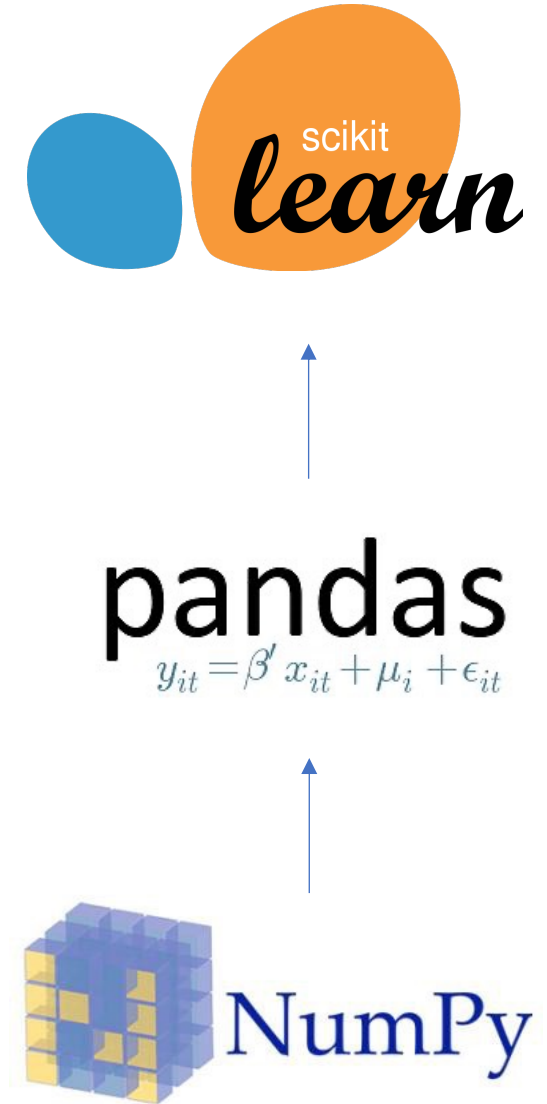


Follow me: L4_Code.py

Introduction to Python for Data Science

Dr Ekaterina Abramova

Lecture 4
Autumn 2021



Broadcasting (1)

Broadcasting – describes how arithmetic works between **arrays of different shapes**.

- Typically you have a larger and smaller array; you want to use the smaller array multiple times to perform an operation (+, *, etc) on the larger array.

Simplest Example of Broadcasting: occurs when combining a scalar value with an array:

```
y = np.arange(4)
y.shape # (4,)
y
>>> array([0, 1, 2, 3])
```

```
y * 2
>>> array([0, 2, 4, 6])
```

We say: “scalar value 2 has been broadcast to all of the other elements in multiplication operation”.

Rules for Operations Between Arrays. The Dimensions are Compatible if:

- For each trailing dimension (i.e. starting from the end) the axis lengths match**. Simple case is when dimensions are equal for both arrays e.g. (3,4) and (3,4).

```
x = np.arange(12).reshape(3,4)
y = np.arange(12).reshape(3,4)
>>> array([[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]])
```

```
x * y # (3,4) and (3,4)
>>> array([[ 0,  1,  4,  9],
          [16, 25, 36, 49],
          [64, 81, 100, 121]])
```

Note: element-wise operation was carried out.

Note: `x * y` just calls universal function `np.multiply(x, y)` internally.

- If either of the lengths is 1**. [see next slide for example explanations]

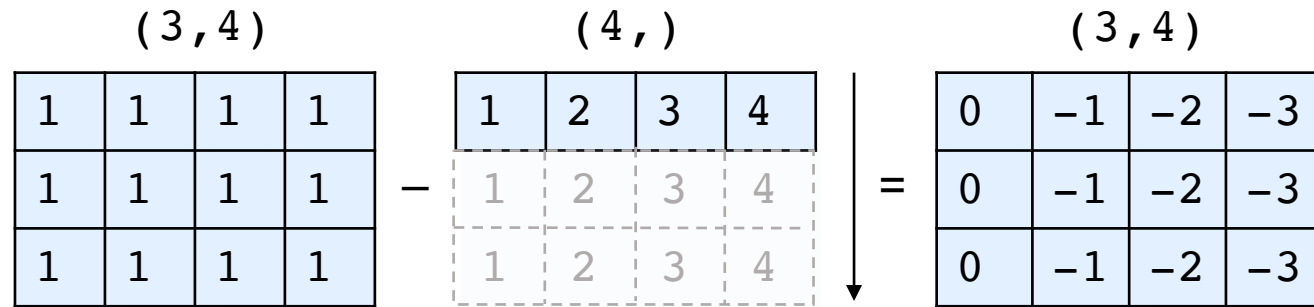
Broadcasting (2)

Example: broadcasting column-wise

- ✓ $(3,4) - (4,)$ i.e. matrix minus vector
- ✓ $(3,4) - (1,4)$ i.e. matrix minus matrix

```
x = np.ones((3,4), dtype = np.int) # (3,4)
>>> array([[1, 1, 1, 1],
          [1, 1, 1, 1],
          [1, 1, 1, 1]])
```

```
y = np.arange(1,5) # (4,) vector
>>> array([1, 2, 3, 4])
```



```
x - y # (3,4) (4,)
x - y.reshape((1,4)) # (3,4)matrix (1,4)matrix
```

Reminder: operations are performed element-wise

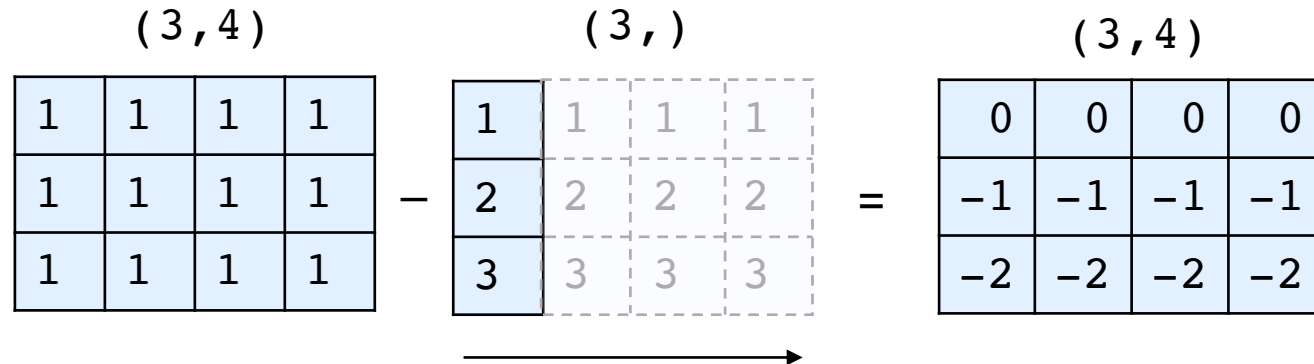
✗ $(3,4) - (4,1)$ i.e. if used `x - y.reshape((4,1))` then won't work

Example: broadcasting row-wise

- ✓ $(3,4) + (3,1)$

```
z = np.array([1,2,3]) # (3,)
>>> array([1, 2, 3])
```

```
x - z.reshape((3,1)) # (3,4) (3,1)
```



✗ $(3,4) - (3,)$ i.e. `x - z`

Reminder: to obtain dimensionality of an array use `a.shape` command

Indexing and Slicing

1D arrays – just like for Lists

Array a:

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

When using start:stop remember **stop-1**.

```
a[0] # 10
```

```
a[2] # 12
```

```
a[2:5] # start, stop Slice: [12, 13, 14]
```

```
a[1:4:2] # start, stop, step Slice: [11, 13]
```

2D arrays – use row i, col j

Array b:

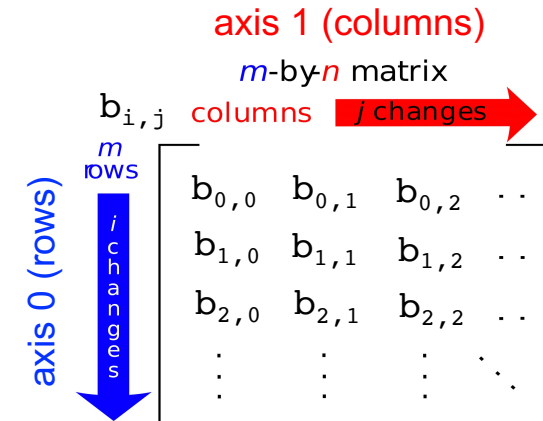
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
b[0, 0] # 0
```

```
b[1, :] or b[1, ] or b[1] # 2nd row: [3, 4, 5]
```

```
s = b[1:, 1:] # array([[4, 5],
                     [7, 8]])
```

Note: Array **slices** are **VIEWS** on the original array i.e. data is NOT COPIED and modifications to the view WILL be seen in the original array.



We can broadcast a value to elements within an array using slicing and assignment.

“bare slice” →

```
a[3:5] = 0 # 0 is broadcast to all elements of the slice a
>>> array([10, 11, 12, 0, 0, 15])
```

Note: To ensure you obtain a copy of the array slice use:

```
s = b[1:, 1:].copy()
```

```
s[: ] = 0 # 0 is broadcast to all elements of the slice s
```

```
s
```

```
>>> array([[0, 0],
           [0, 0]])
```

b # since s has an alias with b, the original array is also changed:

```
>>> array([[0, 1, 2],
           [3, 0, 0],
           [6, 0, 0]])
```

Boolean Indexing

We can index an array using another array, which contains boolean values. This means we can select elements of one array based on truth value of another.

Given the following two arrays:

```
gender = np.array(["Female", "Male", "Male", "Male", "Female"])
data = np.array([4, 8, 3, 9, 2])
```

Let's use boolean indexing to select elements from data array based on boolean array's truth values. i.e. only select those elements of data array where indexing boolean value is True:

```
bools = gender == "Male" # array([False,  True,  True,  True, False])
data   # array([4, 8, 3, 9, 2])
s = data[bools]           # array([8, 3, 9])
# This can be compactly written on a single line:
s = data[gender == "Male"] # array([8, 3, 9])
```

Element-wise (vectorized) **relational check** between array gender and string "Male" results in a boolean array.

Boolean array must be of the same length as the array it is indexing.

Select logical opposite of bools using **logical operator ~** (remember this means not)

```
s = data[~ bools]           # array([4, 2])
# where:
~ bools                     # array([ True, False, False, False,  True])
```

Boolean array always creates a copy of the data.

In a similar way, we can use boolean indexing to set values.

```
data[gender == "Male"] = 0 # array([4, 0, 0, 0, 2])
```

L4_LAB_Questions.py Q1

Libraries



pandas – Series

```
import pandas as pd
```

• **Series** (data structure) – 1D array-like object containing sequence of values and an associated array of data labels (called *index*).

```
s = pd.Series([1,2,3], index=['a','b','c']) # a 1
                                     b 2
                                     c 3
                                     dtype: int64
```

list or array

s.values obtain values only # array([1, 2, 3])

s.index obtain index only # Index(['a', 'b', 'c'], dtype='object')

- Think of Series as a fixed length **ordered** dict, as it is a mapping of index values to data values. Also can convert dict → Series

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000}
s = pd.Series(sdata) # Ohio 35000
                    Texas 71000
                    Oregon 16000
                    dtype: int64
```

- name** attribute for Series and index objects:

```
s.name = 'population' # state
s.index.name = 'state'
Ohio 35000
Texas 71000
Oregon 16000
Name: population, dtype: int64
```

Indexing Series:

s[0] By position
s['a'] By index value

Slicing Series:

s[0:2] start : (stop-1)
s['a':'c'] end point inclusive

Use NumPy Universal Functions:

s2 = np.add(s, 10)

Use NumPy Operations and Methods:

Preserves the index-value link.

s + s
s * 2
'a' in s # check membership
s.size # num of elems in a Series
s.dtype # Series data type

pandas – DataFrame

DataFrame – 2D table with ordered collection of columns with different data type: int, float, string, boolean and a labelled axes (i.e. **row index**, **column index**). It can be created by typecasting lists/arrays to dataframe:

Construct dataframe from a dict with values as lists of equal length:

```
data = {'GDP' : [ 3.8,  3.7,  3.2],
        'CPI' : [ 1.6,    3,  2.1],
        'year': [2019, 2020, 2021]}
```

```
df = pd.DataFrame(data, columns=['year', 'GDP', 'CPI', 'result'])
```

	year	GDP	CPI	result
0	2019	3.8	1.6	NaN
1	2020	3.7	3.0	NaN
2	2021	3.2	2.1	NaN

If you pass a column that isn't contained in data dict, column will of missing vals (NaN) is created.

df.values yields a numpy array of dataframe's values.

```
array([[2019, 3.8, 1.6, nan],
       [2020, 3.7, 3.0, nan],
       [2021, 3.2, 2.1, nan]], dtype=object)
```

df.dtypes display data type for each column of the dataframe.

Retrieve column as Series:

df['CPI'] dict-like notation
df.CPI col name notation

Retrieve multiple columns:

df[['CPI', 'year']]

Create new column:

df['new'] = 0

	year	GDP	CPI	result	new
0	2019	3.8	1.6	NaN	0
1	2020	3.7	3.0	NaN	0
2	2021	3.2	2.1	NaN	0

Delete row, using row index:

df.drop([0], inplace=True)

Delete column:

df.drop(['new'], axis=1, inplace = True)

del df['new'] using col index

pandas – Creating DataFrames

```
df = pd.DataFrame(data,
                  index = ['row1', 'row2', 'row3'],
                  columns = ['year', 'GDP', 'CPI', 'result'])
```

Just like for a Series:

```
df.index.name = 'obs'
```

```
df.columns
```

obtain column labels

```
Index(['year', 'GDP', 'CPI', 'result'], dtype='object')
```

```
df.index
```

obtain index (row) labels

```
Index(['row1', 'row2', 'row3'], dtype='object', name='year')
```

	year	GDP	CPI	result
obs				
row1	2019	3.8	1.6	NaN
row2	2020	3.7	3.0	NaN
row3	2021	3.2	2.1	NaN

Creating a DataFrame based on the shape and names of an existing DataFrame:

```
df2 = pd.DataFrame(np.nan, index = df.index, columns = df.columns)
```

✓ initialises dataframe df2 to NaN with col data type float64

✓ if you don't use np.nan col data type would be object

	year	GDP	CPI	result
row1	NaN	NaN	NaN	NaN
row2	NaN	NaN	NaN	NaN
row3	NaN	NaN	NaN	NaN

```
df3 = pd.DataFrame(np.nan, index = np.arange(N), columns = df.columns)
```

✓ row labels are now numeric

✓ ready to be filled with results.

	year	GDP	CPI	result
0	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN

```
df4 = pd.DataFrame(data)
```

Convert other containers to DataFrame, e.g. lists.

DataFrame – Methods

Some useful methods for dataframes:

Syntax / Optional Arguments	Example	Description
<code>df.head()</code> / <code>df.tail()</code>	<code>df.head()</code> # default is 5 <code>df.head(n)</code> # n top rows	Examine <i>top / bottom</i> rows of df Examine <i>top / bottom</i> n rows of DataFrame
<code>df.set_option('display.max_columns', n)</code>	n should be number of cols	Print out all columns of the dataframe.
<code>df.shape</code>	(100, 4) is 100 rows; 4 cols	Tuple containing size of the dataframe
<code>df.describe()</code>	<code>df.describe()</code>	Basic summary statistics (down each column)

We can apply methods to individual columns at a time or to an entire dataframe and/or specify axis of interest:

Syntax (col, loc, iloc)	Column Example	Description
<code>df['col'].mean()</code> <code>df['col'].std()</code>	<code>df['Return'].mean()</code> <code>df['Return'].std()</code>	Reduces column to a single value, mean of the col. Reduces column to a single value, std of the col.
<code>df.mean(axis = 0)</code>		“down rows ” ↓ (default)
<code>df.mean(axis = 1)</code>		“across columns ” →

Example: timeseries data ordered old to new, `shift(1)` creates a **lag** on the column.

	year	CPI			year	CPI	tAhead	tLag
0	2016	1.6	$\xrightarrow{\begin{array}{l} \text{df['tLag']} = \text{df['CPI'].shift(1)} \\ \text{df['tAhead']} = \text{df['CPI'].shift(-1)} \end{array}}$	0	2016	1.6	3.0	NaN
1	2017	3.0		1	2017	3.0	2.1	1.6
2	2018	2.1		2	2018	2.1	NaN	3.0

*shift(1) is useful for
creating returns*

DataFrame – Indexing & Slicing

Selection with **loc** (axis **labels**) and **iloc** (**integers**) – more [here](#).

Select rows and columns from a DataFrame with NumPy-like notation.

```
"""
    year  GDP  CPI  result
row1  2016  3.8  1.6    NaN
row2  2017  3.7  3.0    NaN
row3  2018  3.2  2.1    NaN
"""
```

loc Syntax	Example	Description
df.loc[val]	df.loc['row1'] df.loc['row1':'row3'] df.loc[['row3','row1']]	Single row selection using label . Subset of rows using labels with slicing . End inclusive . using seq of values .
df.loc[:, val]	df.loc[:, 'CPI'] df.loc[:, 'year':'CPI'] df.loc[:, ['result','CPI']]	Single column selection using label . Subset of columns using labels with slicing . End inclusive . using seq of values .
df.loc[val1, val2]	Combinations of above	Select rows and columns by label .

iloc Syntax	Example	Description
df.iloc[where]	df.iloc[0] 1 st row df.iloc[0:3] row 1-3 df.iloc[[2,0]] row 3,1	Single row selection using integer position . Subset of rows using integers with slicing . Non end incl . using seq of integers .
df.iloc[:, where]	df.iloc[:, 2] CPI df.iloc[:, 0:2] year, GDP df.iloc[:, [3,2]] result, CPI	Single column selection using integer position . Subset of columns using integers with slicing . Non end incl . using seq of integers .
df.iloc [where_i, where_j]	Combinations of above	Select rows and columns by integer .

pandas – Detect Missing Data

NUMPY ARRAY:

Use the Universal function `np.isnan()`:

1. Obtain boolean vector True/False

```
a = np.array([np.nan, 1, 2])
np.isnan(a)
>>> array([ True, False, False])
```

2. Obtain a single value of number of NaNs

```
np.isnan(a).sum()
>>> 1
```

3. Obtain a single True / False

```
np.isnan(a).any()
>>> True
```

✓ Chain methods using dot notation:
method1().method2()

Note: the following won't work since None is not NaN:

If we populate an array with None it does not get typecast

```
a = np.array([None, 1, 2])
>>> array([None, 1, 2], dtype=object)
np.isnan(a)
>>> TypeError: ufunc 'isnan' not supported
```

PANDAS SERIES:

Can also use `np.isnan()` or pandas `pd.isnull()` using the same syntax as for numpy array since a Series is just a named array. Or using pandas methods:

1. Obtain boolean vector True/False

```
a = np.array([np.nan, 1, 2])
s = pd.Series(a)
s.isnull()
```

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

2. Obtain a single value of number of NaNs

```
s.isnull().sum() # 1
```

3. Obtain a single True / False

```
s.isnull().any() # True
```

PANDAS DATAFRAME:

1. Obtain number of NaNs per feature / column

```
df.isnull().sum()
```

```
year      0
GDP       0
CPI       0
result    3
dtype: int64
```

2. Obtain a single value of number of NaNs in entire dataframe

```
df.isnull().values.sum() # 3
```

pandas – Data Preprocessing (1)

- **Import a .csv** – ensure to use the suggested optional arguments:

```
df = pd.read_csv('xxx.csv', index_col = 0, parse_dates = True)
```

Use 1st column of csv as index column (row names).

Convert string date to datetime obj.

- **Remove NaN** – any rows with missing values get removed from the dataframe:

```
df.dropna(inplace = True)
```

'in place' means affect the original data frame, rather than return a new one.

- **Fill in missing values** – decide how you would like to replace missing values for a Series (defaults shown).

```
df['col'].fillna(value = None, method = None, inplace = False)
```

scalar: e.g. 0,
Series: with values to use at each index

'ffill': forward fill propagate last valid obs forward to next valid entry.
'bfill': back fill use next valid observation to fill in gap.

Say we have df:

	gender	age	x
0	male	3.3	1.6
1	female	3.7	NaN
2	male	3.2	2.1
3	female	4.1	NaN
4	male	3.3	NaN
5	female	1.6	2.2

Replace NaN with 0:

```
df['x'].fillna(value = 0)
```

0	1.6
1	0.0
2	2.1
3	0.0
4	0.0
5	2.2

Name: x, dtype: float64

Forward fill NaNs with last valid obs values:

```
df['x'].fillna(method = 'ffill')
```

0	1.6
1	1.6
2	2.1
3	2.1
4	2.1
5	2.2

Name: x, dtype: float64

pandas – Data Preprocessing (2)

Read L4_LAB.pdf

L4_LAB_Questions.py Q2

Let's fill values in one column based on values of another column.

- Say we want to fill in `x` based on average age of a person. It is also alleged that males and females have different average lifespan, so perhaps we also want to group by gender first.

```
df['col'].fillna(value = None, inplace = True)
```

Remember value can also be a Series, indicating a value to be used at each index.

We can also affect the array directly by using `inplace = True`.

Steps:

1. Group the data by gender:

```
gp = df.groupby('gender')
```

```
# <pandas.core.groupby.generic.DataFrameGroupBy>
```

2. Find the median value for each gender and create a Series placing respective median values against male/female rows:

```
val = gp.transform('median').age
```

3. Fill in missing values with the obtained Series:

```
df['x'].fillna(val, inplace = True)
```

Say we have **df**:

	gender	age	x
0	male	3.3	1.6
1	female	3.7	NaN
2	male	3.2	2.1
3	female	4.1	NaN
4	male	3.3	NaN
5	female	1.6	2.2

df['x']
0 1.6
1 NaN
2 2.1
3 NaN
4 NaN
5 2.2

→

val
0 3.3
1 3.7
2 3.3
3 3.7
4 3.3
5 3.7

→

df['x']
0 1.6
1 3.7
2 2.1
3 3.7
4 3.3
5 2.2


Values filled based on median age value of each gender. Notice original data kept intact, only NaNs filled.

Sklearn Library



Machine Learning Library – **scikit-learn**

```
import sklearn
```

- SciKit-Learn depends on 2 scientific libraries: NumPy and SciPy (e.g. linear algebra, mathematical constants and functions, optimisation tools, root searching algorithms and statistical distributions).
- For plotting use **matplotlib** and  libraries (also contains data), `conda install seaborn`
- Python SciKit-Learn (library documentation [link](#)) list for:
 - Data Pre-Processing (documentation [link](#))
 - Supervised Learning (algorithms reference [link](#))
 - Unsupervised Learning (algorithms ref [link](#))
 - Currently no API for Reinforcement Learning algorithms
- Import toy datasets (e.g. Iris) or benchmark datasets (e.g. MNIST) into dict object called 'Bunch':

Generic syntax:

```
from sklearn.datasets import load_name  
data = load_name()
```

Example:

```
from sklearn.datasets import load_iris  
iris = load_iris()
```

- Rows and columns of data are called *samples* and *features*. Overall, the data matrix is called **feature / design matrix**, stored as a 2D ndarray or DataFrame called `X` of shape `[n_samples, n_features]`.
- Dependent variable is called **label** or **target**, its a 1D numpy array or Series of length `n_samples`. It contains continuous numerical values or discrete class / labels.

Sklearn Iris Dataset

- One of the available datasets is Iris (further datasets [here](#)):

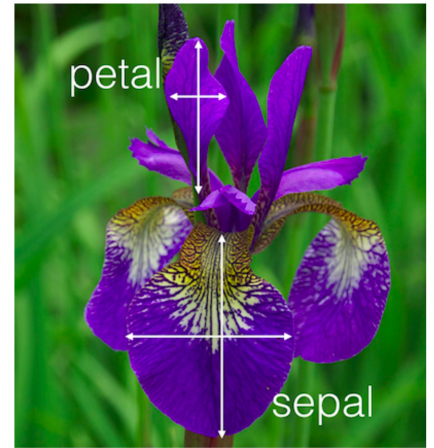
```
from sklearn.datasets import load_iris
iris = load_iris()
```

- Examine type and shape of the resulting object:

```
type(iris) # sklearn.utils.Bunch (this is a dictionary)
iris.keys() # dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])
iris['data'].shape # (150, 4)
type(iris['data']) # numpy.ndarray
iris['target'].shape # (150,)
type(iris['target']) # numpy.ndarray
```

- Obtain feature and target names

```
iris['target_names'] # array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
iris['feature_names'] # ['sepal len (cm)', 'sepal width (cm)', 'petal len (cm)', 'petal width (cm)']
```



- Observations:** **150 flowers** identified by expert as belonging to 3 species: *setosa*, *versicolor*, *virginica* (i.e. labels)
- Features:** **4 cols of inputs** in [cm]: *petal length*, *petal width*, *sepal length*, *sepal width* i.e. design matrix $\mathbb{X} \in \mathbb{R}^{150 \times 4}$.

Key Sklearn Principles

1. Import appropriate estimator class, e.g.

```
from sklearn.linear_model import LinearRegression
```

2. Create an instance of the class by specifying formal / optional function parameters

```
model = LinearRegression(fit_intercept=True)
```

3. Arrange data in features matrix and label vector

Depending on algorithm:

- ❖ Create: train, validate, test data split
- ❖ Randomly shuffle rows of the data

The above can be performed with `train_test_split()` method

4. Fit the model to data using `fit()` method

Depending on the algorithm:

- ❖ Use validation set to tune *hyperparameters* (anything that you need to set rather than model in order to learn)

5. Apply model to new data:

For Supervised learning: use `predict()` method

For Unsupervised learning: use `predict()` or `transform()` method

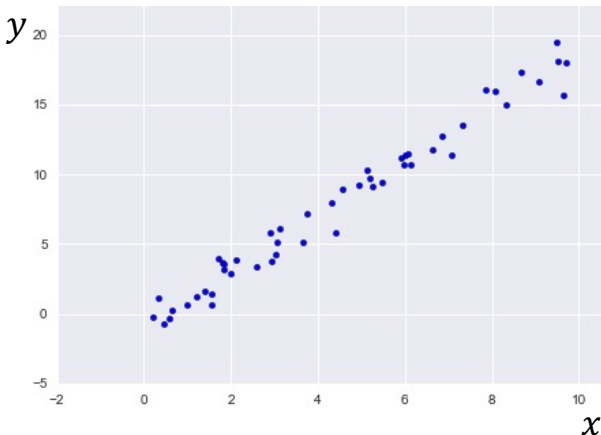
Sklearn: Regression Toy Example

Ordinary least squares Linear Regression

```
from sklearn.linear_model import LinearRegression (documentation link)
```

```
LinearRegression(fit_intercept=True)
```

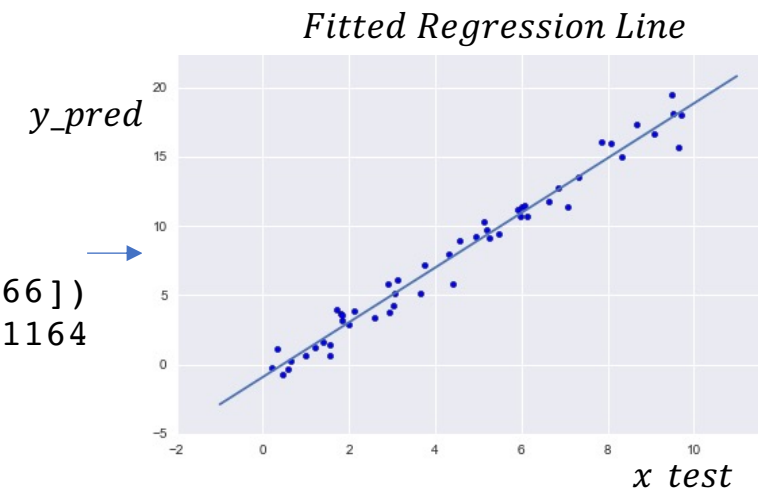
- `fit_intercept = True` (default behaviour) **offset is added automatically** (if `False` data is expected to be centered)
- **NaN values are not allowed** in the design matrix `X` (pre-process data)
- Create an instance of the class with necessary parameters: `model = LinearRegression()`
- Ensure `y` is vector of target values and `X` is a matrix of `[n_samples, n_features]`, i.e. **if 1 feature `(n, 1)`, not `(n,)`**
- Fit model to data: `model.fit(X, y)`
- Extract weights with:
 - ❖ `model.coef_` array with weights
 - ❖ `model.intercept_` `numpy.float64`
- Evaluate model based on unseen data



```
model = LinearRegression()  
x.shape # (50,)  
X = x.reshape(50, 1) # (50, 1)
```

```
→ model.fit(X, y)  
model.coef_ # array with results: array([1.9776566])  
model.intercept_ # single value: -0.9033107255311164
```

```
X_test = x_test.reshape(n, 1) # (50,) -> (50,1)  
y_pred = model.predict(X_test)
```



Error Metrics Regression Results

- Calculating model error:

Mean Absolute Error (MAE)

$$MAE = \frac{1}{k} \sum_{i=1}^k |Y_i - \hat{Y}_i|$$

Low weight to large errors
i.e. less sensitive to outliers

Mean Absolute Percentage Error (MAPE)

$$MAPE = \frac{100\%}{k} \sum_{i=1}^k \frac{|Y_i - \hat{Y}_i|}{Y_i}$$

Like MAE but normalised by true observation i.e. relative errors

Mean Squared Error (MSE)

$$MSE = \frac{1}{k} \sum_{i=1}^k (Y_i - \hat{Y}_i)^2$$

High weight to large errors.
In squared original units of Y.

Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{k} \sum_{i=1}^k (Y_i - \hat{Y}_i)^2}$$

High weight to large errors.
In original units of Y.

Depending on how you want the model to treat outliers, use either MAE or RMSE. In practice, RMSE is typically used more often.

- Use metrics sklearn library:

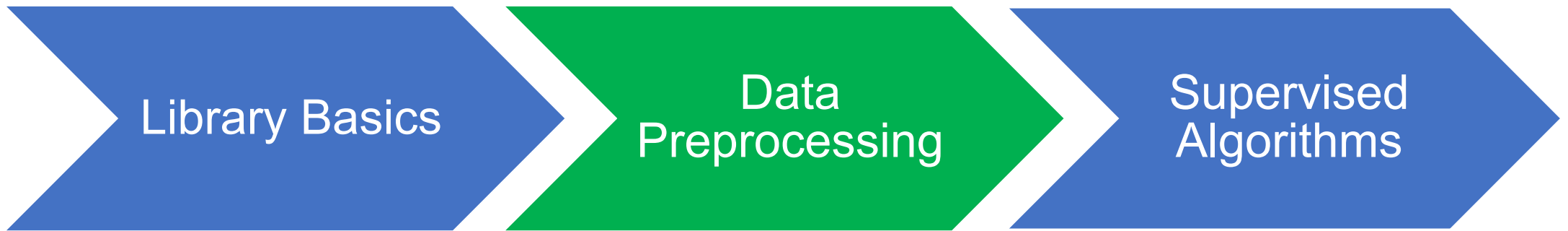
```
from sklearn import metrics
```

```
mae = metrics.mean_absolute_error(y_test, y_pred)
```

```
mse = metrics.mean_squared_error(y_test, y_pred)
```

```
rmse = metrics.mean_squared_error(y_test, y_pred, squared = False) # np.sqrt(mse)
```

Sklearn Library



Split Data: Train / Validate / Test

```
from sklearn.model_selection import train_test_split
```

- Shuffles rows of data randomly (shuffle = True by default)
- Splits data into training and testing data sets ([documentation](#))

For splitting data into train / test: (to use 70% - 30% set `test_size=0.30`)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,  
random_state = None, shuffle = True)
```

- `X` input matrix of shape `[n_samples, n_features]`
- `y` output vector of length `n_samples`
- `test_size` percentage split between training and testing data set (default 0.25 for 75%–25%)
- `random_state` seed to be used by random number generator (use any `int` value, e.g. 1234)
- `shuffle` randomly re-arranges observations in the data, keeping alignment with labels

For splitting data into train / validate / test: (to use 60% - 20% - 20% do the following)

Split the data with `train_test_split` twice: (i) first time use `test_size = 0.40`, (ii) second time use `X_test` and `y_test` and split further in half using `test_size = 0.50`.

Pre-Processing Data (1)

- Many Machine Learning algorithms are sensitive to feature's **units**, because Euclidean Distance calculations would favor a particular dimension purely because of its large units (e.g. units measured in *cm* and in *m*, the *cm* will have a large variance, *m* will hardly vary).

```
from sklearn import preprocessing
```

- In these instances data is typically pre-processed by doing either of the following:

Standardizing (I) – standardize each d feature's n^{th} observation as: $x_{n,d} \leftarrow \frac{x_{n,d} - \mu_d}{\sigma_d}$

```
X_train = np.array([[1., -1., 2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])
X_scaled = preprocessing.scale(X_train, with_mean = True, with_std = True)
X_scaled
array([[ 0. ...., -1.22....,  1.33....],
       [ 1.22....,  0. ...., -0.26....],
       [-1.22....,  1.22...., -1.06....]])
```

de-mean
scale

Doc [link](#)

Now each feature is $x \sim N(0, 1)$

```
X_scaled.mean(axis=0) # array([0., 0., 0.])
X_scaled.std(axis=0) # array([1., 1., 1.])
```

Pre-Processing Data (2)

- **Standardizing (II)** – standardizes and stores computed mean and standard deviation of the training set to allow to later to re-apply the same transformation on the testing set:

```
scaler = preprocessing.StandardScaler(with_mean = True, with_std = True).fit(X_train)
scaler.mean_
# array([1. ..., 0. ..., 0.33...])
scaler.scale_
# array([0.81..., 0.81..., 1.24...])

X_train = scaler.transform(X_train)
array([[ 0. ..., -1.22..., 1.33...],
       [ 1.22..., 0. ..., -0.26...],
       [-1.22..., 1.22..., -1.06...]])
```

de-mean scale

Doc [link](#)

- **Normalize to [0, 1] range** – convert features to be contained in [0, 1] range in order to address small standard deviation issue and preserve zero elements in sparse data.

```
mms = preprocessing.MinMaxScaler()
X_train_minmax = mms.fit_transform(X_train)
X_test_minmax = mms.fit_transform(X_test)
```

$$x_{n,d} = \frac{x_{n,d} - \min(x_d)}{\max(x_d) - \min(x_d)}$$

Pre-Processing Data (3)

- **Centering** – de-meaning the data: take away mean of feature, d , from each observation of that column, n . Repeat for each dimension (column / feature): $x_{n,d} \leftarrow x_{n,d} - \mu_d$
`scaler = preprocessing.StandardScaler(with_std = False).fit(X_train)`
- **Scaling** – dividing the (centered) data by standard deviation: obtain std dev for column, d , and divide each observation of that column by its std dev. Repeat for each column: $x_{n,d} \leftarrow \frac{x_{n,d}}{\sigma_d}$
`scaler = preprocessing.StandardScaler(with_mean = False).fit(X_train)`
- Often data needs to be **randomised** first (especially for supervised learning).
Remember that `train_test_split()` uses default parameter `shuffle = True`

Sklearn Library



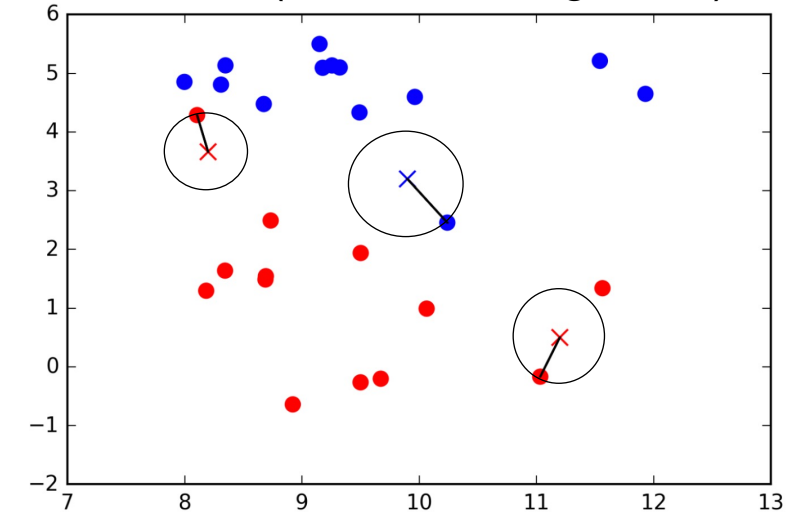
- KNN (Kanishka)
- SVM (see Homework L4)

K-Nearest Neighbours (KNN)

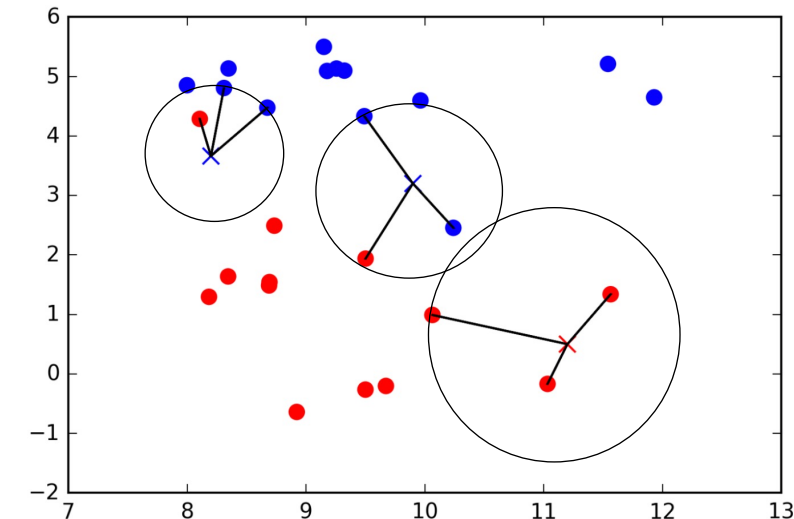
- Suppose you have a dataset comprising of N_k points in class C_k with N points in total, s.t. $\sum_k N_k = N$.
- To classify new point \mathbb{x} we draw a sphere centered on \mathbb{x} containing K points.
- We can show (see p125, Bishop) using Bayes' Theorem that Posterior Probability of class membership is:

$$p(C_k|\mathbb{x}) = \frac{p(\mathbb{x}|C_k) p(C_k)}{p(\mathbb{x})} = \frac{K_k}{K}$$
- Classify the new point as belonging to the class with the largest posterior probability i.e. to class of points which are of highest presence in the K closest points.

K = 1 (1 nearest neighbour)



K = 3 (3 nearest neighbours)



KNN with different values of K

Decision Boundaries: based on varied numbers of neighbours. K controls the degree of smoothing. Optimal K is data dependent, it should be big enough to not be affected by noise, and small enough so 1 feature won't dominate it.

Binary classification problem:
Toy data.

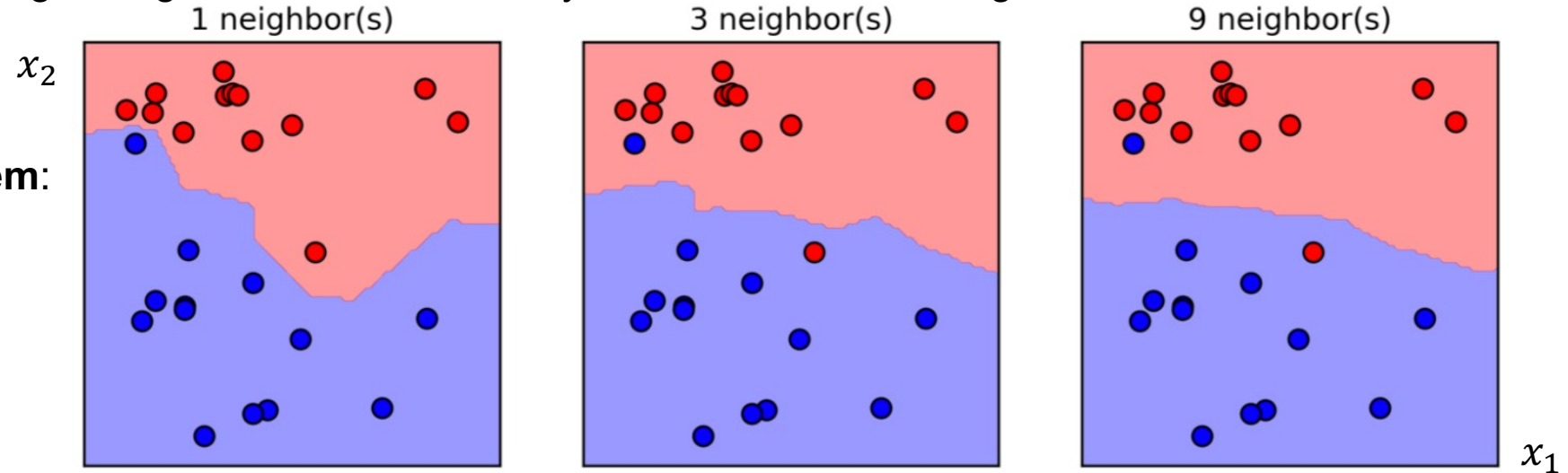


Image Source: Introduction to
Machine Learning with Python,
Guido p40

Multiclass classification :
Oil data.

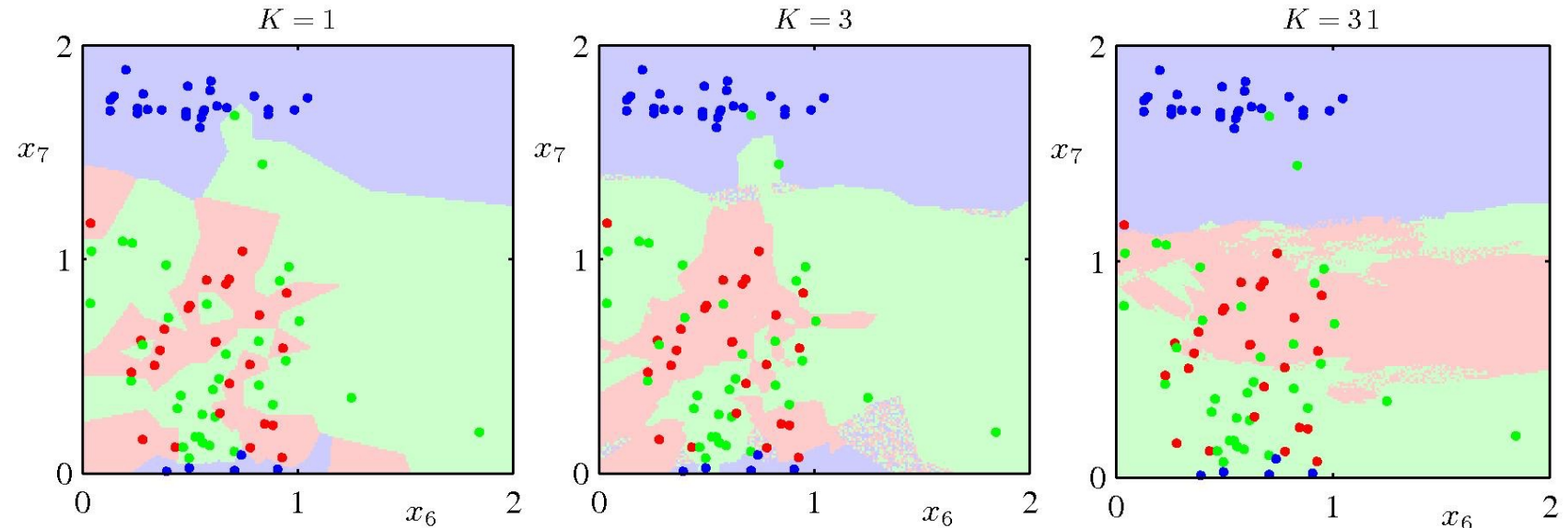


Image Source: PRML, Bishop
p126

KNN in Python

- Import estimator class (documentation [link](#))

```
from sklearn.neighbors import KNeighborsClassifier
```

- Create an instance of the class by specifying formal / optional function parameters:

```
knn = KNeighborsClassifier(n_neighbors = n) # e.g. 3
```

- Fit the model to data

```
knn.fit(X_train, y_train) # fits the algorithm in place (i.e. knn obj is changed directly)
```

- Obtain a prediction on unseen data and check accuracy

```
y_pred = knn.predict(X_test)  
knn.score(X_test, y_test) # does the following: np.mean(y_pred == y_test)
```

- Predict a single new point

```
y_pred_pt = knn.predict(x_new)
```

Read L4_LAB.pdf Q3

Answer in: L4_LAB_Questions.py

EXTRA SLIDES

Motivation – numpy Library

Arrays – containers which structure other fundamental objects in rows & columns. Typical shapes:

Vector – 1 dimensional i

Matrix – 2 dimensional $i \times j$

Cube – 3 dimensional $i \times j \times k$ etc

Arrays with Lists:

1D: $v = [1, 2, 3, 4, 5]$ single indexing e.g. $v[0]$ is 1

2D: $m = [v, v, v]$ double indexing e.g. $m[1]$ is $[1, 2, 3, 4, 5]$ and $m[1][0]$ is 1

3D: $v1 = [0.5, 1.5]$ $v2 = [1, 2]$ $m = [v1, v2]$

$c = [m, m]$ $[[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]$

triple indexing e.g. $c[1][1][0]$ is 1

Q: What is the potential problem of storing lists inside lists in the way shown e.g. for $m = [v, v, v]$?

A: Lists are mutable, so if we change an element of v , m will be affected everywhere. Need to use

```
from copy import deepcopy
```

```
m = 3 * [deepcopy(v)]
```

Arrays with Numpy: much neater representation that we can achieve with lists. Arrays are a specialized class of data structures optimised for numeric computing. Indexing them is easier than for lists.

Linear Algebra with numpy

Reminder! Multiplying 2D numpy arrays using `*` is an element-wise product instead of a dot product.

Instead, we need to use the `dot()` command (official documentation [link](#)):

- top-level NumPy function `np.dot(x, y)`
- array method `x.dot(y)`

The dot product of 2 arrays rules:

1. If either x or y is 0-D (scalar), it is equivalent to `np.multiply(x, y)` or `x*y`.

```
x = 2
y = np.array([3, 4, 5]) # (3,)
>>> array([3, 4, 5])
```

```
np.dot(x, y)
>>> array([ 6, 8, 10])
```

2. If both x and y are 1-D arrays, it is inner product of vectors. $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$

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

```
np.dot(x, y) # 32
```

3. If both x and y are 2-D arrays, it is matrix multiplication, equivalent to `np.matmul(x, y)` or `x@y`.

```
x = np.arange(6).reshape(2,3) # (2, 3)
>>> array([[0, 1, 2],
          [3, 4, 5]])
y = np.arange(6).reshape(2,3) # (2, 3)
```

```
y.T >>> array([[0, 3],
               [1, 4],
               [2, 5]])
```

✗

```
np.dot(x, y) # 2x3 2x3
>>> ValueError: shapes (2,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

Note: inner dims must agree, hence transpose array y from 2x3 into a 3x2.

✓

```
np.dot(x, y.T) # 2x3 3x2
>>> array([[5, 14],
          [14, 50]]) # 2x2
```

Manual calculation reminder:
 $0*0 + 1*1 + 2*2 = 5$

Linear Algebra with numpy

The dot product of 2 arrays rules:

4. If x is an N-D array and y is a 1-D array, it is a sum product over the **last axis of x and y** .
(In our case, we will only work with 2-D arrays at maximum).

```
x = np.ones(2) # (2,)
>>> array([1., 1.])
y = np.arange(4).reshape(2,2) # (2,2)
>>> array([[0, 1],
          [2, 3]])
```

Manual calculation reminder:

$$1*0 + 1*2 = 2$$

```
np.dot(x, y)
>>> array([2., 4.])
```

Computing outer product of two vectors (i.e. vector x vector = matrix, official documentation [link](#)):

```
x = np.ones(4) # (4,)
>>> array([1., 1., 1., 1.])
y = np.array([22, -14]) # (2,)
>>> array([22, -14])
```

```
np.outer(x, y)
>>> array([[ 22., -14.],
          [ 22., -14.],
          [ 22., -14.],
          [ 22., -14.]])
```

Computing inverse of a square matrix (e.g. variance-covariance, official documentation [link](#)):

Given a dataset of 1000 obs on 2 stock returns, one can find a 2x2 variance-cov matrix. Some calculations require finding an inverse of such a matrix. This can be done using `numpy.linalg` standard set of matrix decompositions.

```
A = np.array([[0.33, 0.22], [0.22, 0.27]])
>>> array([[0.33, 0.22],
          [0.22, 0.27]])
```

```
np.linalg.inv(A)
>>> array([[ 6.63390663, -5.40540541],
          [-5.40540541,  8.10810811]])
```

pandas – origin 2010

```
import pandas as pd
```

pandas is a play on words “**Python Data Analysis**”. This library adopts coding idioms from *numpy* with the biggest difference that it is designed to work with *heterogeneous* data.

Known for:

- Array-based functions for fast and easy data processing without `for` loops.
- Blends high-performance array-computing of *NumPy* with capabilities of *spreadsheets* and relational *databases* (such as SQL).
- Sophisticated indexing for data cleaning, easy reshaping, slicing and aggregating.

Often used in tandem with:

- Numerical computing tools (*NumPy*, *SciPy*)
- Analytical libraries (*statsmodels*, *sklearn*)
- Data visualization libraries (*matplotlib*)

Historical info:

- Emerged in 2010 by Wes McKinney (we use his book for this course: “Python for Data Analysis, 2017”).
- Many features are either part of base R (since R *dataframes* are built-in data structures) or provided by add-in packages.

pandas – (Dis)Allowed Shortcuts

Non-standard ways of selecting rows/columns.

Syntax	Example	Description
<code>df[val]</code> <code>df.val</code>	<code>df['CPI']</code> <code>df.CPI</code>	Single column selection (dict-like notation) with value . (by attribute)
<code>df[[val1,val2]]</code>	<code>df[['CPI','year']]</code>	Multiple column selection using seq of values .
<code>df[start:stop]</code> <code>df[start]</code> <code>df[start:stop, :]</code>	<code>df[0:2]</code> <code>df[0]</code> <code>df[0:2, :]</code>	Multiple row selection (finish at stop-1) Not allowed. Not allowed.

If you are just starting with Pandas, it is best to use the standard approach shown on the slide for Indexing and Slicing.

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

Creating Simple Return Data (1)

Copyright © 2021 Abramova 36

Simple Return of financial price data can be calculated in Pandas either for a single column or on the entire dataframe at once.

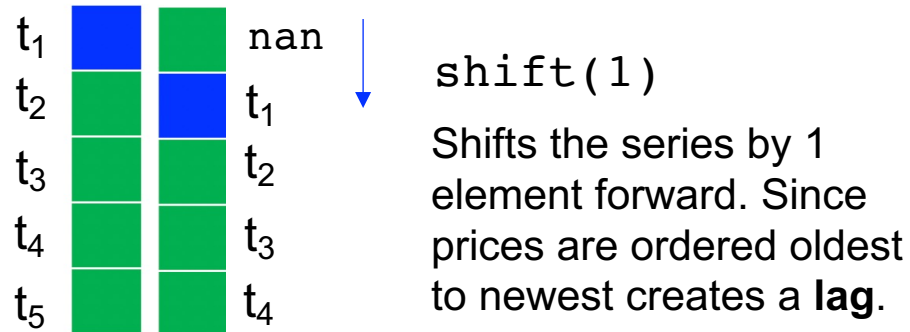
There are two methods: 1) `DataFrame.shift()` and 2) `DataFrame.pct_change()`

1) `shift()`

a) Applied on one column:

```
df["ret"] = (df["price"] - df["price"].shift(1)) / df["price"].shift(1)
```

where the shift operation has the following effect



Example:

```
(df["CPI"] - df["CPI"].shift(1)) / df["CPI"].shift(1)
>>>
0      NaN
1    0.875
2   -0.300
Name: CPI, dtype: float64
```

b) Applied on whole dataframe:

```
df_ret = (df - df.shift(1)) / df.shift(1)
```

```
>>>
year    GDP    CPI result
0      NaN    NaN    NaN    NaN
1  0.000495 -0.026316  0.875    NaN
2  0.000495 -0.135135 -0.300    NaN
```

Creating Simple Return Data (2)

2) `pct_change()`

Percentage change between the current and a prior element i.e. $\frac{P_t - P_{t-1}}{P_{t-1}}$ (but not multiplied by 100%).

a) Applied on one column:

```
df["ret"] = df["price"].pct_change()
```

Example:

```
df["ret"] = df["CPI"].pct_change()
>>>
0      NaN
1    0.875
2   -0.300
Name: CPI, dtype: float64
```

b) Applied on whole dataframe:

```
df_ret = df.pct_change()
```

```
>>>
year    GDP    CPI result
0     NaN    NaN    NaN    NaN
1 0.000495 -0.026316 0.875    NaN
2 0.000495 -0.135135 -0.300    NaN
```

Conclusion: can use either Pandas method to calculate returns.

Practice: L4_LAB.pdf Part I, Q 3. Part II Q 1-2.

Creating Log Return

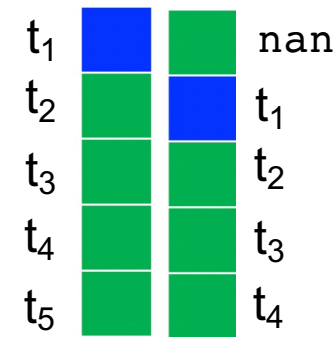
Log return of financial price data is calculated as follows:

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1})$$

where P_t is price at time step t , and P_{t-1} is price at the previous time step.

Given a Series of prices called `prices`, ordered oldest to newest, we can find log returns by:

1. Converting prices to log form using `np.log()` to obtain: `logPrices = np.log(prices)`
2. Shift time series forward 1 step `logPrices.shift(1)`



`shift(1)`

Shifts the series by 1 element forward. Since prices are ordered oldest to newest creates a **lag**.

3. Obtain the difference between `logPrice` and a shifted version of itself
`logR = logPrices - logPrices.shift(1)`

DataFrame – Plotting (1)

Plotting columns of a DataFrame with pandas method `plot()`:

- Plot **all** columns of a DataFrame on a single axis

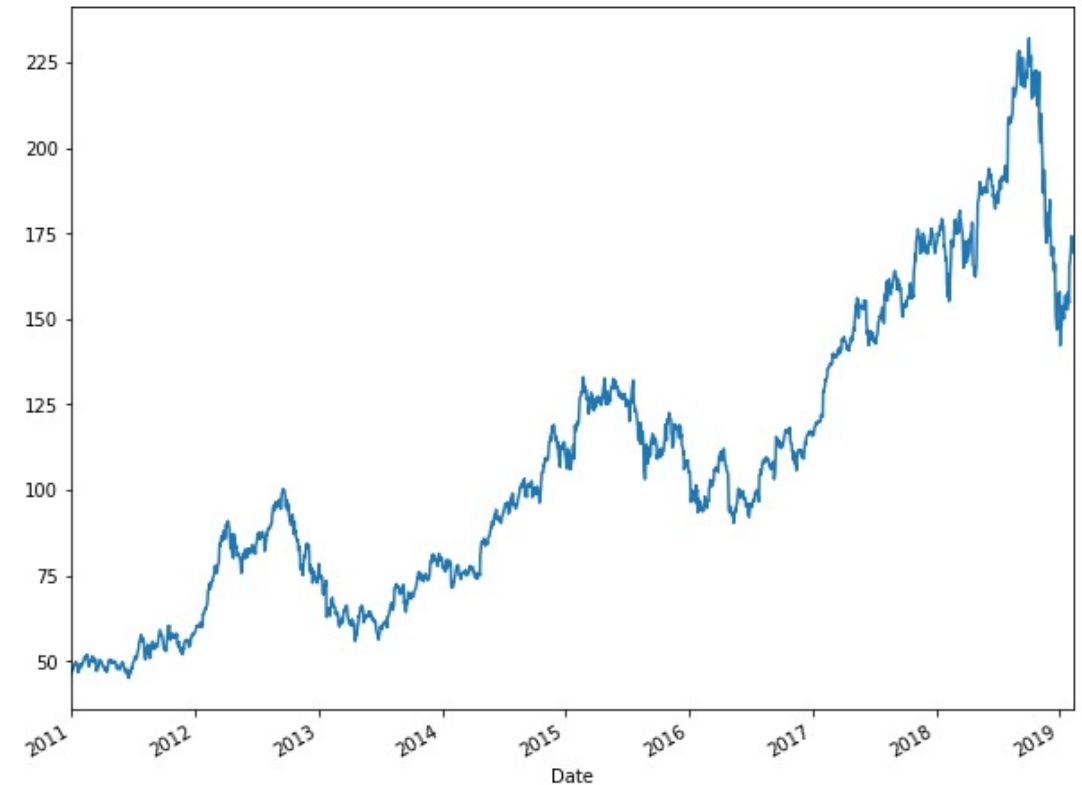
```
df.plot()
```



Plot entire dataframe contents (e.g. 3 columns):
'AAPL', 'MA_Short', 'MA_Long'

- Plot **1 column** of a DataFrame on a single axis

```
df['colName'].plot()
```

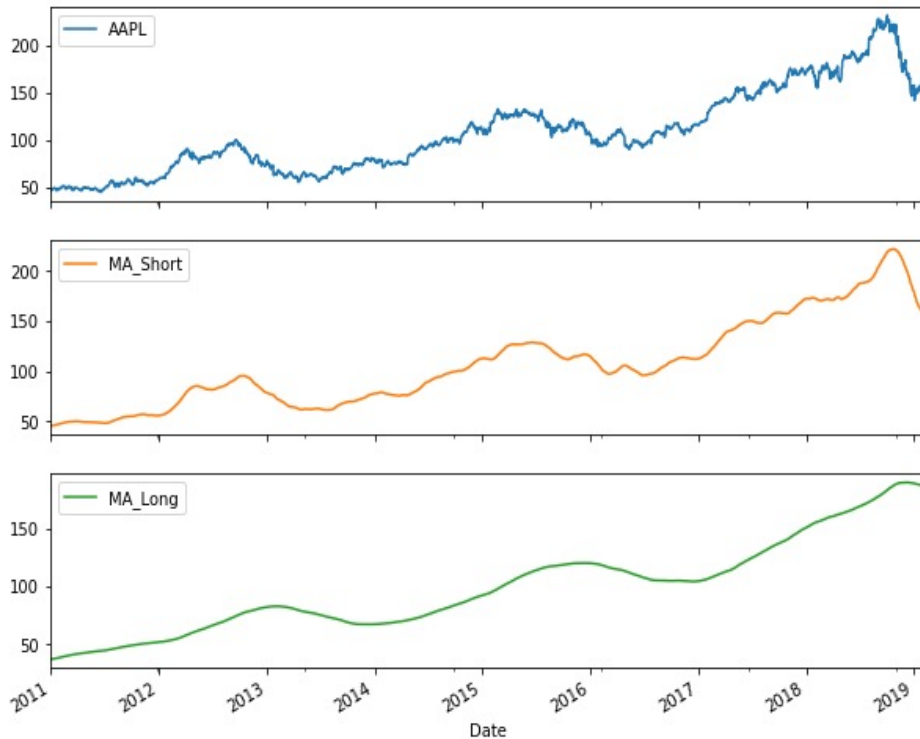


Plot only one of the columns, e.g. 'AAPL'.

DataFrame – Plotting (2)

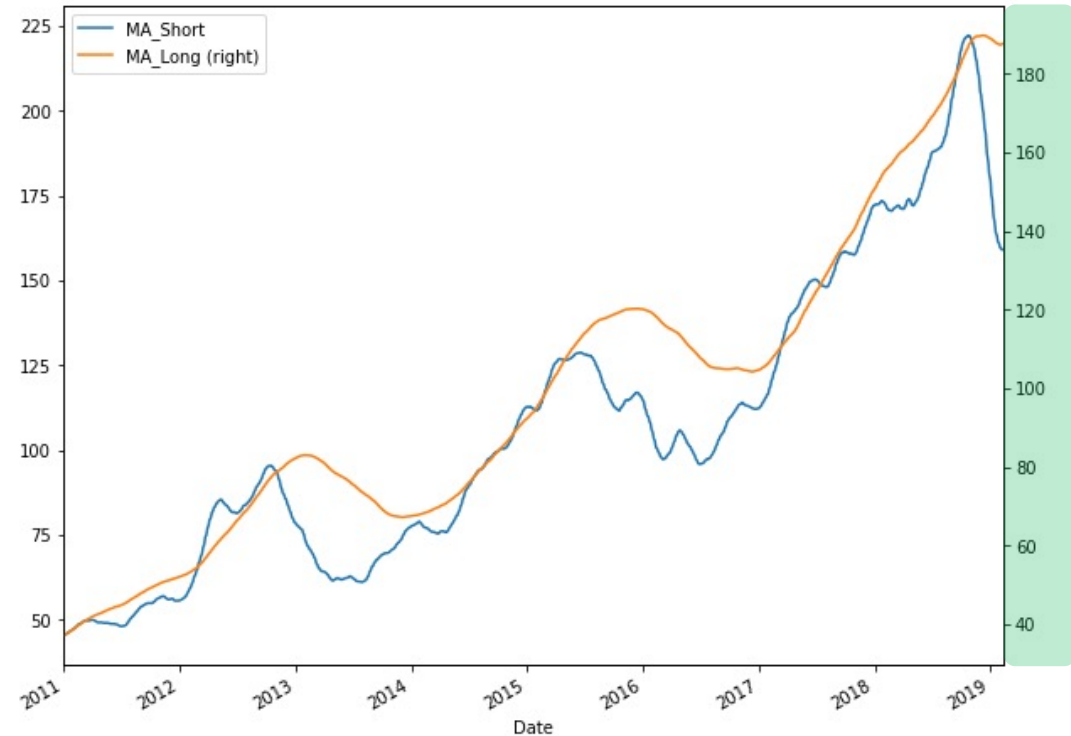
Plot each feature/column of a DataFrame as a **sub-plot** within 1 figure.

```
df.plot(subplots = True)
```



Plot **2 features on individual scales**: 1 column scale on left y-axis, 1 column scale on right y-axis.

```
df[['col1', 'col2']].plot(secondary_y = 'col2')
```



- Optional argument to control figure size: `df.plot(figsize = (m, n))` m – length; n – height.