Experimental design and analysis

Intro to Jupyter and descriptive statistics

https://www.lri.fr/~appert/eval/

Caroline Appert

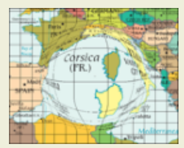
Using Jupyter for analyzing data

Experiment that we use as an example

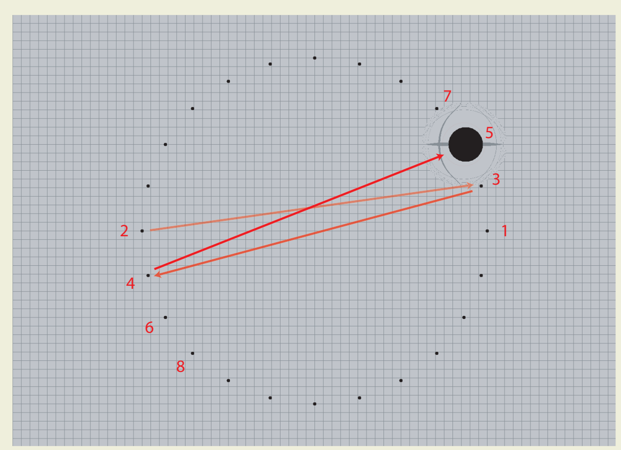
Pointing performance of different types of magnifying lenses

2 factors (5 x 5 design) - 10 participants

Lens type (5 levels):

 **ML** (Manhattan Lens) ,  **FL** (Fisheye Lens) , **SCF** , **SCB** ,  **BL** (Blending Lens)

Lens' magnification (5 levels): **2**, **4**, **6**, **10**, **14**

Task  target acquisition

Measure Pointing time (in ms)

Experiment that we use as an example

Pointing performance of different types of magnifying lenses

2 factors (5 x 5 design) - 10 participants

2 Factors

1 Measure

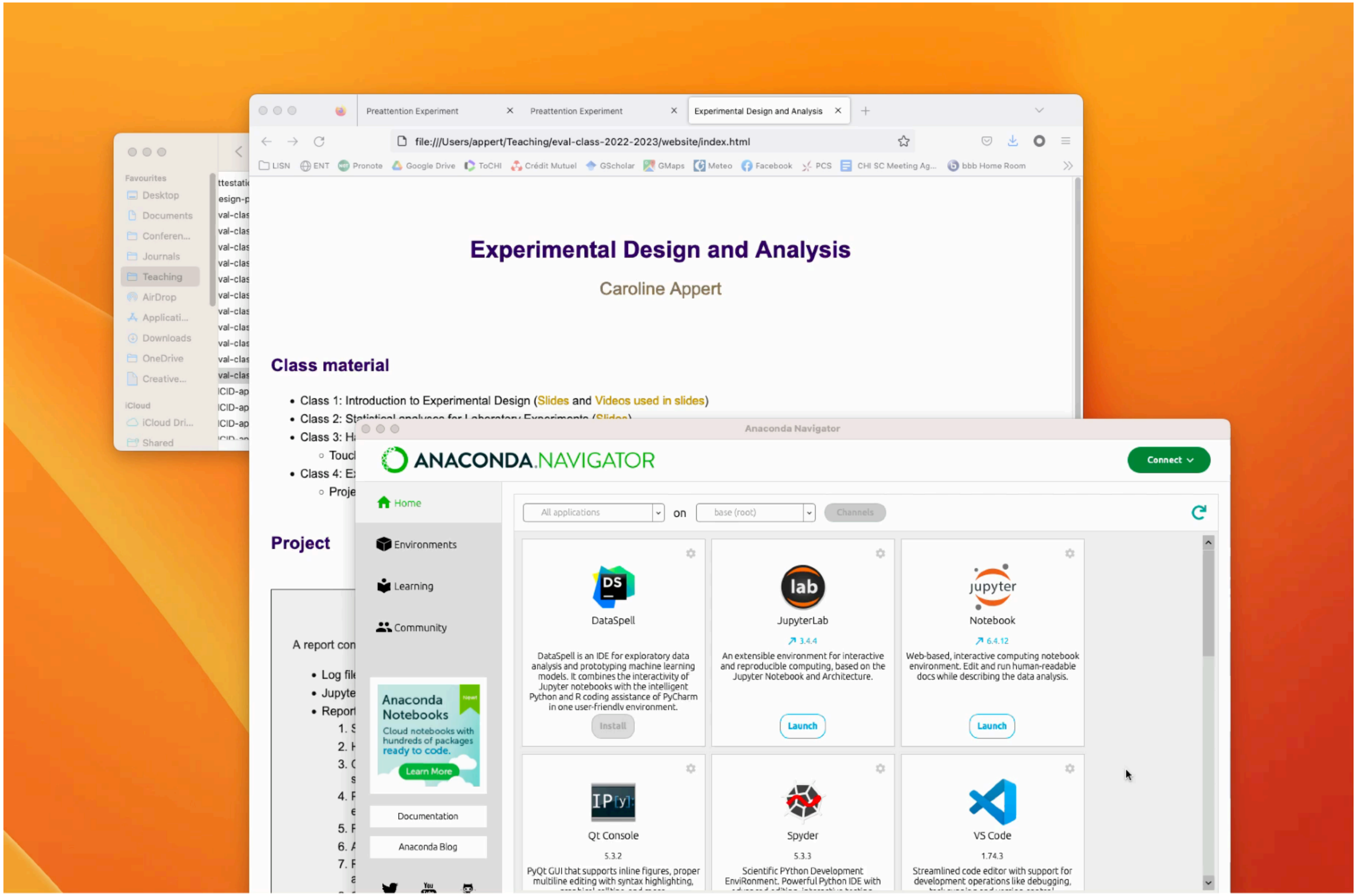
Collected data

(log file `lens_experiment.csv`)

Note: When we analyze collected results, all logs are in a single file (≠ one file per participant)

Participant	Block	Trial	Lens	Magnification	ID	PointingTime
1	4	0	FL	6	6.0035549	2297
1	4	0	FL	6	6.0035549	1485
1	4	0	FL	6	6.0035549	2000
1	4	0	FL	6	6.0035549	1843
1	4	0	FL	6	6.0035549	1813
…						
10	2	9	SCF	6	6.0035549	2375
10	2	9	SCF	6	6.0035549	2359
10	2	9	SCF	6	6.0035549	2313
10	2	9	SCF	6	6.0035549	2453
10	2	9	SCF	6	6.0035549	2187
10	2	9	SCF	6	6.0035549	2875
10	2	9	SCF	6	6.0035549	2688

Creating a Jupyter notebook



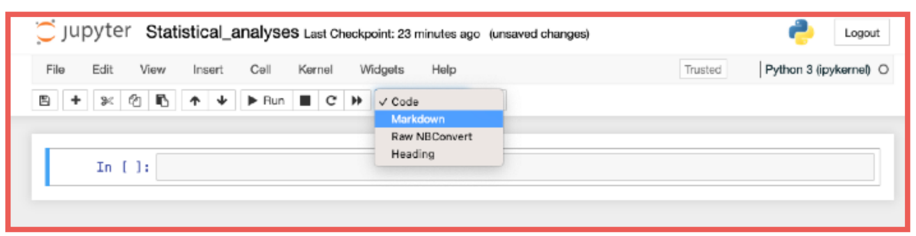
Jupyter notebook

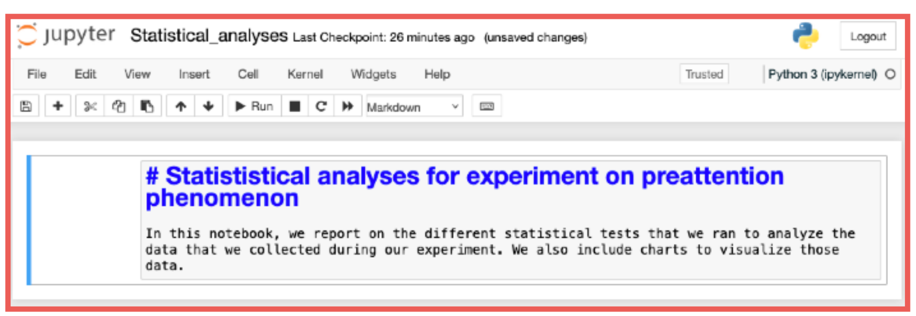
Notebook = web-based interactive computing platform

Two types of cell:

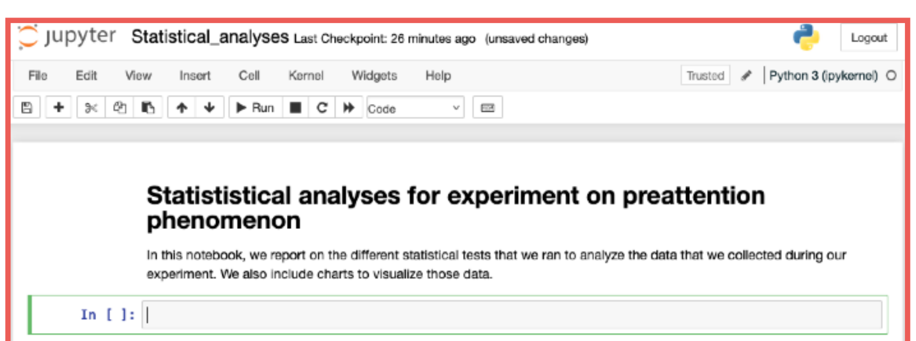
Markdown cells

Set cell type to markdown



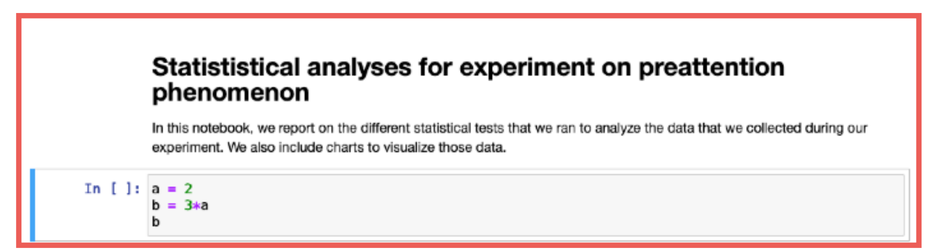


Interpret (Shift+Enter)

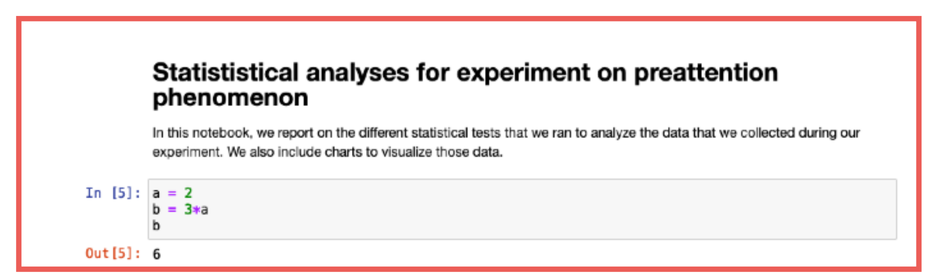


Code cells

(default type)

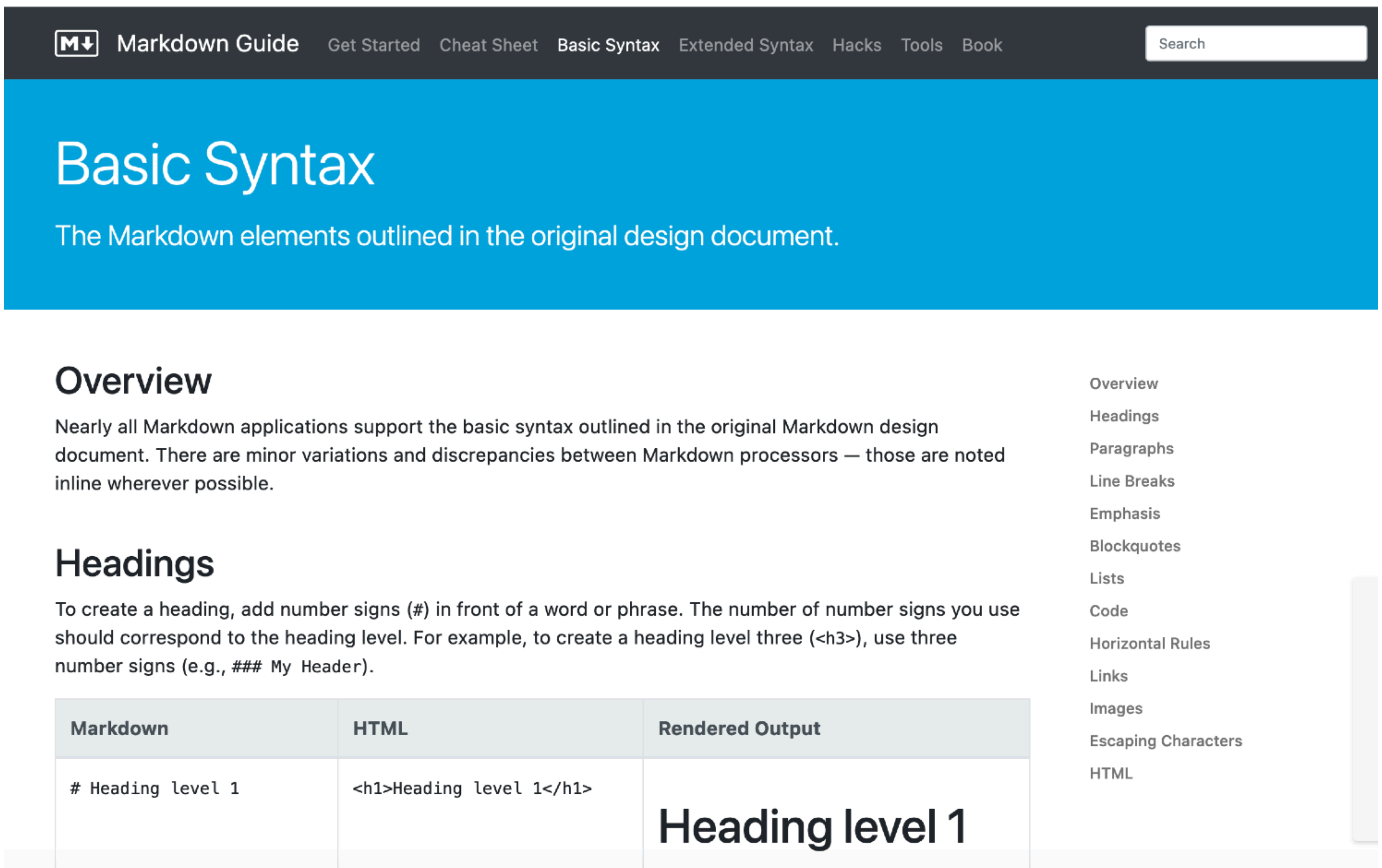


Interpret (Shift+Enter)



Markdown

https://www.markdownguide.org/basic-syntax/



Markdown Guide Get Started Cheat Sheet Basic Syntax Extended Syntax Hacks Tools Book Search

Basic Syntax

The Markdown elements outlined in the original design document.

Overview

Nearly all Markdown applications support the basic syntax outlined in the original Markdown design document. There are minor variations and discrepancies between Markdown processors — those are noted inline wherever possible.

Headings

To create a heading, add number signs (#) in front of a word or phrase. The number of number signs you use should correspond to the heading level. For example, to create a heading level three (`<h3>`), use three number signs (e.g., `### My Header`).

Markdown	HTML	Rendered Output
`# Heading level 1`	`<h1>Heading level 1</h1>`	**Heading level 1**

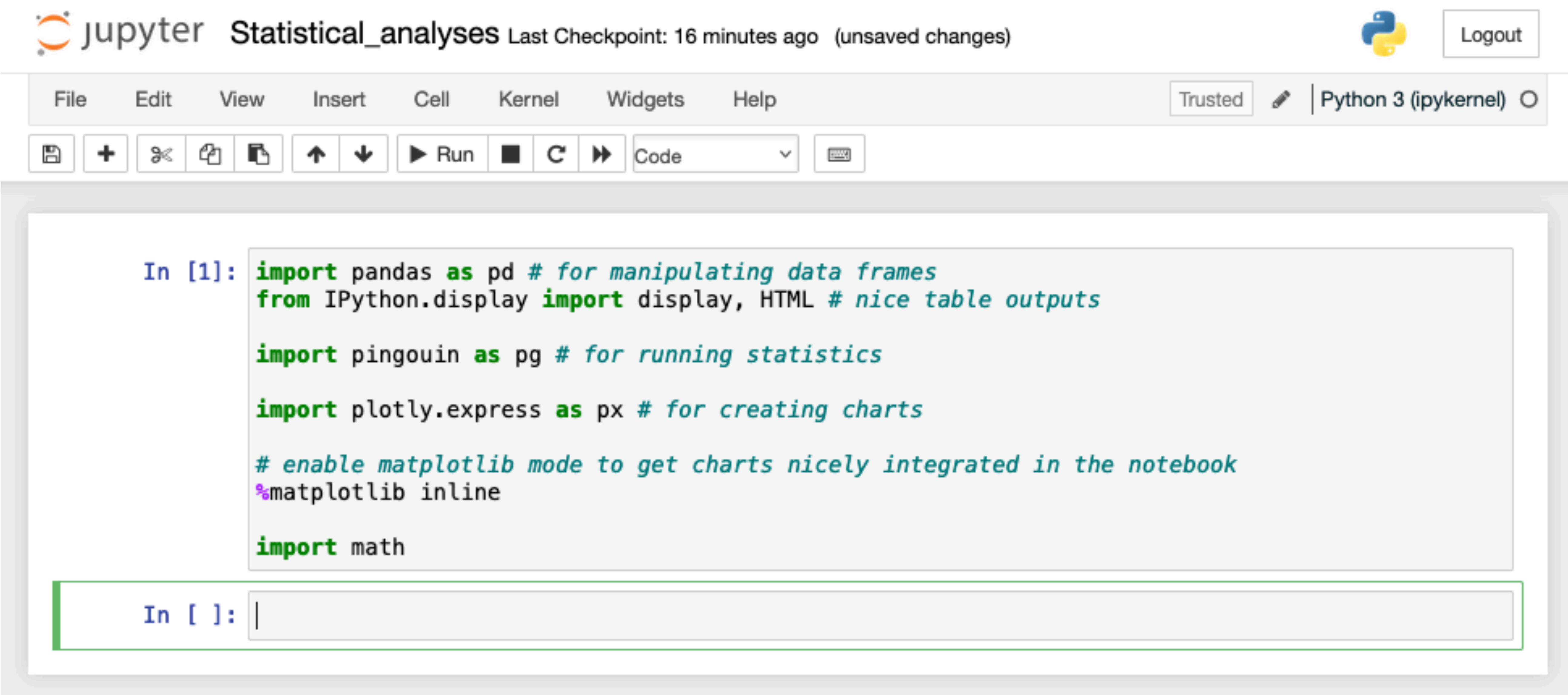
Markdown in Jupyter includes HTML

```
# <span style='color:blue'>Descriptive statistics</span>
```

Interpret
(Shift+Enter)

Descriptive statistics

Useful libraries



Pandas

```
import pandas as pd
```

The pandas library facilitates working with tabular data in Python with functions for reading, writing and manipulating those data.

A pandas DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet.

```
pd.read_csv
pd.DataFrame.dtypes
pd.DataFrame.describe
pd.DataFrame.query
pd.DataFrame.groupby
...
```

Pandas

From CSV To DataFrame

lens_experiment/lens_experiment.csv

```
Participant;Block;Trial;Lens;Magnification;ID;PointingTime
1;4;0;FL;6;6.0035549;2297
1;4;0;FL;6;6.0035549;1485
1;4;0;FL;6;6.0035549;2000
1;4;0;FL;6;6.0035549;1843
1;4;0;FL;6;6.0035549;1813
...
10;2;9;SCF;6;6.0035549;2313
10;2;9;SCF;6;6.0035549;2453
10;2;9;SCF;6;6.0035549;2187
10;2;9;SCF;6;6.0035549;2875
10;2;9;SCF;6;6.0035549;2688
```

```python
data = pd.read_csv('lens_experiment/lens_experiment.csv', sep=';')
data
```

	Participant	Block	Trial	Lens	Magnification	ID	PointingTime
0	1	4	0	FL	6	6.003555	2297
1	1	4	0	FL	6	6.003555	1485
2	1	4	0	FL	6	6.003555	2000
...	...	...	...	...	...	...	...
11997	10	2	9	SCF	6	6.003555	2187
11998	10	2	9	SCF	6	6.003555	2875
11999	10	2	9	SCF	6	6.003555	2688

12000 rows × 7 columns

```python
data.dtypes
```

```
Participant        int64
Block              int64
Trial              int64
Lens              object
Magnification      int64
ID               float64
PointingTime       int64
dtype: object
```

columns in a dataframe can have different types

Pandas

Types

```
data.dtypes
```

```
Participant       int64
Block             int64
Trial             int64
Lens             object
Magnification     int64
ID              float64
PointingTime      int64
dtype: object
```

Participant should not be int. It is actually an identifier. Let's turn it into str.

```
data['Participant'] = data['Participant'].astype('str')
data.dtypes
```

```
Participant                           object
Block                                  int64
Trial                                  int64
Lens                                  object
Magnification                          int64
ID                                   float64
PointingTime                           int64
Condition: Lens, Magnification, ID    object
dtype: object
```

Pandas

DataFrame - Access

Access column

```
data['Participant']
```

```
0        1
1        1
2        1
3        1
4        1
        ..
11995   10
11996   10
11997   10
11998   10
11999   10
Name: Participant, Length: 12000, dtype: int64
```

Access row

```
data.iloc[2]
```

```
Participant            1
Block                  4
Trial                  0
Lens                  FL
Magnification          6
ID              6.003555
PointingTime        2000
Name: 2, dtype: object
```

Access rows that satisfy a given criterion

```
data_fl = data.query('Lens == \'FL\'') # filter out data to get only data for the Lens=FL condition
data_fl
```

	Participant	Block	Trial	Lens	Magnification	ID	PointingTime
0	1	4	0	FL	6	6.003555	2297
1	1	4	0	FL	6	6.003555	1485
...	...	...	...	...	...	...	...
11038	10	3	9	FL	6	6.003555	1312
11039	10	3	9	FL	6	6.003555	2282

2400 rows × 7 columns

Pandas

DataFrame - Adding columns

Building a dataframe example

```
d = {'id':[1,3,4],'starttime': [124, 357, 489], 'endtime': [202, 476, 604]}
df = pd.DataFrame(data=d)
df
```

	id	starttime	endtime
0	1	124	202
1	3	357	476
2	4	489	604

Adding columns

Adding a column "completiontime" whose value is endtime-starttime

```
df['completiontime'] = df['endtime'] - df['starttime']

df.loc[df['id'] <= 3, 'difficulty'] = 'easy'
df.loc[df['id'] > 3, 'difficulty'] = 'medium'

df
```

Adding a column "difficulty" whose value is 'easy' if ID <= 3, 'medium' otherwise

	id	starttime	endtime	completiontime	difficulty
0	1	124	202	78	easy
1	3	357	476	119	easy
2	4	489	604	115	medium

Descriptive statistics

Pandas

DataFrame - Aggregating

The **describe** function provides descriptive statistics of a dataframe

```
data.describe(include = 'all')
```

	Participant	Block	Trial	Lens	Magnification	ID	PointingTime
count	12000.000000	12000.000000	12000.000000	12000	12000.000000	12000.000000	12000.000000
unique	NaN	NaN	NaN	5	NaN	NaN	NaN
top	NaN	NaN	NaN	FL	NaN	NaN	NaN
freq	NaN	NaN	NaN	2400	NaN	NaN	NaN
mean	5.500000	3.000000	4.500000	NaN	7.200000	6.112107	2561.142083
std	2.872401	1.414272	2.872401	NaN	4.308311	1.327710	1828.979452
min	1.000000	1.000000	0.000000	NaN	2.000000	4.200952	766.000000
25%	3.000000	2.000000	2.000000	NaN	4.000000	5.288921	1500.000000
50%	5.500000	3.000000	4.500000	NaN	6.000000	6.003555	2000.000000
75%	8.000000	4.000000	7.000000	NaN	10.000000	7.069674	2969.000000
max	10.000000	5.000000	9.000000	NaN	14.000000	7.997436	32906.000000

Pandas proposes several functions for aggregating data:
mean, min, max, sum, ...

```
data.PointingTime.mean()
```

```
2561.142083333333
```

DataFrame - Aggregating

Combined with the **`groupby`** function, we can get a breakdown per group:

```
data.groupby('Lens').PointingTime.mean()
```

```
Lens
BL     2666.405417
FL     2399.813750
ML     3498.588750
SCB    1881.055417
SCF    2359.847083
Name: PointingTime, dtype: float64
```

type of result: series

Combined with the **`aggregate`** function, we can specify elaborate aggregating strategies:

```
data.groupby('Lens').aggregate({'Trial': 'sum', 'PointingTime': 'mean'})
```

	Trial	PointingTime
Lens		
BL	10800	2666.405417
FL	10800	2399.813750
ML	10800	3498.588750
SCB	10800	1881.055417
SCF	10800	2359.847083

Counts

When designing an experiment, a good sanity check consists of looking at the number of observations (trials) per experimental condition to double check that we actually have the same number of observations per condition (*i.e.*, our design is properly **balanced**).

```
# make a copy of column Magnification and change its type from int to str
magAsStr = data['Magnification'].copy().astype('str')
# now that we have strings, we can concatenate them using function 'cat'
data['Condition: Lens, Magnification'] = data['Lens'].str.cat(magAsStr, sep=", ")
data
```

	Participant	Block	Trial	Lens	Magnification	ID	PointingTime	Condition: Lens, Magnification
0	1	4	0	FL	6	6.003555	2297	FL, 6
1	1	4	0	FL	6	6.003555	1485	FL, 6
2	1	4	0	FL	6	6.003555	2000	FL, 6
3	1	4	0	FL	6	6.003555	1843	FL, 6
4	1	4	0	FL	6	6.003555	1813	FL, 6
...	...	...	...	...	...	...	...	...
11995	10	2	9	SCF	6	6.003555	2313	SCF, 6
11996	10	2	9	SCF	6	6.003555	2453	SCF, 6
11997	10	2	9	SCF	6	6.003555	2187	SCF, 6
11998	10	2	9	SCF	6	6.003555	2875	SCF, 6
11999	10	2	9	SCF	6	6.003555	2688	SCF, 6

12000 rows × 8 columns

Counts

When designing an experiment, a good sanity check consists of looking at the number of observations (trials) per experimental condition to double check that we actually have the same number of observations per condition (*i.e.*, our design is properly **balanced**).

```
data.groupby('Participant').count()
```

	Block	Trial	Lens	Magnification	ID	PointingTime	Condition: Lens, Magnification
Participant							
1	1200	1200	1200	1200	1200	1200	1200
2	1200	1200	1200	1200	1200	1200	1200
3	1200	1200	1200	1200	1200	1200	1200
4	1200	1200	1200	1200	1200	1200	1200
5	1200	1200	1200	1200	1200	1200	1200
6	1200	1200	1200	1200	1200	1200	1200
7	1200	1200	1200	1200	1200	1200	1200
8	1200	1200	1200	1200	1200	1200	1200
9	1200	1200	1200	1200	1200	1200	1200
10	1200	1200	1200	1200	1200	1200	1200

Each participant completed 1200 tasks in each experimental condition

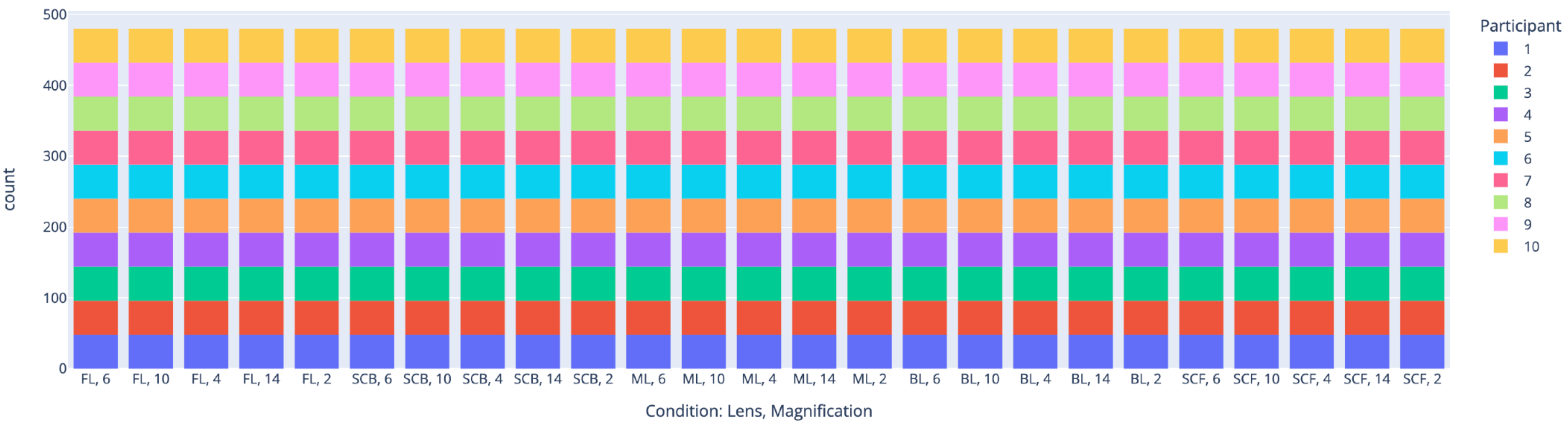
Distribution

nominal/ordinal variables

Function **`histogram`** from library **`plotly`** visualizes distributions. (https://plotly.com/python/histograms/)

We can use it to visualize our counts:

```
fig = px.histogram(data, x='Condition: Lens, Magnification', color='Participant')
fig.show()
```



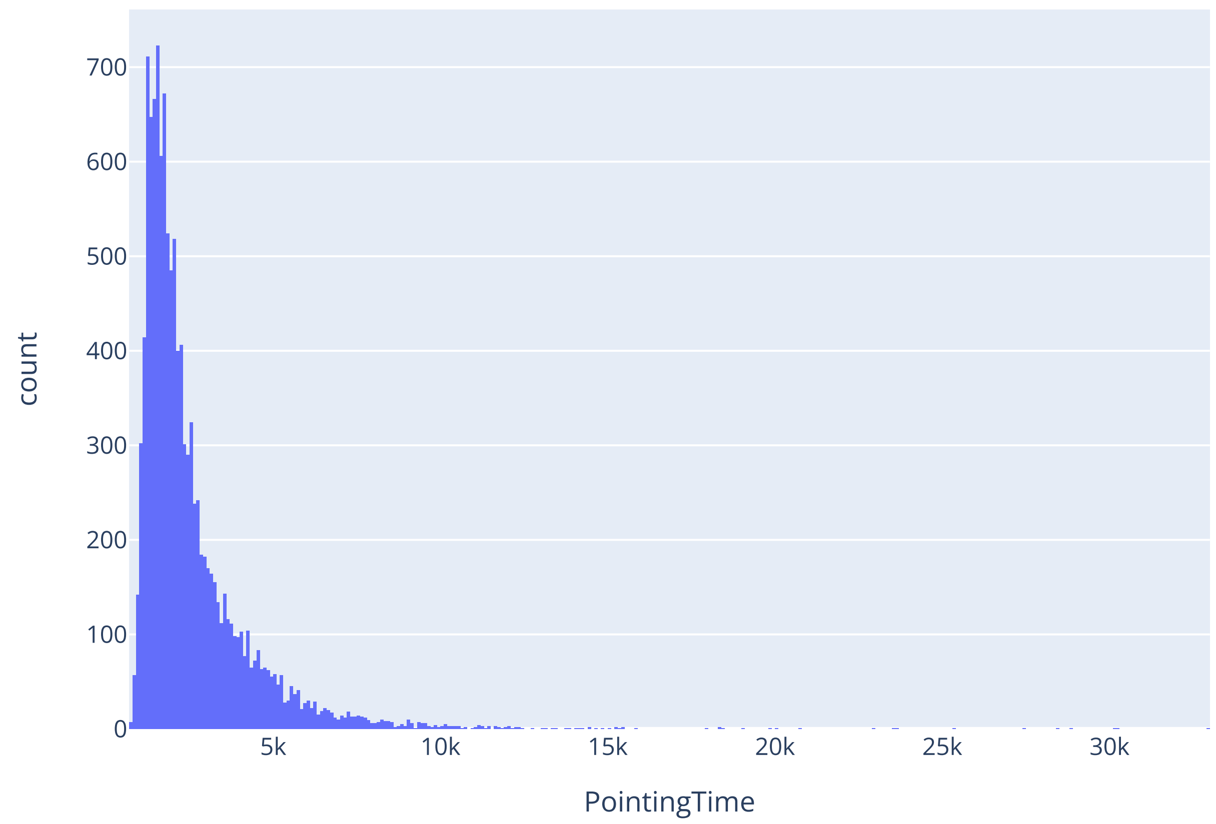
Distribution

nominal/ordinal variables

Function **`histogram`** from library **`plotly`** visualizes distributions. (https://plotly.com/python/histograms/)

We can use it to visualize the distribution of PointingTime:

```
fig = px.histogram(data, x='PointingTime')
fig.show()
```

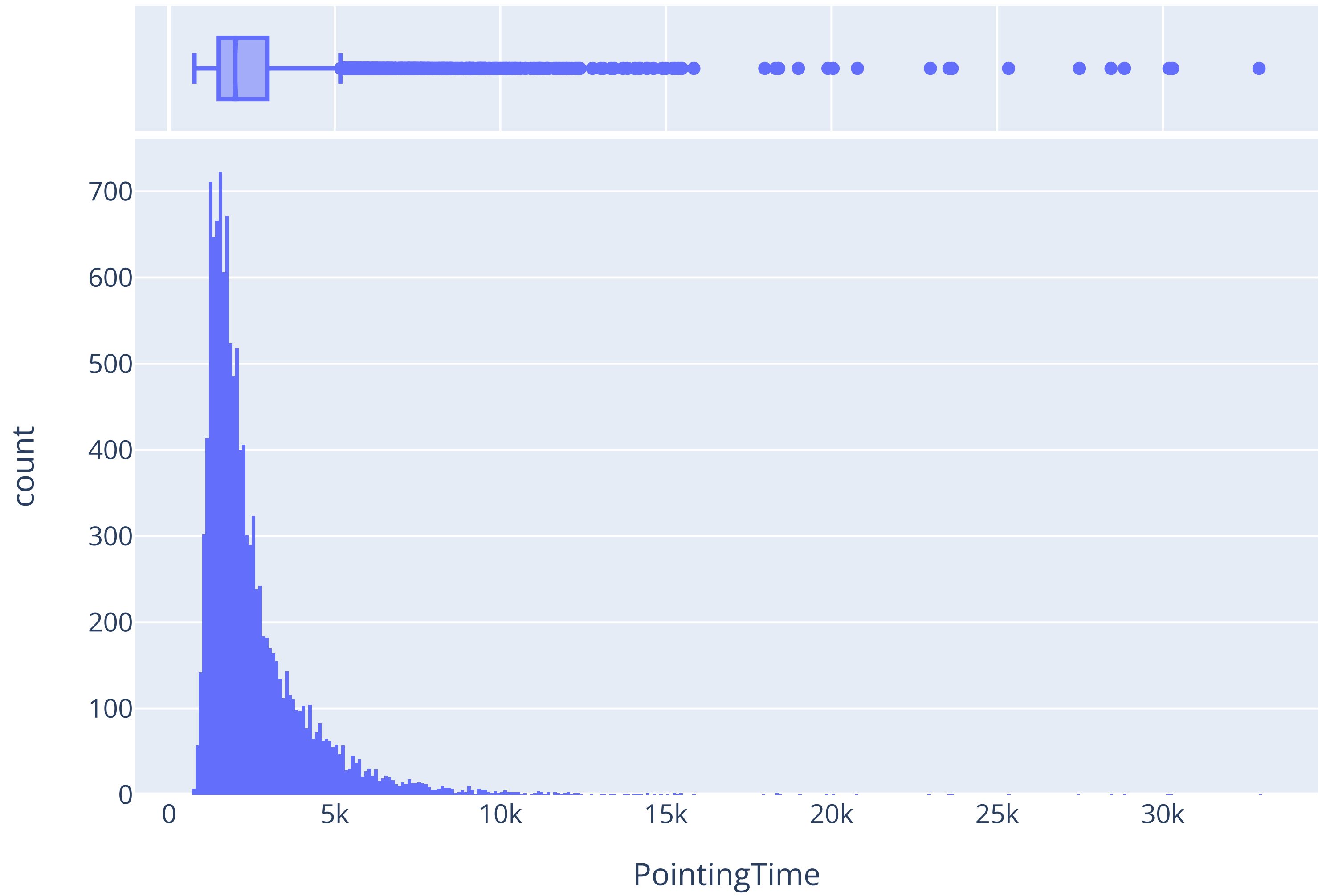


Distribution

nominal/ordinal variables

We can add a box plot:

```
fig = px.histogram(data, x='PointingTime', marginal='box')
fig.show()
```



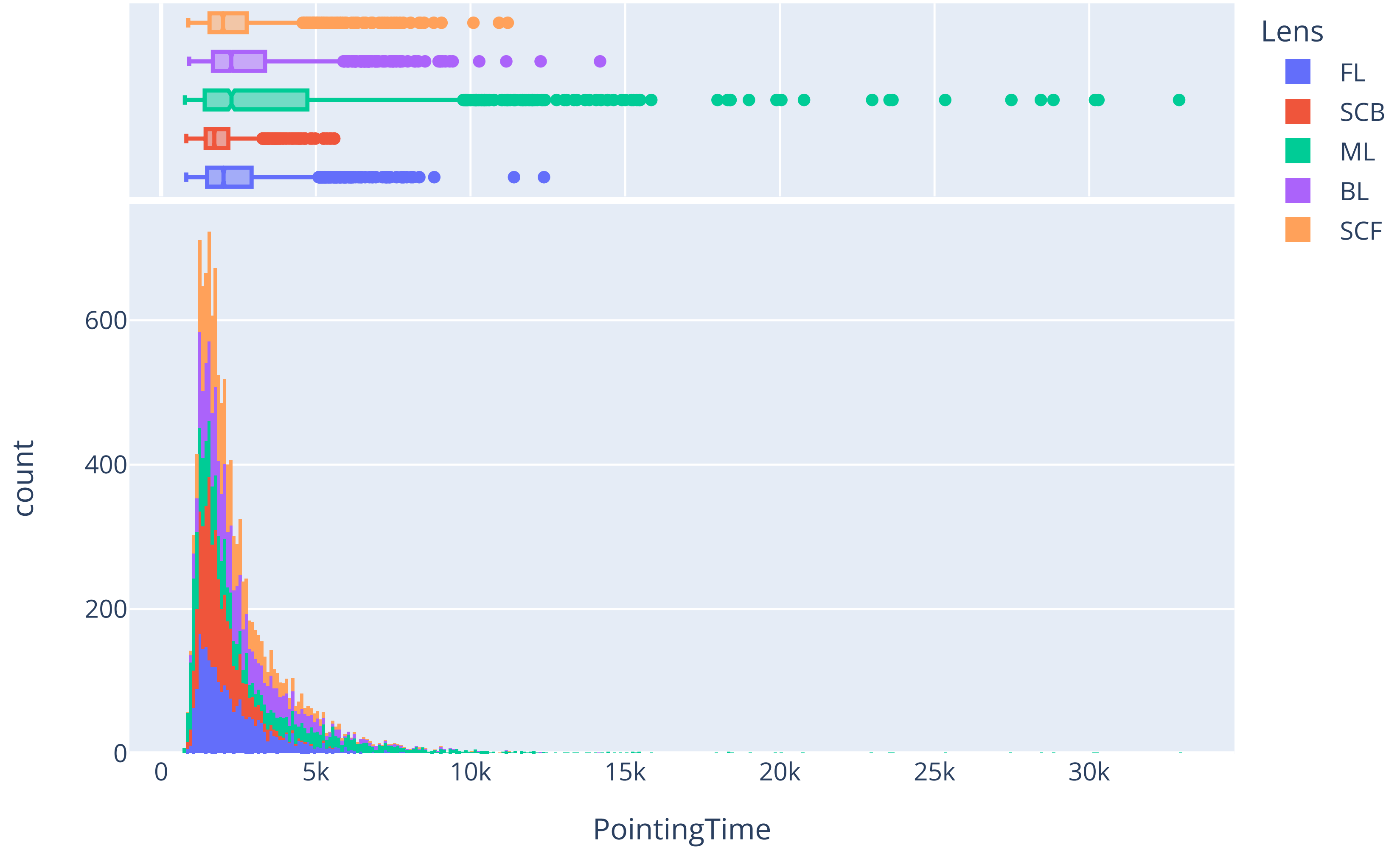
Distribution

nominal/ordinal variables

And even get a breakdown per group:

```
fig = px.histogram(data, x='PointingTime', color='Lens', marginal='box')
fig.show()

# If needed (e.g., for inclusion in a report), we can save the last plot as a PDF file
fig.write_image('images/pointingtime_distribution_with_box_per_lens.pdf')
```

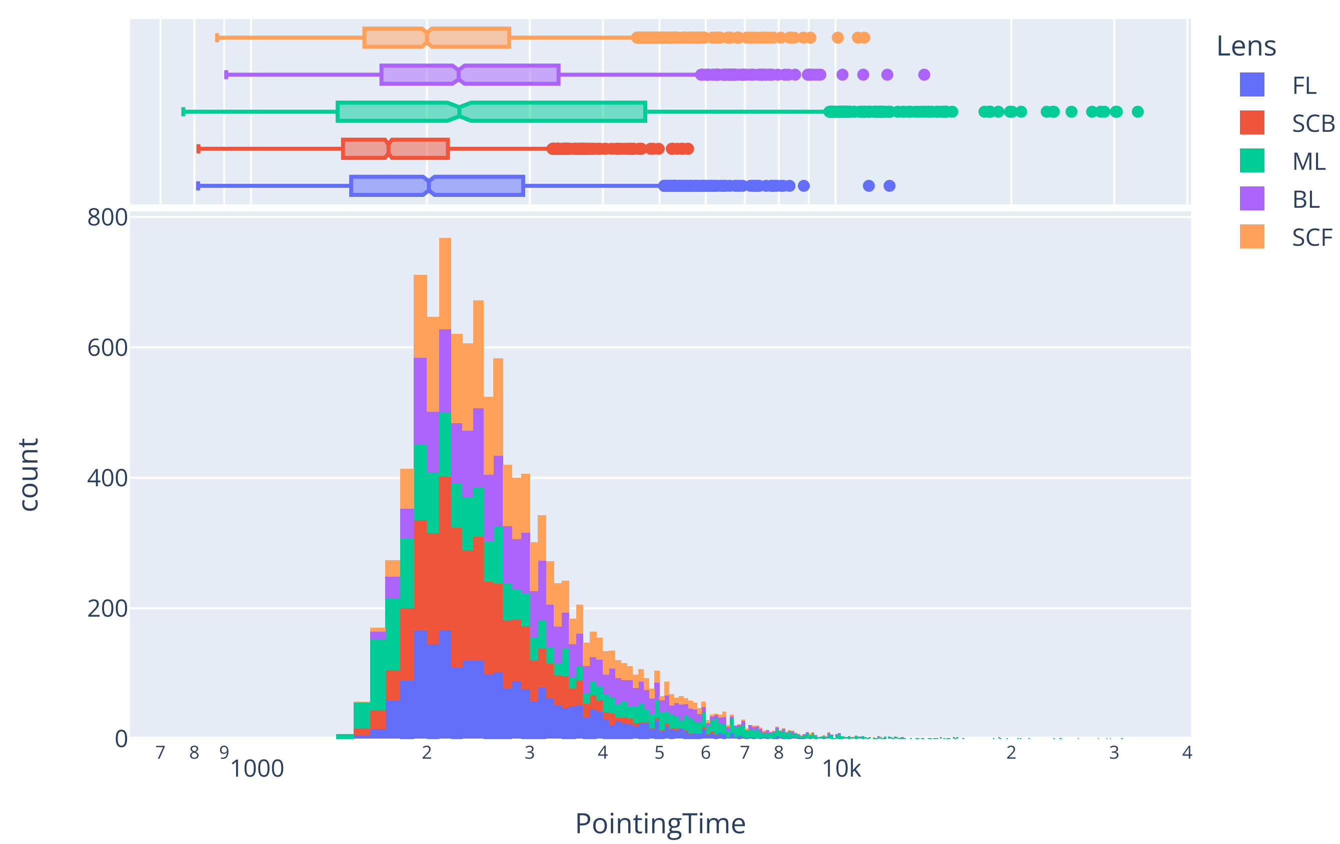


Distribution

nominal/ordinal variables

We can also *spread* the histogram by using a logarithimc scale to better visualize what happens for low values where most observations lie in our case:

```
fig = px.histogram(data, x='PointingTime', color='Lens', marginal='box', log_x=True)
fig.show()
```



Correlation / Linear Regression

ratio (continuous) variables

The correlation coefficient is a simple descriptive statistic that measures the strength of the linear relationship between two ratio (continuous) variables

We use it to test if there is a relationship between two ratio variables.

The linear regression analysis then identifies what this linear relationship is.

Correlation (r statistics)

The Pearson's correlation coefficient, r, measures how linear a relationship between two ratio variables is

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) \times (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \qquad (-1 \leq r \leq 1)$$

Usually, X is a factor, Y is a measure

r^2 is interpreted as the proportion of the variability of Y that is associated with the variability of X

$1 - r^2$ is the residual variance (not explained)

Correlation

Pearson's correlation coefficient (*r*) tells how much one variable tends to change when the other one does

$r = 0$, there is no relationship
$r > 0$, there is a trend that one variable goes up as the other one goes up
$r < 0$, there is a trend that one variable goes up as the other one goes down

correlation is a measure of dependence

correlation ≠ causality

```python
import pingouin as pg
```

Correlation with `pingouin`

Hypothesis: Pointing Time linearly goes up when Magnification factor goes up

```python
correlation_table = pg.pairwise_corr(data['Magnification'], data['PointingTime'])
correlation_table
```

	X	Y	method	alternative	n	r	CI95%	p-unc	BF10	power
0	Magnification	PointingTime	pearson	two-sided	12000	0.622792	[0.61, 0.63]	0.0	inf	1.0

```python
r2 = correlation_table['r'] * correlation_table['r']
r2
```

```
0    0.38787
Name: r, dtype: float64
```

We observe a positive relationship between variables PointingTime and Magnification with $r(12000) = 0.62$ (r^2=0.39).

Correlation and aggregation

It is also common practice to look at the correlation between two variables after having aggregated observations within experimental conditions. For example, we can consider only one mean PointingTime per LensxMagnification for each participant (i.e., aggregating replications).

Aggregating removes variance and thus mechanically increases the correlation coefficient. There is no best solution between aggregating and not aggregating. What is important is to make it clear if data were aggregated or not by reporting the number of observations (n).

```
data_agg = data.groupby(['Participant', 'Lens', 'Magnification'], as_index=False)['PointingTime'].mean()
correlation_table = pg.pairwise_corr(data_agg, columns=['Magnification','PointingTime'])
correlation_table
```

	X	Y	method	alternative	n	r	CI95%	p-unc	BF10	power
0	Magnification	PointingTime	pearson	two-sided	250	0.782979	[0.73, 0.83]	4.762157e-53	4.195e+49	1.0

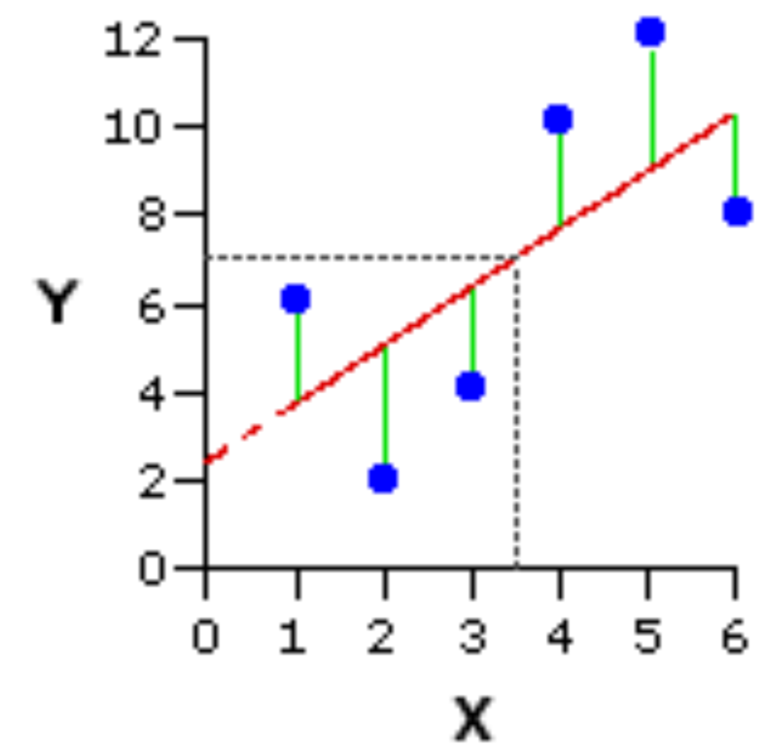
```
r2 = correlation_table['r'] * correlation_table['r']
r2
```

```
0    0.613056
Name: r, dtype: float64
```

After having aggregated observations per Participant x Lens x Magnification, we found $r(250) = 0.78$ ($r^2=0.61$) so there is a positive relation between variables PointingTime and Magnification.

Linear regression

Computing linear regression means defining the *regression line* that best fits the bivariate distribution of data points (Makes the squared vertical distances between the data points and regression line as small as possible)



Linear regression can be used as a predictive model when your experiment design is sound enough and the result of statistical tests is significant to support a cause-effect relation

```python
import pingouin as pg
```

Linear regression with `pingouin`

```python
lm = pg.linear_regression(data_agg['Magnification'], data_agg['PointingTime'])
lm
```

	names	coef	se	T	pval	r2	adj_r2	CI[2.5%]	CI[97.5%]
0	Intercept	657.534174	111.912728	5.875419	1.351668e-08	0.613056	0.611496	437.11359	877.954758
1	Magnification	264.389987	13.338075	19.822200	4.762157e-53	0.613056	0.611496	238.11964	290.660335

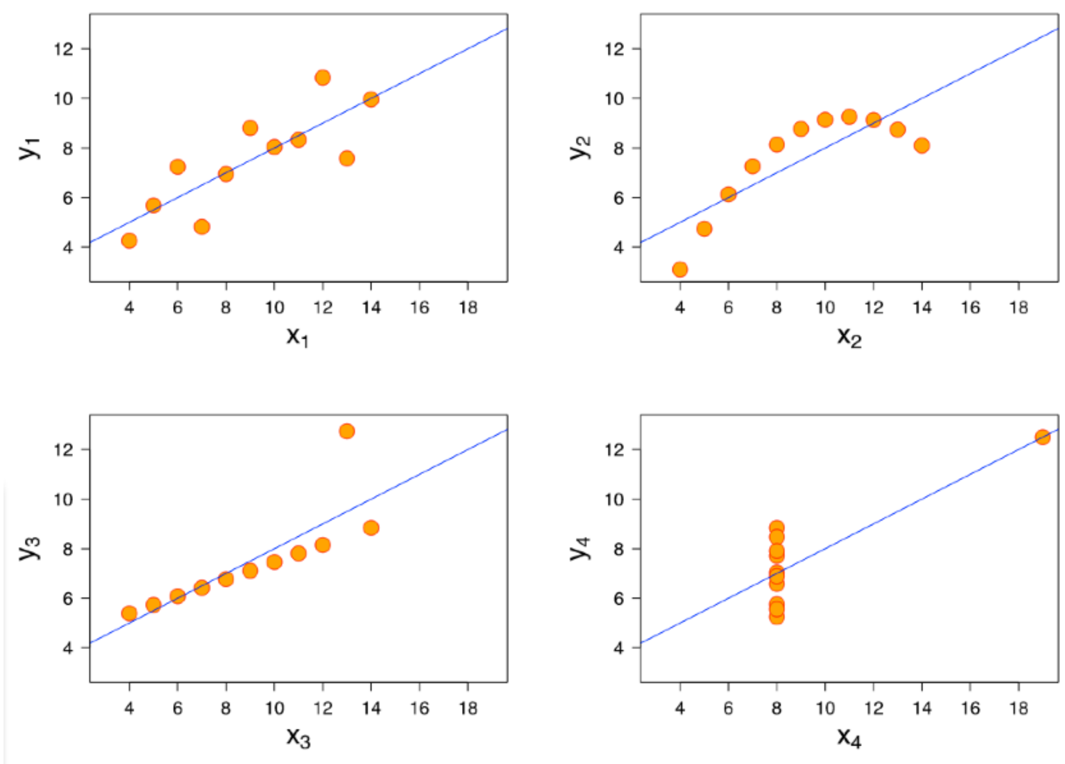
Final report:

After having aggregated observations per ParticipantxLensxMagnification, we found $r(250) = 0.78$ ($r^2=0.61$) so there is a positive relation between variables PointingTime and Magnification. Predicted PointingTime in ms is equal to 657 + 264 x Magnification.

Linear regression and visualization

Linear regression must be interpreted with caution

Needs to be visualised



Famous example: Anscombe's quartet (above)
same linear regression line but very different datasets…

Linear regression and visualization

```
fig = px.scatter(
    data_agg, x='Magnification', y='PointingTime', opacity=0.65,
    trendline='ols'
)
fig.show()
```