NumPy

Object Oriented Programming and Scripting in Python

Basic information

- Package for scientific computing with Python
- High level functions (linear algebra, Fourier transform etc.)
- Multi-dimensional arrays (ndarray)

- Website: http://numpy.scipy.org/
- Import:

```
>>> from numpy import *
>>> import numpy
>>> import numpy as np
```

Arrays

- NumPy arrays are homogeneous. Unlike Python lists, each element of an array must be of the same type (dtype)
- It handles high dimensional data
- Creation of arrays from a list or from a tuple
- Array definition:

```
array(*args)
```

Arrays

- Array definitions:
- ```
  #from list
  >>> a = numpy.array([4,7,3])
  >>> a
        array([4, 7, 3])
  ```
- ```
  #from tuple
  >>> a = numpy.array((4,7,3))
  >>> a
        array([4, 7, 3])
  ```
- ```
  >>> a = numpy.array((4,7,3), float)  #or
  ```
- ```
  >>> a = numpy.array((4,7,3), dtype = float)
  >>> a
      array([4., 7., 3.])
  ```
- ```
  >>> a = numpy.array(1,2,3,4) # error: only list and tuples are accepted!!
  ```
  ```

Arrays

- Array definition :
- ```
  #arange: similar to 'range' function
  >>> a = numpy.arange(10)
  >>> a
      array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```
- ```
  >>> a = numpy.arange(1, 4)
  >>> a
      array([1, 2, 3])
  ```
- ```
  >>> a = numpy.arange(3, 24, 5)
  >>> a
      array([ 3,  8, 13, 18, 23])
  ```
  ```

Arrays

- Array definition :
- #**linspace(start, stop, npoints)**: defines a set of points ranging from start to stop; the number of (equidistant) points are specified bye npoints

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

Multidimensional Arrays

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

```
>>> L1 = [0, 1, 2, 3]
>>> L2 = [4, 5, 6, 7]
>>> a = numpy.array([L1, L2])
>>> a
    array([[0, 1, 2, 3],
           [4, 5, 6, 7, 8, 9]])
```

Basic functions

- Given a created array **a**:
- **a.ndim** (or ndim(a)) → Number of dimensions
- **a.shape** (or **shape(a)**) → Number of elements of each dimension
- **a.size** (or **size(a)**) → Total number of elements
- **type(a)** → Data type
- **a.copy()** (or **copy(a)**) → Copy creation
- **a.reshape(x)** → Shape modification

Basic functions

- Examples:

```
#Monodimensional arrays
```

- ```
  >>> a = numpy.arange(10)
  ```
- ```
  >>> a.ndim
      1
  ```
- ```
  >>> a.shape
      (10,)
  ```
- ```
  >>> a.size
      10
  ```
- ```
  >>> a.dtype.name
      'int32'
  ```
- ```
  >>> type(a)
      <type 'numpy.ndarray'>
  ```
- ```
  >>> b = numpy.copy(a)
  >>> b
      array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```
  ```

Basic functions

- Examples:
  #Multidimensional arrays
- >>> a = numpy.arange([[0, 1, 2, 3], [4, 5, 6, 7]])
- >>> a.ndim
  2
- >>> a.shape
  (2, 4)
- >>> a.size
- 8
- >>> b = numpy.copy(a)
  >>> b
  array([[0, 1, 2, 3, 4],
  [5, 6, 7, 8, 9]])

Basic functions

- Reshaping:

```
>>> a = np.array(range(10), float)
    array([ 0., 1., 2., 3., 4., 5., 6.,
7., 8., 9.])
>>> a = a.reshape((5, 2))
>>> a
    array([[ 0., 1.],
           [ 2., 3.],
           [ 4., 5.],
           [ 6., 7.],
           [ 8., 9.]])
>>> a.shape
    (5, 2)
```

N.B. Reshaping does not not work if the total number of elements of the new array is not the same of the original array.
Ex. : **a.reshape((5,3))**

Structured Arrays

- Numpy is able to create arrays of structured datatype, permitting to manipulate the data by named fields
- Conveniently, one can access any field of the array by indexing using the string that names that field.

```
>>> x = np.array([(1, 2), (3, 4)]),
         dtype = [('a', float), ('b', float)])
>>> x
    array((1., 2.) (3., 4.)
          dtype=[('a', '<f8'), ('b', '<f8')])
>>> x['a']
    [1.  3.]
>>> x.dtype.names
    ('a', 'b')
```

Functions

- Given a created array **a** :
- **a.zeros(X)** → Creates a matrix initialized with zeros ; X is the shape of the created array
- **a.ones(X)** → Creates a matrix initialized with ones; X is the shape of the created array
- **zeros_like(a)** → Creates a matrix initialized with zeros, having the same shape of a
- **a.fill(i)** → Sets all elements of a to the value specified by the parameter *i*
- **a.transpose()** → Transpose the array a
- **a.flatten()** → Transform a generic array in a monodimensional array
- **concatenate(X)** → Concatenates two or more arrays, specified in the t-uple *X*

Functions

- Examples:
-
```
>>> numpy.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```
-
```
>>> numpy.zeros((3,3), int)
array([[ 0,  0,  0],
       [ 0,  0,  0],
       [ 0,  0,  0]])
```
-
```
>>> numpy.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Functions

- Examples:
- `>>> array1 = numpy.array([[1, 2], [3, 4]])`

```
>>> array1
    array([[1 2]
           [3 4]])
>>> array2 = numpy.zeros_like(array1)
>>> array2
    array([[0 0]
          [0 0]])
```

- `>>> a = numpy.array([[1, 2], [3, 4]])`
- `>>> a.fill(3)`

```
>>> a
    array([[3 3]
           [3 3]])
```

Functions

```
>>> a
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
>>> a.transpose()
array([[ 0., 3.],
       [ 1., 4.],
       [ 2., 5.]])
>>> a.flatten()
array([ 1., 2., 3., 4., 5., 6.])
>>> a = numpy.array([[1, 2], [3, 4]])
>>> b = numpy.array([[5, 6], [7,8]])
>>> numpy.concatenate((a,b))
array([[ 1, 2],
       [ 3, 4],
       [ 5, 6],
       [ 7, 8]])
```

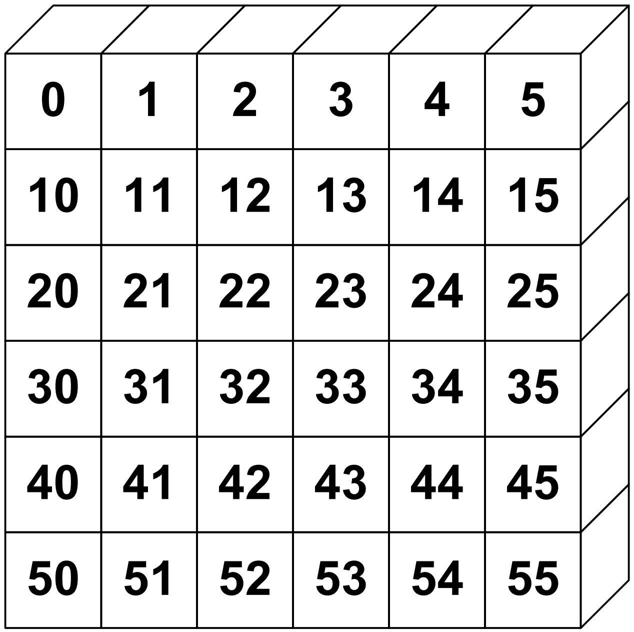
Indexing

- Like lists and t-uples, arrays are accessible with indexes [ ].
- Examples
- #indexing
- ```
  >>> a = numpy.array([0,1,2,3], int)
  >>> a[0]
      0
  ```
- ```
  >>> b = numpy.array([[4,5], [6,7]], int)
  >>> b[0,1]
      5
  ```
- ```
  #assignment
  >>> a[0] = 10
  >>> a
      array([10, 1, 2, 3])
  ```
- ```
  >>> a[0] = 10.6 #casting!!
  >>> a
      array([10, 1, 2, 3])
  ```

Slicing

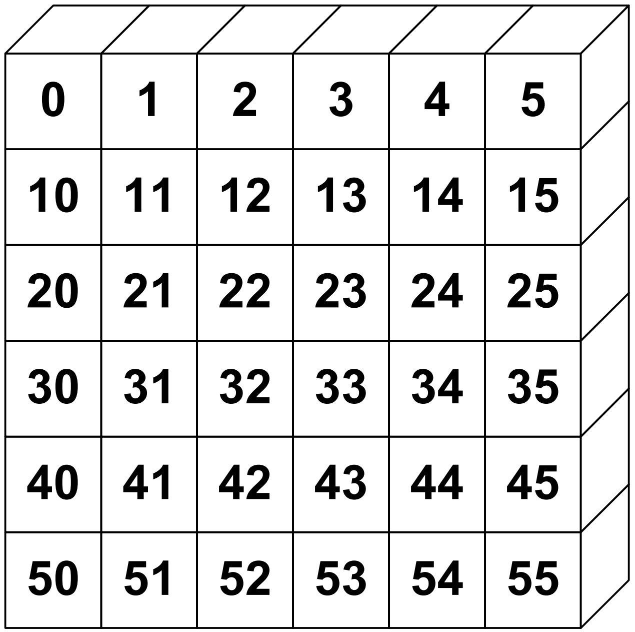
- As for lists and t-uples, an array could be accessed through slicing.
- Examples: monodimensional arrays
```
>>> a = numpy.array([0,1,2,3,4,5], int)
```
- ```
  >>> a[0:2]
      array([0,1])
  ```
- ```
  >>> a[2:4]
      array([2,3,4])
  ```
- ```
  >>> a[0:3:2]
      array([0,2])
  ```
- ```
  >>> a[3:0:-1]
      array([3,2,1])
  ```
- ```
  >>> a[3:]
      array([3,4,5])
  ```
  ```

Multidimensional Slicing



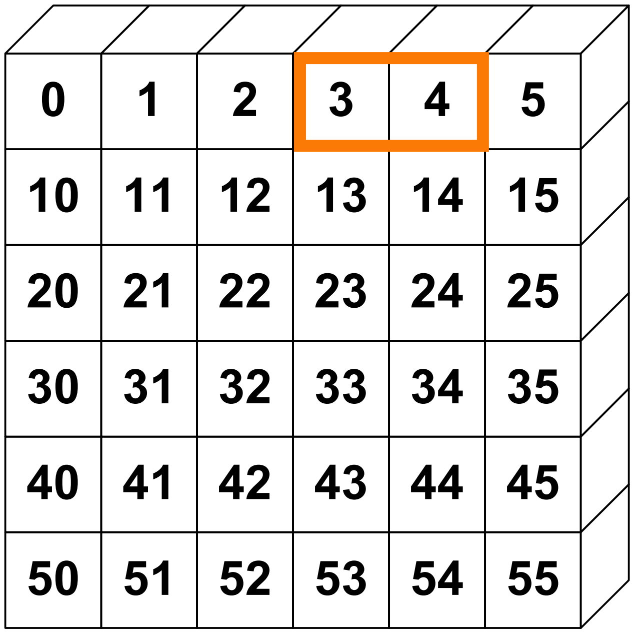
Multidimensional Slicing

```
>>> a[0,3:5]
```



Multidimensional Slicing

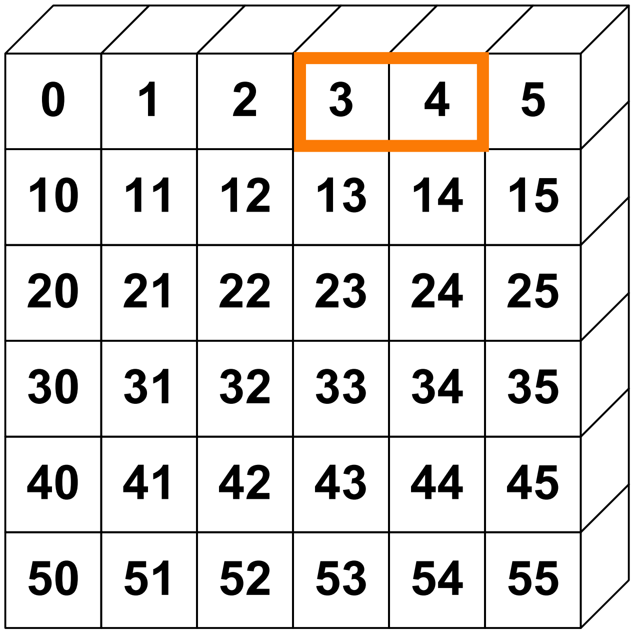
```
>>> a[0,3:5]
array([3, 4])
```



Multidimensional Slicing

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

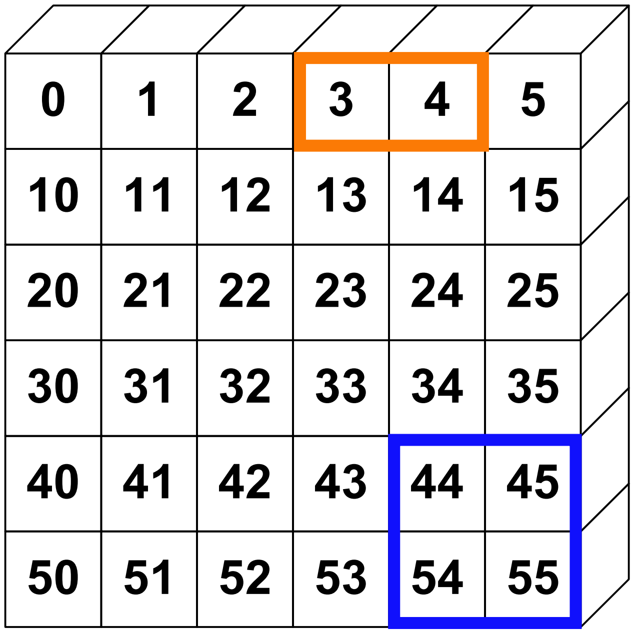
>>> a[4:,4:]
```



Multidimensional Slicing

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

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

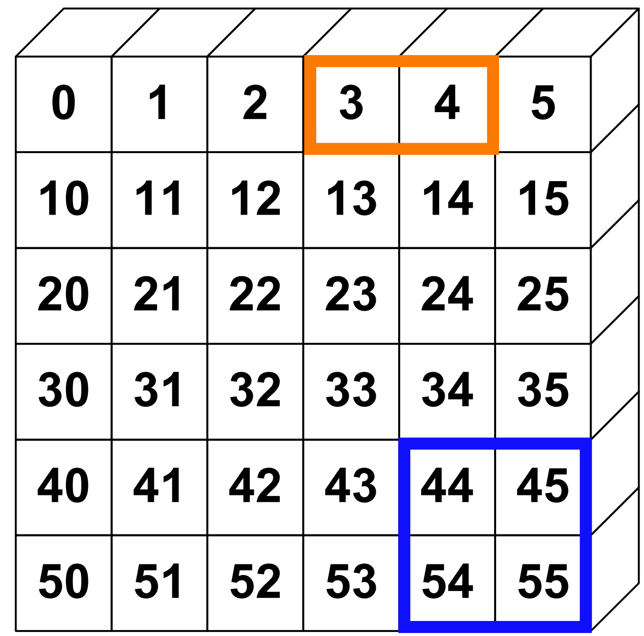


Multidimensional Slicing

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

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

>>> a[:,2]
```

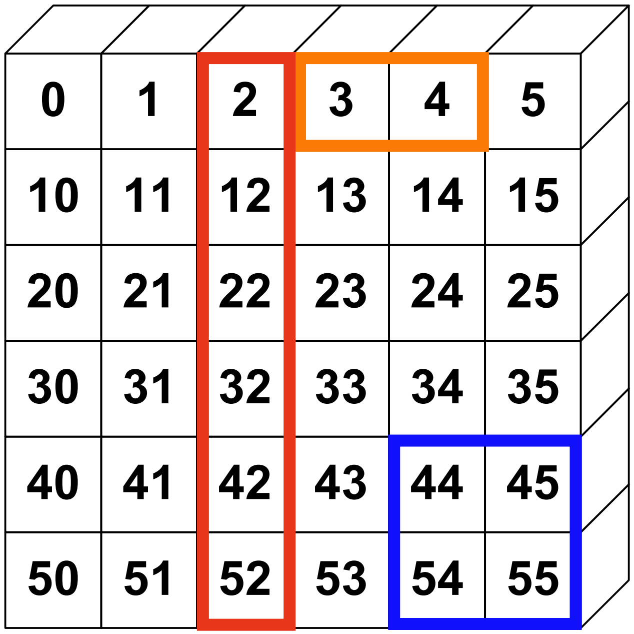


Multidimensional Slicing

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

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

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



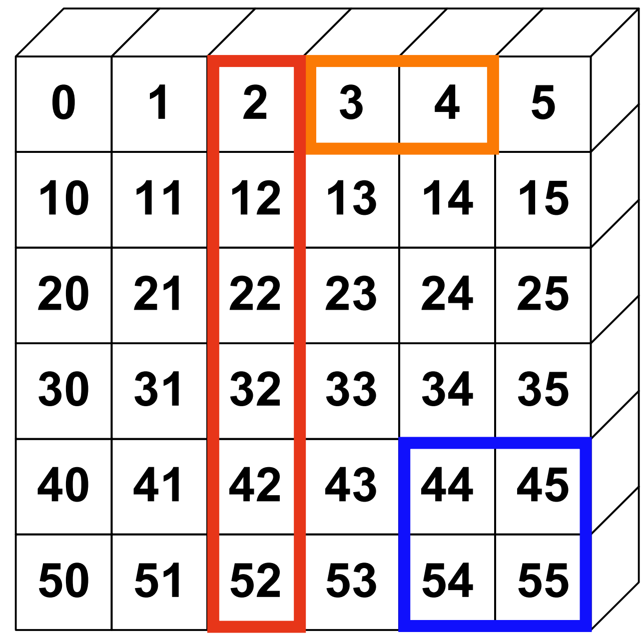
Multidimensional Slicing

```
>>> 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]
```



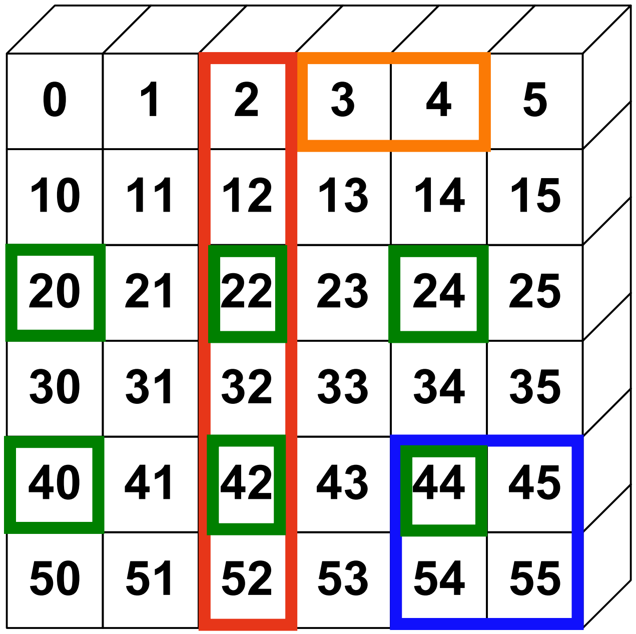
Multidimensional Slicing

```
>>> 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]])
```



Slicing : references

- Slicing creates references to the original array!

```
>>> a = array((0,1,2,3,4))
>>> b = a[2:4]
>>> b
    array([2,3])
>>> b[0] = 10
>>> b
    array([10,3])
# change in  b → same change in a!
>>> a
    array([ 1,  2, 10, 3, 4])
```

Fancy Indexing

```
>>> a = numpy.arange(0,80,10)
#positional fancy indexing
>>> y = a[[1, 2, 5]]
>>> y
    array[10 20 50]

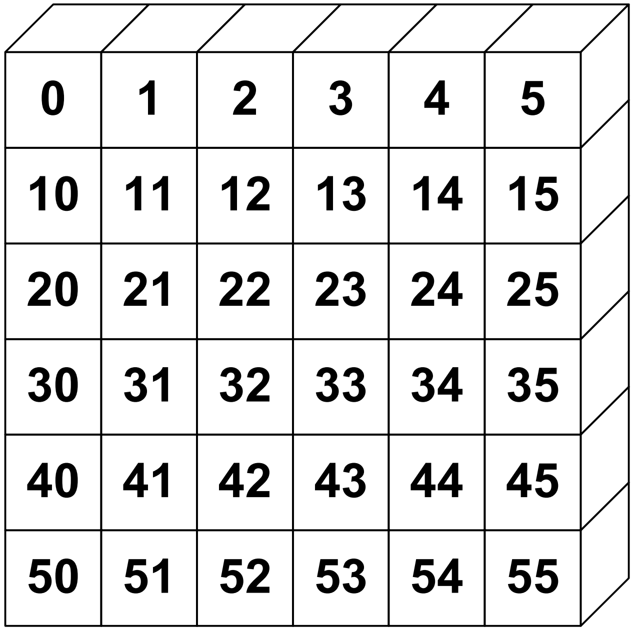
#booloean fancy indexing
>>> mask = numpy.array([0,1,1,0,0,1,0,0],
dtype=bool)
>>> y = a[mask]
>>> y
    array[10,20,50]
```

Multidimensional Fancy Indexing

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

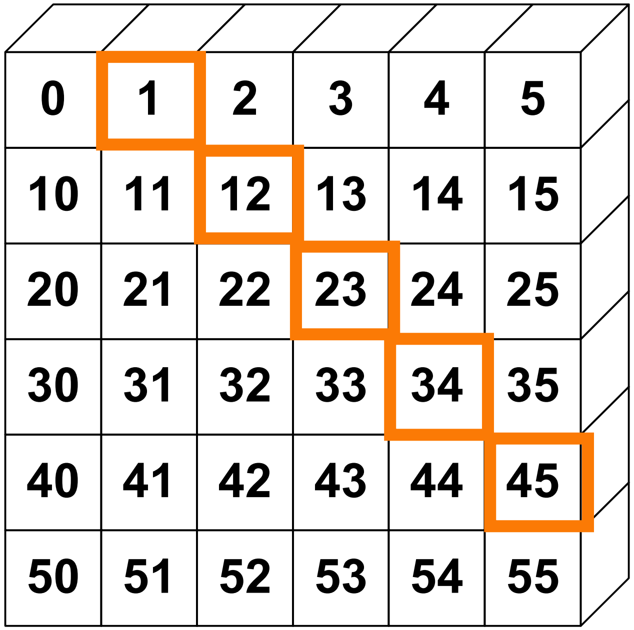
Multidimensional Fancy Indexing

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
```



Multidimensional Fancy Indexing

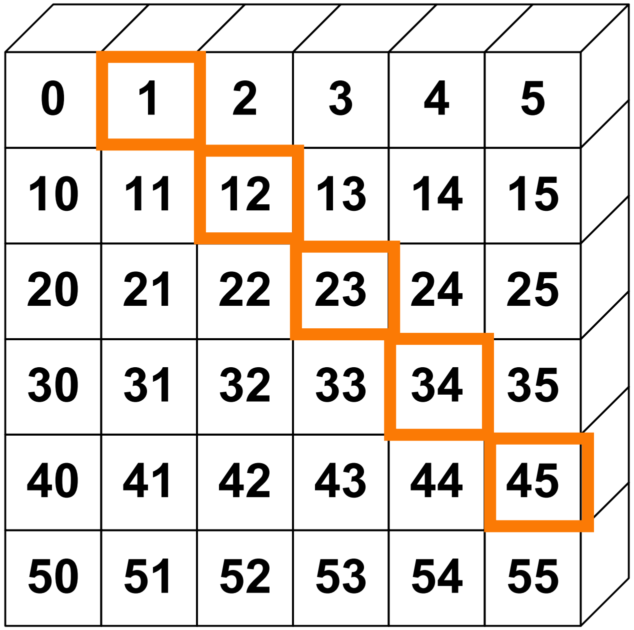
```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])
```



Multidimensional Fancy Indexing

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

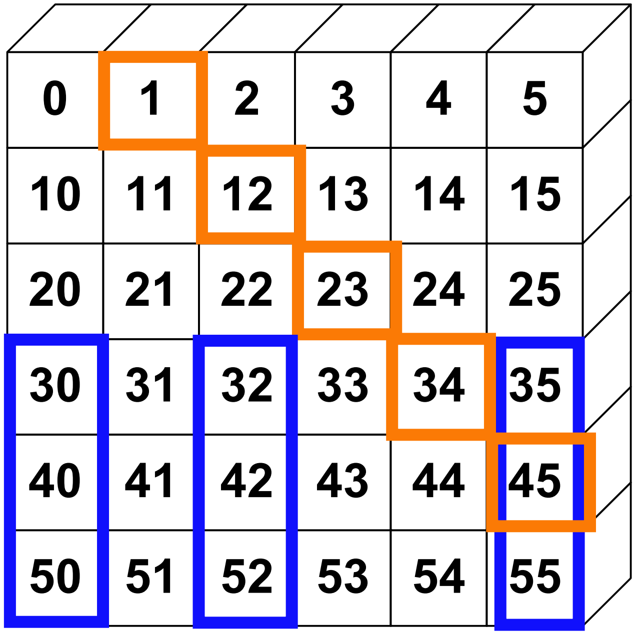
>>> a[3:,[0, 2, 5]]
```



Multidimensional Fancy Indexing

```
>>> 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]])
```

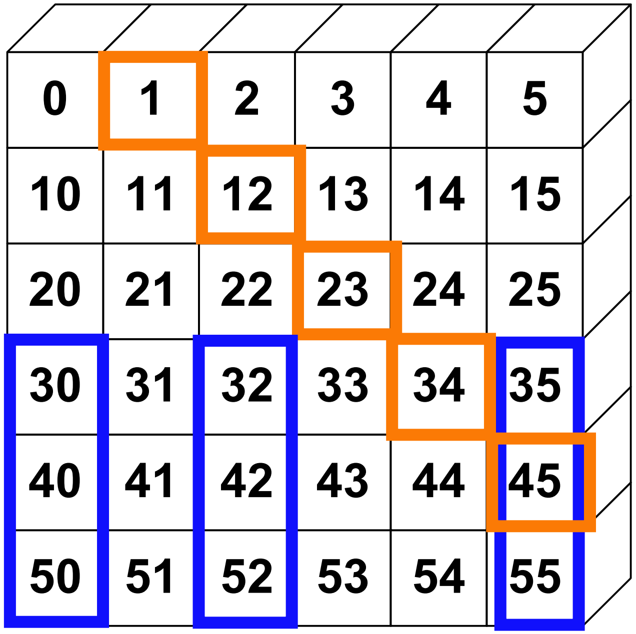


Multidimensional Fancy Indexing

```
>>> 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]
```

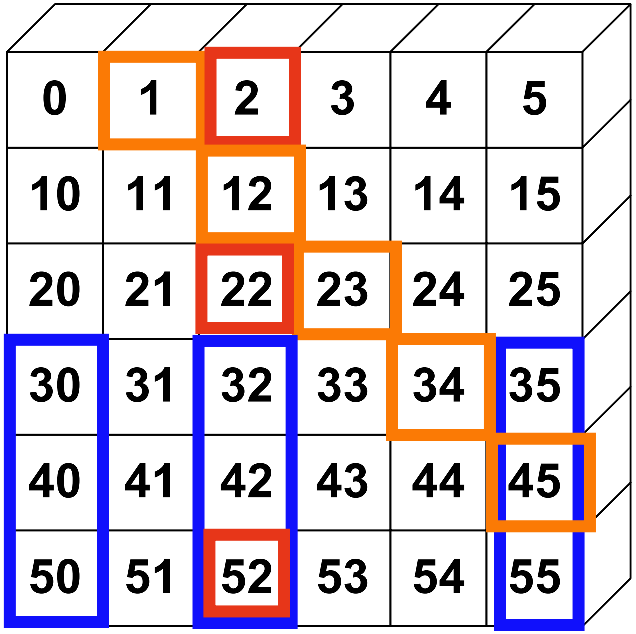


Multidimensional Fancy Indexing

```
>>> 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])
```

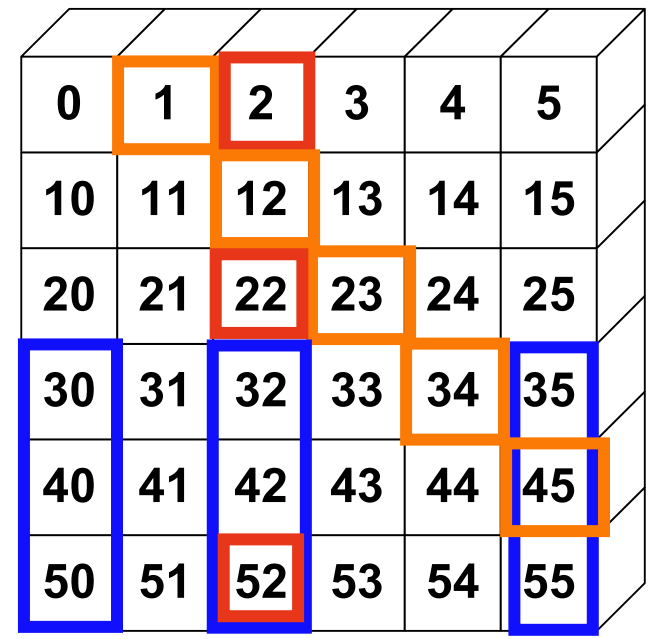


Multidimensional Fancy Indexing

```
>>> 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])
```



Unlike slicing, a fancy indexing creates copies of the original arrays, not references!

Basic operations

- Given a created array **a**:

- **a.sum()** → Sum of all elements of the array
- **a.prod()** → Product of all elements of the array
- **a.min()** → Returns the minimum value of the array
- **a.max()** → Returns the maximum value of the array
- **a.argmin()** → Returns the index of the minimum value of the array
- **a.argmax()** → returns the index of the maximum value of the array

Basic operations

- Examples:
- `>>> a = numpy.array([[1,2,3], [4,5,6]])`
- `>>> a.sum()`
  `21`
- `>>> a.prod()`
  `720`
- `>>> a.min()`
  `1`
- `>>> a.max()`
  `6`
- `>>> a.argmin()`
  `0`
- `>>> a.argmax()`
  `5`

Other Operations

- Given a created array **a**:

- **a.clip(x,y)** → Sets x value for all elements with value lesser than x, whereas sets y value for all elements with value gretaer than y
- **floor(a)** → Round down each element of a
- **ceil(a)** → Round up each element of a
- **rint(a)** → approximates each element of a to the closest integer
- Statistics:
- **a.mean()** → Returns the mean value of elements
- **a.median()** → Returns the median value of data
- **a.std()** → Returns the standard deviation of the elements
- **a.var()** → Returns the variance of the elements
- **a.cov()** → Returns the covariance matrix of a 1D or 2D array

Other Operations

- Examples:
- `>>> a = numpy.array([[1,2,3], [4,5,6]])`
- `>>> a.clip(3,5)`
  `array([[3,3,3], [4,5,5]])`
- `>>> a.mean()`
  `3.5`
- `>>> a.std()`
  `1.707825127659933`
- `>>> a.var()`
  `2.9166666666666665`

- `>>> a = np.array([1.1, 1.5, 1.9], float)`
- `>>> a.floor()`
  `array([1., 1. 1.])`
- `>>> a.ceil()`
  `array([2., 2. 2.])`
- `>>> a.floor()`
  `array([1., 2. 2.])`

Operations Along Axes

- All tprevious operations are computed considering the array as flatten, by default.
- An optinal parameter (`axis`) specifies the axes along which the operation is computed (ex: axis=0 returns the computation of each column). Default value is `None`.
  If `axis` is a tuple of ints, the operation is performed over multiple axes, instead of a single axis or all the axes as before.
- #Examples

```
>>> a
    np.array([[1, 2], [3, 4]])
>>> np.mean(a, axis=0)
    array([ 2.,  3.])
>>> np.mean(a, axis=1)
    array([ 1.5,  3.5])
```

Operators

- Aritmethic operators are applied element by element
- Operations more efficient than a simple iteration
- Examples:

```
>>> a = numpy.array([20,30,40,50])
>>> b = numpy.array([0,1,2,3])
```

```
>>> a+b
    array([20, 31, 42, 53])
```

```
>>> a-b
    array([20, 29, 38, 47])
```

```
>>> b*b
    array([0, 1, 4, 9])
```

```
>>> b**2
    array([0, 1, 4, 9])
```

Operators

Correspondence between operators and fuctions

a + b → numpy.add(a,b)
a – b → numpy.subtract(a,b)
a * b → numpy.multiply(a,b)
a % b → numpy.remainder(a,b)
a / b → numpy.divide(a,b)
a ** b → numpy.power(a,b)
a == b → numpy.equal(a,b)
a != b → numpy.not_equal(a,b)
a > b → numpy.greater(a,b)
a >= b → numpy.greater_equal(a,b)
a < b → numpy.less(a,b)
a <= b → numpy.less_equal(a,b)

Further functions

Trigonometric

```
sin(a)

cos(a)
tan(a)
arcsin(a)
arccos(a)
arctan(a)
sinh(a)
cosh(a)
tanh(a)
arcsinh(a)
arccosh(a)
arctanh(a)
```

Other Functions

```
exp(x)
log(x)
log10(x)
sqrt(x)
absolute(x)c
conjugate(x)
negative(x)f
fabs(x)
hypot(x,y)
fmod(x,y)
maximum(x,y)
minimum(x,y)
```

Matrices

- **dot(a, b)** → Dot product between two monodimensional arrays

- **linalg** → Linear algebra functions module. It provides, for example:

- **det(a)** → Determinant of a square matrix
- **inv(a)** → Inverse of a square matrix
- **eig(a)** → Returns eigenvalues and eigenvectors of a square matrix

Matrices

- Examples:

- ```
  >>> a = numpy.array([1, 2, 3],float)
  >>> b = numpy.array([0, 1, 1],float)
  >>> numpy.dot(a, b)
      5.0
  ```

- ```
  >>> a = numpy.array([[4, 2, 0], [9, 3, 7],
  [1, 2, 1]], float)
  >>> a
      array([[4., 2., 0.],
             [9., 3., 7.],
             [1., 2., 1.]])
  >>> numpy.linalg.det(a)
      -53.999999999999993
  ```

Matrices

- ```
  >>> vals, vecs = numpy.linalg.eig(a)
  >>> vals
  array([ 9. , 2.44948974, -2.44948974])
  >>> vecs
  array([[-0.3538921, -0.56786837, 0.27843404],
         [-0.88473024, 0.44024287, -0.89787873],
         [-0.30333608, 0.69549388, 0.34101066]])
  ```
- ```
  >>> b = numpy.linalg.inv(a)
  >>> b
  array([[0.14814815, 0.07407407, -0.25925926],
         [0.2037037 , -0.14814815, 0.51851852],
         [-0.27777778, 0.11111111, 0.11111111]])
  ```

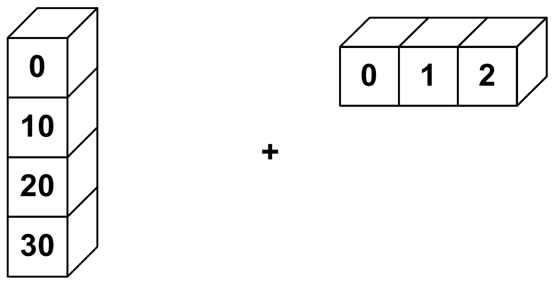
Broadcasting

- Operations between arrays with different shape: when it is possibles, both arrays are converted in arrays having the same shape

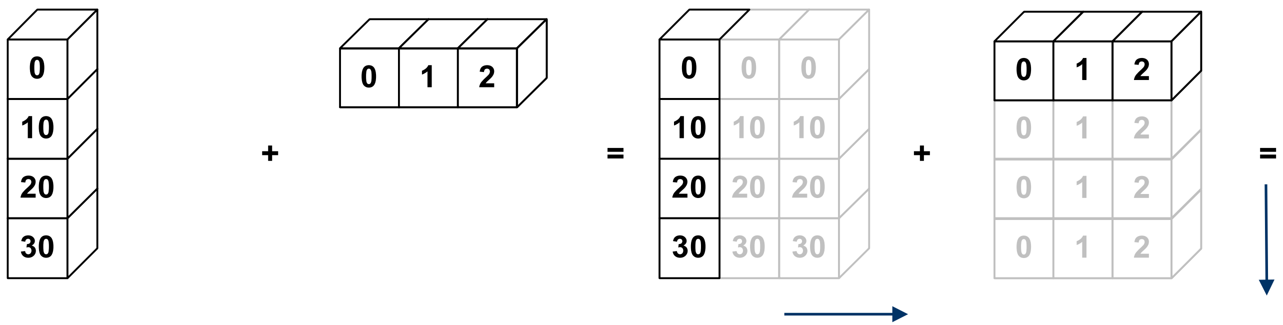
- Esempio:

```
>>> x = numpy.array([[10],[20],[30]])
>>> y = numpy.array([1,2,3,4])
>>> numpy.add(x,y)
        array([[11 12 13 14]
                [21 22 23 24]
                [31 32 33 34]])
```

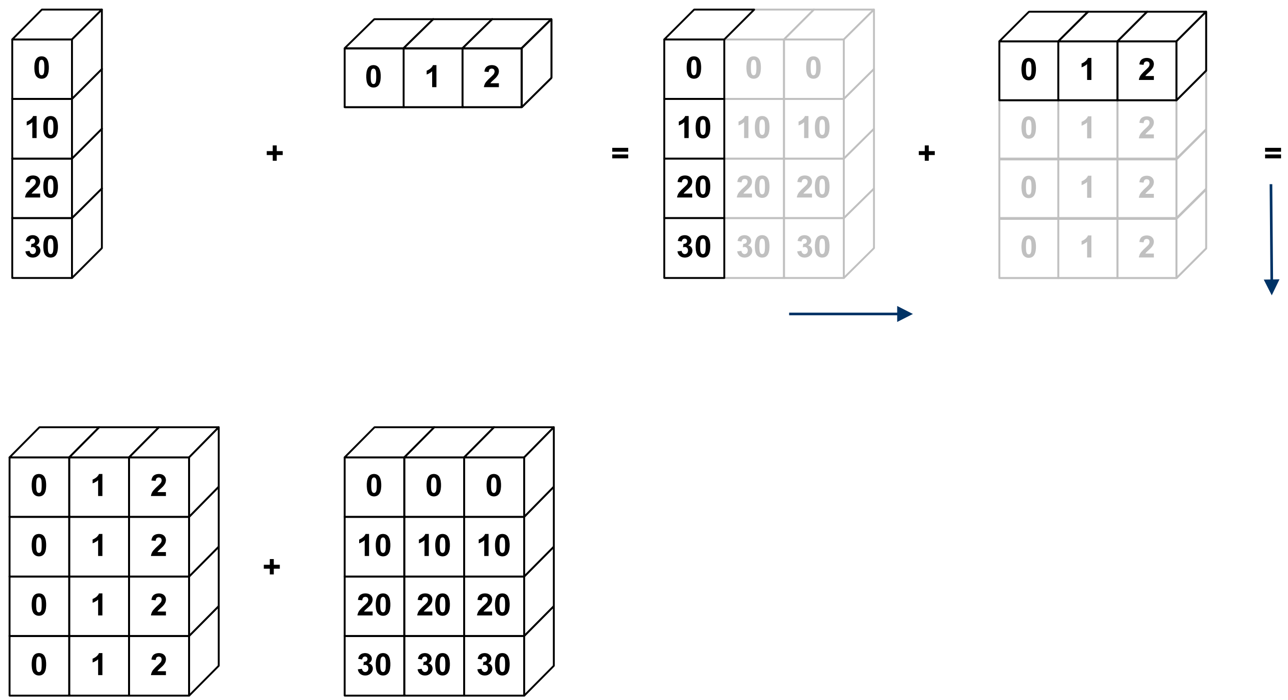
Broadcasting



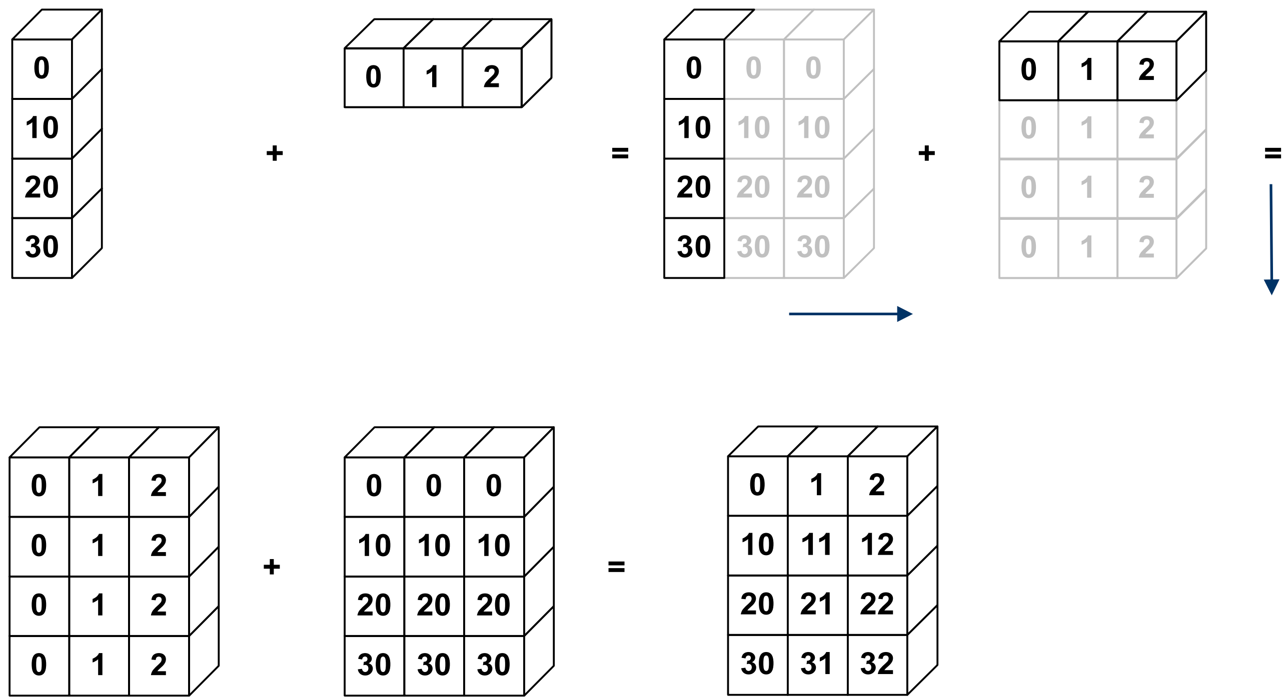
Broadcasting



Broadcasting



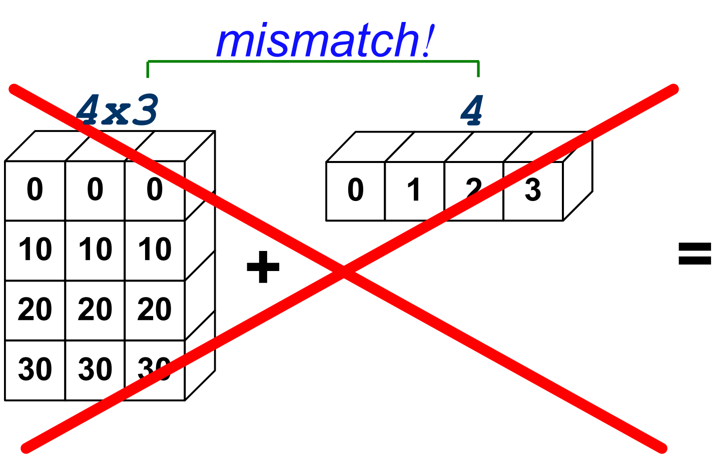
Broadcasting



Broadcasting

- Broadcasting: it is possible when the number of columns of first array is equal to the number of rows of the second array, otherwise the following exception will be returned:

```
"ValueError: frames are not aligned"
```



SciPy

SciPy: Basic Info

- Scientific algorithm and functions library.
- Provides:
- Optimizations module
- Linear algebra modules
- Numeric integration
- Special functions
- Signal and image processing tools

- Website: http://www.scipy.org

SciPy: Basic Info

- Extensions:

- **scipy.constants**: physics and math constants
- **scipy.special**: physics and math special functions (elliptical, Bessel, hypergeometric, …)
- **scipy.integrate**: numerical and differential equations integration
- **scipy.optimize**: optimization (e.g., simulated annealing)
- **scipy.linalg**: numpy.linalg extension.
- **scipy.sparse**: sparse matrices handling

SciPy: Basic Info

- Other extensions:

- **scipy.interpolate**: data interpolation
- **scipy.fftpack**: Fast Fourier Transform
- **scipy.signal**: signal processing (filtering, correlations, convolutions, smoothing, …)
- **scipy.stats**: continuous and discrete probability distributions, statistics, test …

SciPy: Linear Algebra - `linalg`

- Import
```
>>> from scipy import linalg
```
- Main functions:
- Basic functions: `inv, solve, det, norm, lstsq, pinv`
- Decomposition: `eig, lu, svd, orth, cholesky, qr, schur`
- Matrix functions: `expm, logm, sqrtm, cosm, coshm`
- ...

SciPy: Linear Algebra - `linalg`

- Eigenvalues and eigenvectors
- `>>> a = numpy.array([[1,3,5], [2,5,1], [2,3,6]])`
- ```
  #eigenvalues and eigenvectors computation
  >>> aval, avec = linalg.eig(a)
  >>> aval
      array([ 9.39895873+0.j,
             -0.73379338+0.j,
              3.33483465+0.j])
  ```
- ```
  #eigenvectors are the columns of the
  "avec" matrix. For examples, the first
  vector is:
  >>> avec[:,0]
      array([-0.57028326,
             -0.41979215,
             -0.70608183])
  ```

SciPy: Linear Algebra - `linalg`

- Inverse matrix

```
>>> A = numpy.array([[1,2], [3,4]])
>>> linalg.inv(A)
    array([[-2. ,  1. ],
           [ 1.5, -0.5]])
#proof: A·A⁻¹ = I
>>> A.dot(linalg.inv(A))
    array([[1.000e+00,0.000e+00],
          [4.441e-16, 1.000e+00]])
```

- Determinant

```
>>> A = linalg.det(A)
    -2
```

SciPy: Linear Algebra - `linalg`

- Linear systems (Ax = b)

```
>>> A = numpy.array([[1,2], [3,4]])
>>> b = numpy.array([[5], [6]])

#solutions
>>> x = linalg.solve(A,b)
>>> x
    array([[-4. ],
           [ 4.5]])
#proof
>>> A.dot(x)-b
    array([[ 0.],
           [ 0.]])
```

SciPy: Integration - `integrate`

- Import

```
>>> from scipy import integrate
```

- Main functions:
- `quad` basic integration
- `dblquad` double integral
- `tplquad` triple integral
- `simps` integration by smaples
- `odeint` differential equations integration
- `...`

- NB: `quad`, `dblquad` e `tplquad` take functions as parameters

SciPy: Integration - `integrate`

- Basic integral

- ```
  #quad(func, infL, supL, …)
  #func: function being integrated
  #infL: lower bound
  #supL: upper bound
  >>> x = integrate.quad(sin, 0, numpy.pi)
  ```

- The function returns the solved integration and an error estimation
  ```
  >>> x
      (2.0, 2.220446049250313e-14)
  ```
  ```

ScyPy: Funzioni Statistiche - `stats`

- Random variables
- Several random variable distributions

  `norm` gaussian distribution (or normal)

  `chi2` chi-squared distribution

  `t` T-student distribution

  ...

- Gaussian variable

```
>>> from scipy.stats import norm
#normal distribution definition
>>> x = norm()
>>> x = norm(loc=3.5, scale=2.0)
#loc:mean, #scale: standard deviation
```

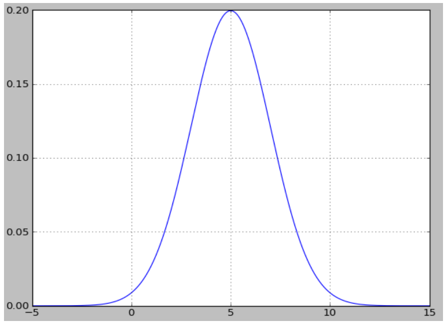
ScyPy: Funzioni Statistiche - `stats`

- Gaussian variable

```
>>> y = norm(loc=5, scale=2)
>>> x = numpy.linspace(-5, 5, 100)
```

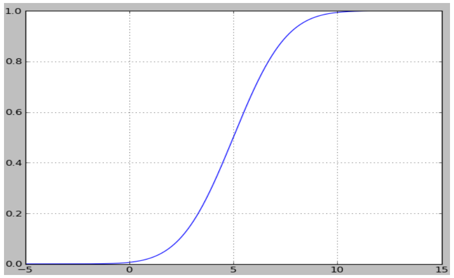
- `#probability density function`

```
>>> pdf = y.pdf(x)
```



- `#cumulative distribution function`

```
>>> cdf = y.cdf(x)
```



End !