



**OPENCOURSEWARE  
ADVANCED PROGRAMMING  
STATISTICS FOR DATA SCIENCE  
Ricardo Aler**

# NumPy

# What is NumPy?

- It is a Python module/library  
import numpy as np
- It is useful for computing with numeric vectors, matrices, and multi-dimensional arrays in general
- Standard Python lists could be used, but +, -, \*, /, etc. cannot be used with numeric lists:

```
>>> 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]
```

# Creating NumPy arrays

- One-dimension vectors:

```
# From lists  
  
>>> a = np.array([1,3,5,7,9])  
>>> b = np.array([3,5,6,7,9])  
>>> c = a + b  
>>> print c  
[4, 8, 11, 14, 18]  
  
>>> type(c)  
(<type 'numpy.ndarray'>)  
  
>>> c.shape  
(5, )
```

# Creating NumPy arrays

- Matrices:

```
>>> # convert a list to an array
>>> a = np.array([[1, 2, 3], [3, 6, 9], [2, 4, 6]])
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
>>> a.shape
(3, 3)
```

# Shape of NumPy arrays

- 1-dimensional arrays
- 2-dimensional arrays  
(matrices)
- 3-dimensional arrays

# Shape of NumPy arrays

- Important: a 1-dimensional vector is different from a matrix with 1 row (or 1 column)

```
vector_1d = np.array([1,2,3,4])
print(vector_1d)
print(vector_1d.shape)
```

```
[1 2 3 4]
(4,)
```

```
matrix_1row = np.array([[1,2,3,4]])
print(matrix_1row)
print(matrix_1row.shape)
```

```
[[1 2 3 4]]
(1, 4)
```

```
matrix_1col = np.array([[1],[2],[3],[4]])
print(matrix_1col)
print(matrix_1col.shape)
```

```
[[1]
 [2]
 [3]
 [4]]
(4, 1)
```

# Types of NumPy arrays

- All elements in a NumPy array **must belong to the same type** (dtype)
- dtypes are inferred automatically
- But dtypes can also be stated explicitly
- Available dtypes

```
a = np.array([1,2,3])  
print(a.dtype)
```

int32

```
b = np.array([1,2,3.141516])  
print(b.dtype)
```

float64

```
c = np.array([1,2,3], dtype=np.float32)  
print(c.dtype)
```

float32

np.sctypes

```
{'int': [numpy.int8, numpy.int16, numpy.int32, numpy.int64],  
'uint': [numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64],  
'float': [numpy.float16, numpy.float32, numpy.float64],  
'complex': [numpy.complex64, numpy.complex128],  
'others': [bool, object, bytes, str, numpy.void]}
```

# Beware! (NumPy types)

- A NumPy array belongs to a single type

```
>>> d = np.arange(5)
>>> print(d.dtype)
>>> print(d)
int32
[0 1 2 3 4]

# We try to assign a real number to an integer array
# but the value is converted to integer
>>> d[1] = 9.7
print(d)
[0 9 2 3 4]
```

# Creating NumPy vectors with functions

- `arange(x)` is similar to `list(range(x))`, but it generates NumPy vectors, rather than Python vectors

```
>>> x = np.arange(0, 10, 1) # arguments: start, stop, step
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x.dtype
dtype('int32')

# Real-valued (float) vectors can also be created
>>> d = np.arange(5, dtype=numpy.float)
>>> print(d)
[ 0.  1.  2.  3.  4.]

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

# Creating NumPy vectors with functions

- `linspace` is useful for generating n real-valued vectors within an interval

```
>>> np.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.
])
```

# Creating NumPy vectors with functions

- `diag`: diagonal matrices
- arrays of zeros or ones

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

# An identity matrix
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
# A vector of zeros
>>> b = np.zeros(5)
>>> print(b)
[ 0.  0.  0.  0.  0.]

# A matrix of ones
>>> c = np.ones((3,3))
>>> print(c)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

# Creating NumPy vectors with functions

- Generating **random real numbers** from a **uniform** distribution in [0,1)

```
>>> np.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]])
```

- `randn(a,b)` returns a  $a \times b$  matrix with **random real numbers** from the standard **normal** distribution
- `randint(low, high, size=(a,b))` returns **uniform random integers** in the low-high interval

# Creating NumPy matrices from vectors

- Using reshape

```
In []: a = np.arange(10)
In []: a
Out[]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In []: a.shape
Out[]: (10,)
```

```
In []: a = a.reshape((2,5))
In []: a
Out[]:
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
In []: a.shape
Out[]: (2, 5)
```

# Using NumPy matrices: slicing (indexing with slices)

```
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]

# this is just like a list of lists
>>> print(a[0])
[1 2 3]

# arrays can be given comma separated indices
>>> print(a[1, 2])
9

# and slices
>>> print(a[1, 1:3])
[6 9]
>>> print(a[:,1])
[2 6 4]
```

0	1	2	
[1 2 3]			0
[3 6 9]			1
[2 4 6]			2
[1 2 3]			0
[3 6 9]			1
[2 4 6]			2
[1 2 3]			0
[3 6 9]			1
[2 4 6]			2
[1 2 3]			0
[3 6 9]			1
[2 4 6]			2

# Using NumPy matrices: indexing with booleans

```
In [99]: a = np.array([[0, np.nan], [np.nan, 3], [4, np.nan]])
```

```
In [100]: a
```

Out[100]:

```
array([[ 0., nan],
       [nan, 3.],
       [ 4., nan]])
```

```
In [101]: a<4
```

# This array of booleans shows where a<4 is true

Out[101]:

```
array([[ True, False],
       [False, True],
       [False, False]])
```

# Here, we can see what elements in the array are < 4

```
In [103]: a[a<4]
```

Out[103]: array([0., 3.])

```
# This array of booleans shows where a contains nan
```

```
In [104]: np.isnan(a)
```

Out[104]:

```
array([[False, True],
       [ True, False],
       [False, True]])
```

# Here we transform nan's into 0

```
In [106]: a[np.isnan(a)] = 0
```

```
In [107]: a
```

Out[107]:

```
array([[0., 0.],
       [0., 3.],
       [4., 0.]])
```

# Using NumPy matrices: modification (setting)

```
# We can modify a single element in the matrix
>>> a[1, 2] = 7
>>> print(a)
[[1 2 3]
 [3 6 7]
 [2 4 6]]


# We can also modify whole columns
>>> a[:, 0] = [0, 9, 8]
>>> print(a)
[[0 2 3]
 [9 6 7]
 [8 4 6]]


# And whole rows
>>> a[0, :] = [1, 1, 1]
>>> print(a)
[[1 1 1]
 [9 6 7]
 [8 4 6]]
```

# Using NumPy matrices: modification (setting)

- Important, for arrays,  $a = 0$  is not the same as  $a[:] = 0$  (or  $a[0:] = 0$ )

```
In [18]: a = np.array(np.arange(10))
```

```
In [19]: a
```

```
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [20]: a[:] = 0
```

```
In [21]: a
```

```
Out[21]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [22]: a = 0
```

```
In [23]: a
```

```
Out[23]: 0
```

# Using NumPy matrices: views (references)

- **b=a** does not copy b's content into a. Rather, it creates a reference (view). This is standard Python behavior.

```
In [10]: import numpy as np
```

```
In [11]: a = np.array(np.arange(10))
```

```
In [12]: a
```

```
Out[12]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [13]: b = a
```

```
In [14]: b[0] = 1000
```

```
In [15]: a
```

```
Out[15]: array([1000, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [16]: b
```

```
Out[16]: array([1000, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# Using NumPy matrices: views (references)

- Beware, indexing also creates a view (reference)!

```
In [26]: a = np.array(np.arange(10))
```

```
In [27]: a
```

```
Out[27]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [28]: # This is a view into a
```

```
In [29]: b = a[2:4]
```

```
In [30]: b
```

```
Out[30]: array([2, 3])
```

```
In [31]: # If we modify the view, we modify the original variable
```

```
In [32]: b[:] = -1
```

```
In [33]: b
```

```
Out[33]: array([-1, -1])
```

```
In [34]: a
```

```
# a is modified aswell!!
```

```
Out[34]: array([ 0, 1, -1, -1, 4, 5, 6, 7, 8, 9])
```

```
# We can print ownData to distinguish views from copies
```

```
In [35]: a.flags.ownData
```

```
Out[35]: True
```

```
In [36]: b.flags.ownData
```

```
Out[36]: False
```

# Using NumPy matrices: views (references)

- We can use `copy()` to actually copy the object

```
In [58]: a = np.array(np.arange(10))
```

```
In [59]: a
```

```
Out[59]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [60]: b = a[2:4].copy()
```

```
In [61]: b[:] = -1
```

```
In [62]: a
```

```
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [63]: b
```

```
Out[63]: array([-1, -1])
```

```
In [64]: a.flags.owndata
```

```
Out[64]: True
```

```
In [65]: b.flags.owndata
```

```
Out[65]: True
```

# Exercise

1. Create a 3x5 matrix of normal random numbers named *my\_matrix*. Print it.
2. Now, we are going to introduce some NA's into the matrix (in Python NA's are represented as numpy.nan = not a number)
  1.  $x = [0,2]$
  2.  $y = [3,1]$
  3. We are going to use x and y to introduce NA's into *my\_matrix*, at positions (0,3) and (2,1) by doing this: *my\_matrix[x,y] = np.nan*.
  4. Print the result.
3. Now, use boolean indexing and *isna()* for replacing all NA's by zero, and print the result

# Solution

```
In []: my_matrix = np.random.randn(3,5)
In []: my_matrix
array([-1.48413505, -0.23568385, -1.22030818, -0.81259558,  1.68216758],
      [-0.24242369, -2.51793289,  1.70739294,  1.30946991, -1.74124409],
      [-0.17144277, -1.42001248, -0.23261268,  1.08373964,  1.41257598])
```

```
In []: x = [0,2]
In []: y = [3,1]
```

```
In []: my_matrix
array([-1.48413505, -0.23568385, -1.22030818,      nan,  1.68216758],
      [-0.24242369, -2.51793289,  1.70739294,  1.30946991, -1.74124409],
      [-0.17144277,      nan, -0.23261268,  1.08373964,  1.41257598]))
```

```
In []: my_matrix[np.isnan(my_matrix)] = 0
In []: print(my_matrix)
[[-1.48413505 -0.23568385 -1.22030818  0.      1.68216758]
 [-0.24242369 -2.51793289  1.70739294  1.30946991 -1.74124409]
 [-0.17144277  0.      -0.23261268  1.08373964  1.41257598]]
```

# Universal functions

- They are functions that operate **element-wise** on one or more arrays

```
In [69]: a = np.arange(10)
```

```
In [70]: a
```

```
Out[70]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Universal function sqrt
```

```
In [71]: a = np.sqrt(a)
```

```
In [72]: a
```

```
Out[72]:
```

```
array([0. , 1. , 1.41421356, 1.73205081, 2. , 2.23606798, 2.44948974, 2.64575131, 2.82842712, 3. ])
```

```
In [73]: b = np.arange(10)*8.7
```

```
In [75]: c = a + b
```

```
In [76]: c
```

```
Out[76]:
```

```
array([ 0. , 9.7 , 18.81421356, 27.83205081, 36.8 , 45.73606798, 54.64948974, 63.54575131, 72.42842712, 81.3 ])
```

# Available universal functions

<https://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs>

- comparison: `<, <=, ==, !=, >=, >`
- arithmetic: `+, -, *, /, reciprocal, square`
- exponential: `exp, expm1, exp2, log, log10, log1p, log2, power, sqrt`
- trigonometric: `sin, cos, tan, acsin, arccos, atctan`
- hyperbolic: `sinh, cosh, tanh, acsinh, arccosh, atctanh`
- 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`

# Reduction functions

- Reduction functions allow to transform an array to a single number:
  - sum, mean, ...

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

```
In [11]: a
```

```
Out[11]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [12]: a.sum()
```

```
Out[12]: 45
```

```
In [13]: a.mean()
```

```
Out[13]: 4.5
```

```
In [14]: a = np.array([[0, 1, 2, 3],
```

```
[4, 5, 6, 7],
```

```
[8, 9, 10, 11]])
```

```
In [15]: a
```

```
Out[15]:
```

```
array([[0, 1, 2, 3, 4],
```

```
[5, 6, 7, 8, 9],
```

```
[10, 11, 12, 13, 14]])
```

```
In [16]: a.sum()
```

```
Out[16]: 105
```

# Reduction functions

- In general, reduction functions transform arrays into arrays of smaller dimensionality by reducing along an axis
- A 1-dimensional array has one axis,  $\text{axis}=0$
- A 2-dimensional array (matrix) has two axis,  $\text{axis}=0$  (rows),  $\text{axis}=1$ (columns)
- We could, for instance, sum the columns of a matrix (sum along the 0-axis)

# Reduction functions

```
# Sum along axis 0 / rows (sum all rows elements in a column)
```

```
In [19]: a.sum(axis=0)
```

```
Out[19]: array([15, 18, 21, 24, 27])
```

```
# Sum along axis 1 / columns (sum all column elements in a row)
```

```
In [20]: a.sum(axis=1)
```

```
Out[20]: array([ 6, 22, 38])
```

```
# Sum along axis 0 / rows
```

```
In [21]: np.sum(a, axis=0)
```

```
Out[21]: array([12, 15, 18, 21])
```

```
# Sum along axis 1
```

```
In [22]: np.sum(a, axis=1)
```

```
Out[22]: array([ 6, 22, 38])
```

# Reduction functions

- Other reduction functions: max, min, mean,  
...

# Broadcasting

- Why does this work?

```
In []: a = np.array([0, 1, 2, 3, 4, 5])
```

```
In []: b = np.array([10])
```

```
In []: a
```

```
Out[]: array([0, 1, 2, 3, 4, 5])
```

```
In []: b
```

```
Out[]: array([10])
```

```
In [48]: c = a+b
```

```
In [49]: c
```

```
Out[49]: array([10, 11, 12, 13, 14, 15])
```

- Broadcasting allows to have operations between arrays with different sizes

# Broadcasting

- Broadcast requires all dimensions to be 1 or equal.

```
In []: a = np.array([[0, 1], [2, 3], [4, 5]])
```

```
In []: b = np.array([[10], [20], [30]])
```

```
In []: a
```

```
Out[]:
```

```
array([[0, 1],  
       [2, 3],  
       [4, 5]])
```

```
In []: b
```

```
Out[]:
```

```
array([[10],  
       [20],  
       [30]])
```

```
In [61]: c = a + b
```

```
In [62]: c
```

```
Out[62]:
```

```
array([[10, 11],  
       [22, 23],  
       [34, 35]])
```

The diagram shows three 3x2 matrices labeled a, b, and c. Matrix a has values 0, 1, 2, 3, 4, 5. Matrix b has values 10, 20, 30. Matrix c is the result of the addition, having values 10, 11, 22, 23, 34, 35. A plus sign between a and b indicates the operation, and an equals sign followed by c indicates the result.

0	1	+	10	10	=	10	11
2	3		20	20		22	23
4	5		30	30		34	35

a                    b                    c

a.shape = (3,2)

b.shape = (3,1) => (3,2)

# Exercise: normalization (scaling features to a range)

1. Create a 3x5 matrix of normal random numbers named *my\_matrix*. Print it.
2. Now, use reduction functions (*max* and *min*) to compute two vectors *maxima* and *minima* with the maximum and minimum values (respectively) of the columns of *my\_matrix*
3. Now, compute new matrix *normalized\_matrix*, so that columns of *my\_matrix* become normalized between zero and one.
  1. Definition of normalization:  $x'_{ij} = (x_{ij} - \min(x_j)) / (\max(x_j) - \min(x_j))$
4. Check that all values of *normalized\_matrix* are  $\geq 0$ , and  $\leq 1$
5. Finally, compute *standarized\_matrix* (mean removal/variance scaling)
  1. Def of standarization:  $x'_{ij} = (x_{ij} - \text{mean}(x_j)) / \text{std}(x_j)$
6. Verify that the mean of all columns is zero, and the standard deviation is 1 (approximately)

```
In [201]: my_matrix = np.random.randn(3,5)
In [202]: maxima = my_matrix.max(axis=0)
In [203]: print(maxima)
[ 2.19405637 0.54857877 -0.77583136 -0.75875882 1.22463799]

In [204]: minima = my_matrix.min(axis=0)
In [205]: print(minima)
[ 1.03488226 -0.82966138 -1.55133288 -1.46959842 -0.76071212]

In [206]: normalized_matrix = (my_matrix - minima) / (maxima-minima)
In [207]: print(normalized_matrix)
[[0.28223942 0. 0.37006178 1. 1. ]
 [1. 1. 0. 0. 0. ]
 [0. 0.14074337 1. 0.64842885 0.40670394]]

In [208]: normalized_matrix >= 0
Out[208]:
array([[ True, True, True, True, True],
 [ True, True, True, True, True],
 [ True, True, True, True, True]])

In [209]: normalized_matrix <= 1
Out[209]:
array([[ True, True, True, True, True],
 [ True, True, True, True, True],
 [ True, True, True, True, True]])

In [210]: standarized_matrix = (my_matrix - my_matrix.mean(axis=0))/(my_matrix.std(axis=0))
In [211]: print(standarized_matrix)
[[-0.34486633 -0.86032467 -0.20983943 1.08769345 1.29343692]
 [ 1.36020436 1.40221228 -1.10626795 -1.32659331 -1.14196153]
 [-1.01533803 -0.54188761 1.31610738 0.23889987 -0.15147539]]

In [212]: standarized_matrix.mean(axis=0)
Out[212]:
array([ 2.22044605e-16, 3.70074342e-17, 3.70074342e-16, -3.88578059e-16,
 4.62592927e-17])

In [213]: standarized_matrix.std(axis=0)
Out[213]: array([1., 1., 1., 1., 1.])
```

# Loading and saving numpy arrays to files

- Reading files: `np.genfromtxt("BodyTemperature.txt", skip_header=True)`
  - `np.loadtxt` is faster, but allows for less user control (header, handling NA's)
- Writing to text files: `np.savetxt(filename, data)`
- For pickle (binary format, faster):
  - `np.save(filename, data)`
  - `my_array = np.load(filename, data)`

```
# Read textdatafile, ignore the header
```

```
# The header is Gender Age HeartRate Temperature
```

```
In [313]: data = np.genfromtxt("BodyTemperature.txt" ,  
skip_header=True )
```

```
# Any nan?
```

```
In [315]: np.any(np.isnan(data))
```

```
Out[315]: False
```

```
# Number of males
```

```
In [317]: np.sum(data[:,0] == 0)
```

```
Out[317]: 49
```

```
# Number of females
```

```
In [319]: np.sum(data[:,0] == 1)
```

```
Out[319]: 51
```

```
# Ignoring gender, the remaining columns are averages for:
```

```
# Age HeartRate Temperature
```

```
In [322]: data.mean(axis=0)[1:]
```

```
Out[322]: array([37.62, 73.66, 98.33])
```

```
In [323]: males = data[data[:,0] == 0]
```

```
In [324]: females = data[data[:,0] == 1]
```

```
In [325]: males_mean = males.mean(axis=0)[1:]
```

```
In [326]: males_max = males.max(axis=0)[1:]
```

```
In [327]: males_min = males.min(axis=0)[1:]
```

```
In [328]: females_mean = females.mean(axis=0)[1:]
```

```
In [329]: females_max = females.max(axis=0)[1:]
```

```
In [330]: females_min = females.min(axis=0)[1:]
```

```
In [331]: table = np.array([males_mean, males_max,  
males_min, females_mean, females_max, females_min])
```

```
In [332]: table
```

```
Out[332]:
```

```
array([[ 37.81632653, 73.91836735, 98.19795918],
```

```
[ 50. , 87. , 101.3 ],
```

```
[ 22. , 61. , 96.2 ],
```

```
[ 37.43137255, 73.41176471, 98.45686275],
```

```
[ 49. , 87. , 100.8 ],
```

```
[ 21. , 67. , 96.8 ]])
```

```
In [333]: np.savetxt("BD_results.txt", table)
```

# Pandas

# PANDAS

- Pandas is the Python library to work with dataframes (similar to R data.frames)  
`import pandas as pd`
- An advantage of Pandas over numpy is that all elements in a numpy array must belong to the same type, while Pandas allows to have different columns with different types (integers, reals, strings, ...)

# PANDAS data structures

- Pandas contains two data structures:
  - Series: a series is like a vector, but with an index
  - Dataframes: it is similar to R dataframes (a matrix with column names. Each column may belong to different data types: integer, real numbers, strings, ...)
    - A dataframe is made of:
      - index
      - column names
      - values

# Example of series

```
# Using the default index 0, 1, ...
```

```
s = pd.Series(np.random.randn(5))
```

```
In [224]: s
```

```
Out[224]:
```

```
0    1.037685  
1    0.403077  
2   -1.814123  
3   -0.005181  
4    1.692980  
dtype: float64
```

```
# We can get the values of a series as a numpy array
```

```
In [225]: s.values
```

```
Out[225]: array([ 1.03768522, 0.40307685, -1.81412276, -0.005181 ,  
1.69298038])
```

```
In [226]: s.index
```

```
Out[226]: RangeIndex(start=0, stop=5, step=1)
```

```
# Using a custom index
```

```
In [228]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [229]: s
```

```
Out[229]:
```

```
a    0.226183  
b   -0.564569  
c   -1.058691  
d    0.970553  
e   -0.857780  
dtype: float64
```

```
In [230]: s.values
```

```
Out[230]: array([ 0.22618273, -0.564569 , -1.05869052, 0.97055338, -  
0.85777957])
```

```
In [231]: s.index
```

```
Out[231]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

Note: although indices can be useful in some cases (time series, ...), this tutorial will not focus on them

# Reading files as dataframes

```
In [121]: import pandas as pd
```

```
# Read file in csv format into a Pandas dataframe
```

```
In [122]: flights = pd.read_csv("flights.csv")
```

```
In [124]: flights.shape
```

```
Out[124]: (336776, 19)
```

```
# head: print the first rows
```

```
In [22]: flights.head(  
...: )
```

```
Out[22]:
```

	year	month	day	...	hour	minute	time_hour
0	2013	1	1	...	5	15	2013-01-01 05:00:00
1	2013	1	1	...	5	29	2013-01-01 05:00:00
2	2013	1	1	...	5	40	2013-01-01 05:00:00
3	2013	1	1	...	5	45	2013-01-01 05:00:00
4	2013	1	1	...	6	0	2013-01-01 06:00:00

```
[5 rows x 19 columns]
```

```
# Getting the names of the columns
```

```
In [130]: list(flights.columns)
```

```
Out[130]:
```

```
['year', 'month', 'day', 'dep_time',  
'sched_dep_time', 'dep_delay',  
'arr_time', 'sched_arr_time', 'arr_delay',  
'carrier', 'flight', 'tailnum', 'origin',  
'dest', 'air_time', 'distance', 'hour', 'minute', 'time_hour']
```

Index (by default 0, 1, 2, ...)

```
In []: flights.index
```

```
Out[]: RangeIndex(start=0, stop=336776, step=1)
```

# Describing the dataframe

```
In [23]: flights.describe()
```

```
Out[23]:
```

	year	month	...	hour	minute
count	336776.0	336776.000000	...	336776.000000	336776.000000
mean	2013.0	6.548510	...	13.180247	26.230100
std	0.0	3.414457	...	4.661316	19.300846
min	2013.0	1.000000	...	1.000000	0.000000
25%	2013.0	4.000000	...	9.000000	8.000000
50%	2013.0	7.000000	...	13.000000	29.000000
75%	2013.0	10.000000	...	17.000000	44.000000
max	2013.0	12.000000	...	23.000000	59.000000

```
[8 rows x 14 columns]
```

```
In [24]: flights.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 336776 entries, 0 to 336775
Data columns (total 19 columns):
year            336776 non-null int64
month           336776 non-null int64
day             336776 non-null int64
dep_time        328521 non-null float64
sched_dep_time 336776 non-null int64
dep_delay       328521 non-null float64
arr_time        328063 non-null float64
sched_arr_time 336776 non-null int64
arr_delay       327346 non-null float64
carrier         336776 non-null object
flight          336776 non-null int64
tailnum         334264 non-null object
origin          336776 non-null object
dest            336776 non-null object
air_time        327346 non-null float64
distance        336776 non-null int64
hour            336776 non-null int64
minute          336776 non-null int64
time_hour       336776 non-null object
dtypes: float64(5), int64(9), object(5)
memory usage: 48.8+ MB
```

# Setting the index

- By default: 0, 1, 2, ...  
`In [12]: flights.index`  
`Out[12]: RangeIndex(start=0, stop=336776, step=1)`
- In most cases, this is what you need
- We can set one of the columns as the index:  
`flights.set_index("month")`

# Setting the index

- New index. We could also use dates ...

```
In [21]: flights.index = np.arange(10, 336776+10)

In [22]: flights
Out[22]:
```

	year	month	day	...	hour	minute	time_hour
10	2013	1	1	...	5	15	2013-01-01 05:00:00
11	2013	1	1	...	5	29	2013-01-01 05:00:00
12	2013	1	1	...	5	40	2013-01-01 05:00:00
13	2013	1	1	...	5	45	2013-01-01 05:00:00
14	2013	1	1	...	6	0	2013-01-01 06:00:00
15	2013	1	1	...	5	58	2013-01-01 05:00:00
16	2013	1	1	...	6	0	2013-01-01 06:00:00
17	2013	1	1	...	6	0	2013-01-01 06:00:00
18	2013	1	1	...	6	0	2013-01-01 06:00:00
19	2013	1	1	...	6	0	2013-01-01 06:00:00
20	2013	1	1	...	6	0	2013-01-01 06:00:00
21	2013	1	1	...	6	0	2013-01-01 06:00:00
22	2013	1	1	...	6	0	2013-01-01 06:00:00
23	2013	1	1	...	6	0	2013-01-01 06:00:00
24	2013	1	1	...	6	0	2013-01-01 06:00:00
25	2013	1	1	...	5	59	2013-01-01 05:00:00
26	2013	1	1	...	6	0	2013-01-01 06:00:00
27	2013	1	1	...	6	0	2013-01-01 06:00:00
28	2013	1	1	...	6	0	2013-01-01 06:00:00
29	2013	1	1	...	6	0	2013-01-01 06:00:00
30	2013	1	1	...	6	10	2013-01-01 06:00:00
31	2013	1	1	...	6	5	2013-01-01 06:00:00
32	2013	1	1	...	6	10	2013-01-01 06:00:00
33	2013	1	1	...	6	10	2013-01-01 06:00:00
34	2013	1	1	...	6	7	2013-01-01 06:00:00
35	2013	1	1	...	6	0	2013-01-01 06:00:00
36	2013	1	1	...	6	0	2013-01-01 06:00:00
37	2013	1	1	...	6	10	2013-01-01 06:00:00
38	2013	1	1	...	6	15	2013-01-01 06:00:00

Terminal de IPython

Historial de comandos

Permisos: RW

Fin de lín

Note: although índices can be useful in some cases (time series, ...), this tutorial will not focus on them

## Extracting the values from a Pandas dataframe or series to a numpy matrix / array

In [39]: flights.values

Out[39]:

```
array([[2013, 1, 1, ..., 5, 15, '2013-01-01 05:00:00'],
       [2013, 1, 1, ..., 5, 29, '2013-01-01 05:00:00'],
       [2013, 1, 1, ..., 5, 40, '2013-01-01 05:00:00'],
       ...,
       [2013, 9, 30, ..., 12, 10, '2013-09-30 12:00:00'],
       [2013, 9, 30, ..., 11, 59, '2013-09-30 11:00:00'],
       [2013, 9, 30, ..., 8, 40, '2013-09-30 08:00:00']], dtype=object)
```

# Selecting rows and columns (indexing)

- Label selection: both rows and columns can have labels:  
**loc**
  - the labels of the rows are the indices (index)
  - the labels of the columns are the column names
- Position (integer) selection: **iloc**
  - Rows: e.g. select rows from 0 to 10
  - Columns: e.g. select rows from 3 to 7
- Boolean selection: selecting rows that satisfy a condition
  - E.g.: Select all rows where age > 35

# Selecting rows and columns (indexing)

- Label selection: both rows and columns can have labels: **loc**
  - the labels of the rows are the indices (index)
  - the labels of the columns are the column names
- Position (integer) selection: **iloc**
  - Rows: select rows from 0 to 10
  - Columns: select rows from 3 to 7
- Boolean selection: selecting rows that satisfy a condition
  - E.g.: Select all rows where age > 35

# Label selection: rows .loc

```
In [22]: flights.head()
....)
Out[22]:
   year  month  day    ...  hour  minute      time_hour
0  2013     1    1    ...    5     15  2013-01-01 05:00:00
1  2013     1    1    ...    5     29  2013-01-01 05:00:00
2  2013     1    1    ...    5     40  2013-01-01 05:00:00
3  2013     1    1    ...    5     45  2013-01-01 05:00:00
4  2013     1    1    ...    6      0  2013-01-01 06:00:00
[5 rows x 19 columns]
```

In [25]: flights.loc[2:4]

Out[25]:

	year	month	day	...	hour	minute	time_hour
2	2013	1	1	...	5	40	2013-01-01 05:00:00
3	2013	1	1	...	5	45	2013-01-01 05:00:00
4	2013	1	1	...	6	0	2013-01-01 06:00:00

# Label selection: columns

## .loc

List of columns

```
In [26]: flights.loc[:, ['month', 'day', 'time_hour']]  
Out[26]:
```

	month	day	time_hour
0	1	1	2013-01-01 05:00:00
1	1	1	2013-01-01 05:00:00
2	1	1	2013-01-01 05:00:00
3	1	1	2013-01-01 05:00:00
4	1	1	2013-01-01 06:00:00
5	1	1	2013-01-01 05:00:00
6	1	1	2013-01-01 06:00:00
7	1	1	2013-01-01 06:00:00
8	1	1	2013-01-01 06:00:00
9	1	1	2013-01-01 06:00:00
10	1	1	2013-01-01 06:00:00
11	1	1	2013-01-01 06:00:00
12	1	1	2013-01-01 06:00:00
13	1	1	2013-01-01 06:00:00
14	1	1	2013-01-01 06:00:00
15	1	1	2013-01-01 05:00:00
16	1	1	2013-01-01 06:00:00
17	1	1	2013-01-01 06:00:00
18	1	1	2013-01-01 06:00:00
19	1	1	2013-01-01 06:00:00
20	1	1	2013-01-01 06:00:00

Range of columns

```
In [30]: flights.loc[:, 'year':'dep_time']  
Out[30]:
```

	year	month	day	dep_time
0	2013	1	1	517.0
1	2013	1	1	533.0
2	2013	1	1	542.0
3	2013	1	1	544.0
4	2013	1	1	554.0
5	2013	1	1	554.0
6	2013	1	1	555.0
7	2013	1	1	557.0
8	2013	1	1	557.0
9	2013	1	1	558.0
10	2013	1	1	558.0
11	2013	1	1	558.0
12	2013	1	1	558.0
13	2013	1	1	558.0
14	2013	1	1	559.0
15	2013	1	1	559.0
16	2013	1	1	559.0
17	2013	1	1	600.0
18	2013	1	1	600.0
19	2013	1	1	601.0
20	2013	1	1	602.0

# Labels: rows and columns

## .loc

```
In [31]: flights.loc[2:4, ['month', 'day', 'time_hour']]  
Out[31]:  
   month  day          time_hour  
2      1    1  2013-01-01 05:00:00  
3      1    1  2013-01-01 05:00:00  
4      1    1  2013-01-01 06:00:00
```

# Beware! series vs. dataframe

This returns a dataframe

In []: flights.loc[:,['month']]

Out[]:

```
month  
0 1  
1 1  
2 1  
3 1  
4 1  
5 1  
6 1  
7 1  
8 1  
9 1
```

In []: type(flights.loc[:,['month']])

Out[]: pandas.core.frame.DataFrame

This returns a series!

In [35]: flights.loc[:, 'month']

Out[35]:

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

In []: type(flights.loc[:, 'month'])

Out[]: pandas.core.series.Series

# Selecting single columns (series)

This returns a series!

In [35]: flights.loc[:, 'month']

Out[35]:

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

The same, with dot notation

In [42]: flights.month

Out[42]:

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

Note: we can get the values as a numpy array

In [43]: flights.month.values

Out[43]: array([1, 1, 1, ..., 9, 9, 9], dtype=int64)

# Shorthand for column selection

- `flights.loc[:, 'month']` is equivalent to  
`flights['month']`
- `flights.loc[:, ['year', 'month']]` is equivalent to  
`flights[['year', 'month']]`

# Selecting rows and columns (indexing)

- Label selection: both rows and columns can have labels:  
**loc**
  - the labels of the rows are the indices (index)
  - the labels of the columns are the column names
- Position (integer) selection: **iloc**
  - Rows: select rows from 0 to 10
  - Columns: select rows from 3 to 7
- Boolean selection: selecting rows that satisfy a condition
  - E.g.: Select all rows where age > 35

# Position (integer) selection: iloc

```
In [22]: flights.head(  
      ...: )  
Out[22]:  
    year  month  day      ...      hour  minute      time_hour  
0  2013      1     1      ...      5      15  2013-01-01 05:00:00  
1  2013      1     1      ...      5      29  2013-01-01 05:00:00  
2  2013      1     1      ...      5      40  2013-01-01 05:00:00  
3  2013      1     1      ...      5      45  2013-01-01 05:00:00  
4  2013      1     1      ...      6       0  2013-01-01 06:00:00  
  
[5 rows x 19 columns]
```

In [41]: `flights.iloc[2:4, 1:3]`

Out[41]:

	month	day
2	1	1
3	1	1

# Combining iloc for rows and loc for columns

- What if we want to select rows by position but columns by name?

```
# Just one column
```

```
In [51]: flights.iloc[2:4, flights.columns.get_loc('month')]
```

```
Out[51]:
```

```
2 1  
3 1
```

```
Name: month, dtype: int64
```

```
# Several columns
```

```
In [53]: flights.iloc[2:4, flights.columns.get_indexer(['month','day'])]
```

```
Out[53]:
```

```
month day  
2 1      1  
3 1      1
```

# Selecting rows and columns (indexing)

- Label selection: both rows and columns can have labels:  
**loc**
  - the labels of the rows are the indices (index)
  - the labels of the columns are the column names
- Position (integer) selection: **iloc**
  - Rows: select rows from 0 to 10
  - Columns: select rows from 3 to 7
- Boolean selection: selecting rows that satisfy a condition
  - E.g.: Select all rows where age > 35

# Boolean indexing: selecting rows on condition

- Both loc and iloc can be used, but **loc** is recommended
- List of flights for January the first?

```
In []: flights.loc[(flights.month == 1) & (flights.day == 1)]
```

```
Out[]:
```

	year	month	day	...	hour	minute	time_hour
0	2013	1	1	...	5	15	2013-01-01 05:00:00
1	2013	1	1	...	5	29	2013-01-01 05:00:00
2	2013	1	1	...	5	40	2013-01-01 05:00:00
3	2013	1	1	...	5	45	2013-01-01 05:00:00
4	2013	1	1	...	6	0	2013-01-01 06:00:00

- Note: we can also write (**flights.loc[:, "month"] == 1**)

# Boolean indexing: selecting rows on condition

- Same thing in two lines (clearer code)
- List of flights for January the first?

```
In []: condition = (flights.month == 1) & (flights.day == 1)
```

```
In []: flights.loc[condition]
```

Out[]:

	year	month	day	...	hour	minute	time_hour
0	2013	1	1	...	5	15	2013-01-01 05:00:00
1	2013	1	1	...	5	29	2013-01-01 05:00:00
2	2013	1	1	...	5	40	2013-01-01 05:00:00
3	2013	1	1	...	5	45	2013-01-01 05:00:00
4	2013	1	1	...	6	0	2013-01-01 06:00:00

# Boolean indexing: selecting rows on condition

- List of flights for January the first?
  - A shorter version

```
• In []: flights.query("month == 1 & day == 1")
```

```
Out[]:
```

	year	month	day	...	hour	minute	time_hour
0	2013	1	1	...	5	15	2013-01-01 05:00:00
1	2013	1	1	...	5	29	2013-01-01 05:00:00
2	2013	1	1	...	5	40	2013-01-01 05:00:00
3	2013	1	1	...	5	45	2013-01-01 05:00:00
4	2013	1	1	...	6	0	2013-01-01 06:00:00

# Boolean conditions

- In order to create conditions, we can use:
  - <, >, ==, <=, >=, !=
  - &: and
  - |: or
  - ~: not
  - isin: is value in a list of values?
  - isnull: is value nan?

# Boolean selection: selecting rows on condition

- What flights start at EWR or JFK airports?

```
In [61]: flights.loc[flights.origin.isin(['EWR', 'JFK']), ['origin', 'dest']]
```

```
Out[61]:
```

```
origin dest
0    EWR  IAH
2    JFK  MIA
3    JFK  BQN
5    EWR  ORD
6    EWR  FLL
8    JFK  MCO
10   JFK  PBI
```

- Note: we are also selecting origin and dest columns

# Creating new columns

- Compute speed for every flight
  - $\text{speed} = \text{distance} / \text{airtime}$
- Two ways:
  - First: **flights.loc[:, 'speed'] =**
    - `flights.loc[:, 'speed'] = flights.distance - flights.air_time`
    - `flights.loc[:, 'speed'] = flights.loc[:, 'distance'] - flights.loc[:, 'air_time']`
    - `flights.loc[:, 'speed'] = flights['distance'] - flights['air_time']`
  - Shorthand: **flights['speed'] =**
    - `flights['speed'] = flights['distance'] - flights['air_time']`

# Creating new columns

```
In [74]: flights['speed'] = flights['distance'] - flights['air_time']
```

```
In [75]: flights
```

```
Out[75]:
```

	year	month	day	...	minute	time_hour	speed
0	2013	1	1	...	15	2013-01-01 05:00:00	1173.0
1	2013	1	1	...	29	2013-01-01 05:00:00	1189.0
2	2013	1	1	...	40	2013-01-01 05:00:00	929.0
3	2013	1	1	...	45	2013-01-01 05:00:00	1393.0
4	2013	1	1	...	0	2013-01-01 06:00:00	646.0
5	2013	1	1	...	58	2013-01-01 05:00:00	569.0
6	2013	1	1	...	0	2013-01-01 06:00:00	907.0
7	2013	1	1	...	0	2013-01-01 06:00:00	176.0
8	2013	1	1	...	0	2013-01-01 06:00:00	804.0
9	2013	1	1	...	0	2013-01-01 06:00:00	595.0
10	2013	1	1	...	0	2013-01-01 06:00:00	879.0
11	2013	1	1	...	0	2013-01-01 06:00:00	847.0
12	2013	1	1	...	0	2013-01-01 06:00:00	2130.0
13	2013	1	1	...	0	2013-01-01 06:00:00	2204.0
14	2013	1	1	...	0	2013-01-01 06:00:00	1132.0
15	2013	1	1	...	59	2013-01-01 05:00:00	143.0
16	2013	1	1	...	0	2013-01-01 06:00:00	1890.0
17	2013	1	1	...	0	2013-01-01 06:00:00	924.0
18	2013	1	1	...	0	2013-01-01 06:00:00	628.0
19	2013	1	1	...	0	2013-01-01 06:00:00	876.0
20	2013	1	1	...	10	2013-01-01 06:00:00	850.0
21	2013	1	1	...	5	2013-01-01 06:00:00	397.0
22	2013	1	1	...	10	2013-01-01 06:00:00	933.0
23	2013	1	1	...	10	2013-01-01 06:00:00	632.0
24	2013	1	1	...	7	2013-01-01 06:00:00	928.0

# Modifying subsets of the dataframe (setting)

- Let's put *nan* on the first three rows and columns 'year', 'month' and 'day'

```
# Let's create a copy first
```

```
In [81]: flights_copy = flights.copy()
```

```
# Let's see the content of the first three rows and the  
first three columns
```

```
In [83]: flights_copy.iloc[0:4,  
flights.columns.get_indexer(['year', 'month', 'day'])]
```

```
Out[83]:
```

```
year month day  
0 2013 1 1  
1 2013 1 1  
2 2013 1 1  
3 2013 1 1
```

```
# Now, we do the assignment
```

```
In [85]: flights_copy.iloc[0:4,
```

```
flights.columns.get_indexer(['year', 'month', 'day'])] = np.nan
```

```
In [86]: flights_copy
```

```
Out[86]:
```

	year	month	day	...	minute	time_hour	speed
0	NaN	NaN	NaN	...	15	2013-01-01 05:00:00	1173.0
1	NaN	NaN	NaN	...	29	2013-01-01 05:00:00	1189.0
2	NaN	NaN	NaN	...	40	2013-01-01 05:00:00	929.0
3	NaN	NaN	NaN	...	45	2013-01-01 05:00:00	1393.0
4	2013.0	1.0	1.0	...	0	2013-01-01 06:00:00	646.0
5	2013.0	1.0	1.0	...	58	2013-01-01 05:00:00	569.0
6	2013.0	1.0	1.0	...	0	2013-01-01 06:00:00	907.0
7	2013.0	1.0	1.0	...	0	2013-01-01 06:00:00	176.0

# Subset (modification) with boolean selection

- Let's create a column 'satisfaction' with 'good' if arrival delay  $\leq 75$ , and 'bad' otherwise

```
In [107]: flights['satisfaction'] = 'bad'
```

```
In [108]: flights.loc[flights.arr_delay <= 75, 'satisfaction'] = 'good'
```

```
In [109]: flights.loc[:, ['arr_delay', 'satisfaction']].head()
```

Out[109]:

```
arr_delay satisfaction
0 11.0 good
1 20.0 good
2 33.0 good
3 -18.0 good
4 -25.0 good
```