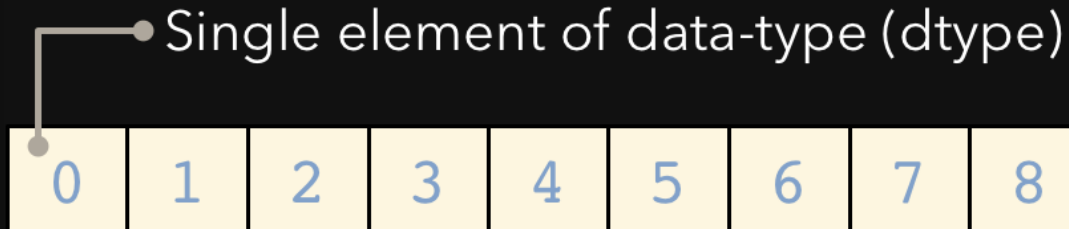


Introduction to NumPy

Bryan Van de Ven

What is NumPy

- NumPy is a Python C extension library for array-oriented computing
 - Efficient
 - In-memory
 - Contiguous (or Strided)
 - Homogeneous (but types can be algebraic)



- NumPy is suited to many applications
 - Image processing
 - Signal processing
 - Linear algebra
 - A plethora of others

Arrays – Numerical Python (Numpy)

- Lists ok for storing small amounts of one-dimensional data

```
>>> a = [1,3,5,7,9]
>>> print(a[2:4])
[5, 7]
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

```
>>> a = [1,3,5,7,9]
>>> b = [3,5,6,7,9]
>>> c = a + b
>>> print c
[1, 3, 5, 7, 9, 3, 5, 6, 7, 9]
```

- But, can't use directly with arithmetical operators (+, -, *, /, ...)
- Need efficient arrays with arithmetic and better multidimensional tools

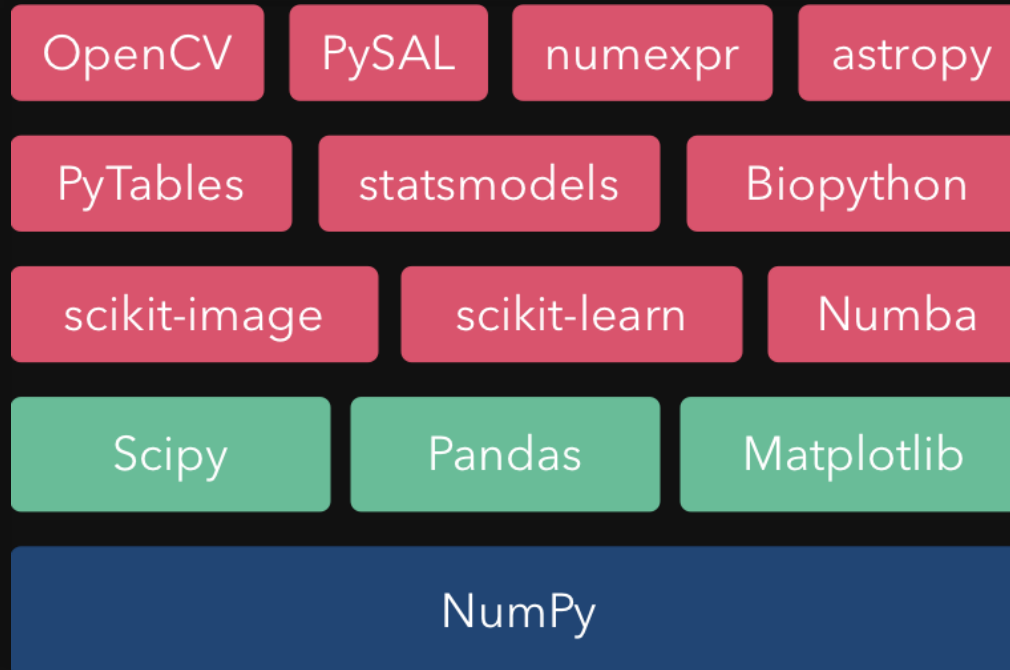
- **Numpy**

```
>>> import numpy (as np)
```

- Similar to lists, but much more capable, except fixed size

**NumPy is the foundation of the
python scientific stack**

NumPy Ecosystem



Numpy – Creating vectors

- From lists
 - `numpy.array`

```
# as vectors from lists
>>> a = numpy.array([1,3,5,7,9])
>>> b = numpy.array([3,5,6,7,9])
>>> c = a + b # common operations element-wise (more later)
>>> print c
[4, 8, 11, 14, 18]

>>> type(c)
(<type 'numpy.ndarray'>)

>>> c.shape
(5,)
```

Numpy – Creating Arrays

```
>>> l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]] # create a list
>>> a = numpy.array(l) # convert a list to an array
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
>>> a.shape
(3, 3)
>>> print(a.dtype) # get type of an array
int64
```

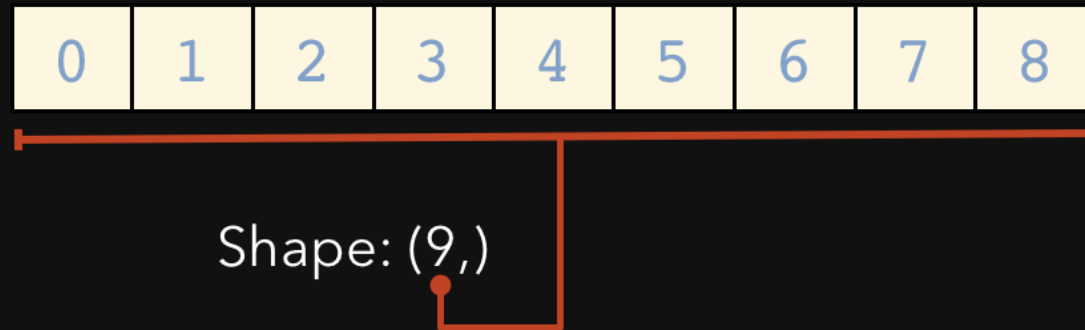
```
# or directly as matrix
>>> M = array([[1, 2], [3, 4]])
>>> M.shape
(2,2)
>>> M.dtype
dtype('int64')
```

```
#only one type
>>> M[0,0] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for long() with base 10:
'hello'

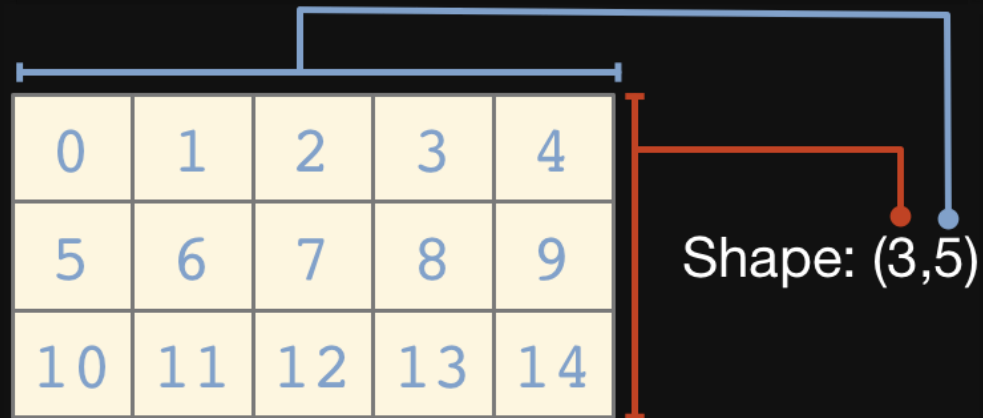
>>> M = numpy.array([[1, 2], [3, 4]], dtype=complex)
>>> M
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Array Shape

One dimensional arrays have a 1-tuple for their shape



...Two dimensional arrays have a 2-tuple

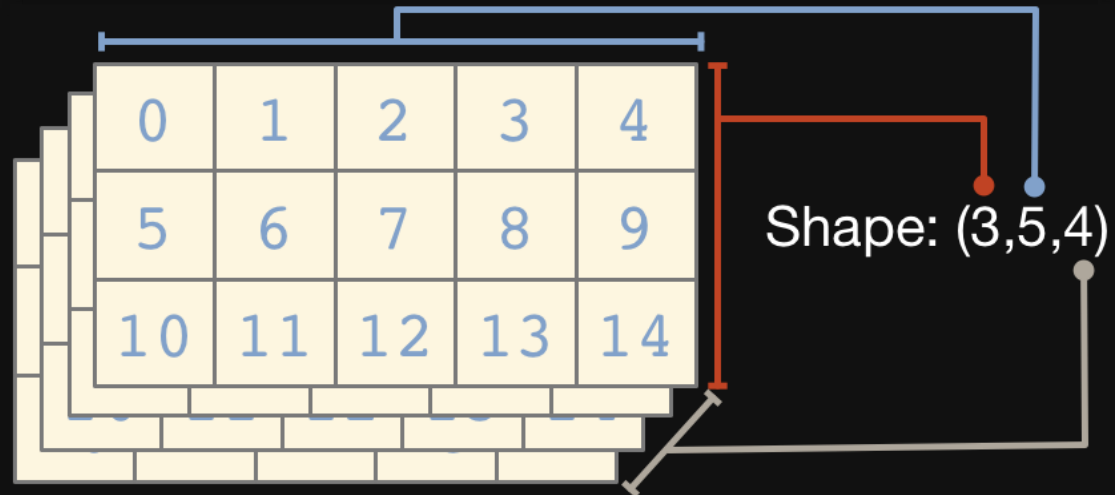


Shape must be maintained

- `A = np.array([[1,2,3], [4,5], [6]])`
- `A.shape` `#(3,)`
- `A.dtype` `#object`

- `A = np.array([[1,2,3], [4,5,6], [6,7,8]])`
- `A.shape` `#(3,3)`
- `A.dtype` `#int32`

...And so on



Indexing and Slicing

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

all values

`arr[0:2,:]`

`arr[2,1:]`

Implied end

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

`arr[:2, 2:3]`

Implied zero

NumPy array indices can also take an optional stride

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

`arr[:, ::2]`

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

`arr[:, :2, ::3]`

Array Views

Simple assignments do not make copies of arrays (same semantics as Python). Slicing operations do not make copies either; they return views on the original array.

```
In [2]: a = np.arange(10)

In [3]: b = a[3:7]

In [4]: b
Out[4]: array([3, 4, 5, 6])

In [5]: b[:] = 0

In [6]: a
Out[6]: array([0, 1, 3, 0, 0, 0, 0, 7, 8, 9])

In [7]: b.flags.owndata
Out[7]: False
```

Array views contain a pointer to the original data, but may have different shape or stride values. Views always have `flags.owndata` equal to `False`,

Numpy – Creating arrays

- Generation functions

```
>>> x = numpy.arange(0, 10, 1) # arguments: start, stop, step
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> numpy.linspace(0, 10, 25)
array([ 0.          ,  0.41666667,  0.83333333,  1.25          ,
        1.66666667,  2.08333333,  2.5          ,  2.91666667,
        3.33333333,  3.75          ,  4.16666667,  4.58333333,
        5.          ,  5.41666667,  5.83333333,  6.25          ,
        6.66666667,  7.08333333,  7.5          ,  7.91666667,
        8.33333333,  8.75          ,  9.16666667,  9.58333333, 10.          ])

>>> numpy.logspace(0, 10, 10, base=numpy.e)
array([ 1.00000000e+00,  3.03773178e+00,  9.22781435e+00,
        2.80316249e+01,  8.51525577e+01,  2.58670631e+02,
        7.85771994e+02,  2.38696456e+03,  7.25095809e+03,
        2.20264658e+04])
```

Numpy – Creating arrays

```
# a diagonal matrix
>>> numpy.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

>>> b = numpy.zeros(5)
>>> print(b)
[ 0.  0.  0.  0.  0.]
>>> b.dtype
dtype('float64')
>>> n = 1000
>>> my_int_array = numpy.zeros(n, dtype=numpy.int)
>>> my_int_array.dtype
dtype('int32')

>>> c = numpy.ones((3,3))
>>> c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Numpy – array creation and use

```
>>> d = numpy.arange(5)  # just like range()
>>> print(d)
[0 1 2 3 4]

>>> d[1] = 9.7
>>> print(d)  # arrays keep their type even if elements changed
[0 9 2 3 4]

>>> print(d*0.4)  # operations create a new array, with new type
[ 0.   3.6  0.8  1.2  1.6]

>>> d = numpy.arange(5, dtype=numpy.float)
>>> print(d)
[ 0.  1.  2.  3.  4.]

>>> numpy.arange(3, 7, 0.5)  # arbitrary start, stop and step
array([ 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

Numpy – array creation and use

```
>>> x, y = numpy.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
>>> x
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
# random data
>>> numpy.random.rand(5,5)
array([[ 0.51531133,  0.74085206,  0.99570623,  0.97064334,  0.5819413 ],
       [ 0.2105685 ,  0.86289893,  0.13404438,  0.77967281,  0.78480563],
       [ 0.62687607,  0.51112285,  0.18374991,  0.2582663 ,  0.58475672],
       [ 0.72768256,  0.08885194,  0.69519174,  0.16049876,  0.34557215],
       [ 0.93724333,  0.17407127,  0.1237831 ,  0.96840203,  0.52790012]])
```

<https://docs.scipy.org/doc/numpy/user/basics.creation.html>

<https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html#routines-array-creation>

Numpy – Creating arrays

- File I/O

```
>>> os.system('head DeBilt.txt')
"Stn", "Datum", "Tg", "qTg", "Tn", "qTn", "Tx", "qTx"
001, 19010101, -49, 00, -68, 00, -22, 40
001, 19010102, -21, 00, -36, 30, -13, 30
001, 19010103, -28, 00, -79, 30, -5, 20
001, 19010104, -64, 00, -91, 20, -10, 00
001, 19010105, -59, 00, -84, 30, -18, 00
001, 19010106, -99, 00, -115, 30, -78, 30
001, 19010107, -91, 00, -122, 00, -66, 00
001, 19010108, -49, 00, -94, 00, -6, 00
001, 19010109, 11, 00, -27, 40, 42, 00
```

```
>>> data = numpy.genfromtxt('DeBilt.txt', delimiter=',', skip_header=1)
>>> data.shape
(25568, 8)
```

```
>>> numpy.savetxt('datasaved.txt', data)
>>> os.system('head datasaved.txt')
1.0000000000000000e+00 1.901010100000000000e+07 -4.9000000000000000e+01
0.0000000000000000e+00 -6.8000000000000000e+01 0.0000000000000000e+00 -
2.2000000000000000e+01 4.0000000000000000e+01
```

Numpy – Creating arrays

```
>>> M = numpy.random.rand(3,3)
>>> M
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464   ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> numpy.save('saved-matrix.npy', M)
>>> numpy.load('saved-matrix.npy')
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464   ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> os.system('head saved-matrix.npy')
NUMPYF{'descr': '<f8', 'fortran_order': False, 'shape': (3, 3), }
ï<
£¾ðê?sy²æ?$/÷ÖVñë?Ù4ê?%dn,í?Ã[Äjóë?Ä,zÑ?Ç
îâNê?ó7L{êá?0
>>>
```

<https://docs.scipy.org/doc/numpy/user/basics.io.html>

<https://docs.scipy.org/doc/numpy/reference/routines.io.html>

Universal Functions(ufuncs)

NumPy ufuncs are functions that operate element-wise on one or more arrays

a

0	1	2	3	4
---	---	---	---	---

b

0	10	20	30	40
---	----	----	----	----

c

0	11	22	33	44
---	----	----	----	----

$$c = a + b$$

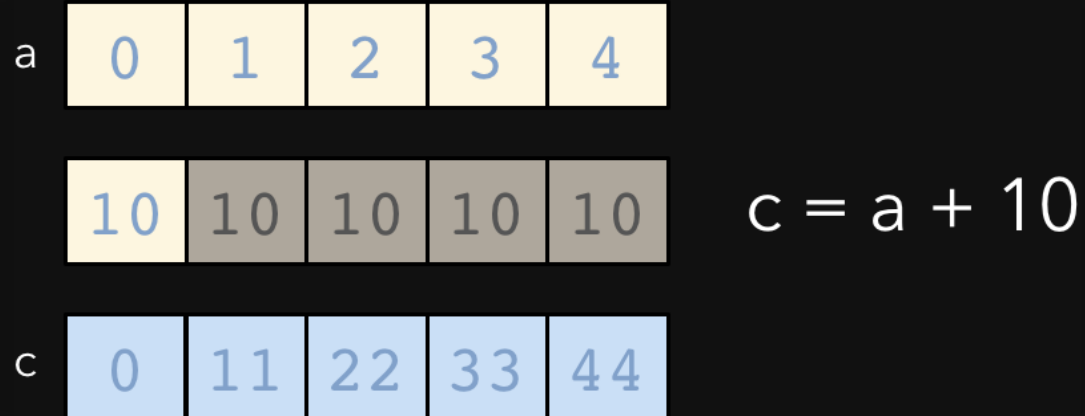
ufuncs dispatch to optimized C inner-loops based on array dtype

NumPy has many built-in ufuncs

- **comparison:** `<`, `<=`, `==`, `!=`, `>=`, `>`
- **arithmetic:** `+`, `-`, `*`, `/`, `reciprocal`, `square`
- **exponential:** `exp`, `expm1`, `exp2`, `log`, `log10`, `log1p`, `log2`, `power`, `sqrt`
- **trigonometric:** `sin`, `cos`, `tan`, `asin`, `arccos`, `atan`
- **hyperbolic:** `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- **bitwise operations:** `&`, `|`, `~`, `^`, `left_shift`, `right_shift`
- **logical operations:** `and`, `logical_xor`, `not`, `or`
- **predicates:** `isfinite`, `isinf`, `isnan`, `signbit`
- **other:** `abs`, `ceil`, `floor`, `mod`, `modf`, `round`, `sinc`, `sign`, `trunc`

Broadcasting

A key feature of NumPy is broadcasting, where arrays with different, but compatible shapes can be used as arguments to ufuncs



In this case an array scalar is broadcast to an array with shape (5,)

A slightly more involved broadcasting example in two dimensions

$$c = a + b$$

0	1
2	3
4	5

a

+

10	10
20	20
30	30

b

=

0	11
22	23
34	35

c

Here an array of shape $(3, 1)$ is broadcast to an array with shape $(3, 2)$

Broadcasting Rules

In order for an operation to broadcast, the size of all the trailing dimensions for both arrays must either:

be equal OR be one

A	(1d array):	3
B	(2d array):	2 x 3
Result	(2d array):	2 x 3

A	(2d array):	6 x 1
B	(3d array):	1 x 6 x 4
Result	(3d array):	1 x 6 x 4

A	(4d array):	3 x 1 x 6 x 1
B	(3d array):	2 x 1 x 4
Result	(4d array):	3 x 2 x 6 x 4

Array Methods

- **Predicates**
 - `a.any()`, `a.all()`
- **Reductions**
 - `a.mean()`, `a.sum()`, `a.argmin()`, `a.argmax()`,
`a.trace()`, `a.cumsum()`, `a.cumprod()`
- **Manipulation**
 - `a.argsort()`, `a.transpose()`, `a.reshape(...)`,
`a.ravel()`, `a.fill(...)`, `a.clip(...)`
- **Complex Numbers**
 - `a.real`, `a.imag`, `a.conj()`

Reduction: Axis

Array method reductions take an optional `axis` parameter that specifies over which axes to reduce `axis=None` reduces into a single scalar

```
In [7]: a.sum()  
Out[7]: 105
```

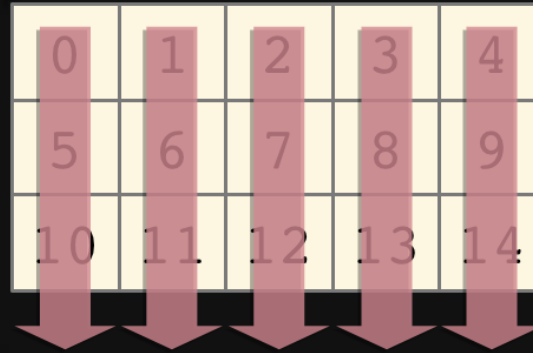
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

`axis=None`

`axis=None` is the default

`axis=0` reduces into the zeroth dimension

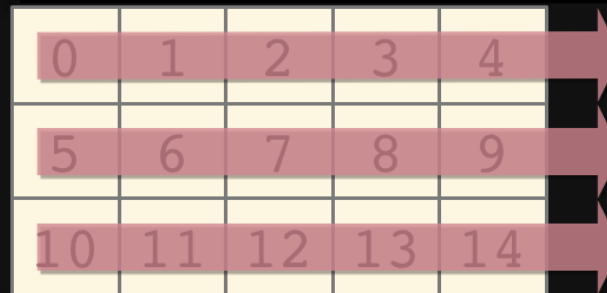
```
In [8]: a.sum(axis=0)  
Out[8]: array([15, 18, 21, 24,  
27])
```



`axis=0`

`axis=1` reduces into the first dimension

```
In [9]: a.sum(axis=1)  
Out[9]: array([10, 35, 60])
```



`axis=1`

Numpy – array creation and use

```
>>> arr = numpy.arange(10, 20 )
>>> div_by_3 = arr%3 == 0  # comparison produces boolean array
>>> print(div_by_3)
[ False False  True False False  True False False  True False]
>>> print(arr[div_by_3])  # can use boolean lists as indices
[12 15 18]

>>> arr = numpy.arange(10, 20) . reshape((2,5))
[[10 11 12 13 14]
 [15 16 17 18 19]]
```

Numpy – array methods

```
>>> arr.sum()
145
>>> arr.mean()
14.5
>>> arr.std()
2.8722813232690143
>>> arr.max()
19
>>> arr.min()
10
>>> div_by_3.all()
False
>>> div_by_3.any()
True
>>> div_by_3.sum()
3
>>> div_by_3.nonzero()
(array([2, 5, 8]),)
```

Numpy – array methods - sorting

```
>>> arr = numpy.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
>>> arr.sort() # acts on array itself
>>> print(arr)
[ 1.2  1.8  2.3  4.5  5.5  6.7]

>>> x = numpy.array([4.5, 2.3, 6.7, 1.2, 1.8, 5.5])
>>> numpy.sort(x)
array([ 1.2,  1.8,  2.3,  4.5,  5.5,  6.7])

>>> print(x)
[ 4.5  2.3  6.7  1.2  1.8  5.5]

>>> s = x.argsort()
>>> s
array([3, 4, 1, 0, 5, 2])
>>> x[s]
array([ 1.2,  1.8,  2.3,  4.5,  5.5,  6.7])
>>> y[s]
array([ 6.2,  7.8,  2.3,  1.5,  8.5,  4.7])
```

Numpy – array operations

```
>>> a = array([[1.0, 2.0], [4.0, 3.0]])
>>> print a
[[ 1. 2.]
 [ 3. 4.]]

>>> a.transpose()
array([[ 1., 3.],
       [ 2., 4.]])

>>> inv(a)
array([[-2. , 1. ],
       [ 1.5, -0.5]])

>>> u = eye(2) # unit 2x2 matrix; "eye" represents "I"

>>> u
array([[ 1., 0.],
       [ 0., 1.]])

>>> j = array([[0.0, -1.0], [1.0, 0.0]])

>>> dot (j, j) # matrix product
array([[-1., 0.],
       [ 0., -1.]])
```

Numpy – statistics

In addition to the mean, var, and std functions, NumPy supplies several other methods for returning statistical features of arrays. The median can be found:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form `[[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...]` where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.          ,  0.72870505],
       [ 0.72870505,  1.          ]])
```

Here the return array `c[i,j]` gives the correlation coefficient for the *i*th and *j*th observables. Similarly, the covariance for data can be found::

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Using arrays wisely

- Array operations are implemented in C or Fortran
- Optimised algorithms - i.e. fast!
- Python loops (i.e. `for i in a:...`) are much slower
- Prefer array operations over loops, especially when speed important
- Also produces shorter code, often more readable

Numpy – matrices

For **two dimensional** arrays NumPy defined a special matrix class in module matrix. Objects are created either with `matrix()` or `mat()` or converted from an array with method `asmatrix()`.

```
>>> import numpy
>>> m = numpy.mat([[1,2],[3,4]])
or
>>> a = numpy.array([[1,2],[3,4]])
>>> m = numpy.mat(a)
or
>>> a = numpy.array([[1,2],[3,4]])
>>> m = numpy.asmatrix(a)
```

Note that the statement `m = mat(a)` creates a copy of array 'a'. Changing values in 'a' will not affect 'm'. On the other hand, method `m = asmatrix(a)` returns a new reference to the same data. Changing values in 'a' will affect matrix 'm'.

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html>

Numpy – matrices

Array and matrix operations may be quite different!

```
>>> a = array([[1,2],[3,4]])
>>> m = mat(a) # convert 2-d array to matrix
>>> m = matrix([[1, 2], [3, 4]])
>>> a[0]          # result is 1-dimensional
array([1, 2])
>>> m[0]          # result is 2-dimensional
matrix([[1, 2]])
>>> a*a          # element-by-element multiplication
array([[ 1,  4], [ 9, 16]])
>>> m*m          # (algebraic) matrix multiplication
matrix([[ 7, 10], [15, 22]])
>>> a**3          # element-wise power
array([[ 1,  8], [27, 64]])
>>> m**3          # matrix multiplication m*m*m
matrix([[ 37, 54], [ 81, 118]])
>>> m.T          # transpose of the matrix
matrix([[1, 3], [2, 4]])
>>> m.H          # conjugate transpose (differs from .T for complex matrices)
matrix([[1, 3], [2, 4]])
>>> m.I          # inverse matrix
matrix([[ -2. ,  1. ], [ 1.5, -0.5]])
```

Numpy – matrices

- Operator `*`, `dot()`, and `multiply()`:
 - For array, `*` **means element-wise multiplication**, and the `dot()` function is used for matrix multiplication.
 - For matrix, `*` **means matrix multiplication**, and the `multiply()` function is used for element-wise multiplication.
- Handling of vectors (rank-1 arrays)
 - For array, the vector shapes `1xN`, `Nx1`, and `N` are all different things. Operations like `A[:,1]` return a rank-1 array of shape `N`, not a rank-2 of shape `Nx1`. Transpose on a rank-1 array does nothing.
 - For matrix, rank-1 arrays are always upgraded to `1xN` or `Nx1` matrices (row or column vectors). `A[:,1]` returns a rank-2 matrix of shape `Nx1`.
- Handling of higher-rank arrays (rank > 2)
 - array objects can have rank > 2.
 - matrix objects always have exactly rank 2.
- Convenience attributes
 - array has a `.T` attribute, which returns the transpose of the data.
 - matrix also has `.H`, `.I`, and `.A` attributes, which return the conjugate transpose, inverse, and `asarray()` of the matrix, respectively.
- Convenience constructor
 - The array constructor takes (nested) Python sequences as initializers. As in `array([[1,2,3],[4,5,6]])`.
 - The matrix constructor additionally takes a convenient string initializer. As in `matrix("[1 2 3; 4 5 6]")`

Task no. 4

- Supervised Machine Learning: regression
- Wine quality dataset
 - Dataset from UCI Repository, <https://archive.ics.uci.edu/ml/datasets/Wine+Quality>
 - winequality-white.csv
 - Sensorical quality prediction based on chemical properties of white wines
 - First 10% instances are test set (490 wines), the rest is train set

- Error metric: Root Mean Square Error (RMSE)

https://en.wikipedia.org/wiki/Root-mean-square_deviation

$$\sqrt{\sum_i (\text{predicted}_i - \text{trueValue}_i)^2 / n}$$

- **Load dataset** (csv file, all attributes are numeric, use some variant of numPy I/O)
- Optional, highly recommended: basic **statistics of data** (mean, median, min/max, correlation matrix, histograms...)

<https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.statistics.html>

- *Optional: feature extraction / normalization / ...*

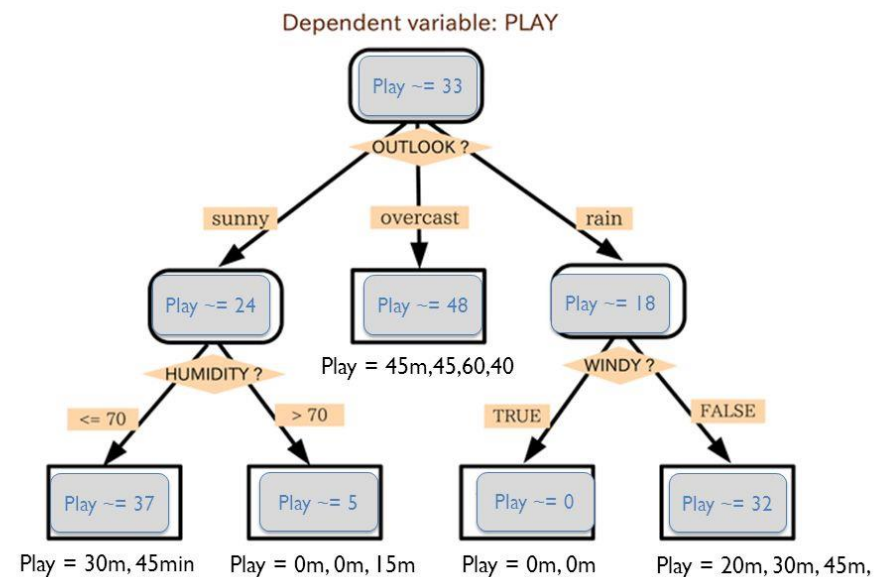
```
import numpy as np
import matplotlib.pyplot as plt
hist, edges = np.histogram([1, 1, 2, 2, 3], bins = range(5))
plt.bar(edges[:-1], hist, width = 1)
```

- **Fit model** on the train set
- **Predict** test set output based on the model, evaluate, report results

Task no. 4

- Some learning methods to choose from:
 - K-NN (you already know)
<http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html#sklearn.neighbors.KNeighborsRegressor>
 - Linear regression
 - You may make it polynomial by adding k-th powers of original features
 - <https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html>
 - http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
 - Decision Trees
 - Binary splits on attributes (segmentation of the data)
 - More common for classification, but can be used for regression
 - Understandable rules in the output
 - <http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html#sklearn.tree.DecisionTreeRegressor>
- .fit(input, output)
- .predict(input)

A regression tree



Install std. Modules (package managers)

- If running Annaconda, most of the common scientific stack packages are pre-installed
 - However, not all
- PIP, CONDA
 - PIP has slightly better coverage of modules
 - Conda handles dependancies better and has some advanced features, e.g. environments
- Basic commands (command line):
 - `pip install package_name`
 - `conda install package_name`
- PyCharm: Settings -> project interpreter -> „+“

