Scientific Python

Alexander Richards

Imperial College Sci. & Med. UK
(IC)

July 2019

Numpy

> *NumPy is the fundamental package for scientific computing with Python. It contains among other things:*
>
> - *a powerful N-dimensional array object*
> - *sophisticated (broadcasting) functions*
> - *tools for integrating C/C++ and Fortran code*
> - *useful linear algebra, Fourier transform, and random number capabilities*
>
> *numpy.org*

As the quote suggests, most scientific python packages build upon Numpy.

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

Numpy: Arrays

Numpy's main object is the multi-dimensional array *ndarray*. Unlike python lists all the data must be of the same type.

It has a number of attributes:

- **ndim:** The number of axes (dimensions) of the array
- **shape:** For a matrix of n rows and m columns this is the tuple (n, m)
- **size** The total number of elements in the array
- **dtype** The data type of the elements in the array
- **itemsize** The size in bytes of the elements in the array
- **data** A buffer containing the actual array elements

```
>>> a = np.array(range(6))
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.ndim
1
>>> a.dtype.name
'int32'
>>> a.shape
(6,)
>>> a.size
6
>>> a.itemsize
8
```

Numpy: Array creation

There are several ways to create an array, common ones include:

- From regular python list or tuple. dtype is inferred from values given

```
>>> np.array([1, 2, 3])
array([1, 2, 3])
>>> np.array([4.78, 5.92, 6.34])
array([4.78, 5.92, 6.34])
>>> np.array([1,2,3],dtype=complex)
array([1.+0.j, 2.+0.j, 3.+0.j])
>>> np.array([[1, 2, 3],[4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.array(range(4, 12, 2))
array([4, 6, 8, 10])
```

Common Error

```
>>> a=np.array(1,2,3,4) # WRONG
>>> a=np.array([1,2,3,4]) # RIGHT
```

Numpy: Array creation

There are several ways to create an array, common ones include:

- From regular python list or tuple. dtype is inferred from values given
- *zeros* function, creates array of zeros with given shape.
- *ones* function, creates array of ones with given shape.
- *eye* function, creates identity matrix of given dimension.

```
>>> np.zeros(3) # dtype=np.float64
array([0., 0., 0.]) # by default
>>> np.ones((2, 3), dtype=np.int64)
array([[1, 1, 1],
       [1, 1, 1]])
>>> np.eye(2)
array([[1., 0.],
       [0., 1.]])
>>> np.eye(2, 3)
array([[1., 0., 0.],
       [0., 1., 0.]])
```

Numpy: Array creation

There are several ways to create an array, common ones include:

- From regular python list or tuple. dtype is inferred from values given
- *zeros* function, creates array of zeros with given shape.
- *ones* function, creates array of ones with given shape.
- *eye* function, creates identity matrix of given dimension.
- *arange* function is the numpy analogous to pythons range function
- *linspace* function allows for N linearly spaced values between min and max

```
    #  recall
>>> np.array(range(4, 12, 2))
array([4, 6, 8, 10])
    # can instead use
>>> np.arange(4, 12, 2)
array([4, 6, 8, 10])
>>> np.arange(2.1, 2.5, 0.1)
array([2.1, 2.2, 2.3, 2.4])
>>> np.linspace(5, 200.2, 3)
array([  5.  , 102.6, 200.2])
```

Numpy: Array operations

Unlike Python lists, arithmetic operations on arrays apply *element-wise*

Python Lists

```
>>> a = [1, 2, 3]
>>> a + [4]
[1, 2, 3, 4]
>>> a + a
[1, 2, 3, 1, 2, 3]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- For Python lists, these operations act to change the size of the list rather than modify the value of it's elements

Numpy Arrays

```
>>> b = np.array([1, 2, 3])
>>> b + 1
array([2, 3, 4])
>>> b + [1]
array([2, 3, 4])
>>> b + [1, 2, 3]
array([2, 4, 6])
>>> b + b
array([2, 4, 6])
>>> b * 3
array([3, 6, 9])
>>> b**2
array([1, 4, 9])
```

Numpy: Matrix multiplication / dot / inner product

- The * operator will be the element wise product as seen before.
- To do matrix multiplication we can use the *@* operator
- Or the *dot* method or Numpy function.

1-D Arrays

```
>>> a = np.array(range(3))
>>> b = np.arange(1, 4)
>>> a * b
array([0, 2, 6])
>>> a @ b
8
>>> a.dot(b)
8
>>> np.dot(a, b)
8
```

2-D Arrays

```
>>> e = np.array([[1, 2], [3, 4]]
>>> f = np.array([[5, 6], [7, 8]])
>>> e * f
array([[ 5, 12],
       [21, 32]])
>>> e @ f # Matrix multiplication
array([[19, 22],
       [43, 50]])
>>> e.dot(f)
array([[19, 22],
       [43, 50]])
```

Numpy: Type and Shape Manipulation

Type manipulation methods:

- **astype** Convert array content to given type
- **tolist** Convert array itself to a Python list

Shape manipulation methods:

- **reshape** Returns copy of array conformed to required shape
- **resize** Conforms array to required shape *in-place*
- **transpose** Returns transpose of array. A short-cut exists via the .T property

```
>>> a = np.arange(6)
```

```
>>> a.astype(np.float64)
array([0., 1., 2., 3., 4., 5.])
>>> a.tolist()
[0, 1, 2, 3, 4, 5]
```

```
>>> a.reshape(3, 2)
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.resize(3, 2) # in-place
>>> a.T # same as a.transpose()
array([[0, 2, 4],
       [1, 3, 5]])
```

Numpy: Stacking

```
>>> a = np.array([[1, 2],
                  [3, 4]])
>>> b = np.array([[5, 6],
                  [7, 8]])
>>> c = np.hstack((a, b))
>>> c
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
>>> d = np.vstack((a, b))
>>> d
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

Numpy stacking functions:

hstack	Horizontal stacking
vstack	Vertical stacking
concatenate	Generic join along a given axis

```
>>> np.concatenate((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.concatenate((a, b), 1)
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Numpy: Splitting

Numpy splitting functions:

hsplit	Horizontal splitting
vsplit	Vertical splitting
split	Generic split along a given axis
array_split	As above except can split with unequal size

```
>>> np.split(c, 2)
[array([[1, 2, 5, 6]]),
 array([[3, 4, 7, 8]])]
>>> np.split(c, (2, 3), 1)
[array([[1, 2],
        [3, 4]]),
 array([[5],
        [7]]),
 array([[6],
        [8]])]
```

```
>>> np.hsplit(c, 2)
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]])]
>>> np.vsplit(c, 2)
[array([[1, 2, 5, 6]]),
 array([[3, 4, 7, 8]])]
>>> np.vsplit(d, (2, 3))
[array([[1, 2],
        [3, 4]]),
 array([[5, 6]]),
 array([[7, 8]])]
```

Numpy: Indexing and Slicing

- 1D arrays are indexed/sliced just like Python lists
- Multidimensional arrays can have one index per axis. These indices are given as a tuple and each follows the standard Python indexing rules.
- When fewer indices are provides than number of axes, missing indices are assumed to be complete slices :

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

```
>>> e[1]   # Get the second row
array([3, 4])
>>> e[1][1]   # double indexing
4
>>> e[1, 1]
4
>>> e[:, -1]  # Get last column
array([2, 4])
```

Numpy: Indexing and Slicing

- 1D arrays are indexed/sliced just like Python lists
- Multidimensional arrays can have one index per axis. These indices are given as a tuple and each follows the standard Python indexing rules.
- When fewer indices are provides than number of axes, missing indices are assumed to be complete slices :
- If these missing indices are to be located elsewhere than at the end then we can use ...
- Can assign multiple elements at once using slices

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

```
>>> e[1]   # Get the second row
array([3, 4])
>>> e[1][1]   # double indexing
4
>>> e[1, 1]
4
>>> e[:, -1]  # Get last column
array([2, 4])
```

```
>>> e[..., 1]  # Could be many :
array([2, 4])
>>> e[:, 1] = 10
>>> e
array([[ 1, 10],
       [ 3, 10]])
```

Numpy: Advanced indexing/slicing

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can also be indexed as follows:

By Arrays:

- indexing by array allows us to pick the elements we want
- conform the returned array to a given shape

```
>>> a = np.arange(12)**2
>>> a[np.array([1, 1, 3, 8, 5])]
array([ 1,  1,  9, 64, 25])
>>> a[np.array([[3, 4], [9, 7]])]
array([[ 9, 16], # doesnt work with
       [81, 49]]) # python lists
```

Numpy: Advanced indexing/slicing

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can also be indexed as follows:

By Arrays:

- indexing by array allows us to pick the elements we want
- conform the returned array to a given shape

```
>>> a = np.arange(12)**2
>>> a[np.array([1, 1, 3, 8, 5])]
array([ 1,  1,  9, 64, 25])
>>> a[np.array([[3, 4], [9, 7]])]
array([[ 9, 16], # doesnt work with
       [81, 49]])# python lists
```

By Boolean Arrays:

- tell numpy which elements we want and which we don't
- can be same size as array we are indexing (mask)

```
>>> a.resize(3,4)
>>> b = a > 4
>>> b
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]])
>>> a[b] # or just a[a>4]
array([5, 6, 7, 8, 9, 10, 11])
```

Numpy: Advanced indexing/slicing

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can also be indexed as follows:

By Arrays:

- indexing by array allows us to pick the elements we want
- conform the returned array to a given shape

```
>>> a = np.arange(12)**2
>>> a[np.array([1, 1, 3, 8, 5])]
array([ 1,  1,  9, 64, 25])
>>> a[np.array([[3, 4], [9, 7]])]
array([[ 9, 16],  # doesnt work with
       [81, 49]]) # python lists
```

By Boolean Arrays:

- tell numpy which elements we want and which we don't
- can be same size as array we are indexing (mask)
- can be 1D array representing a given axis

```
b1 = np.array([False,True,True])
b2=np.array([True,False,True,False])
>>> a[b1]
array([[4, 5, 6, 7],
       [8, 9, 10, 11]])
>>> a [:, b2]
array([[0, 2],
       [4, 6],
       [8, 10]])
```

Numpy: Iterating

- 1D arrays are iterated over just like Python lists
- Multidimensional arrays iterate over the first axis
- We can use the flat attribute to iterate over all elements

1-D Arrays

```
>>> f = np.arange(6)
>>> f
array([0, 1, 2, 3, 4, 5])
>>> for element in f:
...     print(element)
0
1
2
3
4
5
```

N-D Arrays

```
>>> g = np.array([[1, 2],
                  [3, 4]])
>>> for row in g:
...     print(row)
[1 2]
[3 4]
>>> for element in g.flat:
...     print(element)
1
2
3
4
```

Numpy: Array unary operations

- Many operations exist as methods of ndarray or as functions of numpy module
- By default they ignore shape and apply to the *entire* array
- Can specify an axis parameter to apply them only over one axis

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

Array methods

```
>>> h.max(), h.min()
(4, 1)
>>> h.sum(), h.mean()
(10, 2.5)
>>> h.sum(axis=0) # column-wise
array([4, 6])
>>> h.mean(axis=1) # row-wise
array([1.5, 3.5])
```

Numpy functions

```
>>> np.max(h), np.min(h)
(4, 1)
>>> np.sum(h), np.mean(h)
(10, 2.5)
>>> np.sum(h, axis=0)
array([4, 6])
>>> np.mean(h, axis=1)
array([1.5, 3.5])
```

Numpy: Universal Functions

- Numpy provides familiar mathematical functions.
- They operate on arrays *element-wise* producing and array as output

```
>>> i = np.linspace(0., 2*np.pi, 5)
>>> i.round(2)
array([0., 1.57, 3.14, 4.71, 6.28])
>>> np.exp(i).round(1)
array([1., 4.8, 23.1, 111.3, 535.5])
>>> np.sin(i).round(2)
array([0., 1., 0., -1., -0.])
>>> np.sqrt(i).round(2)
array([0., 1.25, 1.77, 2.17, 2.51])
```

Numpy: Universal Functions

- Numpy provides familiar mathematical functions.
- They operate on arrays *element-wise* producing and array as output
- How you would do this without Numpy?

```
[math.exp(i) for i in f]
```

```
>>> i = np.linspace(0., 2*np.pi, 5)
>>> i.round(2)
array([0., 1.57, 3.14, 4.71, 6.28])
>>> np.exp(i).round(1)
array([1., 4.8, 23.1, 111.3, 535.5])
>>> np.sin(i).round(2)
array([0., 1., 0., -1., -0.])
>>> np.sqrt(i).round(2)
array([0., 1.25, 1.77, 2.17, 2.51])
```

Numpy: Universal Functions

- Numpy provides familiar mathematical functions.
- They operate on arrays *element-wise* producing and array as output
- How you would do this without Numpy?

```
[math.exp(i) for i in f]
```

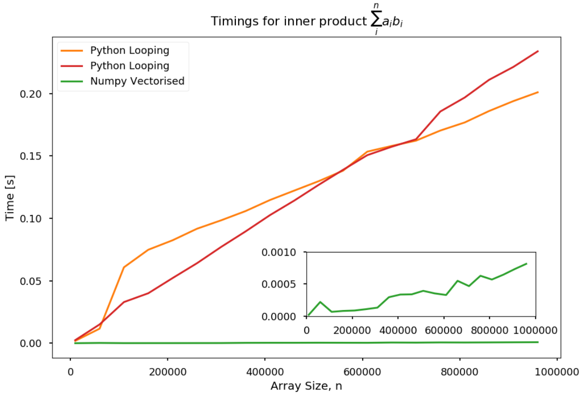
```
>>> i = np.linspace(0., 2*np.pi, 5)
>>> i.round(2)
array([0., 1.57, 3.14, 4.71, 6.28])
>>> np.exp(i).round(1)
array([1., 4.8, 23.1, 111.3, 535.5])
>>> np.sin(i).round(2)
array([0., 1., 0., -1., -0.])
>>> np.sqrt(i).round(2)
array([0., 1.25, 1.77, 2.17, 2.51])
```

- Use of these universal functions (vectorised approach) is more efficient as the looping is done within compiled Numpy libraries rather than in Python.

Numpy: Vectorised efficiency boost

How much faster is the vectorised approach?

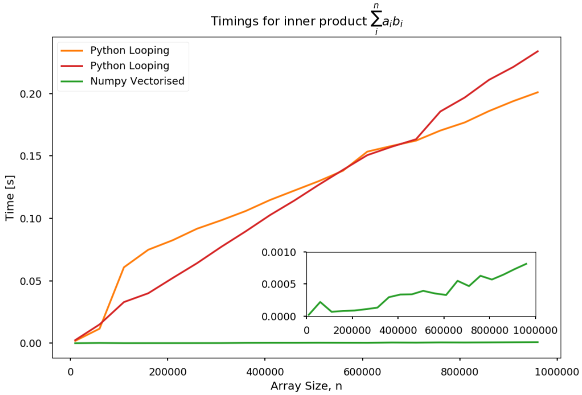
- Create two random number arrays of size n (between 10K and 1M)
- Calculate the dot/scalar/inner product looping through the arrays in python using different algorithms
- Use the Numpy *dot* function to do the same.
- Time the above and plot



Numpy: Vectorised efficiency boost

How much faster is the vectorised approach?

- Create two random number arrays of size n (between 10K and 1M)
- Calculate the dot/scalar/inner product looping through the arrays in python using different algorithms
- Use the Numpy *dot* function to do the same.
- Time the above and plot



Take Home Message

Vectorised approach is orders of magnitude faster and scales much better!

Numpy: Copies and Views demystified

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are really *three* cases:

- No copy at all
  - Simple assignments make no copy of array objects or of their data.
  - Python passes mutable objects as references, so function calls make no copy.

No Copy

```
>>> a = np.arange(12)
>>> b = a
>>> b is a
True
>>> def f(x):
...     return id(x)
>>> id(a) == f(a)
True
>>> b.shape = 3,4
>>> a.shape
(3, 4)
```

Numpy: Copies and Views demystified

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are really *three* cases:

- No copy at all
  - Simple assignments make no copy of array objects or of their data.
  - Python passes mutable objects as references, so function calls make no copy.
- View or shallow copy
  - Different array objects can share the same data. The *view* method creates a new array object that looks at the same data.
  - Slicing an array returns a view of it

View or Shallow Copy

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6
>>> a.shape
(3, 4)
>>> c[0,4] = 1234
>>> a.reshape(2,6)[0,4]
1234
```

Numpy: Copies and Views demystified

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are really *three* cases:

- No copy at all
  - Simple assignments make no copy of array objects or of their data.
  - Python passes mutable objects as references, so function calls make no copy.
- View or shallow copy
  - Different array objects can share the same data. The *view* method creates a new array object that looks at the same data.
  - Slicing an array returns a view of it
- Deep copy
  - The *copy* method makes a complete copy of the array and its data.

Deep Copy

```
>>> d = a.copy()
>>> d is a
False
>>> d.base is a
False
>>> d[0,0] = 9999
>>> a[0,0]
0
```

NOTE: Sometimes copy should be called after slicing if the original array is not required anymore otherwise, since the new array references it, the original will persist in memory even if it goes out of scope.

Numpy: I/O

Numpy has several options for I/O:

Note

Most of these functions take as their target argument either:

- A filename as a string
- A pathlib.Path
- An open Python file-like object

Numpy: I/O

Numpy has several options for I/O:

Numpy Binary Files (NPY/NPZ)

- A simple format for saving numpy arrays to disk with the full information about them.
- The .npy format is the standard binary format in NumPy for persisting a *single* NumPy array on disk. It stores all of the shape and dtype information necessary to reconstruct the array correctly even on another machine with a different architecture. The format is designed to be as simple as possible while achieving its limited goals.
- The .npz format is the standard format for persisting *multiple* NumPy arrays on disk. A .npz file is a zip file containing multiple .npy files, one for each array.

Numpy: I/O

Numpy has several options for I/O:

Numpy Binary Files (NPY/NPZ)

load	Load arrays or pickled objects from .npy, .npz or pickled files
save	Save an array to a binary file in NumPy .npy format
savez	Save several arrays into a single file in uncompressed .npz format.
savez_compressed	As above but compressed

```
>>> a = np.arange(3)
>>> np.save("output.npy", a)
>>> np.load("output.npy")
array([0, 1, 2])
>>> np.savez("out.npz", a, a+3)
>>> dict(np.load("out.npz"))
{'arr_0': array([0, 1, 2]),
 'arr_1': array([3, 4, 5])}
```

```
>>> with open("tmp.npz","wb") as f:
...     np.savez(f, a=a, b=a+3)
>>> dict(np.load("tmp.npz"))
{'a': array([0, 1, 2]),
 'b': array([3, 4, 5])}
```

Numpy: I/O

Numpy has several options for I/O:

Numpy Binary Files (NPY/NPZ)

load	Load arrays or pickled objects from .npy, .npz or pickled files
save	Save an array to a binary file in NumPy .npy format
savez	Save several arrays into a single file in uncompressed .npz format.
savez_compressed	As above but compressed

```
>>> with open("tmp.pkl","wb") as f:
...     pickle.dump([a, a+3], f)
>>> np.load("tmp.pkl",
          allow_pickle=True)
[array([0, 1, 2]),
 array([3, 4, 5])]
```

```
>>> url = "http://"\
        "www.hep.ph.ic.ac.uk/"\
        "~arichard/data.npy"
>>> with urlopen(url) as f:
...     np.load(BytesIO(f.read()))
array([0, 1, 2])
```

Numpy: I/O

Numpy has several options for I/O:

Text Files

loadtxt	Load data from a text file.
savetxt	Save an array to a text file.
genfromtxt	Load data from text file, with missing values handled as specified.
fromregex	Create array from text file, using regular expression parsing.

```
>>> np.loadtxt(StringIO("1 2 3"))
array([1., 2., 3.])
>>> np.loadtxt(StringIO("1 2 3\n4 5 6"), dtype=np.int64)
array([ [1, 2, 3], [4, 5, 6] ])
>>> np.loadtxt(StringIO(" 1,2, 3"), delimiter=',')
array([1., 2., 3.])
>>> np.savetxt("out.txt", np.arange(6))
>>> np.loadtxt("out.txt")
array([0., 1., 2., 3., 4., 5.])
```

Numpy: I/O

Numpy has several options for I/O:

Raw Binary Files

fromfile Construct an array from data in a text or binary file.

ndarray.tofile Write array to a file as text or binary (default).

Note: Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Also no dtype information is stored.

```
>>> a = np.arange(6)
>>> a.tofile("out.bin")
>>> np.fromfile("out.bin", dtype=np.int64)
array([0, 1, 2, 3, 4, 5])
>>> with open("out.bin", "rb") as f:   # Note cant use StringIO with
...     np.fromfile(f, dtype=np.int64) # these as must have a filno()
array([0, 1, 2, 3, 4, 5])              # method
```

NumPy: Random Sampling (numpy.random)

- The NumPy.random module contains many functions for creating random samples.
- For simple random samples there are:

  rand(d0,...,dn) random sample of given shape from a uniform dist. [0,1)

  randn(d0,...,dn) random sample of given shape from **standard** ($\mu = 0, \sigma = 1$) normal dist.

  randint(low, <high>, <size>) random sample of ints with size in range [low, high)

  choice(a, size, replace, p) Generates a random sample from a given 1-D array a
- There are many well known distributions available to sample from:

  uniform(low, high, size) uniform dist. [low, high)

  normal(loc, scale, size) normal dist. with $\mu = loc, \sigma = scale$

  exponential(scale, size) exponential dist.

  poisson(lambda, size) poisson dist.
- This list is not exhaustive, merely the ones you will see most often

Scipy

> *The SciPy library is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines such as routines for numerical integration, interpolation, optimization, linear algebra and statistics.*
>
> *scipy.org/scipylib*

```
>>> import scipy as sp
```

SciPy: Linear Algebra (scipy.linalg)

```
>>> import scipy.linalg as spl
```

Note: scipy.linalg vs numpy.linalg

- scipy.linalg contains all the functions in numpy.linalg. plus some other more advanced ones not contained in numpy.linalg.
- Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for numpy this is optional. Therefore, the scipy version might be faster depending on how numpy was installed.
- Therefore, unless you don't want to add scipy as a dependency to your numpy program, use scipy.linalg instead of numpy.linalg.

SciPy (scipy.linalg): Determinant, Norm and Inverse

Find Determinant

```
>>> a = np.array([[1, 2],
                  [3, 4]])
>>> spl.det(a)
-2.0
>>> b = np.random.rand(3, 3)
>>> b.round(2)
array([[0.19, 0.9 , 0.02],
       [0.65, 0.88, 0.07],
       [0.92, 0.88, 0.79]])
>>> spl.det(b)
-0.2808283293126066
```

Computing Norm/Magnitude

```
>>> a = np.array([3, 4])
>>> spl.norm(a)
5.0
```

Matrix Inversion

```
>>> a = np.array([[1, 3, 5],
                  [2, 5, 1],
                  [2, 3, 8]])
>>> b = spl.inv(a)
>>> b
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
>>> c = a.dot(b).round(3)
>>> c  # double check
array([[ 1., -0.,  0.],
       [ 0.,  1.,  0.],
       [ 0., -0.,  1.]])
>>> (c == np.eye(3)).all()
True
```

SciPy (scipy.linalg): Solving Linear Systems

Solve:

$$\begin{aligned} x + 2y &= 5 \\ 3x + 4y &= 6 \end{aligned} \implies \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

- Could simple solve using the inverse

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

- However, it is better to use the *linalg.solve* command which can be faster and more numerically stable.

```
>>> a = np.array([[1, 2],
                  [3, 4]])
>>> b = np.array([[5],
                  [6]])
>>> spl.inv(a) @ b  # slower
array([[-4.],
       [4.5]])
```

```
>>> spl.solve(a, b)  # faster
array([[-4.],
       [4.5]])
>>> a @ spl.solve(a, b) # check
array([[5.],
       [6.]])
```

SciPy (scipy.linalg): Eigenvalues/Eigenvectors

$$\bar{\boldsymbol{A}}\vec{\boldsymbol{v}} = \lambda\vec{\boldsymbol{v}}$$

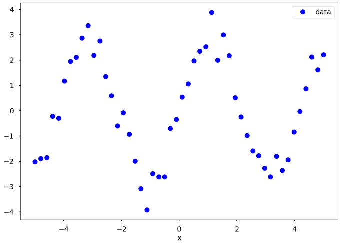
```
>>> a = np.array([[1, 2],
                  [3, 4]])
>>> (l1, l2), v = spl.eig(a)
>>> v1, v2 = np.hsplit(v, 2)
>>> l1, l2    # eigenvalues
((-0.3722813232690143+0j),
 (5.372281323269014+0j))
```

```
>>> v1 # eigenvectors
array([[-0.82456484],
       [ 0.56576746]])
>>> v2
array([[-0.41597356],
       [-0.90937671]])
```

```
>>> # check
>>> ((a@v1).round(3) == (l1*v1).round(3)).all()
True
>>> ((a@v2).round(3) == (l2*v2).round(3)).all()
True
```

SciPy (scipy.optimize): Curve Fitting

```
>>> import scipy.optimize as spo
```



Can use least squares curve fitting to fit to any given function, in this case a sine curve with an unknown amplitude and period.

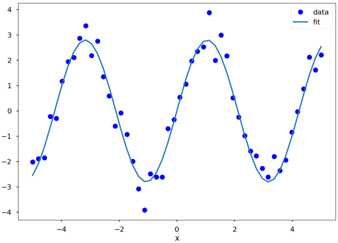
```
>>> def func(x, a, b):
...         return a*np.sin(b*x)
```

```
>>> x = np.linspace(-5, 5, 50)
>>> y = 2.9 * np.sin(1.5 * x_data)
>>> y += np.random.normal(0, 0.5,
                          size=50)

>>> p,cov = spo.curve_fit(func,
                          x, y,
                          p0=[2,2])
>>> p
array([2.80091825, 1.48388696])
>>> cov
array([[ 0.0094366 , -0.00010755],
       [-0.00010755, 0.00012318]])
```

SciPy (scipy.optimize): Curve Fitting

```
>>> import scipy.optimize as spo
```



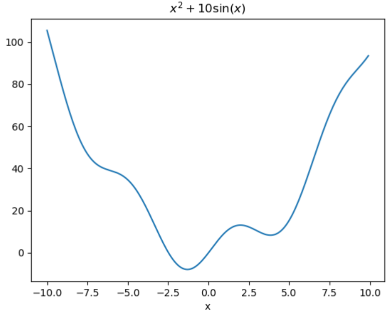
Can use least squares curve fitting to fit to any given function, in this case a sine curve with an unknown amplitude and period.

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

```
>>> x = np.linspace(-5, 5, 50)
>>> y = 2.9 * np.sin(1.5 * x_data)
>>> y += np.random.normal(0, 0.5,
                         size=50)

>>> p,cov = spo.curve_fit(func,
                         x, y,
                         p0=[2,2])
>>> p
array([2.80091825, 1.48388696])
>>> cov
array([[ 0.0094366 , -0.00010755],
       [-0.00010755, 0.00012318]])
```

SciPy (scipy.optimize): Minimisation

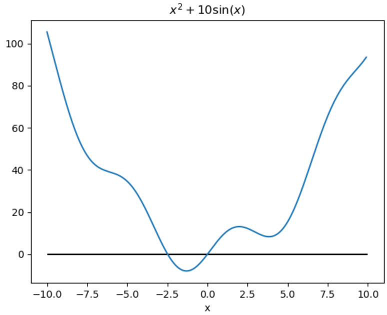



```
>>> def func(x):
...     return x**2 + 10*np.sin(x)
```

One can change the algorithm used in the minimisation as well as setting bounds using additional arguments.

```
>>> spo.minimize(func, x0=0)
      fun: -7.945823375615215
 hess_inv: array([[0.08589237]])
      jac: array([-1.1920929e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
      nit: 5
     njev: 6
   status: 0
  success: True
        x: array([-1.30644012])
>>> m = spo.minimize(func, x0=5)
>>> m.x, m.fun
(array([3.83746713]), 8.31558557947746)
```

SciPy (scipy.optimize): Roots



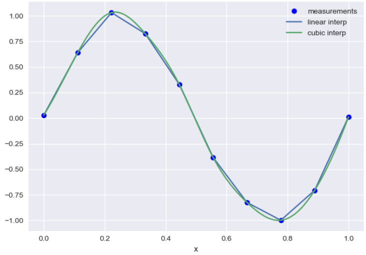

```
>>> def func(x):
...     return x**2 + 10*np.sin(x)
```

```
>>> spo.root(func, x0=0)
    fjac: array([[-1.]])
     fun: array([0.])
 message: 'The solution converged.'
    nfev: 3
     qtf: array([0.])
       r: array([-10.00000001])
  status: 1
 success: True
       x: array([0.])
>>> spo.root(func, x0=-4).x
array([-2.47948183])
```

Note: to find the second root we have to give a different starting point

SciPy (scipy.interpolate): Interpolation

```
>>> import scipy.interpolate as spi
```



- Based on Fortran FITPACK subroutines
- Can evaluate points where no measure exists
- Interpolation points must stay within the range of given data
- interp2d exists for 2D arrays.

```
>>> x = np.linspace(0, 1, 10)
>>> y = np.sin(2*np.pi*x) + np.random.normal(0, 0.05, size=10)
>>> linear_interp = spi.interp1d(x, y)
>>> cubic_interp = spi.interp1d(x, y, kind="cubic")
>>> linear_interp(0.28), cubic_interp(0.28)
(array(0.96105582), array(1.01977364))
```

SciPy (scipy.integrate): Numerical Integration

```
>>> import scipy.integrate as spi
```

- Integration techniques including an ordinary differential equation integrator
- *quad* is the general purpose integrator, uses technique from Fortran QUADPACK library

Compute:

$$\int_0^{\pi/2} \sin(t)dt$$

$$\int_0^1 \int_0^1 \cos(x^2 + y^3)dxdy$$

```
>>> res, err = spi.quad(np.sin, 0, np.pi/2)
>>> res, err # err is an estimate of the error
(0.9999999999999999, 1.1102230246251564e-14)
>>> def func(y, x):
...     return np.cos(x**2 + y**3)
>>> spi.dblquad(func, 0, 1, lambda x:0, lambda x:1)
(0.7701944818578854, 1.0343984979535415e-14)
```

SciPy (scipy.integrate): Numerical Integration

- Can solve a system of n first order ODEs defined in a given function
- Old API *odeint* exists and wraps fast Fortran ODEPACK algorithm
- New API is *solve_ivp*. Note arguments for the function are the other way around. Solutions are row-wise not column-wise

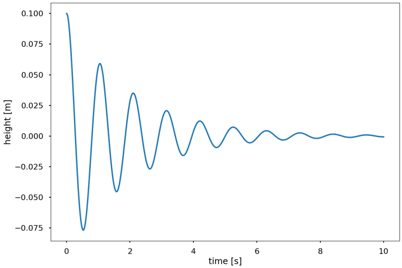
$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\omega_0^2 x - \gamma v$$

```
>>> m, k, b = 0.5, 18, 0.5
>>> def f(x, t):
...     return x[1], -k*x[0]/m - b*x[1]/m
>>> t=np.linspace(0, 10, 1000)
>>> x0 = [0.1, 0]
>>> soln = spi.odeint(f, x0, t)
```

```
>>> soln
array([[0.1,   0.],
       [0.09, -0.02],
       ...,
       [-0.036, 0.206],
       [-0.034, 0.21]])
```



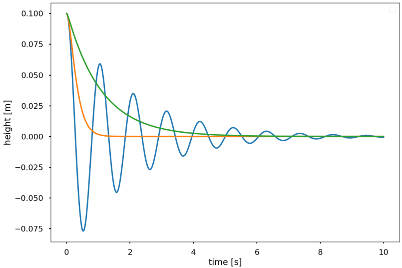
$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\omega_0^2 x - \gamma v$$

```
>>> m, k, b = 0.5, 18, 0.5
>>> def f(x, t):
...         return x[1], -k*x[0]/m - b*x[1]/m
>>> t=np.linspace(0, 10, 1000)
>>> x0 = [0.1, 0]
>>> soln = spi.odeint(f, x0, t)
```

```
>>> soln
array([[0.1,    0.],
       [0.09, -0.02],
        ...,
       [-0.036, 0.206],
       [-0.034, 0.21]])
```



$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

$$\frac{dx}{dt} = v$$

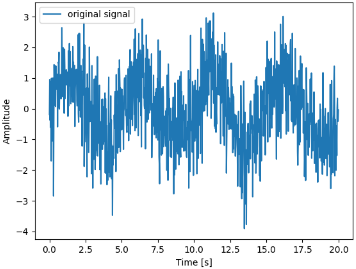
$$\frac{dv}{dt} = -\omega_0^2 x - \gamma v$$

```
>>> m, k, b = 0.5, 18, 0.5
>>> def f(x, t):
...     return x[1], -k*x[0]/m - b*x[1]/m
>>> t=np.linspace(0, 10, 1000)
>>> x0 = [0.1, 0]
>>> soln = spi.odeint(f, x0, t)
```

```
>>> soln
array([[0.1,    0.],
       [0.09, -0.02],
        ...,
       [-0.036, 0.206],
       [-0.034, 0.21]])
```

SciPy (scipy.fftpack): Fast Fourier Transforms

```
>>> import scipy.fftpack as spf
```



This module computes fast Fourier transforms (FFTs) and offers utilities to handle them. The main functions are:

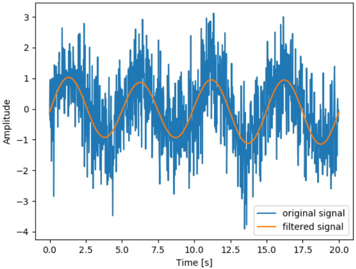
- **fft** Computes the FFT
- **fftfreq** Generates the sampling frequencies
- **ifft** Computes the inverse FFT, from frequency space to signal space

```
>>> t = np.arange(0, 20, time_step)
>>> sig = np.sin(2*np.pi*t/period) + np.random.randn(1000)

>>> sig_fft = spf.fft(sig)
>>> power = np.abs( sig_fft )
>>> freqs = spf.fftfreq(sig.size, d=time_step)
```

SciPy (scipy.fftpack): Fast Fourier Transforms

```
>>> import scipy.fftpack as spf
```



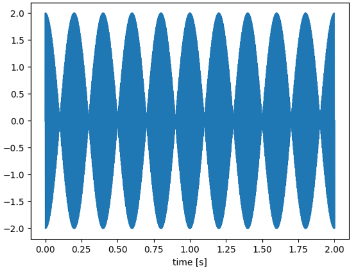
This module computes fast Fourier transforms (FFTs) and offers utilities to handle them. The main functions are:

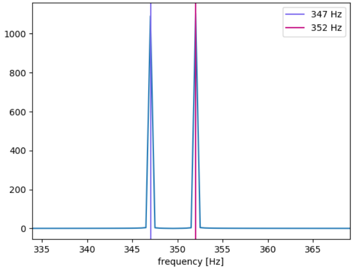
- **fft** Computes the FFT
- **fftfreq** Generates the sampling frequencies
- **ifft** Computes the inverse FFT, from frequency space to signal space

```
>>> max_freq = freqs[freqs>0][power[freqs>0].argmax()]

>>> filtered_sig_fft = sig_fft.copy()
>>> filtered_sig_fft[np.abs(freqs)>max_freq] = 0
>>> filtered_sig = spf.ifft(filtered_sig_fft)
```

SciPy (scipy.fftpack): Fourier Transforms






```
>>> f1 = 347
>>> f2 = 352
>>> t = np.linspace(0, 2., 200000)
>>> sig = np.sin(2*np.pi*f1*t) + np.sin(2*np.pi*f2*t)
>>> freq = spf.fftfreq(sig.size, 1/100000)
>>> sig_fft = np.abs(spf.fft(sig).real)[freq>0]
>>> freq = freq[freq>0]
```

SciPy (scipy.stats): Statistical Functions and Distributions

```
>>> import scipy.stats as sps
```

- This module contains a large number of probability distributions as well as a growing library of statistical functions.
- As with NumPy.random module, can sample from these distributions.
- However these don't just generate random samples, they are much more flexible objects.
- For example, can find the probability from the PDF at any point
- Or use likelihood fit to estimate parameters from generic data

```
>>> sps.norm.pdf(1, loc=2, scale=2)
0.17603266338214976
>>> a = np.random.normal(loc=4, scale=10, size=10000)
>>> sps.norm.fit(a) # Note name is norm not normal as in numpy
(4.063734094442489, 10.045846251468474)
```

- Plus much more!
- There are also functions for calculating summary statistics, correlation, statistical tests and transformations

Plotting in Python: Matplotlib

> *Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.*
>
> *matplotlib.org*

```
>>> import matplotlib.pyplot as plt
```

Matplotlib: Plotting, a tale of two Interfaces

Matplotlib has two interfaces.

- The first is an object-oriented (OO) interface. In this case, we utilize an instance of axes.Axes in order to render visualizations on an instance of figure.Figure.
- The second is based on MATLAB and uses a state-based interface. This is encapsulated in the pyplot module.

	pyplot	OO (Axes)
make plot	plot()	plot()
add grid	grid()	grid()
create plot legend	legend()	legend()
set title	title()	set_title()
set [xy] axis label	[xy]label()	set_[xy]label()
set [xy] axis range	[xy]lim()	set_[xy]lim()
set [xy] axis scale	[xy]scale()	set_[xy]scale

Matplotlib: Plotting, a tale of two Interfaces

Matplotlib has two interfaces.

- The first is an object-oriented (OO) interface. In this case, we utilize an instance of axes.Axes in order to render visualizations on an instance of figure.Figure.
- The second is based on MATLAB and uses a state-based interface. This is encapsulated in the pyplot module.

	pyplot	OO (Axes)
make plot	plot()	plot()
add grid	grid()	grid()
create plot legend	legend()	legend()
set title	title()	set_title()
set [xy] axis label	[xy]label()	set_[xy]label()
set [xy] axis range	[xy]lim()	set_[xy]lim()
set [xy] axis scale	[xy]scale()	set_[xy]scale

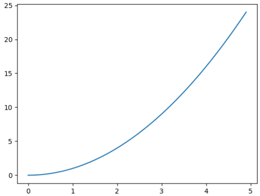
Note

In general, try to use the object-oriented interface over the pyplot interface.

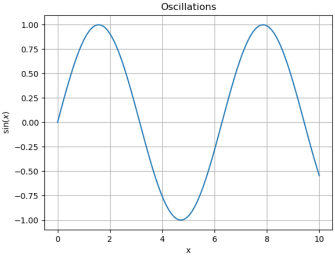
Matplotlib: Plotting Examples

```
>>> plt.plot([xdata], ydata)
```

```
>>> data = np.arange(0, 5, 0.1)
>>> plt.plot(data, data**2)
>>> plt.show()
```

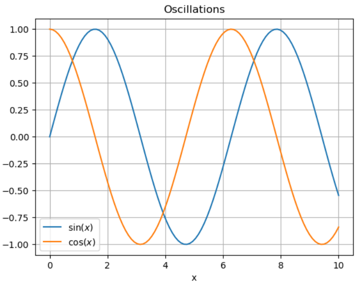



```
>>> xdata = np.linspace(0, 10, 100)
>>> plt.plot(xdata, np.sin(xdata))
>>> plt.title("Oscillations")
>>> plt.xlabel("x")
>>> plt.ylabel("$\sin(x)$")
>>> plt.grid()
>>> plt.show()
```



Matplotlib: Multiple series and legends

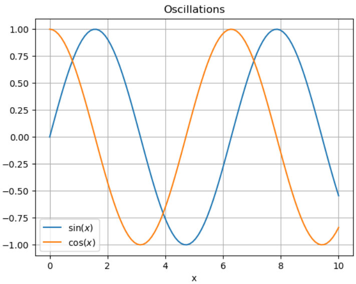
- Plotting multiple series is as easy as calling the plot function multiple times
- plot takes a *label* argument which is automatically picked up when creating the legend
- Legend created with the *legend* method




```
>>> xdata = np.linspace(0, 10, 100)
>>> plt.plot(xdata, np.sin(xdata), label="$\sin(x)$")
>>> plt.plot(xdata, np.cos(xdata), label="$\cos(x)$")
>>> plt.title("Oscillations")
>>> plt.xlabel("x")
>>> plt.grid()
>>> plt.legend()
>>> plt.show()
```

Matplotlib: Multiple series and legends

- Can disable a label by starting it with an underscore
- Instead of giving label argument to plot method, can pass the artists and labels as tuple arguments to legend method.




```
>>> xdata = np.linspace(0, 10, 100)
>>> plt.plot(xdata, np.sin(xdata), label="$\sin(x)$")
>>> plt.plot(xdata, np.cos(xdata), label="$\cos(x)$")
>>> plt.title("Oscillations")
>>> plt.xlabel("x")
>>> plt.grid()
>>> plt.legend()
>>> plt.show()
```

Matplotlib: Styles

- Can check available styles with:

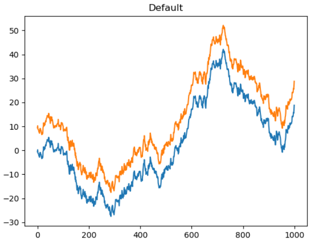
```
>>> print(plt.style.available)
```

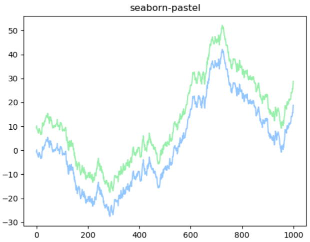
- Can see these at:

https://matplotlib.org/3.1.1/gallery/style_sheets/style_sheets_reference.html

- Which on my machine gives →
- To activate a given style e.g. seaborn-pastel use either *context* or *use* functions:

```
>>> plt.style.use('seaborn-pastel')
```





Styles
Solarize_Light2
_classic_test
bmh
classic
dark_background
fast
fivethirtyeight
ggplot
grayscale
seaborn
seaborn-bright
seaborn-colorblind
seaborn-dark
seaborn-dark-palette
seaborn-darkgrid
seaborn-deep
seaborn-muted
seaborn-notebook
seaborn-paper
seaborn-pastel
seaborn-poster
seaborn-talk
seaborn-ticks
seaborn-white
seaborn-whitegrid
tableau-colorblind10

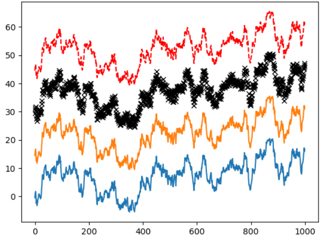
Matplotlib: Markers, colours, linestyle (the quick way)

```
>>> plt.plot([xdata], ydata, [fmt])
```

Where the syntax for fmt is '[marker][line][color]'

- Each is optional
- If not given then the default from the current style is used
- Exception to this is if *line* is given without *marker*. In this case only the line is plotted without markers.
- This is adequate for *most* simple plotting needs.
- Not so good if you require something other than the basic (or style default) colours

```
>>> data=np.random.randn(1000).cumsum()
>>> plt.plot(data)
>>> plt.plot(data + 15)
>>> plt.plot(data + 30, 'xk')
>>> plt.plot(data + 45, '--r')
>>> plt.show()
```



Matplotlib: Format fmt reference

Markers

Char.	Desc.
.	point marker
,	pixel marker
o	circle marker
v	triangle-down marker
^	triangle-up marker
<	triangle-left marker
>	triangle-right marker
1	tri-down marker
2	tri-up marker
3	tri-left marker
4	tri-right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin diamond marker
|	vline marker
_	hline marker

Line Styles

Char.	Desc.
-	solid line style
–	dashed line style
-.	dash-dot line style
:	dotted line style

Colours

Char.	Desc.
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white
CN	CN color spec, (see next slide)

- Note: if color is the only part of the format string given then you can use any color spec (see next slide).

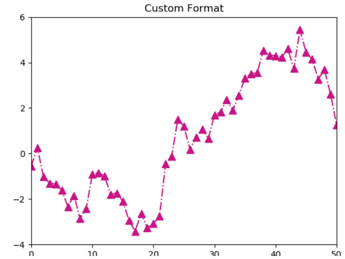
Matplotlib: Markers, colours, linestyle

- More control using the following keywords to plt.plot.
- *marker* recognise all the quick format ones plus some more
- *linestyle*, as the quick format but can also specify draw and dash styles
- *linewidth* controls the width of the line
- *color* recognises many more colours than the quick formatting string, including:
  - An RGB or RGBA (red, green, blue, alpha) tuple of float values in [0, 1]
  - A case-insensitive hex RGB or RGBA *string*
  - A *string* representation of a float value in [0, 1] for grey level
  - One of the quick formatting strings ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']
  - A X11/CSS4 color name (case-insensitive)
  - A name from the xkcd color survey (https://xkcd.com/color/rgb/), prefixed with 'xkcd:'
  - Ane of the Tableau Colors from the 'T10' categorical palette (the default color cycle): ['tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan']
  - a "CN" color spec *string*, i.e. 'C' followed by a number, which is an index into style colour cycle. e.g "C1" is the second colour of the current style

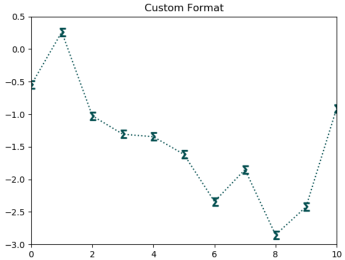
Matplotlib: Markers, colours, linestyle Examples

```
>>> data = np.random.randn(1000).cumsum()
```

```
>>> plt.plot(data, marker='^',
             linestyle='-.',
             color="mediumvioletred",
             markersize=8)
>>> plt.xlim(0, 50), plt.ylim(-4, 6)
>>> plt.title("Custom Format")
>>> plt.show()
```




```
>>> plt.plot(data, marker='$\Sigma$',
             linestyle=':',
             color="xkcd:dark teal",
             markersize=10)
>>> plt.xlim(0,10), plt.ylim(-3,0.5)
>>> plt.title("Custom Format")
>>> plt.show()
```

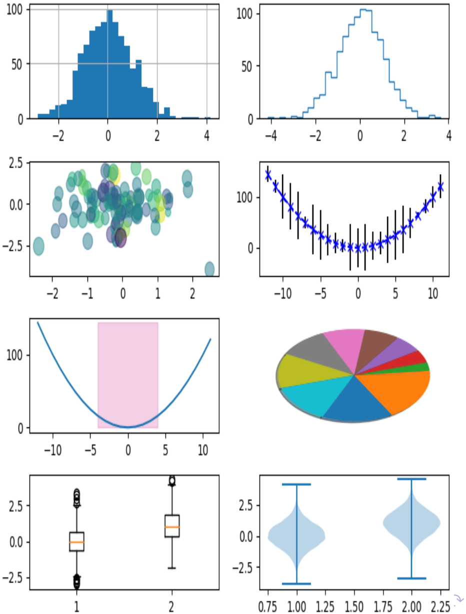


Matplotlib: It's not all about plot()

- While the plot() function is the main line plotting tool
- There are several other functions for creating types of plots that cannot be created using the plot() method.

hist()
scatter()
errorbar()
fill_between()
pie()
boxplot()
violinplot()

- you are encouraged to look at the online API reference to learn how to use them.



Matplotlib: Subplots

- Creating subplots is done via the subplot() / subplots() functions.
- Which one to use depends on which interface you prefer
- With the state-based pyplot interface we switch between the current plots using the following.

```
>>> plt.subplot(ijk)
```

- Where:
  - i The number of rows
  - j The number of columns
  - k The plot to select

- With the OO (Axes) interface we create the separate axes in one go and then manipulate/plot to them individually.

```
>>> fig, axes = plt.subplots(nrows=nrows, ncols=ncolumns)
```

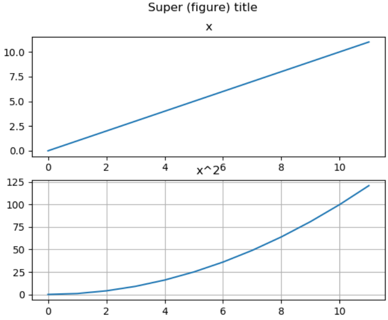
- If you prefer the OO interface then can use the subplots() command even when we only want a single plot (simply call with no args). This is an easy way of getting handles to both the figure and axes object.
- Irregular sized subplots are possible using GridSpec()

Matplotlib: Subplots Example

- The following is an example using both interfaces to create the same plot:

Pyplot Interface

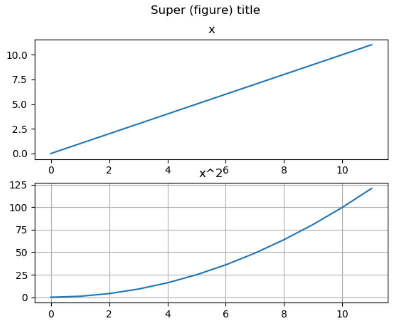
```
>>> plt.subplot(211)
>>> plt.plot(np.arange(12))
>>> plt. title ("x")
>>> plt.subplot(212)
>>> plt.plot(np.arange(12)** 2)
>>> plt. title ("x^2")
>>> plt.grid()
>>> plt. suptitle ("Super   (figure)
title ")
>>> plt.show()
```



- The following is an example using both interfaces to create the same plot:
- Note: the figure (super) title method suptitle() for the OO interface exists as a method of the figure rather than the axes.

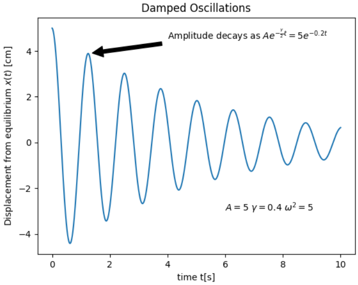
OO (Axes Interface)

```
fig, (ax1, ax2) = plt.subplots(2)
>>> ax1.plot(np.arange(12))
>>> ax2.plot(np.arange(12)** 2)
>>> ax1. set_title ("x")
>>> ax2. set_title ("x^2")
>>> ax2.grid ()
>>> fig. suptitle ("Super (figure)
title ")
>>> plt.show()
```



Matplotlib: Working With Text

- We have already seen the *title*, *xlabel* and *ylabel* methods for adding text in the corresponding places within a plot
- Can use the *text* method to add text in an arbitrary position.




```
>>> plt.text(x, y, text)
```

- Can *annotate* parts of the plot.

```
>>> plt.annotate(text, xy=(x, y), xytext=(x, y))
```

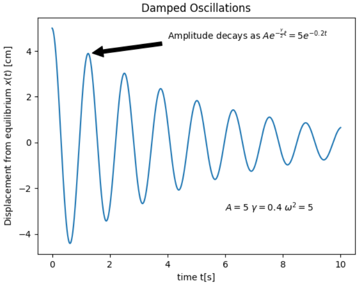
- matplotlib accepts TeX equation expressions in any text expression, simply enclose the TeX expression with $ sign.

```
>>> plt.title("$\sigma_i=15$")
```

Matplotlib: Working With Text

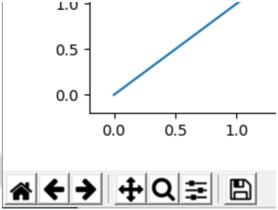
- We have already seen the *title*, *xlabel* and *ylabel* methods for adding text in the corresponding places within a plot
- Can use the *text* method to add text in an arbitrary position.

```
>>> plt.text(x, y, text)
```




```
>>> xdata = np.linspace(0, 10, 10000)
>>> ydata = lambda xdata: A*np.exp(-gamma*xdata/2) * np.cos(w2*xdata)
>>> plt.plot(xdata, ydata(xdata))
>>> plt.text(6, -3, f'$A={A}\ \gamma={gamma}\ \omega^2={w2}$')
>>> plt.annotate(rf"Amplitude decays as $A e^{{- \frac {{\gamma}}{{2}} t }
                 xy=(2*np.pi/w2, ydata(2*np.pi/w2)),
                 xytext=(4, 4.5),
                 arrowprops={'facecolor':'black', 'shrink':0.05})
```

Matplotlib: Saving plots



- If viewing the plots interactively, saving is as simple as using the save button on the canvas toolbar (see diagram).
- Can save programmatically using:

```
>>> fig.savefig('sales.png')
```

- To get available output formats on your system

```
>>> print(fig.canvas.get_supported_filetypes())
```

- Some useful options include:

transparent=True makes the background of the saved figure transparent if the format supports it.

dpi controls the resolution (dots per square inch) of the output.

bbox_inches="tight" fits the bounds of the figure to our plot.

Python Data Analysis Library: Pandas

> *pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.*
>
> *pandas.pydata.org*

```
>>> import pandas as pd
```

- Built on top of numpy it provides two primary types of data structure:
  - **Series** A 1-dimensional labeled (indexed) array capable of holding any data type
  - **DataFrame** a 2-dimensional labeled (indexed) data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object.

Pandas: Series

Series are created using the syntax:

```
>>> s = pd.Series(data, [index=index])
```

Where *data* can be:

- An iterable like Python list/tuple or Numpy array. In this case if *index* is given it must be the same length as data. If not given then range(0, len(data), 1) or $[0, len(data))$ is used.
- A scalar value e.g. 5. or 'a' In this case index *must* be given and the scalar value is repeated len(index) times.
- A dict. In this case if no *index* is given, the indices are taken from the dict's keys. Note the ordering will depend on versions of Python and Pandas. If *index* is given then the values in the dict for the corresponding keys are pulled out. Note if the dict doesn't contain a given key then it will get the value NaN
- Another Series. Copy constrution allowing you to specify indecies. Specifying indices which don't exist in the original will result in an entry of NaN

Pandas: Series Examples

Check that your understand these examples:

```
>>> s1 = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> s2 = pd.Series(5, index=[0, 1, 2, 3, 4])
>>> data = {'one': 1, 'two': 2, "three": 3}
>>> s3 = pd.Series(data, index=('one', 'three', 'four'))
```

```
>>> s1
a    0.057661
b    1.896328
c   -0.175784
d   -1.549775
```

```
>>> s2
0    5
1    5
2    5
3    5
4    5
```

```
>>> s3
one      1.0
three    3.0
four     NaN
```

Pandas: Series are ndarray-like

```
>>> s = pd.Series(np.random.randn(5), index=['a','b','c','d','e'])
```

- Series acts very similarly to a ndarray, and are a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
>>> s[:3]
a   -0.788871
b    0.993635
c   -2.123184
```

- If you need an *actual* ndarray

```
>>> s.to_numpy()
array([-0.788871,  0.993635, -2.123184,  0.419299, 0.292790])
```

Pandas: Not exactly ndarray-like

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
>>> s[1:] + s[:-1]
a         NaN
b    1.987270
c   -4.246368
d    0.838598
e         NaN
```

- We will cover slicing Series later.
- Suffice to say that we are adding two series with 4 elements each. The first has indices b - e while the second has indices a - d

Pandas: Series are dict like

- A Series is like a fixed-size dict in that you can get and set values by index label

```
>>> s['a']
-0.7888708311903035
>>> s['e'] = 10.1
>>> 'e' in s
True
>>> 'f' in s
False
```

- Invalid index raises a *KeyError*

```
>>> s['f']
KeyError: 'f'
```

- The *get* method works just as with dicts.

```
>>> s.get('f')
>>> s.get('f', np.nan)
nan
```

Pandas: DataFrames

DataFrames are created using the syntax:

```
>>> s = pd.DataFrame(data, [index=index], [columns=columns])
```

Where *data* can be:

- A 2D Numpy array.
- A dict of iterables like Python lists/tuples or Numpy arrays. All iterables *must* be of the same length. If *index* is passed it *must also* be of the same length. If not then range(0, len(n), 1) is used where n is the length of the list/tuple/array. Keys are used as column labels and if *columns* is given only these keys are used or column full of NaN if entry in *columns* isn't a key.
- A dict of Series or dicts. If *index* not given then it will be the union of the indexes of the various Series (dict keys). If it is given then requested indexes are looked up in each Series/dict and the corresponding value used or NaN if index doesn't exist.

Pandas: DataFrames

DataFrames are created using the syntax:

```
>>> s = pd.DataFrame(data, [index=index], [columns=columns])
```

Where *data* can be:

- A Series. The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).
- Another DataFrame. A copy construction allowing you to specify indcies and/or columns. Specifying either indices or columns which don't exist in the original will result in either a column or row of NaNs

Pandas: DataFrame Examples

Check that your understand these examples:

```
>>> d1 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]),
                      columns=('a', 'b', 'c'))
>>> d2 = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
```

```
>>> d1
   a  b  c
0  1  2  3
1  4  5  6
```

```
>>> d2
   a  b
0  1  4
1  2  5
2  3  6
```

Pandas: Viewing data

We can get views of the DataFrame using the following:

head	Top n rows
tail	Bottom n rows
index	indexes
columns	columns
describe	show a short statistic summary of data
T	(*attribute*) Transpose the data rows <-> columns
sort_index	sort by an axis
sort_values	sort by value

```
>>> df.describe()
              A           B
count  6.000000   6.000000
mean   0.073711  -0.431125
std    0.843157   0.922818
min   -0.861849  -2.104569
25%   -0.611510  -0.600794
50%    0.022070  -0.228039
75%    0.658444   0.041933
max    1.212112   0.567020
```

Pandas: Descriptive Statistics

- As with NumPy arrays, there exists a large number of methods for computing descriptive statistics and other related operations on Series and DataFrames
- Most of these are aggregations (hence producing a lower-dimensional result) like sum(), mean(), and quantile()
- Some of them, like cumsum() and cumprod(), produce an object of the same size
- Generally speaking, these methods take an axis argument, just like ndarray.sum, std, ..., but the axis can be specified by name or integer

Series	no axis argument needed
DataFrame	"index" (axis=0, default), "columns" (axis=1)

- All such methods have a skipna option signaling whether to exclude missing data (True by default):

Pandas: Missing Data

Pandas primarily uses the value np.nan to represent missing data.

- We can locate (get masks for) the missing data using the isna() or notna() functions/methods
- They will *not* by default be included in calculations like sum, mean, cumsum etc.
- Rows (or columns) containing missing data can be dropped with the dropna() method
- Missing data can be filled in using the fillna() method
- Pandas can use interpolation to fill missing data using the interpolate() method

```
>>> df
     0    1    2
0  1.0  2.0  3.0
1  4.0  NaN  6.0
2  NaN  8.0  NaN
>>> df.dropna()
     0    1    2
0  1.0  2.0  3.0
```

```
>>> df.isna()
       0      1      2
0  False  False  False
1  False   True  False
2   True  False   True
```

```
>>> df.fillna(value=10)
      0     1     2
0   1.0   2.0   3.0
1   4.0  10.0   6.0
2  10.0   8.0  10.0
```

Pandas: Interoperability with NumPy functions

- Elementwise NumPy Universal functions (ufuncs) like log, exp, sqrt, etc. and various other NumPy functions can be used with no issues on Series and DataFrame
- The caveat is that the data within must be numeric

```
>>> d = pd.DataFrame(np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8 ]]),
                    columns=['A','B','C'], dtype=np.float64)
>>> np.exp(d)
            A            B            C
0    1.000000     2.718282     7.389056
1   20.085537    54.598150   148.413159
2  403.428793  1096.633158  2980.957987
>>> np.sqrt(d)
          A         B         C
0  0.000000  1.000000  1.414214
1  1.732051  2.000000  2.236068
2  2.449490  2.645751  2.828427
```

Pandas: Group by

- Familiar to those who have use SQL
- Refers to the process involving one (or more) of the following steps:
  - **Splitting** the data into groups based on some criteria.
  - **Applying** a function to each group independently.
  - **Combining** the results into a data structure.
- During the apply step we may like to do one of the following:
  - **Aggregation** compute a summary statistic (or statistics) for each group
  - **Transformation** perform some group-specific computations and return a like-indexed object
  - **Filtration** discard some groups, according to a group-wise computation that evaluates True or False

Consider the following DataFrame.

```
>>>df=pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                    'B': ['one', 'one', 'two', 'three',
                          'two', 'two', 'one', 'three'],
                    'C': np.random.randn(8), 'D': np.random.randn(8)})
```

Pandas: Group by

```
>>> df
     A      B         C         D
0  foo    one -1.202872 -0.055224
1  bar    one -1.814470  2.395985
2  foo    two  1.018601  1.552825
3  bar  three -0.595447  0.166599
4  foo    two  1.395433  0.047609
5  bar    two -0.392670 -0.136473
6  foo    one  0.007207 -0.561757
7  foo  three  1.928123 -1.623033
```

Pandas: Group by

```
>>> df
     A      B         C         D
0  foo    one -1.202872 -0.055224
1  bar    one -1.814470  2.395985
2  foo    two  1.018601  1.552825
3  bar  three -0.595447  0.166599
4  foo    two  1.395433  0.047609
5  bar    two -0.392670 -0.136473
6  foo    one  0.007207 -0.561757
7  foo  three  1.928123 -1.623033
```

- Grouping and then applying the sum() function to the resulting groups

```
>>> df.groupby('A').sum()
            C        D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958
```

Pandas: Group by

```
>>> df
     A      B         C         D
0  foo    one -1.202872 -0.055224
1  bar    one -1.814470  2.395985
2  foo    two  1.018601  1.552825
3  bar  three -0.595447  0.166599
4  foo    two  1.395433  0.047609
5  bar    two -0.392670 -0.136473
6  foo    one  0.007207 -0.561757
7  foo  three  1.928123 -1.623033
```

- Grouping and then applying the sum() function to the resulting groups

```
>>> df.groupby('A').sum()
            C        D
A
bar -2.802588  2.42611
foo  3.146492 -0.63958
>>> df.groupby(['A', 'B']).sum()
                  C         D
A   B
bar one   -1.814470  2.395985
    three -0.595447  0.166599
    two   -0.392670 -0.136473
foo one   -1.195665 -0.616981
    three  1.928123 -1.623033
    two    2.414034  1.600434
```

- Grouping by multiple columns forms a hierarchical index, and again we can apply the sum function.

Pandas: Indexing/slicing

Object Type	Syntax
Series	s[indexer]
DataFrame	df[indexer]

Where indexers are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index). For DataFrames this selects columns.
- A list or array of labels ['a', 'b', 'c']. For DataFrames this selects columns. Note doesn't work with tuples
- A slice object e.g. 0:3:1 using integer positions not row labels. Note for Dataframes this slices on rows not columns.
- A boolean array. For DataFrames this slices on rows not columns. If a 2D array is used, returns a DataFrame with NaN for elements that are False.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

Pandas: Indexing/slicing

Object Type	Syntax
Series	s[indexer]
DataFrame	df[indexer]

Where indexers are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index). For DataFrames this selects columns.

Note

There is a convenience shortcut for this via an attribute with the same name as the index or column. This only works however if the name of the index or column is a valid Python identifier, e.g. string, no spaces etc. e.g:

```
>>> s.a
>>> d.a
```

Pandas: Indexing/slicing Examples

```
>>> a=pd.Series(range(4), index=range(1, 5))
>>> b=pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
```

Consider the following objects

```
>>> a
1    0
2    1
3    2
4    3
```

```
>>> b
   0  1  2
0  1  2  3
1  4  5  6
2  7  8  9
```

Pandas: Indexing/slicing Examples

```
>>> a=pd.Series(range(4), index=range(1, 5))
>>> b=pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
```

```
>>> a[3]
2
>>> a[[2,4]]
2    1
4    3
>>> a[1::2]
2    1
4    3
>>> a[a>2]
4    3
>>> a[lambda x: x>1]
3    2
4    3
```

```
>>> b[1]
0    2
1    5
2    8
>>> b[[0,2]]
   0  2
0  1  3
1  4  6
2  7  9
>>> b[::2]
   0  1  2
0  1  2  3
2  7  8  9
```

```
#b[[True, False, True]]
>>> b[np.array([True,
                False,
                True])]
   0  1  2
0  1  2  3
2  7  8  9
>>> b[b>4]
     0    1    2
0  NaN  NaN  NaN
1  NaN  5.0  6.0
2  7.0  8.0  9.0
```

Pandas: Indexing/slicing using loc/iloc

Object Type	Syntax
Series	s.loc[indexer]
DataFrame	df.loc[row-indexer, column-indexer]

Where indexers are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index unlike iloc.).
- A list or array of labels ['a', 'b', 'c']. Note, doesn't work with tuples.

Gotcha!

With DataFrames you *can* use a tuple rather than a list but be aware!

```
>>> b.loc[[1,2]]
   0  1  2
1  4  5  6
2  7  8  9
```

```
>>> b.loc[(1,2)]# treats as row, columns args
6
```

```
>>> b.loc[1,2]
6
>>> b.loc[(1,2), 1]
1     5
2     8
```

```
>>> b.loc[[1,2], 1]
1     5
2     8
>>> b.loc[(1,2 ),:]
   0  1  2
1  4  5  6
2  7  8  9
```

Pandas: Indexing/slicing using loc/iloc

Object Type	Syntax
Series	s.loc[indexer]
DataFrame	df.loc[row-indexer, column-indexer]

Where indexers are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index unlike iloc.).
- A list or array of labels ['a', 'b', 'c']. Note, doesn't work with tuples.
- A slice object with labels 'a':'f' (Note that contrary to usual python slices, both the start and the stop are included, when present in the index! Not so for iloc which uses integer positions and doesn't include the stop.
- A boolean array. Must be 1D as each argument is for a single axis
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

Pandas: loc/iloc vs df[]

Recall that using square brackets with a DataFrame we can only select either rows or columns, depending on the indexer. This is not the case with loc/iloc which are more flexible than simply using the square brackets. consider the following:

```
>>> df = pd.DataFrame(np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]]),
                      columns=["A", "B", "C"])}
```

- The behave the same for the following:
  - df['A'] is the same as df.loc[:, 'A'] and df.A
  - df[['A', 'B', 'C']] is the same as df.loc[:, ['A', 'B', 'C']]
  - df[1:3] is the same as df.iloc[1:3]
- The following however are not possible with square brackets:
  - select a single row with df.loc[row_label]
  - select a list of rows with df.loc[[row_label1, row_label2]]
  - slice columns with df.loc[:, 'A':'C']
- With .loc, you are guaranteed to modify the original DataFrame
  - df[1:3]['A'] = 5 will raise a warning for trying to set a view/copy rather than the original object

Pandas: Iteration

- Basic iteration (`for i in obj`) produces:

Series	values (array/list like)
DataFrame	column labels (dict like)

- All Pandas objects have the dict-like `items()` method to iterate over the (key, value) pairs.
- DataFrames also have:

iterrows	Iterate over rows of a DataFrame as (index, Series) pairs. This converts rows to Series objects, which can change the dtypes and has some performance implications.
itertuples	Iterate over rows of a DataFrame as namedtuples of the values. This is a lot faster than iterrows(), and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning

Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided by looking for a vectorized solution or a compiled solution using cython etc

Pandas: I/O functions/methods

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Pandas: IO Examples

Where a file needs to be specified, the IO functions usually accept the following:

- filename or URL as string
- a pathlib.Path object (Python 3)
- a file-like object with a read() method

```
>>> pd.read_csv(StringIO("A,B,C\n1,2,3\n4,5,6\n7,8,9"))
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
>>> url = "http://www.hep.ph.ic.ac.uk/~arichard/csv_data.csv"
>>> pd.read_csv(url, header=None)
   0  1  2  3
0  1  2  3  4
1  5  6  7  8
```

Pandas: IO Examples

Where a file needs to be specified, the IO functions usually accept the following:

- filename or URL as string
- a pathlib.Path object (Python 3)
- a file-like object with a read() method

```
>>> p = Path('/home/hep/arichard/public_html/csv_data.csv')
>>> pd.read_csv(p, header=None)
   0  1  2  3
0  1  2  3  4
1  5  6  7  8
>>> pd.DataFrame(np.random.rand(2,2)).to_pickle("data.pkl")
>>> pd.read_pickle("data.pkl")
          0         1
0  0.327380  0.101108
1  0.783487  0.314491
```

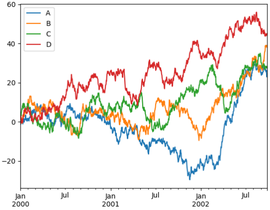
Pandas: Plotting

- Can plot directly from pandas DataFrame/Series objects
- Can select several different types of plot using the *kind* argument

bar/barh	For bar plots
hist	For histogram
box	For boxplot
kde/density	For density plots
area	For area plots
scatter	For scatter plots
hexbin	For hexagonal bin plots
pie	For pie plots

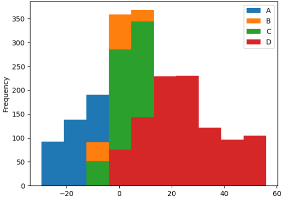
- You can also create these other plots using the methods DataFrame.plot.<kind> instead of providing the kind keyword argument
- In addition to these kind s, there are the DataFrame.hist(), and DataFrame.boxplot() methods, which use a separate interface.
- Finally, there are several plotting functions in pandas.plotting that take a Series or DataFrame as an argument. These include: Scatter Matrix, Andrews Curves, Parallel Coordinates, Lag Plot, Autocorrelation Plot, Bootstrap Plot, RadViz

Pandas: Plotting



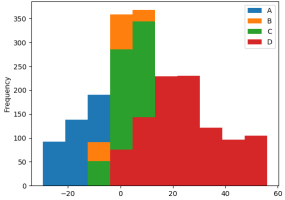

```
>>> i = pd.date_range('1/1/2000', periods=1000)
>>> df = pd.DataFrame(np.random.randn(1000, 4),
                      index=i,
                      columns=['A', 'B', 'C', 'D'])
>>> df = df.cumsum()
>>> df.plot()
>>> plt.show()
```

Pandas: Plotting

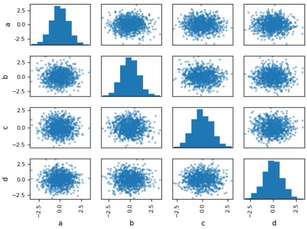



```
>>> i  = pd.date_range(' 1/1/2000 ', periods=1000)
>>> df  = pd.DataFrame(np.random.randn(1000, 4),
                    index=i ,
                    columns=['A',  'B',  'C',  'D'])
>>> df  =  df.cumsum()
>>> df. plot (kind=" hist " )
>>> plt .show()
```

Pandas: Plotting




```
>>> i = pd.date_range('1/1/2000', periods=1000)
>>> df = pd.DataFrame(np.random.randn(1000, 4),
                      index=i,
                      columns=['A', 'B', 'C', 'D'])
>>> df = df.cumsum()
>>> df.plot(kind="hist")
>>> plt.show()
```




```
>>> from pandas.plotting import scatter_matrix
>>> df = pd.DataFrame(np.random.randn(1000, 4),
                       columns=['a', 'b', 'c', 'd'])
>>> scatter_matrix(df)
>>> plt.show()
```

The End....Or just the beginning

- Now do Worksheet1: http://www.hep.ph.ic.ac.uk/ arichard/pgtasks/Worksheet1.ipynb