MihaiBront_LAB1_NB_1_Numpy_&_Pandas

September 29, 2024

IMPORTANT: Make a copy of this noteboook into your Drive.

1 NUMPY

https://numpy.org/doc/stable/

https://numpy.org/doc/stable/reference/index.html#reference

Is Numpy really fast?

[1]:
```python
import numpy as np
import random

# Set the size of the vectors
n = 1_000_000

# Create two random vectors
a = [random.random() for _ in range(n)]
b = [random.random() for _ in range(n)]

# Convert to NumPy arrays
np_a = np.array(a)
np_b = np.array(b)

# Plain Python multiplication
def python_multiply(x, y):
    return [a * b for a, b in zip(x, y)]

# NumPy multiplication
def numpy_multiply(x, y):
    return np.multiply(x, y)

# Time the Plain Python method
%timeit python_multiply(a, b)

# Time the NumPy method
%timeit numpy_multiply(np_a, np_b)
```

```
55.1 ms ± 1.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
2.41 ms ± 151  s per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

2 PANDAS first steps

PANDAS is the most popular library for data science (https://pandas.pydata.org/). A complete guide of this library can be found at https://pandas.pydata.org/docs/user_guide/index.html

Pandas relies on the **numpy** library for some operations, so it is convenient to import both libraries at ones.

Pandas has been integrated with matplotlib so that visualising data frames becomes an easy task with this library.

Notice that other many libraries for machine learning like **sckit-learn** as well for big data like **pySpark** have been adapted to interact/migrate into Panda's data frames. To sum up, Pandas is a standard de facto for data processing.

[2]:
```
# Pandas and numpy are already installed in Colab and included in miniconda␣
↪installations

import pandas as pd
import numpy as np
```

3 SERIES

A series is the most basic elemento of Pandas and consists of an indexed list of elements.

Note: When using functions of imported packages, Colab will help you in knowing which arguments to use and a brief description.

[3]:
```
# A series

s = pd.Series(data=[1.5,2,2,3,4,5,6])

s
```

[3]:
```
0    1.5
1    2.0
2    2.0
3    3.0
4    4.0
5    5.0
6    6.0
dtype: float64
```

[4]:
```
s.astype('float') + 3
```

[4]:
```
0    4.5
1    5.0
2    5.0
3    6.0
4    7.0
5    8.0
6    9.0
dtype: float64
```

[5]:
```
## how can we get the index and type of the series?
```

[6]:
```
# Another series

s2 = pd.Series(data=[1.5,2,3,4,5,6,7])

s3 = s + s2

s3
```

[6]:
```
0     3.0
1     4.0
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
dtype: float64
```

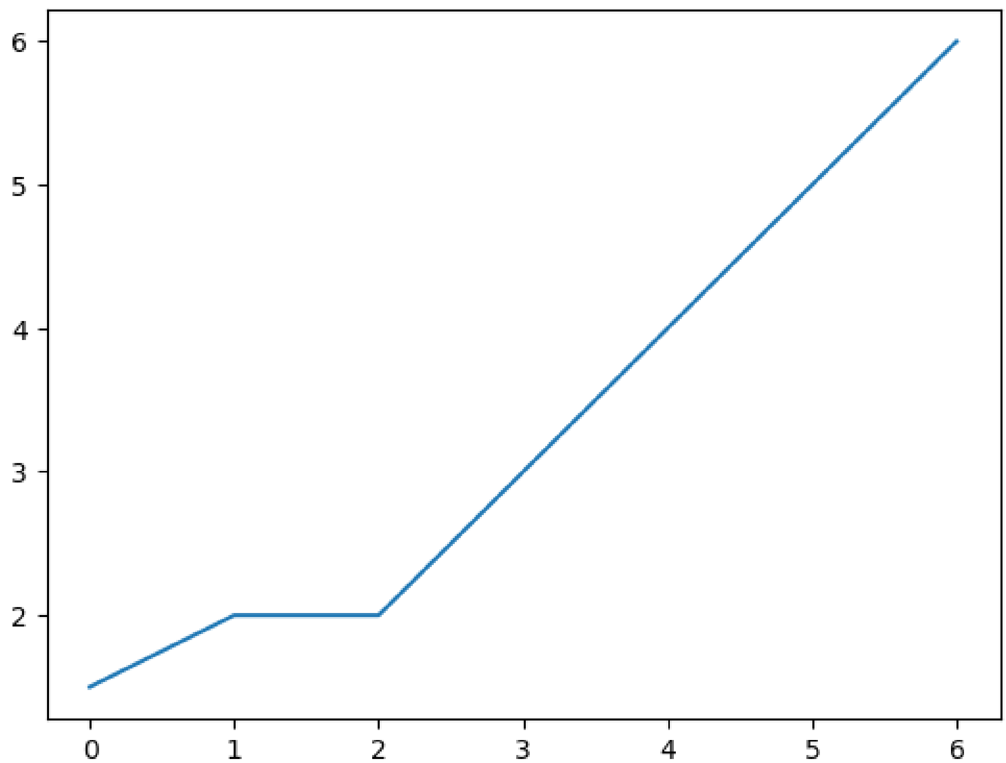
[7]:
```
## Notice that like Numpy operators are overridden to be applied over series
```

Visualising series (and data frames) is direct with the method .plot, which is a wrapper on the matplotlib.

(https://pandas.pydata.org/docs/user_guide/visualization.html)

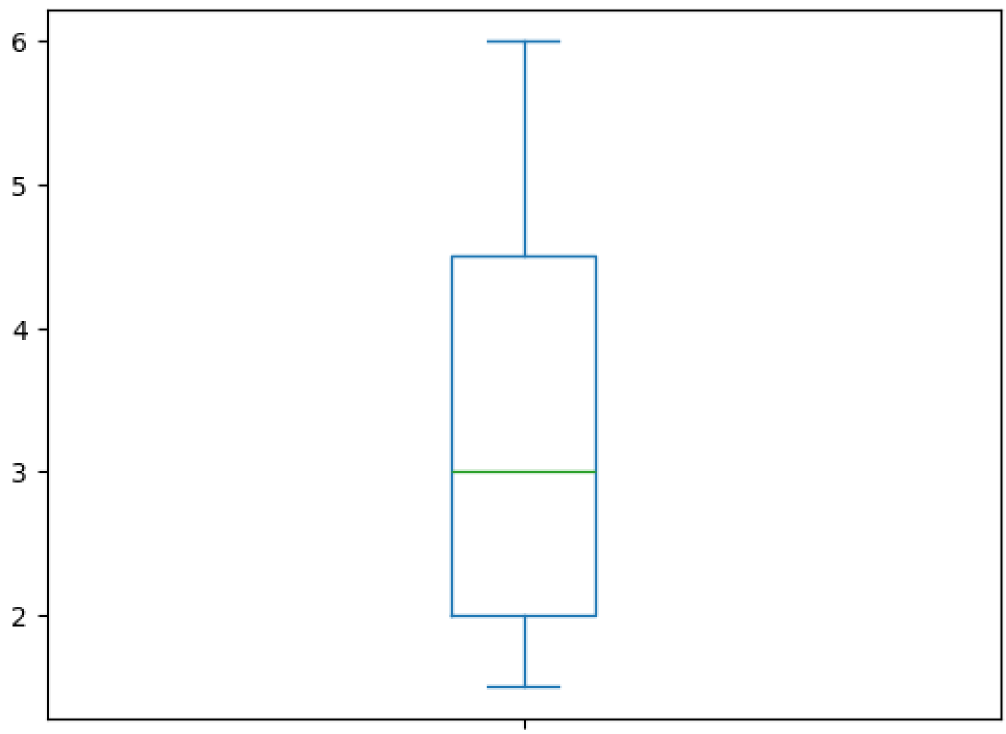
[8]:
```
# Visualización con el método plot

s.plot.line()
```

[8]:
```
<Axes: >
```



[9]:
```
s.plot.box()
```

[9]: <Axes: >



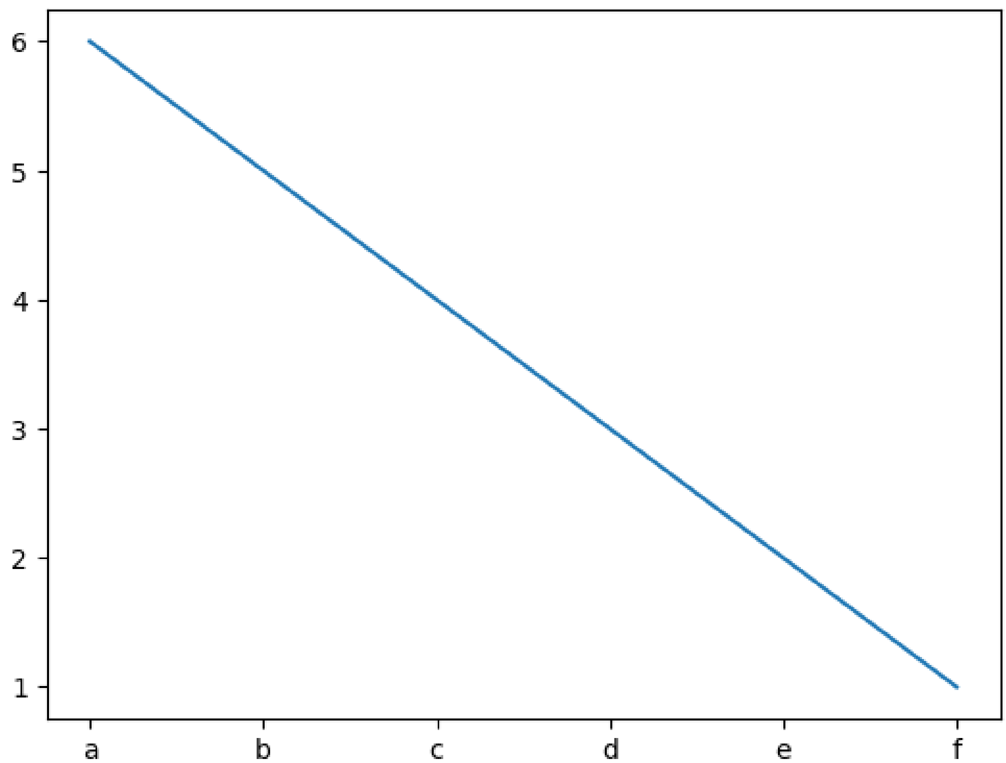
We can define the series indexes by explicitely passing the elements that will serve as index.

By default indexes are natural numbers (0 … N), but we can change the index to be any arbitrary list of numbers or strings.

[10]:
```
s = pd.Series(data=[6,5,4,3,2,1], index=['a','b','c','d','e','f'])
```

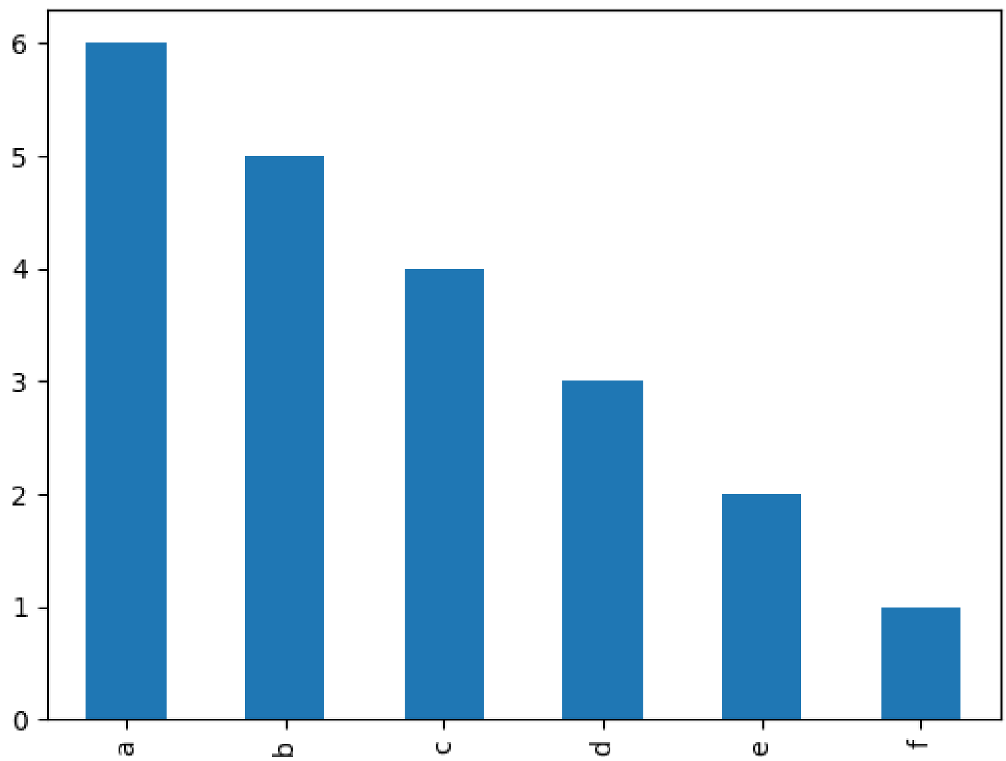
[11]:
```
s.plot.line()
```

[11]: <Axes: >



[12]:
```
s.plot.bar()
```

[12]: <Axes: >



[13]:
```
# Another way to create a series is by means a dictionary

s = pd.Series(data= {'b': 1, 'a': 0, 'c': 2})

s
```

[13]:
```
b    1
a    0
c    2
dtype: int64
```

[14]:
```
# We can easily to transform series to numpy arrays

v = s.to_numpy()

np.log(v)
```

```
C:\Users\mihaibro\AppData\Local\Temp\ipykernel_38144\3266822628.py:5:
RuntimeWarning: divide by zero encountered in log
  np.log(v)
```

[14]:
```
array([0.        ,        -inf, 0.69314718])
```

Question: Find two ways to transform series into Python lists.

[ ]:

```

```

#DATA FRAMES

DataFrames are collections of series sharing the same index (https://pandas.pydata.org/docs/user_guide/dsintro.html).

Series (columns) are organised into two axes:

When importing with Pandas, data will be automatically converted into a dataframe. The usual way is as follows:

[15]:

```python
# The first argument is required and must be a local file or a remote file (URL)

df = pd.read_csv('https://krono.act.uji.es/IDIA/airline-passengers.csv')
```

[16]:

```python
# Data frames can be easily explored with the following methods
# Statistics

df.describe()
```

[16]:

	Passengers
count	144.000000
mean	280.298611
std	119.966317
min	104.000000
25%	180.000000
50%	265.500000
75%	360.500000
max	622.000000

[17]:

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Month       144 non-null    object
 1   Passengers  144 non-null    int64
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
```

[18]:

```python
df.dtypes
```

[18]:

```
Month         object
Passengers     int64
```

```
dtype: object
```

[19]:
```
df.head(2) #first two rows
```

[19]:
```
      Month  Passengers
0   1949-01         112
1   1949-02         118
```

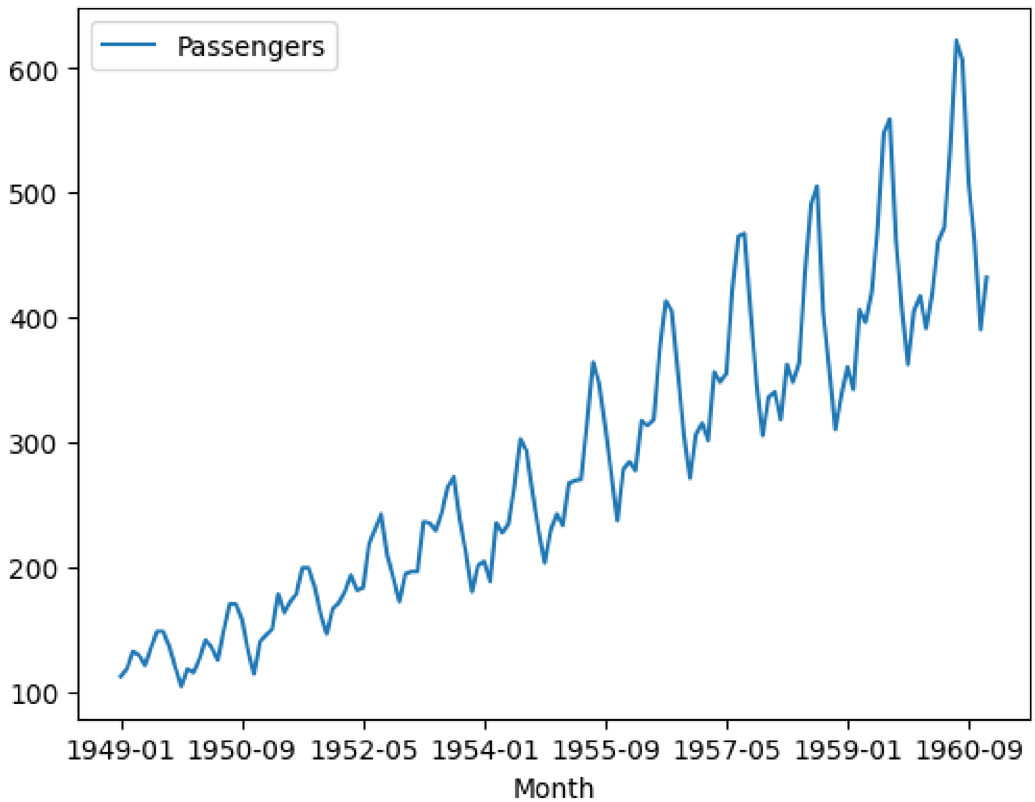
[20]:
```
df.tail(1) # last row
```

[20]:
```
        Month  Passengers
143   1960-12         432
```

[21]:
```
# If we want that a column becomes the index, we should include␣
↪index_col='Month' in read_csv

# or once loaded, with set_index('Month')

# we can also create the index with several columns (multi-index)

df = df.set_index('Month')
#df.set_index('Month', inplace=True)
```

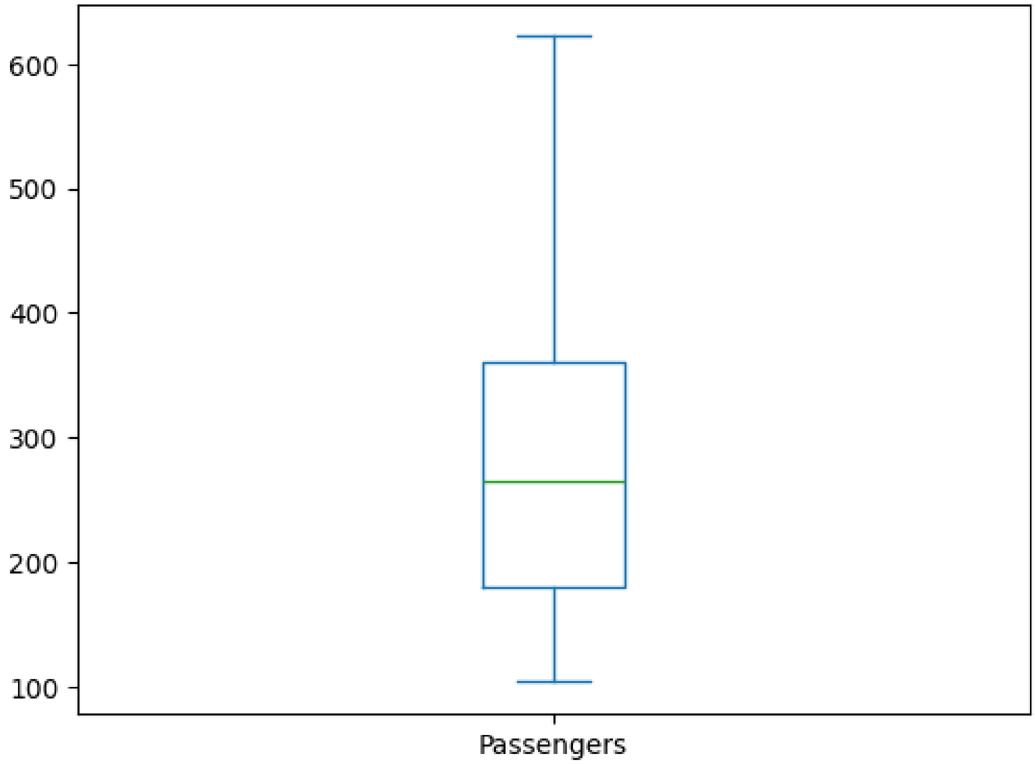
[22]:
```
df.plot.line()
```

[22]:
```
<Axes: xlabel='Month'>
```



[23]:
```
df.plot.box()
```

[23]: <Axes: >



Exercise 1: Let's create a column with a true date ("Date") from the existing "Month" column. Let's complete the date by appending the day to string of Month and tgen we will change the column's datatype with pd.to_datetime Finally, we can get rid of the column "Month" with df.drop. You can then change the index to the new column with:

```
df.index = pd.to_datetime(....)
```

[24]:
```
# Reset DataFrame for testing purpose while doing the exercise (uncomment line␣
↪below)
# df = pd.read_csv('https://krono.act.uji.es/IDIA/airline-passengers.csv')

df = df.reset_index()

## Define a new column adding to Month añadiendo the string '-1' and convert␣
↪the result to datetime

df['Date'] = pd.to_datetime(df['Month'] + "-1")#complete date from Month

df
```

[24]:
```
      Month  Passengers       Date
0   1949-01         112 1949-01-01
1   1949-02         118 1949-02-01
```

```
2    1949-03          132  1949-03-01
3    1949-04          129  1949-04-01
4    1949-05          121  1949-05-01
..       …          …       …
139  1960-08          606  1960-08-01
140  1960-09          508  1960-09-01
141  1960-10          461  1960-10-01
142  1960-11          390  1960-11-01
143  1960-12          432  1960-12-01

[144 rows x 3 columns]
```

[25]:
```
df = df.set_index("Date")
df
```

[25]:
```
              Month  Passengers
Date
1949-01-01  1949-01         112
1949-02-01  1949-02         118
1949-03-01  1949-03         132
1949-04-01  1949-04         129
1949-05-01  1949-05         121
…              …         …
1960-08-01  1960-08         606
1960-09-01  1960-09         508
1960-10-01  1960-10         461
1960-11-01  1960-11         390
1960-12-01  1960-12         432

[144 rows x 2 columns]
```

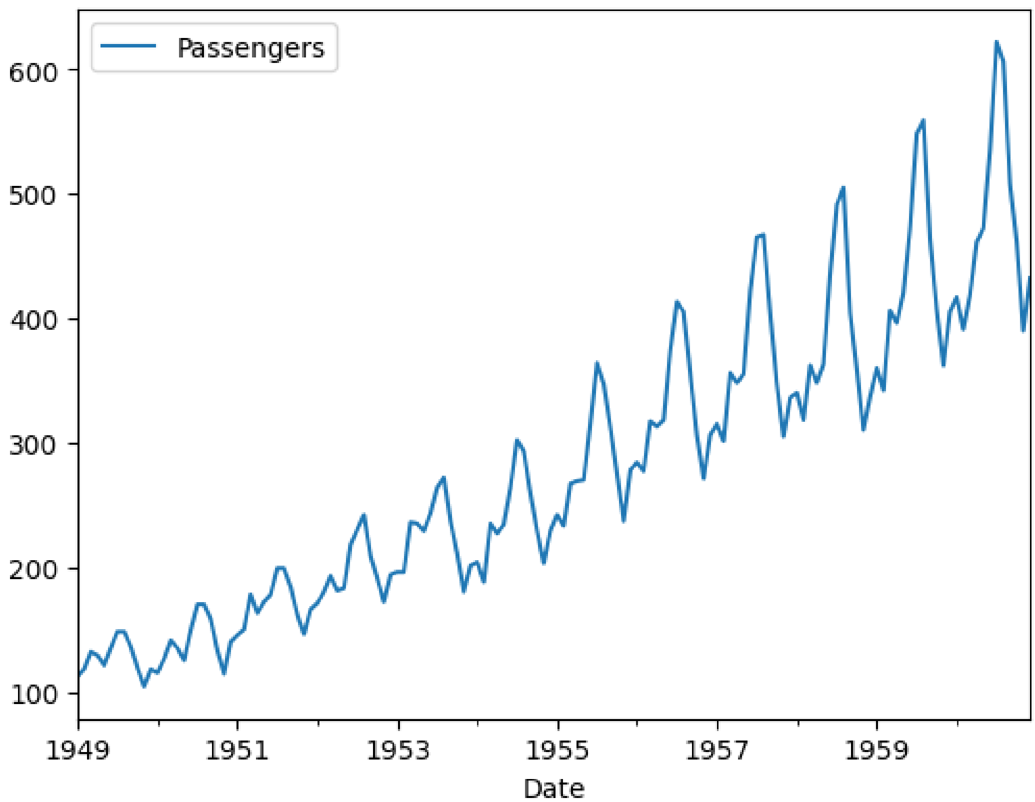
[26]:
```
df = df.drop(columns=['Month'])
df
```

[26]:
```
            Passengers
Date
1949-01-01         112
1949-02-01         118
1949-03-01         132
1949-04-01         129
1949-05-01         121
…                …
1960-08-01         606
1960-09-01         508
1960-10-01         461
1960-11-01         390
1960-12-01         432
```

[144 rows x 1 columns]

[27]:
```
#plot the new series
df.plot.line()
```

[27]: <Axes: xlabel='Date'>



Exercise 2: By querying the documentation of Pandas of the `read_csv` function, find a way to directly import the data with the (incomplete) date of `Month` as a datetime.

[28]:
```
df = pd.read_csv('https://krono.act.uji.es/IDIA/airline-passengers.csv',␣
↪parse_dates=["Month"])

df
```

[28]:

	Month	Passengers
0	1949-01-01	112
1	1949-02-01	118
2	1949-03-01	132
3	1949-04-01	129

```
4   1949-05-01         121
..         …          …
139 1960-08-01         606
140 1960-09-01         508
141 1960-10-01         461
142 1960-11-01         390
143 1960-12-01         432

[144 rows x 2 columns]
```

3.1 Creation of Dataframe (without importing data)

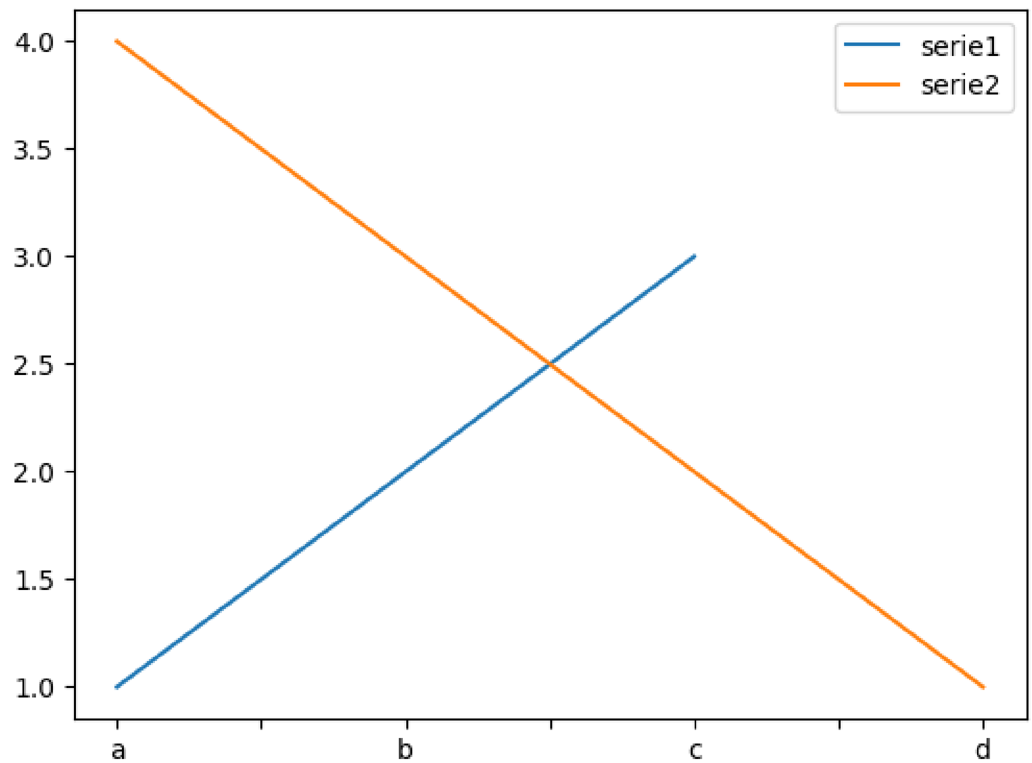
[29]:
```python
# From series to dataframes

serie1 = pd.Series([1., 2., 3.], index=['a', 'b', 'c'])
serie2 = pd.Series([4., 3., 2., 1.], index=['a', 'b', 'c', 'd'])

df2 = pd.DataFrame({'serie1': serie1, 'serie2': serie2}) ## con el argumento␣
↪index=[...] podemos cambiar los nombres de las columnas

df2
```

[29]:

	serie1	serie2
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	NaN	1.0

[30]:
```python
df2.plot.line(y=['serie1','serie2'])
```

[30]: <Axes: >



[31]:
```
# Columns and indexes are also series, which can be modified

print(df2.index, df2.columns)
```

```
Index(['a', 'b', 'c', 'd'], dtype='object') Index(['serie1', 'serie2'],
dtype='object')
```

Exercise 3: Change the names of the series to 'S1' and 'S2', and complete properly the series 'series1'. Plot the result.

Solution:

[32]:
```
df_modded = df2.copy()
df_modded.rename(columns={"serie1":"Rename1", "serie2":"Rename2"}, inplace=True)

df_modded
```

[32]:

	Rename1	Rename2
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	NaN	1.0

We can easlify find correlations between the numerical columns as follows:

[33]:
```python
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.
 ↪corr.html

df2.corr()
```

[33]:

	serie1	serie2
serie1	1.0	-1.0
serie2	-1.0	1.0

The columns of the dataframe can be selected in a similar way to what we saw with dictionaries.

[34]:
```python
## We select the series from column 'one'

df2['serie1']
```

[34]:
```
a    1.0
b    2.0
c    3.0
d    NaN
Name: serie1, dtype: float64
```

[35]:
```python
## We can also add and delete columns in a similar way to dictionary keys

df2['serie3'] = 20

print(df2)

del df2['serie3'] ## deleted forever!

print(df2)
```

```
   serie1  serie2  serie3
a     1.0     4.0      20
b     2.0     3.0      20
c     3.0     2.0      20
d     NaN     1.0      20
   serie1  serie2
a     1.0     4.0
b     2.0     3.0
c     3.0     2.0
d     NaN     1.0
```

[36]:
```python
df2['serie3'] = 20

df2.drop(columns='serie3', inplace=True) ## returns a copy, unless you use␣
 ↪inplace=True
```

[37]:
```python
df2
```

[37]:
```
   serie1  serie2
a     1.0     4.0
b     2.0     3.0
c     3.0     2.0
d     NaN     1.0
```

[38]:
```
## Creating new columns from existing ones is very intuitive

df2['serie3'] = df2['serie1'] * 10 + df2['serie2']

df2
```

[38]:
```
   serie1  serie2  serie3
a     1.0     4.0    14.0
b     2.0     3.0    23.0
c     3.0     2.0    32.0
d     NaN     1.0     NaN
```

Applying comparisons to series generates new series of boolean values.

If any of the compared values is `NaN`, will return `False`

[39]:
```
df2['serie1'] > df2['serie2']
```

[39]:
```
a    False
b    False
c     True
d    False
dtype: bool
```

Lambda functions can be applied as well to generate new columns.

[40]:
```
df2.assign(serie4 = lambda x: (x.serie1 > x.serie2).astype(float)) ## returns a␣
↪copy
```

[40]:
```
   serie1  serie2  serie3  serie4
a     1.0     4.0    14.0     0.0
b     2.0     3.0    23.0     0.0
c     3.0     2.0    32.0     1.0
d     NaN     1.0     NaN     0.0
```

[41]:
```
df2
```

[41]:
```
   serie1  serie2  serie3
a     1.0     4.0    14.0
b     2.0     3.0    23.0
c     3.0     2.0    32.0
d     NaN     1.0     NaN
```

Treating null values can be applied to all the dataframe.

[42]:
```python
# Remove rows with null values

df2.dropna() #Note: returns a copy, use inplace=True to modify the existing
 ↪dataframe
```

[42]:

	serie1	serie2	serie3
a	1.0	4.0	14.0
b	2.0	3.0	23.0
c	3.0	2.0	32.0

Question: How can you remove the columns having some null value?

Solution:

[43]:
```python
df2.dropna(axis=1)
```

[43]:

	serie2
a	4.0
b	3.0
c	2.0
d	1.0

[44]:
```python
## Fill null values with some fix value

df2.fillna(value=0) #Note: this is not a copy
```

[44]:

	serie1	serie2	serie3
a	1.0	4.0	14.0
b	2.0	3.0	23.0
c	3.0	2.0	32.0
d	0.0	1.0	0.0

The above operations return new dataframes, and therefore do not alter the content of the original dataframe. To change the content of the dataframe we can either assign the result to the same dataframe, or use the inplace argument (more efficient).

[45]:
```python
#df2 = df2.dropna()

df2.dropna(inplace=True)
```

[46]:
```python
df2
```

[46]:

	serie1	serie2	serie3
a	1.0	4.0	14.0
b	2.0	3.0	23.0
c	3.0	2.0	32.0

APPLY: This function allows you to perform any type of transformation on any of the axes using anonymous or user-defined functions. Same as assign, returns the transformed dataframe.

[47]:
```python
# The lambda function iterates over the rows (axis=1)

df2.apply(lambda x: x.serie1 * x.serie2, axis=1)
```

[47]:
```
a    4.0
b    6.0
c    6.0
dtype: float64
```

[48]:
```python
df2.apply(np.sqrt) ## applied to all the dataframe
```

[48]:

	serie1	serie2	serie3
a	1.000000	2.000000	3.741657
b	1.414214	1.732051	4.795832
c	1.732051	1.414214	5.656854

[49]:
```python
# we can generate new columns with the apply function

df2['producto'] = df2.apply(lambda x: x['serie1'] * x['serie2'], axis=1)

df2.head()
```

[49]:

	serie1	serie2	serie3	producto
a	1.0	4.0	14.0	4.0
b	2.0	3.0	23.0	6.0
c	3.0	2.0	32.0	6.0

[50]:
```python
## or we can use apply to just one specific column

df2['dos_mayor_3'] = df2['serie2'].apply(lambda x: 1 if x > 3 else 0)

df2.head()
```

[50]:

	serie1	serie2	serie3	producto	dos_mayor_3
a	1.0	4.0	14.0	4.0	1
b	2.0	3.0	23.0	6.0	0
c	3.0	2.0	32.0	6.0	0

Question: Can we use apply over a subset of columns? How can we do this?

Solution:

[51]:
```python
# For columns (axis 0), we iterate over the columns (series)

sumas = df2[['serie1', 'producto']].apply(lambda x: np.sum(x))

sumas
```

[51]:
```
serie1       6.0
producto    16.0
dtype: float64
```

3.2 EXERCISES

Exercise 4. Numpy allows you to create arrays of random numbers with the random object (see the attached links). Create a dataframe with three series and column names that you want. Then create an extra column with the sum of the previous columns (called "sum"), and another column called "sign_sum" where it indicates whether the sign of the sum is positive (1) or negative (-1).

https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html

https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

[52]:
```python
n = 100

df_base = pd.DataFrame()

df_base["DF4_S01"] = pd.Series(data=np.random.rand(n)*n-150)
df_base["DF4_S02"] = pd.Series(data=np.random.rand(n)*2*n)
df_base["DF4_S03"] = pd.Series(data=np.random.rand(n)*0.3*n)
```

[53]:
```python
df_ex4 = df_base.copy()
df_ex4["sum"] = df_ex4.sum(axis=1)

# # with masking
# df_ex4["sign_sum"] = df_ex4["sum"]
# df_ex4["sign_sum"][df_ex4["sum"]<0] = False
# df_ex4["sign_sum"][df_ex4["sum"]>=0] = True

# with apply
df_ex4["sign_sum"] = df_ex4.apply(lambda x: x["sum"]>=0, axis=1)
df_ex4
```

[53]:

	DF4_S01	DF4_S02	DF4_S03	sum	sign_sum
0	-78.527134	19.714662	19.273719	-39.538754	False
1	-145.236812	85.893100	13.473935	-45.869778	False
2	-64.002996	92.919915	25.966140	54.883059	True
3	-63.298594	83.048948	15.792530	35.542884	True
4	-71.449857	172.668618	7.318345	108.537107	True
..	…	…	…	…	…
95	-88.204886	109.847349	2.835227	24.477690	True
96	-102.962792	2.968864	21.315160	-78.678768	False
97	-91.529286	162.106340	16.912270	87.489324	True
98	-68.526895	164.051381	3.353997	98.878483	True
99	-148.675562	152.915443	23.752173	27.992054	True

[100 rows x 5 columns]

Exercise 5. Using the *apply* function and the previous dataframe, normalize the series so that they always add up to one.

[54]:
```
df_ex5 = df_base.copy()

# totals per column
totals = df_ex5.sum()
print(f"Before normalization, sum is: \n{totals}\n")

# divide series per totals
df_ex5 = df_ex5.div(totals)

# proof
print(f"After normalization, sum is: \n{df_ex5.sum()}")
```

```
Before normalization, sum is:
DF4_S01   -10437.956587
DF4_S02    10736.254416
DF4_S03     1717.215857
dtype: float64

After normalization, sum is:
DF4_S01    1.0
DF4_S02    1.0
DF4_S03    1.0
dtype: float64
```

Exercise 6. With the `shift` function you can shift a series according to the specified period. From the passenger dataframe (df), add a shifted series and calculate the **autocorrelation** between the original and the shifted series.

[55]:
```
df_ex6 = pd.read_csv('https://krono.act.uji.es/IDIA/airline-passengers.csv',␣
 ↪parse_dates=["Month"])

# add the shited series as a new column
df_ex6["Shifted"] = df_ex6["Passengers"].shift(15).fillna(value=0)

# calculate the correlation
df_ex6[["Passengers", "Shifted"]].corr()
```

[55]:

	Passengers	Shifted
Passengers	1.000000	0.843229
Shifted	0.843229	1.000000

Note: You can use the random functions of numpy to generate the necessary data for the exercises.

[56]:
```
# randint between 100 and 300
100 + np.random.randint(200, size=(5,))
```

[56]: array([170, 218, 154, 213, 114], dtype=int32)

[57]:
```
#rand between 100 and 300
100 + np.random.rand(5) * 200
```

[57]: array([217.65662638, 164.12919735, 268.43824731, 232.17244738,
        138.91256137])

4 CHALLENGE EXERCISE

This exercise is optional and it is not necessary to include it in the weekly deliverable. We will work all together to find out the solution.

If you don't have reported your skills, please fill in the following Google form:

https://forms.gle/PkkagVDxPwrozg4M6

Problem: Given the results of the questionaire about the students' Data Science skills (see link below), you have to import and transform the results reported of the Google Sheet to obtain similar visualizations to those provided by Google Forms. Additionally, you have to calculate the correlations between the different skills.

https://docs.google.com/spreadsheets/d/1-Cw9X9zLtEVuCn611cDltmFUSAdI6BR7AkMXYDMOg6o/edit?usp=

TASKS: 1. Connect and import data from a shared Google Sheet (or download it as a CSV file) 2. Create a new dataframe with the imported data 4. Calculate statistics and visualize the summarised data

4.1 My approach:

My approach is focusing on getting a series for every skill, where its value will be true if the user (rows of the dataframe) knows that skill.

First, i will define a method

This method will get as parameters: - A `pd.Series` object, a column from the source CSV file, the rows of which will contain, sepparated by comas, the skills the user has selected for that branch of knowledge - A separator, which identifies how the skills are separated (defaults as ",") - A list of exclusions, which i called "`excludeForConvenience`" which will help sepparating the skills from each row (for instance, there is one branch of knowledge that contains one skill which includes a clarification "(XML, JSON)" which will mess up the sepparation of the data in that cell, since it contains a coma) - A boolean parameter called `replicate` that will overwrite the passed parameter (defaulting to false)

The method will: 1. Create a DataFrame 2. Take the pd series and finde which different skills they appear (with help of the passed separaor) 3. Create a new column for each different skill and fill each one of its rows with True or False depending on weather it appeared in the selection for that user or not.

[58]:
```
def bIsOptionInStrList(series: pd.Series, sep: str = ",", excludeForConvenience:
 ↪ list | None = None, replicate: bool = False):
```

```
  df_data = pd.DataFrame()
  df_data["data"] = series.copy()

  if excludeForConvenience is not None:
    for element in excludeForConvenience:
      print(f"Removing {element} from {series.name}...")
      df_data["data"] = df_data["data"].apply(lambda x: x.replace(element, ""))

  df_data["data"] = df_data["data"].apply(lambda x: x.replace(" ", ""))

  comma_sep = df_data["data"].str.split(",")
  unique_itm = set(item for sublist in comma_sep for item in sublist)

  for item in unique_itm:
    df_data[item] = comma_sep.apply(lambda x: 1 if item in x else 0)

  if not replicate:
    df_data.drop(columns=["data"], inplace=True)
  return df_data
```

[59]:
```
#1. Connect and import data from a shared Google Sheet (or download it as a CSV␣
↪file)
df_skills = pd.read_csv(".res/DataScience Skills - Respuestas de formulario 1.
↪csv").drop(columns=["Dirección de correo electrónico"]).fillna("")

#1.1 One dataframe per
df_data = bIsOptionInStrList(df_skills["Data Processing/Engineering"])
df_ml = bIsOptionInStrList(df_skills["Machine Learning"])
df_dl = bIsOptionInStrList(df_skills["Deep Learning"])
df_bd = bIsOptionInStrList(df_skills["Big Data"], excludeForConvenience=["␣
↪(XML, JSON)"])
df_ncd = bIsOptionInStrList(df_skills["Non conventional data"])
df_oth = bIsOptionInStrList(df_skills["Others"])

df_skills_sep=pd.concat([df_skills[["Marca temporal"]], df_data, df_ml, df_dl,␣
↪df_bd, df_ncd, df_oth], axis=1)
df_skills_sep = df_skills_sep.drop(columns=[""], axis=1) # dropping columns␣
↪with no name, appearing between concatenated for some reason
```

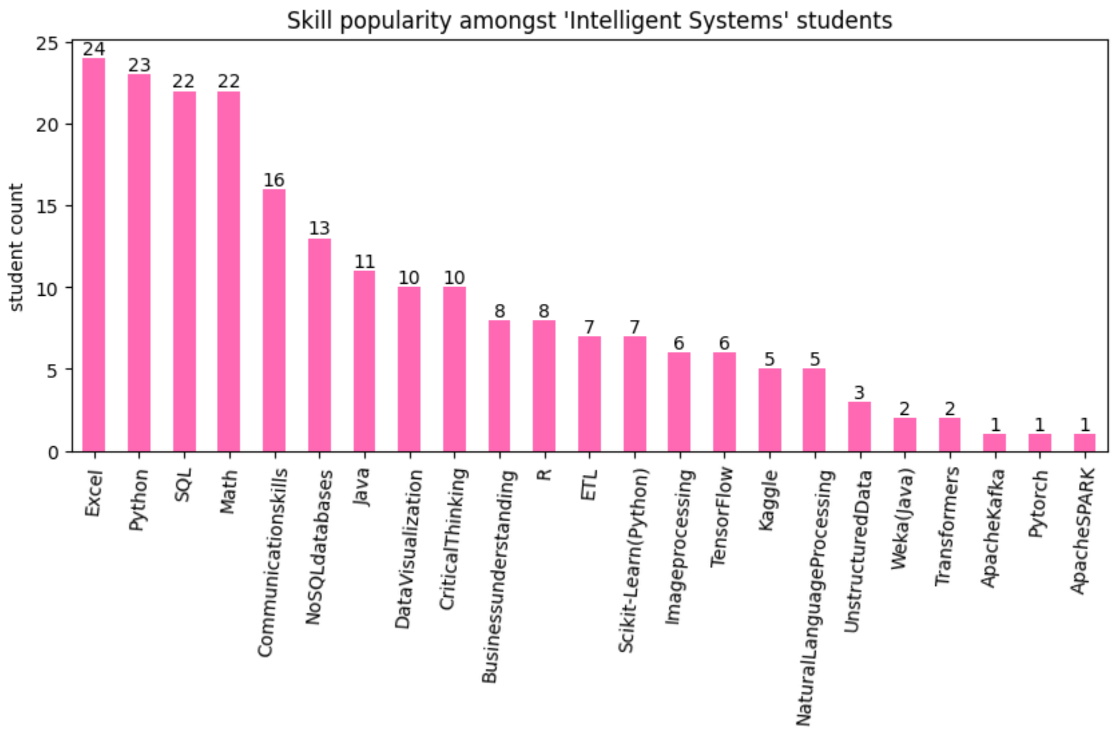
```
Removing  (XML, JSON) from Big Data…
```

For representing the data:

- One bar graph of the coverage for each skill (also, sorted from most popular to less)

[60]:
```
ax=df_skills_sep.iloc[:, 1:].sum().sort_values(ascending=False).plot.
↪bar(title="Skill popularity amongst 'Intelligent Systems' students",␣
↪ylabel="student count",
```

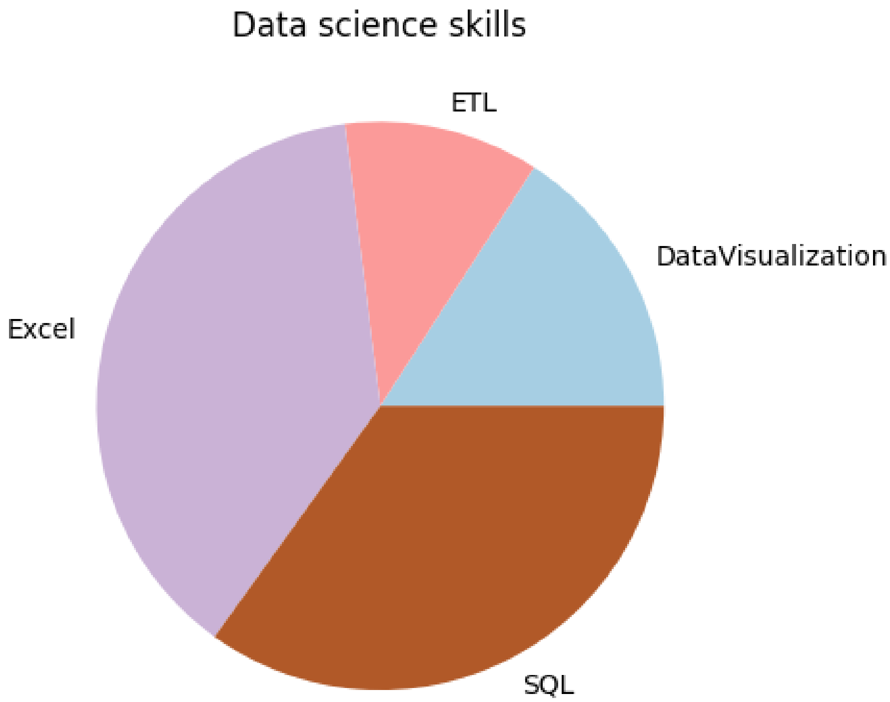
```
↪figsize=(10,4), color="hotpink", rot=85)
_ = ax.bar_label(ax.containers[0], color="black")
```



- A sample of a pie chart for Data Science Skills

[61]:
```
df_data.sum().plot.pie(title="Data science skills", colormap="Paired")
```

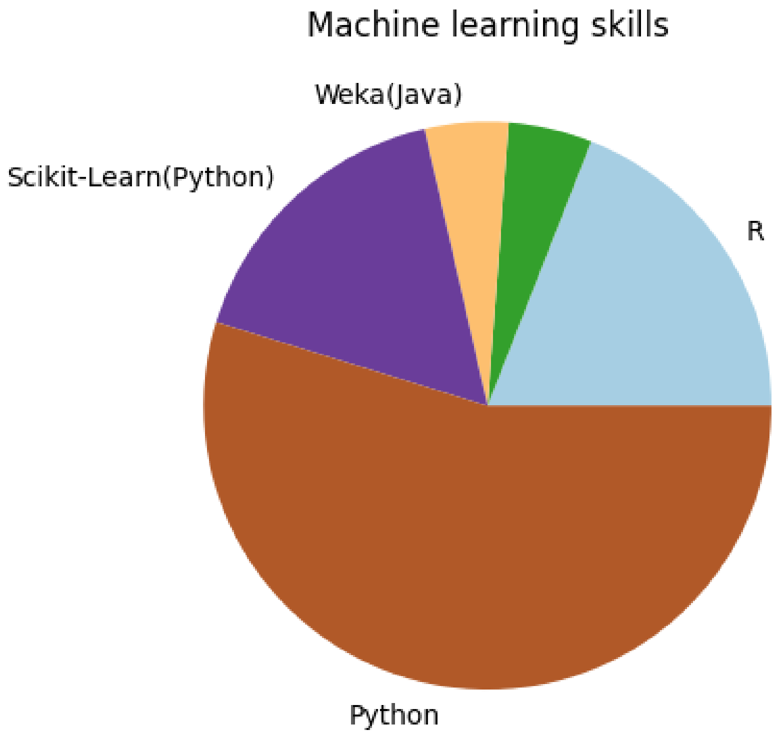
[61]: <Axes: title={'center': 'Data science skills'}>



- And another one for Machine learning Skills

```
[62]: df_ml.sum().plot.pie(title="Machine learning skills", colormap="Paired")
```

```
[62]: <Axes: title={'center': 'Machine learning skills'}>
```


Machine learning skills
Weka(Java)
Scikit-Learn(Python)
R
Python