> **Subject**: Notes on pandas library
>
> **Course:** Artificial Intelligence - LM
>
> **Department**: DISI (Department of Computer Science and Engineering) - University of Bologna, Italy
>
> **Author**: Roberto Zanolli
>
> **Sources:** https://pandas.pydata.org/docs/user_guide/index.html#user-guide

Pandas library for data science

Pandas stands for panel data.
It allows efficient management of excel like tables called **dataframes** and is mostly used for data science tasks.

To work with pandas of course we need to import it, it is a convention to alias the import as pd.

```
import pandas as pd
```

It introduces the most elementary data structure of the library: the **series**.

Series

> A **series** is a **one-dimensional labelled array**.

```
fruits = pd.Series(['apple', 'banana', 'cherry', 'date', 'elderberry'],
                   index=['a', 'b', 'c', 'd', 'e'])

print(fruits)
print(f"\nIndex: {fruits.index.tolist()}")
```

Output:

```
a         apple
b        banana
c        cherry
d          date
e    elderberry
dtype: object

Index: ['a', 'b', 'c', 'd', 'e']
```

iloc method

```
print("=== iloc examples (position-based) ===")
print(fruits.iloc[0])
```

```python
print(fruits.iloc[2])
print(fruits.iloc[1:4])    # slice from position 1 to 3
```

Output:

```
=== iloc examples (position-based) ===
apple
cherry
b    banana
c    cherry
d      date
dtype: object
```

loc method

```python
print("\n=== loc examples (label-based) ===")
print(fruits.loc['a'])
print(fruits.loc['c'])
print(fruits.loc['b':'d']) # slice from label 'b' to 'd' (inclusive!)
```

Output:

```
=== loc examples (label-based) ===
apple
cherry
b    banana
c    cherry
d      date
dtype: object
```

> **Remember:** `iloc` uses Python-style indexing (0-based, exclusive end), while `loc` uses label-based indexing (inclusive end).

Updating and filtering series

Here's a super easy example using days of the week and calories:

Creating the Series

```python
calories = pd.Series([2200, 1800, 2100, 1900, 2300, 2800, 1700],
                     index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])

print("Original Calorie Data:")
print(calories)
```

Output:

```
Original Calorie Data:
Mon    2200
Tue    1800
```

```
Wed    2100
Thu    1900
Fri    2300
Sat    2800
Sun    1700
dtype: int64
```

Filtering

```
print("\n=== FILTERING ===")

# filter days with more than 2000 calories
high_calorie_days = calories[calories > 2000]
print("Days with >2000 calories:")
print(high_calorie_days)

# filter specific days using labels
weekend_calories = calories[['Sat', 'Sun']]
print("\nWeekend calories:")
print(weekend_calories)

# filter using multiple conditions
medium_calories = calories[(calories >= 1800) & (calories <= 2200)]
print("\nMedium calorie days (1800-2200):")
print(medium_calories)
```

Output:

```
=== FILTERING ===
Days with >2000 calories:
Mon    2200
Wed    2100
Fri    2300
Sat    2800
dtype: int64

Weekend calories:
Sat    2800
Sun    1700
dtype: int64

Medium calorie days (1800-2200):
Mon    2200
Tue    1800
Wed    2100
Thu    1900
dtype: int64
```

updating

```
print("\n=== UPDATING VALUES ===")

# change Monday's calories (position 0)
calories.iloc[0] = 2000
```

```
print("After updating Monday (position 0):")
print(calories)

# change weekend calories (positions 5 and 6)
calories.iloc[5:7] = [2600, 1750]
print("\nAfter updating weekend (positions 5-6):")
print(calories)

# update every other day using positions
calories.iloc[1:6:2] = [1850, 1950, 2250]  # update Tue, Thu, Fri
print("\nAfter updating Tue, Thu, Fri (positions 1,3,5):")
print(calories)
```

Output:

```
=== UPDATING VALUES ===
After updating Monday (position 0):
Mon    2000
Tue    1800
Wed    2100
Thu    1900
Fri    2300
Sat    2800
Sun    1700
dtype: int64

After updating weekend (positions 5-6):
Mon    2000
Tue    1800
Wed    2100
Thu    1900
Fri    2300
Sat    2600
Sun    1750
dtype: int64

After updating Tue, Thu, Fri (positions 1,3,5):
Mon    2000
Tue    1850
Wed    2100
Thu    1950
Fri    2250
Sat    2600
Sun    1750
dtype: int64
```

Dataframes

> A **DataFrame** is a **two-dimensional labelled data structure** with columns of potentially different types.

```
# create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
```

```
    'Age': [25, 30, 35, 28],
    'City': ['New York', 'London', 'Tokyo', 'Paris'],
    'Salary': [50000, 60000, 70000, 55000]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

Output:

```
Original DataFrame:
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Tokyo   70000
3    Diana   28     Paris   55000
```

> It is possible to access to dataframes cells by using the notation [row,column].

iloc method

```
print("=== iloc examples (position-based) ===")
print(df.iloc[0])           # first row
print(df.iloc[2, 1])        # element at row 2, column 1 (Age=35)
print(df.iloc[1:3])         # rows 1 to 2 (exclusive end)
```

Output:

```
=== iloc examples (position-based) ===
Name        Alice
Age            25
City     New York
Salary      50000
Name: 0, dtype: object
35
    Name  Age    City  Salary
1    Bob   30  London   60000
2  Charlie 35   Tokyo   70000
```

loc method

```
print("\n=== loc examples (label-based) ===")
print(df.loc[0])            # row with index label 0
print(df.loc[2, 'City'])    # element at row 2, column 'City'
print(df.loc[1:3])          # rows 1 to 3 (inclusive end)
```

Output:

```
=== loc examples (label-based) ===
Name        Alice
Age            25
```

```
City      New York
Salary       50000
Name: 0, dtype: object
Tokyo
      Name  Age    City  Salary
1      Bob   30  London   60000
2  Charlie   35   Tokyo   70000
3    Diana   28   Paris   55000
```

The methods loc and iloc works the same as in series

Filtering and updating DataFrames

Filtering

```
print("\n=== FILTERING ===")

# filter people over 28 years old
older_than_28 = df[df['Age'] > 28]
print("People older than 28:")
print(older_than_28)

# filter specific columns
name_city = df[['Name', 'City']]
print("\nOnly Name and City columns:")
print(name_city)

# multiple conditions
high_earners_young = df[(df['Salary'] > 55000) & (df['Age'] < 35)]
print("\nHigh earners under 35:")
print(high_earners_young)
```

Output:

```
=== FILTERING ===
People older than 28:
      Name  Age    City  Salary
1      Bob   30  London   60000
2  Charlie   35   Tokyo   70000

Only Name and City columns:
      Name      City
0    Alice  New York
1      Bob    London
2  Charlie     Tokyo
3    Diana     Paris

High earners under 35:
    Name  Age    City  Salary
1    Bob   30  London   60000
```

Updating values

```
print("\n=== UPDATING VALUES ===")

# update single value - Alice's salary (row 0, column 'Salary')
df.loc[0, 'Salary'] = 52000
print("After updating Alice's salary:")
print(df)

# update multiple values - give raises to rows 1 and 3
df.loc[[1, 3], 'Salary'] = [65000, 58000]
print("\nAfter giving raises to Bob and Diana:")
print(df)

# update using iloc - change cities for first two people
df.iloc[0:2, 2] = ['Boston', 'Manchester']
print("\nAfter updating cities (positions 0-1):")
print(df)
```

Output:

```
=== UPDATING VALUES ===
After updating Alice's salary:
      Name  Age      City  Salary
0    Alice   25  New York   52000
1      Bob   30    London   60000
2  Charlie   35     Tokyo   70000
3    Diana   28     Paris   55000

After giving raises to Bob and Diana:
      Name  Age      City  Salary
0    Alice   25  New York   52000
1      Bob   30    London   65000
2  Charlie   35     Tokyo   70000
3    Diana   28     Paris   58000

After updating cities (positions 0-1):
      Name  Age        City  Salary
0    Alice   25      Boston   52000
1      Bob   30  Manchester   65000
2  Charlie   35       Tokyo   70000
3    Diana   28       Paris   58000
```

Adding a Row using concat

Adding a single row using concat

```
print("\n=== ADDING A ROW WITH CONCAT ===")

# create a new row as a DataFrame
new_person = pd.DataFrame({
    'Name': ['Evan'],
    'Age': [32],
    'City': ['Berlin'],
    'Salary': [62000]
})
```

```
# use concat to add the new row
df = pd.concat([df, new_person], ignore_index=True)
print("After adding Evan:")
print(df)
```

Output:

```
=== ADDING A ROW WITH CONCAT ===
After adding Evan:
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Tokyo   70000
3    Diana   28     Paris   55000
4     Evan   32    Berlin   62000
```

Adding multiple rows

```
print("\n=== ADDING MULTIPLE ROWS ===")

# create multiple new rows
new_people = pd.DataFrame({
    'Name': ['Fiona', 'George'],
    'Age': [29, 40],
    'City': ['Sydney', 'Toronto'],
    'Salary': [58000, 75000]
})

df = pd.concat([df, new_people], ignore_index=True)
print("After adding Fiona and George:")
print(df)
```

Output:

```
=== ADDING MULTIPLE ROWS ===
After adding Fiona and George:
      Name  Age      City  Salary
0    Alice   25  New York   50000
1      Bob   30    London   60000
2  Charlie   35     Tokyo   70000
3    Diana   28     Paris   55000
4     Evan   32    Berlin   62000
5    Fiona   29    Sydney   58000
6   George   40   Toronto   75000
```

> **Remember**: Always use `ignore_index=True` when adding rows with `concat()` to maintain clean sequential indexing.

Important: Using ignore_index

```
print("\n=== WITHOUT ignore_index ===")

# show what happens without ignore_index
```

```
temp_df = pd.concat([df, new_person])
print("Without ignore_index (duplicate indices):")
print(temp_df.tail(3))
```

Output:

```
=== WITHOUT ignore_index ===
Without ignore_index (duplicate indices):
      Name  Age     City  Salary
6   George   40  Toronto   75000
0     Evan   32   Berlin   62000
```

> If you had a custom index in the first `df` you are adding rows true, you can put a specific index using `index = ["placeholder"]` also in the second `df`.

Alternative: Adding row as Series

```
print("\n=== ADDING ROW AS SERIES ===")

# create new row as a Series
new_row = pd.Series({
    'Name': 'Hannah',
    'Age': 27,
    'City': 'Rome',
    'Salary': 53000
})

# convert to DataFrame and concat
df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
print("After adding Hannah:")
print(df.tail(3))
```

Output:

```
=== ADDING ROW AS SERIES ===
After adding Hannah:
      Name  Age     City  Salary
5    Fiona   29   Sydney   58000
6   George   40  Toronto   75000
7   Hannah   27     Rome   53000
```

Key points

- **`concat()`** combines DataFrames along a particular axis (rows by default)
- **`ignore_index=True`** resets the index to avoid duplicate indexes
- New rows must be passed as **DataFrame** (even if single row)
- Alternative to `append()` (which is deprecated)

Reading datas from file

> Pandas allows us to read csv and excel files and load them directly inside a dataframe.

> It also allows to set custom **separators** and **index**.

```
# basic csv reading
df = pd.read_csv('data.csv')
print("basic csv reading:")
print(df.head())
```

Output:

```
basic csv reading:
   id     name  age      city
0   1    alice   25  new york
1   2      bob   30    london
2   3  charlie   35     tokyo
```

The main methods to interact with the filesystem are:

- **read_csv()** - reads data from comma-separated values file into a dataframe.
- **read_excel()** - reads data from excel spreadsheet into a dataframe.
- **to_csv()** - writes dataframe to a csv file.
- **to_excel()** - writes dataframe to an excel file.

reading with custom settings

```
# reading csv with custom index and delimiter
df = pd.read_csv('data.csv', index_col='id', delimiter=',')
print("\nwith custom index (id column):")
print(df.head())
```

Output:

```
with custom index (id column):
       name  age      city
id
1     alice   25  new york
2       bob   30    london
3   charlie   35     tokyo
```

handling different delimiters

```
# for tab-separated files
df_tsv = pd.read_csv('data.tsv', delimiter='\t')

# for semicolon-separated files
df_semicolon = pd.read_csv('data.csv', delimiter=';')

# for files with custom separators
df_custom = pd.read_csv('data.txt', delimiter='|')
```

using head() method

```python
# head() shows first few rows (default 5)
print("first 3 rows:")
print(df.head(3))

print("\nfirst 5 rows (default):")
print(df.head())
```

Output:

```
first 3 rows:
        name  age      city
id
1      alice   25  new york
2        bob   30    london
3    charlie   35     tokyo

first 5 rows (default):
        name  age      city
id
1      alice   25  new york
2        bob   30    london
3    charlie   35     tokyo
```

The method **tail()** does the opposite.

using describe() method

```python
# describe() shows statistical summary for numerical columns
print("statistical summary:")
print(df.describe())
```

Output:

```
statistical summary:
             age
count   3.000000
mean   30.000000
std     5.000000
min    25.000000
25%    27.500000
50%    30.000000
75%    32.500000
max    35.000000
```

complete example with all methods

```python
# complete file reading example
df = pd.read_csv('employees.csv',
                 index_col='employee_id',  # set custom index
                 delimiter=',')           # set delimiter

print("dataset overview:")
print(f"shape: {df.shape}")  # shows (rows, columns)
```

```
print(f"columns: {df.columns.tolist()}")

print("\nfirst 5 rows:")
print(df.head())

print("\nstatistical summary:")
print(df.describe())

#print("\nfirst 10 rows:")
#print(df.head(10))
```

Output:

```
dataset overview:
shape: (100, 4)
columns: ['name', 'age', 'department', 'salary']

first 5 rows:
                 name  age department  salary
employee_id
1               alice   25         hr   50000
2                 bob   30      sales   60000
3             charlie   35 engineering   70000
4               diana   28  marketing   55000
5                evan   32         it   62000

statistical summary:
              age         salary
count  100.000000     100.000000
mean    35.200000   65230.000000
std      8.503872   14923.654321
min     22.000000   40000.000000
25%     29.000000   55000.000000
50%     35.000000   65000.000000
75%     41.000000   75000.000000
max     55.000000   95000.000000
```

Main data inspection methods and attributes

The essential pandas methods for inspecting the data are:

> **head**() - shows the first few rows to quickly preview your data
> **tail**() - displays the last few rows to see the end of your dataset
> **info**() - provides technical overview including data types and missing values
> **describe**() - gives statistical summary like mean, min, max for numerical columns

There are also attributes of dataframes we can access (not methods):

> **shape** - tells you the size of your dataset in (rows, columns) format
> **columns** - lists all the column names in your dataframe
> **index** - shows the row labels or identifiers used in your data

Examples:

```python
df = pd.DataFrame({
    'name': ['alice', 'bob', 'charlie', 'diana'],
    'age': [25, 30, 35, 28],
    'city': ['new york', 'london', 'tokyo', 'paris'],
    'salary': [50000, 60000, 70000, 55000]
}, index=['emp001', 'emp002', 'emp003', 'emp004'])

print("our dataframe:")
print(df)
```

Output:

```
our dataframe:
          name  age      city  salary
emp001    alice   25  new york   50000
emp002      bob   30    london   60000
emp003  charlie   35     tokyo   70000
emp004    diana   28     paris   55000
```

using shape

```python
print(f"dataframe shape: {df.shape}")
print(f"we have {df.shape[0]} rows and {df.shape[1]} columns")
```

Output:

```
dataframe shape: (4, 4)
we have 4 rows and 4 columns
```

using columns

```python
print(f"columns: {df.columns.tolist()}")
print(f"number of columns: {len(df.columns)}")
print("individual columns:")
for col in df.columns:
    print(f"  - {col}")
```

Output:

```
columns: ['name', 'age', 'city', 'salary']
number of columns: 4
individual columns:
  - name
  - age
  - city
  - salary
```

using index

```
print(f"index: {df.index.tolist()}")
print(f"index type: {type(df.index)}")
print("row identifiers:")
for idx in df.index:
    print(f"  - {idx}")
```

Output:

```
index: ['emp001', 'emp002', 'emp003', 'emp004']
index type: <class 'pandas.core.indexes.base.Index'>
row identifiers:
  - emp001
  - emp002
  - emp003
  - emp004
```

Data analysis methods

When working with data analysis in pandas, aggregation methods are essential for summarizing and understanding your dataset. These functions allow to transform raw data into insights by calculating statistics, counting occurrences, and grouping information logically.

using groupby

```
# Group data by person to analyze each individual separately
person_groups = calories.groupby('person')
print("Data grouped by person for individual analysis")
```

using mean

```
# Calculate average calories per person
avg_calories = calories.groupby('person')['calories'].mean()
print("Average daily calories per person:")
print(avg_calories)
```

Output:

```
Average daily calories per person:
person
Alice    2114.29
Bob      2366.67
Name: calories, dtype: float64
```

using sum

```
# Calculate total calories consumed per person
total_calories = calories.groupby('person')['calories'].sum()
print("Total calories consumed per person:")
print(total_calories)
```

Output:

```
Total calories consumed per person:
person
Alice    14800
Bob       7100
Name: calories, dtype: int64
```

using count

```
# Count number of days tracked per person
days_tracked = calories.groupby('person')['day'].count()
print("Number of days tracked per person:")
print(days_tracked)
```

Output:

```
Number of days tracked per person:
person
Alice    7
Bob      3
Name: day, dtype: int64
```

using agg

```
# Comprehensive analysis using multiple aggregation functions
summary = calories.groupby('person').agg({
    'calories': ['mean', 'max', 'min', 'sum'],
    'meals': ['mean', 'count']
})
print("Complete nutritional summary:")
print(summary)
```

Output:

```
Complete nutritional summary:
       calories                         meals
           mean   max   min    sum   mean count
person
Alice  2114.29  2800  1700  14800  3.00     7
Bob    2366.67  2500  2200   7100  3.67     3
```

Adding columns

> A DataFrame can be extended by adding new labelled columns, created from constants, lists, operations or functions.

adding a column with a constant value

```
df['Country'] = 'Unknown'
```

```python
print(df)
```

Output:

```
      Name  Age      City  Salary  Country
0    Alice   25  New York   50000  Unknown
1      Bob   30    London   60000  Unknown
2  Charlie   35     Tokyo   70000  Unknown
3    Diana   28     Paris   55000  Unknown
```

adding a column from a list

```python
df['Experience'] = [1, 4, 7, 3]
print(df)
```

Output:

```
      Name  Age      City  Salary  Country  Experience
0    Alice   25  New York   50000  Unknown           1
1      Bob   30    London   60000  Unknown           4
2  Charlie   35     Tokyo   70000  Unknown           7
3    Diana   28     Paris   55000  Unknown           3
```

adding a column using operations on existing columns

```python
df['Salary_in_k'] = df['Salary'] / 1000
print(df[['Name', 'Salary', 'Salary_in_k']])
```

Output:

```
      Name  Salary  Salary_in_k
0    Alice   50000         50.0
1      Bob   60000         60.0
2  Charlie   70000         70.0
3    Diana   55000         55.0
```

adding a column with conditional logic

```python
df['Is_Adult'] = df['Age'] >= 18
print(df[['Name', 'Age', 'Is_Adult']])
```

Output:

```
      Name  Age  Is_Adult
0    Alice   25      True
1      Bob   30      True
2  Charlie   35      True
3    Diana   28      True
```

adding a column using apply() and a custom function

```
def categorize_salary(s):
    return "High" if s > 60000 else "Normal"

df['Salary_Level'] = df['Salary'].apply(categorize_salary)
print(df[['Name', 'Salary', 'Salary_Level']])
```

Output:

```
      Name  Salary Salary_Level
0    Alice   50000       Normal
1      Bob   60000       Normal
2  Charlie   70000         High
3    Diana   55000       Normal
```

adding multiple columns with assign()

```
df = df.assign(
    Age_plus_10 = df['Age'] + 10,
    Double_Salary = df['Salary'] * 2
)

print(df)
```

Output:

```
      Name  Age      City  Salary  Country  Experience  Salary_in_k  Is_Adult
Salary_Level  Age_plus_10  Double_Salary
0    Alice   25  New York   50000  Unknown           1         50.0      True
Normal           35         100000
1      Bob   30    London   60000  Unknown           4         60.0      True
Normal           40         120000
2  Charlie   35     Tokyo   70000  Unknown           7         70.0      True
High            45         140000
3    Diana   28     Paris   55000  Unknown           3         55.0      True
Normal           38         110000
```

Removing columns

> You can remove one or more columns using the drop() method.

> Remember: by default, drop returns a **new DataFrame**, unless inplace=True is specified.

removing a single column

```
df_no_country = df.drop('Country', axis=1)
print(df_no_country)
```

Output:

```
      Name  Age      City  Salary  Experience  Salary_in_k  Is_Adult Salary_Level
Age_plus_10  Double_Salary
0    Alice   25  New York   50000           1         50.0      True       Normal
35          100000
1      Bob   30    London   60000           4         60.0      True       Normal
40          120000
2  Charlie   35     Tokyo   70000           7         70.0      True         High
45          140000
3    Diana   28     Paris   55000           3         55.0      True       Normal
38          110000
```

removing multiple columns

```
df_no_salary = df.drop(['Salary', 'Salary_in_k'], axis=1)
print(df_no_salary)
```

Output:

```
      Name  Age      City  Country  Experience  Is_Adult Salary_Level  Age_plus_10
Double_Salary
0    Alice   25  New York  Unknown           1      True       Normal           35
100000
1      Bob   30    London  Unknown           4      True       Normal           40
120000
2  Charlie   35     Tokyo  Unknown           7      True         High           45
140000
3    Diana   28     Paris  Unknown           3      True       Normal           38
110000
```

removing a column permanently (inplace)

```
df.drop('Experience', axis=1, inplace=True)
print(df)
```

Output:

```
      Name  Age      City  Salary  Country  Salary_in_k  Is_Adult Salary_Level
Age_plus_10  Double_Salary
0    Alice   25  New York   50000  Unknown         50.0      True       Normal
35          100000
1      Bob   30    London   60000  Unknown         60.0      True       Normal
40          120000
2  Charlie   35     Tokyo   70000  Unknown         70.0      True         High
45          140000
3    Diana   28     Paris   55000  Unknown         55.0      True       Normal
38          110000
```

removing a column using del

```
del df['Country']
print(df)
```

Output:

```
      Name  Age      City  Salary  Salary_in_k  Is_Adult Salary_Level  Age_plus_10
Double_Salary
0    Alice   25  New York   50000         50.0      True       Normal           35
100000
1      Bob   30    London   60000         60.0      True       Normal           40
120000
2  Charlie   35     Tokyo   70000         70.0      True         High           45
140000
3    Diana   28     Paris   55000         55.0      True       Normal           38
110000
```

splitting X and y

> In many machine learning workflows you need to separate the **features** (X) from the **target** (y).

> Usually, X contains all columns except the target, and y contains only the target column.

example dataset

```
data = {
    'Age': [25, 30, 35, 28],
    'Salary': [50000, 60000, 70000, 55000],
    'City': ['New York', 'London', 'Tokyo', 'Paris'],
    'Purchased': [0, 1, 0, 1]  # target column
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
   Age  Salary      City  Purchased
0   25   50000  New York          0
1   30   60000    London          1
2   35   70000     Tokyo          0
3   28   55000     Paris          1
```

split X and y (classic approach)

```
X = df.drop('Purchased', axis=1)
y = df['Purchased']

print("X (features):")
print(X)

print("\ny (target):")
print(y)
```

Output:

```
X (features):
   Age  Salary      City
0   25   50000  New York
1   30   60000    London
2   35   70000     Tokyo
3   28   55000     Paris

y (target):
0    0
1    1
2    0
3    1
Name: Purchased, dtype: int64
```

alternative: selecting only feature columns

```
feature_cols = ['Age', 'Salary', 'City']
X = df[feature_cols]
y = df['Purchased']

print(X)
print(y)
```

Output:

```
   Age  Salary      City
0   25   50000  New York
1   30   60000    London
2   35   70000     Tokyo
3   28   55000     Paris

0    0
1    1
2    0
3    1
Name: Purchased, dtype: int64
```