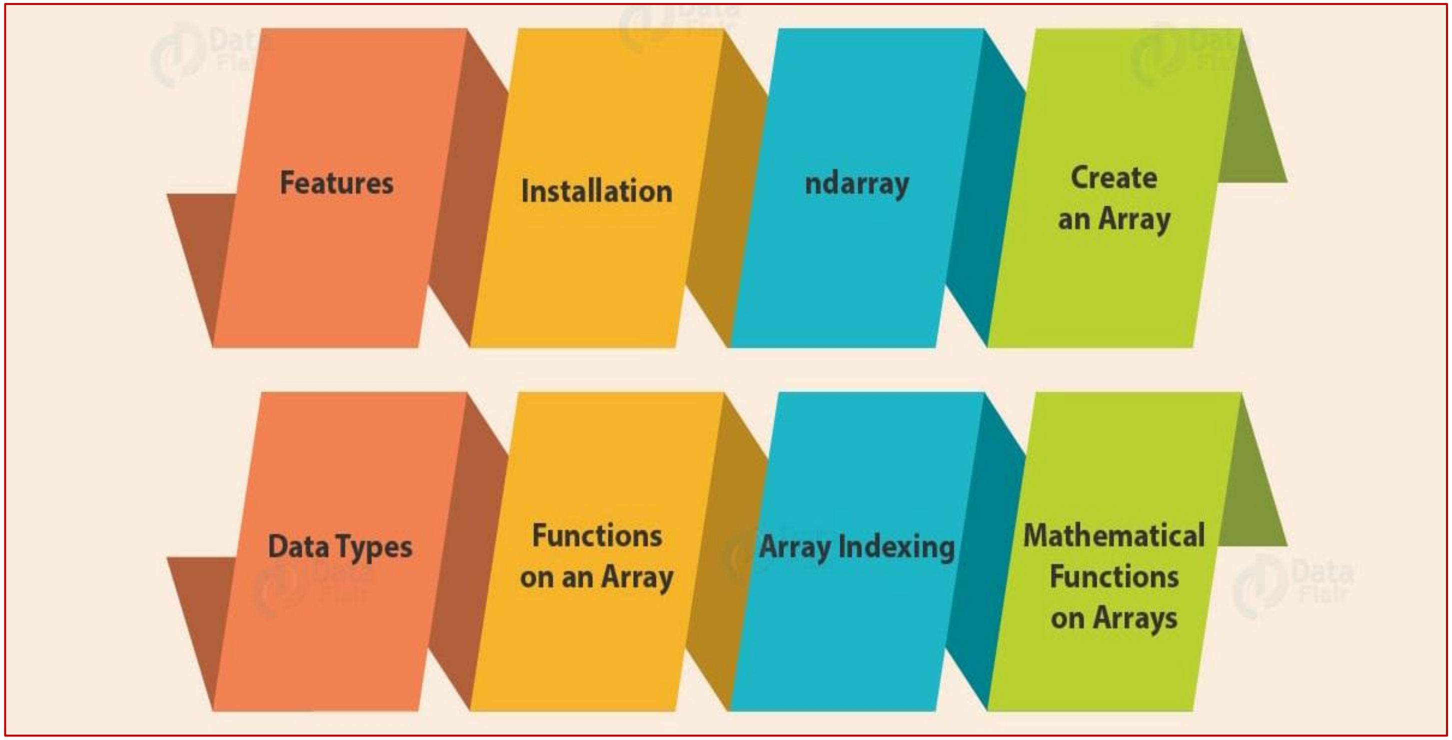
UNIT V
SCIPY ,NUMPY AND SIGNAL PROCESSING

- SciPy, numpy, matplotlib
- Basic array methods in numpy, Changing the shape of an array
- Maximum and minimum values
- Reading and writing an array to a fle
- Statistical methods in numpy
- Histograms
- Solving equations- Linear least squares solutions- Beer-Lambert Law
- One-Dimensional Fast Fourier Transforms
- Matplotlib basics- Plotting on a single axes object, scatter plot, Bar charts and pie charts
- Choosing the Length of the DFT
- Filters in Signal Processing
- Lab 13: numpy file reading and data analysis
- Lab 14: the correlation coefficient between pressure and temperature
- Lab 15: Numpy signal processing

Courtesy:Christian Hill- Learning Scientific Programming

Prepared by
Dr.J.Subhashini



- NumPy has become the de facto standard package for general scientific programming in Python.
- Its core object is the ndarray, a multidimensional array of a single data type which can be sorted, reshaped, subject to mathematical operations and statistical analysis, written to and read from files, and much more.
- Numpy is implemented as precompiled C code and so approach the speed of execution of a program written in C itself;
- Second, NumPy supports *vectorization: a single operation can be carried out on an entire array,* rather than requiring an explicit loop over the array's elements

Features of Numpy

01 High-performace

02 Integrating code from C/C++

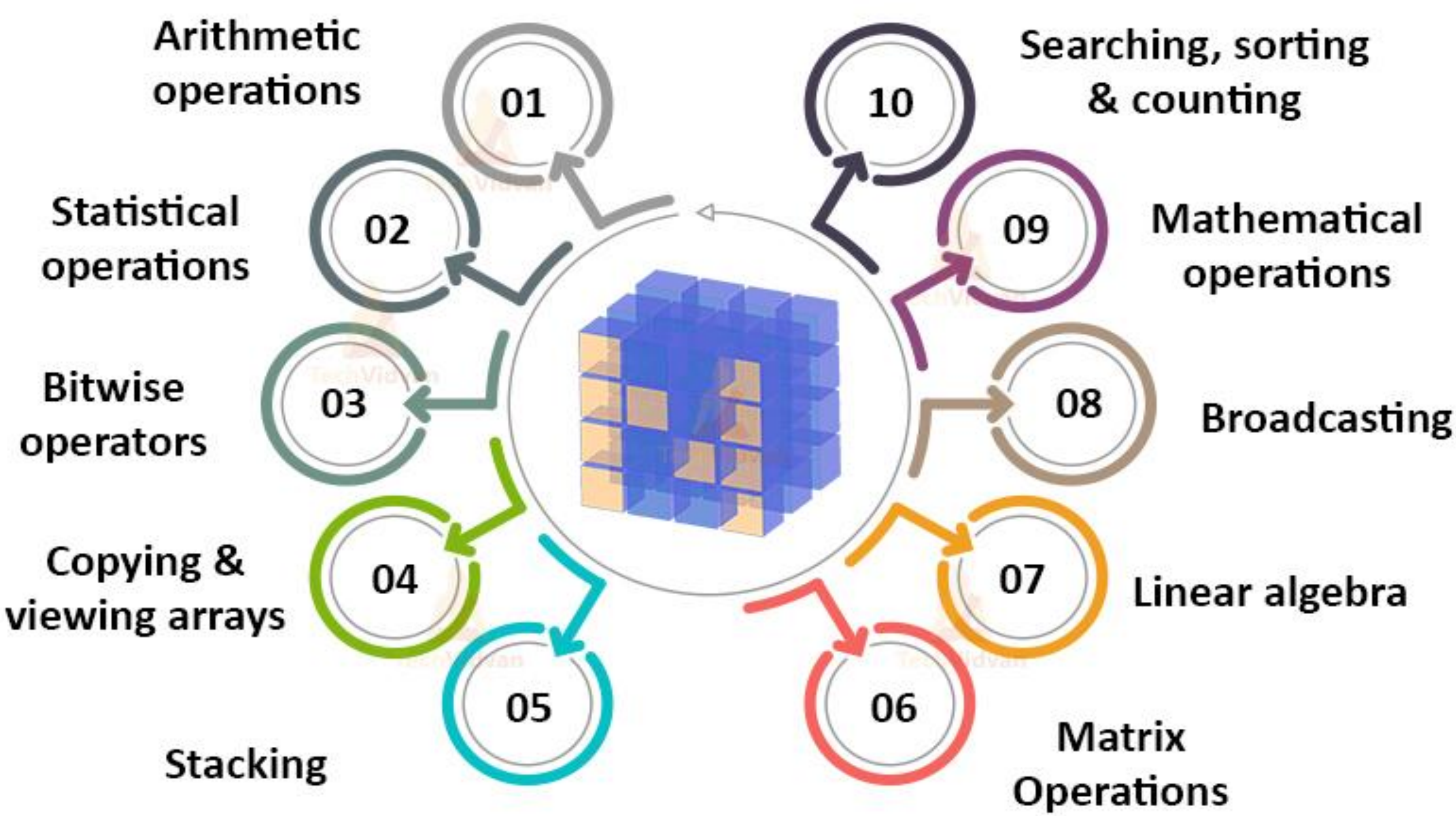
03 Multidimensional container

04 Broadcasting functions

05 Work with varied databases

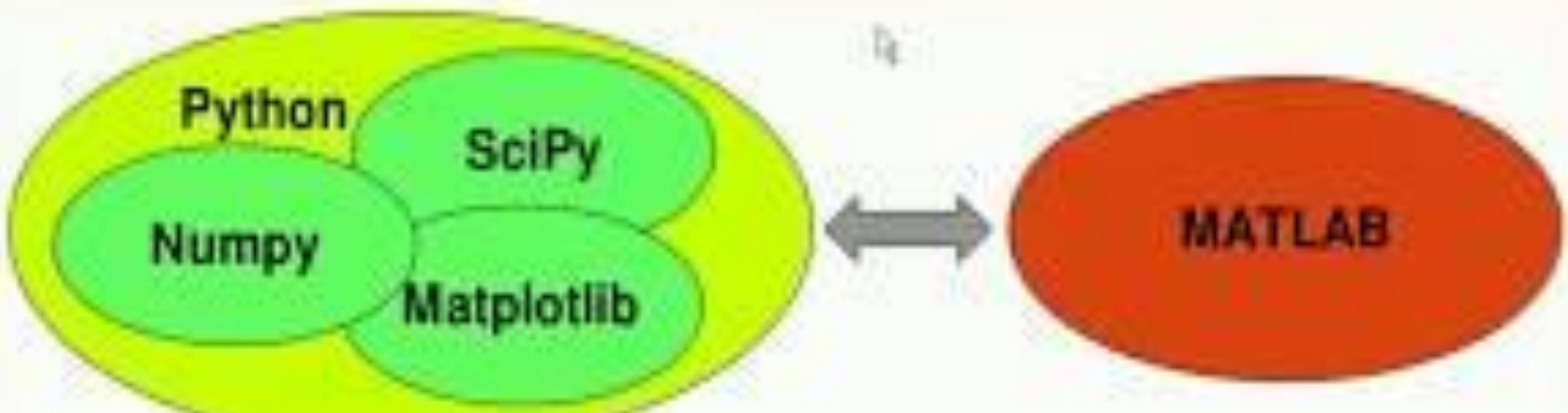
06 Additional Linear algebra

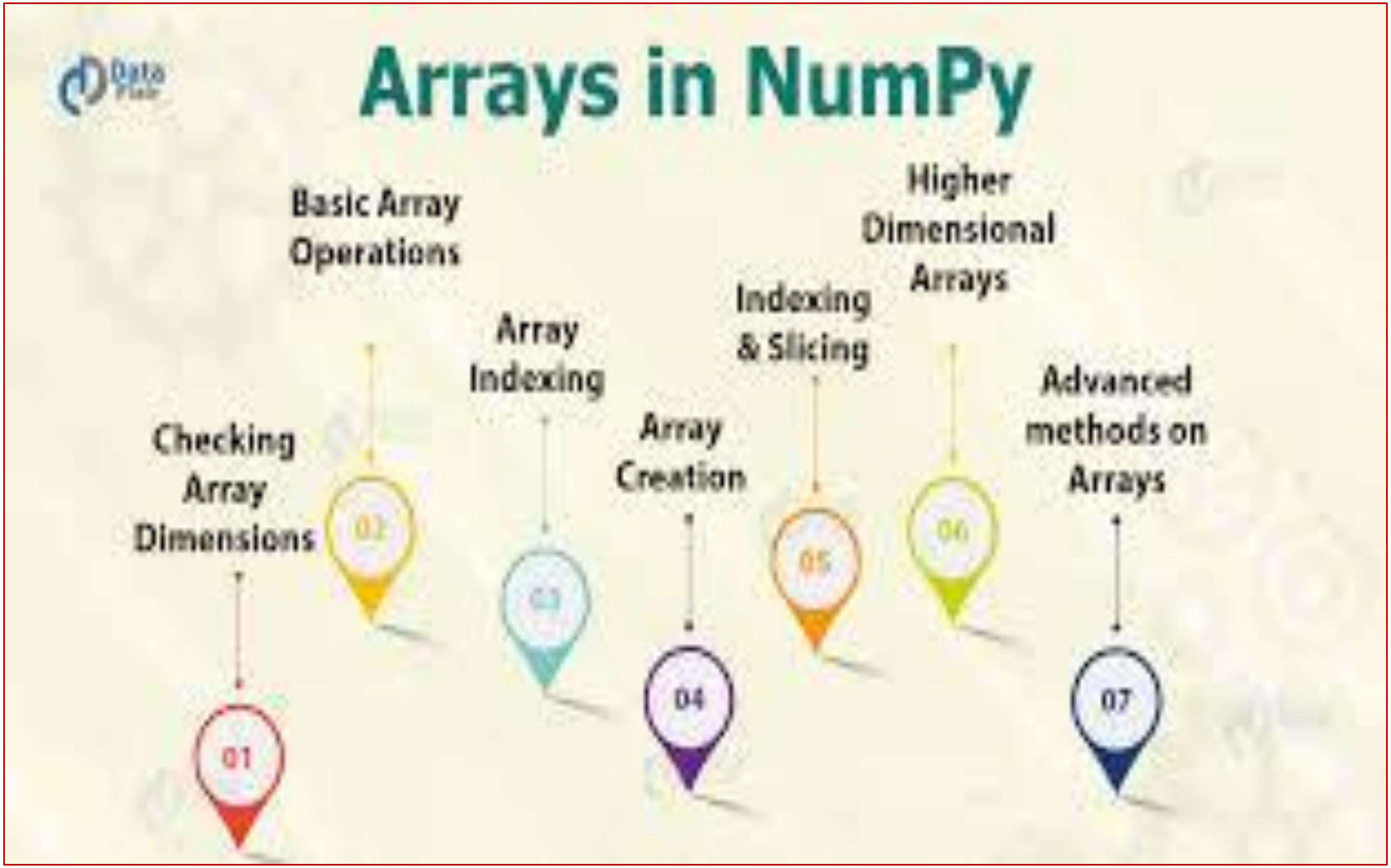
Uses of NumPy



What NumPy can do?

- array oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation





Arrays in NumPy
Checking Array Dimensions
01
Basic Array Operations
Array Indexing
Array Creation
04
Indexing & Slicing
05
Higher Dimensional Arrays
06
Advanced methods on Arrays
07

Basic array methods in numpy

- The NumPy array class is ndarray, which consists of a multidimensional table of elements indexed by a tuple of integers.
- Unlike Python lists and tuples, the elements cannot be of different types: each element in a NumPy array has the same type, which is specified by an associated data type object (dtype).
- *The dtype of an array specifies* not only the broad class of element (integer, floating point number, etc.) but also how it is represented in memory
- The dimensions of a NumPy array are called axes*; the number of axes an array has is* called its rank.

Creating an array- Basic array creation

Example 1:

```
import numpy as np
a = np.array( (100, 101, 102, 103) )
a
Output:[array([100, 101, 102, 103])
b = np.array( [[1.,2.], [3.,4.]] )
Output: array([[ 1., 2.],
[ 3., 4.]])
```

Example 2: data type using the optional dtype argument

```
np.array([0, 4, -4], dtype=complex)
Output:array([ 0.+0.j, 4.+0.j, -4.+0.j])
```

Basic array methods in numpy

The simplest and fastest, np.empty, takes a tuple of the array's shape and creates the array without initializing its elements, the initial element values are undefined

Creating an array- Basic array creation
Example 3:

```
np.empty((2,2))
Output:array([[ -2.31584178e+077, -1.72723381e-077],
[ 2.15686807e-314, 2.78134366e-309]])
```

np.zeros and np.ones, which create an array of the specified shape with elements prefilled with 0 and 1 respectively.

- np.empty, np.zeros and np.ones also take the optional dtype argument

Example 4:

```
np.zeros((3,2)) # default dtype is 'float'
Output:array([[ 0., 0.],
[ 0., 0.],
[ 0., 0.]])
np.ones((3,3), dtype=int)
Output: array([[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
```

Basic array methods in numpy

•To create an array containing a sequence of numbers there are two methods: np.arange and np.linspace.

•np.arange is the NumPy equivalent of range(), except that it can generate floating point sequences.

Creating an array- Initializing an array from a sequence
Example 5:

```
np.arange(7)
Output: array([0, 1, 2, 3, 4, 5, 6])
np.arange(1.5, 3., 0.5)
Output: array([ 1.5, 2. , 2.5]))
```

Basic array methods in numpy

NumPy's basic data types (dtypes)

Table 6.1 `ndarray` **Attributes**

Attribute	Description
`shape`	The array dimensions: the size of the array along each of its axes, returned as a tuple of integers
`ndim`	Number of axes (dimensions). Note that `ndim == len(shape)`
`size`	The total number of elements in the array, equal to the product of the elements of `shape`
`dtype`	The array's data type (see Section 6.1.2)
`data`	The "buffer" in memory containing the actual elements of the array
`itemsize`	The size in bytes of each element

Table 6.3 Common NumPy data type strings

String	Description
`i`	Signed integer
`u`	Unsigned integer
`f`	Floating point number[a]
`c`	Complex floating point number
`b`	Boolean value
`S`, `a`	String (fixed-length sequence of characters)
`U`	Unicode

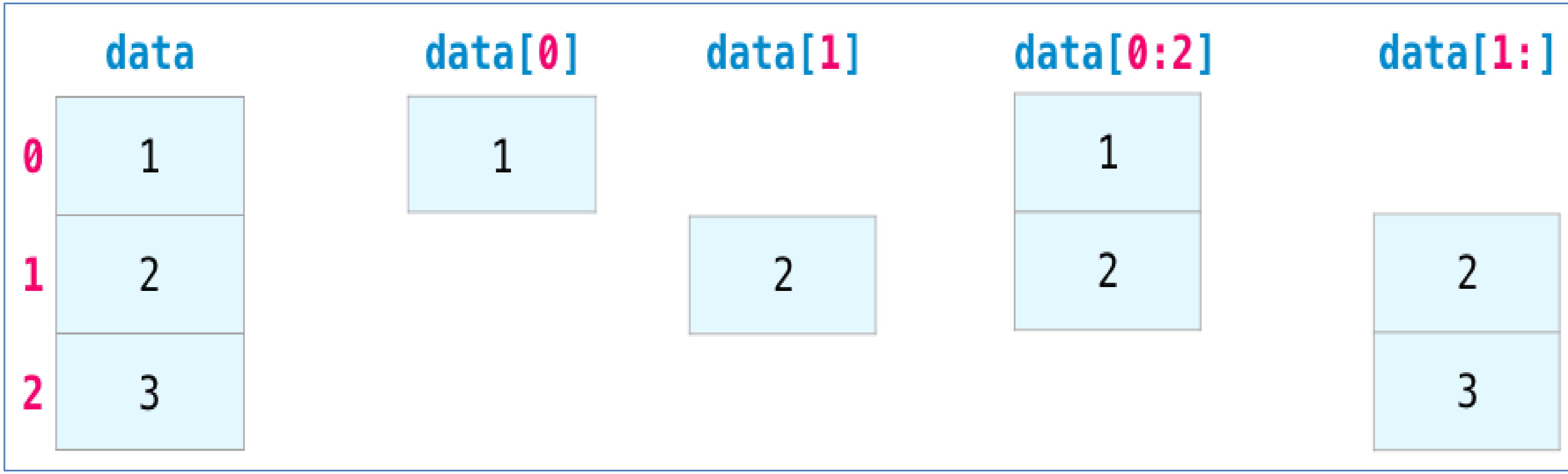
Basic array methods in numpy

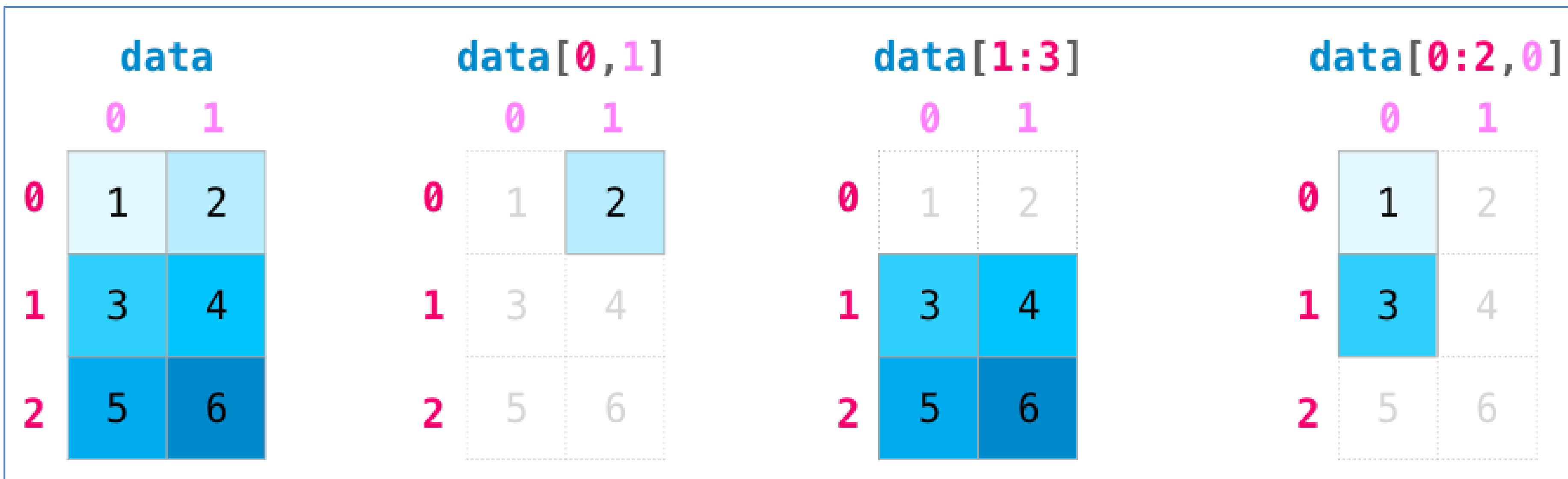
NumPy's basic data types (dtypes)

Table 6.2 Commmon NumPy data types

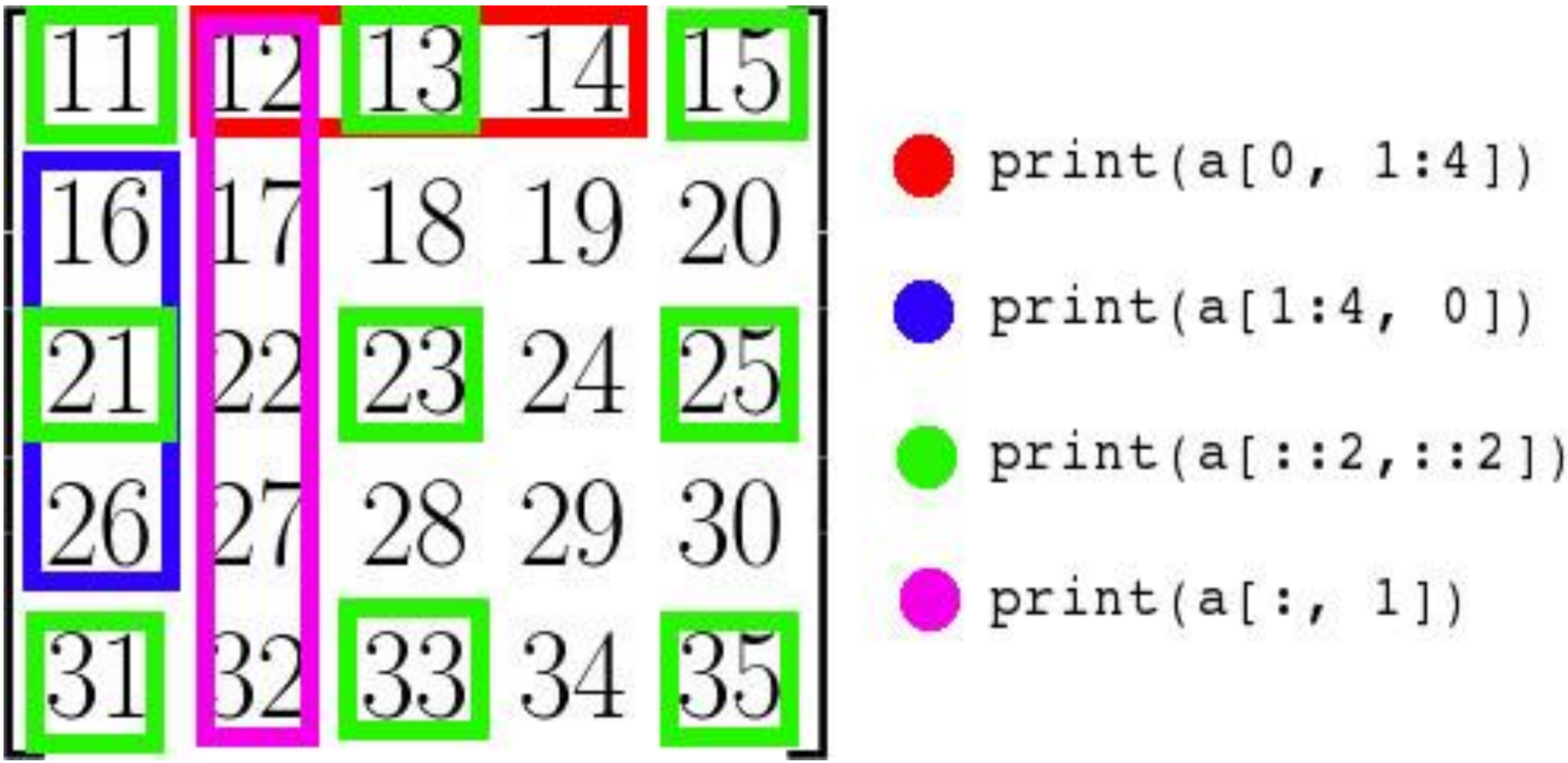
Data Type	Description
`int_`	The default integer type, corresponding to C's `long`: *platform-dependent*
`int8`	Integer in a single byte: -128 to 127
`int16`	Integer in 2 bytes: -32768 to 32767
`int32`	Integer in 4 bytes: -2147483648 to 2147483647
`int64`	Integer in 8 bytes: -2^{63} to $2^{63}-1$
`uint8`	Unsigned integer in a single byte: 0 to 255
`uint16`	Unsigned integer in 2 bytes: 0 to 65535
`uint32`	Unsigned integer in 4 bytes: 0 to 4294967295
`uint64`	Unsigned integer in 8 bytes: 0 to $2^{64}-1$
`float_`	The default floatng point number type, another name for `float64`
`float32`	Single-precision, signed float: $\sim 10^{-38}$ to $\sim 10^{38}$ with ~ 7 decimal digits of precision
`float64`	Double-precision, signed float: $\sim 10^{-308}$ to $\sim 10^{308}$ with ~ 15 decimal digits of precision
`complex_`	The default complex number type, another name for `complex128`
`complex64`	Single-precision complex number (represented by 32-bit floating point real and imaginary components)
`complex128`	Double-precision complex number (represented by 64-bit floating point real and imaginary components)
`bool_`	The default boolean type represented by a single byte

Basic array indexing



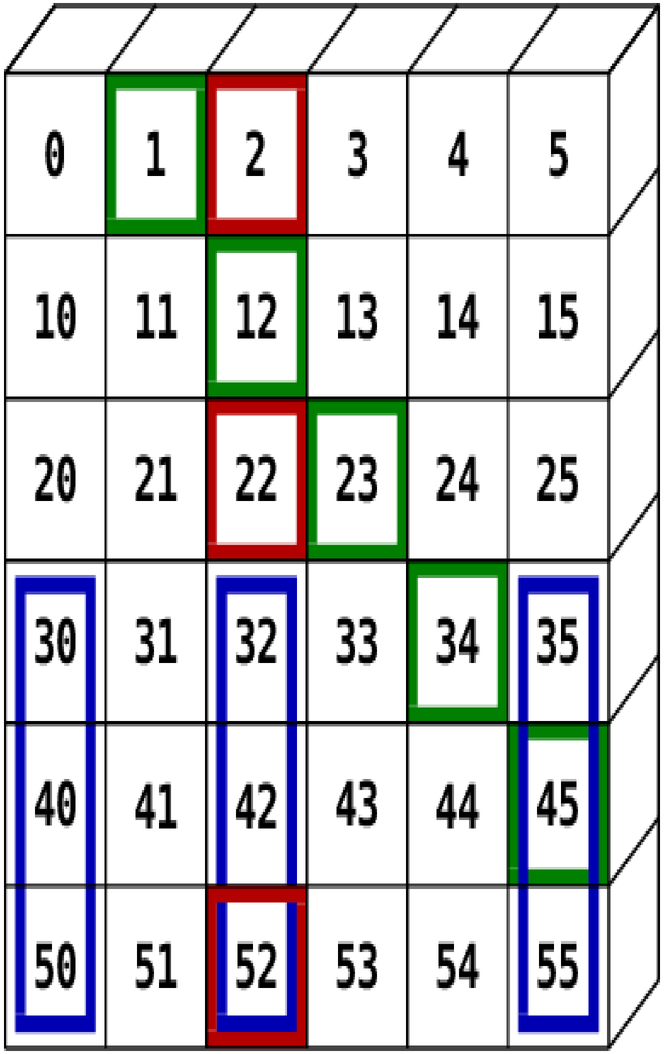


Basic array indexing

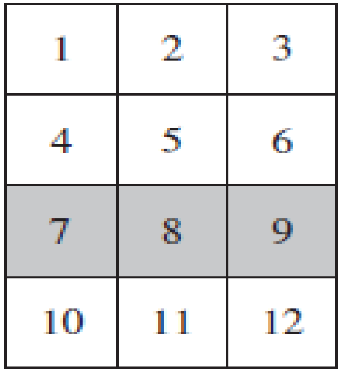


Basic array indexing

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

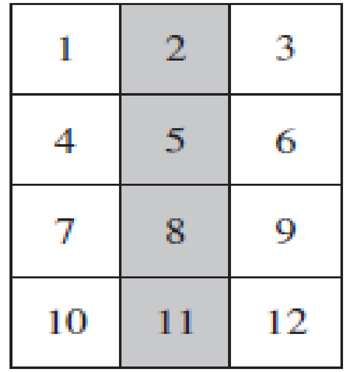


Slicing an array



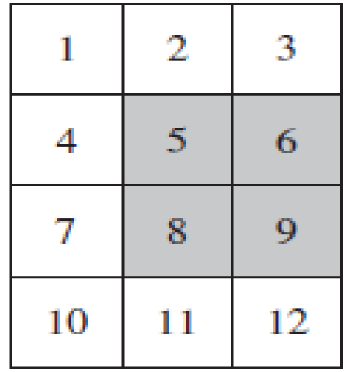
`a[2, :]`

(a)



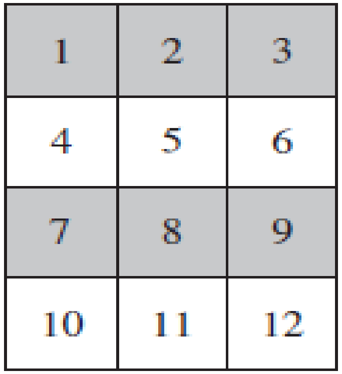
`a[:, 1]`

(b)



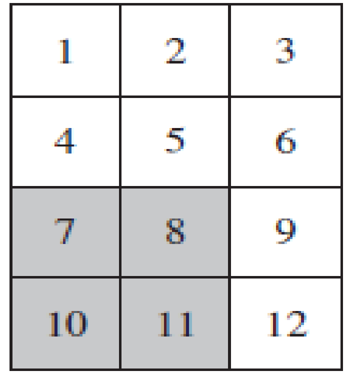
`a[1:-1, 1:]`

(c)



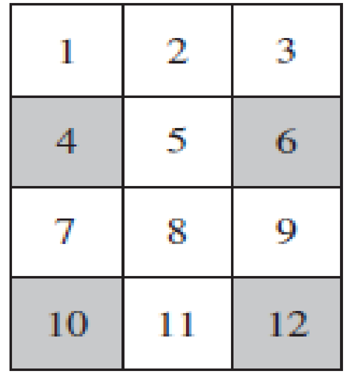
`a[::2, :]`

(d)



`a[2:, :2]`

(e)



`a[1::2, ::2]`

(f)

Changing the shape of an array

flatten and ravel

flatten() returns an independent *copy of the* elements and is generally slower than ravel() which, tries to return a *view to the flattened* array.

```
a = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
b = a.flatten() # create and independent, flattened copy of ‘a’
b
Output: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
b[3] = 0; b
Output: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
a # a is unchanged
Output:
array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
c = a.ravel()
Output: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
c[3] = 0
Output: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
a
Output:
array([[1, 2, 3],
[0, 5, 6],
[7, 8, 9]])
```

Changing the shape of an array

resize and reshape

- An array may be resized (in place) to a compatible shape with the resize() method, which takes the new dimensions as its arguments.
- If the array doesn't reference another array's data and doesn't have references to it, resizing to a smaller shape is allowed and truncates the array; resizing to a larger shape pads with zeros.

```
a = np.linspace(1, 4, 4) # the array [1. 2. 3. 4.]
 print(a)
Output: [1. 2. 3. 4.]
a.resize(2,2) # reshapes a in place, doesn't return
anything
print(a)
Output:
[[ 1. 2.]
[ 3. 4.]]
a.resize(3,2) # OK: nothing else references a
 print(a)
Output:
[[ 1. 2.]
[ 3. 4.]
[ 0. 0.]]
```

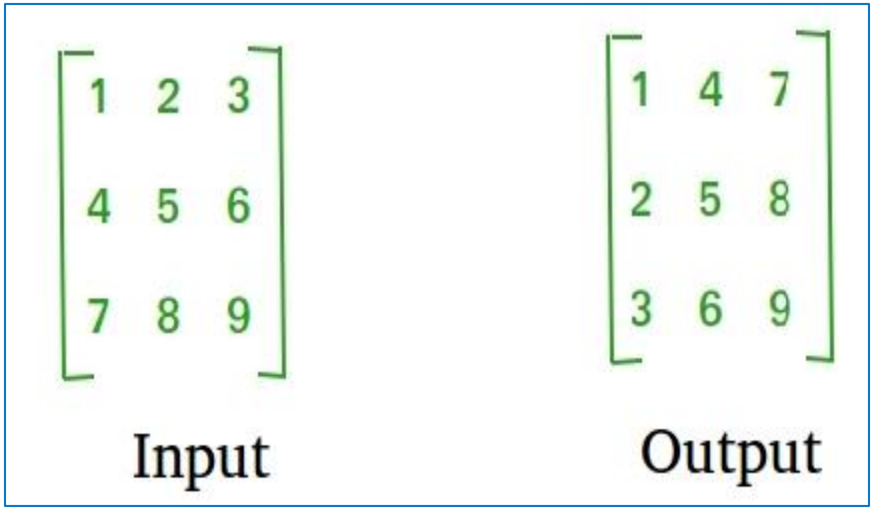
Changing the shape of an array

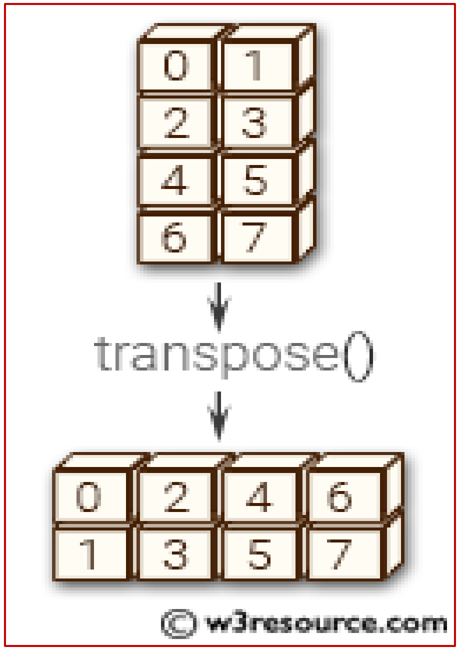
resize and reshape

- The reshape() method returns a view on the array with its elements reshaped as required.
- The original array is not modified.

```
a = np.linspace(1, 4, 4)
a.resize(3,2)
a
Output:
[[ 1. 2.]
[ 3. 4.]
[ 0. 0.]]
b = a.reshape(6)
 print(b)
Output:
[ 1. 2. 3. 4. 0. 0.]
b.resize(3,2) # OK: same number of elements
 b.resize(2,2) # not OK: b is a view on (shares) the same data as
a
Output:
ValueError: cannot resize this array: it does not own its data
a.resize(2,2) # also not OK: a shares its data with b
ValueError: cannot resize this array: it does not own its data
```

Changing the shape of an array





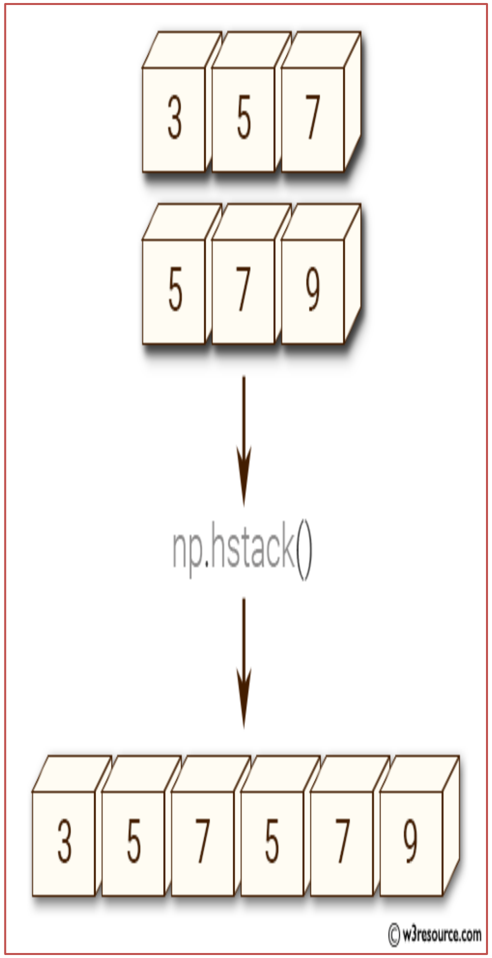
a.transpose() # or simply a.T

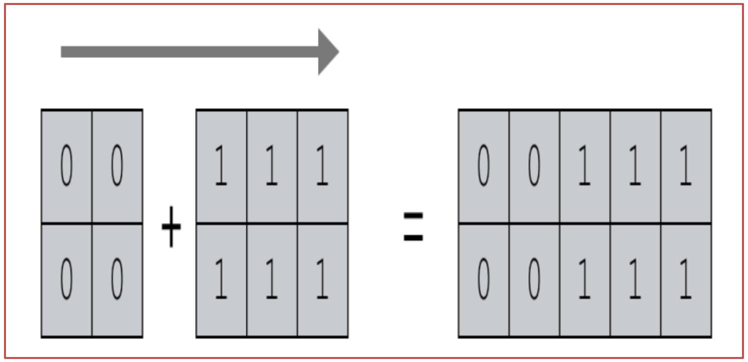
Changing the shape of an array

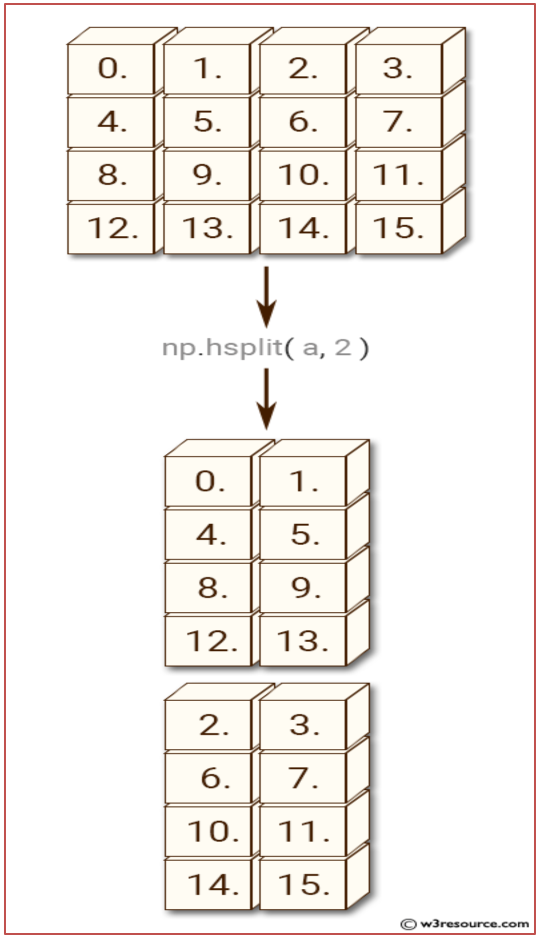
Merging and splitting arrays

•A clutch of NumPy methods merge and split arrays in different ways.

•np.vstack, np.hstack and np.dstack stack arrays vertically (in sequential rows), horizontally (in sequential columns) and depthwise (along a third axis).

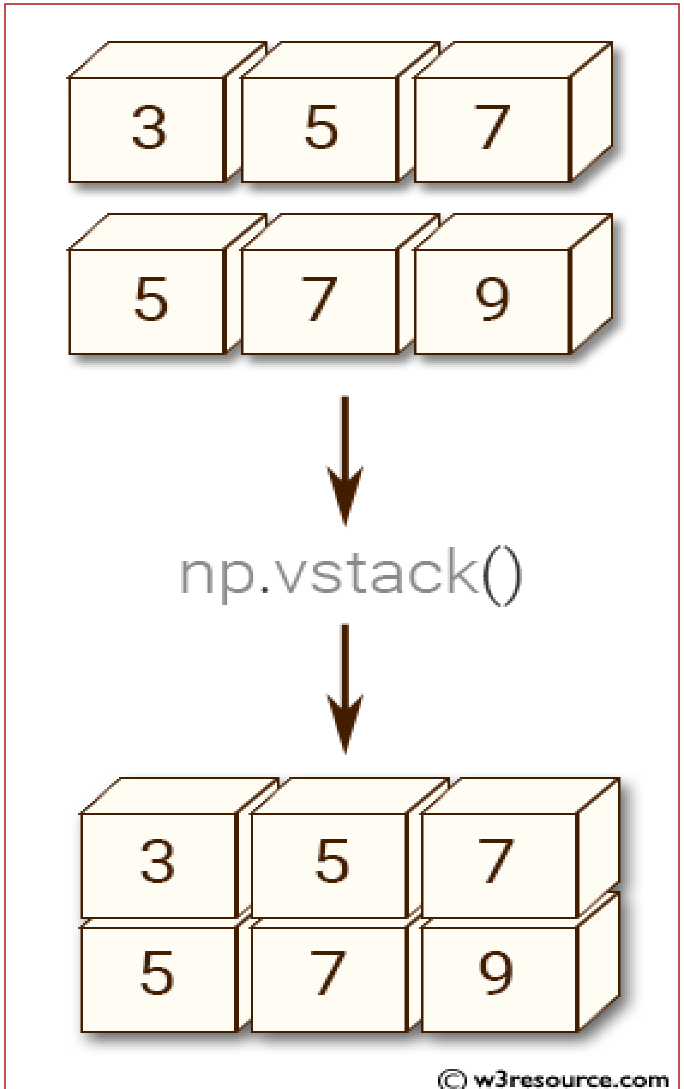


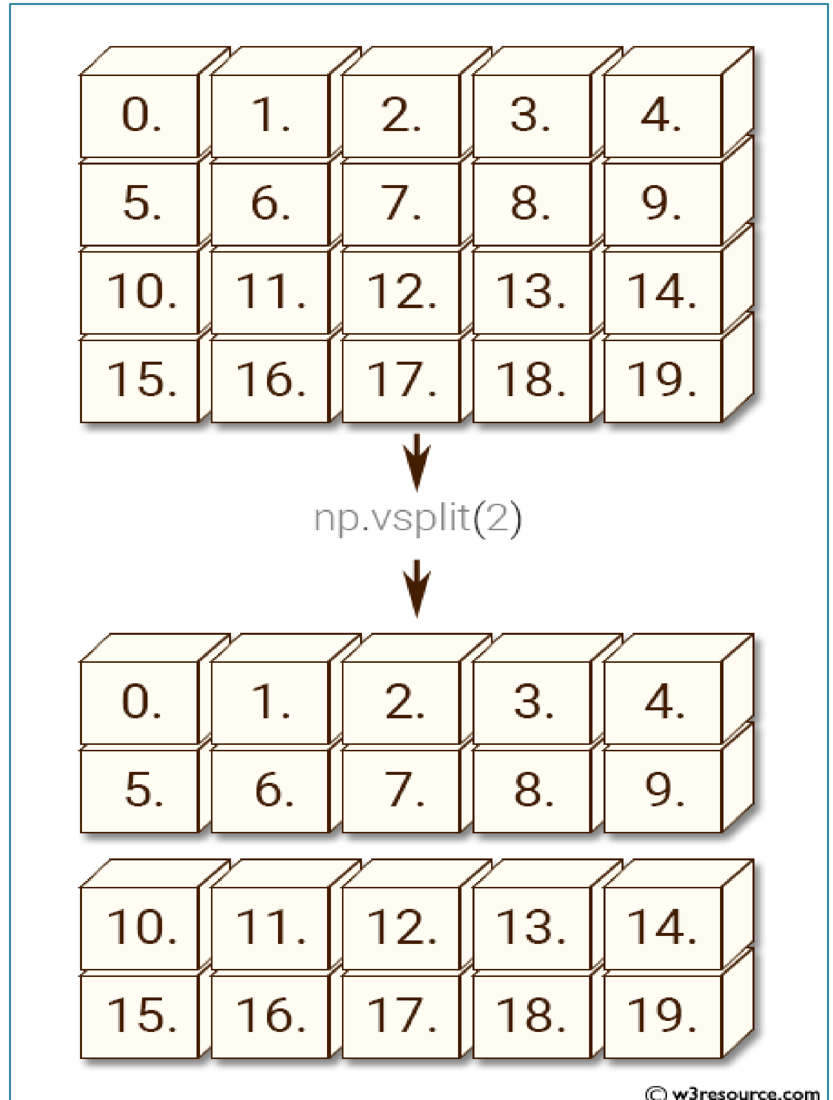




Changing the shape of an array

Merging and splitting arrays





Changing the shape of an array

Merging and splitting arrays

The inverse operations, np.vsplit, np.hsplit and np.dsplit split a single array into multiple arrays by rows, columns or depth.

```
a = np.array([0, 0, 0, 0])
 b = np.array([1, 1, 1, 1])
 c = np.array([2, 2, 2, 2])
 np.vstack((a,b,c))
Output:
array([[0, 0, 0, 0],
[1, 1, 1, 1],
[2, 2, 2, 2]])
np.hstack((a,b,c))
Output:
array([0, 0, 0, 0, 1, 1, 1,
1, 2, 2, 2, 2])
np.dstack((a,b,c))
Output:
array([[[0, 1, 2],
[0, 1, 2],
[0, 1, 2],
[0, 1, 2]]])
```

```
a = np.arange(6)
a
Output: array([ 0, 1, 2, 3, 4, 5])
np.hsplit(a, 3)
Output: [array([ 0, 1]), array([ 2,
3]), array([ 4, 5])]
a
Output: array([ 0, 1, 2, 3, 4, 5])
np.hsplit(a, (2, 3, 5))
Output:
[array([0, 1]), array([2]), array([3,
4]), array([5])]
```

Maximum and minimum values

- NumPy arrays have the methods *min* and *max*, which return the minimum and maximum values in the array.
- By default, a single value for the flattened array is returned; to find maximum and minimum values along a given axis, use the *axis* argument:

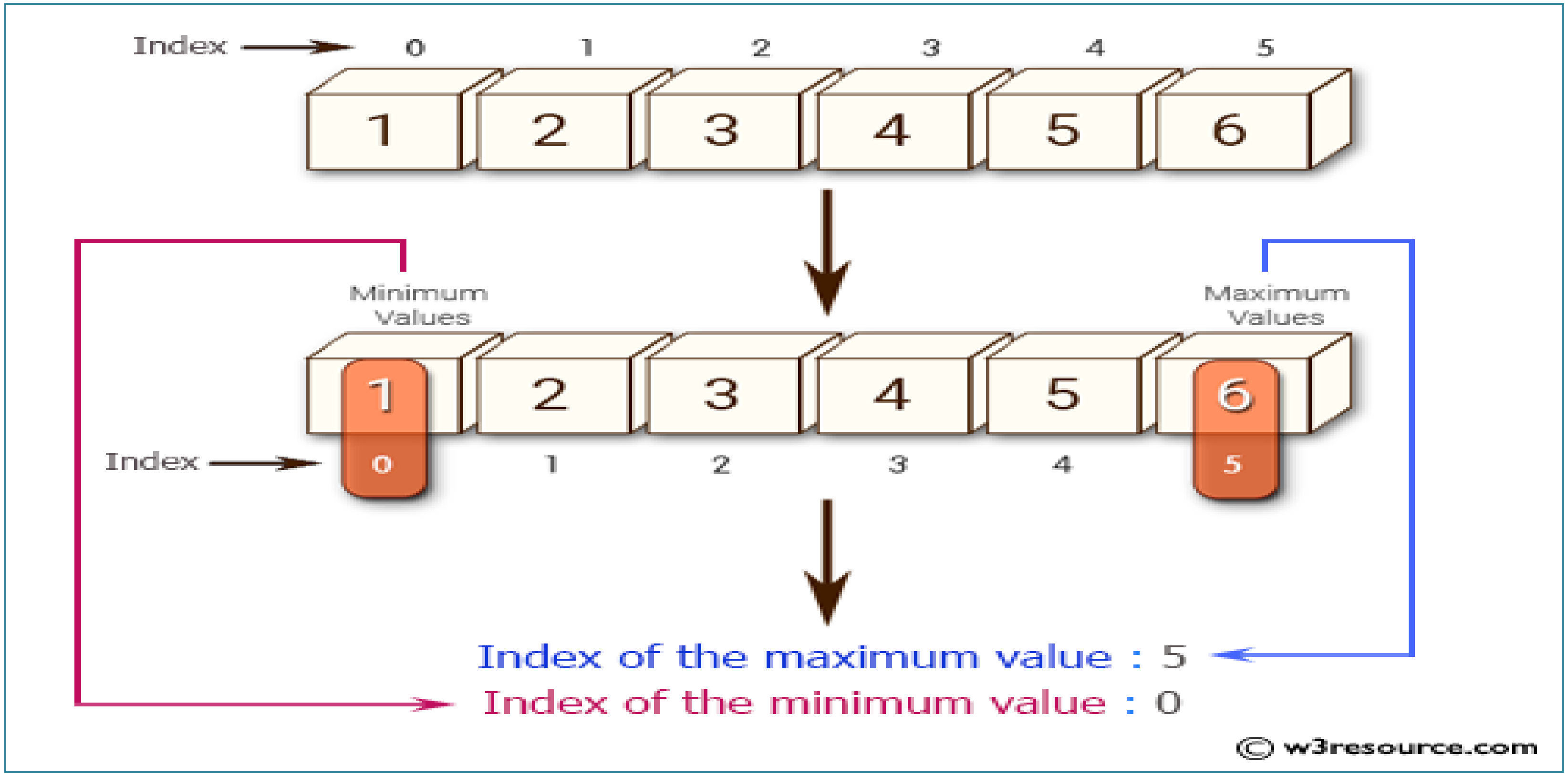
```
a = np.array([[3, 0, -1, 1], [2, -1, -2, 4], [1, 7, 0, 4]])
print(a)
Output:
[[ 3  0 -1 1]
[  2 -1 -2 4]
[  1  7  0 4]]

a.min() # "global" minimum
Output: -2
a.max() # "global" maximum
Output: 7
print( a.min(axis=0) )
Output:[ 1 -1 -2 1] # minima in each column
print( a.max(axis=1) )
Output:[3 4 7] # maxima in each row
```

Maximum and minimum values

•Often one wants not the maximum (or minimum) value itself but its index in the array.

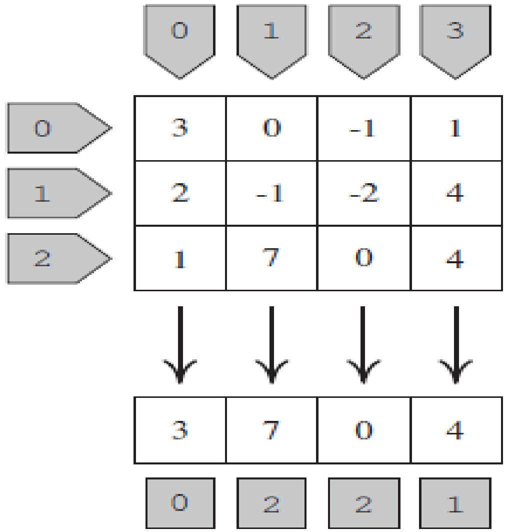
•This is what the methods *argmin* and *argmax* do.

•By default, the index returned is into the flattened array, so the actual value can be retrieved using a view on the array created by ravel

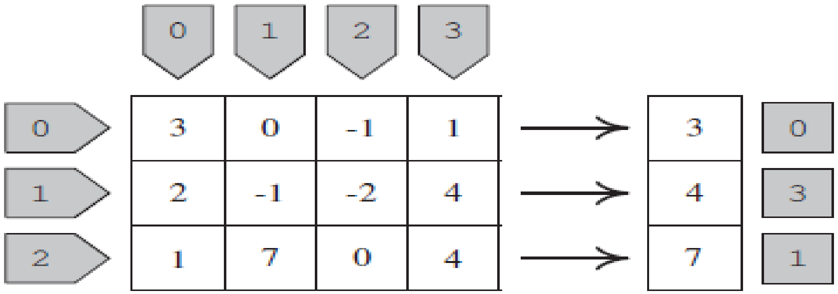


Maximum and minimum values

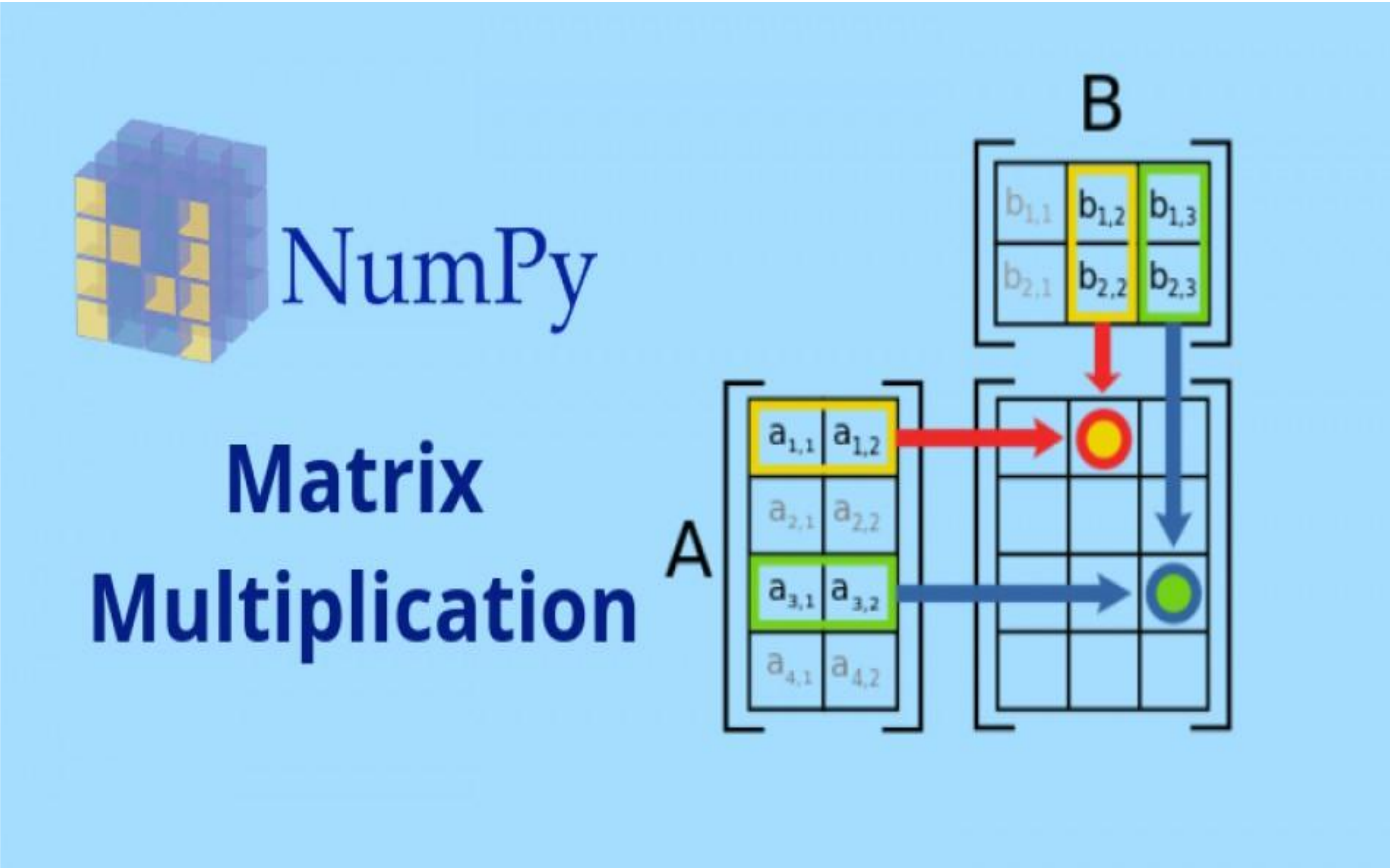
```
[3, 0, -1, 1 ,2, -1, -2, 4, 1, 7, 0, 4]- Flattened array
a.argmin()
Output: 6
 a.ravel()[a.argmin()]
Output: -2
 print(a.argmax(axis=0))
Output: [0 2 2 1] # row indexes of maxima in each column
 print(a.argmax(axis=1))
Output: [0 3 1] # column indexes of maxima in each row
```



(a) axis=0



(b) axis=1


NumPy
Matrix
Multiplication
B
b1,1 b1,2 b1,3
b2,1 b2,2 b2,3
A
a1,1 a1,2
a2,1 a2,2
a3,1 a3,2
a4,1 a4,2

Reading and writing an array to a file

•Scientific data are frequently read in from a text file, which may contain comments, missing values and blank lines.

•Columns of values may be either aligned in a fixed width format or separated by one or more delimiting characters (such as spaces, tabs or commas).

•Furthermore, there may be a descriptive header and even footnotes to the file,
•which make it hard to parse directly using Python’s string methods.

•NumPy provides several functions for reading data from a text file.

•The simpler np.loadtxt handles many common cases; the more sophisticated np.genfromtxt allows for better handling of missing values and footers.

```
np.save(’my-array.npy’, a)
a = np.load(’my-array.npy’)
```

Reading and writing an array to a file

np.loadtxt(fname, dtype=<class ’float’>, comments=’#’, delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)

The arguments are as follows:

- fname: The only required argument, fname, which can be a filename, an open file, or a generator returning the lines of data to be parsed.
- dtype: The data type of the array defaults to float but can be set explicitly by the dtype argument.
- comments: Comments in a file are usually started by some character such as # (as with Python) or %.
- delimiter: The string used to separate columns of data in the file; by default it is None, meaning that any amount of whitespace (spaces, tabs) delimits the data. To read a comma-separated (csv) file, set delimiter=’,’.
- converters: An optional dictionary mapping the column index to a function converting string values in that column to data (e.g., float).
- skiprows: An integer giving the number of lines at the start of the file to skip over before reading the data (e.g., to pass over header lines). Its default is 0 (no header).
- usecols: A sequence of column indexes determining which columns of the file to return as data; by default it is None, meaning all columns will be parsed and returned.
- unpack: By default, the data table is returned in a single array of rows and columns reflecting the structure of the file read in. Set unpack=True will transpose this array so that individual columns can be picked off an assigned to different variables.
- ndmin: The minimum number of dimensions the returned array should have. By default, 0 (so a file containing a single number is read in as a scalar), it can be set to 1 or 2.

Statistical methods in numpy

- If the array contains one or more NaN values, the corresponding minimum or maximum value will be np.nan.
- To ignore NaN values instead, use np.nanmin and np.nanmax:

```
a = np.sqrt(np.linspace(-2, 2, 4))
print(a)
Output: [ nan   nan   0.    1.    1.41421356]

np.min(a), np.max(a)
Output: (nan, nan)

np.nanmin(a), np.nanmax(a)
Output: (0.0, 1.4142135623730951)

np.argmin(a), np.argmax(a)
Output: (0, 0) # The first nan in the array

np.nanargmin(a), np.nanargmax(a)
Output: (2, 4) # The indexes of 0, 1.41421356
```

Statistical methods in numpy

•The related methods, np.fmin / np.fmax and np.minimum / np.maximum, compare two arrays, element by element and return another array of the same shape.

•The first pair of methods ignores NaN values and the second pair propagates them into the output array

```
np.fmin([1, -5, 6, 2], [0, np.nan, -1, -1])
Output: array([ 0., -5., -1., -1.]) # NaNs are ignored

np.maximum([1, -5, 6, 2], [0, np.nan, -1, -1])
Output: array([ 1., nan, 6., 2.]) # NaNs are propagated
```

Percentiles

- Data set:
- 2,2,3,4,5,5,5,6,7,8,8,8,8,8,9,9,10,11,11,12
- What Value Exist at the **percentile** ranking of $25^{\%}$?

$$\textbf{Value}\ \# = \frac{percentile}{100}(n+1)$$

$$\text{Value}\ \# = \frac{25}{100}(20+1) = 5.\ 25$$

There is no "5.25^{th}", so I take the average of the 5^{th} & 6^{th} values to find what value exist at the 25^{th} percentile.

$$\frac{5+5}{2} = 5$$

Percentiles

```
a = np.array([[0., 0.6, 1.2], [1.8, 2.4, 3.0]])
np.percentile(a, 50)
Output:1.5

np.percentile(a, 75)
Output:2.25

np.percentile(a, 50, axis=1)
Output: array([ 0.6, 2.4])

np.percentile(a, 75, axis=1)
Output: array([ 0.9, 2.7])
```

Averages

NumPy provides methods for calculating the weighted average, median, standard deviation and variance

```
x = np.array([1., 4., 9., 16.])
np.mean(x)
Output:7.5
np.median(x)
Output:6.5
np.average(x, weights=[0., 3., 1., 0.])
Output:5.25  # ie (3.* 4. + 1.*9.) / (3. + 1.)
x = np.array( [[1., 8., 27], [-0.5, 1., 0.]] )
av, sw = np.average(x, weights=[0., 1., 0.1], axis=1,
returned=True)
print(av)
Output:[ 9.72727273 0.90909091]
print(sw)
Output:[ 1.1 1.1]
```

The averages are therefore $(1 \times 8 + 0.1 \times 27)/1.1 = 9.72727273$ and $(1 \times 1.)/1.1 = 0.90909091$ where 1.1 is the sum of the weights.

Standard deviations and variances

The function np.std calculates, by default, the uncorrected sample standard deviation:

$$\sigma_N = \sqrt{\frac{1}{N}\sum_i^N (x_i - \bar{x})^2}.$$

To calculate the corrected sample standard deviation,

$$\sigma = \sqrt{\frac{1}{N-\delta}\sum_i^N (x_i - \bar{x})^2},$$

```
x = np.array([1., 2., 3., 4.])
np.std(x) # or x.std(), #uncorrected standard deviation
Output:1.118033988749895
np.std(x, ddof=1) # corrected standard deviation
Output:1.2909944487358056
```

The covariance is returned by the np.cov method

$$C_{ij} = \frac{1}{N-1}\sum_k [(x_{ik} - \mu_i)(x_{jk} - \mu_j)]$$

Covariance and Correlation coefficient

```
X = np.array([ [0.1, 0.3, 0.4, 0.8, 0.9],
[3.2, 2.4, 2.4, 0.1, 5.5],
[10., 8.2, 4.3, 2.6, 0.9] ])

print( np.cov(X) )
[[ 0.115 , 0.0575, -1.2325],
[ 0.0575, 3.757 , -0.8775],
[ -1.2325, -0.8775, 14.525 ]]

print(np.var(X, axis=1, ddof=1))
[ 0.115 3.757 14.525]
```

The correlation coefficient matrix is often used in preference to the covariance matrix as it is normalized by dividing Cij by the product of the variables' standard deviations:

$$P_{ij} = \mathrm{corr}(x_i, x_j) = \frac{C_{ij}}{\sigma_i \sigma_j} = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}.$$

Unit-5,Session 6-15

Histograms
Solving equations- Linear least squares solutions- Beer-Lambert Law
Lab 14: the correlation coefficient between pressure and temperature
One-Dimensional Fast Fourier Transforms
Matplotlib basics- Plotting on a single axes object, scatter plot, Bar charts and pie charts
Choosing the Length of the DFT
Filters in Signal Processing
Lab 15: Numpy signal processing

Reference:

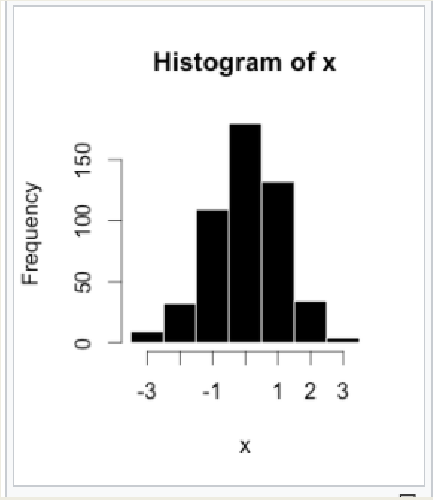
2. Christian Hill, "Learning Scientific Programming with Python", Cambridge University Press, 2015.

Prepared by
Dr.V.Sarada

Histograms

- A histogram represents the distribution of data as a series of (usually vertical) bars with lengths in proportion to the number of data items falling into predefined ranges (known as bins).
- the range of data values is divided into intervals and the histogram constructed by counting the number of data values in each interval.

Bin	Count
−3.5 to −2.51	9
−2.5 to −1.51	32
−1.5 to −0.51	109
−0.5 to 0.49	180
0.5 to 1.49	132
1.5 to 2.49	34
2.5 to 3.49	4



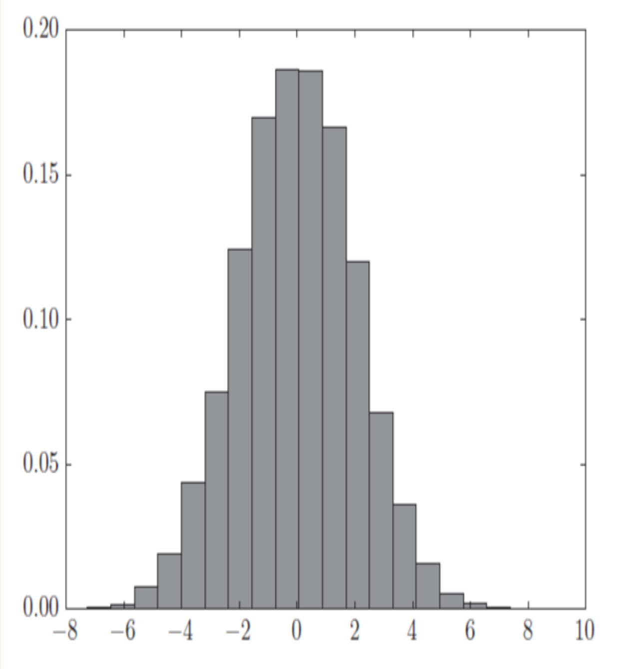
Histogram-Function

- **The pylab function hist** produces a histogram from a sequence of data values.

- The **number of bins** can be passed as an **optional** argument, bins; its default value is 10.

- attribute **normed=True** , its area (the height times width of each bar summed over the total number of bars) is unity.

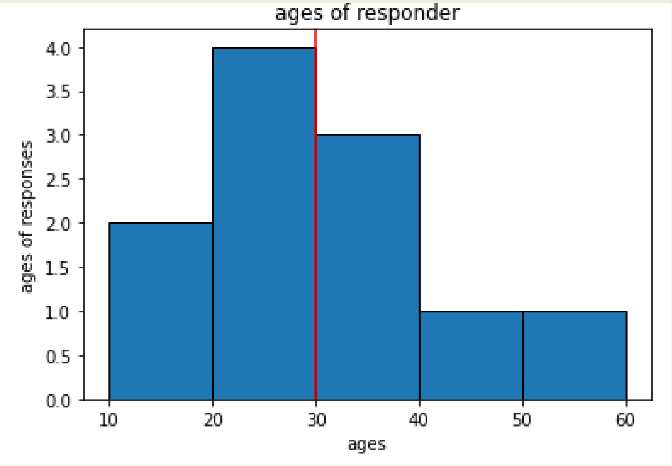
- For example, take 5,000 random values from the normal distribution with mean 0 and standard deviation 2



Histogram-Example

Program:

```
from matplotlib import pyplot as plt
ages = [18,19,21,25,26,26,30,32,38,45,55]
#pylab.hist(ages, bins=5,edgecolor="black")
#pylab.hist(ages, bins=5,edgecolor="black“)
bins=[10,20,30,40,50,60]
pylab.hist(ages, bins=bins,edgecolor="black“)
median_age=30
plt.axvline(median_age,color="r",label="age_median")
plt.xlabel("ages")
plt.ylabel("ages of responses")
plt.title("ages of responder")
#plt.tight_layout()
pylab.show()
```

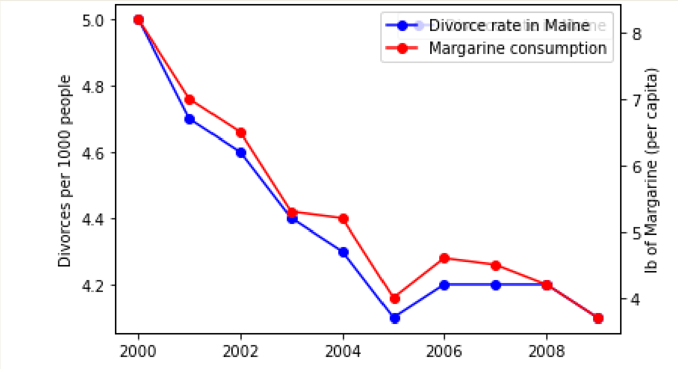


Multiple axes

- The command plot.twinx() starts a new set of axes with the same x-axis as the one, but a new y-scale
- This is useful for plotting two or more data series,which share x-axis but with y values which correlation between margarine consumption in the United States and the divorce rate in Maine

Program:

```
years = range(2000, 2010)
divorce_rate = [5.0, 4.7, 4.6, 4.4, 4.3, 4.1, 4.2, 4.2, 4.2, 4.1]
margarine_consumption = [8.2, 7, 6.5, 5.3, 5.2, 4, 4.6, 4.5, 4.2, 3.7]
line1 = plt.plot(years, divorce_rate, "b-o",label="Divorce rate in Maine")
plt.ylabel("Divorces per 1000 people")
plt.legend()
plt.twinx()
line2 = pylab.plot(years, margarine_consumption, "r-o",label="Margarine consumption")
plt.ylabel("lb of Margarine (per capita)")
# Jump through some hoops to get the both line's labels in the same legend:
lines = line1 + line2
labels = []
for line in lines:
   labels.append(line.get_label())
plt.legend(lines, labels)
plt.show()
```



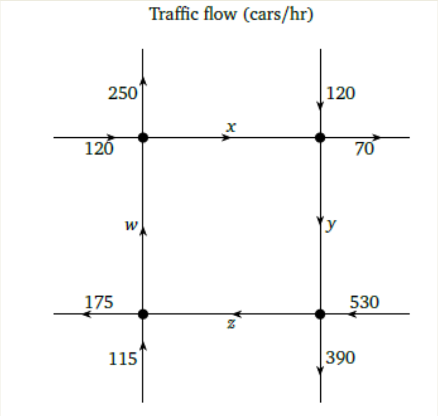
Linear Algebra

- Uses of Linear Algebra in Engineering

Most engineering problems, no matter how complicated, can be reduced to linear algebra:

$$Ax = b \quad \text{or} \quad Ax = \lambda x \quad \text{or} \quad Ax \approx b.$$

- Example (Civil Engineering). The following diagram represents traffic flow around the town square. The streets are all one way, and the numbers and arrows indicate the number of cars per hour flowing along each street, as measured by sensors underneath the roads.



$$\begin{cases} w + 120 = x + 250 \\ x + 120 = y + 70 \\ y + 530 = z + 390 \\ z + 115 = w + 175. \end{cases}$$

There are no sensors underneath some of the streets, so we do not know how much traffic is flowing around the square itself. What are the values of x, y, z,w? Since the number of cars entering each intersection has to equal the number of cars leaving that intersection, we obtain a system of linear equations(above):

Ref:Interactive Linear Algebra /Georgia Institute of Technology

Example

- For example, the three simultaneous equations

$3x - 2y = 8$

$-2x + y - 3z = -20$

$4x + 6y + z = 7$

can be represented as the matrix equation Mx = b

$$\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}$$

and solved by passing arrays corresponding to matrix M and vector b to np.linalg. x = 2, y = −1, z = 5.

If no unique solution exists (for nonsquare or singular matrix, M), a LinAlgError is raised.

```
M = np.array([[3,-2,0],[-2,1,-3],[4,6,1]])
b = np.array([8,-20,7])
np.linalg.solve(M, b)
```

Output: array([2., -1., 5.])

Linear least squares solutions

Suppose that $Ax = b$ does not have a solution. What is the best approximate solution?

For our purposes, the best approximate solution is called the least-squares solution.

We will present two methods for finding least-squares solutions, and we will give several applications to best-fit problems.

Linear least squares solutions

- Linear least squares solutions ("best fit") Where a set of equations, Mx = b, does not have a unique solution, a least squares solution that minimizes the L2 norm, ||b − Mx||2 (sum of squared residuals) may be sought using the np.linalg.lstsq method.

- This is the type of problem described as over-determined (more data points than the two unknown quantities, m and c). Passed M and b, np.linalg.lstsq returns the solution array x, the sum of squared residuals, the rank of M and the singular values of M.

- A typical use of this method is to find the "line of best-fit", y = mx+c, through some data thought to be linearly related as in the following example.

Linear least squares solutions

Numpy | Linear Algebra

- The Linear Algebra module of NumPy offers various methods to apply linear algebra on any numpy array.
  One can find:
  - rank, determinant, trace, etc. of an array.
  - eigen values of matrices
  - matrix and vector products (dot, inner, outer,etc. product), matrix exponentiation
  - solve linear or tensor equations and much more!
- The **numpy.linalg.solve()** function gives the solution of linear equations in the matrix form.
- The solution of this system of equations (the vector x) is returned by the np.linalg

Linear least squares example

Example : The Beer-Lambert Law relates the concentration, c, of a substance in a solution sample to the intensity of light transmitted through the sample, It across a given path length, l, at a given wavelength, λ:

$$I_t = I_0 e^{-\alpha c l},$$

where I0 is the incident light intensity and α is the absorption coefficient at λ.

Given a series of measurements of the fraction of light transmitted, It/I0, α may be determined through a least squares fit to the straight line:

$$y = \ln \frac{I_t}{I_0} = -\alpha c l.$$

Although this line passes through the origin (y = 0 for c = 0), we will fit the more general linear relationship:

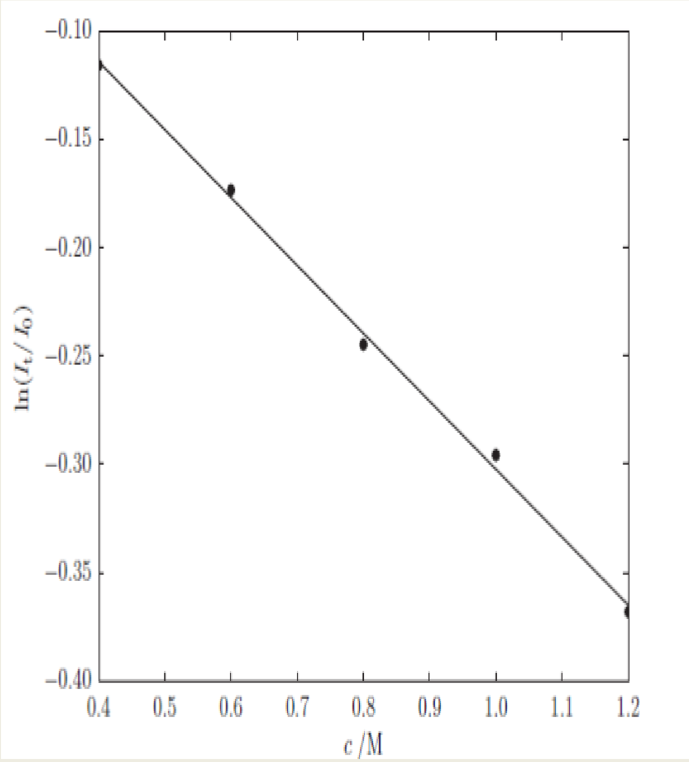
$$y = mc + k$$

where m = −αl, and verify that k is close to zero.

Given a sample with path length l = 0.8 cm, the following data were measured for It/I0 at five different concentrations:

Linear least squares program

```
# Path length, cm
path = 0.8
# The data: concentrations (M) and It/I0
c = np.array([0.4, 0.6, 0.8, 1.0, 1.2])
It_over_I0 = np.array([ 0.891 , 0.841, 0.783, 0.744, 0.692])
n = len(c)
A = np.vstack((c, np.ones(n))).T
T = np.log(It_over_I0)
x, resid, _, _ = np.linalg.lstsq(A, T)
m, k = x
alpha = - m / path
print("alpha = {:.3f} M-1.cm-1".format(alpha))
print("k =", k)
print("rms residual = ", np.sqrt(resid[0]))
pylab.plot(c, T, "o")
pylab.plot(c, m*c + k)
pylab.xlabel("$c\;/\mathrm{M}$")
pylab.ylabel("$\ln(I_\mathrm{t}/I_0)$")
pylab.show()
```



Discrete Fourier transforms

- One-dimensional Fast Fourier Transforms

  numpy.fft is NumPy's Fast Fourier Transform (FFT) library for calculating the discrete Fourier transform (DFT) using the ubiquitous Cooley and Tukey algorithm.
- The definition for the DFT of a function defined on n points, fm,m = 1, 2, · · · , n − 1 used by NumPy is

$$F_k = \sum_{m=0}^{n-1} f_m \exp\left(-\frac{2\pi imk}{n}\right), \quad k = 0, 1, 2, \cdots, n-1$$

- NumPy's basic DFT method, for real and complex functions, is np.fft.fft.
- For example, consider the following waveform in the time domain with some synthetic Gaussian noise added:

$$f(t) = 2\sin(20\pi t) + \sin(100\pi t).$$

Program:

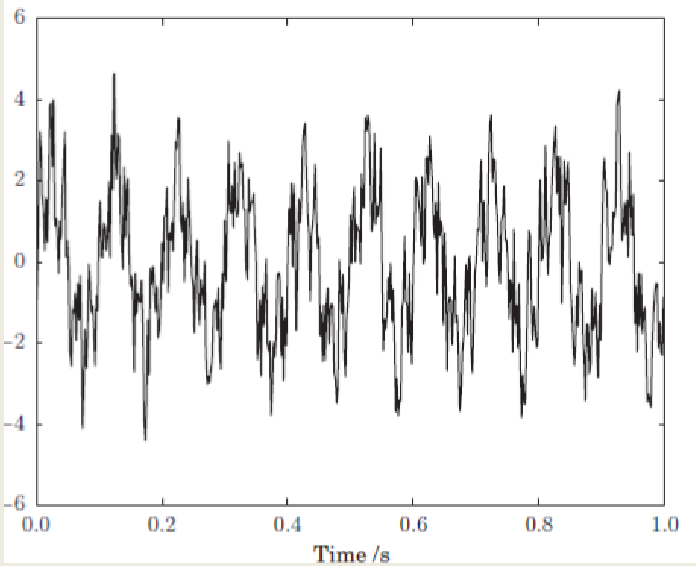
```
A1, A2 = 2, 1
freq1,freq2 = 10, 50
fsamp = 500
t = np.arange(0, 1, 1/fsamp)
n = len(t)
f = A1*np.sin(2*np.pi*freq1*t) + A2*np.sin(2*np.pi*freq2*t)
f += 0.2 * np.random.randn(n)
pylab.plot(t, f)
pylab.xlabel("Time /s")
pylab.show().
```

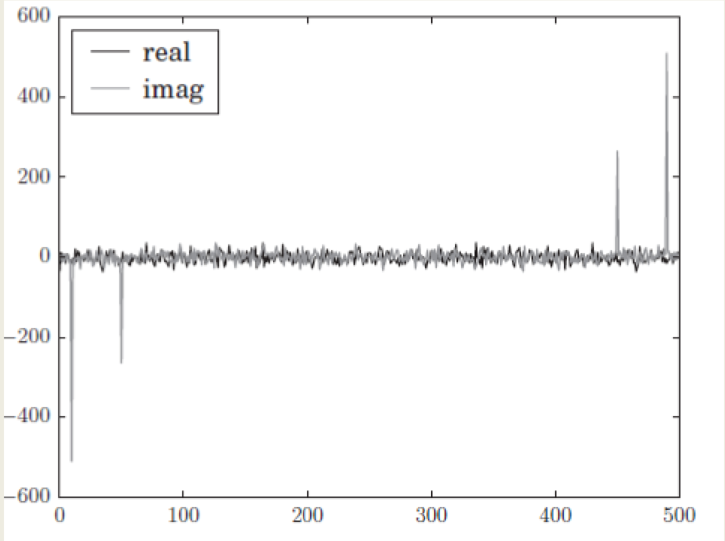
FFT –Example..

- The Fourier transform of this function is complex; its real and imaginary components are plotted here

Program:

```
F = np.fft.fft(f)
plt.plot(F.real, "k", label="real")
plt.plot(F.imag, "gray", label="imag")
plt.legend(loc=2)
plt.show()
```



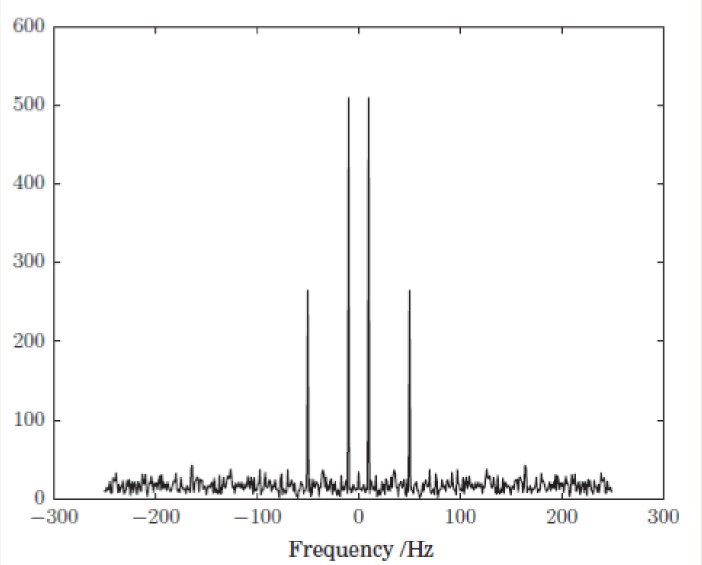


FFT –Example..

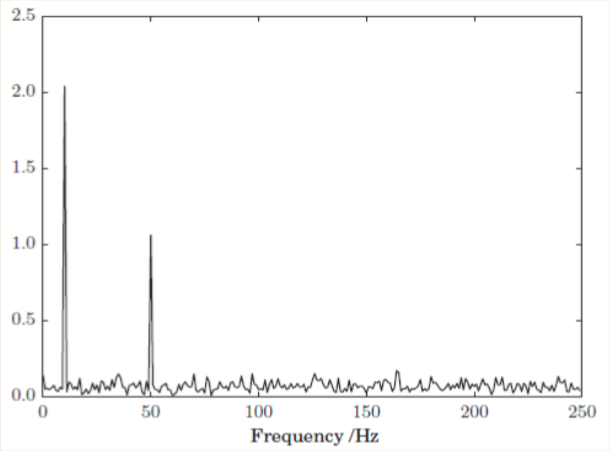
Now look at the shifted amplitude spectrum with the zero-frequency component at the center

Program:

```
freq = np.fft.fftfreq(n, 1/fsamp)
F_shifted = np.fft.fftshift(F)
freq_shifted = np.fft.fftshift(freq)
plt.plot(freq_shifted, np.abs(F_shifted))
plt.xlabel("Frequency /Hz")
plt.show()
```




```
spec = 2/n * np.abs(F[:n/2])
plt.plot(freq[:n/2], spec, "k")
plt.xlabel("Frequency /Hz")
plt.show()
```



Two-dimensional Fast Fourier Transforms

Discrete Fourier transforms and their inverses in two and higher dimensions are possible using the np.fft methods fft2, ifft2, fftn and ifftn.

The two-dimensional DFT is defined as ,and higher dimensions follow similarly

$$F_{jk} = \sum_{p=0}^{m-1} \sum_{q=0}^{n-1} f_{pq} \exp\left[-2\pi i \left(\frac{pj}{m} + \frac{qk}{n}\right)\right],$$

$$j = 0, 1, 2, \cdots, m-1; \; k = 0, 1, 2, \cdots, n-1.$$

Example: The two-dimensional DFT is widely used in image processing.For example, multiplying the DFT of an image by a two-dimensional Gaussian function is a common way to blur an image by decreasing the magnitude of its high-frequency components.

Two-dimensional Fast Fourier Transforms example..

Blurring an image with a Gaussian filter

```
# eg6-fft2-blur.py
# image size, square side length, number of squares
ncols, nrows = 120, 120
sq_size, nsq = 10, 20
# The image array (0=background, 1=square) and boolean array of allowed places
# to add a square so that it doesn't touch another or the image sides
image = np.zeros((nrows, ncols))
sq_locs = np.zeros((nrows, ncols), dtype=bool)
sq_locs[1:-sq_size-1:,1:-sq_size-1] = True
def place_square():
#""" Place a square at random on the image and update sq_locs. """
# valid_locs is an array of the indexes of True entries in sq_locs
    valid_locs = np.transpose(np.nonzero(sq_locs))
# pick one such entry at random, and add the square so its top left
# corner is there; then update sq_locs
    i, j = valid_locs[np.random.randint(len(valid_locs))]
    image[i:i+sq_size, j:j+sq_size] = 1
    imin, jmin = max(0,i-sq_size-1), max(0, j-sq_size-1)
    sq_locs[imin:i+sq_size+1, jmin:j+sq_size+1] = False
# Add the required number of squares to the image
for i in range(nsq):
    place_square()
plt.imshow(image)
plt.show()
```

Take the two-dimensional DFT and center the frequencies

```
# Take the two-dimensional DFT and center the frequencies
ftimage = np.fft.fft2(image)
ftimage = np.fft.fftshift(ftimage)
pylab.imshow(np.abs(ftimage))
pylab.show()
# Build and apply a Gaussian filter.
sigmax, sigmay = 10, 10
cy, cx = nrows/2, ncols/2
x = np.linspace(0, nrows, nrows)
y = np.linspace(0, ncols, ncols)
X, Y = np.meshgrid(x, y)
gmask = np.exp(-(((X-cx)/sigmax)**2 + ((Y-cy)/sigmay)**2))
ftimagep = ftimage * gmask
plt.imshow(np.abs(ftimagep))
plt.show()
# Finally, take the inverse transform and show the blurred image
imagep = np.fft.ifft2(ftimagep)
plt.imshow(np.abs(imagep))
plt.show()
```