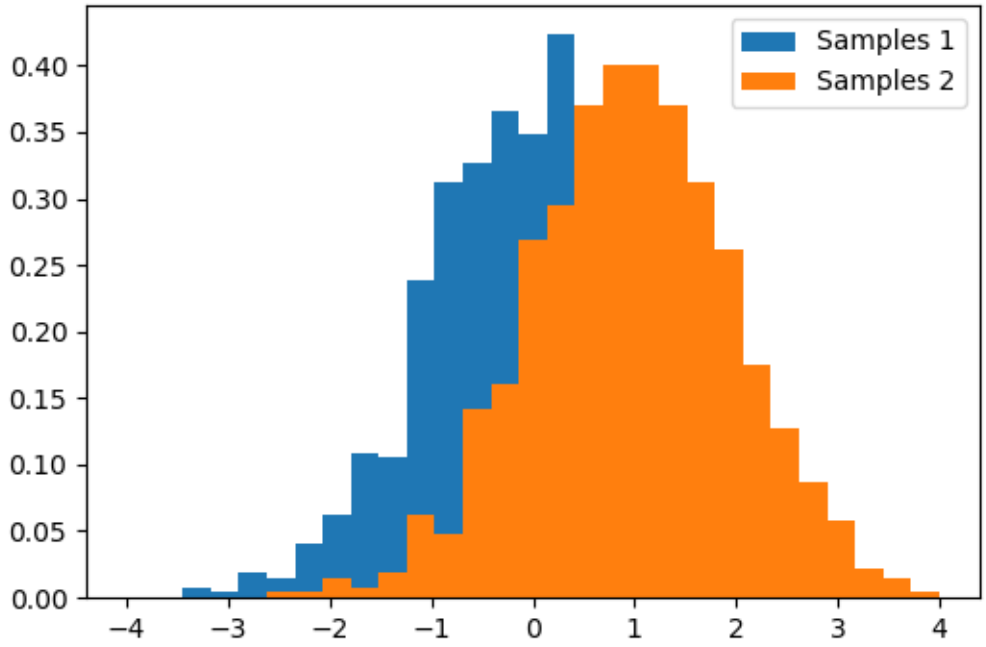
Exercise

Check that the area of the reconstructed square is smaller than the area of the initial square. (The opposite would occur if the closing step was performed *before* the opening).

For *gray-valued* images, eroding (resp. dilating) amounts to replacing a pixel by the minimal (resp. maximal) value among pixels covered by the structuring element centered on the pixel of interest.

```
>>> a = np.zeros((7, 7), dtype=np.int)
>>> a[1:6, 1:6] = 3
>>> a[4, 4] = 2; a[2, 3] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 1, 3, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 3, 2, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_erosion(a, size=(3, 3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

3.5.4 Comparing 2 sets of samples from Gaussians

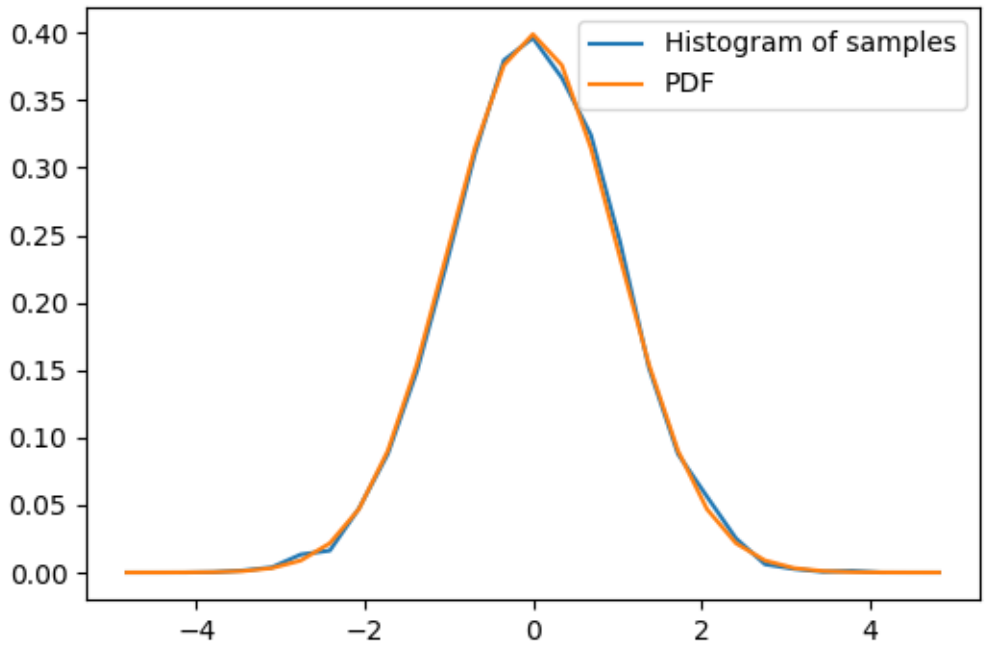



```
import numpy as np
from matplotlib import pyplot as plt

# Generates 2 sets of observations
samples1 = np.random.normal(0, size=1000)
samples2 = np.random.normal(1, size=1000)

# Compute a histogram of the sample
bins = np.linspace(-4, 4, 30)
histogram1, bins = np.histogram(samples1, bins=bins, normed=True)
histogram2, bins = np.histogram(samples2, bins=bins, normed=True)

plt.figure(figsize=(6, 4))
plt.hist(samples1, bins=bins, normed=True, label="Samples 1")
plt.hist(samples2, bins=bins, normed=True, label="Samples 2")
plt.legend(loc='best')
plt.show()
```

Total running time of the script: (0 minutes 0.107 seconds)

3.5.5 Normal distribution: histogram and PDF

Explore the normal distribution: a histogram built from samples and the PDF (probability density function).




```
import numpy as np

# Sample from a normal distribution using numpy's random number generator
samples = np.random.normal(size=10000)

# Compute a histogram of the sample
bins = np.linspace(-5, 5, 30)
histogram, bins = np.histogram(samples, bins=bins, normed=True)

bin_centers = 0.5*(bins[1:] + bins[:-1])

# Compute the PDF on the bin centers from scipy distribution object
from scipy import stats
pdf = stats.norm.pdf(bin_centers)

from matplotlib import pyplot as plt
plt.figure(figsize=(6, 4))
plt.plot(bin_centers, histogram, label="Histogram of samples")
plt.plot(bin_centers, pdf, label="PDF")
plt.legend()
plt.show()
```

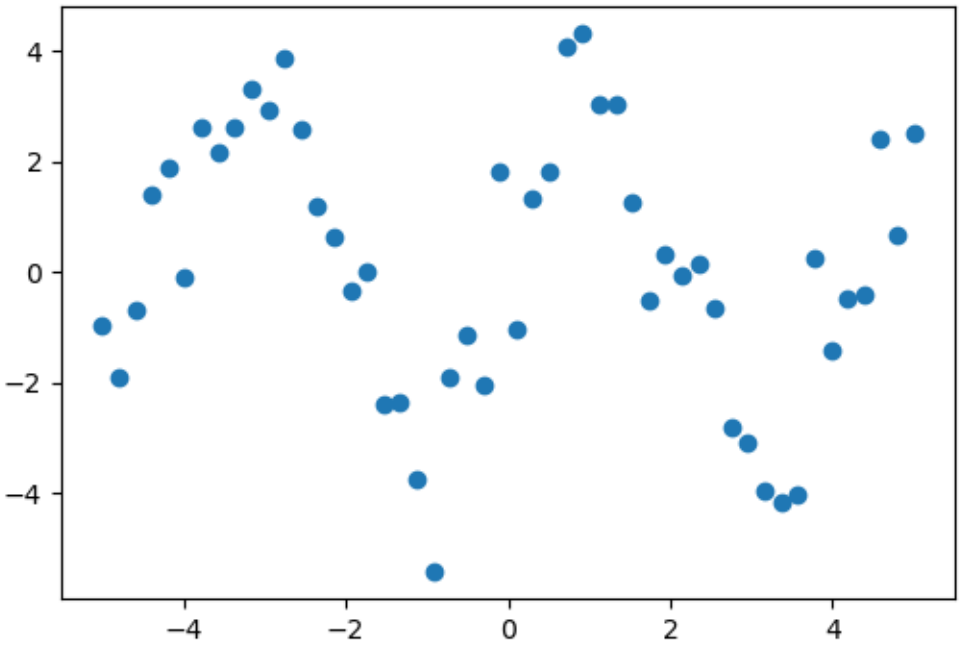
Total running time of the script: (0 minutes 0.056 seconds)

3.5.6 Curve fitting

Demos a simple curve fitting

First generate some data

```
import numpy as np

# Seed the random number generator for reproducibility
np.random.seed(0)

x_data = np.linspace(-5, 5, num=50)
y_data = 2.9 * np.sin(1.5 * x_data) + np.random.normal(size=50)

# And plot it
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data)
```



Now fit a simple sine function to the data

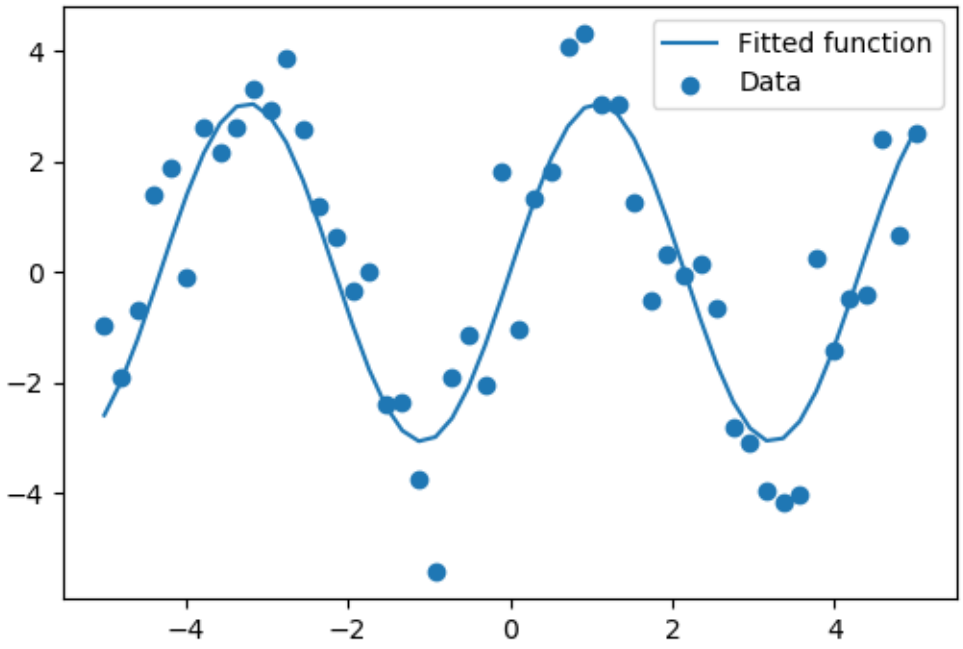
```
from scipy import optimize

def test_func(x, a, b):
    return a * np.sin(b * x)

params, params_covariance = optimize.curve_fit(test_func, x_data, y_data,
                                               p0=[2, 2])

print(params)
```

Out:

```
[ 3.05931973  1.45754553]
```

And plot the resulting curve on the data

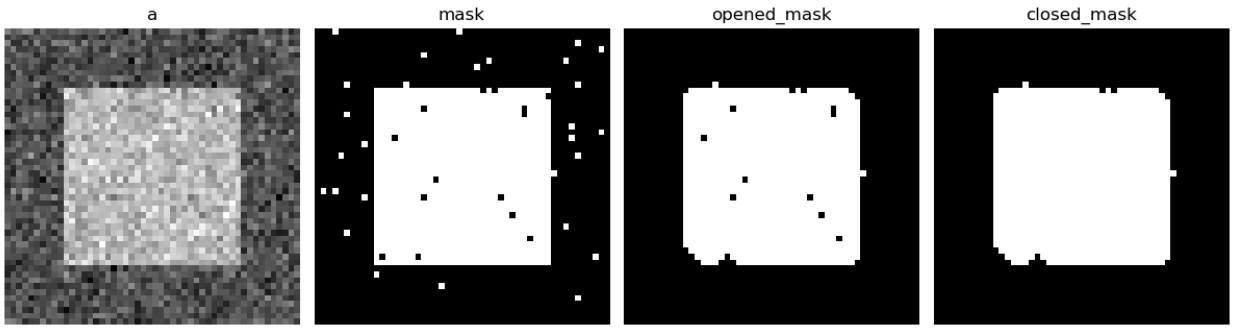
```
plt.figure(figsize=(6, 4))
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, test_func(x_data, params[0], params[1]),
         label='Fitted function')

plt.legend(loc='best')

plt.show()
```



Total running time of the script: (0 minutes 0.107 seconds)

3.5.7 Demo mathematical morphology

A basic demo of binary opening and closing.



```
# Generate some binary data
import numpy as np
np.random.seed(0)
a = np.zeros((50, 50))
a[10:-10, 10:-10] = 1
a += 0.25 * np.random.standard_normal(a.shape)
mask = a>=0.5

# Apply mathematical morphology
from scipy import ndimage
opened_mask = ndimage.binary_opening(mask)
closed_mask = ndimage.binary_closing(opened_mask)

# Plot
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(a, cmap=plt.cm.gray)
plt.axis('off')
plt.title('a')

plt.subplot(142)
plt.imshow(mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('mask')

plt.subplot(143)
plt.imshow(opened_mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('opened_mask')

plt.subplot(144)
plt.imshow(closed_mask, cmap=plt.cm.gray)
plt.title('closed_mask')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 0.202 seconds)

3.5.8 Plot geometrical transformations on images

Demo geometrical transformations of images.

```
# Load some data
from scipy import misc
face = misc.face(gray=True)

# Apply a variety of transformations
from scipy import ndimage
from matplotlib import pyplot as plt
shifted_face = ndimage.shift(face, (50, 50))
shifted_face2 = ndimage.shift(face, (50, 50), mode='nearest')
rotated_face = ndimage.rotate(face, 30)
cropped_face = face[50:-50, 50:-50]
zoomed_face = ndimage.zoom(face, 2)
zoomed_face.shape

plt.figure(figsize=(15, 3))
plt.subplot(151)
plt.imshow(shifted_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(152)
plt.imshow(shifted_face2, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(153)
plt.imshow(rotated_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(154)
plt.imshow(cropped_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplot(155)
plt.imshow(zoomed_face, cmap=plt.cm.gray)
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 1.133 seconds)

3.5.9 Demo connected components

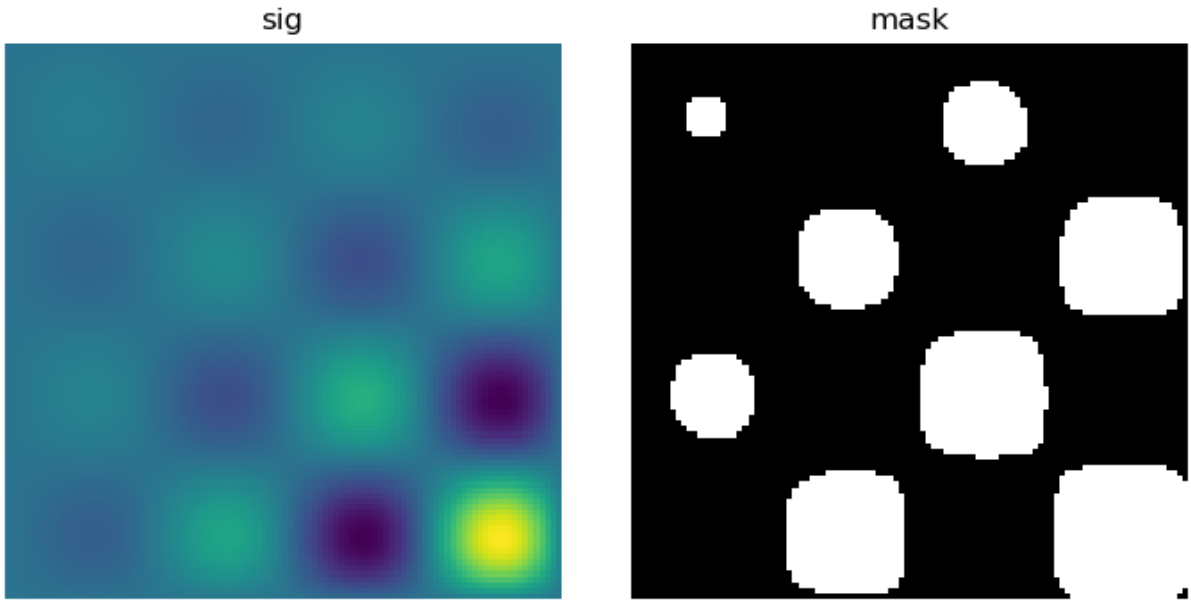
Extracting and labeling connected components in a 2D array

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

Generate some binary data

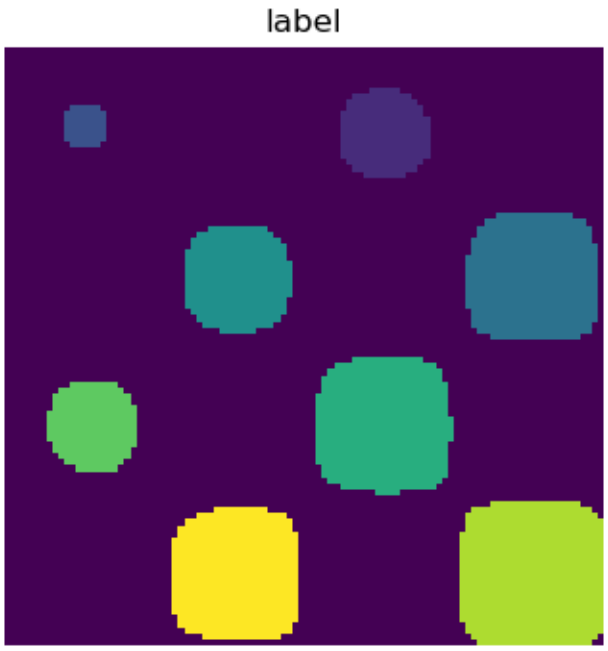
```
np.random.seed(0)
x, y = np.indices((100, 100))
sig = np.sin(2*np.pi*x/50.) * np.sin(2*np.pi*y/50.) * (1+x*y/50.**2)**2
mask = sig > 1

plt.figure(figsize=(7, 3.5))
plt.subplot(1, 2, 1)
```

```
plt.imshow(sig)
plt.axis('off')
plt.title('sig')

plt.subplot(1, 2, 2)
plt.imshow(mask, cmap=plt.cm.gray)
plt.axis('off')
plt.title('mask')
plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)
```



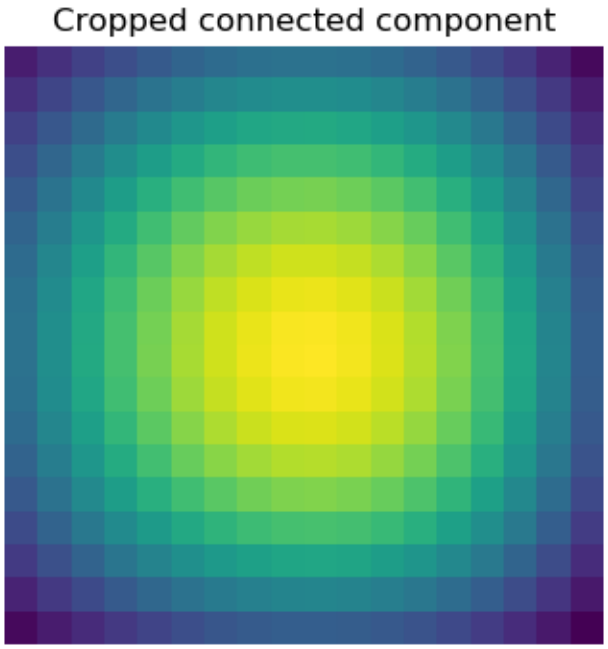
Label connected components

```
from scipy import ndimage
labels, nb = ndimage.label(mask)

plt.figure(figsize=(3.5, 3.5))
plt.imshow(labels)
plt.title('label')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)
```



Extract the 4th connected component, and crop the array around it

```
sl = ndimage.find_objects(labels==4)
plt.figure(figsize=(3.5, 3.5))
plt.imshow(sig[sl[0]])
plt.title('Cropped connected component')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.9)

plt.show()
```



Total running time of the script: (0 minutes 0.264 seconds)

3.5.10 Minima and roots of a function

Demos finding minima and roots of a function.

Define the function

```
import numpy as np

x = np.arange(-10, 10, 0.1)
def f(x):
    return x**2 + 10*np.sin(x)
```

Find minima

```
from scipy import optimize

# Global optimization
grid = (-10, 10, 0.1)
xmin_global = optimize.brute(f, (grid, ))
print("Global minima found %s" % xmin_global)

# Constrain optimization
xmin_local = optimize.fminbound(f, 0, 10)
print("Local minimum found %s" % xmin_local)
```

Out:

```
Global minima found [-1.30641113]
Local minimum found 3.8374671195
```

Root finding

```
root = optimize.root(f, 1)  # our initial guess is 1
print("First root found %s" % root.x)
root2 = optimize.root(f, -2.5)
print("Second root found %s" % root2.x)
```

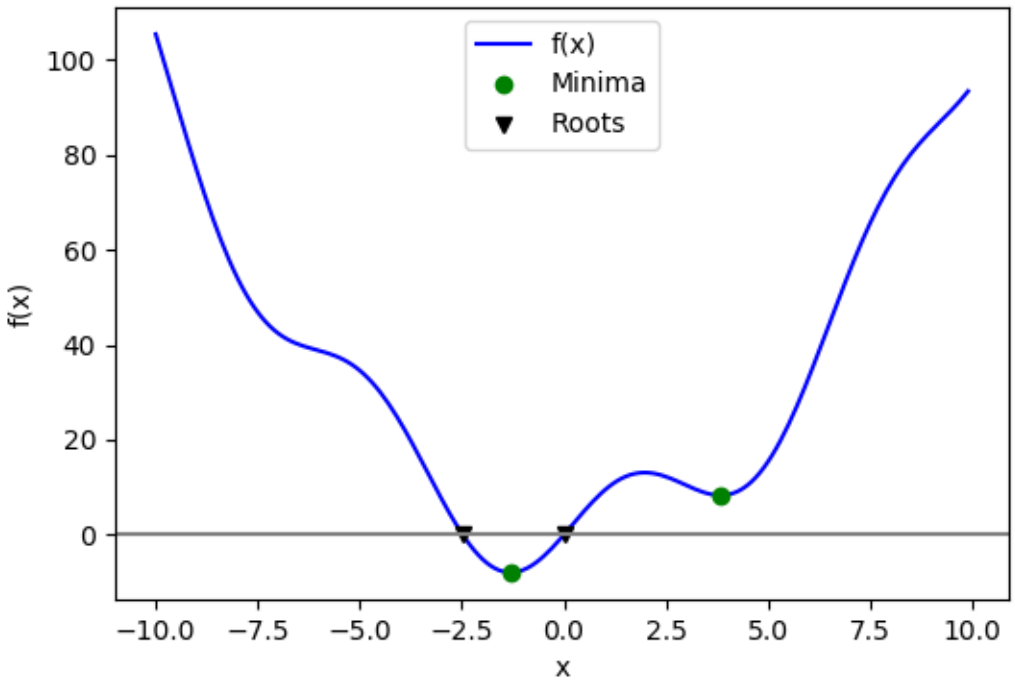
Out:

```
First root found [ 0.]
Second root found [-2.47948183]
```

Plot function, minima, and roots

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot(111)
```

```
# Plot the function
ax.plot(x, f(x), 'b-', label="f(x)")

# Plot the minima
xmins = np.array([xmin_global[0], xmin_local])
ax.plot(xmins, f(xmins), 'go', label="Minima")

# Plot the roots
roots = np.array([root.x, root2.x])
ax.plot(roots, f(roots), 'kv', label="Roots")

# Decorate the figure
ax.legend(loc='best')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.axhline(0, color='gray')
plt.show()
```



Total running time of the script: (0 minutes 0.062 seconds)

3.5.11 Optimization of a two-parameter function

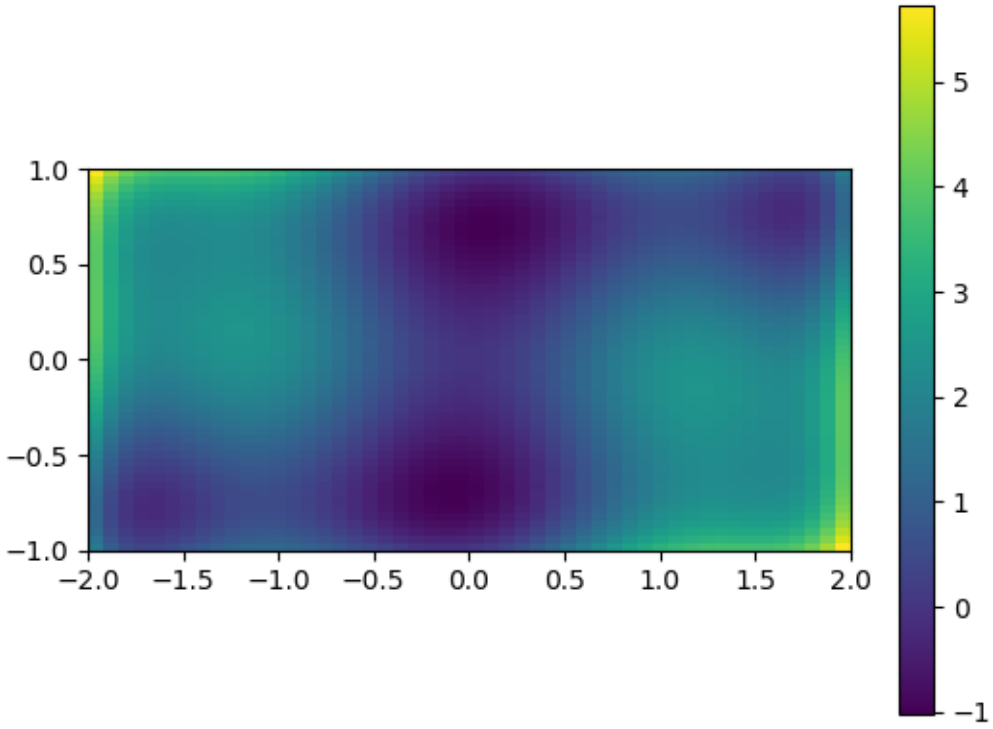
```
import numpy as np


# Define the function that we are interested in
def sixhump(x):
    return ((4 - 2.1*x[0]**2 + x[0]**4 / 3.) * x[0]**2 + x[0] * x[1]
            + (-4 + 4*x[1]**2) * x[1] **2)
```

```
# Make a grid to evaluate the function (for plotting)
x = np.linspace(-2, 2)
y = np.linspace(-1, 1)
xg, yg = np.meshgrid(x, y)
```

A 2D image plot of the function

Simple visualization in 2D

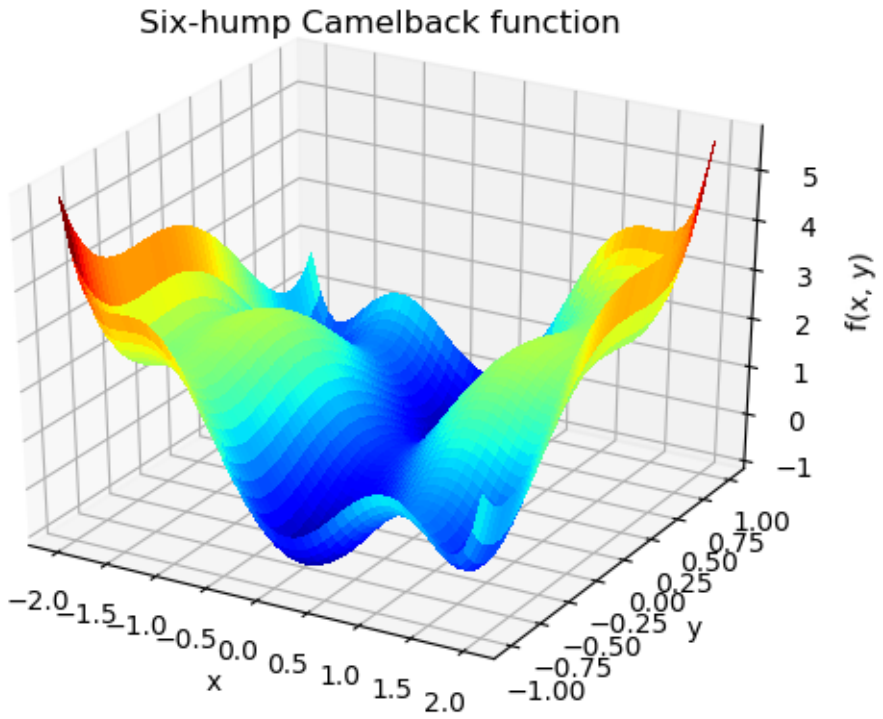
```
import matplotlib.pyplot as plt
plt.figure()
plt.imshow(sixhump([xg, yg]), extent=[-2, 2, -1, 1])
plt.colorbar()
```



A 3D surface plot of the function

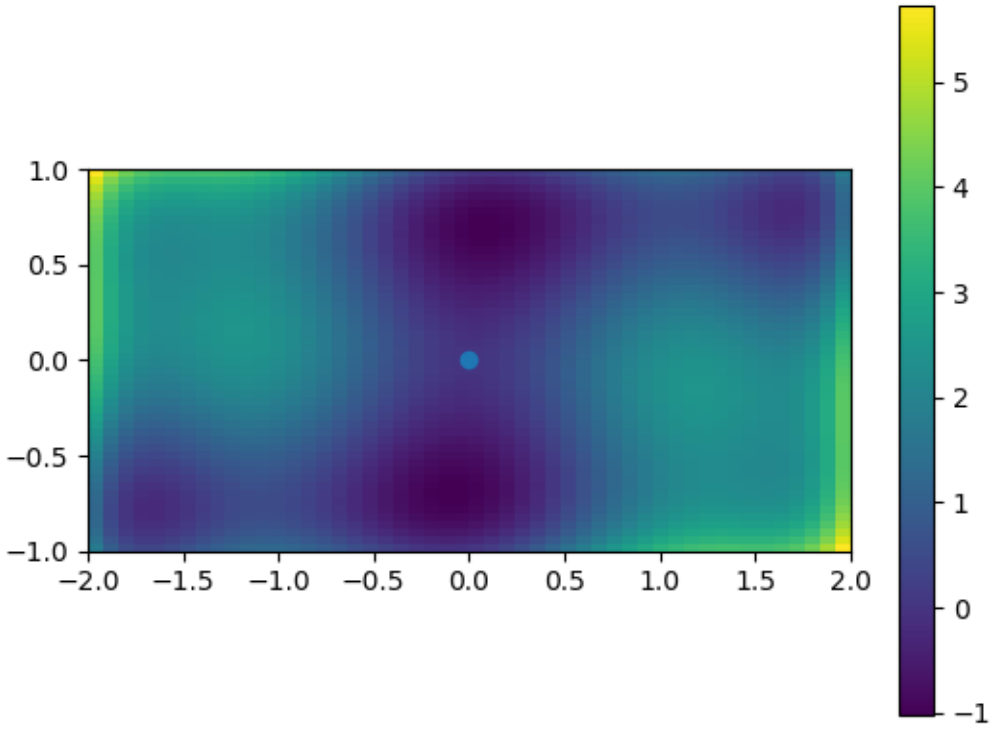
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(xg, yg, sixhump([xg, yg]), rstride=1, cstride=1,
                       cmap=plt.cm.jet, linewidth=0, antialiased=False)
```

```
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('Six-hump Camelback function')
```



Find the minima

```
from scipy import optimize

x_min = optimize.minimize(sixhump, x0=[0, 0])

plt.figure()
# Show the function in 2D
plt.imshow(sixhump([xg, yg]), extent=[-2, 2, -1, 1])
plt.colorbar()
# And the minimum that we've found:
plt.scatter(x_min.x[0], x_min.x[1])

plt.show()
```



Total running time of the script: (0 minutes 0.388 seconds)

3.5.12 Plot filtering on images

Demo filtering for denoising of images.


```
# Load some data
from scipy import misc
face = misc.face(gray=True)
face = face[:512, -512:]  # crop out square on right

# Apply a variety of filters
```

```
from scipy import ndimage
from scipy import signal
from matplotlib import pyplot as plt

import numpy as np
noisy_face = np.copy(face).astype(np.float)
noisy_face += face.std() * 0.5 * np.random.standard_normal(face.shape)
blurred_face = ndimage.gaussian_filter(noisy_face, sigma=3)
median_face = ndimage.median_filter(noisy_face, size=5)
wiener_face = signal.wiener(noisy_face, (5, 5))

plt.figure(figsize=(12, 3.5))
plt.subplot(141)
plt.imshow(noisy_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('noisy')

plt.subplot(142)
plt.imshow(blurred_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Gaussian filter')

plt.subplot(143)
plt.imshow(median_face, cmap=plt.cm.gray)
plt.axis('off')
plt.title('median filter')

plt.subplot(144)
plt.imshow(wiener_face, cmap=plt.cm.gray)
plt.title('Wiener filter')
plt.axis('off')

plt.subplots_adjust(wspace=.05, left=.01, bottom=.01, right=.99, top=.99)

plt.show()
```

Total running time of the script: (0 minutes 0.653 seconds)

CHAPTER 4

Advanced NumPy

Author: *Pauli Virtanen*

NumPy is at the base of Python's scientific stack of tools. Its purpose to implement efficient operations on many items in a block of memory. Understanding how it works in detail helps in making efficient use of its flexibility, taking useful shortcuts.

This section covers:

- Anatomy of NumPy arrays, and its consequences. Tips and tricks.
- Universal functions: what, why, and what to do if you want a new one.
- Integration with other tools: NumPy offers several ways to wrap any data in an ndarray, without unnecessary copies.
- Recently added features, and what's in them: PEP 3118 buffers, generalized ufuncs, ...

Prerequisites

- NumPy
- Cython
- Pillow (Python imaging library, used in a couple of examples)

Chapter contents

Tip: In this section, numpy will be imported as follows:

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

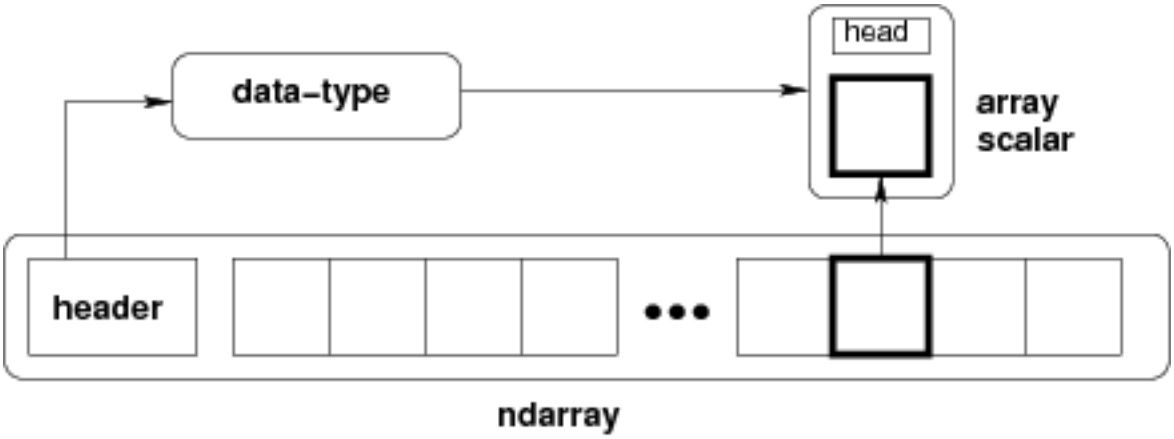
4.1 Life of ndarray

4.1.1 It's...

ndarray =

block of memory + indexing scheme + data type descriptor

- raw data
- how to locate an element
- how to interpret an element




```
typedef struct PyArrayObject {
        PyObject_HEAD

        /* Block of memory */
        char *data;

        /* Data type descriptor */
        PyArray_Descr *descr;

        /* Indexing scheme */
        int nd;
        npy_intp *dimensions;
        npy_intp *strides;

        /* Other stuff */
        PyObject *base;
        int flags;
        PyObject *weakreflist;
} PyArrayObject;
```

4.1.2 Block of memory

```
>>> x = np.array([1, 2, 3], dtype=np.int32)
>>> x.data
<... at ...>
>>> str(x.data)
'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
```

Memory address of the data:

```
>>> x.__array_interface__['data'][0]
64803824
```

The whole `__array_interface__`:

```
>>> x.__array_interface__
{'data': (35828928, False),
 'descr': [('', '<i4')],
 'shape': (4,),
 'strides': None,
 'typestr': '<i4',
 'version': 3}
```

Reminder: two `ndarrays` may share the same memory:

```
>>> x = np.array([1, 2, 3, 4])
>>> y = x[:-1]
>>> x[0] = 9
>>> y
array([9, 2, 3])
```

Memory does not need to be owned by an `ndarray`:

```
>>> x = b'1234'      # The 'b' is for "bytes", necessary in Python 3
```

x is a string (in Python 3 a bytes), we can represent its data as an array of ints:

```
>>> y = np.frombuffer(x, dtype=np.int8)
>>> y.data
<... at ...>
>>> y.base is x
True

>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : False
  ALIGNED : True
  UPDATEIFCOPY : False
```

The `owndata` and `writeable` flags indicate status of the memory block.

4.1.3 Data types

The descriptor

`dtype` describes a single item in the array:

type	**scalar type** of the data, one of: int8, int16, float64, *et al.* (fixed size) str, unicode, void (flexible size)
itemsize	**size** of the data block
byteorder	**byte order**: big-endian > / little-endian < / not applicable |
fields	sub-dtypes, if it's a **structured data type**
shape	shape of the array, if it's a **sub-array**

```
>>> np.dtype(int).type
<type 'numpy.int64'>
>>> np.dtype(int).itemsize
8
>>> np.dtype(int).byteorder
'='
```

Example: reading `.wav` files

The `.wav` file header:

chunk_id	`"RIFF"`
chunk_size	4-byte unsigned little-endian integer
format	`"WAVE"`
fmt_id	`"fmt "`
fmt_size	4-byte unsigned little-endian integer
audio_fmt	2-byte unsigned little-endian integer
num_channels	2-byte unsigned little-endian integer
sample_rate	4-byte unsigned little-endian integer
byte_rate	4-byte unsigned little-endian integer
block_align	2-byte unsigned little-endian integer
bits_per_sample	2-byte unsigned little-endian integer
data_id	`"data"`
data_size	4-byte unsigned little-endian integer

- 44-byte block of raw data (in the beginning of the file)
- ... followed by `data_size` bytes of actual sound data.

The `.wav` file header as a NumPy *structured* data type:

```
>>> wav_header_dtype = np.dtype([
...     ("chunk_id", (bytes, 4)), # flexible-sized scalar type, item size 4
...     ("chunk_size", "<u4"),    # little-endian unsigned 32-bit integer
...     ("format", "S4"),         # 4-byte string
...     ("fmt_id", "S4"),
...     ("fmt_size", "<u4"),
...     ("audio_fmt", "<u2"),     #
...     ("num_channels", "<u2"),  # .. more of the same ...
...     ("sample_rate", "<u4"),   #
...     ("byte_rate", "<u4"),
...     ("block_align", "<u2"),
...     ("bits_per_sample", "<u2"),
...     ("data_id", ("S1", (2, 2))), # sub-array, just for fun!
...     ("data_size", "u4"),
...     #
```

```
...     # the sound data itself cannot be represented here:
...     # it does not have a fixed size
...    ])
```

See also:

wavreader.py

```
>>> wav_header_dtype['format']
dtype('S4')
>>> wav_header_dtype.fields
dict_proxy({'block_align': (dtype('uint16'), 32), 'format': (dtype('S4'), 8), 'data_id': (dtype((
↪'S1', (2, 2))), 36), 'fmt_id': (dtype('S4'), 12), 'byte_rate': (dtype('uint32'), 28), 'chunk_id
↪': (dtype('S4'), 0), 'num_channels': (dtype('uint16'), 22), 'sample_rate': (dtype('uint32'), 24),
↪ 'bits_per_sample': (dtype('uint16'), 34), 'chunk_size': (dtype('uint32'), 4), 'fmt_size':␣
↪(dtype('uint32'), 16), 'data_size': (dtype('uint32'), 40), 'audio_fmt': (dtype('uint16'), 20)})
>>> wav_header_dtype.fields['format']
(dtype('S4'), 8)
```

- The first element is the sub-dtype in the structured data, corresponding to the name `format`
- The second one is its offset (in bytes) from the beginning of the item

Exercise

Mini-exercise, make a "sparse" dtype by using offsets, and only some of the fields:

```
>>> wav_header_dtype = np.dtype(dict(
...   names=['format', 'sample_rate', 'data_id'],
...   offsets=[offset_1, offset_2, offset_3], # counted from start of structure in bytes
...   formats=list of dtypes for each of the fields,
... ))
```

and use that to read the sample rate, and `data_id` (as sub-array).

```
>>> f = open('data/test.wav', 'r')
>>> wav_header = np.fromfile(f, dtype=wav_header_dtype, count=1)
>>> f.close()
>>> print(wav_header)
[ ('RIFF', 17402L, 'WAVE', 'fmt ', 16L, 1, 1, 16000L, 32000L, 2, 16, [['d', 'a'], ['t', 'a']],␣
↪17366L)]
>>> wav_header['sample_rate']
array([16000], dtype=uint32)
```

Let's try accessing the sub-array:

```
>>> wav_header['data_id']
array([[['d', 'a'],
        ['t', 'a']]],
      dtype='|S1')
>>> wav_header.shape
(1,)
>>> wav_header['data_id'].shape
(1, 2, 2)
```

When accessing sub-arrays, the dimensions get added to the end!

Note: There are existing modules such as `wavfile`, `audiolab`, etc. for loading sound data...

Casting and re-interpretation/views

casting

- on assignment
- on array construction
- on arithmetic
- etc.
- and manually: `.astype(dtype)`

data re-interpretation

- manually: `.view(dtype)`

Casting

- Casting in arithmetic, in nutshell:
  - only type (not value!) of operands matters
  - largest "safe" type able to represent both is picked
  - scalars can "lose" to arrays in some situations
- Casting in general copies data:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.float)
>>> x
array([ 1.,  2.,  3.,  4.])
>>> y = x.astype(np.int8)
>>> y
array([1, 2, 3, 4], dtype=int8)
>>> y + 1
array([2, 3, 4, 5], dtype=int8)
>>> y + 256
array([257, 258, 259, 260], dtype=int16)
>>> y + 256.0
array([ 257.,  258.,  259.,  260.])
>>> y + np.array([256], dtype=np.int32)
array([257, 258, 259, 260], dtype=int32)
```

- Casting on setitem: dtype of the array is not changed on item assignment:

```
>>> y[:] = y + 1.5
>>> y
array([2, 3, 4, 5], dtype=int8)
```

Note: Exact rules: see numpy documentation

Re-interpretation / viewing

- Data block in memory (4 bytes)

0x01	0x02	0x03	0x04

  - 4 of uint8, OR,
  - 4 of int8, OR,
  - 2 of int16, OR,
  - 1 of int32, OR,
  - 1 of float32, OR,
  - ...

  How to switch from one to another?

1. Switch the dtype:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.uint8)
>>> x.dtype = "<i2"
>>> x
array([ 513, 1027], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
```

0x01	0x02	0x03	0x04

Note: little-endian: least significant byte is on the *left* in memory

---

2. Create a new view:

```
>>> y = x.view("<i4")
>>> y
array([67305985], dtype=int32)
>>> 0x04030201
67305985
```

0x01	0x02	0x03	0x04

Note:

- `.view()` makes *views*, does not copy (or alter) the memory block
- only changes the dtype (and adjusts array shape):

```
>>> x[1] = 5
>>> y
array([328193], dtype=int32)
>>> y.base is x
True
```

Mini-exercise: data re-interpretation

See also:

view-colors.py

You have RGBA data in an array:

```
>>> x = np.zeros((10, 10, 4), dtype=np.int8)
>>> x[:, :, 0] = 1
>>> x[:, :, 1] = 2
>>> x[:, :, 2] = 3
>>> x[:, :, 3] = 4
```

where the last three dimensions are the R, B, and G, and alpha channels.

How to make a (10, 10) structured array with field names 'r', 'g', 'b', 'a' without copying data?

```
>>> y = ...

>>> assert (y['r'] == 1).all()
>>> assert (y['g'] == 2).all()
>>> assert (y['b'] == 3).all()
>>> assert (y['a'] == 4).all()
```

Solution

```
>>> y = x.view([('r', 'i1'),
...             ('g', 'i1'),
...             ('b', 'i1'),
...             ('a', 'i1')]
...              )[:, :, 0]
```

Warning: Another array taking exactly 4 bytes of memory:

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
>>> x
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> y
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> x.view(np.int16)
array([[ 513],
       [1027]], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
>>> y.view(np.int16)
array([[ 769, 1026]], dtype=int16)
```

- What happened?
- ... we need to look into what `x[0,1]` actually means

```
>>> 0x0301, 0x0402
(769, 1026)
```

4.1.4 Indexing scheme: strides

Main point

The question:

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int8)
>>> str(x.data)
'\x01\x02\x03\x04\x05\x06\x07\x08\t'

At which byte in ``x.data`` does the item ``x[1, 2]`` begin?
```

The answer (in NumPy)

- **strides**: the number of bytes to jump to find the next element
- 1 stride per dimension

```
  >>> x.strides
  (3, 1)
  >>> byte_offset = 3*1 + 1*2   # to find x[1, 2]
  >>> x.flat[byte_offset]
  6
  >>> x[1, 2]
  6

- simple, **flexible**
```

C and Fortran order

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6]], dtype=np.int16, order='C')
>>> x.strides
(6, 2)
>>> str(x.data)
'\x01\x00\x02\x00\x03\x00\x04\x00\x05\x00\x06\x00'
```

- Need to jump 6 bytes to find the next row
- Need to jump 2 bytes to find the next column

```
>>> y = np.array(x, order='F')
>>> y.strides
(2, 4)
>>> str(y.data)
'\x01\x00\x04\x00\x02\x00\x05\x00\x03\x00\x06\x00'
```

- Need to jump 2 bytes to find the next row
- Need to jump 4 bytes to find the next column
- Similarly to higher dimensions:
  - C: last dimensions vary fastest (= smaller strides)
  - F: first dimensions vary fastest

$$\text{shape} = (d_1, d_2, ..., d_n)$$
$$\text{strides} = (s_1, s_2, ..., s_n)$$
$$s_j^C = d_{j+1} d_{j+2} ... d_n \times \text{itemsize}$$
$$s_j^F = d_1 d_2 ... d_{j-1} \times \text{itemsize}$$

Note: Now we can understand the behavior of `.view()`:

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
```

Transposition does not affect the memory layout of the data, only strides

```
>>> x.strides
(2, 1)
>>> y.strides
(1, 2)
```

```
>>> str(x.data)
'\x01\x02\x03\x04'
>>> str(y.data)
'\x01\x03\x02\x04'
```

- the results are different when interpreted as 2 of int16
- `.copy()` creates new arrays in the C order (by default)

Slicing with integers

- *Everything* can be represented by changing only `shape`, `strides`, and possibly adjusting the `data` pointer!
- Never makes copies of the data

```
>>> x = np.array([1, 2, 3, 4, 5, 6], dtype=np.int32)
>>> y = x[::-1]
>>> y
array([6, 5, 4, 3, 2, 1], dtype=int32)
>>> y.strides
(-4,)

>>> y = x[2:]
>>> y.__array_interface__['data'][0] - x.__array_interface__['data'][0]
8

>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
>>> x[::2,::3,::4].strides
(1600, 240, 32)
```

- Similarly, transposes never make copies (it just swaps strides):

```
>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
```

```
>>> x.T.strides
(8, 80, 800)
```

But: not all reshaping operations can be represented by playing with strides:

```
>>> a = np.arange(6, dtype=np.int8).reshape(3, 2)
>>> b = a.T
>>> b.strides
(1, 2)
```

So far, so good. However:

```
>>> str(a.data)
'\x00\x01\x02\x03\x04\x05'
>>> b
array([[0, 2, 4],
       [1, 3, 5]], dtype=int8)
>>> c = b.reshape(3*2)
>>> c
array([0, 2, 4, 1, 3, 5], dtype=int8)
```

Here, there is no way to represent the array `c` given one stride and the block of memory for `a`. Therefore, the `reshape` operation needs to make a copy here.

Example: fake dimensions with strides

Stride manipulation

```
>>> from numpy.lib.stride_tricks import as_strided
>>> help(as_strided)
as_strided(x, shape=None, strides=None)
   Make an ndarray from the given array with the given shape and strides
```

Warning: `as_strided` does **not** check that you stay inside the memory block bounds...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> as_strided(x, strides=(2*2, ), shape=(2, ))
array([1, 3], dtype=int16)
>>> x[::2]
array([1, 3], dtype=int16)
```

See also:

stride-fakedims.py

Exercise

```
array([1, 2, 3, 4], dtype=np.int8)

-> array([[1, 2, 3, 4],
          [1, 2, 3, 4],
          [1, 2, 3, 4]], dtype=np.int8)
```

using only `as_strided`.:

```
Hint: byte_offset = stride[0]*index[0] + stride[1]*index[1] + ...
```

Spoiler

Stride can also be *0*:

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int8)
>>> y = as_strided(x, strides=(0, 1), shape=(3, 4))
>>> y
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int8)
>>> y.base.base is x
True
```

Broadcasting

- Doing something useful with it: outer product of `[1, 2, 3, 4]` and `[5, 6, 7]`

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> x2 = as_strided(x, strides=(0, 1*2), shape=(3, 4))
>>> x2
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int16)
```

```
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> y2 = as_strided(y, strides=(1*2, 0), shape=(3, 4))
>>> y2
array([[5, 5, 5, 5],
       [6, 6, 6, 6],
       [7, 7, 7, 7]], dtype=int16)
```

```
>>> x2 * y2
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

... seems somehow familiar ...

```
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> x[np.newaxis,:] * y[:,np.newaxis]
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

- Internally, array **broadcasting** is indeed implemented using 0-strides.

More tricks: diagonals

See also:

stride-diagonals.py

Challenge

- Pick diagonal entries of the matrix: (assume C memory order):

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int32)

>>> x_diag = as_strided(x, shape=(3,), strides=(???,))
```

- Pick the first super-diagonal entries `[2, 6]`.
- And the sub-diagonals?

(Hint to the last two: slicing first moves the point where striding starts from.)

Solution

Pick diagonals:

```
>>> x_diag = as_strided(x, shape=(3, ), strides=((3+1)*x.itemsize, ))
>>> x_diag
array([1, 5, 9], dtype=int32)
```

Slice first, to adjust the data pointer:

```
>>> as_strided(x[0, 1:], shape=(2, ), strides=((3+1)*x.itemsize, ))
array([2, 6], dtype=int32)

>>> as_strided(x[1:, 0], shape=(2, ), strides=((3+1)*x.itemsize, ))
array([4, 8], dtype=int32)
```

Note: Using np.diag

```
>>> y = np.diag(x, k=1)
>>> y
array([2, 6], dtype=int32)
```

However,

```
>>> y.flags.owndata
False
```

Note This behavior has changed: before numpy 1.9, np.diag would make a copy.

See also:

stride-diagonals.py

Challenge

Compute the tensor trace:

```
>>> x = np.arange(5*5*5*5).reshape(5, 5, 5, 5)
>>> s = 0
>>> for i in range(5):
...     for j in range(5):
...         s += x[j, i, j, i]
```

by striding, and using `sum()` on the result.

```
>>> y = as_strided(x, shape=(5, 5), strides=(TODO, TODO))
>>> s2 = ...
>>> assert s == s2
```

Solution

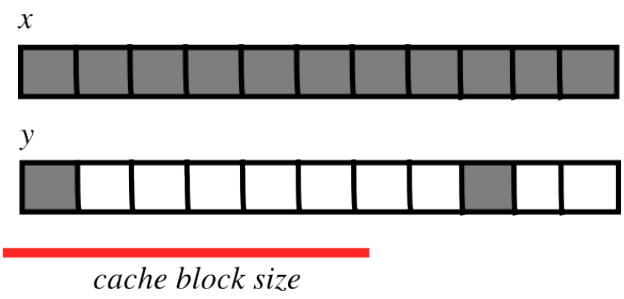
```
>>> y = as_strided(x, shape=(5, 5), strides=((5*5*5 + 5)*x.itemsize,
...                                          (5*5 + 1)*x.itemsize))
>>> s2 = y.sum()
```

CPU cache effects

Memory layout can affect performance:

```
In [1]: x = np.zeros((20000,))

In [2]: y = np.zeros((20000*67,))[::67]

In [3]: x.shape, y.shape
((20000,), (20000,))

In [4]: %timeit x.sum()
100000 loops, best of 3: 0.180 ms per loop

In [5]: %timeit y.sum()
100000 loops, best of 3: 2.34 ms per loop

In [6]: x.strides, y.strides
((8,), (536,))
```

Smaller strides are faster?



- CPU pulls data from main memory to its cache in blocks
- If many array items consecutively operated on fit in a single block (small stride):
  - ⇒ fewer transfers needed
  - ⇒ faster

See also:

numexpr is designed to mitigate cache effects in array computing.

Example: inplace operations (caveat emptor)

- Sometimes,

```
>>> a -= b
```

  is not the same as

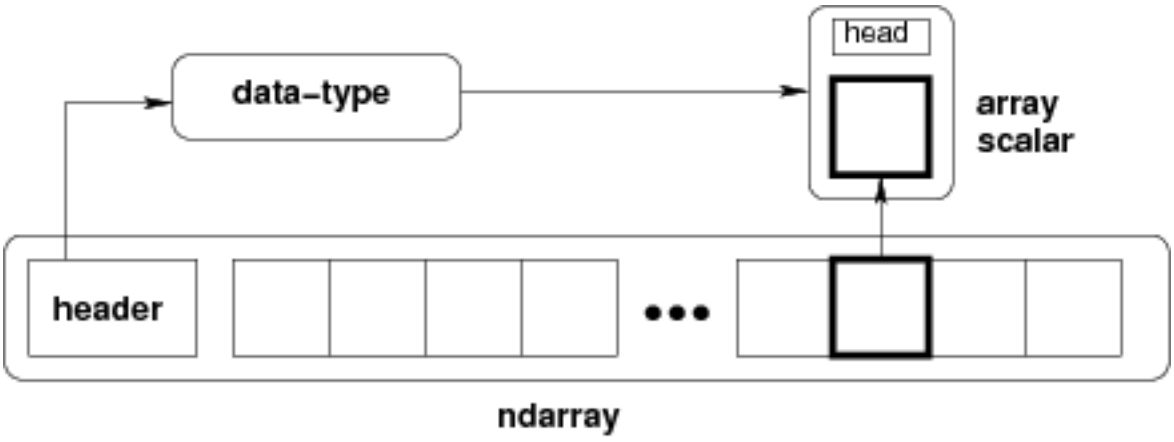
```
>>> a -= b.copy()
```

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x -= x.transpose()
>>> x
array([[ 0, -1],
       [ 4,  0]])
```

```
>>> y = np.array([[1, 2], [3, 4]])
>>> y -= y.T.copy()
>>> y
array([[ 0, -1],
       [ 1,  0]])
```

- `x` and `x.transpose()` share data
- `x -= x.transpose()` modifies the data element-by-element...
- because `x` and `x.transpose()` have different striding, modified data re-appears on the RHS

4.1.5 Findings in dissection



- *memory block*: may be shared, `.base`, `.data`
- *data type descriptor*: structured data, sub-arrays, byte order, casting, viewing, `.astype()`, `.view()`
- *strided indexing*: strides, C/F-order, slicing w/ integers, `as_strided`, broadcasting, stride tricks, `diag`, CPU cache coherence

4.2 Universal functions

4.2.1 What they are?

- Ufunc performs and elementwise operation on all elements of an array.

  Examples:

```
np.add, np.subtract, scipy.special.*, ...
```

- Automatically support: broadcasting, casting, ...
- The author of an ufunc only has to supply the elementwise operation, NumPy takes care of the rest.
- The elementwise operation needs to be implemented in C (or, e.g., Cython)

Parts of an Ufunc

1. Provided by user

```
void ufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    /*
     * int8 output = elementwise_function(int8 input_1, int8 input_2)
     *
     * This function must compute the ufunc for many values at once,
     * in the way shown below.
     */
    char *input_1 = (char*)args[0];
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];
    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        *output = elementwise_function(*input_1, *input_2);
        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

2. The NumPy part, built by

```
char types[3]

types[0] = NPY_BYTE   /* type of first input arg */
types[1] = NPY_BYTE   /* type of second input arg */
types[2] = NPY_BYTE   /* type of third input arg */

PyObject *python_ufunc = PyUFunc_FromFuncAndData(
    ufunc_loop,
    NULL,
    types,
    1, /* ntypes */
    2, /* num_inputs */
    1, /* num_outputs */
    identity_element,
```

```
    name,
    docstring,
    unused)
```

- A ufunc can also support multiple different input-output type combinations.

Making it easier

3. `ufunc_loop` is of very generic form, and NumPy provides pre-made ones

`PyUfunc_f_f`	`float elementwise_func(float input_1)`
`PyUfunc_ff_f`	`float elementwise_func(float input_1, float input_2)`
`PyUfunc_d_d`	`double elementwise_func(double input_1)`
`PyUfunc_dd_d`	`double elementwise_func(double input_1, double input_2)`
`PyUfunc_D_D`	`elementwise_func(npy_cdouble *input, npy_cdouble* output)`
`PyUfunc_DD_D`	`elementwise_func(npy_cdouble *in1, npy_cdouble *in2, npy_cdouble* out)`

- Only `elementwise_func` needs to be supplied
- ... except when your elementwise function is not in one of the above forms

4.2.2 Exercise: building an ufunc from scratch

The Mandelbrot fractal is defined by the iteration

$$z \leftarrow z^2 + c$$

where $c = x + iy$ is a complex number. This iteration is repeated – if z stays finite no matter how long the iteration runs, c belongs to the Mandelbrot set.

- Make ufunc called `mandel(z0, c)` that computes:

```
z = z0
for k in range(iterations):
    z = z*z + c
```

  say, 100 iterations or until `z.real**2 + z.imag**2 > 1000`. Use it to determine which c are in the Mandelbrot set.

- Our function is a simple one, so make use of the `PyUFunc_*` helpers.
- Write it in Cython

See also:

mandel.pyx, mandelplot.py

```
#
# Fix the parts marked by TODO
#

#
# Compile this file by (Cython >= 0.12 required because of the complex vars)
#
#    cython mandel.pyx
```

```
#    python setup.py build_ext -i
#
# and try it out with, in this directory,
#
#    >>> import mandel
#    >>> mandel.mandel(0, 1 + 2j)
#
#

# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    #
    # TODO: write the Mandelbrot iteration for one point here,
    #       as you would write it in Python.
    #
    #       Say, use 100 as the maximum number of iterations, and 1000
    #       as the cutoff for z.real**2 + z.imag**2.
    #

    TODO: mandelbrot iteration should go here

    # Return the answer for this point
    z_out[0] = z


# Boilerplate Cython definitions
#
# Pulls definitions from the Numpy C headers.
# -------------------------------------------

from numpy cimport import_array, import_ufunc
from numpy cimport (PyUFunc_FromFuncAndData,
                    PyUFuncGenericFunction)
```

```
from numpy cimport NPY_CDOUBLE, NP_DOUBLE, NPY_LONG

# Import all pre-defined loop functions
# you won't need all of them - keep the relevant ones

from numpy cimport (
    PyUFunc_f_f_As_d_d,
    PyUFunc_d_d,
    PyUFunc_f_f,
    PyUFunc_g_g,
    PyUFunc_F_F_As_D_D,
    PyUFunc_F_F,
    PyUFunc_D_D,
    PyUFunc_G_G,
    PyUFunc_ff_f_As_dd_d,
    PyUFunc_ff_f,
    PyUFunc_dd_d,
    PyUFunc_gg_g,
    PyUFunc_FF_F_As_DD_D,
    PyUFunc_DD_D,
    PyUFunc_FF_F,
    PyUFunc_GG_G)


# Required module initialization
# ------------------------------

import_array()
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

#
# Reminder: some pre-made Ufunc loops:
#
# ================  =======================================================
# ``PyUfunc_f_f``   ``float elementwise_func(float input_1)``
# ``PyUfunc_ff_f``  ``float elementwise_func(float input_1, float input_2)``
# ``PyUfunc_d_d``   ``double elementwise_func(double input_1)``
# ``PyUfunc_dd_d``  ``double elementwise_func(double input_1, double input_2)``
# ``PyUfunc_D_D``   ``elementwise_func(complex_double *input, complex_double* complex_double)``
# ``PyUfunc_DD_D``  ``elementwise_func(complex_double *in1, complex_double *in2, complex_double*␣
 ↪out)``
# ================  =======================================================
#
# The full list is above.
#
#
# Type codes:
#
# NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
# NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
```

```
# NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
# NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
#

loop_func[0] = ... TODO: suitable PyUFunc_* ...
input_output_types[0] = ... TODO ...
... TODO: fill in rest of input_output_types ...

# This thing is passed as the ``data`` parameter for the generic
# PyUFunc_* loop, to let it know which function it should call.
elementwise_funcs[0] = <void*>mandel_single_point

# Construct the ufunc:

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    TODO, # number of input args
    TODO, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes z*z + c", # docstring
    0 # unused
    )
```

Reminder: some pre-made Ufunc loops:

PyUfunc_f_f	float elementwise_func(float input_1)
PyUfunc_ff_f	float elementwise_func(float input_1, float input_2)
PyUfunc_d_d	double elementwise_func(double input_1)
PyUfunc_dd_d	double elementwise_func(double input_1, double input_2)
PyUfunc_D_D	elementwise_func(complex_double *input, complex_double* output)
PyUfunc_DD_D	elementwise_func(complex_double *in1, complex_double *in2, complex_double* out)

Type codes:

```
NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_DATETIME,
NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID
```

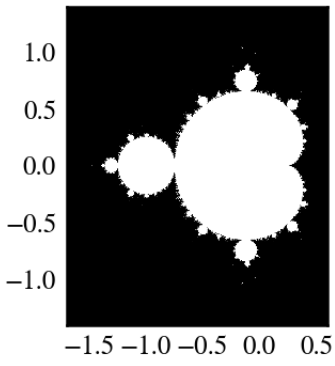
4.2.3 Solution: building an ufunc from scratch

```
# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
```

```
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
    #
    # - Note also that this function receives *pointers* to the data;
    #   the "traditional" solution to passing complex variables around
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    # Straightforward iteration

    for k in range(100):
        z = z*z + c
        if z.real**2 + z.imag**2 > 1000:
            break

    # Return the answer for this point
    z_out[0] = z


# Boilerplate Cython definitions
#
# Pulls definitions from the Numpy C headers.
# -------------------------------------------

from numpy cimport import_array, import_ufunc
from numpy cimport (PyUFunc_FromFuncAndData,
                    PyUFuncGenericFunction)
from numpy cimport NPY_CDOUBLE
from numpy cimport PyUFunc_DD_D

# Required module initialization
# ------------------------------

import_array()
import_ufunc()


# The actual ufunc declaration
# ----------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

loop_func[0] = PyUFunc_DD_D
```

```
input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE

elementwise_funcs[0] = <void*>mandel_single_point

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

```
"""
Plot Mandelbrot
===============

Plot the Mandelbrot ensemble.

"""

import numpy as np
import mandel
x = np.linspace(-1.7, 0.6, 1000)
y = np.linspace(-1.4, 1.4, 1000)
c = x[None,:] + 1j*y[:,None]
z = mandel.mandel(c, c)

import matplotlib.pyplot as plt
plt.imshow(abs(z)**2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])
plt.gray()
plt.show()
```



Note: Most of the boilerplate could be automated by these Cython modules:

http://wiki.cython.org/MarkLodato/CreatingUfuncs

Several accepted input types

E.g. supporting both single- and double-precision versions

```
cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
   ...

cdef void mandel_single_point_singleprec(float complex *z_in,
                                         float complex *c_in,
                                         float complex *z_out) nogil:
   ...

cdef PyUFuncGenericFunction loop_funcs[2]
cdef char input_output_types[3*2]
cdef void *elementwise_funcs[1*2]

loop_funcs[0] = PyUFunc_DD_D
input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE
elementwise_funcs[0] = <void*>mandel_single_point

loop_funcs[1] = PyUFunc_FF_F
input_output_types[3] = NPY_CFLOAT
input_output_types[4] = NPY_CFLOAT
input_output_types[5] = NPY_CFLOAT
elementwise_funcs[1] = <void*>mandel_single_point_singleprec

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    2, # number of supported input types   <----------------
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

4.2.4 Generalized ufuncs

ufunc

`output = elementwise_function(input)`

Both `output` and `input` can be a single array element only.

generalized ufunc

`output` and `input` can be arrays with a fixed number of dimensions

For example, matrix trace (sum of diag elements):

```
input shape = (n, n)
output shape = ()      i.e.  scalar

(n, n) -> ()
```

Matrix product:

```
input_1 shape = (m, n)
input_2 shape = (n, p)
output shape  = (m, p)

(m, n), (n, p) -> (m, p)
```

- This is called the *"signature"* of the generalized ufunc
- The dimensions on which the g-ufunc acts, are *"core dimensions"*

Status in NumPy

- g-ufuncs are in NumPy already ...
- new ones can be created with `PyUFunc_FromFuncAndDataAndSignature`
- most linear-algebra functions are implemented as g-ufuncs to enable working with stacked arrays:

```
>>> import numpy as np
>>> np.linalg.det(np.random.rand(3, 5, 5))
array([ 0.00965823, -0.13344729,  0.04583961])
>>> np.linalg._umath_linalg.det.signature
'(m,m)->()'
```

- we also ship with a few g-ufuncs for testing, ATM:

```
>>> import numpy.core.umath_tests as ut
>>> ut.matrix_multiply.signature
'(m,n),(n,p)->(m,p)'

>>> x = np.ones((10, 2, 4))
>>> y = np.ones((10, 4, 5))
>>> ut.matrix_multiply(x, y).shape
(10, 2, 5)
```

- in both examples the last two dimensions became *core dimensions*, and are modified as per the *signature*
- otherwise, the g-ufunc operates "elementwise"
- matrix multiplication this way could be useful for operating on many small matrices at once

Generalized ufunc loop

Matrix multiplication `(m,n),(n,p) -> (m,p)`

```
void gufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    char *input_1 = (char*)args[0];  /* these are as previously */
```

```
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];

    int input_1_stride_m = steps[3];  /* strides for the core dimensions */
    int input_1_stride_n = steps[4];  /* are added after the non-core */
    int input_2_strides_n = steps[5]; /* steps */
    int input_2_strides_p = steps[6];
    int output_strides_n = steps[7];
    int output_strides_p = steps[8];

    int m = dimension[1]; /* core dimensions are added after */
    int n = dimension[2]; /* the main dimension; order as in */
    int p = dimension[3]; /* signature */

    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        matmul_for_strided_matrices(input_1, input_2, output,
                                    strides for each array...);

        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

4.3 Interoperability features

4.3.1 Sharing multidimensional, typed data

Suppose you

1. Write a library than handles (multidimensional) binary data,
2. Want to make it easy to manipulate the data with NumPy, or whatever other library,
3. ... but would **not** like to have NumPy as a dependency.

Currently, 3 solutions:

1. the "old" buffer interface
2. the array interface
3. the "new" buffer interface (**PEP 3118**)

4.3.2 The old buffer protocol

- Only 1-D buffers
- No data type information
- C-level interface; `PyBufferProcs tp_as_buffer` in the type object
- But it's integrated into Python (e.g. strings support it)

Mini-exercise using Pillow (Python Imaging Library):

See also:

pilbuffer.py

```
>>> from PIL import Image
>>> data = np.zeros((200, 200, 4), dtype=np.int8)
>>> data[:, :] = [255, 0, 0, 255] # Red
>>> # In PIL, RGBA images consist of 32-bit integers whose bytes are [RR,GG,BB,AA]
>>> data = data.view(np.int32).squeeze()
>>> img = Image.frombuffer("RGBA", (200, 200), data, "raw", "RGBA", 0, 1)
>>> img.save('test.png')
```

Q:

Check what happens if `data` is now modified, and `img` saved again.

4.3.3 The old buffer protocol

```
"""
From buffer
============

Show how to exchange data between numpy and a library that only knows
the buffer interface.
"""

import numpy as np
import Image

# Let's make a sample image, RGBA format

x = np.zeros((200, 200, 4), dtype=np.int8)

x[:,:,0] = 254 # red
x[:,:,3] = 255 # opaque

data = x.view(np.int32) # Check that you understand why this is OK!

img = Image.frombuffer("RGBA", (200, 200), data)
img.save('test.png')

#
# Modify the original data, and save again.
#
# It turns out that PIL, which knows next to nothing about Numpy,
# happily shares the same data.
#

x[:,:,1] = 254
img.save('test2.png')
```

4.3.4 Array interface protocol

- Multidimensional buffers
- Data type information present
- NumPy-specific approach; slowly deprecated (but not going away)
- Not integrated in Python otherwise

See also:

Documentation: http://docs.scipy.org/doc/numpy/reference/arrays.interface.html

```
>>> x = np.array([[1, 2], [3, 4]])
>>> x.__array_interface__
{'data': (171694552, False),      # memory address of data, is readonly?
 'descr': [('', '<i4')],          # data type descriptor
 'typestr': '<i4',                # same, in another form
 'strides': None,                 # strides; or None if in C-order
 'shape': (2, 2),
```

```
 'version': 3,
}
```

::

```
>>> from PIL import Image
>>> img = Image.open('data/test.png')
>>> img.__array_interface__
{'data': ...,
 'shape': (200, 200, 4),
 'typestr': '|u1'}
>>> x = np.asarray(img)
>>> x.shape
(200, 200, 4)
```

Note: A more C-friendly variant of the array interface is also defined.

4.4 Array siblings: `chararray`, `maskedarray`, `matrix`

4.4.1 `chararray`: vectorized string operations

```
>>> x = np.array(['a', '  bbb', '  ccc']).view(np.chararray)
>>> x.lstrip(' ')
chararray(['a', 'bbb', 'ccc'],
      dtype='...')
>>> x.upper()
chararray(['A', '  BBB', '  CCC'],
      dtype='...')
```

Note: `.view()` has a second meaning: it can make an ndarray an instance of a specialized ndarray subclass

---

4.4.2 `masked_array` missing data

Masked arrays are arrays that may have missing or invalid entries.

For example, suppose we have an array where the fourth entry is invalid:

```
>>> x = np.array([1, 2, 3, -99, 5])
```

One way to describe this is to create a masked array:

```
>>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx
masked_array(data = [1 2 3 -- 5],
             mask = [False False False  True False],
       fill_value = 999999)
```

Masked mean ignores masked data:

```
>>> mx.mean()
2.75
>>> np.mean(mx)
2.75
```

Warning: Not all NumPy functions respect masks, for instance `np.dot`, so check the return types.

The `masked_array` returns a **view** to the original array:

```
>>> mx[1] = 9
>>> x
array([  1,   9,   3, -99,   5])
```

The mask

You can modify the mask by assigning:

```
>>> mx[1] = np.ma.masked
>>> mx
masked_array(data = [1 -- 3 -- 5],
             mask = [False  True False  True False],
       fill_value = 999999)
```

The mask is cleared on assignment:

```
>>> mx[1] = 9
>>> mx
masked_array(data = [1 9 3 -- 5],
             mask = [False False False  True False],
       fill_value = 999999)
```

The mask is also available directly:

```
>>> mx.mask
array([False, False, False,  True, False], dtype=bool)
```

The masked entries can be filled with a given value to get an usual array back:

```
>>> x2 = mx.filled(-1)
>>> x2
array([ 1,  9,  3, -1,  5])
```

The mask can also be cleared:

```
>>> mx.mask = np.ma.nomask
>>> mx
masked_array(data = [1 9 3 -99 5],
             mask = [False False False False False],
       fill_value = 999999)
```

Domain-aware functions

The masked array package also contains domain-aware functions:

```
>>> np.ma.log(np.array([1, 2, -1, -2, 3, -5]))
masked_array(data = [0.0 0.6931471805599453 -- -- 1.0986122886681098 --],
             mask = [False False  True  True False  True],
       fill_value = 1e+20)
```

Note: Streamlined and more seamless support for dealing with missing data in arrays is making its way into NumPy 1.7. Stay tuned!

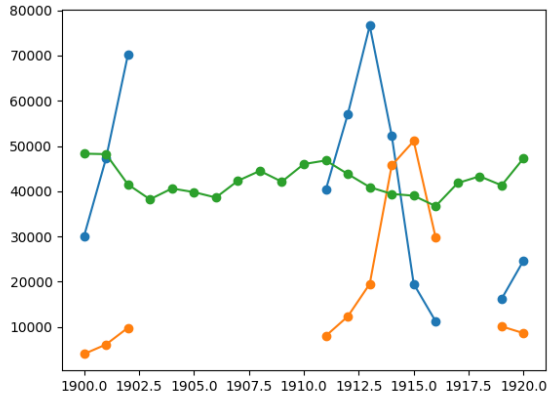
Example: Masked statistics

Canadian rangers were distracted when counting hares and lynxes in 1903-1910 and 1917-1918, and got the numbers are wrong. (Carrot farmers stayed alert, though.) Compute the mean populations over time, ignoring the invalid numbers.

```
>>> data = np.loadtxt('data/populations.txt')
>>> populations = np.ma.masked_array(data[:,1:])
>>> year = data[:, 0]

>>> bad_years = (((year >= 1903) & (year <= 1910))
...              | ((year >= 1917) & (year <= 1918)))
>>> # '&' means 'and' and '|' means 'or'
>>> populations[bad_years, 0] = np.ma.masked
>>> populations[bad_years, 1] = np.ma.masked

>>> populations.mean(axis=0)
masked_array(data = [40472.72727272727 18627.272727272728 42400.0],
             mask = [False False False],
       fill_value = 1e+20)

>>> populations.std(axis=0)
masked_array(data = [21087.656489006717 15625.799814240254 3322.5062255844787],
             mask = [False False False],
       fill_value = 1e+20)
```

Note that Matplotlib knows about masked arrays:

```
>>> plt.plot(year, populations, 'o-')
[<matplotlib.lines.Line2D object at ...>, ...]
```



4.4.3 `recarray`: purely convenience

```
>>> arr = np.array([('a', 1), ('b', 2)], dtype=[('x', 'S1'), ('y', int)])
>>> arr2 = arr.view(np.recarray)
>>> arr2.x
chararray(['a', 'b'],
      dtype='|S1')
>>> arr2.y
array([1, 2])
```

4.4.4 `matrix`: convenience?

- always 2-D
- `*` is the matrix product, not the elementwise one

```
>>> np.matrix([[1, 0], [0, 1]]) * np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

4.5 Summary

- Anatomy of the ndarray: data, dtype, strides.
- Universal functions: elementwise operations, how to make new ones
- Ndarray subclasses
- Various buffer interfaces for integration with other tools
- Recent additions: PEP 3118, generalized ufuncs

4.6 Contributing to NumPy/Scipy

Get this tutorial: http://www.euroscipy.org/talk/882

4.6.1 Why

- "There's a bug?"
- "I don't understand what this is supposed to do?"
- "I have this fancy code. Would you like to have it?"
- "I'd like to help! What can I do?"

4.6.2 Reporting bugs

- Bug tracker (prefer **this**)
  - http://projects.scipy.org/numpy
  - http://projects.scipy.org/scipy

  - Click the “Register” link to get an account
- Mailing lists (scipy.org/Mailing_Lists)
  - If you’re unsure
  - No replies in a week or so? Just file a bug ticket.

Good bug report

```
Title: numpy.random.permutations fails for non-integer arguments

I'm trying to generate random permutations, using numpy.random.permutations

When calling numpy.random.permutation with non-integer arguments
it fails with a cryptic error message::

    >>> np.random.permutation(12)
    array([11,  5,  8,  4,  6,  1,  9,  3,  7,  2, 10,  0])
    >>> np.random.permutation(12.)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "mtrand.pyx", line 3311, in mtrand.RandomState.permutation
      File "mtrand.pyx", line 3254, in mtrand.RandomState.shuffle
    TypeError: len() of unsized object

This also happens with long arguments, and so
np.random.permutation(X.shape[0]) where X is an array fails on 64
bit windows (where shape is a tuple of longs).

It would be great if it could cast to integer or at least raise a
proper error for non-integer types.

I'm using NumPy 1.4.1, built from the official tarball, on Windows
64 with Visual studio 2008, on Python.org 64-bit Python.
```

0. What are you trying to do?
1. **Small code snippet reproducing the bug** (if possible)
   - What actually happens
   - What you’d expect
2. Platform (Windows / Linux / OSX, 32/64 bits, x86/PPC, . . .)
3. Version of NumPy/Scipy

```
>>> print(np.__version__)
1...
```

Check that the following is what you expect

```
>>> print(np.__file__)
/...
```

In case you have old/broken NumPy installations lying around.

If unsure, try to remove existing NumPy installations, and reinstall. . .

4.6.3 Contributing to documentation

1. Documentation editor
   - http://docs.scipy.org/doc/numpy
   - Registration
     - Register an account
     - Subscribe to `scipy-dev` mailing list (subscribers-only)
     - Problem with mailing lists: you get mail
       * But: **you can turn mail delivery off**
       * "change your subscription options", at the bottom of http://mail.scipy.org/mailman/listinfo/scipy-dev
     - Send a mail @ `scipy-dev` mailing list; ask for activation:

```
To: scipy-dev@scipy.org

Hi,

I'd like to edit NumPy/Scipy docstrings. My account is XXXXX

Cheers,
N. N.
```

   - Check the style guide:
     - http://docs.scipy.org/doc/numpy/
     - Don't be intimidated; to fix a small thing, just fix it
   - Edit
2. Edit sources and send patches (as for bugs)
3. Complain on the mailing list

4.6.4 Contributing features

0. Ask on mailing list, if unsure where it should go
1. Write a patch, add an enhancement ticket on the bug tracket
2. OR, create a Git branch implementing the feature + add enhancement ticket.
   - Especially for big/invasive additions
   - http://projects.scipy.org/numpy/wiki/GitMirror
   - http://www.spheredev.org/wiki/Git_for_the_lazy

```
# Clone numpy repository
git clone --origin svn http://projects.scipy.org/git/numpy.git numpy
cd numpy

# Create a feature branch
git checkout -b name-of-my-feature-branch  svn/trunk
```

```
<edit stuff>

git commit -a
```

- Create account on https://github.com (or anywhere)
- Create a new repository @ Github
- Push your work to github

```
git remote add github git@github:USERNAME/REPOSITORYNAME.git
git push github name-of-my-feature-branch
```

4.6.5 How to help, in general

- Bug fixes always welcome!
  - What irks you most
  - Browse the tracker
- Documentation work
  - API docs: improvements to docstrings
    - Know some Scipy module well?
  - *User guide*
    - Needs to be done eventually.
    - Want to think? Come up with a Table of Contents

      http://scipy.org/Developer_Zone/UG_Toc
- Ask on communication channels:
  - `numpy-discussion` list
  - `scipy-dev` list

CHAPTER 5

Debugging code

Author: *Gaël Varoquaux*

This section explores tools to understand better your code base: debugging, to find and fix bugs.

It is not specific to the scientific Python community, but the strategies that we will employ are tailored to its needs.

Prerequisites

- Numpy
- IPython
- nosetests
- pyflakes
- gdb for the C-debugging part.

Chapter contents

5.1 Avoiding bugs

5.1.1 Coding best practices to avoid getting in trouble

Brian Kernighan

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

- We all write buggy code. Accept it. Deal with it.
- Write your code with testing and debugging in mind.
- Keep It Simple, Stupid (KISS).
  - What is the simplest thing that could possibly work?
- Don't Repeat Yourself (DRY).
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - Constants, algorithms, etc...
- Try to limit interdependencies of your code. (Loose Coupling)
- Give your variables, functions and modules meaningful names (not mathematics names)

5.1.2 pyflakes: fast static analysis

They are several static analysis tools in Python; to name a few:

- pylint
- pychecker
- pyflakes
- flake8

Here we focus on *pyflakes*, which is the simplest tool.

- **Fast, simple**
- Detects syntax errors, missing imports, typos on names.

Another good recommendation is the *flake8* tool which is a combination of pyflakes and pep8. Thus, in addition to the types of errors that pyflakes catches, flake8 detects violations of the recommendation in PEP8 style guide.

Integrating pyflakes (or flake8) in your editor or IDE is highly recommended, it **does yield productivity gains**.

Running pyflakes on the current edited file

You can bind a key to run pyflakes in the current buffer.

- **In kate** Menu: 'settings -> configure kate
  - In plugins enable 'external tools'
  - In external Tools', add *pyflakes*:

```
kdialog --title "pyflakes %filename" --msgbox "$(pyflakes %filename)"
```

- **In TextMate**

  Menu: TextMate -> Preferences -> Advanced -> Shell variables, add a shell variable:

```
TM_PYCHECKER = /Library/Frameworks/Python.framework/Versions/Current/bin/pyflakes
```

  Then *Ctrl-Shift-V* is binded to a pyflakes report

- **In vim** In your *.vimrc* (binds F5 to *pyflakes*):

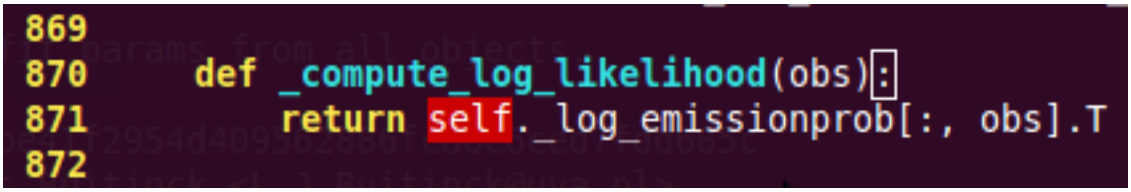
```
autocmd FileType python let &mp = 'echo "*** running % ***" ; pyflakes %'
autocmd FileType tex,mp,rst,python imap <Esc>[15~ <C-O>:make!^M
autocmd FileType tex,mp,rst,python map  <Esc>[15~ :make!^M
autocmd FileType tex,mp,rst,python set autowrite
```

- **In emacs** In your *.emacs* (binds F5 to *pyflakes*):

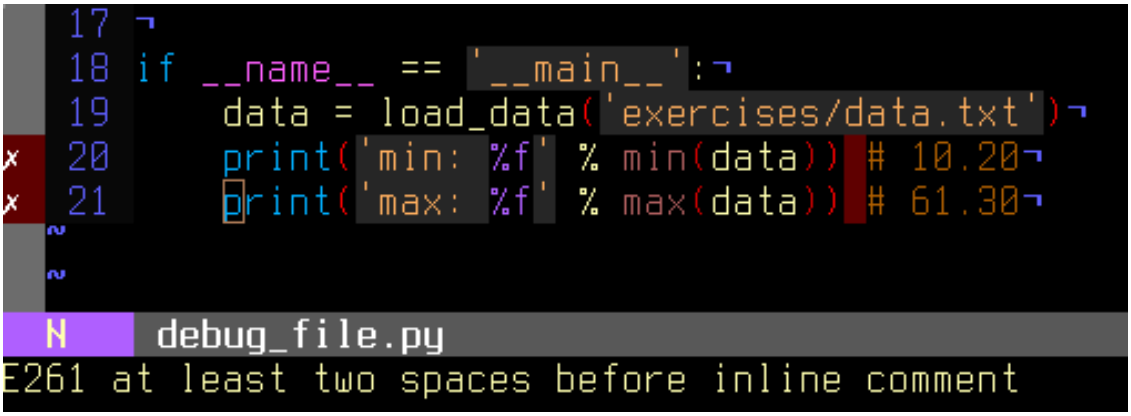
```
(defun pyflakes-thisfile () (interactive)
       (compile (format "pyflakes %s" (buffer-file-name)))
)

(define-minor-mode pyflakes-mode
    "Toggle pyflakes mode.
    With no argument, this command toggles the mode.
    Non-null prefix argument turns on the mode.
    Null prefix argument turns off the mode."
    ;; The initial value.
    nil
    ;; The indicator for the mode line.
    " Pyflakes"
    ;; The minor mode bindings.
    '( ([f5] . pyflakes-thisfile) )
)

(add-hook 'python-mode-hook (lambda () (pyflakes-mode t)))
```

A type-as-go spell-checker like integration

- **In vim**
  - Use the pyflakes.vim plugin:
    1. download the zip file from http://www.vim.org/scripts/script.php?script_id=2441
    2. extract the files in `~/.vim/ftplugin/python`
    3. make sure your vimrc has `filetype plugin indent on`



– Alternatively: use the syntastic plugin. This can be configured to use `flake8` too and also handles on-the-fly checking for many other languages.



- **In emacs** Use the flymake mode with pyflakes, documented on http://www.plope.com/Members/chrism/flymake-mode : add the following to your .emacs file:

```
(when (load "flymake" t)
        (defun flymake-pyflakes-init ()
        (let* ((temp-file (flymake-init-create-temp-buffer-copy
                           'flymake-create-temp-inplace))
            (local-file (file-relative-name
                         temp-file
                         (file-name-directory buffer-file-name))))
            (list "pyflakes" (list local-file))))

        (add-to-list 'flymake-allowed-file-name-masks
                '("\\.py\\'" flymake-pyflakes-init)))

(add-hook 'find-file-hook 'flymake-find-file-hook)
```

5.2 Debugging workflow

If you do have a non trivial bug, this is when debugging strategies kick in. There is no silver bullet. Yet, strategies help:

> **For debugging a given problem, the favorable situation is when the problem is isolated in a small number of lines of code, outside framework or application code, with short modify-run-fail cycles**

1. Make it fail reliably. Find a test case that makes the code fail every time.
2. Divide and Conquer. Once you have a failing test case, isolate the failing code.
   - Which module.
   - Which function.
   - Which line of code.

=> isolate a small reproducible failure: a test case

3. Change one thing at a time and re-run the failing test case.
4. Use the debugger to understand what is going wrong.
5. Take notes and be patient. It may take a while.

Note: Once you have gone through this process: isolated a tight piece of code reproducing the bug and fix the bug using this piece of code, add the corresponding code to your test suite.

5.3 Using the Python debugger

The python debugger, `pdb`: https://docs.python.org/library/pdb.html, allows you to inspect your code interactively.

Specifically it allows you to:

- View the source code.
- Walk up and down the call stack.
- Inspect values of variables.
- Modify values of variables.
- Set breakpoints.

print

Yes, `print` statements do work as a debugging tool. However to inspect runtime, it is often more efficient to use the debugger.

5.3.1 Invoking the debugger

Ways to launch the debugger:

1. Postmortem, launch debugger after module errors.
2. Launch the module with the debugger.
3. Call the debugger inside the module

Postmortem

Situation: You're working in IPython and you get a traceback.

Here we debug the file `index_error.py`. When running it, an `IndexError` is raised. Type `%debug` and drop into the debugger.

```
In [1]: %run index_error.py
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in <module>()
      6
```

```
      7 if __name__ == '__main__':
----> 8     index_error()
      9

/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in index_error()
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':

IndexError: list index out of range

In [2]: %debug
> /home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py(5)index_error()
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6

ipdb> list
      1 """Small snippet to raise an IndexError."""
      2
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':
      8     index_error()
      9

ipdb> len(lst)
6
ipdb> print lst[len(lst)-1]
r
ipdb> quit

In [3]:
```

Post-mortem debugging without IPython

In some situations you cannot use IPython, for instance to debug a script that wants to be called from the command line. In this case, you can call the script with `python -m pdb script.py`:

```
$ python -m pdb index_error.py
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(1)<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
File "/usr/lib/python2.6/pdb.py", line 1296, in main
    pdb._runscript(mainpyfile)
File "/usr/lib/python2.6/pdb.py", line 1215, in _runscript
    self.run(statement)
File "/usr/lib/python2.6/bdb.py", line 372, in run
    exec cmd in globals, locals
File "<string>", line 1, in <module>
File "index_error.py", line 8, in <module>
    index_error()
File "index_error.py", line 5, in index_error
    print lst[len(lst)]
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(5)index_error()
-> print lst[len(lst)]
```

Step-by-step execution

Situation: You believe a bug exists in a module but are not sure where.

For instance we are trying to debug `wiener_filtering.py`. Indeed the code runs, but the filtering does not work well.

- Run the script in IPython with the debugger using `%run -d wiener_filtering.p`:

```
In [1]: %run -d wiener_filtering.py
*** Blank or comment
*** Blank or comment
*** Blank or comment
Breakpoint 1 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()
```

- Set a break point at line 34 using `b 34`:

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(4)<module>()
      3
1---> 4 import numpy as np
      5 import scipy as sp

ipdb> b 34
Breakpoint 2 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.
↪py:34
```

- Continue execution to next breakpoint with `c(ont(inue))`:

```
ipdb> c
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(34)iterated_
↪wiener()
     33     """
2--> 34     noisy_img = noisy_img
     35     denoised_img = local_mean(noisy_img, size=size)
```

- Step into code with `n(ext)` and `s(tep)`: `next` jumps to the next statement in the current execution context, while `step` will go across execution contexts, i.e. enable exploring inside function calls:

```
ipdb> s
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(35)iterated_
↪wiener()
2    34     noisy_img = noisy_img
---> 35     denoised_img = local_mean(noisy_img, size=size)
     36     l_var = local_var(noisy_img, size=size)

ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(36)iterated_
↪wiener()
     35     denoised_img = local_mean(noisy_img, size=size)
---> 36     l_var = local_var(noisy_img, size=size)
     37     for i in range(3):
```

- Step a few lines and explore the local variables:

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(37)iterated_
↪wiener()
     36     l_var = local_var(noisy_img, size=size)
---> 37     for i in range(3):
     38         res = noisy_img - denoised_img
ipdb> print l_var
[[5868 5379 5316 ..., 5071 4799 5149]
 [5013  363  437 ...,  346  262 4355]
 [5379  410  344 ...,  392  604 3377]
 ...,
 [ 435  362  308 ...,  275  198 1632]
 [ 548  392  290 ...,  248  263 1653]
 [ 466  789  736 ..., 1835 1725 1940]]
ipdb> print l_var.min()
0
```

Oh dear, nothing but integers, and 0 variation. Here is our bug, we are doing integer arithmetic.

Raising exception on numerical errors

When we run the `wiener_filtering.py` file, the following warnings are raised:

```
In [2]: %run wiener_filtering.py
wiener_filtering.py:40: RuntimeWarning: divide by zero encountered in divide
    noise_level = (1 - noise/l_var )
```

We can turn these warnings in exception, which enables us to do post-mortem debugging on them, and find our problem more quickly:

```
In [3]: np.seterr(all='raise')
Out[3]: {'divide': 'print', 'invalid': 'print', 'over': 'print', 'under': 'ignore'}
In [4]: %run wiener_filtering.py
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
/home/esc/anaconda/lib/python2.7/site-packages/IPython/utils/py3compat.pyc in execfile(fname,␣
↪*where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.py␣
↪in <module>()
     55 pl.matshow(noisy_face[cut], cmap=pl.cm.gray)
     56
---> 57 denoised_face = iterated_wiener(noisy_face)
     58 pl.matshow(denoised_face[cut], cmap=pl.cm.gray)
     59

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.py␣
↪in iterated_wiener(noisy_img, size)
     38         res = noisy_img - denoised_img
     39         noise = (res**2).sum()/res.size
---> 40         noise_level = (1 - noise/l_var )
     41         noise_level[noise_level<0] = 0
     42         denoised_img += noise_level*res

FloatingPointError: divide by zero encountered in divide
```

Other ways of starting a debugger

- **Raising an exception as a poor man break point**

  If you find it tedious to note the line number to set a break point, you can simply raise an exception at the point that you want to inspect and use IPython's `%debug`. Note that in this case you cannot step or continue the execution.

- **Debugging test failures using nosetests**

  You can run `nosetests --pdb` to drop in post-mortem debugging on exceptions, and `nosetests --pdb-failure` to inspect test failures using the debugger.

  In addition, you can use the IPython interface for the debugger in nose by installing the nose plugin ipdb-plugin. You can than pass `--ipdb` and `--ipdb-failure` options to nosetests.

- **Calling the debugger explicitly**

  Insert the following line where you want to drop in the debugger:

```
import pdb; pdb.set_trace()
```

Warning: When running `nosetests`, the output is captured, and thus it seems that the debugger does not work. Simply run the nosetests with the `-s` flag.

Graphical debuggers and alternatives

- For stepping through code and inspecting variables, you might find it more convenient to use a graphical debugger such as winpdb.
- Alternatively, pudb is a good semi-graphical debugger with a text user interface in the console.
- Also, the pydbgr project is probably worth looking at.

5.3.2 Debugger commands and interaction

`l(list)`	Lists the code at the current position
`u(p)`	Walk up the call stack
`d(own)`	Walk down the call stack
`n(ext)`	Execute the next line (does not go down in new functions)
`s(tep)`	Execute the next statement (goes down in new functions)
`bt`	Print the call stack
`a`	Print the local variables
`!command`	Execute the given **Python** command (by opposition to pdb commands

Warning: Debugger commands are not Python code

You cannot name the variables the way you want. For instance, if in you cannot override the variables in the current frame with the same name: **use different names than your local variable when typing code in the debugger**.

Getting help when in the debugger

Type `h` or `help` to access the interactive help:

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pdef   r        tbreak   w
a      c          continue  exit    l     pdoc   restart  u        whatis
alias  cl         d         h       list  pinfo  return   unalias  where
args   clear      debug     help    n     pp     run      unt
b      commands   disable   ignore  next  q      s        until
break  condition  down      j       p     quit   step     up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

5.4 Debugging segmentation faults using gdb

If you have a segmentation fault, you cannot debug it with pdb, as it crashes the Python interpreter before it can drop in the debugger. Similarly, if you have a bug in C code embedded in Python, pdb is useless. For this we turn to the gnu debugger, gdb, available on Linux.

Before we start with gdb, let us add a few Python-specific tools to it. For this we add a few macros to our `~/.gdbinit`. The optimal choice of macro depends on your Python version and your gdb version. I have added a simplified version in `gdbinit`, but feel free to read DebuggingWithGdb.

To debug with gdb the Python script `segfault.py`, we can run the script in gdb as follows

```
$ gdb python
...
(gdb) run segfault.py
Starting program: /usr/bin/python segfault.py
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
_strided_byte_copy (dst=0x8537478 "\360\343G", outstrides=4, src=
    0x86c0690 <Address 0x86c0690 out of bounds>, instrides=32, N=3,
    elsize=4)
        at numpy/core/src/multiarray/ctors.c:365
365             _FAST_MOVE(Int32);
(gdb)
```

We get a segfault, and gdb captures it for post-mortem debugging in the C level stack (not the Python call stack). We can debug the C call stack using gdb's commands:

```
(gdb) up
#1  0x004af4f5 in _copy_from_same_shape (dest=<value optimized out>,
    src=<value optimized out>, myfunc=0x496780 <_strided_byte_copy>,
    swap=0)
at numpy/core/src/multiarray/ctors.c:748
748         myfunc(dit->dataptr, dest->strides[maxaxis],
```

As you can see, right now, we are in the C code of numpy. We would like to know what is the Python code that triggers this segfault, so we go up the stack until we hit the Python execution loop:

```
(gdb) up
#8  0x080ddd23 in call_function (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.
↪py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=<module at remote␣
↪0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:3750
3750    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c

(gdb) up
#9  PyEval_EvalFrameEx (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.
↪py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=<module at remote␣
↪0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:2412
2412    in ../Python/ceval.c
(gdb)
```

Once we are in the Python execution loop, we can use our special Python helper function. For instance we can find the corresponding Python code:

```
(gdb) pyframe
/home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.py (158): _leading_trailing
(gdb)
```

This is numpy code, we need to go up until we find code that we have written:

```
(gdb) up
...
(gdb) up
#34 0x080dc97a in PyEval_EvalFrameEx (f=
    Frame 0x82f064c, for file segfault.py, line 11, in print_big_array (small_array=<numpy.ndarray␣
↪at remote 0x853ecf0>, big_array=<numpy.ndarray at remote 0x853ed20>), throwflag=0) at ../Python/
↪ceval.c:1630
1630    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c
(gdb) pyframe
segfault.py (12): print_big_array
```

The corresponding code is:

```
def make_big_array(small_array):
    big_array = stride_tricks.as_strided(small_array,
                                         shape=(2e6, 2e6), strides=(32, 32))
    return big_array
```

```
def print_big_array(small_array):
    big_array = make_big_array(small_array)
```

Thus the segfault happens when printing `big_array[-10:]`. The reason is simply that `big_array` has been allocated with its end outside the program memory.

Note: For a list of Python-specific commands defined in the *gdbinit*, read the source of this file.

Wrap up exercise

The following script is well documented and hopefully legible. It seeks to answer a problem of actual interest for numerical computing, but it does not work... Can you debug it?

Python source code: `to_debug.py`