Computational Tools for Big Data — Python Libraries

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

September 15, 2015
Overview

Numpy — numerical arrays with fast computation

Scipy — computation science functions

Scikit-learn (sklearn) — machine learning

Pandas — Annotated numpy arrays

Cython — write C program in Python
Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
```
Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]  # Not [3, 6, 9]!
```
Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]  # Not [3, 6, 9] !

>>> [1, 2, 3] + 1  # Wants [2, 3, 4] ...
```
Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3] # Not [3, 6, 9] !

>>> [1, 2, 3] + 1 # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```
Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3] # Not [3, 6, 9] !

>>> [1, 2, 3] + 1 # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> # Matrix multiplication
>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]
```

Finn Årup Nielsen
Computational Tools for Big Data — Python libraries

Python numerics

The problem with Python:

```python
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]  # Not [3, 6, 9]!

>>> [1, 2, 3] + 1  # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> # Matrix multiplication
>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```
map(), reduce() and filter()

Poor-man’s vector operations: the map() built-in function:

```python
>>> map(lambda x: 3*x, [1, 2, 3])
[3, 6, 9]
```

lambda is for an anonymous function, x the input argument, and 3*x the function and the return argument. Also possible with ordinary functions:

```python
>>> from math import sin, pow
>>> map(sin, [1, 2, 3])
[0.8414709848078965, 0.90929742682568171, 0.14112000805986721]

>>> map(pow, [1, 2, 3], [3, 2, 1])
[1.0, 4.0, 3.0]
```
... or with comprehensions

List comprehensions:

```python
>>> [3*x for x in [1, 2, 3]]
[3, 6, 9]
```

Generator

```python
>>> (3*x for x in [1, 2, 3])
<generator object <genexpr> at 0x7f9060a72eb0>
```
Problem

But control structures are usually slow in Python.
Solution: Numpy

>>> from numpy import *
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
```
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
```
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
```
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
```
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * array([[5, 6], [7, 8]])
```

Finn Årup Nielsen 16 September 2015
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * array([[5, 6], [7, 8]])
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...
Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...

```python
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]  # error
```

Solution: Numpy

```python
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...

```python
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
matrix([[19, 22],
        [43, 50]])
```
Numpy arrays . . .

There are two basic types array and matrix:

An array may be a vector (one-dimensional array)

```python
>>> from numpy import *
>>> array([1, 2, 3, 4])
array([1, 2, 3, 4])
```

Or a matrix (a two-dimensional array)

```python
>>> array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```
... Numpy arrays ...

Or a higher-order tensor. Here a 2-by-2-by-2 tensor:

```python
>>> array([[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]])
array([[[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]]])
```

$N \times 1 \times N$ and $N \times 1$ array-s are different:

```python
>>> array([[1, 2, 3, 4]])[2]  # There is no third row
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index out of bounds
```
Numpy arrays

A matrix is always two-dimensional, e.g., this

```python
>>> matrix([[1, 2, 3, 4]])
```

is a two-dimensional data structure with one row and four columns

...and with a 2-by-2 matrix:

```python
>>> matrix([[1, 2], [3, 4]])
```
Numpy arrays copy

To copy by reference use `asmatrix()` or `mat()` (from an array) or `asarray()` (from a matrix)

```python
>>> a = array([1, 2, 3, 4])
>>> m = asmatrix(a)  # Copy as reference
>>> a[0] = 1000
>>> m
matrix([[1000, 2, 3, 4]])
```

To copy elements use `matrix()` or `array()`

```python
>>> a = array([1, 2, 3, 4])
>>> m = matrix(a)  # Copy elements
>>> a[0] = 1000
>>> m
matrix([[1, 2, 3, 4]])
```
Datatypes

Elements are 4 bytes integers or 8 bytes float per default:

```python
>>> array([1, 2, 3, 4]).itemsize # Number of bytes for each 4
```

```python
>>> array([1., 2., 3., 4.]).itemsize
8
```

```python
>>> array([1., 2, 3, 4]).itemsize # Not heterogeneous
8
```

array and matrix can be called with datatype to set it otherwise:

```python
>>> array([1, 2], 'int8').itemsize
1
```

```python
>>> array([1, 2], 'float32').itemsize
4
```
Initialization of arrays

Functions ones(), zeros(), eye() (also identity()), linspace() work “as expected” (from Matlab), though the first argument for ones() and zeros() should contain the size in a list or tuple:

```python
>>> zeros((1, 2))          # zeros(1, 2) doesn’t work
array([[ 0., 0.]])
```

To generated a list of increasing numbers:

```python
>>> r_[1:11]
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> arange(1, 11)
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```
Computational Tools for Big Data — Python libraries

**Diagonal . . .**

The `diagonal()` function works both for matrix and two-dimensional array:

```python
>>> diagonal(matrix([[1, 2], [3, 4]]))
array([1, 4])
```

Note now the vector/matrix is an one-dimensional array.

It “works” for higher order arrays:

```python
>>> diagonal(array([[1, 2], [3, 4], [[5, 6], [7, 8]]]))
array([[1, 7],
       [2, 8]])
```

`diagonal()` does not work for one-dimensional arrays.
Yet another function: `diag()`. Works for one- and two-dimensional matrix and array.

```python
>>> m = matrix([[1, 2], [3, 4]])
>>> d = diag(m)
>>> d
array([1, 4])
>>> diag(d)
array([[1, 0],
       [0, 4]])
```

Like Matlab's `diag()`.

It is also possible to specify the diagonal: `diag(m, 1)`
Matrix transpose

Matrix transpose is different with Python’s array and matrix and Matlab

```python
>>> A = array([[1+1j, 1+2j], [2+1j, 2+2j]]); A
array([[ 1.+1.j, 1.+2.j],
       [ 2.+1.j, 2.+2.j]])

.T for array and matrix (like Matlab “.’”) and .H for matrix (like Matlab “,’”):

>>> A.T # No conjugation. Also: A.transpose() or transpose(A)
array([[ 1.+1.j, 2.+1.j],
       [ 1.+2.j, 2.+2.j]])

>>> mat(A).H # Complex conjugate transpose. Or: A.conj().T
matrix([[ 1.-1.j, 2.-1.j],
        [ 1.-2.j, 2.-2.j]])
```
Matrix transpose and copy

```python
>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T
>>> B[0,0] = 45
>>> A
array([[45,  2],
       [ 3,  4]])
```

B is a reference to the transposed version.
Matrix transpose and copy

```python
>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T
>>> B[0,0] = 45
>>> A
array([[45,  2],
       [ 3,  4]])

B is a reference to the transposed version.

>>> A = np.array([[1, 2], [3, 4]])
>>> B = A.T.copy()  # Copy!
>>> B[0,0] = 45
>>> A
array([[1,  2],
       [3,  4]])
```
Sizes and reshaping

A 2-by-2-by-2 tensor:

```python
c>>> a = array ([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
c>>> a.ndim  # Number of dimensions
3
c>>> a.shape  # Number of elements in each dimension
(2, 2, 2)
c>>> a.shape = (1, 8); a  # Reshaping: 1-by-8 matrix
array ([[1, 2, 3, 4, 5, 6, 7, 8]])
c>>> a.shape = 8; a  # 8-element vector
array ([1, 2, 3, 4, 5, 6, 7, 8])
```

There is a related function for the last line:

```python
c>>> a.flatten ()  # Always copy
array ([1, 2, 3, 4, 5, 6, 7, 8])
```
Indexing . . .

Ordinary lists:

```python
>>> L = [[1, 2], [3, 4]]
>>> L[1, 1]
```

What happens here?
Indexing . . .

Ordinary lists:

```python
>>> L = [[1, 2], [3, 4]]
>>> L[1, 1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been \( L[1][1] \). With matrices:

```python
>>> mat(L)[1, 1]
```
Indexing ... 

Ordinary lists:

```python
>>> L = [[1, 2], [3, 4]]
```

```python
>>> L[1, 1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been `L[1][1]`. With Numpy matrices:

```python
>>> mat(L)[1, 1]
```

4
...Indexing...

>>> mat(L)[1][1]

What happens here?
. . . Indexing . . .

```python
>>> mat(L)[1][1]
```

What happens here? Error message:

```
IndexError: index out of bounds
```

```python
>>> mat(L)[1]
```
...Indexing...

```python
>>> mat(L)[1][1]

What happens here? Error message:

IndexError: index out of bounds

>>> mat(L)[1]
matrix([[3, 4]])
>>> asarray(L)[1]
array([3, 4])
>>> asarray(L)[1][1]
4
>>> asarray(L)[1,1]
4
```
Indexing with multiple indices

```python
>>> A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> i = [1, 2]  # Second and third row/column
>>> A[i,i]
matrix([[5, 9]])  # Elements from diagonal
>>> A[ix_(i,i)]  # Block matrix
matrix([[5, 6],
        [8, 9]])

Take out third, “fourth” and “fifth” column with wrapping:

```python
>>> A.take([2, 3, 4], axis=1, mode='wrap')
matrix([[3, 1, 2],
        [6, 4, 5],
        [9, 7, 8]])
```
Conditionals & concatenation

Construct a matrix based on a condition (first input argument)

```python
>>> where(mod(A.flatten(), 2), 1, 0)  # Find odd numbers
matrix([[1, 0, 1, 0, 1, 0, 1, 0, 1]])
```

Horizontal concatenate as in Matlab “[ A A ]” and note tuple input!

```python
>>> B = concatenate((A, A), axis=1)  # Or:
>>> B = bmat('A A')  # Note string input. Or:
>>> hstack((A, A))
matrix([[1, 2, 3, 1, 2, 3],
        [4, 5, 6, 4, 5, 6],
        [7, 8, 9, 7, 8, 9]])
```

For concatenating rows use `vstack()`, `bmat('A ; A')` or `concatenate()`
Random initialization with `random`

Python with one element at a time with the `random` module:

```python
>>> import random
>>> random.random()  # Float between 0 and 1
0.54669095362942288
>>> random.gauss(mu=10, sigma=3)
7.7026739697957005
```

Other probability functions: beta, exponential, gamma, lognormal, Pareto, `randint`, `randrange`, `uniform`, Von Mises and Weibull.

```python
>>> a = [1, 2, 3, 4]; random.shuffle(a); a
[3, 1, 4, 2]
```

Other functions `choice`, `sample`, etc.
Random initialization with numpy

Initialize an array with random numbers between zero and one:

```python
>>> import numpy.random
>>> numpy.random.random((2,3))  # Tuple input!
array([[ 0.98872329,  0.73451282,  0.54337299],
       [ 0.69088015,  0.59413038,  0.71935909]])
```

Standard Gaussian (normal distribution):

```python
>>> numpy.random.randn(2, 3)  # Individual input arg.
array([[-0.19301411, -1.37092092, -0.1666896],
       [ 1.41485887,  2.24646526, -1.27417696]])
>>> N = numpy.random.standard_normal((2, 3))  # Tuple input!
>>> N = numpy.random.normal(0, 1, (2, 3))
```
Multiplications and divisions . . .

Numpy multiplication and divisions are confusing.

```python
>>> from numpy import *
>>> A = array([[1, 2], [3, 4]])

With `array` the “default” is elementwise:

```python
>>> A * A  # '*' is elementwise, Matlab: A.*A
array([[ 1,  4],
       [ 9, 16]])

>>> dot(A, A)  # dot() is matrix multiplication
array([[ 7, 10],
       [15, 22]])
```
... Multiplications and divisions

With matrix the default is matrix multiplication

```python
>>> mat(A) * mat(A)  # Here '*' is matrix multiplication
matrix([ [ 7, 10],
         [15, 22]])

>>> multiply(mat(A), mat(A))  # 'multiply' is elementwise multiplication
matrix([ [ 1, 4],
         [ 9, 16]])

>>> dot(mat(A), mat(A))  # dot() is matrix multiplication
matrix([ [ 7, 10],
         [15, 22]])

>>> mat(A) / mat(A)  # Division always elementwise
matrix([ [1, 1],
         [1, 1]])
```
Matrix inversion

“Ordinary” matrix inversion available as inv() in the linalg module of numpy

```python
>>> linalg.inv(mat([[2, 1], [1, 2]]))
matrix([[ 0.66666667, -0.33333333],
        [-0.33333333,  0.66666667]])
```

Pseudo-inverse linalg.pinv() for singular matrices:

```python
>>> linalg.pinv(mat([[2, 0], [0, 0]]))
matrix([[ 0.5,  0. ],
        [ 0. ,  0. ]])
```
Singular value decomposition

`svd()` function in the `linalg` module returns by default three argument:

```python
>>> from numpy.linalg import svd
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]))
```

gives a 2-by-2 matrix, a 2-vector with singular values and a 3-by-3 matrix:

```python
>>> U * diag(s) * V  # Gives not aligned error
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]),
                 full_matrices=False)
>>> U * diag(s) * V  # Now ok: V.shape == (2, 3)
```

Note V is transposed compared to Matlab!

If only the singular values are required use `compute_uv` argument:

```python
>>> s = svd(mat([[1, 0, 0], [0, 0, 0]]), compute_uv=False)
```
Non-negative matrix factorization

from numpy import mat, random, multiply

def nmf(M, components=5, iterations=5000):
    """Factorize non-negative matrix.""
    W = mat(random.rand(M.shape[0], components))
    H = mat(random.rand(components, M.shape[1]))
    for n in range(iterations):
        H = multiply(H, (W.T * M) / (W.T * W * H + 0.001))
        W = multiply(W, (M * H.T) / (W * (H * H.T) + 0.001))
        print "%d/%d" % (n, iterations)
    return (W, H)

Two matrices are returned.

Note ‘0.001’ needs to be set to some ‘appropriate’ for the dataset.
Scipy
# Scipy

Scipy (scientific python) has many submodules:

<table>
<thead>
<tr>
<th>Subpackage</th>
<th>Function examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>vq.keans, vq.vq, hierarchy.dendrogram</td>
<td>Clustering algorithms</td>
</tr>
<tr>
<td>fftpack</td>
<td>fft, ifft, fftfreq, convolve</td>
<td>Fast Fourier transform, etc.</td>
</tr>
<tr>
<td>io</td>
<td>loadmat</td>
<td>Input/output functions</td>
</tr>
<tr>
<td>optimize</td>
<td>fmin, fmin_cg, brent</td>
<td>Function optimization</td>
</tr>
<tr>
<td>signal</td>
<td>butter, welch</td>
<td>Signal processing</td>
</tr>
<tr>
<td>spatial</td>
<td>ConvexHull, Voronoi, distance.cityblock</td>
<td>Functions for spatial data</td>
</tr>
<tr>
<td>stats</td>
<td>nanmean, chi2, kendalltau</td>
<td>Statistical functions</td>
</tr>
</tbody>
</table>

...
Optimization with scipy.optimize

scipy.optimize contains functions for mathematical function optimization:

fmin() is Nelder-Mead simplex algorithm for function optimization without derivatives.

```python
>>> import scipy.optimize, math
>>> scipy.optimize.fmin(math.cos, [1])
Optimization terminated successfully.
   Current function value: -1.000000
   Iterations: 19
   Function evaluations: 38
array([ 3.14160156])
```

Other optimizers: fmin_cg(), leastsq(), ...
**scipy optimization**

Custom multidimensional function (the *Rosenbrock function* aka banana function):

```python
def rosenbrock((x, y)):
    a, b = 1, 100
    return (a - x) ** 2 + b * (y - x ** 2) ** 2
```

Minimum of the function is at (1, 1).

Optimize with Scipy and its general `minimize` function:

```python
>>> from scipy.optimize import minimize
>>> result = minimize(rosenbrock, x0=(0, 0))
>>> result.x
array([ 0.99999561,  0.99999125])
```

Here the gradient is estimated numerically.
Not quite there yet, trying Nelder-Mead method with many iterations:

```python
>>> minimize(rosenbrock, x0=(0, 0), method='Nelder-Mead',
            tol=0, options={'maxiter': 1000})

status: 0
nfev: 324
success: True
fun: 0.0
x: array([ 1.,  1.])
message: 'Optimization terminated successfully.'
nit: 168
```
Statistics with `scipy.stats`

Showing the $\chi^2$ probability density function with different degrees of freedom:

```python
from pylab import *
from scipy.stats import chi2
x = linspace(0.1, 25, 200)
for dof in [1, 2, 3, 5, 10, 50]:
    plot(x, chi2.pdf(x, dof))
```

Other functions such as missing data mean:

```python
>>> x = [1, nan, 3, 4, 5, 6, 7, 8, nan, 10]
>>> scipy.stats.stats.nanmean(x)
5.5
```
Random initialization with `scipy.stats`

10 discrete distributions and 81 continuous distributions

```python
>>> scipy.stats.uniform.rvs(size=(2, 3))  # Uniform array([[ 0.23273417,  0.17636535,  0.88709937],
           [ 0.07573364,  0.04084195,  0.45961136]])

>>> scipy.stats.norm.rvs(size=(2, 3))   # Gaussian array([[ 0.89339055, -0.05093851,  0.12449392],
            [ 0.49639535, -1.39487053,  0.38580828]])
```
Kernel density estimation from `scipy.stats`...

```python
from pylab import *; import scipy.stats

x0 = array([1.1, 1.2, 2, 6, 6, 5, 6.3])
x = arange(-3, 10, 0.1)
plot(x, scipy.stats.gaussian_kde(x0).evaluate(x))
```

Bandwidth computed via ‘scott’s factor’...?

Estimation in higher dimensions possible
... Kernel density estimation

```python
>>> class kde(scipy.stats.gaussian_kde):
...     def covariance_factor(dummy): return 0.05
...

>>> plot(x, kde(x0).evaluate(x))
```

Defining a new class “kde” from the gaussian_kde class from the scipy.stats.gaussian_kde module with
Fourier transformation

```python
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

What is `x`?
Fourier transformation ...

```python
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

What is `x`?

```python
>>> x[:20]
array([1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1])
```

Zoom in on the first 51 data points of the square wave signal

With the `pylab` module:

```python
>>> plot(x[:50], linewidth=2)
```

```python
>>> axis((0, 50, -1, 2))
```
Forward Fourier transformation with the \texttt{fft()} from \texttt{scipy.fftpack} module (also loaded with \texttt{numpy.fft}, \texttt{pylab}):

\begin{verbatim}
>>> abs(fft(x))
\end{verbatim}

Back to “time” domain with inverse Fourier transformation:

\begin{verbatim}
>>> from scipy.fftpack import *
>>> abs(ifft(fft(x)))[:6].round()
array([ 1., 1., 1., 1., 0., 0.])
\end{verbatim}

Other: \texttt{fft2()}, \texttt{fftn()}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fourier_transformation.png}
\end{figure}
sklearn aka scikit-learn
# Machine learning in Python

<table>
<thead>
<tr>
<th>Name</th>
<th>KLoC</th>
<th>GS-cites</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SciPy.linalg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statsmodels</td>
<td>92</td>
<td>27</td>
<td>(Seabold and Perktold, 2010)</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>440</td>
<td>1.860</td>
<td>(Pedregosa et al., 2011)</td>
</tr>
<tr>
<td>PyMVPA</td>
<td>136</td>
<td>147 + 60</td>
<td>(Hanke et al., 2009a; Hanke et al., 2009b)</td>
</tr>
<tr>
<td>Orange</td>
<td>286</td>
<td>75</td>
<td>(Demšar et al., 2013)</td>
</tr>
<tr>
<td>Mlpy</td>
<td>75</td>
<td>8</td>
<td>(Albanese et al., 2012)</td>
</tr>
<tr>
<td>MDP</td>
<td>31</td>
<td>58</td>
<td>(Zito et al., 2008)</td>
</tr>
<tr>
<td>PyBrain</td>
<td>36</td>
<td>147</td>
<td>(Schaul et al., 2010)</td>
</tr>
<tr>
<td>Pylearn2</td>
<td></td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td></td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Gensim</td>
<td>9</td>
<td>160</td>
<td>(Řehůřek and Sojka, 2010)</td>
</tr>
<tr>
<td>NLTK</td>
<td>215</td>
<td>1.200</td>
<td>(Bird et al., 2009)</td>
</tr>
<tr>
<td>PyPR</td>
<td>?</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>Caffe</td>
<td>?</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Statistics not completely updated)
Scikit-learn/sklearn

In sklearn each algorithm is implemented as an object.

Here, e.g., non-negative matrix factorization (NMF) with data in a numpy.array called datamatrix:

```python
from sklearn.decomposition import NMF
decomposer = NMF(n_components=2)  # Instancing an algorithm
decomposer.fit(datamatrix)  # Parameter estimation
```

or K-means clustering algorithm:

```python
from sklearn.cluster import KMeans
clusterer = KMeans(n_clusters=2)
clusterer.fit(datamatrix)
```
sklearn object methods

Sklearn algorithm objects shares methods, such as:

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_params</td>
<td>—</td>
<td>Get parameter</td>
</tr>
<tr>
<td>set_params</td>
<td>Parameters</td>
<td>Set parameters</td>
</tr>
<tr>
<td>decision_function</td>
<td>X</td>
<td>Estimate model parameters</td>
</tr>
<tr>
<td>fit</td>
<td>X, y</td>
<td>Performs clustering and return cluster labels</td>
</tr>
<tr>
<td>fit_predict</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>fit_transform</td>
<td>X, (y)</td>
<td>Fit and transform</td>
</tr>
<tr>
<td>inverse_transform</td>
<td>Y</td>
<td>Opposite operation of transform</td>
</tr>
<tr>
<td>predict</td>
<td>X</td>
<td>Estimate output</td>
</tr>
<tr>
<td>predict_proba</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>score</td>
<td>X, y</td>
<td>Coefficient of determination</td>
</tr>
<tr>
<td>transform</td>
<td>X</td>
<td>Transform data, e.g., through dimensionality reduction</td>
</tr>
</tbody>
</table>

Estimated parameters and estimation “metadata” is available in attributes with trailing underscore, e.g., “components_”. 
sklearn object methods

The common interface means that machine learning classifiers can be used interchangeably, see Classifier comparison SciKit-learn example.

From an example by Gael Varoquaux and Andreas Muller.
sklearn example

Non-negative matrix factorization with a data matrix called `feature_matrix`:

```python
from sklearn.decomposition import NMF
decomposer = NMF(n_components=2)
decomposer.fit(feature_matrix)
transformed = decomposer.transform(feature_matrix)
```

Sizes of data matrix and resulting factorization matrices:

```python
>>> feature_matrix.shape
(94, 256)
>>> transformed.shape
(94, 2)
>>> decomposer.components_.shape
(2, 256)
```
Pandas
Pandas dataframe

A =

<table>
<thead>
<tr>
<th>Index</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>9</td>
<td>ok</td>
</tr>
</tbody>
</table>

Table represented in a Pandas DataFrame:

```python
>>> import pandas as pd
>>> A = pd.DataFrame([[4, 5, 'yes'], [6.5, 7, 'no'], [8, 9, 'ok']], index=[2, 3, 6], columns=['a', 'b', 'c'])
>>> A
   a   b  c
 2 4.0  5  yes
 3 6.5  7   no
 6 8.0  9   ok
```
Pandas column indexing

```python
>>> A.c
2  yes
3  no
6  ok
Name: c, dtype: object

>>> A['c']
2  yes
3  no
6  ok
Name: c, dtype: object

>>> A.ix[:, 'c']
2  yes
3  no
6  ok
Name: c, dtype: object
```
Pandas row indexing

```python
>>> A.ix[6, :]
   a   b   c
0  8   9  ok
Name: 6, dtype: object
```

```python
>>> A.iloc[2, :]
   a   b   c
0  8   9  ok
Name: 6, dtype: object
```

And an element:

```python
>>> A.ix[6, 'b']
9
```
Conditional indexing

```python
>>> A
     a  b  c
2  4.0 5  yes
3  6.5 7   no
6  8.0 9   ok

>>> A.ix[(A.a > 7.0) | (A.c == 'yes'), ['b', 'c']]
     b  c
2  5  yes
6  9   ok
```
Pandas join and merge

In two Pandas DataFrames:

```python
>>> import pandas as pd

>>> A = pd.DataFrame([[4, 5], [6, 7]], index=[1, 2], columns=['a', 'b'])

>>> B = pd.DataFrame([[8, 9], [10, 11]], index=[1, 3], columns=['a', 'c'])
```

In two Pandas DataFrames:

$$A = \begin{array}{c|cc}
\text{Index} & a & b \\
1 & 4 & 5 \\
2 & 6 & 7 \\
\end{array}$$

$$B = \begin{array}{c|cc}
\text{Index} & a & c \\
1 & 8 & 9 \\
3 & 10 & 11 \\
\end{array}$$
Pandas concat and merge

```python
>>> pd.concat((A, B))  # implicit outer join
   a   b   c
1  4   5 NaN
2  6   7 NaN
1  8   NaN 9
3 10   NaN 11
```

```python
>>> pd.concat((A, B), join='inner')
   a
1  4
2  6
1  8
3 10
```
Pandas concat and merge

```python
gf A.merge(B, how='inner', left_index=True, right_index=True)
a_x  b  a_y  c
1   4  5   8   9

gf A.merge(B, how='outer', left_index=True, right_index=True)
a_x  b  a_y  c
1   4  5   8   9
2   6  7   NaN NaN
3   NaN NaN 10  11

gf A.merge(B, how='left', left_index=True, right_index=True)
a_x  b  a_y  c
1   4  5   8   9
2   6  7   NaN NaN
```
## Descriptive statistics

```python
>>> A.describe()  # see also .sum, .std, ...
    a    b
count 3.000000 3
mean  6.166667 7
std   2.020726 2
min   4.000000 5
25%   5.250000 6
50%   6.500000 7
75%   7.250000 8
max   8.000000 9
```

Note the result is only for numerical columns
Pandas input and output

read_csv — Comma-separated values file, works for URLs

read_sql — Read from SQL database

read_excel — Microsoft Excel

and a number of other formats.

Also note that the db.py package can interface between a SQL database and Pandas.
Other Pandas datatypes

Pandas Series is an annotated vector:

```python
>>> pd.Series([1, 2, 4], index=['a', 'b', 'f'])
a 1
b 2
f 4
dtype: int64
```

Pandas Panel is a three-dimensional structure:

```python
>>> pd.Panel([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: 0 to 1
Major_axis axis: 0 to 1
Minor_axis axis: 0 to 1
```
### Pandas conversion to Numpy

```python
>>> A
     a   b   c
   2 4.0  5  yes
   3 6.5  7   no
   6 8.0  9    ok

>>> A.values
array([[4.0, 5,  'yes'],
       [6.5, 7,  'no'],
       [8.0, 9,  'ok']], dtype=object)

>>> A.ix[:, :2].values
array([[ 4. ,  5. ],
       [ 6.5,  7. ],
       [ 8. ,  9. ]])
```
Cython
Calling C et al.

You can call C, C++ and Fortran functions from Python:

Either with “manual” wrapping

or by using a automated wrapper: SWIG, Boost.Python, CFFI.

or by direct calling existing libraries via ctypes

You can make C-programs in Python with Cython or Pyrex and calling compiled modules from Python
Why calling C et al.?  

Because you already have code in that language.

Because ordinary Python is not fast enough.
Why calling C et al.? 

Why calling C et al.?

- Because you already have code in that language.

- Because ordinary Python is not fast enough. **Cython useful here!**
Cython

Write a Python file (possibly with extended Cython syntax for static types), compile to C and compile the C.

Cython is a fork of Pyrex.

Simplest example with compilation of a python file helloworld.py, containing print("Hello, World"):

$ cython --embed helloworld.py
$ gcc -I/usr/include/python2.7 -o helloworld helloworld.c -lpython2.7
$ ./helloworld

More: You can compile to a module instead (callable from Python); you can include static types in the Python code to make it faster (often these files have the extension *.pyx).
Calling Cython module from Python

hello.pyx Python file with

```python
def world():
    return "Hello, World"
```

Compile to C and compile to module (here for Linux):

```
$ cython hello.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \ -I/usr/include/python2.7 -o hello.so hello.c -lpython2.7
```

Back in Python use the hello module:

```python
>>> import hello
>>> hello.world()    # Call the function in the module
'Hello, World'
```
Other compilation methods

For simple Cython modules compiling automagically:

```python
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x7f7dc39f20d0>)
>>> import hello
>>> hello.world()
'Hello, World'
```
Other compilation methods

For simple Cython modules compiling automagically:

```python
>>> import pyximport
>>> pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x7f7dc39f20d0>)
>>> import hello
>>> hello.world()
'Hello, World'
```

Otherwise construct a setup.py file with:

```python
from distutils.core import setup
from Cython.Build import cythonize

setup(name='Hello', ext_modules=cythonize("hello.pyx"))
```

and compile with

```
$ python setup.py build_ext --inplace
```
Optional types ...

slow.py with plain python

```python
def count_ascendings(alist):
    count = 0
    for first, second in zip(alist[:-1], alist[1:]):
        if first < second:
            count += 1
    return count
```

fast.pyx with integer type (note “cdef int count”!):

```python
def count_ascendings(alist):
    cdef int count = 0
    for first, second in zip(alist[:-1], alist[1:]):
        if first < second:
            count += 1
    return count
```
... Optional types

Profiling the plain Python and the Cython version:

```python
>>> import timeit
>>> import pyximport; pyximport.install()

>>> timeit.timeit("slow.count_ascendings(range(1000))", setup="import slow", number=10000)
1.358449291229248

>>> timeit.timeit("fast.count_ascendings(range(1000))", setup="import fast", number=10000)
0.8554580211639404
```

The Cython version without static types yield a timing between the plain Python and the statically typed Cython (1.06 seconds here).
Six advices from Hans Petter Langtangen

Section *Optimization of Python code* (Langtangen, 2005, p. 426+)

Avoid loops, use NumPy (see also my blog)

Avoid prefix in often called functions, i.e., `sin` instead of `math.sin`

Plain functions run faster than class methods

Don’t use NumPy for scalar arguments

Use `xrange` instead of `range` (in Python < 3)

`if-else` is faster than `try-except` (sometimes!)
More information

http://www.scipy.org/NumPy_for_Matlab_Users

MATLAB commands in numerical Python (NumPy), Vidar Bronken Gundersen, mathesaurus.sf.net

Guide to NumPy (Oliphant, 2006)

Videolectures.net: John D. Hunter overview of Matplotlib
http://videolectures.net/mloss08_hunter_mat/
Emerging Python modules for big data

blaze and dask

castra

pySpark


References


