# Programming for Data Science – The NumPy and pandas libraries

Henrik Boström
Prof. of Computer Science - Data Science Systems
Division of Software and Computer Systems
Department of Computer Science
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
bostromh@kth.se

# Outline

## The NumPy library

- Creating and accessing arrays
- Operating on arrays (broadcasting)
- Missing values

## The pandas library

- Creating, accessing and updating DataFrames
- Creating groups in DataFrames
- Categorical values
- Importing and exporting DataFrames

# NumPy

The NumPy library was created by Travis Oliphant in 2005. Main features include:

- It provides operations and functions for efficient handling of large multidimensional arrays
- It defines the `ndarray` class, which represent homogeneously typed arrays. An ndarray cannot (easily) be extended or reshaped without creating a copy of the data.
- NumPy includes a large number of highly optimized functions for linear algebra, random number generation, etc. Efficient integration with existing numerical libraries, e.g., BLAS, LAPACK, is enabled, as ndarrays provide views into memory buffers without having to copy data.

# Working with NumPy

- Importing the library

```
import numpy as np
```

- Using functions

```
np.sqrt(9)                          # 3.0
np.square(3)                        # 9 (= 3**2)
np.maximum(2,3)                     # 3
np.absolute(-10)                    # 10 (= abs(-10))
np.sign(-10)                        # -1
```

# Creating arrays with NumPy

- Creating arrays

```
a1 = np.array([1,2,3,4])
isinstance(a1,np.ndarray)        # True
a1.shape                         # (4,) - vector w. 4 el.

a2 = np.array([[1,2,3],[4,5,6]])
a2.shape                         # (2,3) - 2 rows, 3 cols.
a2.dtype                         # dtype('int64')

a3 = np.arange(27).reshape(3,3,3)
a3.shape                         # (3,3,3)
```

# Creating arrays with NumPy (cont.)

- Creating arrays (cont.)

```
a4 = np.zeros((5,2),dtype=bool)  # All values set to False

a5 = np.empty(10)                # Uninit. vec. of floats
```

- The datatypes; `int`, `float`, `bool`, object, may be inferred or declared

```
a6 = np.array([[1,2],[3,4],[5]],dtype=object)
a6.shape                         # (3,)
a6.dtype                         # dtype('O') (object)

a7 = np.array([1,2,3.14])        # a7 = array([1,2,3.14],
                                               dtype=float)
```

# Accessing NumPy arrays

- Accessing arrays

```
a = np.array([1,2,3,4])
a[0]                        # 1
a[1:3]                      # array([2,3])
a[:3]                       # array([1,2,3])
a[2:]                       # array([3,4])
a[[1,1]]                    # array([2,2])
a[[True,False,False,True]]  # array([1, 4])

b = np.arange(12).reshape(4,3)
b[3,2]                      # 11
b[1:3,:2]                   # array([[3,4],[6,7]])
b[[0,3],[0,2]]              # array([0,11])
```

# Accessing NumPy arrays (cont.)

- Assigning values to array elements

```
a[0] += 1                    # a = array([2,2,3,4])
a[2:] = np.array([5,6])      # a = array([2,2,5,6])
a[:2] = [3,4]                # a = array([3,4,5,6])

b = a[1:3]                   # view (not copy) of array([4,5])
b[0] = 3                     # b = array([3,5])
                             # a = array([3,3,5,6])

c = a.copy()                 # c = array([3,3,5,6])
c[1] = 7                     # c = array([3,7,5,6])
                             # a = array([3,3,5,6])
```

# Operating on NumPy arrays

- Operations involving arrays

```
a = np.array([1,2,3])
np.square(a)                    # array([1, 4, 9]) (= a**2)
np.sign(a)                      # array([1, 1, 1])
```

- Operations involving multiple arrays of same size

```
a = np.array([1,2,3])
b = np.array([4,5,6])
c = a+b                         # c = array([5,7,9])
```

# Operating on NumPy arrays: broadcasting

- Operations involving operands of different size (*broadcasting*); works when differing dimension size equals 1 for one operand

```
a = np.array([1,2])
b = np.array([4,5,6])
c = a+b                    # Error

d = 5*b                    # d = array([20,25,30])
e = b > 5                  # e = array([False,False,True])
f = np.array([[1,2],[3,4]])
g = a+f                    # g = array([[2,4],[4,6]])

h = np.array([[1],[2]])    # 2x1 array
i = np.array([1,2])        # 1x2 array
j = h-i                    # j = array([[0,-1],[1,0]])
```

# Operating on NumPy arrays (cont.)

- Operating on multiple array elements with `apply_along_axis`

```
a = np.arange(15).reshape(5,3)

np.apply_along_axis(lambda x: np.sum(x),0,a)
                                      # Sum over rows:
                                      # array([30, 35, 40])
def f(x):
   return np.sum(x)

np.apply_along_axis(f,1,a)            # Sum over columns:
                                      # array([ 3, 12, 21, 30, 39])

np.apply_along_axis(np.sum,1,a)# Equivalent
```

# Operating on NumPy arrays (cont.)

- Finding the position of the maximum value with `argmax`

```
a = (np.arange(15)*2).reshape(5,3)

np.argmax(a)                # Position over flattened array:
                            # 14

np.argmax(a,0)              # Positions over rows:
                            # array([4, 4, 4])

np.argmax(a,1)              # Positions over columns:
                            # array([2, 2, 2, 2, 2])
```

# Missing values in NumPy arrays

- Representing missing values by (the float) `np.nan`

```
a = np.array([1,2,3,4],dtype=float)
a[2] = np.nan

b = np.array([1,2,3,4])  # dtype = int
b[2] = np.nan            # Error

c = np.array([True,np.nan,False,False])
                         # c = array([ 1., nan,  0.,  0.])

np.nan == np.nan         # False
np.nan is np.nan         # True
np.isnan(np.nan)         # True
```

# Missing values in NumPy arrays (cont.)

- Counting with missing values

```
np.sum(np.array([1,2,np.nan,4]))        # np.nan

np.nansum(np.array([1,2,np.nan,4]))     # 7.0

np.nanprod(np.array([1,2,np.nan,4]))    # 8.0

np.nanmin(np.array([1,2,np.nan,4]))     # 1.0

np.nanmax(np.array([1,2,np.nan,4]))     # 4.0

np.nanmean(np.array([1,2,np.nan,4]))    # 2.333...
```

# Missing values in NumPy arrays (cont.)

Other approaches to handling missing values in NumPy arrays include:

- using a special value that represents missingness (does not work for Boolean values)
- using a mask (Boolean array) to represent missing values, see e.g., the numpy.ma module

# pandas

The pandas library was created by Wes McKinney in 2008. Main features include:

- It provides operations and functions for efficient manipulation of tabular data
- It defines the `DataFrame` class, which represent heterogeneous matrices, where columns and rows may be labeled
- pandas includes a large number of functions for slicing, indexing, merging, joining, importing and exporting data, and provides convenient handling of missing values. The library is highly optimized for performance, partly implemented in Cython and C, and relying on NumPy.

# Creating DataFrames with pandas

- Importing the library

```
import pandas as pd
```

- Creating DataFrames

```
values = np.arange(15).reshape(5,3)
df1 = pd.DataFrame(values,columns=["A","B","C"])
print(df1)
```

```
    A   B   C
0   0   1   2
1   3   4   5
2   6   7   8
3   9  10  11
4  12  13  14
```

# Creating DataFrames with pandas (cont.)

- Creating DataFrames (cont.)

```
df2 = pd.DataFrame(values,index=list("abcde"),
                   columns=list("ABC"))
print(df2)
```

```
    A   B   C
a   0   1   2
b   3   4   5
c   6   7   8
d   9  10  11
e  12  13  14
```

# Accessing DataFrames

- Accessing DataFrames

```
df1["B"]                       # Values in column B, df1.B
isinstance(df1["B"],pd.Series)# True
df1["B"].values                # values as a ndarray
df1["B"].dtype                 # dtype('int64')

df1.loc[1:3,["B","C"]]         # Subset of rows and columns
df1.iloc[1:4,1:]               # Same using integer index
df1.iloc[1:3]                  # Subset of rows
df1.iloc[:,1:]                 # All rows, subset of cols.
```

# Accessing DataFrames (cont.)

- Accessing DataFrames

```
df2.loc["a",["C","B"]]        # First row and columns C & B
df2.loc["a":"d"]              # All but last row, all cols.

df2.loc[:,["A","B"]].iloc[0:3] # Two columns, three rows
df2.loc["a":"c",["A","B"]]    # Three rows, two columns

df2.loc[[True,True,False,False,False],[True,False,True]]
                              # Two rows, two columns

df2[[True,True,False,False,False]]
                              # Two rows, all columns
```

# Accessing DataFrames (cont.)

- Accessing DataFrames

```
df2.loc["a",["C","B"]]      # First row and columns C & B
df2.loc["a":"d"]            # All but last row, all columns

df2.iloc[0:4]               # Same using integer index

df2.loc["a","A"]            # Access to single value
df2.iloc[0,0]               # As previous, using int. index
```

# Accessing DataFrames (cont.)

- Accessing DataFrames through boolean indexing

```
df1[[True,False,True,False,True]]
```

| | A | B | C |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 2 | 6 | 7 | 8 |
| 4 | 12 | 13 | 14 |

```
df1[df1["B"] % 2 == 0]
```

| | A | B | C |
|---|---|---|---|
| 1 | 3 | 4 | 5 |
| 3 | 9 | 10 | 11 |

# Accessing DataFrames (cont.)

- Accessing DataFrames through boolean indexing (cont.)

```
df1[df1["C"].isin([5,8,11])]
```

```
   A   B   C
1  3   4   5
2  6   7   8
3  9  10  11
```

```
df1[(df1["A"] > 3) & (df1["C"].isin([5,8,11]))]
```

```
   A   B   C
2  6   7   8
3  9  10  11
```

# Accessing DataFrames (cont.)

- Accessing DataFrames through boolean indexing (cont.)

```
df1[(df1["B"] % 2 == 0) | (df1["A"] > 10)]
```

| | A | B | C |
|---|---|---|---|
| 1 | 3 | 4 | 5 |
| 3 | 9 | 10 | 11 |
| 4 | 12 | 13 | 14 |

# Updating DataFrames

- Assigning values to DataFrames

```
df1["B"] = [True,True,False,False,False]
df1["B"].dtype                              # dtype('bool')
df1["D"] = np.arange(5,dtype=float)
df1["E"] = 1
df1.loc[1,"A"] = np.nan                     # A gets type float
df1.loc[2,"B"] = np.nan                     # B gets type float
df1.isnull().values.any()                   # True
```

```
      A    B   C    D  E
0   0.0  1.0   2  0.0  1
1   NaN  1.0   5  1.0  1
2   6.0  NaN   8  2.0  1
3   9.0  0.0  11  3.0  1
4  12.0  0.0  14  4.0  1
```

# Updating DataFrames (cont.)

- Dropping rows and columns in DataFrames

```
df1.drop(columns="A",inplace=True) # Remove col A from df1
df2 = df1.drop(columns="A",        # Copy df1 without col A
               errors="ignore")
df3 = df2.drop(index=[2,3])        # Copy w/o rows 2 and 3
```

- Copying DataFrames

```
df = df1.copy()                    # Copy of df1
df1.loc[0,"D"] = 2.0
df.loc[0,"D"] = 3.0
df.loc[0,"D"] == df1.loc[0,"D"]    # False
```

# Concatenating DataFrames

- Concatenating DataFrames

```
df = pd.DataFrame({"A": list("ababab"), "B":[0,0,0,1,1,1],
                   "C": [10,20,30,40,50,60]})
pd.concat([df.iloc[3:],df.iloc[:3]])
```

|   | A | B | C |
|---|---|---|---|
| 3 | b | 1 | 40 |
| 4 | a | 1 | 50 |
| 5 | b | 1 | 60 |
| 0 | a | 0 | 10 |
| 1 | b | 0 | 20 |
| 2 | a | 0 | 30 |

# Merging DataFrames

- Merging DataFrames (SQL style)

```
df1 = pd.DataFrame({"LKey": list("abcdef"),
                    "A":[0,0,0,1,1,1]})
df2 = pd.DataFrame({"RKey": list("fedcba"),
                    "B":[0,0,0,1,1,1]})
df1.merge(df2,how="outer",left_on="LKey",right_on="RKey")
```

```
  LKey  A RKey  B
0    a  0    a  1
1    b  0    b  1
2    c  0    c  1
3    d  1    d  0
4    e  1    e  0
5    f  1    f  0
```

# Creatings groups in DataFrames

- Creating groupings

```
df = pd.DataFrame({"A": list("ababab"), "B":[0,0,0,1,1,1],
                   "C": [10,20,30,40,50,60]})
g = df.groupby("A")
g.get_group("a")
   A  B   C
0  a  0  10
2  a  0  30
4  a  1  50
g.sum()                                  # Alt: g.aggregate(np.sum)
   B    C
A
a  1   90
b  2  120
```

# Creating groups in DataFrames (cont.)

- Creating groupings (cont.)

```
df = pd.DataFrame({"A": list("ababab"), "B":[0,0,0,1,1,1],
                   "C": [10,20,30,40,50,60]})
g = df.groupby(["A","B"])
g.get_group(("a",0))
   A  B   C
0  a  0  10
2  a  0  30
g.size()
A B
a 0    2
  1    1
b 0    1
  1    2
```

# Categorical values in DataFrames

- Defining and using categorical values

```
df = pd.DataFrame({"id":[1,2,3,4,5],
                   "award":["silver", "gold", "silver",
                            "silver", "gold"]})
df["award"] = df["award"].astype("category")
df["award"].cat.categories    # Index(['gold', 'silver'],
                              #        dtype='object')
df["award"] = df["award"].cat.set_categories(["gold",
                            "silver","bronze"])
g = df.groupby("award").size()
award
gold      2
silver    3
bronze    0
g.get("iron",0)               # Returns 0 (None w/o def.)
```

# Importing and exporting DataFrames

- Reading and writing to comma-separated text (csv) files

```
df = pd.DataFrame({"id":[np.nan,2,3,4,5],
                   "grade":[np.nan,"b",np.nan,"c","a"]})
df.to_csv("myfile.csv",index=False)
df2 = pd.read_csv("myfile.csv")
```

```
    id grade
0  NaN   NaN
1  2.0     b
2  3.0   NaN
3  4.0     c
4  5.0     a
```

- Plenty of other formats available; Excel, JSON, HTML, SQL, ...

# Summary

- NumPy and pandas are crucial libraries for data scientists using Python
- We have only touched upon what can be done with these; there lots of additional functionality (check the documentation)
- Since the libraries are not formally part of the language, they evolve much more rapidly; look out for new (and deprecated) functionality!