Lecture 4: Statistics

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA
UNIVERSITET

Content

Pandas

Pandas

Pandas: Problem

Numpy arrays can store, operate on and, with matplotlib, plot data.

But how can we **load the data** into Python?

Data often comes in **spreadsheet-like format**, can we represent that in Python?

We use a package called **pandas**?

Use shorthand import by convention:

```
import pandas as pd
```

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
https://pandas.pydata.org/pandas-docs/stable/user_guide/cookbook.html

Pandas: Data objects

Pandas contain two principal data containing objects.

Series

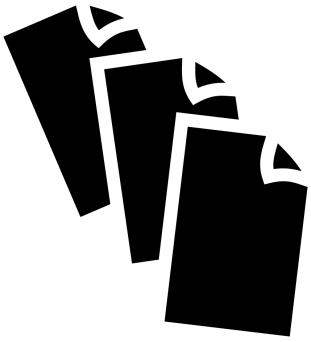
Represents a 1D array of data.

DataFrame

Represents a 2D array of data.
Spread-sheet/excel type data structure.
Unlike Numpy arrays, the columns typically have a label that we can use to index.

Some operators and functions behave similarly to Numpy. Others behave differently.

Built in functions specifically for data processing.



https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html

Pandas: Creation, indexing and selection

Dataframes can be initialized in many different ways.

E.g. by giving a numpy array as input and then setting the names of the columns afterwards.

A column can be retrieved as a Series by indexing into the dataframe using the name of a column.

Operations such removing missing values can be performed on data from a single column like this.

https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html

```python
import numpy as np
import pandas as pd

# Generate a Numpy array with random numbers
rng = np.random.default_rng(seed=0)
X = rng.integers(0, 6, size=(4, 2))

# Createa DataFrame from numpy array
df = pd.DataFrame(X)

# Set the names of the columns in the
    DataFrame
df.columns = ['Group A', 'Group B']
print(df)

# Get Group A as a Series
A = df['Group A']
print(A)
```

```
>>    Group A  Group B
>> 0        5        3
>> 1        3        1
>> 2        1        0
>> 3        0        0
>> 0    5
>> 1    3
>> 2    1
>> 3    0
```

Pandas: Operators

Mathematical operations are available also for DataFrame and Series.

- `+` Add
- `-` Subtract
- `*` Multiply
- `**` Power
- `/` Divide
- `//` Integer division
- `%` Modulus

```python
# Create datframe of random integers
import numpy as np
import pandas as pd
rng = np.random.default_rng(seed=0)
X = rng.integers(0, 9, size=(4, 2))
df = pd.DataFrame(X)
df.columns = ['Group A', 'Group B']

# Original dataframe
print(df)

# Dataframe after adding 2 to first column
df['Group A'] = df['Group A'] + 2
print(df)
```

```
>>    Group A  Group B
>> 0        7        5
>> 1        4        2
>> 2        2        0
>> 3        0        0
>>    Group A  Group B
>> 0        9        5
>> 1        6        2
>> 2        4        0
>> 3        2        0
```

Pandas: Aggregates

Aggregate functions are similar to the ones in Numpy.

- `count`
- `first`
- `last`
- `mean`
- `median`
- `min`
- `max`
- `sum`
- `prod`
- `std`
- `var`

```
# Create datframe of random integers
import numpy as np
import pandas as pd
rng = np.random.default_rng(seed=0)
X = rng.integers(0, 9, size=(4, 2))
df = pd.DataFrame(X)
df.columns = ['Group A', 'Group B']

# Original dataframe
print(df)

print("Count and mean values:")
print(df.count())
print(df.mean())
```

```
>>    Group A  Group B
>> 0        7        5
>> 1        4        2
>> 2        2        0
>> 3        0        0
>> Count and mean values:
>> Group A    4
>> Group B    4
>> dtype: int64
>> Group A    3.25
>> Group B    1.75
>> dtype: float64
```

Pandas: Describe

The function `describe()` can be used to quickly generate a summary of a dataset.

```
# Create datframe of random integers
import numpy as np
import pandas as pd
rng = np.random.default_rng(seed=0)
X = rng.integers(0, 9, size=(4, 2))
df = pd.DataFrame(X)
df.columns = ['Group A', 'Group B']

# Original dataframe
print(df)

# Create a description of the dataframe
d = df.describe()
print(d)
```

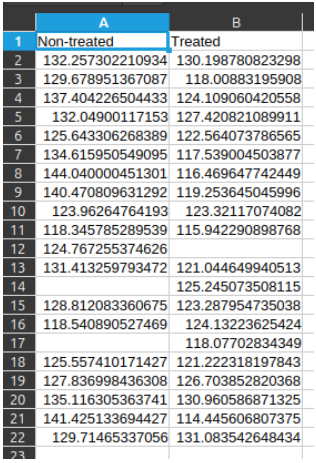
```
>>    Group A  Group B
>> 0         7         5
>> 1         4         2
>> 2         2         0
>> 3         0         0
>>          Group A   Group B
>> count  4.000000  4.000000
>> mean   3.250000  1.750000
>> std    2.986079  2.362908
>> min    0.000000  0.000000
>> 25%    1.500000  0.000000
>> 50%    3.000000  1.000000
>> 75%    4.750000  2.750000
>> max    7.000000  5.000000
```

Pandas: Handling of missing data

Data can be missing from a data set for several reasons.

- The instrument produced an erroneous output.
- A patient did not show up for follow up measurements.
- Data files can become partly corrupted.
- If data is entered manually entries may be skipped by accident.

Numpy has **no built in utilities** to handle this. But **Pandas does**!



	A	B
1	Non-treated	Treated
2	132.257302210934	130.198780823298
3	129.678951367087	118.00883195908
4	137.404226504433	124.109060420558
5	132.04900117153	127.420821089911
6	125.643306268389	122.564073786565
7	134.615950549095	117.539004503877
8	144.040000451301	116.469647742449
9	140.470809631292	119.253645045996
10	123.96264764193	123.32117074082
11	118.345785289539	115.942290898768
12	124.767255374626	
13	131.413259793472	121.044649940513
14		125.245073508115
15	128.812083360675	123.287954735038
16	118.540890527469	124.13223625424
17		118.07702834349
18	125.557410171427	121.222318197843
19	127.836998436308	126.703852820368
20	135.116305363741	130.960586871325
21	141.425133694427	114.445606807375
22	129.71465337056	131.083542648434
23		

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html

Pandas: NaN values

Floating point numbers have a few special, **invalid, numbers**.

- `nan`
- `inf`
- `-inf`

They can't be used with mathematical operations an functions since they **poison the results**.

We want to **remove them from our data** before any further processing.
Integers do not have any equivalent to the `nan`, `inf` or `-inf` values.

```python
import numpy as np

# Create an array and divide by zero
a = np.array([9, 0, -3])
b = a / 0.0
print("This array contains only invalid values")
print(b)

# Create a new array with values
c = np.array([7, -4, 2])
print("This array contains normal values")
print(c)

# Add the invalid array to the new array
d = c + b
print("Add the two arrays.")
print("Resulting array contains invalid values")
print(d)
```

```
>> This array contains only invalid values
>> [ inf  nan -inf]
>> This array contains normal values
>> [ 7 -4  2]
>> Add the two arrays.
>> Resulting array contains invalid values
>> [ inf  nan -inf]
```

Pandas: Operating on null values

Pandas has several ways of **identifying** and **handling missing values**.

`df.isnull()`	Generates Boolean mask.
`df.notnull()`	Opposite of `isnull()`.
`df.dropna()`	Get filtered df.
`df.fillna(...)`	Get copy of data with filled or imputed missing values.

```
import numpy as np
import pandas as pd
# Create random data and set one value to nan
rng = np.random.default_rng(seed=0)
X = rng.uniform(0, 9, size=(4, 2))
X[0,0] = np.nan
# Create dataframe
df = pd.DataFrame(X)
df.columns = ['Group 1', 'Group 2']
print("## Original dataframe")
print(df)
# Remove rows containting a nan
df_no_nan = df.dropna()
print("## Dataframe without nan")
print(df_no_nan)
```

```
>> ## Original dataframe
>>      Group 1   Group 2
>> 0        NaN  2.428080
>> 1   0.368762  0.148749
>> 2   7.319432  8.214800
>> 3   5.459722  6.565469
>> ## Dataframe without nan
>>      Group 1   Group 2
>> 1   0.368762  0.148749
>> 2   7.319432  8.214800
>> 3   5.459722  6.565469
```

Pandas: Combining datasets

Data sets can be joined together, *e.g.* by **concatenation**.

Note that nan values will be added wherever there is *e.g.* a missing column of data in one dataframe.

DataFrames can be joined in complex ways using **set operators**.

```
1  import numpy as np
2  import pandas as pd
3  #Create random data
4  rng = np.random.default_rng(seed=0)
5  X1 = rng.normal(0.0, 1.0, size=(3, 2))
6  X2 = rng.normal(0.0, 1.0, size=(2, 3))
7
8  # Create dataframes from andom data
9  df1 = pd.DataFrame(X1)
10 df2 = pd.DataFrame(X2)
11 df1.columns = ['Group 1', 'Group 2']
12 df2.columns = ['Group 1', 'Group 2', 'Group 3']
13
14 # Concatenate two dataframes
15 df3 = pd.concat([df1, df2], ignore_index=True, sort=
       False)
16 print(df3)
```

```
1  >>      Group 1   Group 2   Group 3
2  >> 0  0.125730 -0.132105       NaN
3  >> 1  0.640423  0.104900       NaN
4  >> 2 -0.535669  0.361595       NaN
5  >> 3  1.304000  0.947081 -0.703735
6  >> 4 -1.265421 -0.623274  0.041326
```

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Pandas: File reading and writing

Pandas can read and write many of the common file formats for general data storage.

Rule of thumb, binary formats are both smaller in size and faster to read.

Just because you can open a file does not mean you can actually read its content into a DataFrame!

Sometimes you need to provide additional settings for the parsing, *e.g.* the sep argument when reading csv files.

IO tools (text, CSV, HDF5, …)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	NA
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX	Styler.to_latex	NA
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboar
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	NA
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	to_orc
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	NA
binary	SPSS	read_spss	NA
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

Pandas: File reading and writing example

```python
import os
import numpy as np
import pandas as pd

rng = np.random.default_rng(seed=0)
X = rng.normal(0.0, 1.0, size=(6, 2))

df_wr = pd.DataFrame(X)
df_wr.columns = ['Group 1', 'Group 2']

filepath = 'test-data.csv'
df_wr.to_csv(filepath, sep='\t', index=False)

df_rd = pd.read_csv(filepath, sep='\t')

# Print the original DataFrame and
# the one read from file
print('## Original data')
print(df_wr)
print('## Data read from file')
print(df_rd)

# Clean up by deleting the test file
os.remove(filepath)
```

Pandas can read and write many of the common file formats for general data storage.

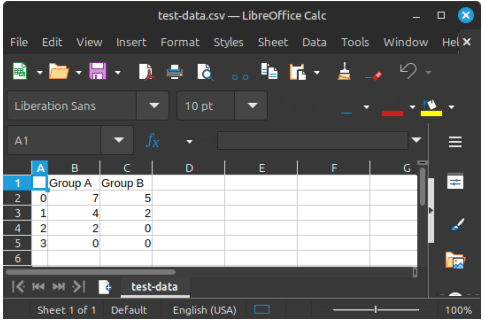
```
>> ## Original data
>>      Group 1   Group 2
>> 0   0.125730 -0.132105
>> 1   0.640423  0.104900
>> 2 -0.535669  0.361595
>> 3   1.304000  0.947081
>> 4 -0.703735 -1.265421
>> 5 -0.623274  0.041326
>> ## Data read from file
>>      Group 1   Group 2
>> 0   0.125730 -0.132105
>> 1   0.640423  0.104900
>> 2 -0.535669  0.361595
>> 3   1.304000  0.947081
>> 4 -0.703735 -1.265421
>> 5 -0.623274  0.041326
```

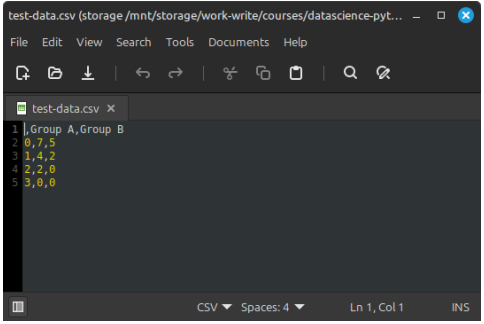
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html
https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-read-csv-table

Pandas: File formats (spreadsheet-like)

Spreadsheet like data is data organized in a multi-column table, like in Excel or other similar softwares.

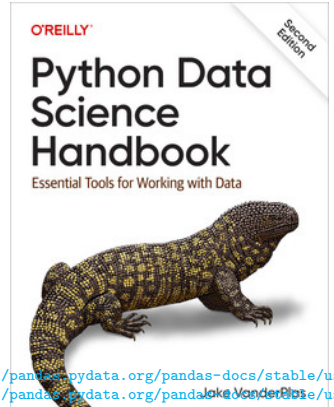
Examples are `.xls`, the open source variant `.ods` and the pure text based format `.csv`.





Pandas: Pandas

There is a lot more to pandas so please look in the documentation or in the chapters in the book.



https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
https://pandas.pydata.org/pandas-docs/stable/user_guide/cookbook.html

Summary statistics

Summary statistics: Problem

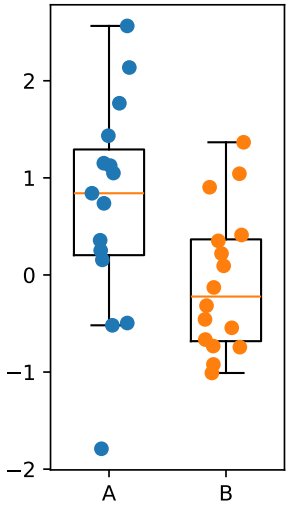
We have data *e.g.* from two treatment arms containing many samples in each.

Now we want to

1. Present the results from this data.
2. Compare the two arms. Where there any difference between them?

The first point is achieved by summary statistics such as **mean**, **median**. Can estimate **spread** with **variance** or **standard deviation**.

Comparisons are performed using **statistical tests**.



Summary statistics: Mean value

The arithmetic **mean value**, μ, is defined as the **sum of all values divided** by the **number of data points**.

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

This expression **can be written as a Python function**.

Normally we would use *e.g.* `np.mean` or a similar function from some package.

```python
def mean_value(
    X: List[float|int],
) -> float:
    """ Compute the arithmetic mean of array.
    """
    m = 0.0
    N = len(X)
    for i in range(N):
        m += X[i]
    m /= N
    return m

# Initialize a list of data, compute mean
data = [3.0, 5.0, 9.0, 4.0]
m = mean_value(data)
import numpy as np
u = np.mean(data)

# Print the data its mean value
print(f"Data: {data}")
print(f"Mean value (ours):  {m:.2f}")
print(f"Mean value (numpy): {u:.2f}")
```

```
>> Data: [3.0, 5.0, 9.0, 4.0]
>> Mean value (ours):  5.25
>> Mean value (numpy): 5.25
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html

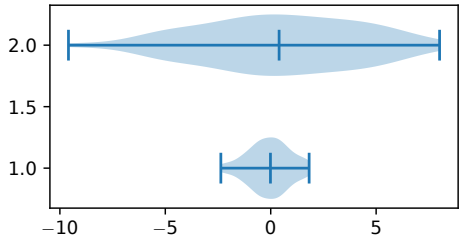
Summary statistics: Variance

The **variance** describes spread of data around **expected value**, the mean μ. **Variance** is defined as

$$v = \mathrm{Var}(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Unbiased variance is defined as

$$v = \mathrm{Var}(X) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2$$



Data with different amounts of variance.

https://en.wikipedia.org/wiki/Variance

Summary statistics: Standard deviation (Python)

Variance is, again, defined as

$$v = \mathrm{Var}(X) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Standard deviation (σ) is defined as

$$\sigma^2 = v$$
$$\sigma = \sqrt{v}$$

```python
import numpy as np
def std(
    X: List[float|int],
) -> float:
    """ Compute standard standard deviation.
    """
    m = np.mean(X)
    N = len(X)
    v = 0.0
    for i in range(N):
        v += (X[i] - m)**2
    v /= N
    s = np.sqrt(v)
    return s

# Initialize a list of data, compute mean
data = [3.0, 5.0, 9.0, 4.0]
s = std(data)
c = np.std(data)

# Print the data its mean value
print(f"Data: {data}")
print(f"Std (ours):  {s:.2f}")
print(f"Std (numpy): {c:.2f}")
```

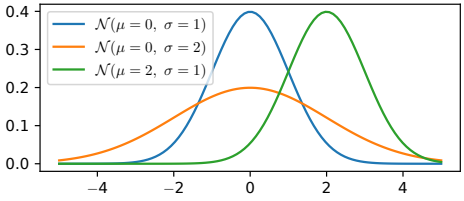
```
>> Data: [3.0, 5.0, 9.0, 4.0]
>> Std (ours):  2.28
>> Std (numpy): 2.28
```

Summary statistics: Normal distribution

Data often sampled from a **normal distribution** which is **parametrized** by the **mean** and **standard deviation**.

Its probability density is defined by the Gaussian function

$$\mathcal{N}(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$



https://en.wikipedia.org/wiki/Normal_distribution

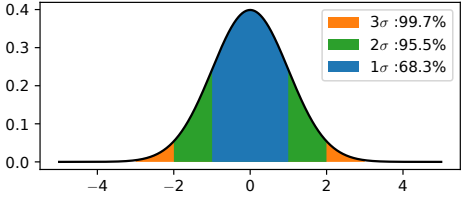
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
def gaussian(
    x: np.ndarray,
    m: float,
    s: float,
) -> np.ndarray:
    """ Compute Normal (Gaussian) distribution
    """
    return (1/np.sqrt(2*np.pi*s**2)) * np.exp
     (-np.pow(x-m, 2) / (2*s**2))

# Samples on x-axis and function parameters
x = np.linspace(-5, 5, 100)
m = [0.0, 0.0, 2.0] # Means
s = [1.0, 2.0, 1.0] # SDs

# Plot three probability density curves
matplotlib.use('pgf')
ax = plt.figure(figsize=(5, 2)).gca()
for i in range(len(m)):
    y = gaussian(x, m[i], s[i])
    label = r'$\mathcal{N}(\mu ='
    label += f'{m[i]:.0f},~'
    label += r'\sigma = '
    label += f'{s[i]:.0f})$'
    ax.plot(x, y, label=label)
ax.legend()
```

Summary statistics: 68%, 95% and 99,7% rule

Rule of thumb that 68.3% of all normally distributed samples lie within one standard deviation centered on the mean.

Similarly 95.5% and 99.7% of the samples lie within two and three standard deviations respectively.

68%, 95%, 99% rule.



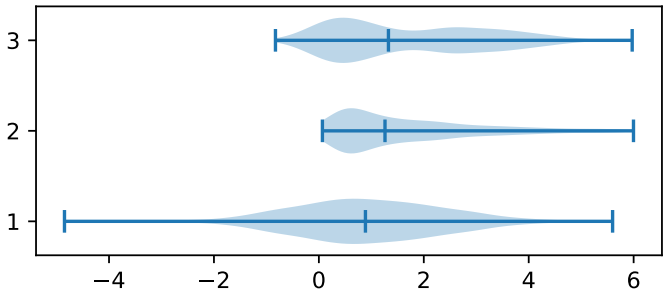
https://en.wikipedia.org/wiki/Normal_distribution

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
def gaussian(
    x: np.ndarray,
) -> np.ndarray:
    return (1/np.sqrt(2*np.pi)) * np.exp(-np.
      pow(x, 2) / 2)
x = np.linspace(-5, 5, 1000)
y = gaussian(x)
# Plot samples
matplotlib.use('pgf') # Use the pgf backend!
ax = plt.figure(figsize=(5,2)).gca()
s = [3, 2, 1]
p = [r'99.7\%', r'95.5\%', r'68.3\%']
c = ['tab:orange', 'tab:green', 'tab:blue']
zeros = np.zeros(x.shape)
for i in range(len(s)):
    ind = np.abs(x) < s[i]
    label = f'{s[i]}' + r'$\sigma$ :' + p[i]
    ax.fill_between(x[ind],
                    zeros[ind], y[ind],
                    facecolor=c[i],
                    label=label)
ax.plot(x, y, c='k')
ax.legend()
#plt.show()
```

Summary statistics: Non-normal distributions

For data sampled from non-normal distributions the **median** can give a better summary description than the **mean** value.
(More generally could also look at the **mode** of the data.)

Similarly, **percentiles** can **describe the spread** of the data. *E.g.* can report percentiles analogous to 1, 2 or 3 **standard deviations**.



https://en.wikipedia.org/wiki/Mode_(statistics)

Summary statistics: Median and percentiles

Calculating the **median** is **different** from calculating the **mean**.

There is **no mathematical formula**, instead we must **sort the data**. Only after it is sorted can we find the value in the middle, which is the median.

The **percentiles** are calculated in an **analogous** way.

```
>> Data:   [3.0, 5.0, 9.0, 4.0, 1.0]
>> Sorted: [1. 3. 4. 5. 9.]
>> Median value (ours):  4.00
>> Median value (numpy): 4.00
>> Percentile value (numpy): 3.00
```

```python
import numpy as np

def median_value(
    X: List[float|int],
) -> float:
    """ Compute the median of array.
    """
    X = np.sort(X)
    m = X[len(X)//2]
    return m

# Initialize a list of data, compute mean
data = [3.0, 5.0, 9.0, 4.0, 1.0]
m = median_value(data)
u = np.median(data)

sorted_data = np.sort(data)

# Compute the percentile
# 40% of all data is at or below
v = np.percentile(data, 40,
                  method='closest_observation')

# Print the data its mean value
print(f"Data:   {data}")
print(f"Sorted: {sorted_data}")
print(f"Median value (ours):  {m:.2f}")
print(f"Median value (numpy): {u:.2f}")
print(f"Percentile value (numpy): {v:.2f}")
```

Single group test

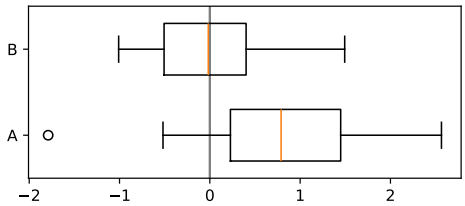
Single group test

Single group test: Statistical tests

Statistical tests are performed to see if or data agrees or disagrees with our research hypothesis.

E.g. comparing a treated patient group to some baseline value.

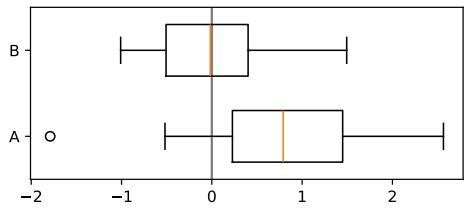
Or comparing patients in one group exposed to a new type of treatment with patients in another group receiving the best supportive care.



Single group test: Single group test (`ttest_1samp(x)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=20)
B = rng.normal(loc=0.0, scale=1.0, size=20)

sA = stats.ttest_1samp(A, popmean=0.0)
sB = stats.ttest_1samp(B, popmean=0.0)

print(f"Return type:\n    {type(sA)}")
print(f"p value (A): {sA.pvalue:.5f}")
print(f"p value (B): {sB.pvalue:.5f}")

fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal')
ax.plot((0, 0), ax.get_ylim(),
        c='tab:gray', zorder=0)
#plt.show()
```

The **one sample T-test** compares the mean of the data to some value.




```
>> Return type:
>>     <class 'scipy.stats._stats_py.
    TtestResult'>
>> p value (A): 0.00351
>> p value (B): 0.70097
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_1samp.html

Two related groups tests

Two related groups tests: Paired samples

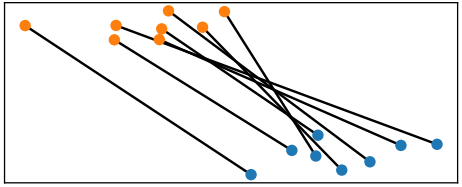
Paired samples are created when we have the same, or a very similar, source for individual data points.

E.g. a group of patients with diabetes are put on a novel drug, and then they are given a placebo treatment.

The order in which they receive the drug and placebo could be randomized so we accommodate for bias.

Then blood sugar levels would be measured both after receiving the drug and the placebo.
As such, the data points from the two

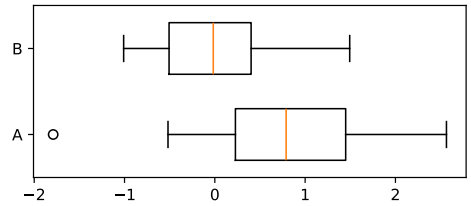
As such, the data points from the two measurements are paired since they come from the same patient.



Two related groups tests: Paired t-test (`ttest_rel(x, y)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=20)
B = rng.normal(loc=0.0, scale=1.0, size=20)

s = stats.ttest_rel(A, B)

print(f"Return type:\n    {type(s)}")
print(f"p value (A - B): {s.pvalue:.5f}")

fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal')
#plt.show()
try:
    C = rng.normal(loc=0.0, scale=1.0, size
     =21)
    _ = stats.ttest_rel(A, C)
except Exception as e:
    print(f"error: {e}")
```

The parametric **paired T-test** compares two sets of paired samples. The number of samples in both sets should be equal!



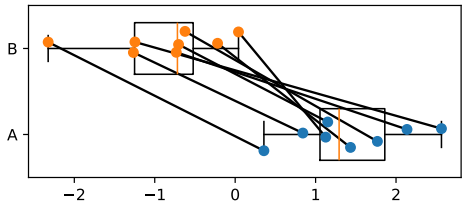

```
>> Return type:
>>     <class 'scipy.stats._stats_py.
    TtestResult'>
>> p value (A - B): 0.02026
>> error: unequal length arrays
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_rel.html

Two related groups tests: Wilcoxon signed rank test (`wilcoxon(x, y)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=8)
B = rng.normal(loc=0.0, scale=1.0, size=8)
# Generate random offsets for scatter plots
a = 1 + rng.uniform(-0.2, 0.2, 8)
b = 2 + rng.uniform(-0.2, 0.2, 8)
# Perform statistical test, print result
s = stats.wilcoxon(A, B)
print(f"Return type:\n    {type(s)}")
print(f"p value (A - B): {s.pvalue:.5f}")

# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal')
for i in range(len(A)):
    ax.plot((A[i], B[i]), (a[i], b[i]), c='k')
ax.scatter(A, a, zorder=3)
ax.scatter(B, b, zorder=3)
#plt.show()
```

The non-parametric **Wilcoxon signed rank test** compares two sets of paired samples. The number of samples in both sets should be equal!




```
>> Return type:
>>     <class 'scipy.stats._morestats.
    WilcoxonResult'>
>> p value (A - B): 0.00781
```

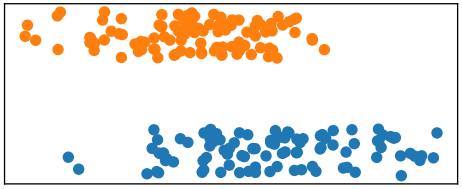
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html

Two unrelated groups tests

Two unrelated groups tests

Two unrelated groups tests: Non-paired samples

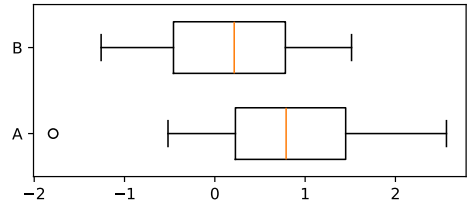
Non-paired samples are created when we have different sources for individual data points.
E.g. a group of patients with diabetes are put on a novel drug and another, control group, is given a placebo treatment.



Two unrelated groups tests: Two sample t-test (`ttest_ind(A, B)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=20)
B = rng.normal(loc=0.0, scale=1.0, size=25)
# Perform statistical test, print result
s = stats.ttest_ind(A, B)
print(f"Number of samples in A: {len(A)}")
print(f"Number of samples in B: {len(B)}")
print(f"Return type:\n    {type(s)}")
print(f"p value (A - B): {s.pvalue:.5f}")

# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal')
#plt.show()
```

The parametric **two sampled T test** compares two sets of independent samples. The number of samples in both sets might not be equal.



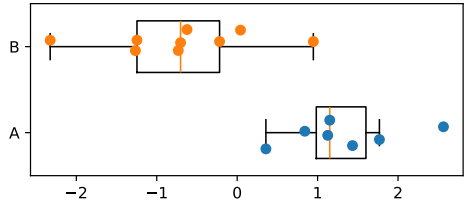

```
>> Number of samples in A: 20
>> Number of samples in B: 25
>> Return type:
>>     <class 'scipy.stats._stats_py.
    TtestResult'>
>> p value (A - B): 0.02876
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

Two unrelated groups tests: Wilcoxon rank sum test (`ranksums(A, B)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=7)
B = rng.normal(loc=0.0, scale=1.0, size=9)
# Generate random offsets for scatter plots
a = 1 + rng.uniform(-0.2, 0.2, 7)
b = 2 + rng.uniform(-0.2, 0.2, 9)
# Perform statistical test, print result
s = stats.ranksums(A, B)
print(f"Return type:\n    {type(s)}")
print(f"p value (A - B): {s.pvalue:.5f}")

# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal',
           showfliers=False)
ax.scatter(A, a, zorder=3)
ax.scatter(B, b, zorder=3)
#plt.show()
```

The non-parametric **Wilcoxon rank sum test** compares two sets of independent samples. The number of samples in both sets might not be equal.



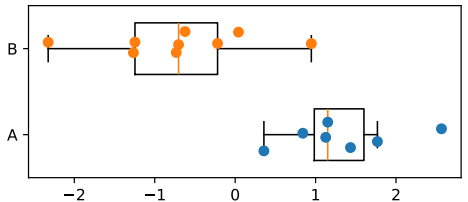

```
>> Return type:
>>     <class 'scipy.stats._stats_py.
    RanksumsResult'>
>> p value (A - B): 0.00179
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ranksums.html

Two unrelated groups tests: Mann-Whitney test (`mannwhitneyu(A, B)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=7)
B = rng.normal(loc=0.0, scale=1.0, size=9)
# Generate random offsets for scatter plots
a = 1 + rng.uniform(-0.2, 0.2, 7)
b = 2 + rng.uniform(-0.2, 0.2, 9)
# Perform statistical test, print result
s = stats.mannwhitneyu(A, B)
print(f"Return type:\n    {type(s)}")
print(f"p value (A - B): {s.pvalue:.5f}")

# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B],
           widths=0.60,
           tick_labels=['A', 'B'],
           orientation='horizontal',
           showfliers=False)
ax.scatter(A, a, zorder=3)
ax.scatter(B, b, zorder=3)
#plt.show()
```

The non-parametric **Mann-Whitney U rank test** compares two sets of independent samples. The number of samples in both sets might not be equal.




```
>> Return type:
>>     <class 'scipy.stats._mannwhitneyu.
    MannwhitneyuResult'>
>> p value (A - B): 0.00070
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

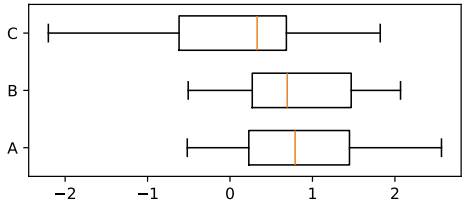
More than two groups tests

More than two groups tests

More than two groups tests: Multiple comparison testing

We get more than two groups to compare if we have, *e.g.* more treatment options that we compare.

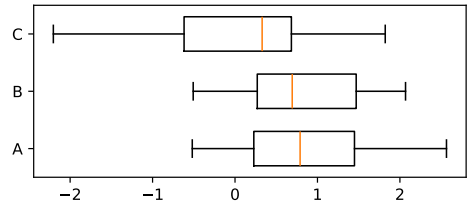
E.g. a group of patients with diabetes are put on a novel drug, another is given a different drug and a third, control, group is given a placebo treatment.



More than two groups tests: ANOVA (`f_oneway(A, B, C)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=20)
B = rng.normal(loc=0.5, scale=0.8, size=30)
C = rng.normal(loc=0.0, scale=1.0, size=25)
# Perform statistical test, print result
s = stats.f_oneway(A, B, C)
print(f"Return type:\n    {type(s)}")
print(f"p value (A-B-C): {s.pvalue:.5f}")

# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B, C],
           widths=0.60,
           tick_labels=['A', 'B', 'C'],
           orientation='horizontal',
           showfliers=False)
#plt.show()
```

The parametric analysis of variance **(ANOVA)** compares multiple sets of independent samples.



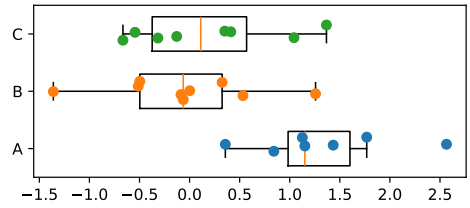

```
>> Return type:
>>     <class 'scipy.stats._stats_py.
    F_onewayResult'>
>> p value (A-B-C): 0.01349
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

More than two groups tests: Kruskal-Wallis H test (`kruskal(A, B, C)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
A = rng.normal(loc=1.0, scale=1.2, size=7)
B = rng.normal(loc=0.5, scale=0.8, size=9)
C = rng.normal(loc=0.0, scale=1.0, size=8)
# Generate random offsets for scatter plots
a = 1 + rng.uniform(-0.2, 0.2, 7)
b = 2 + rng.uniform(-0.2, 0.2, 9)
c = 3 + rng.uniform(-0.2, 0.2, 8)
# Perform statistical test, print result
s = stats.kruskal(A, B, C)
print(f"Return type:\n    {type(s)}")
print(f"p value (A-B-C): {s.pvalue:.5f}")
# Plot samples
fig = plt.figure(figsize=(5,2))
ax = fig.gca()
ax.boxplot([A, B, C], widths=0.60,
           tick_labels=['A', 'B', 'C'],
           orientation='horizontal',
           showfliers=False)
ax.scatter(A, a, zorder=3)
ax.scatter(B, b, zorder=3)
ax.scatter(C, c, zorder=3)
#plt.show()
```

The non-parametric **Kruskal-Wallis H test** compares multiple sets of independent samples.



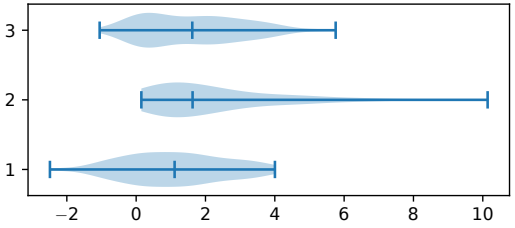

```
>> Return type:
>>     <class 'scipy.stats._stats_py.
    KruskalResult'>
>> p value (A-B-C): 0.01010
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kruskal.html

Normality

Normality: Shapiro-Wilks test (`shapiro(x)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

rng = np.random.default_rng(seed=0)

a = rng.normal(1, 1.5, 100)
b = rng.lognormal(0.5, 1, 100)
c = rng.normal(0.5, 0.5, 50)
c = np.concatenate([c, rng.normal(3.0, 1.0,
    50)])
#b[b>6] = 0.5

# Plot three probability density curves
ax = plt.figure(figsize=(5, 2)).gca()
ax.violinplot([a, b, c],
              showmedians=True,
              orientation='horizontal')

pa = stats.shapiro(a).pvalue
pb = stats.shapiro(b).pvalue
pc = stats.shapiro(c).pvalue

print(f"1 is normal: pvalue = {pa:.5f}")
print(f"2 is normal: pvalue = {pb:.5f}")
print(f"3 is normal: pvalue = {pc:.5f}")
```




```
>> a is normal: pvalue = 0.18734
>> b is normal: pvalue = 0.00000
>> c is normal: pvalue = 0.00247
```

The Shapiro-Wilk test tests the null hypothesis that the data was drawn from a normal distribution.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html

Normality: Skewness and kurtosis

```python
import numpy as np
from scipy import stats

rng = np.random.default_rng(seed=0)

a = rng.normal(1, 1.5, 100)
b = rng.lognormal(0.5, 1, 100)
c = rng.normal(0.5, 0.5, 50)
c = np.concatenate([c, rng.normal(3.0, 1.0,
    50)])

d = [a, b, c]
l = ['a', 'b', 'c']
for i in range(len(d)):
    p_n = stats.shapiro(d[i]).pvalue
    print(f"{l[i]} is normal: pvalue = {p_n:.5
    f}")

    p_s = stats.skewtest(d[i])
    p_k = stats.kurtosistest(d[i])

    print(p_s)
    print(p_k)
```

```
>> a is normal: pvalue = 0.18734
>> SkewtestResult(statistic=np.float64
    (0.0903607469119398), pvalue=np.float64
    (0.9280005484700018))
>> KurtosistestResult(statistic=np.float64
    (-1.3999191988710709), pvalue=np.float64
    (0.16153751613263923))
>> b is normal: pvalue = 0.00000
>> SkewtestResult(statistic=np.float64
    (5.077797156838873), pvalue=np.float64
    (3.8183602639328356e-07))
>> KurtosistestResult(statistic=np.float64
    (3.0368737833520134), pvalue=np.float64
    (0.002390455273602988))
>> c is normal: pvalue = 0.00247
>> SkewtestResult(statistic=np.float64
    (1.9601279117236894), pvalue=np.float64
    (0.0499808416065829))
>> KurtosistestResult(statistic=np.float64
    (-1.1047962649659528), pvalue=np.float64
    (0.26924787931880445))
```

Kurtosis refers to degree of *tailedness*.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skewtest.html
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kurtosistest.html

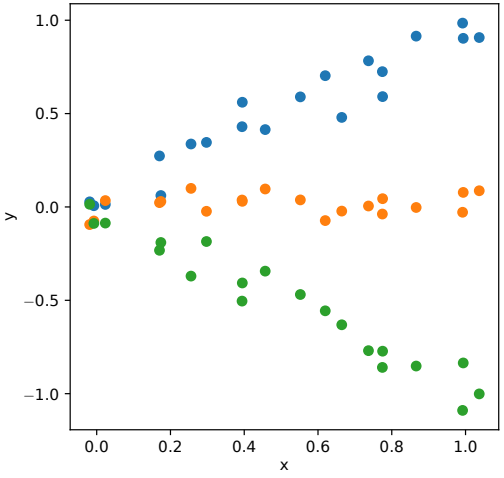
Correlation

Correlation

Correlation: Relation between two features

Very common to want to investigate **linear relation** between two datasets.

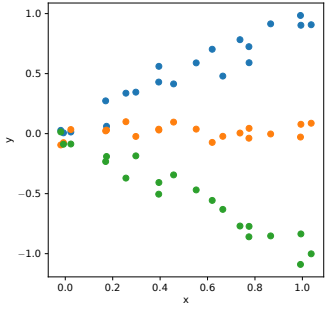
If we increase the dose of some drug, how much of a response can we expect.



Correlation: Pearson (`pearsonr(x, y)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
x = np.linspace(0, 1, 20)
a =  x + rng.uniform(-0.1, 0.1, len(x))
b =  0 + rng.uniform(-0.1, 0.1, len(x))
c = -x + rng.uniform(-0.1, 0.1, len(x))
x += rng.uniform(-0.1, 0.1, len(x))
# Plot samples
fig = plt.figure(figsize=(5,5))
ax = fig.gca()
# Perform statistical test, print result, plot
for y in [a, b, c]:
    r = stats.pearsonr(x, y)
    c = r.correlation
    print(f'Correlation: {c:.3f}')
    ax.scatter(x, y, zorder=3)
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.show()
```

The parametric **Pearson correlation** investigates **linear** relation between two datasets.



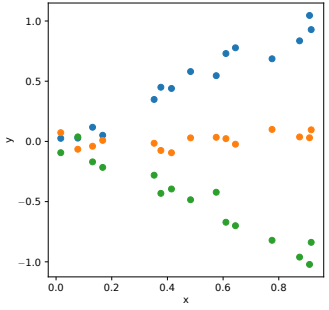

```
>> Correlation: 0.961
>> Correlation: 0.225
>> Correlation: -0.972
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html

Correlation: Spearman (`spearmanr(x, y)`)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
# Generate normaly distributed samples
rng = np.random.default_rng(seed=0)
x = np.linspace(0, 1, 15)
a =  x + rng.uniform(-0.1, 0.1, len(x))
b =  0 + rng.uniform(-0.1, 0.1, len(x))
c = -x + rng.uniform(-0.1, 0.1, len(x))
x += rng.uniform(-0.1, 0.1, len(x))
# Plot samples
fig = plt.figure(figsize=(5,5))
ax = fig.gca()
# Perform statistical test, print result, plot
for y in [a, b, c]:
    r = stats.spearmanr(x, y)
    c = r.correlation
    print(f'Correlation: {c:.3f}')
    ax.scatter(x, y, zorder=3)
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.show()
```

The non-parametric **Spearman correlation** investigates **non-linear** relation between two datasets.




```
>> Correlation: 0.975
>> Correlation: 0.500
>> Correlation: -0.968
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

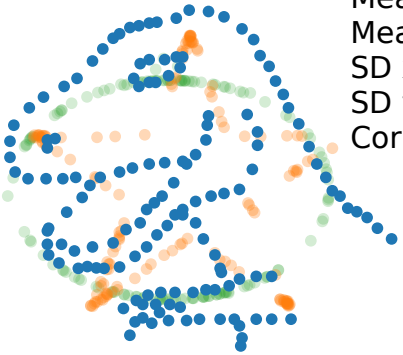
Statistics conclusion

Statistics conclusion

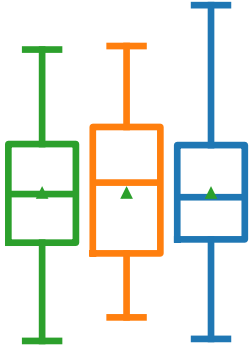
Statistics conclusion: Datasaurus

We need to be **careful** when only looking at **summary statistics and correlation coefficients**!

What is the *shape* of the data?



Mean x: 54.26
Mean y: 47.83
SD x: 16.71
SD y: 26.84
Correlation: -0.06



Datasaurus dozen
https://www.research.autodesk.com/publications/same-stats-different-graphs/

Statistics conclusion: Advanced statistics

Advanced statistics can be performed using the statsmodels package.
Somewhat mimics some of the functionality in the R language.

https://www.statsmodels.org

Statistics conclusion: Conclusion

Pandas

Very useful for both file reading and writing.
Provides spreadsheet-like functionality.
Can handle missing data.

Scipy

Contains many many modules.
The `stats` module contains the common statistical tests..