conda install -c conda-forge jupyter_contrib_nbextensions

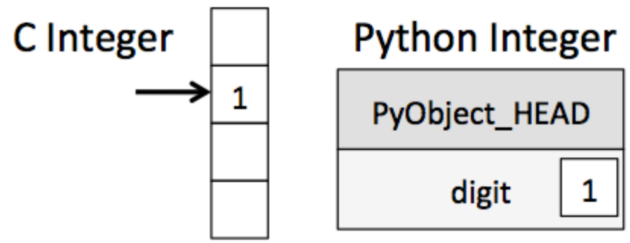
# Python - Strutture Dati speciali

**http://bit.ly/pystruspec**

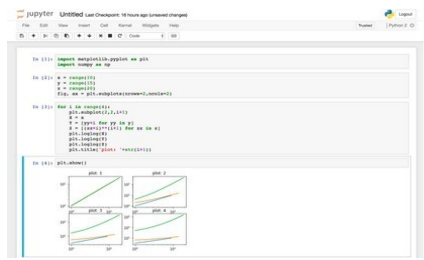
*Rev 4.25*

# Numpy

- Python ... Array... Uhmm what's exactly you mean ?
- Numpy → Python extension add support for
  - large, multi-dimensional arrays and matrices,
  - large collection of high-level mathematical functions to operate on these arrays.
- Object collection related to Scientific Calculation the son of Numeric
- Dynamic data typing vs Static ( C or Java)
  - Any kind of type inside a variable
  - A Python integer is no a simple integer is a pointer to a complex structure
  - A python List is more than a simple list

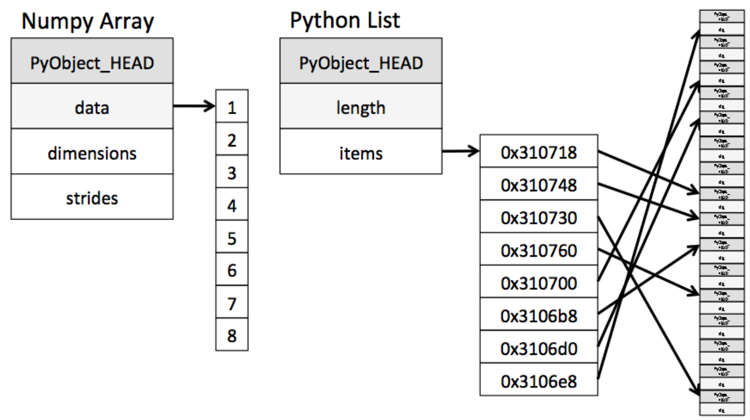



vedi jupyter notebook
Numpy1



# Numpy Data types

- Complex object list of etherogeneous objects → lot of flexibility lot of complexity
- But if all objects are of the same (fixed) type ?
  - NumPy – Style Array efficient storage and operations
  - Python Array efficient storage




```
>>> import numpy as np
>>> np.array([1,2,3,4,5,3])
array([1, 2, 3, 4, 5, 3])
>>> a = np.array([3.14,3,4,5])
>>> a
array([ 3.14,  3.  ,  4.  ,  5.  ])
>>> np.array([1, 2, 3, 4], dtype='float32')
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

```
>>> import array
>>> import array
>>> L = list(range(10))
>>> A = array.array('i', L)
>>> A
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

\# nested lists result in **multi-dimensional** arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])

# Numpy Arrays

- **Attributes**
  - ndim,shape,size,dtype
- **Creating** → x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
  x3 = np.array([1,2,3],[4,5,6],[6,7,8])
- **Accessing** → x2[0, 0] = 12
  - Slicing → x[start:stop:step]
  - Row vs Columns access x2[:, 0] # first column of x2 --- x2[0,:] # first row of x2
  - Sub array view Vs copy x2_sub = x2[:2, :2] vs x2_sub_copy = x2[:2, :2]**.copy()**
- **Reshaping**
  - x = np.array([1, 2, 3])
    x.reshape((1,3) offline
    x.resize(m,n) inline
    array([[1, 2, 3]]) # bidimensionale una riga x[np.newaxis, :]
    x.reshape((3, 1)) # bidimensionale una colonna x[:, np.newaxis]
- **Concatenate** np.concatenate((x,y)) np.concatenate([x,y,axis=1]) sul secondo asse (0 start)
  - .vstack .hstack .dstack (3° asse)
- **Split** grid = np.arange(16).reshape((4, 4))
  - left, right = np.hsplit(grid, [2]) .vsplit, .dsplit

```
x = np.array([[1, 2, 3],[4, 5, 6]])
y= np.array([[4, 5, 6],[1, 2, 3]])
np.concatenate((x,y),axis=1)
```

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

```
np.concatenate((x,y),axis=1)
```

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

# Universal functions

```
import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
output = np.empty(len(values))
for i in range(len(values)):
output[i] = 1.0 / values[i]
return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)
big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

- Slow → PyPy Cython
  - Show on many small op are being repeated
  - Ufuncs → %timeit (1.0 / big_array)
  - Generalized op
  - Executed as vectorized op
  - Applied to all elements of array
  - also (multi)array to (multi)array
  - np.arange(5) / np.arange(1, 6)
  - + - * / // ** %(modulus) -(negative) abs standard order of operations
  - equiv np.add, np.subtract, .negative, .multiply, .divide .power .mod .abstrigonometric
  - also on complex values abs → magnitude
  - trigonometric np.pi np,sin … and inverse trigonometric
  - a = np.linspace(0,np.pi,3) np.sin(a)
  - exponential and log .exp .exp2 .power .log .log2 .log10
  - also hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders ……..
  - from scipy import special (other obscure functions) special.gamma special.erf

# Numpy universal functions advanced

- Specify output  (avoid temporary so speeding op)

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
----
y = np.zeros(10)
np.power(2, x, out=y[::2])  # calculate 2^x to every other element of a specified array
print(y)
1.   0.   2.   0.   4.   0.   8.   0.  16.   0.]
```

- Aggregates   .reduce  .accumulate
  - np.add.reduce(x) sum all x elements, .multiply.reduce calculate product of all elements
  - np.add.accumulate(x) sum storing all intemediate results array([  1,   2,   6,  24, 120])
- Outer products

```
x = np.arange(1, 6)
np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5],
[ 2,  4,  6,  8, 10],
[ 3,  6,  9, 12, 15],
[ 4,  8, 12, 16, 20],
[ 5, 10, 15, 20, 25]])
```

# NumPy array

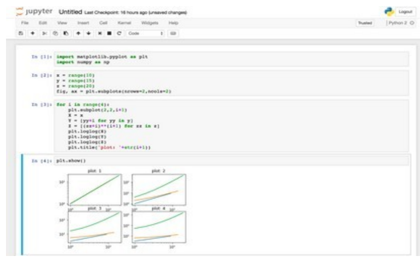
| Function Name | NaN-safe Version | Description |
|---|---|---|
| np.sum | np.nansum | Compute sum of elements |
| np.prod | np.nanprod | Compute product of elements |
| np.mean | np.nanmean | Compute mean of elements |
| np.std | np.nanstd | Compute standard deviation |
| np.var | np.nanvar | Compute variance |
| np.min | np.nanmin | Find minimum value |
| np.max | np.nanmax | Find maximum value |
| np.argmin | np.nanargmin | Find index of minimum value |
| np.argmax | np.nanargmax | Find index of maximum value |
| np.median | np.nanmedian | Compute median of elements |
| np.percentile | np.nanpercentile | Compute rank-based statistics of elements |
| np.any | N/A | Evaluate whether any elements are true |
| np.all | N/A | Evaluate whether all elements are true |

rand(d0, d1, …, dn) Random values in a given shape.
randn(d0, d1, …, dn) Return a sample (or samples) from the “standard normal” distribution.
randint(low[, high, size, dtype]) Return random integers from low (inclusive) to high (exclusive).
random_integers(low[, high, size]) Random integers of type np.int between low and high, inclusive.
random_sample([size]) Return random floats in the half-open interval [0.0, 1.0).
random([size]) Return random floats in the half-open interval [0.0, 1.0).
ranf([size]) Return random floats in the half-open interval [0.0, 1.0).
sample([size]) Return random floats in the half-open interval [0.0, 1.0).
choice(a[, size, replace, p]) Generates a random sample from a given 1-D array
bytes(length) Return random bytes.

- L = np.random.random(100) from module np.random
- sum(L) equiv to np.sum(L) but slower
  - big_array = np.random.rand(1000000)
  - %timeit sum(big_array) → 104msec
    %timeit np.sum(big_array) → 442 μs
  - moreover np.sum is multiple dimensions aware
- NaN version ignore missing values

# Example Aggregations

vedi jupyter notebook
EsempioAggreg



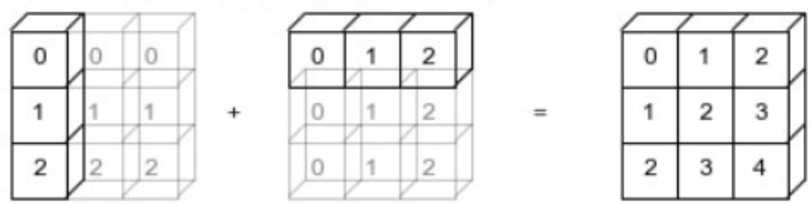
# Computation on array broadcasting

- Another way to vectorize operations
- Broadcasting means apply ufuncs **<u>on array of different sizes</u>**
- a = np.array([0, 1, 2])
- b = np.array([5, 5, 5])
- a + b
- a + 5 with broadcasting      5 is "*broadcasted*" to the dimension of a
- M = np.ones((3,3)) ; M + a

```
array([[ 1.,  2.,  3.],
[ 1.,  2.,  3.],
[ 1.,  2., 3.]])
```

```
array([[ 1.,  1.,  1.],
[ 1.,  1.,  1.],
[ 1.,  1., 1.]])
```

- a = np.arange(3) ; b = np.arange(3)[:, np.newaxis]
- a + b involves to stretch both elements



# Broadcasting rules

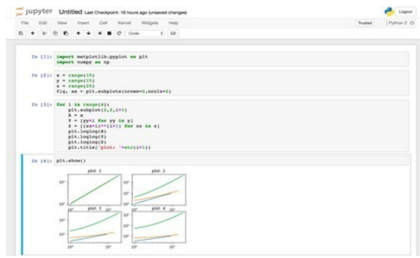
- Rule 1:
  - If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
- Rule 2
  - If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3
  - If in any dimension the sizes disagree and neither is equal to 1, an error is raised.
- Examples
  - centering array of data → 10 observation 1 obs 3-values
  - array 10 x3 → X = np.random.random((10,3))
  - mean across the first X.mean(axis=0) o Xmean = X.Mean(0)
  - center subtracting the mean X_centered = X – Xmean
  - Proof the centered array has 0 mean X_centered.mean(0) robba piccola ;)

```
>>> X = np.random.random((10,3))
>>> X
array([[ 0.20870753,  0.3899719 ,  0.35671064],
       [ 0.19931949,  0.23534587,  0.55731151],
       [ 0.90992898,  0.18694542,  0.85480008],
       [ 0.51044717,  0.63739161,  0.73303621],
       [ 0.73512571,  0.7373901 ,  0.22478389],
       [ 0.3881092 ,  0.14046176,  0.2233794 ],
       [ 0.65882515,  0.46310441,  0.19459834],
       [ 0.31383469,  0.13141054,  0.55104404],
       [ 0.13353636,  0.27612334,  0.26469895],
       [ 0.27995204,  0.77167453,  0.86768981]])
>>> X_mean = X.mean(axis=0)
>>> X_center = X - X_mean
>>> X_center
array([[-0.22507111, -0.00701005, -0.12609464],
       [-0.23445914, -0.16163608,  0.07450622],
       [ 0.47615034, -0.21003653,  0.37199479],
       [ 0.07666854,  0.24040966,  0.25023092],
       [ 0.30134708,  0.34040815, -0.25802139],
       [-0.04566943, -0.25652018, -0.25942589],
       [ 0.22504652,  0.06612246, -0.28820695],
       [-0.11994394, -0.26557141,  0.06823875],
       [-0.30024228, -0.12085861, -0.21810634],
       [-0.15382659,  0.37469258,  0.38488452]])
>>> X_mean
array([ 0.43377863,  0.39698195,  0.48280529])
```

```
>>> X_center.mean(0)
array([ -4.99600361e-17,  -1.11022302e-17,  -3.33066907e-17])
```

# Comparisons

vedi jupyter notebook
EsempioComparison



# Numpy more on indexing

vedi jupyter notebook
EsempioIndexing
Numpy1

x = [51 92 14 71 60 20 82 86 74 74]

- Indexing
  - Index x[0] → 51
  - Slices a [2:4] → [92,14]
  - Boolean Masks → something like [x[3], x[7], x[2]] → [71,86,14]
  - fancy → passing an **array of indices** to access multiple array elements at once
  - ind = [3, 7, 4]
  - x[ind] → array([71, 86, 60])
    - the shape of the result reflects the shape of the index arrays rather than the shape of the array being indexed!
    - so if ind = np.array([[3, 7],
    - [4, 5]]) then x[ind] = [[71, 86],
      [60, 20]]
    - Work also with multiple dimensions (see examples)
  - Combined Indexing
    - fancy + simple

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

0 1 2

X[2, [2, 0, 1]]

```
[[ 6,  4,  5],
 [10,  8,  9]]
```

# Sorting

- Selection sort
  - non un granchè $O[N^2]$
- np.sort $O[N\log^N]$
  - quicksort
  - mergesort
  - heapsort
  - not modifying input → np.sort(x)
  - x.sort() in-place
- np.argsort → return indices of sorted elements
  - x = np.array([2, 1, 4, 3, 5])
  - i = np.argsort(x) =>[1 0 3 2 4]
  - x → [1, 2, 3, 4, 5]

```
import numpy as np

def selection_sort(x):
for i in range(len(x)):
swap = i + np.argmin(x[i:])
(x[i], x[swap]) = (x[swap], x[i])
return x

x = np.array([2, 1, 4, 3, 5])
selection_sort(x)
array([1, 2, 3, 4, 5])
```

# Sorting 2

- **Sort along row/column**
  - x is [[6 ,3, 7, 4, 6, 9],
    [2, 6, 7, 4, 3, 7],
    [7, 2, 5, 4, 1, 7],
    [5, 1, 4, 0, 9, 5]])
  - each row/columns
    as an independent array

```
# Sort each colums
  np.sort(x,axis=0)

      [[2, 1, 4, 0, 1, 5],
[5, 2, 5, 4, 3, 7],
[6, 3, 7, 4, 6, 7],
[7, 6, 7, 4, 9, 9]]
```

```
# Sort each row
  np.sort(x,axis=1)

      [[3, 4, 6, 6, 7, 9],
[2, 3, 4, 6, 7, 7],
[1, 2, 4, 5, 7, 7],
[0, 1, 4, 5, 5, 9]]
```

- **Partitioning**
  - find k smallest value in array
  - np.partition(X,K)
    - smallest K values in X ( to the left ) the remaining to the right in casual order
    - x = np.array([7, 2, 3, 1, 6, 5, 4]); np.partition(x, 3) → array([2, 1, 3, 4, 6, 5, 7])
    - multidimensional np.partition(X, 2, axis=1)
    - np.argpartition return the indices

```
[[3, 4, 6, 7, 6, 9],
[2, 3, 4, 7, 6, 7],
[1, 2, 4, 5, 7, 7],
[0, 1, 4, 5, 9, 5]]
```

# Sorting Example

# Structured arrays

- several categories of data for a subject (e.g. name, age, and weight)
  - name = ['Alice', 'Bob', 'Cathy', 'Doug']
  - age = [25, 45, 37, 19]
  - weight = [55.0, 85.5, 68.0, 61.5]

hard to handle

- but as we can do something like x = np.zeros(4, **dtype=int**)
- we can do something like data = np.zeros(4, dtype={'names':('name', 'age', 'weight'), formats':('U10', 'i4', 'f8')})
  - U10 unicode max10 - i4 int < 4 - f8 8bytes float
  - data['name'] = name
  - data['age'] = age
  - data['weight'] = weight
  - data now is [('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0) ('Doug', 19, 61.5)]
  - and in a single block of memory
- Operations
  - data['name'] # Get all names
  - data[0] # Get first row of data
  - data[data['age'] < 30]['name'] # Data masking to get names where age is under 30

# Structured Arrays 2

- Creating structure arrays
  - np.dtype({'names':('name', 'age', 'weight'), 'formats':('U10', 'i4', 'f8')})
    - or by using **np dtypes**
  - np.dtype({'names':('name', 'age', 'weight'), 'formats':((np.str_, 10), int, np.float32)})
    - or a list of tuples
  - np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
- Complex types
  - with a *mat* component consisting of a 3×33×3 floating-point matrix:
  - tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
  - X = np.zeros(1, dtype=tp)

| Character | Description | Example |
|---|---|---|
| `'b'` | Byte | `np.dtype('b')` |
| `'i'` | Signed integer | `np.dtype('i4') == np.int32` |
| `'u'` | Unsigned integer | `np.dtype('u1') == np.uint8` |
| `'f'` | Floating point | `np.dtype('f8') == np.int64` |
| `'c'` | Complex floating point | `np.dtype('c16') == np.complex128` |
| `'S'`, `'a'` | String | `np.dtype('S5')` |
| `'U'` | Unicode string | `np.dtype('U') == np.str_` |
| `'V'` | Raw data (void) | `np.dtype('V') == np.void` |

# Structured Arrays

- np.recarray
  - fields accessed **as attributes** rather than as dictionary keys.
  - data['age']   array([25, 45, 37, 19], dtype=int32)

  data_rec = data.view(np.recarray)
  - data_rec.age
  - this kind of access is more  time consuming

# Pandas

- On top of numpy
  - Numpy not addressing
    - labels to data
    - working with missing data
    - fucntions to help in worksheet and database point of view
    - Series
    - Dataframe
- pd.<TAB> o pd?

## Pandas Series

- Series
  - one-dimensional array of indexed data
  - data = pd.Series([0.25, 0.5, 0.75, 1.0])
  - wraps → values + indices data.values data.index is a kind of Range
  - data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
  - index als non-contiguous or non sequential
  - data[1] ; data[1:3] ....
  - More flexybile than NumPy one dimensional array
  - A sort of specialize dictionary example
  - To create pd.Series(data, index=index)
  - index optional, data can be one of many entities

# Dataframes


- DataFrame is a generalized NumPy array
  - a collection of Series objects,
  - a single-column DataFrame can be constructed from a single Series
- generally we can construct dataframes
  - from a Series
    - pd.DataFrame({'population': population,'area': area})
  - explicit from  Numpy arrays
    - pd.DataFrame(np.random.rand(3, 2), columns=['foo', 'bar'], index=['a', 'b', 'c'])
  - from structured arrays
    - A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
    - array([(0, 0.0), (0, 0.0), (0, 0.0)], dtype=[('A', '<i8'), ('B', '<f8')])
    - pd.DataFrame(A)

| | A | B |
|---|---|---|
| 0 | 0 | 0.0 |
| 1 | 0 | 0.0 |
| 2 | 0 | 0.0 |

## Dataframes index

- Immutable arrays and ordered set ind = pd.Index([2, 3, 5, 7, 11])
  - ind[1] → 3 ; ind[::2] → 2,5,11 - step 2 all list
  - immutable so ind[3] = 0 → error
  - several attributes
    - prind.size, ind.shape, ind.ndim, ind.dtype)
- As ordered set we can
  - indA & indB
  - indA | indB
  - indA ^ indB symmetric difference (xor)

# Organizzazione ottimale dei dati

Characteristics / Variables / Features
(e.g. height)

Observations
(e.g. football players)

| Player | Nationality | Club | World_Champion | Height | Goals_2018 |
|---|---|---|---|---|---|
| **Lionel Messi** | Argentina | FC Barcelona | False | 1.70 | 45 |
| **Christiano Ronaldo** | Portugal | Juventus FC | False | 1.87 | 44 |
| **Neymar Junior** | Brasil | Paris SG | False | 1.75 | 28 |
| **Kylian Mbappe** | France | Paris SG | True | 1.78 | 21 |
| **Manuel Neuer** | Germany | FC Bayern | True | 1.93 | 0 |

Possibilmente un solo tipo di dati per colonna !!

# Data Indexing and Selection

- NumPy arrays  access, set, and modify
  - indexing → arr[2, 1])
  - slicing --> arr[:, 1:5])
  - masking → arr[arr > 0])
  - fancy indexing → arr[0, [1, 5]])
  - combinations → arr[:, [1, 5]])
- Pandas Series and Dataframe access, set, and modify
  - Series
    - **As dictionary** → data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
      - data['b'] → 0.5    'a' in data → true  data.keys → Index(['a', 'b', 'c', 'd'], dtype = 'object')
      - data['e'] = 1.25
      - list(data.items()) = [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
    - **As one dimensional array**
      - slicing by explicit index data['a':'c']   or implicit integer index   data[0:2]
      - masking data[(data > 0.3) & (data < 0.8)]  fancy  data[['a', 'e']]

# Indexers

- To avoid confusion between explicit e.g. data['a':'c'] (last included) and implicit data[0:2] last excluded
  - when data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])    mess!!
  - # explicit index when indexing data[1]
  - # implicit index when slicing data[1:3]


- loc    allows indexing and slicing that always references the **explicit** index
  - data.loc[1] → c     data.loc[1:3] →

    1  a
    3  b
- iloc  same but refers to **implicit Python-style index**
  - **data.iloc[1] → b     data.iloc[1:3] →**

    3  b
    5  c
- ix    hybrid  for Series objects is equivalent to standard []-based indexing

# Data Selection in DataFrame

- DataFrame acts as
  - dictionary of Series sharing the same index
- area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois': 149995})
- pop = pd.Series({'California': 38332521, 'Texas': 26448193,'New York': 19651127, 'Florida': 19552860, 'Illinois': 12882135})
- data = pd.DataFrame({'area':area, 'pop':pop})

| | area | pop |
|---|---|---|
| California | 423967 | 38332521 |
| Florida | 170312 | 19552860 |
| Illinois | 149995 | 12882135 |
| New York | 141297 | 19651127 |
| Texas | 695662 | 26448193 |

- data['area'] or data.area give us

```
California     423967
Florida        170312
Illinois       149995
New York       141297
Texas          695662
```

- data.area is data['area'] → true
  - possibly collide with data methods ( if existing)
- as a dictionary we can add a new element (a column) with the usual dictionary syntax data['density'] = data['pop'] / data['area']

| | area | pop | density |
|---|---|---|---|
| California | 423967 | 38332521 | 90.413926 |
| Florida | 170312 | 19552860 | 114.806121 |
| Illinois | 149995 | 12882135 | 85.883763 |
| New York | 141297 | 19651127 | 139.076746 |
| Texas | 695662 | 26448193 | 38.018740 |

## Data Selection in DataFrame

- DataFrame acts as
  - two-dimensional or structured array
- data.values give us the underlying structure

```
array([[  4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
       [  1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
       [  1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
       [  1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
       [  6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

- data.T

| | area | pop |
|---|---|---|
| California | 423967 | 38332521 |
| Florida | 170312 | 19552860 |
| Illinois | 149995 | 12882135 |
| New York | 141297 | 19651127 |
| Texas | 695662 | 26448193 |

| | California | Florida | Illinois | New York | Texas |
|---|---|---|---|---|---|
| area | 4.239670e+05 | 1.703120e+05 | 1.499950e+05 | 1.412970e+05 | 6.956620e+05 |
| pop | 3.833252e+07 | 1.955286e+07 | 1.288214e+07 | 1.965113e+07 | 2.644819e+07 |
| density | 9.041393e+01 | 1.148061e+02 | 8.588376e+01 | 1.390767e+02 | 3.801874e+01 |

- the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array so
  - data.values[0] → array([ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01]) a row !
  - data['area'] a column!

example

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
```

we need another indexing way to avoid mess

# DataFrameIndexing

- **iloc** data.iloc[:3, :2]
  - → we can index the underlying array as if it is a simple NumPy array
  - → DataFrame index and column labels are maintained in the result
- **loc** data.loc[:'Illinois', :'pop']
  - → index the underlying data in an array-like style using the explicit index and column names
- **ix** ~~data.ix[:3, :'pop']~~

| | area | pop |
|---|---|---|
| **California** | 423967 | 38332521 |
| **Florida** | 170312 | 19552860 |
| **Illinois** | 149995 | 12882135 |
| **New York** | 141297 | 19651127 |
| **Texas** | 695662 | 26448193 |

| | area | pop |
|---|---|---|
| **California** | 423967 | 38332521 |
| **Florida** | 170312 | 19552860 |
| **Illinois** | 149995 | 12882135 |

More

example

# Metodi di indexing

| Function | Description |
|---|---|
| Dataframe.head() | Return top  n rows of a data frame. |
| Dataframe.tail() | Return bottom n rows of a data frame. |
| Dataframe.at[] | Access a single value for a row/column label pair. |
| Dataframe.iat[] | Access a single value for a row/column pair by integer position. |
| Dataframe.tail() | Purely integer-location based indexing for selection by position. |
| **DataFrame.lookup()** | Label-based “fancy indexing” function for DataFrame. |
| DataFrame.pop() | Return item and drop from frame. |
| **DataFrame.xs()** | Returns a cross-section (row(s) or column(s)) from the DataFrame. |
| DataFrame.get() | Get item from object for given key (DataFrame column, Panel slice, etc.). |
| DataFrame.isin() | Return boolean DataFrame showing whether each element in the DataFrame is contained in values. |
| DataFrame.where() | Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other. |
| DataFrame.mask() | Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other. |
| DataFrame.query() | Query the columns of a frame with a boolean expression. |
| DataFrame.insert() | Insert column into DataFrame at specified location. |

## Pandas and data

- Numpy → ufunc
- Pandas add
  - unary operations e.g - sin(), preserve index and column labels in the output,
  - binary operations (e.g. + *), automatically align indices when passing the objects to the ufunc.
  - Esempi

## Missing data - None

- Real data set are rarely complete, clean, homogeneous, some data often missing
  - Different source can indicate missing data each is own way ( common problem in datawarehousing)
  - *Sentinel ( a specific value i.e. -999999 NaN) → reduces the data range, extra logic*
    - *Python None and float value NaN*
  - Mask (something like a flag) → more data → overhead in handling and computations
- vals1 = np.array([1, None, 3, 4])
  - if we review at the dtype of the vals1 values we find dtype=object
    so the best common representation for these objects is "generic Python object"
  - any operations on the data will be done at the Python level ! A lot of overhead
    - sum a million of object values will be about 20 times the sum of int values
  - It is not possible add/subtract a None value vals1.sum() → Error

## Missing data NaN

- vals2 = np.array([1, np.nan, 3, 4])
- vals2.dtype = float64    the common element is a **float**
  - 1 + np.Nan = NaN, 0*NaN = NaN
  - vals2.sum(), vals2.min(), vals2.max()  → (nan, nan, nan)
  - np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2) #ignore NaN Values
  - (8.0, 1.0, 4.0)
- Pandas
  - handle the two of them nearly interchangeably
  - converting where appropriate   pd.Series([1, np.**nan**, 2, **None**]) →
  - automatic cast
    - x = pd.Series(range(2), dtype=int); x[0] = None

```
0     1.0
1     NaN
2     2.0
3     NaN
dtype: float64
```

```
0     0
1     1
dtype: int64
```

```
0     NaN
1     1.0
dtype: float64
```

## Missing -Data - Operations

`data = pd.Series([1, np.nan, 'hello', None])`

- isnull(): Generate a boolean mask indicating missing values

data.isnull() →

```
0    False
1     True
2    False
3     True
dtype: bool
```

data[data.**notnull**()]

```
0        1
2    hello
dtype: object
```

- notnull(): Opposite of isnull()

**same for dataframes**

```
0        1
2    hello
dtype: object
```

- dropna(): Return a filtered version of the data
  - for DataFrame we can drop only entire rows or columns
    - specific options dflt rows, axis=1 drops columns containing a null value
    - how='any'|'all'  rows or coluns depending on axis in which *a* | *all* values is null

## Missing -Data - Operations

- fillna(): Return a copy of the data with missing values filled or imputed
- data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
  - fill NA entries with a single value, such as zero → data.fillna(0)
  - forward-fill to propagate the previous value forward data.fillna(method='ffill')
  - backardfill back-fill to propagate the next values backward
- Dataframe we have also to specify the axis

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

# Hierarchical indexing (Multi indexing)

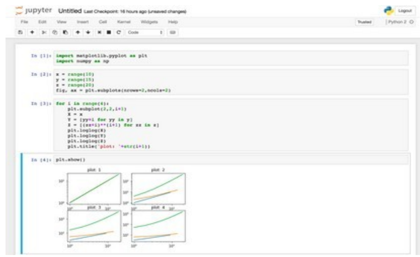
- Data indexed by more than one or two keys.
- Incorporate multiple index levels within a **single index**
  - **Multidimensional data can be represented as series or dataframes**
- First approach → simply use Python tuples as keys
  - create a series like example
  - you can index or slice the series based on this multiple index → true
  - but you haven't a coherent method

## Combining - Concatenation and Append

- Very similar to Numpy concat
- pd.concat
  - pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
    keys=None, levels=None, names=None, verify_integrity=False,
    copy=True)
  - used for a simple concatenation of Series or DataFrame objects
  - DataFrame → rowwis or specify axise
  - Pandas concatenations preserves indices
  - Can join
  - Shothand append → pd.concat([df1, df2]), you can simply call df1.append(df2):

# Combining

vedi jupyter notebook
EsempioPandasConcat



- Merge and Join
  - pd.merge
- Joins
  - one to one → similar to column wise concat   example

df1

| | employee | group |
|---|---|---|
| 0 | Bob | Accounting |
| 1 | Jake | Engineering |
| 2 | Lisa | Engineering |
| 3 | Sue | HR |

df2

| | employee | hire_date |
|---|---|---|
| 0 | Lisa | 2004 |
| 1 | Bob | 2008 |
| 2 | Jake | 2012 |
| 3 | Sue | 2014 |

| | employee | group | hire_date |
|---|---|---|---|
| 0 | Bob | Accounting | 2008 |
| 1 | Jake | Engineering | 2012 |
| 2 | Lisa | Engineering | 2004 |
| 3 | Sue | HR | 2014 |

# Combining – Many to one

- Many to one join
  - one of the two key columns contains duplicate entries

```
df3
```

| | employee | group | hire_date |
|---|---|---|---|
| **0** | Bob | Accounting | 2008 |
| **1** | Jake | Engineering | 2012 |
| **2** | Lisa | Engineering | 2004 |
| **3** | Sue | HR | 2014 |

```
df4
```

| | group | supervisor |
|---|---|---|
| **0** | Accounting | Carly |
| **1** | Engineering | Guido |
| **2** | HR | Steve |

```
pd.merge(df3, df4)
```

| | employee | group | hire_date | supervisor |
|---|---|---|---|---|
| **0** | Bob | Accounting | 2008 | Carly |
| **1** | Jake | Engineering | 2012 | Guido |
| **2** | Lisa | Engineering | 2004 | Guido |
| **3** | Sue | HR | 2014 | Steve |

# Combining – Many to many

- Many to many join
  - key column in both the left and right array contains duplicates

```
df1
```

| | employee | group |
|---|---|---|
| **0** | Bob | Accounting |
| **1** | Jake | Engineering |
| **2** | Lisa | Engineering |
| **3** | Sue | HR |

```
df5
```

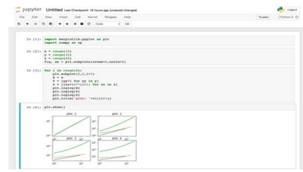
| | group | skills |
|---|---|---|
| **0** | Accounting | math |
| **1** | Accounting | spreadsheets |
| **2** | Engineering | coding |
| **3** | Engineering | linux |
| **4** | HR | spreadsheets |
| **5** | HR | organization |

```
pd.merge(df1, df5)
```

| | employee | group | skills |
|---|---|---|---|
| **0** | Bob | Accounting | math |
| **1** | Bob | Accounting | spreadsheets |
| **2** | Jake | Engineering | coding |
| **3** | Jake | Engineering | linux |
| **4** | Lisa | Engineering | coding |
| **5** | Lisa | Engineering | linux |
| **6** | Sue | HR | spreadsheets |
| **7** | Sue | HR | organization |

## Merging

vedi jupyter notebook
ExampleMerge-Select-Sort



- By default merge looks for one or more matching column names between the two inputs, and uses this as the key
- you can explicitly specify the name of the key column using the on keyword, which takes a column name or a list of column names

```
df1
```

| | employee | group |
|---|---|---|
| **0** | Bob | Accounting |
| **1** | Jake | Engineering |
| **2** | Lisa | Engineering |
| **3** | Sue | HR |

```
df2
```

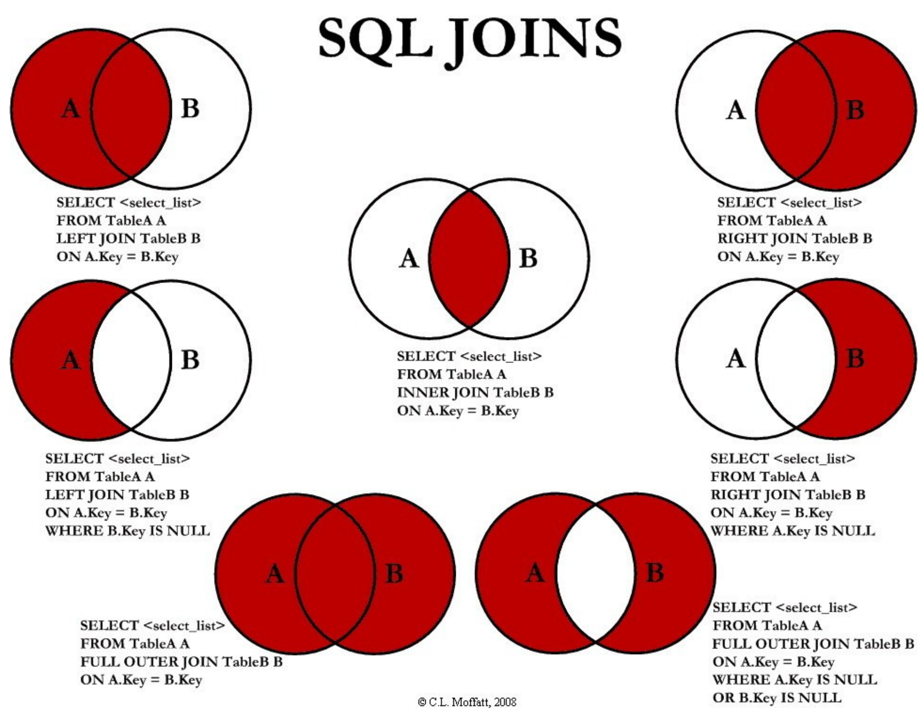
| | employee | hire_date |
|---|---|---|
| **0** | Lisa | 2004 |
| **1** | Bob | 2008 |
| **2** | Jake | 2012 |
| **3** | Sue | 2014 |

```
pd.merge(df1, df2, on='employee')
```

| | employee | group | hire_date |
|---|---|---|---|
| **0** | Bob | Accounting | 2008 |
| **1** | Jake | Engineering | 2012 |
| **2** | Lisa | Engineering | 2004 |
| **3** | Sue | HR | 2014 |

## Merging

- left_on and right_on keywords
  - specify the columns name on wich
  - perform the merge
- using indexes (left_index right index)
  - df1a = df1.set_index('employee')
  - df2a = df2.set_index('employee')
  - display('df1a', 'df2a')
- Mixing
  - display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
- how=inner|outer|left|right
- Suffixes  handle overlapping



# Differences between merge e concat

```
df1 = pd.DataFrame({'Key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df1:
Key  data1
0   b   0
1   b   1
2   a   2
3   c   3
4   a   4
5   a   5
6   b   6
df2 = pd.DataFrame({'Key': ['a', 'b', 'd'], 'data2': range(3)})
df2:
Key data2
0   a   0
1   b   1
2   d   2
```

## #Merge

```
#The 2 dataframes are merged on the basis on values in column "Key" as
it is a common column in 2 dataframes
pd.merge(df1, df2)
Key data data2
0   b   0   1
1   b   1   1
2   b   6   1
3   a   2   0
4   a   4   0
5   a   5   0
```

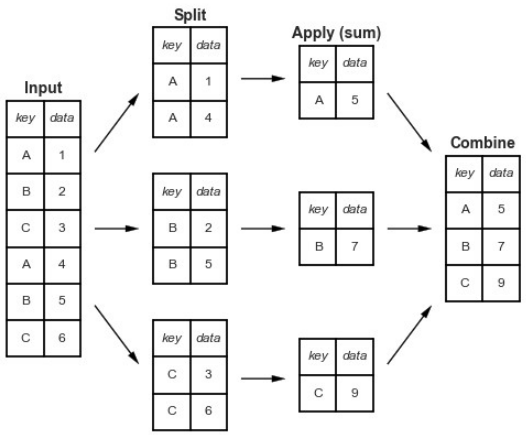
## #Concat

```
# df2 dataframe is appended at the bottom of df1
pd.concat([df1, df2])
Key data1   data2
0   b   0   NaN
1   b   1   NaN
2   a   2   NaN
3   c   3   NaN
4   a   4   NaN
5   a   5   NaN
6   b   6   NaN
0   a   Nan 0
1   b   Nan 1
2   d   Nan 2
```

# **Aggregation** and Grouping

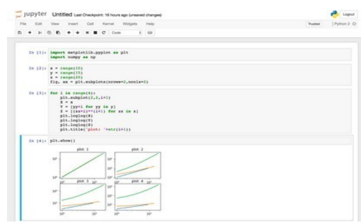
- computing aggregations like sum(), mean(), median(), min(), and max()

| Aggregation | Description |
|---|---|
| `count()` | Total number of items |
| `first()`, `last()` | First and last item |
| `mean()`, `median()` | Mean and median |
| `min()`, `max()` | Minimum and maximum |
| `std()`, `var()` | Standard deviation and variance |
| `mad()` | Mean absolute deviation |
| `prod()` | Product of all items |
| `sum()` | Sum of all items |




- Aggregate by condition
  - Groupbyoperations
    - The **split step** involves breaking up and grouping a DataFrame depending on the value of the specified key.
    - The **apply step** involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
    - The **combine step** merges the results of these operations into an output array.
  - aggregate(), filter(), transform(), and apply()

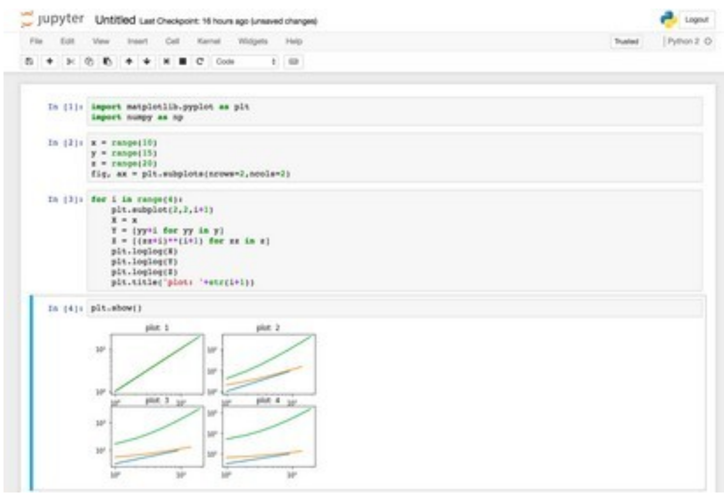
vedi jupyter notebook
EsempioPandasAggregationGrouping



## Pivot

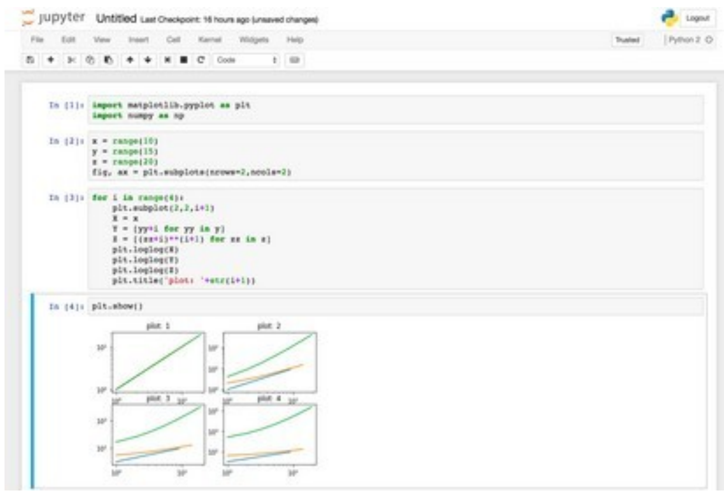
- Pivot table  get column-wise data as input, and groups the entries into a two-dimensional table
  - a multidimensional summarization of the data.
  - multidimensional version of *GroupBy*

vedi jupyter notebook
EsempioPDPivot



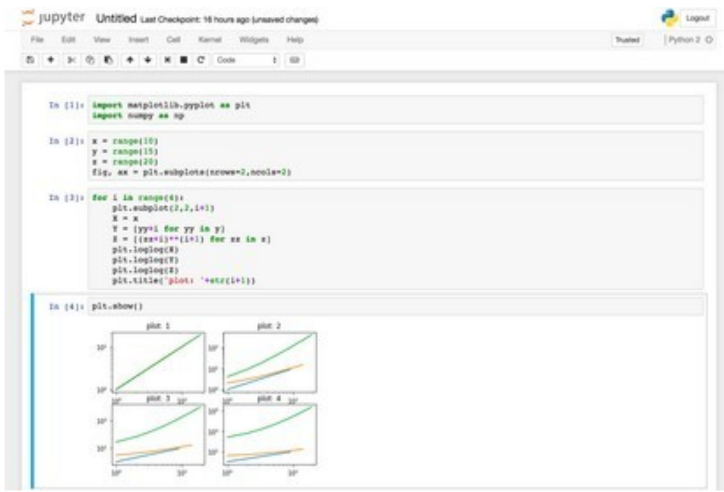
# Working with time series

vedi jupyter notebook
EsempioPandasTimeSeries



# Extend string manipulation with Pandas

vedi jupyter notebook
EsempioWorkingWithStrings



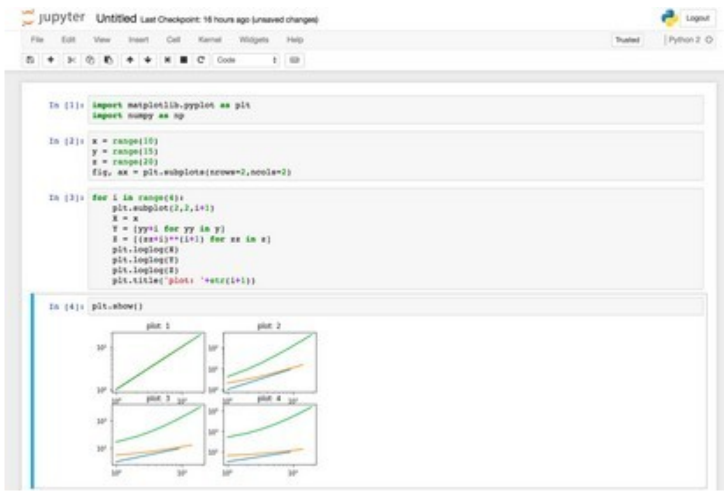
# Pandas IO Manipulation

| 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 | 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 |

- A lot of reader
  - CSV
    - text files (a.k.a. flat files) read_csv() and read_table().

# Esempio di lettura

vedi jupyter notebook
PandasLoadFile



# Getting Data in/out

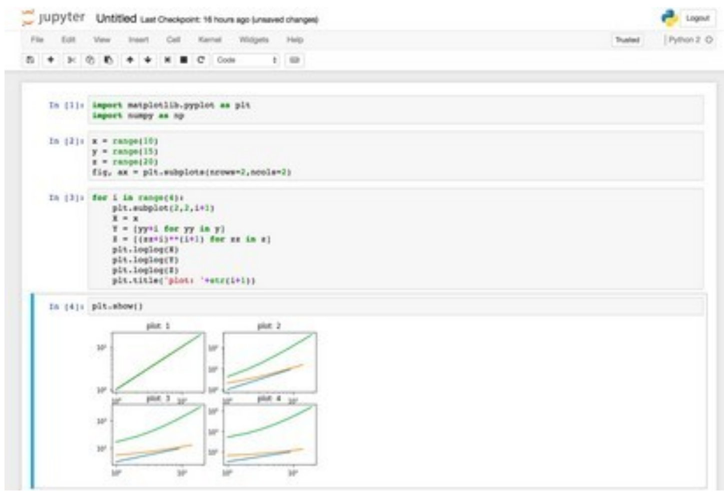
- HDF5 Store
  - HDF5 is a format designed to store large numerical arrays of homogenous type
    - a Python dictionary
    - import numpy as np
    - from pandas importHDFStore,DataFrame# create (or open) an hdf5 file and opens in append mode
    - hdf =HDFStore('storage.h5')
    - df =DataFrame(np.random.rand(5,3), columns=('A','B','C'))# put the dataset in the storage
    - hdf.put('d1', df, format='table', data_columns=True)
      - Accessing
        - print hdf['d1'].shape
        - from pandas import read_hdf
        - # this query selects the columns A and B# where the values of A is greather than 0.5
        - hdf = read_hdf('storage.h5','d1',where=['A>.5'], columns=['A','B'])
      - Appending
        - hdf.append('d1',DataFrame(np.random.rand(5,3),
        - columns=('A','B','C')),
        - format='table', data_columns=True)
        - hdf.close()# closes the file

## Getting data IN/OUT

- CSV/EXCEL
  - df.to_csv('foo.csv')
  - pd.read_csv('foo.csv')
  - df.to_excel('foo.xlsx', sheet_name='Sheet1')
  - pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])

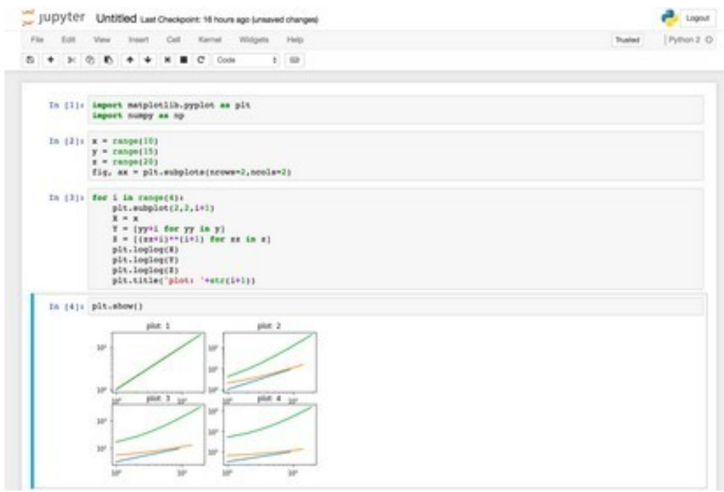
# Pandas IO CSV Basic Examples

vedi jupyter notebook
EsempiPandasIntro



# Extend string manipulation with Pandas

vedi jupyter notebook
EsempioWorkingWithStrings



# Con Pyodbc

- import pyodbc
- import pandas as pd
- ...
- cnxn = pyodbc.connect('DRIVER={{Microsoft Access Driver (*.mdb, *.accdb)}};DBQ=' +
-                       '{};Uid={};Pwd={};'.format(db_file, user, password)
- 
- query = "SELECT * FROM mytable WHERE INST = '796116'"
- dataf = pd.read_sql(query, cnxn)
- cnxn.close()

# Da dizionario

```
          0    1
0   2012-07-02  392
1   2012-07-06  392
2   2012-06-29  391
3   2012-06-28  391
...

In [12]: pd.DataFrame(d.items(), columns=['Date', 'DateValue'])
Out[12]:
        Date  DateValue
0   2012-07-02       392
1   2012-07-06       392
2   2012-06-29       391
```

```python
def get_response(q):
    d=[]
    try:
        cnx = mdb.connect(**DSN)
    except Exception as err:
        hexcept(err,exit=True)
    try:
        with cnx.cursor() as cursor:
            cursor.execute(q)
            desc = cursor.description
            column_names = [col[0] for col in desc]
            d = [dict(zip(column_names, row)) for row in cursor.fetchall()]
    except Exception as err:
        hexcept(err, exit=True)
    else:
        return d
```

# MatPlotLib

- Python 2D plotting library
  - produces publication quality figures in a variety of hardcopy formats and interactive environments.
  - Can generate with just a few lines of code:
  - plots, histograms, power spectra, bar charts,errorcharts, scatterplots, etc, .
  - matplotlib.pyplot
    - provides an object oriented interface to the plotting library (as plt)
- Matplotlib tutorial
- Matplotlib home page

# MatPlotLib vs pylab

- Exactly the same thing
- will run the exact same code, it is just different ways of importing the modules.
- matplotlib has two interface layers,
  - a state-machine layer managed by pyplot
  - OO interface pyplot is built on top of
- pylab is a clean way to bulk import a whole slew of helpful functions (the pyplot state machine function, most of numpy) into a single name space.
  - Import also al lot of numpy functions in pylab namespace
- The main reason this exists is to work with ipython to make a very nice interactive shell which more-or-less replicates MATLAB
- If you are not embedding in a gui (either using a non-interactive backend for bulk scripts or using one of the provided interactive backends) the typical thing to do is

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

## Matplotlib - PyPlot

- savefig
  - vector format pdf eps svg
  - raster format png
- Mode
  - interactive (dflt)
    - pylab.ion()
    - pylab.ioff().
  - non interactive ( needs() show at the end)
- matplotlibrc for defaults
  - lines.linewidth :1.0
  - # interactively mpl.rcParams=['lines.linewidth'] = 1.0

# plot

- figure(figsize=(5,5)) sets the figure size to 5inch by 5inch
- plot(x,y1,label='sin(x)') defines the name of this line.
- The line label will be shown in the legend if the legend() command is used later.
  - **Note that calling the plot command repeatedly, allows you to overlay a number of curves.**
- axis([-2,2,-1,1]) fixes the displayed area to go from xmin=-2 to xmax=2 in x-direction, and from ymin=-1 to ymax=1 in y-direction
- legend() a legend with the labels as defined in the plot command.
  - help("pylab.legend") to learn more about the placement of the legend.
- grid() display a grid on the backdrop.
- xlabel('...') and ylabel('...') labelling the axes further than you can chose different line styles, line thicknesses, symbols and colours for the data be plotted. (The syntax is very similar to MATLAB.)
  - plot(x,y,'og') will plot circles (o) in green (g)
  - plot(x,y,'-r') will plot a line (-) in red (r)
  - plot(x,y,'-b',linewidth=2) will plot a blue line (b) with two two pixel thickness linewidth=2 which is twice as wide as the default.

## Backends

- Matplotlib has a number of "backends" which are responsible for rendering graphs. The different backends are able to generate graphics with different formats and display/event loops. There is a distinction between noninteractive backends (such as 'agg', 'svg', 'pdf', etc.) that are only used to generate image files (e.g. with the savefig function), and interactive backends (such as Qt4Agg, GTK, MaxOSX) that can display a GUI window for interactively exploring figures.
- A list of available backends are
  - print(matplotlib.rcsetup.all_backends)
  - [u'GTK', u'GTKAgg', u'GTKCairo', u'MacOSX', u'Qt4Agg', u'Qt5Agg', u'TkAgg', u'WX', u'WXAgg', u'CocoaAgg', u'GTK3Cairo', u'GTK3Agg', u'WebAgg', u'nbAgg', u'agg', u'cairo', u'emf', u'gdk', u'pdf', u'pgf', u'ps', u'svg', u'template']
- The default backend, called agg, is based on a library for raster graphics which is great for generating raster formats like PNG