# Lectures

Prof. Dr. Markus Löcher

2023-05-28

# Table of contents

# Preface

On this html page you will find a collection of the lectures from Digital Literacy I: Coding A SoSe-2023.

# Lecture 3

In this lecture we will get to know and become experts in:

1. Introduction to numpy
    - DataCamp, Introduction to Python, Chap 4
    - Multiple Dimensions
    - Data Summaries in numpy
2. Introduction to **Simulating Probabilistic Events**
    - Generating Data in numpy

```
import numpy as np
from numpy.random import default_rng
```

## 0.1 Introduction to numpy

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

1. Vectorized, fast mathematical operations.
2. Key features of NumPy is its N-dimensional array object, or ndarray

```
height = [1.79, 1.85, 1.95, 1.55]
weight = [70, 80, 85, 65]

#bmi = weight/height**2
```

```
height = np.array([1.79, 1.85, 1.95, 1.55])
weight = np.array([70, 80, 85, 65])

bmi = weight/height**2
bmi
```

```
array([21.84700852, 23.37472608, 22.35371466, 27.05515088])
```

### 0.1.1 Multiple Dimensions

are handled naturally by numpy, e.g.

```
hw1 = np.array([height, weight])
print(hw1)
print(hw1.shape)
hw2 = hw1.transpose()
print(hw2)
print(hw2.shape)
```

```
[[ 1.79  1.85  1.95  1.55]
 [70.   80.   85.   65.  ]]
(2, 4)
[[ 1.79 70.  ]
 [ 1.85 80.  ]
 [ 1.95 85.  ]
 [ 1.55 65.  ]]
(4, 2)
```

### 0.1.2 Accessing array elements

is similar to lists but allows for multidimensional index:

```
print(hw2[0,1])
```

```
70.0
```

```
print(hw2[:,0])
```

```
[1.79 1.85 1.95 1.55]
```

```
print(hw2[0])
#equivalent to
print(hw2[0,:])
#shape:
print(hw2[0].shape)
```

```
[ 1.79 70.  ]
[ 1.79 70.  ]
(2,)
```

To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
print(hw2[[2,0,1]])
```

```
[[ 1.95 85.  ]
 [ 1.79 70.  ]
 [ 1.85 80.  ]]
```

Negative indices

```
print(hw2)
print("Using negative indices selects rows from the end:")
print(hw2[[-2,-1]])
```

```
[[ 1.79 70.  ]
 [ 1.85 80.  ]
 [ 1.95 85.  ]
 [ 1.55 65.  ]]
Using negative indices selects rows from the end:
[[ 1.95 85.  ]
 [ 1.55 65.  ]]
```

You can pass multiple slices just like you can pass multiple indexes:

```
hw2[:2,:1]
```

```
array([[1.79],
       [1.85]])
```

#### 0.1.2.1 Reshaping

```
np.arange(32).reshape((8, 4))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

#### 0.1.2.2 Boolean indexing

```
height_gt_185 = hw2[:,0]>1.85
print(height_gt_185)
print(hw2[height_gt_185,1])
```

```
[False False  True False]
[85.]
```

numpy arrays cannot contain elements with different types. If you try to build such a list, some of the elements' types are changed to end up with a homogeneous list. This is known as **type coercion**.

```
print(np.array([True, 1, 2]))
print(np.array(["True", 1, 2]))
print(np.array([1.3, 1, 2]))
```

```
[1 1 2]
['True' '1' '2']
[1.3 1.  2. ]
```

Lots of extra useful functions!

```
np.zeros((2,3))
#np.ones((2,3))
```

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

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
np.column_stack([height, weight])
```

```
array([[ 1.79, 70.  ],
       [ 1.85, 80.  ],
       [ 1.95, 85.  ],
       [ 1.55, 65.  ]])
```

## 0.2 Data Summaries in numpy

We can compute simple statistics:

```
print(np.mean(hw2))
print(np.mean(hw2, axis=0))
```

```
38.3925
[ 1.785 75.   ]
```

```
print(np.unique([1,1,2,1,2,3,2,2,3]))

print(np.unique([1,1,2,1,2,3,2,2,3], return_counts=True))
```

```
[1 2 3]
(array([1, 2, 3]), array([3, 4, 2], dtype=int64))
```

## 0.3 Introduction to Simulating Probabilistic Events

### 0.3.1 Generating Data in numpy

Meet your friends:

1. `np.random.permutation`: Return a random permutation of a sequence, or return a permuted range
2. `np.random.integers`: Draw random integers from a given low-to-high range
3. `np.random.choice`: Generates a random sample from a given 1-D array

```
# Do this (new version)
from numpy.random import default_rng
rng = default_rng()

x= np.arange(10)
print(x)
print(rng.permutation(x))
print(rng.permutation(list('intelligence')))
```

```
[0 1 2 3 4 5 6 7 8 9]
[6 7 9 4 1 0 3 8 2 5]
['t' 'c' 'n' 'l' 'e' 'n' 'i' 'e' 'e' 'l' 'i' 'g']
```

```
print(rng.integers(0,10,5))
print(rng.integers(0,10,(5,2)))
```

```
[7 9 7 9 4]
[[9 0]
 [8 6]
 [6 7]
 [0 5]
 [1 5]]
```

```
rng.choice(x,4)
```

```
array([8, 5, 1, 4])
```

### 0.3.2 Examples:

- Spotify playlist
- Movie List

```
movies_list = ['The Godfather', 'The Wizard of Oz', 'Citizen Kane', 'The Shawshank Redempt

# pick a random choice from a list of strings.
movie = rng.choice(movies_list,2)
print(movie)
```

```
['The Shawshank Redemption' 'The Godfather']
```

## 0.4 Birthday "Paradox"

Please enter your birthday on google drive https://forms.gle/CeqyRZ4QzWRmJFvs9

How many people do you think will share a birthday? Would that be a rare, highly unusual event?

```
'''#!pip install openpyxl
import pandas as pd
import numpy as np

url = "https://docs.google.com/spreadsheets/d/1zo8l6xPKm8EPAobHHJXRKjq9vtOnAegMkg_VAhcZ6kY
gResponses = pd.read_excel(url)
#df["date_3"] = pd.to_datetime(df["date_3"])
print(gResponses.dtypes)
#I am not interested in the year, so I am seting it equal to all
gResponses['birthday'] = gResponses['birthday'].apply(lambda x: x.replace(year = 2000))
gResponses'''
```

```
'#!pip install openpyxl\nimport pandas as pd\nimport numpy as np\n\nurl = "https://docs.googl
```

How can we find out how likely it is that across $n$ folks in a room at least two share a birthday?

Hint: can we put our random number generators to task ?

```
# Can you simulate 25 birthdays?
from numpy.random import default_rng
rng = default_rng()


#initialize it to be the empty list:
shardBday = []

n = 40

PossibleBDays = np.arange(1,366)
#now "draw" 25 random bDays:
for i in range(1000):# is the 1000 an important number ??
#no it only determines the precision of my estimate !!
  ran25Bdays = rng.choice(PossibleBDays, n, replace = True)
  #it is of utmost importance to allow for same birthdays !!
  #rng.choice(PossibleBDays, 366, replace = False)
  x , cts = np.unique(ran25Bdays ,return_counts=True)
  shardBday = np.append(shardBday, np.sum(cts>1))#keep this !!
  #shardBday = np.sum(cts>1)
```

```
#np.sum(shardBday>0)/1000
np.mean(shardBday > 0)

#shardBday = 2
```

```
0.893
```

```
5 != 3 #not equal
```

```
True
```

```
#Boolean indexing !!
x[cts > 1]
```

```
array([ 71, 192])
```

```
x[23]
```

182

```
#can you design a coin flip with an arbitary probability p = 0.25
#simulate 365 days with a 1/4 chance of being sunny

#fair coin
coins = np.random.randint(0,2,365)

np.unique(coins, return_counts=True)
```

```
(array([0, 1]), array([189, 176], dtype=int64))
```

---

### Tossing dice and coins

Let us toss many dice or coins to find out: - the average value of a six-faced die - the variation around the mean when averaging - the probability of various "common hands" in the game Liar's Dice: * Full house: e.g., 66111 * Three of a kind: e.g., 44432 * Two pair: e.g., 22551 * Pair: e.g., 66532

Some real world problems: 1. **Overbooking flights**: airlines 2. **Home Office** days: planning office capacities and minimizing social isolation


```
{=html} <!-- quarto-file-metadata:
eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiYm9vaOl0ZW1OdW1iZ
-->
```


# Lecture 4 {.unnumbered}

In this lecture we will get to know and become experts in:

1. Introduction to pandas * Data Frames * Slicing * Counting and Summary Statistics * Handling Files

Relevant DataCamp lessons:

* Data manipulation with pandas, Chaps 1-4 * Matplotlib, Chap 1

::: {.cell}

## Introduction to pandas

While numpy offers a lot of powerful numerical capabilities it lacks some of the necessary convenience and natural of handling data as we encounter them. For example, we would typically like to - mix data types (strings, numbers, categories, Boolean, …) - refer to columns and rows by names - summarize and visualize data in efficient pivot style manners
All of the above (and more) can be achieved easily by extending the concept of an array (or a matrix) to a so called **dataframe**.
There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:
::: {.cell execution-
Info='{"elapsed":427,"status":"ok","timestamp":1683024188638,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='4a41f859-ce49-46a1-e308-5228df560828'}
::: {.cell-output .cell-output-display
execution_count=5}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }

.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const element = document.querySelector('#df-7efc9943-a94c-4abe-8400-c307ef90ad39'); const dataTable = await google.colab.kernel.invokeFunction('convertToInteractive', [key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' + 'data table notebook' + ' to learn more about interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
"'
::: :::
::: {.cell}
::: {.cell execution-Info='{"elapsed":314,"status":"ok","timestamp":1683024441955,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-120}'
outputId='ada23816-c0ea-4474-d85f-22305d57758c'}
::: {.cell-output .cell-output-display execution_count=9}
### Subsetting/Slicing
We first need to understand the attributes **index** (=rownames) and **columns** (= column names):
::: {.cell outputId='eadfec57-e302-45bd-c74a-b93be6fd4c2a'}
::: {.cell-output .cell-output-display execution_count=23}
::: {.cell execution-Info='{"elapsed":308,"status":"ok","timestamp":1683024590608,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-120}'
outputId='4714ea11-ad8c-421e-96e7-e44d7403e4ac'}
::: {.cell-output .cell-output-stdout}
::: {.cell execution-Info='{"elapsed":323,"status":"ok","timestamp":1683024115682,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-120}'
outputId='a3942774-31f5-4631-e5ff-d8d2b06e4452'}

::: {.cell-output .cell-output-display
execution_count=3}
::: {.cell execution-
Info='{"elapsed":610,"status":"ok","timestamp":1683024927760,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='6c362666-e5f7-4b69-d705-e77380be158d'}
::: {.cell-output .cell-output-display
execution_count=18}
### Asking for rows
Unfortunately, we cannot use the simple `[row,col]` notation that we are used to from numpy arrays. (Try asking for `frame[0,1]`)
Instead, row subsetting can be achieved with either the `.loc()` or the `.iloc()` methods. The latter takes integers, the former indices:
::: {.cell execution-
Info='{"elapsed":398,"status":"ok","timestamp":1683024602187,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='fac5a1e4-297c-437a-d37a-e32a58881c03'}
::: {.cell-output .cell-output-display
execution_count=12}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }

.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const element = document.querySelector(‘#df-a1942c22-8fb1-498e-95ae-0dd27211201f’); const dataTable = await google.colab.kernel.invokeFunction(‘convertToInteractive’, [key], {}); if (!dataTable) return;
const docLinkHtml = ‘Like what you see? Visit the’ + ‘data table notebook’ + ’ to learn more about interactive tables.’; element.innerHTML = ’‘;
dataTable[’output_type’] = ’display_data’; await google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement(‘div’);
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
“‘
::: :::
::: {.cell outputId=‘ec6b2a53-1bf7-4385-84ae-72602fc85c2a’}
::: {.cell-output .cell-output-display execution_count=30}
::: {.cell execution-Info=‘{“elapsed”:380,“status”:“ok”,“timestamp”:1683010563095,“user”:{“displayName”:“Ma
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-120}’ outputId=‘aeea9b53-c0ba-4210-84b7-dd521975a80b’}
::: {.cell-output .cell-output-display execution_count=10}
::: {.cell execution-Info=‘{“elapsed”:320,“status”:“ok”,“timestamp”:1683024851857,“user”:{“displayName”:“Ma
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-120}’ outputId=‘999dffbd-f606-4533-8d7c-8385a2e83f82’}
::: {.cell-output .cell-output-stdout}
::: {.cell execution-Info=‘{“elapsed”:373,“status”:“ok”,“timestamp”:1683024804583,“user”:{“displayName”:“Ma
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-120}’ outputId=‘c67188a7-c6db-48a8-9de5-bada14c2c063’}
::: {.cell-output .cell-output-display execution_count=16}
“‘{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }

.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const
element = document.querySelector('#df-c88550dd-
144b-455c-895e-e1d55f1d4036'); const dataTable =
await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' +
'data table notebook' + ' to learn more about
interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await
google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
‘‘‘
::: :::
::: {.cell}
::: {.cell
outputId='a0402acf-d77b-4e18-9ff8-4e90d3356b27'}
::: {.cell-output .cell-output-display
execution_count=39}
### Asking for columns

::: {.cell execution-
Info='{"elapsed":521,"status":"ok","timestamp":1683010794284,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='371e5e00-b584-4258-8d5e-36fa37333d86'}
::: {.cell-output .cell-output-display
execution_count=12}
A column in a DataFrame can be retrieved MUCH
easier: as a Series either by dictionary-like notation or
by using the dot attribute notation:
::: {.cell
outputId='732d0d02-3a0a-493f-d8d8-ff905c29cd87'}
::: {.cell-output .cell-output-display
execution_count=17}
::: {.cell
outputId='7ca7fea4-8e89-4925-9c05-59713798f068'}
::: {.cell-output .cell-output-display
execution_count=18}
### Summary Stats
Just like in numpy you can compute sums, means,
counts and many other summaries along rows and
columns, by specifying the `axis` argument:
::: {.cell execution-
Info='{"elapsed":4,"status":"ok","timestamp":1683025066198,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='212a4691-0400-40a2-c35a-a163be5ba112'} "'
{.python .cell-code} height = np.array([1.79, 1.85, 1.95,
1.55]) weight = np.array([70, 80, 85, 65]) hw =
np.array([height, weight]).transpose()
hw "'
::: {.cell-output .cell-output-display
execution_count=20}
::: {.cell execution-
Info='{"elapsed":361,"status":"ok","timestamp":1683025151434,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='50c15fbb-8402-4db8-d3af-c0b5f2469cfe'}
::: {.cell-output .cell-output-stdout}

::: {.cell execution-
Info='{"elapsed":313,"status":"ok","timestamp":1683025183810,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='20e33e59-4133-42d5-e36a-b68f66780832'}
::: {.cell-output .cell-output-stdout}
Can you extract:
0. All weights 1. Peter's height 2. Bee's full info 3. the
average height 4. get all persons with height greater
than 180cm
::: {.cell}
::: {.cell execution-
Info='{"elapsed":377,"status":"ok","timestamp":1683012383970,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='1c45ebe6-37d5-430d-fe59-87f9fca7e65b'}
::: {.cell-output .cell-output-stdout}
Some methods are neither reductions nor
accumulations. `describe` is one such example,
producing multiple summary statistics in one shot:
::: {.cell
outputId='9e73eec1-32d2-486b-8671-0d631653595a'}
::: {.cell-output .cell-output-display
execution_count=43}
"'{=html}
.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }
::: :::
## Built in data sets
## Gapminder Data
https://www.gapminder.org/fw/world-health-chart/
https://www.ted.com/talks/hans_rosling_the_best_stats_you_ve_ever_seen#t-
241405
> You've never seen data presented like this. With the
drama and urgency of a sportscaster, statistics guru
Hans Rosling debunks myths about the so-called
"developing world."
::: {.cell}

::: {.cell execution-
Info='{"elapsed":8,"status":"ok","timestamp":1683013486502,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='974adae1-7cde-43ff-b4bc-1d4c28403147'}
::: {.cell-output .cell-output-display
execution_count=37}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }


.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const
element = document.querySelector('#df-acc59b03-
c8fa-4715-afe8-4eb0bbcb7128'); const dataTable = await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' +
'data table notebook' + ' to learn more about
interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await
google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
"'
::: :::

::: {.cell execution-
Info='{"elapsed":774,"status":"ok","timestamp":1683013794713,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='0c83252b-2176-4c75-b0d5-81cf7dfe9d1b'} "'
{.python .cell-code} #find the unique years
#get the years: gapminder["year"]
np.unique(gapminder.year) "'
::: {.cell-output .cell-output-display
execution_count=41}
::: {.cell execution-
Info='{"elapsed":1099,"status":"ok","timestamp":1683014342194,"user":{"displayName":"M
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='2e6fc695-c0a6-4fa3-d60a-240b8eba9324'} "'
{.python .cell-code} #get all rows with year 1952:
#Hint: #either use Boolean subsetting
gapminder["year"] == 1952
gapminder[gapminder["year"] == 1952] #or use an
index !!
"'
::: {.cell-output .cell-output-display
execution_count=43}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }


.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const element = document.querySelector(‘#df-4e372233-0124-4621-8cee-92d859b78962’); const dataTable = await google.colab.kernel.invokeFunction(‘convertToInteractive’, [key], {}); if (!dataTable) return; const docLinkHtml = ‘Like what you see? Visit the’ + ‘data table notebook’ + ’ to learn more about interactive tables.’; element.innerHTML = ’‘; dataTable[’output_type’] = ’display_data’; await google.colab.output.renderOutput(dataTable, element); const docLink = document.createElement(‘div’); docLink.innerHTML = docLinkHtml; element.appendChild(docLink); }
“‘
::: :::
## Handling Files
Get to know your friends
* `pd.read_csv` * `pd.read_table` * `pd.read_excel`
::: {.cell execution-Info=‘{“elapsed”:1235,“status”:“ok”,“timestamp”:1682343627220,“user”:{“displayName”:“M Loecher”,“userId”:“02488037228155275753”},“user_tz”:-120}’ outputId=‘34256d0e-6290-43ef-c208-e5e1178a3a01’} “‘ {.python .cell-code} ’‘’url = “https://drive.google.com/file/d/1oIvCdN15UEwt4dCyjkArekHnTrivN43v/view?usp=share_ url=’https://drive.google.com/uc?id=’ + url.split(‘/’)[-2] gapminder = pd.read_csv(url, index_col=0) gapminder.head()’’’
“‘
::: {.cell-output .cell-output-display execution_count=2}
“‘{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }


.colab-df-convert { background-color: #E8F0FE; border: none; border-radius: 50%; cursor: pointer; display: none; fill: #1967D2; height: 32px; padding: 0 0 0 0; width: 32px; }

.colab-df-convert:hover { background-color: #E2EBFA; box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px 3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color: #3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover { background-color: #434B5C; box-shadow: 0px 1px 3px 1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const element = document.querySelector('#df-42dc89b0-0da4-4c66-a3f0-42fcc92029a5'); const dataTable = await google.colab.kernel.invokeFunction('convertToInteractive', [key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' + 'data table notebook' + ' to learn more about interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
‘‘‘
::: :::
::: {.cell outputId='0c07ed68-f605-407f-b3fd-78dd9c63dc48'}
::: {.cell-output .cell-output-display execution_count=71}
“‘{=html}
.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }
::: :::
::: {.cell outputId='6379a0f5-d837-4c9c-9253-926496fc87bc'}
::: {.cell-output .cell-output-display execution_count=63}
::: {.cell outputId='2f46c987-3383-4731-b1ae-fc5249a15d85'}
::: {.cell-output .cell-output-display execution_count=75}

::: {.cell
outputId='def2732a-7153-4a42-a486-5c7194958f4c'}
::: {.cell-output .cell-output-stdout}
::: {.cell-output .cell-output-stderr}
::: {.cell-output .cell-output-display
execution_count=54}
"'{=html}
.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }
::: :::
**Sort the index before you slice!!**
Choose a time range and specific countries
::: {.cell
outputId='614b674e-df99-42e7-c9e4-a68105d86180'}
::: {.cell-output .cell-output-display
execution_count=91}
"'{=html}
.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }
::: :::
::: {.cell
outputId='8a594a1d-ea6c-439a-8de9-f356414dc8cd'}
::: {.cell-output .cell-output-display
execution_count=93}

{=html} <!-- quarto-file-metadata:
eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiYm9vaOlOZW1OdW1iZ
-->
# Lecture 5 {.unnumbered}
In this lecture we will get to know and become experts in:
1. Data Manipulation with pandas * Handling Files * Counting and Summary Statistics * Grouped Operations 2. Plotting * matplotlib * pandas
And if you want to delve deeper, look at the Advanced topics
Relevant DataCamp lessons:
* Data manipulation with pandas, Chaps 2 and 4 * Matplotlib, Chap 1
::: {.cell}
## Data Manipulation with pandas

While we have seen panda's ability to (i) mix data types (strings, numbers, categories, Boolean, …) and (ii) refer to columns and rows by names, this library offers a lot more powerful tools for efficiently gaining insights from data, e.g.
- summarize/aggregate data in efficient pivot style manners - handling missing values - visualize/plot data
::: {.cell}
## Handling Files
Get to know your friends
* `pd.read_csv` * `pd.read_table` * `pd.read_excel`
But before that we need to connect to our Google drives ! (more instructions can be found here)
::: {.cell execution-
Info='{"elapsed":4,"status":"ok","timestamp":1683555854441,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='2c49553e-941c-4125-c2c3-52e6b4c730c2'}
::: {.cell-output .cell-output-display
execution_count=5}
Counting and Summary Statistics
::: {.cell
outputId='af1ef667-f29d-4c92-86b8-2fd1c6ffe159'}
::: {.cell-output .cell-output-display
execution_count=71}
"'{=html}
.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }
::: :::
::: {.cell execution-
Info='{"elapsed":284,"status":"ok","timestamp":1683556772006,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='2a36faee-6c99-4bc8-bc6a-0bb8785462e3'}
::: {.cell-output .cell-output-display
execution_count=10}
## Grouped Operations
The gapminder data is a good example for wanting to apply functions to subsets to data that correspond to categories, e.g. * by year * by country * by continent
The powerful pandas `.groupby()` method enables exactly this goal rather elegantly and efficiently.

First, think how you could possibly compute the average GDP seprataley for each continent. The `numpy.mean(..., axis=...)` will not help you. Instead you will have to manually find all continents and then use Boolean logic:
::: {.cell execution-
Info='{"elapsed":212,"status":"ok","timestamp":1683439733094,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='15481d0e-0491-467c-9721-db0329d72684'}
::: {.cell-output .cell-output-display
execution_count=10}
::: {.cell} "' {.python .cell-code} AfricaRows =
gapminder["continent"]=="Africa"
gapminder[AfricaRows]["gdpPercap"].mean()
"' :::
::: {.cell execution-
Info='{"elapsed":223,"status":"ok","timestamp":1683556339119,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='bc685147-f274-4a07-d91d-8db44044b594'}
::: {.cell-output .cell-output-display
execution_count=7}
Instead, we should embrace the concept of **grouping by a variable**
::: {.cell execution-
Info='{"elapsed":4,"status":"ok","timestamp":1683556453453,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='e401809a-8936-441b-e045-f08629f010e2'}
::: {.cell-output .cell-output-stderr}
::: {.cell-output .cell-output-display
execution_count=8}
::: {.cell execution-
Info='{"elapsed":8,"status":"ok","timestamp":1683556842613,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='18a97f92-9784-4722-c2be-3c52c2f3913e'}
::: {.cell-output .cell-output-stderr}
::: {.cell-output .cell-output-display
execution_count=12}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }

.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const
element = document.querySelector('#df-9e17e362-14cc-
4068-a4ec-3ebcc0a6afcc'); const dataTable = await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' +
'data table notebook' + ' to learn more about
interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await
google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
```
::: :::
::: {.cell execution-
Info='{"elapsed":4,"status":"ok","timestamp":1683556960873,"user":{"displayName":"Mark
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='1cc02fd3-3a03-4576-8f37-393660f66c95'}
::: {.cell-output .cell-output-display
execution_count=16}
```

::: {.cell execution-
Info='{"elapsed":947,"status":"ok","timestamp":1683557058870,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='bd3688a0-cbb4-4529-af96-d9fd89761ad9'}
::: {.cell-output .cell-output-display
execution_count=17}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }


.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const
element = document.querySelector('#df-faa90c88-6b88-
43cc-9e80-c69e3f47eec9'); const dataTable = await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' +
'data table notebook' + ' to learn more about
interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await
google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
"'
::: :::

::: {.cell execution-
Info='{"elapsed":320,"status":"ok","timestamp":1683442764530,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='1b15b924-c0f4-46ce-f5cc-884c324dfa60'}
::: {.cell-output .cell-output-display
execution_count=18}
"'{=html}

.dataframe tbody tr th { vertical-align: top; }
.dataframe thead th { text-align: right; }


.colab-df-convert { background-color: #E8F0FE;
border: none; border-radius: 50%; cursor: pointer;
display: none; fill: #1967D2; height: 32px; padding: 0
0 0 0; width: 32px; }
.colab-df-convert:hover { background-color: #E2EBFA;
box-shadow: 0px 1px 2px rgba(60, 64, 67, 0.3), 0px 1px
3px 1px rgba(60, 64, 67, 0.15); fill: #174EA6; }
[theme=dark] .colab-df-convert { background-color:
#3B4455; fill: #D2E3FC; }
[theme=dark] .colab-df-convert:hover {
background-color: #434B5C; box-shadow: 0px 1px 3px
1px rgba(0, 0, 0, 0.15); filter: drop-shadow(0px 1px 2px
rgba(0, 0, 0, 0.3)); fill: #FFFFFF; }

async function convertToInteractive(key) { const
element = document.querySelector('#df-2c5b6b6a-
c411-4919-aa90-4a512fa77b69'); const dataTable =
await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what you see? Visit the' +
'data table notebook' + ' to learn more about
interactive tables.'; element.innerHTML = '';
dataTable['output_type'] = 'display_data'; await
google.colab.output.renderOutput(dataTable, element);
const docLink = document.createElement('div');
docLink.innerHTML = docLinkHtml;
element.appendChild(docLink); }
"'

::: :::
::: {.cell}
::: {.cell execution-
Info='{"elapsed":211,"status":"ok","timestamp":1683443085773,"user":{"displayName":"Ma
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='e3ea1161-2b2e-481e-eb94-77195c4ae250'}
::: {.cell-output .cell-output-display
execution_count=24}

### 0.4.1 Titanic data

```
# Since pandas does not have any built in data, I am going to "cheat" and
# make use of the `seaborn` library
import seaborn as sns

titanic = sns. load_dataset('titanic')
titanic["3rdClass"] = titanic["pclass"]==3
titanic["male"] = titanic["sex"]=="male"

titanic
```

| | survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | man | True |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | woman | False |
| 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | woman | False |
| 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | woman | False |
| 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | man | True |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 886 | 0 | 2 | male | 27.0 | 0 | 0 | 13.0000 | S | Second | man | True |
| 887 | 1 | 1 | female | 19.0 | 0 | 0 | 30.0000 | S | First | woman | False |
| 888 | 0 | 3 | female | NaN | 1 | 2 | 23.4500 | S | Third | woman | False |
| 889 | 1 | 1 | male | 26.0 | 0 | 0 | 30.0000 | C | First | man | True |
| 890 | 0 | 3 | male | 32.0 | 0 | 0 | 7.7500 | Q | Third | man | True |

```
#overall survival rate
titanic.survived.mean()
```

```
0.3838383838383838
```

Tasks:

- compute the proportion of survived separately for
  - male/female
  - the three classes
  - Pclass and sex
- compute the mean age separately for male/female

```
#I would like to compute the mean survical seprately for each group
bySex = titanic.groupby("sex")
#here I am specifically asking for the mean
bySex["survived"].mean()
#if you want multiple summaries, you can list them all inside the agg():
bySex["survived"].agg([min, max, np.mean ])
```

| sex | min | max | mean |
|---|---|---|---|
| female | 0 | 1 | 0.742038 |
| male | 0 | 1 | 0.188908 |

```
#I would like to compute the mean survical seprately for each group
bySexPclass = titanic.groupby(["pclass", "sex"])
#here I am specifically asking for the mean
bySexPclass["survived"].mean()
```

```
pclass  sex
1       female    0.968085
        male      0.368852
2       female    0.921053
        male      0.157407
3       female    0.500000
        male      0.135447
Name: survived, dtype: float64
```

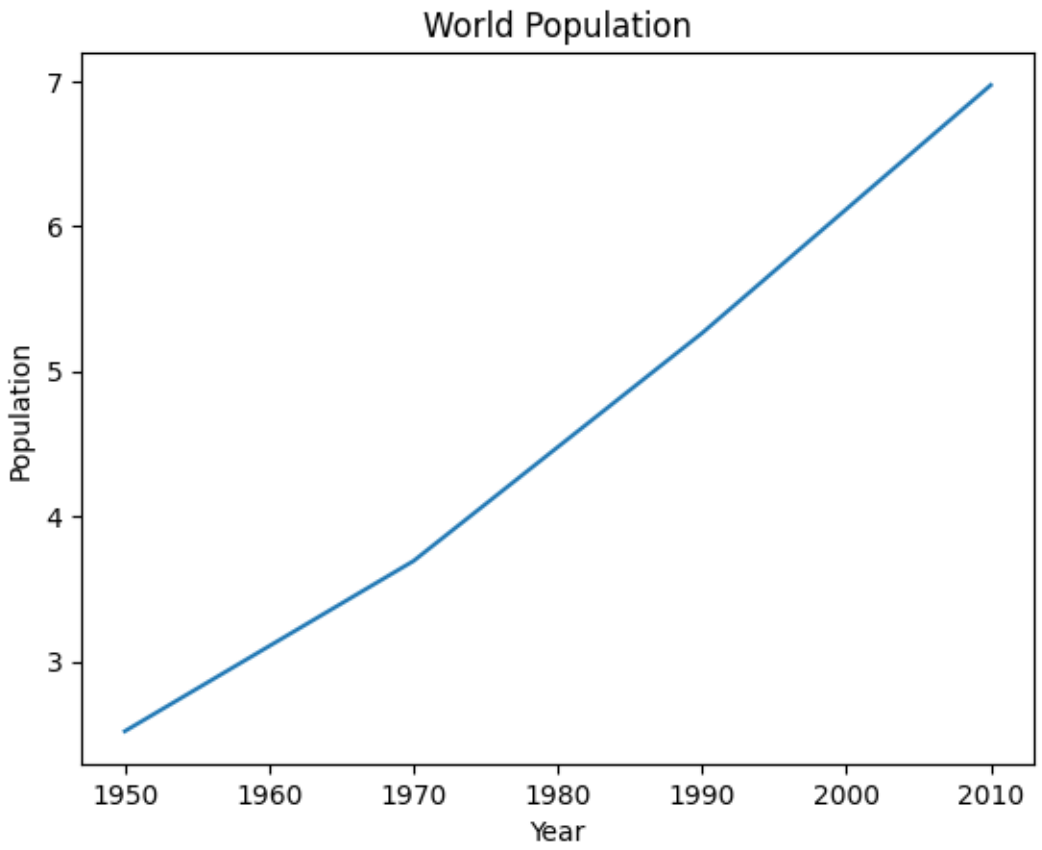
```
bySex = titanic.groupby("sex")
#here I am specifically asking for the mean
bySex["survived"].mean()
```

## 0.5 Plotting

We will not spend much time with basic plots in matplotlib but instead move quickly toward the pandas versions of these functions.

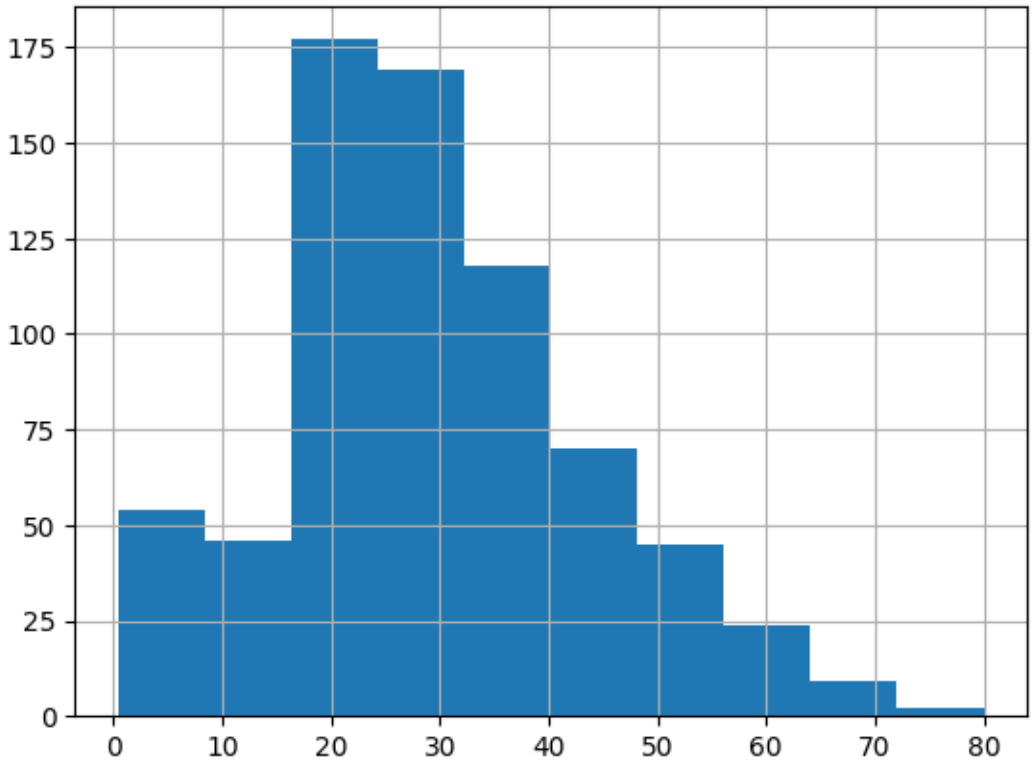
```
#%matplotlib inline
import matplotlib.pyplot as plt

#plt.rcParams['figure.dpi'] = 800
year = [1950, 1970, 1990, 2010]
pop = [2.519, 3.692, 5.263, 6.972]
plt.plot(year, pop)
#plt.bar(year, pop)
#plt.scatter(year, pop)
plt.xlabel('Year')
plt.ylabel('Population')
plt.title('World Population')
x = 1
#plt.show()
```




pandas offers plots directly from its objects

```
titanic.age.hist()
plt.show()
```

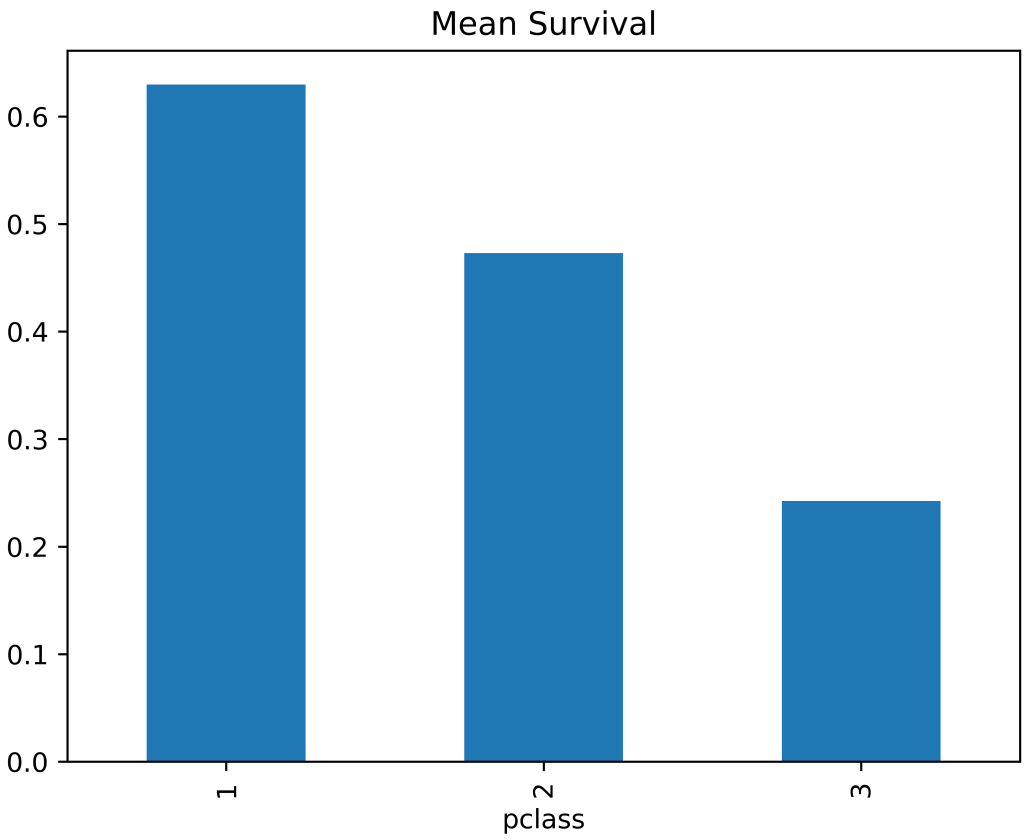



And often the axis labels are taken care of

```
#titanic.groupby("pclass").survived.mean().plot.bar()
SurvByPclass = titanic.groupby("pclass").survived.mean()

SurvByPclass.plot(kind="bar", title = "Mean Survival")
```

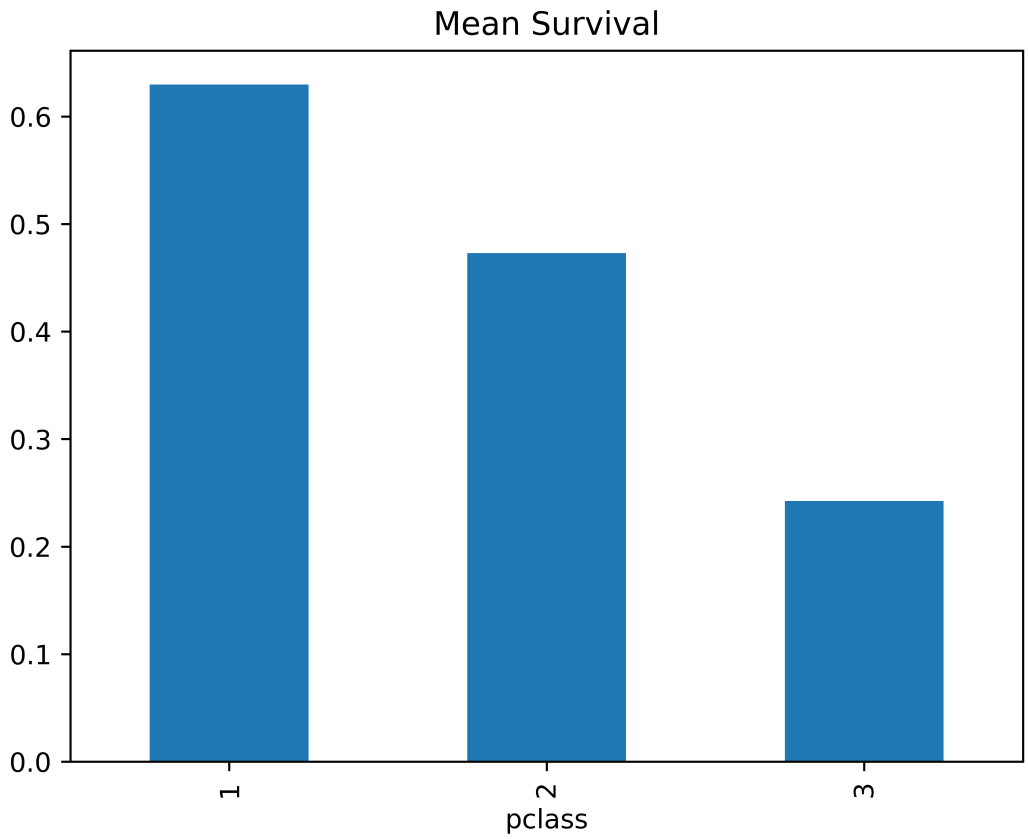
```
<Axes: title={'center': 'Mean Survival'}, xlabel='pclass'>
```




But you can customize each plot as you wish:

```
SurvByPclass.plot(kind="bar", x = "Passenger Class", y = "Survived", title = "Mean Surviva
```

```
<Axes: title={'center': 'Mean Survival'}, xlabel='pclass'>
```



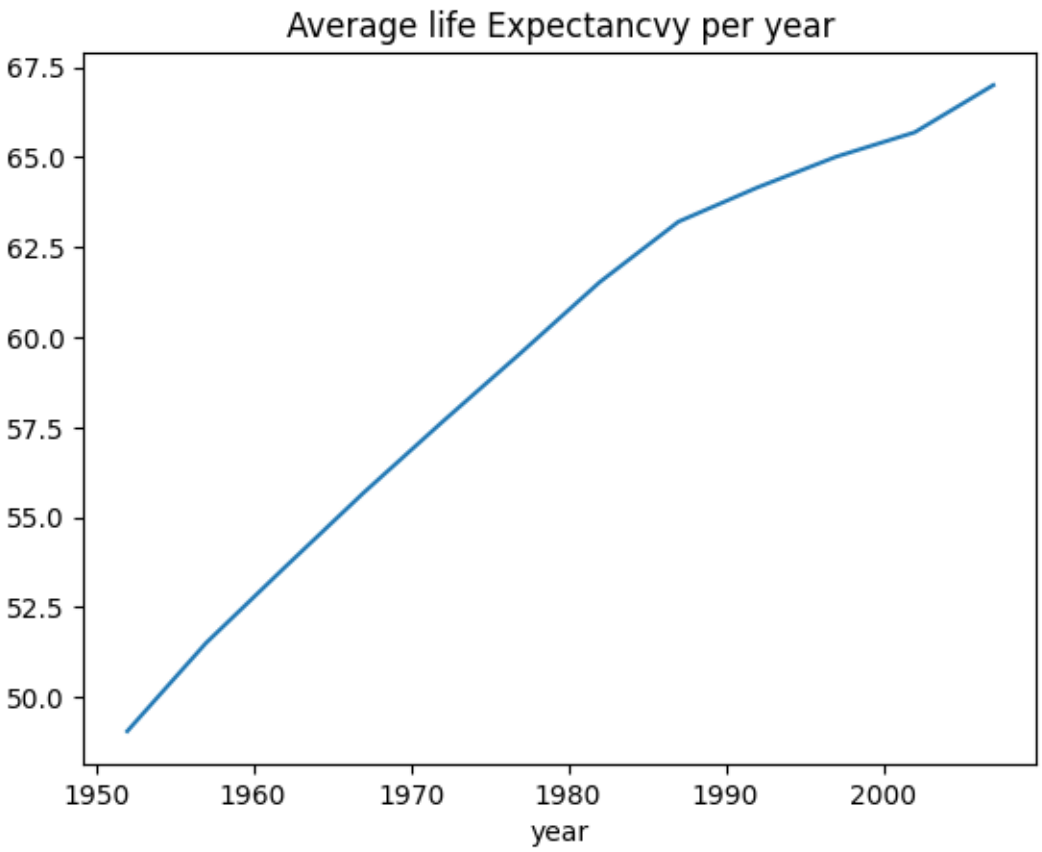

Tasks:

- Compute the avg. life expectancy in the gapminder data for each year
- Plot this as a line plot and give meaningful x and y labels and a title

```
lifeExpbyYear = gapminder.groupby("year")["lifeExp"].mean()

lifeExpbyYear.plot(y= "avg. life Exp", title = "Average life Expectancvy per year")
```

```
<Axes: title={'center': 'Average life Expectancvy per year'}, xlabel='year'>
```




---

## 0.6 Advanced topics

### 0.6.1 Creating Dataframes

1. Zip
2. From list of dicts

### 0.6.2 Indexing:

1. multilevel indexes
2. sorting
3. asking for ranges

## 0.7 Types of columns

1. categorical
2. dates

```
# Creating Dataframes
#using zip
# List1
Name = ['tom', 'krish', 'nick', 'juli']

# List2
Age = [25, 30, 26, 22]

# get the list of tuples from two lists.
# and merge them by using zip().
list_of_tuples = list(zip(Name, Age))
list_of_tuples = zip(Name, Age)
# Assign data to tuples.
#print(list_of_tuples)


# Converting lists of tuples into
# pandas Dataframe.
df = pd.DataFrame(list_of_tuples,
                  columns=['Name', 'Age'])

# Print data.
df
```

| | Name | Age |
|---|---|---|
| 0 | tom | 25 |
| 1 | krish | 30 |
| 2 | nick | 26 |
| 3 | juli | 22 |

```
#from list of dicts
data = [{'a': 1, 'b': 2, 'c': 3},
        {'a': 10, 'b': 20, 'c': 30}]

# Creates DataFrame.
```

```
df = pd.DataFrame(data)

df
```

| | a | b | c |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 10 | 20 | 30 |

```
# Indexing:

advLesson = True
if advLesson:
    frame2 = frame.set_index(["year", "state"])
    print(frame2)
    frame3 = frame2.sort_index()
    print(frame3)
    print(frame.loc[:,"state":"year"])
```

```
            pop
year state
2000 Ohio    1.5
2001 Ohio    1.7
2002 Ohio    3.6
2001 Nevada  2.4
2002 Nevada  2.9
2003 Nevada  3.2
            pop
year state
2000 Ohio    1.5
2001 Nevada  2.4
     Ohio    1.7
2002 Nevada  2.9
     Ohio    3.6
2003 Nevada  3.2
    state  year
0    Ohio  2000
1    Ohio  2001
2    Ohio  2002
3  Nevada  2001
4  Nevada  2002
```

```
5  Nevada  2003
```

### 0.7.1 Inplace

Note that I reassigned the objects in the code above. That is because most operations, such as `set_index`, `sort_index`, `drop`, etc. do not operate **inplace** unless specified!

# 1 Lecture 6

In this lecture we will continue our journey of Data Manipulation with pandas after reviewing some fundamental aspects of the syntax

1. Review
    - Review of "brackets"
2. Pandas
    - Dealing with Missing Values
    - Dealing with Duplicates
3. Plot of the Day
    - boxplot

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

### 1.0.1 Review of "brackets"

In Python, there are several types of brackets used for different purposes. Here's a brief review of the most commonly used brackets:

1. Parentheses `( )`: Parentheses are used for grouping expressions, defining function parameters, and invoking functions. They are also used in mathematical expressions to indicate order of operations.
2. Square brackets `[ ]`: Square brackets are primarily used for indexing and slicing operations on lists, tuples, and strings. They allow you to access individual elements or extract subsequences from these data types.
3. Curly brackets or braces `{ }`: Curly brackets are used to define dictionaries, which are key-value pairs. Dictionaries store data in an unordered manner, and you can access or manipulate values by referencing their corresponding keys within the curly brackets.

It's important to note that the usage of these brackets may vary depending on the specific context or programming paradigm you're working with. Nonetheless, understanding their general purpose will help you navigate Python code effectively.

#### 1.0.1.0.1 Examples

Here are a few examples of how each type of bracket is used in Python:

1. Parentheses `( )`:
   - Grouping expressions:

```
result = (2 + 3) * 4
# Output: 20
```

   - Defining function parameters:

```
def greet(name):
    print("Hello, " + name + "!")

greet("Alice")
# Output: Hello, Alice!
```

   - Invoking functions:

```
result = max(5, 10)
# Output: 10
```

2. Square brackets `[ ]`:
   - Indexing and slicing:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0])
# Output: 1
print(my_list[1:3])
# Output: [2, 3]
```

   - Modifying list elements:

```
my_list = [1, 2, 3]
my_list[1] = 10
print(my_list)
```

```
# Output: [1, 10, 3]
```

3. Curly brackets or braces `{ }`:
   - Defining dictionaries:

```
my_dict = {"name": "Alice", "age": 25, "city": "London"}
print(my_dict["name"])
# Output: Alice
```

   - Modifying dictionary values:

```
my_dict = {"name": "Alice", "age": 25}
my_dict["age"] = 30
print(my_dict)
# Output: {'name': 'Alice', 'age': 30}
```

Remember that the usage of brackets can vary depending on the specific programming context, but these examples provide a general understanding of their usage in Python.

### 1.0.2 Tasks

### 1.0.3 Data Manipulation with pandas

While we have seen panda's ability to (i) mix data types (strings, numbers, categories, Boolean, …) and (ii) refer to columns and rows by names, this library offers a lot more powerful tools for efficiently gaining insights from data, e.g.

- deal with missing values

### 1.0.4 Missing Values

Missing data occurs commonly in many data analysis applications. Most often they are a consequence of

- data entry errors, or
- unknown numbers, or
- `groupby` operations, or
- wrong mathematical operations (`1/0`, $\sqrt{-1}$,$\log(0)$, …)
- "not applicable" questions, or
- ….

For data with float type, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data.

Pandas refers to missing data as `NA`, which stands for *not available*.

The built-in Python `None` value is also treated as NA:

Recall constructing a DataFrame from a dictionary of equal-length lists or NumPy arrays (Lecture 4). What if there were data entry errors or just unknown numbers

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, None, 2001, 2002, 2003],
        "gdp": [1.5, 1.7, 3.6, 2.4, np.nan, 3.2]}
frame = pd.DataFrame(data)# creates a dataframe out of the data given!
df = frame
frame
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.5 |
| 1 | Ohio | 2001.0 | 1.7 |
| 2 | Ohio | NaN | 3.6 |
| 3 | Nevada | 2001.0 | 2.4 |
| 4 | Nevada | 2002.0 | NaN |
| 5 | Nevada | 2003.0 | 3.2 |

```
x = np.array([1,2,3,4])
x==2
#remove those values of x that are equal to 2
#x[[0,2,3]]
x[x!=2]
```

```
array([1, 3, 4])
```

```
frame.columns[frame.isna().any()]
```

```
Index(['year', 'gdp'], dtype='object')
```

(Note the annoying conversion of integers to float, a solution discussed here)

#### 1.0.4.1 Filtering Out Missing Data

There are a few ways to filter out missing data. While you always have the option to do it by hand using `pandas.isna` and Boolean indexing, `dropna` can be helpful.

With DataFrame objects, there are different ways to remove missing data. You may want to drop rows or columns that are all `NA`, or only those rows or columns containing any `NA`s at all. `dropna` by default drops any row containing a missing value:

```
frame.dropna()
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.5 |
| 1 | Ohio | 2001.0 | 1.7 |
| 3 | Nevada | 2001.0 | 2.4 |
| 5 | Nevada | 2003.0 | 3.2 |

Passing `how="all"` will drop only rows that are all NA:

```
frame.dropna(how="all")
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.5 |
| 1 | Ohio | 2001.0 | 1.7 |
| 2 | Ohio | NaN | 3.6 |
| 3 | Nevada | 2001.0 | 2.4 |
| 4 | Nevada | 2002.0 | NaN |
| 5 | Nevada | 2003.0 | 3.2 |

To drop columns in the same way, pass `axis="columns"`:

```
frame.dropna(axis="columns")
```

| | state |
|---|---|
| 0 | Ohio |
| 1 | Ohio |
| 2 | Ohio |
| 3 | Nevada |

| | state |
|---|---|
| 4 | Nevada |
| 5 | Nevada |

Suppose you want to keep only rows containing at most a certain number of missing observations. You can indicate this with the `thresh` argument:

```
frame.iloc[2,2] = np.nan
frame
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.5 |
| 1 | Ohio | 2001.0 | 1.7 |
| 2 | Ohio | NaN | NaN |
| 3 | Nevada | 2001.0 | 2.4 |
| 4 | Nevada | 2002.0 | NaN |
| 5 | Nevada | 2003.0 | 3.2 |

```
frame.dropna(thresh=2)
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.5 |
| 1 | Ohio | 2001.0 | 1.7 |
| 3 | Nevada | 2001.0 | 2.4 |
| 4 | Nevada | 2002.0 | NaN |
| 5 | Nevada | 2003.0 | 3.2 |

#### 1.0.4.2 Filling In Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways. For most purposes, the fillna method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:

```
frame2 = frame
frame2.iloc[0,0] = np.nan#None
frame2.iloc[0,2] = np.nan
```

```
frame2
#frame2.fillna(0)
#type(frame2["state"])
```

| | state | year | gdp |
|---|---|---|---|
| 0 | NaN | 2000.0 | NaN |
| 1 | Ohio | 2001.0 | 1.70 |
| 2 | Ohio | NaN | 3.60 |
| 3 | Nevada | 2001.0 | 2.40 |
| 4 | Nevada | 2002.0 | 2.48 |
| 5 | Nevada | 2003.0 | 3.20 |

Calling `fillna` with a dictionary, you can use a different fill value for each column:

```
frame2.fillna({"state" : "Neverland", "year": -999, "gdp": 0})
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Neverland | 2000.0 | 0.00 |
| 1 | Ohio | 2001.0 | 1.70 |
| 2 | Ohio | -999.0 | 3.60 |
| 3 | Nevada | 2001.0 | 2.40 |
| 4 | Nevada | 2002.0 | 2.48 |
| 5 | Nevada | 2003.0 | 3.20 |

With `fillna` you can do lots of other things such as simple data imputation using the median or mean statistics, at least for purely numeric data types:

```
frame['gdp'] = frame['gdp'].fillna(frame['gdp'].mean())
frame
```

| | state | year | gdp |
|---|---|---|---|
| 0 | NaN | 2000.0 | 1.50 |
| 1 | Ohio | 2001.0 | 1.70 |
| 2 | Ohio | NaN | 3.60 |
| 3 | Nevada | 2001.0 | 2.40 |
| 4 | Nevada | 2002.0 | 2.48 |
| 5 | Nevada | 2003.0 | 3.20 |

Different values for each column

```
df.fillna(df.mean())
```

```
FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future
  df.fillna(df.mean())
```

| | state | year | gdp |
|---|---|---|---|
| 0 | Ohio | 2000.0 | 1.50 |
| 1 | Ohio | 2001.0 | 1.70 |
| 2 | Ohio | 2001.4 | 3.60 |
| 3 | Nevada | 2001.0 | 2.40 |
| 4 | Nevada | 2002.0 | 2.48 |
| 5 | Nevada | 2003.0 | 3.20 |

```
fillna(frame['gdp'].mean())
```

### Duplicate Values

Duplicate rows may be found in a DataFrame for any number of reasons.

Removing duplicates in pandas can be useful in various scenarios, particularly when working with large datasets or performing data analysis. Here's a convincing example to illustrate its usefulness:

Let's say you have a dataset containing sales transactions from an online store. Each transaction record consists of multiple columns, including customer ID, product ID, purchase date, and purchase amount. Due to various reasons such as system glitches or human errors, duplicate entries might exist in the dataset, meaning that multiple identical transaction records are present. In such a scenario, removing duplicates becomes beneficial for several reasons:

1. Accurate Analysis: Duplicate entries can skew your analysis and lead to incorrect conclusions. By removing duplicates, you ensure that each transaction is represented only once, providing more accurate insights and preventing inflated or biased results.
2. Data Integrity: Duplicate entries consume unnecessary storage space and can make data management more challenging. By eliminating duplicates, you maintain data integrity and ensure a clean and organized dataset.
3. Efficiency: When dealing with large datasets, duplicate records can significantly impact computational efficiency. Removing duplicates allows you to streamline your data processing operations, leading to faster analysis and improved performance.

4. Unique Identifiers: Removing duplicates becomes crucial when working with columns that should contain unique values, such as customer IDs or product IDs. By eliminating duplicates, you ensure the integrity of these unique identifiers and prevent issues when performing joins or merging dataframes.
To remove duplicates in pandas, you can use the `drop_duplicates()` function. It identifies and removes duplicate rows based on specified columns or all columns in the dataframe, depending on your requirements.
Overall, removing duplicates in pandas is essential for maintaining data accuracy, integrity, and efficiency, allowing you to derive meaningful insights and make informed decisions based on reliable data.
::: {.cell execution-
Info='{"elapsed":315,"status":"ok","timestamp":1684161804084,"user":{"displayl
Loecher","userId":"02488037228155275753"},"user_tz":-
120}'
outputId='514bc45d-9578-4bbf-
a3ce-abf8cdf794b6'}
::: {.cell-output
.cell-output-display
execution_count=39}
"'{=html}

.dataframe tbody tr th {
vertical-align: top; }
.dataframe thead th { text-align:
right; }

```
.colab-df-convert {
background-color: #E8F0FE;
border: none; border-radius:
50%; cursor: pointer; display:
none; fill: #1967D2; height: 32px;
padding: 0 0 0 0; width: 32px; }
.colab-df-convert:hover {
background-color: #E2EBFA;
box-shadow: 0px 1px 2px
rgba(60, 64, 67, 0.3), 0px 1px 3px
1px rgba(60, 64, 67, 0.15); fill:
#174EA6; }
[theme=dark] .colab-df-convert {
background-color: #3B4455; fill:
#D2E3FC; }
[theme=dark]
.colab-df-convert:hover {
background-color: #434B5C;
box-shadow: 0px 1px 3px 1px
rgba(0, 0, 0, 0.15); filter:
drop-shadow(0px 1px 2px rgba(0,
0, 0, 0.3)); fill: #FFFFFF; }

async function
convertToInteractive(key) { const
element =
document.querySelector('#df-
0e9d0424-13d7-4cb4-900b-
a6a5a04aac1d'); const dataTable
= await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
```

const docLinkHtml = ‘Like what
you see? Visit the’ + ‘data table
notebook’ + ’ to learn more
about interactive tables.’;
element.innerHTML = ’‘;
dataTable[’output_type’] =
’display_data’; await
google.colab.output.renderOutput(dataTable,
element); const docLink =
document.createElement(‘div’);
docLink.innerHTML =
docLinkHtml;
element.appendChild(docLink); }
“‘
::: :::
The DataFrame method
`duplicated` returns a Boolean
Series indicating whether each
row is a duplicate
::: {.cell execution-
Info=‘{“elapsed”:241,“status”:“ok”,“timestamp”:1684161830353,“user”:{“displayl
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-
120}’
outputId=‘fe364e46-a0a2-4cfe-
ac5f-a259b2e3c9ac’}
::: {.cell-output
.cell-output-display
execution_count=40}
Relatedly, `drop_duplicates`
returns a DataFrame with rows
where the `duplicated` array is
`False` filtered out:
::: {.cell execution-
Info=‘{“elapsed”:256,“status”:“ok”,“timestamp”:1684161865458,“user”:{“displayl
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-
120}’
outputId=‘577d9c69-8326-4ec3-
8f79-f09ea5ea217c’}
::: {.cell-output
.cell-output-display
execution_count=41}
“‘{=html}

```
.dataframe tbody tr th {
vertical-align: top; }
.dataframe thead th { text-align:
right; }


.colab-df-convert {
background-color: #E8F0FE;
border: none; border-radius:
50%; cursor: pointer; display:
none; fill: #1967D2; height: 32px;
padding: 0 0 0 0; width: 32px; }
.colab-df-convert:hover {
background-color: #E2EBFA;
box-shadow: 0px 1px 2px
rgba(60, 64, 67, 0.3), 0px 1px 3px
1px rgba(60, 64, 67, 0.15); fill:
#174EA6; }
[theme=dark] .colab-df-convert {
background-color: #3B4455; fill:
#D2E3FC; }
[theme=dark]
.colab-df-convert:hover {
background-color: #434B5C;
box-shadow: 0px 1px 3px 1px
rgba(0, 0, 0, 0.15); filter:
drop-shadow(0px 1px 2px rgba(0,
0, 0, 0.3)); fill: #FFFFFF; }

async function
convertToInteractive(key) { const
element =
document.querySelector('#df-
38894c14-a772-4d62-bc7a-
a066ec8c634e'); const dataTable
= await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
```

const docLinkHtml = ‘Like what
you see? Visit the’ + ‘data table
notebook’ + ’ to learn more
about interactive tables.’;
element.innerHTML = ’‘;
dataTable[’output_type’] =
’display_data’; await
google.colab.output.renderOutput(dataTable,
element); const docLink =
document.createElement(‘div’);
docLink.innerHTML =
docLinkHtml;
element.appendChild(docLink); }
“‘
::: :::
Both methods by default consider
all of the columns; alternatively,
you can specify any `subset` of
them to detect duplicates.
::: {.cell execution-
Info=‘{“elapsed”:226,“status”:“ok”,“timestamp”:1684161928182,“user”:{“displayl
Loecher”,“userId”:“02488037228155275753”},“user_tz”:-
120}’
outputId=‘52bcb8af-5ea3-4443-
de41-8de758ce65fe’}
::: {.cell-output
.cell-output-display
execution_count=42}
“‘{=html}

.dataframe tbody tr th {
vertical-align: top; }
.dataframe thead th { text-align:
right; }


.colab-df-convert {
background-color: #E8F0FE;
border: none; border-radius:
50%; cursor: pointer; display:
none; fill: #1967D2; height: 32px;
padding: 0 0 0 0; width: 32px; }

```
.colab-df-convert:hover {
background-color: #E2EBFA;
box-shadow: 0px 1px 2px
rgba(60, 64, 67, 0.3), 0px 1px 3px
1px rgba(60, 64, 67, 0.15); fill:
#174EA6; }
[theme=dark] .colab-df-convert {
background-color: #3B4455; fill:
#D2E3FC; }
[theme=dark]
.colab-df-convert:hover {
background-color: #434B5C;
box-shadow: 0px 1px 3px 1px
rgba(0, 0, 0, 0.15); filter:
drop-shadow(0px 1px 2px rgba(0,
0, 0, 0.3)); fill: #FFFFFF; }

async function
convertToInteractive(key) { const
element =
document.querySelector('#df-
3ced5523-f090-4537-9982-
64ac22640da3'); const dataTable
= await
google.colab.kernel.invokeFunction('convertToInteractive',
[key], {}); if (!dataTable) return;
const docLinkHtml = 'Like what
you see? Visit the' + 'data table
notebook' + ' to learn more
about interactive tables.';
element.innerHTML = '';
dataTable['output_type'] =
'display_data'; await
google.colab.output.renderOutput(dataTable,
element); const docLink =
document.createElement('div');
docLink.innerHTML =
docLinkHtml;
element.appendChild(docLink); }
```
::: :::

#### 1.0.4.2.1 Titanic data

```
# Since pandas does not have any built in data, I am going to "cheat" and
# make use of the `seaborn` library
import seaborn as sns

titanic = sns. load_dataset('titanic')
titanic["3rdClass"] = titanic["pclass"]==3
titanic["male"] = titanic["sex"]=="male"

titanic
```

| | survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | man | True |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | woman | False |
| 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | woman | False |
| 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | woman | False |
| 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | man | True |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 886 | 0 | 2 | male | 27.0 | 0 | 0 | 13.0000 | S | Second | man | True |
| 887 | 1 | 1 | female | 19.0 | 0 | 0 | 30.0000 | S | First | woman | False |
| 888 | 0 | 3 | female | NaN | 1 | 2 | 23.4500 | S | Third | woman | False |
| 889 | 1 | 1 | male | 26.0 | 0 | 0 | 30.0000 | C | First | man | True |
| 890 | 0 | 3 | male | 32.0 | 0 | 0 | 7.7500 | Q | Third | man | True |

```
#how many missing values in age ?
np.sum(titanic["age"].isna())
```

```
177
```

```
titanic.describe()
```

| | survived | pclass | age | sibsp | parch | fare |
|---|---|---|---|---|---|---|
| count | 891.000000 | 891.000000 | 714.000000 | 891.000000 | 891.000000 | 891.000000 |
| mean | 0.383838 | 2.308642 | 29.699118 | 0.523008 | 0.381594 | 32.204208 |
| std | 0.486592 | 0.836071 | 14.526497 | 1.102743 | 0.806057 | 49.693429 |
| min | 0.000000 | 1.000000 | 0.420000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.000000 | 2.000000 | 20.125000 | 0.000000 | 0.000000 | 7.910400 |

| | survived | pclass | age | sibsp | parch | fare |
|---|---|---|---|---|---|---|
| 50% | 0.000000 | 3.000000 | 28.000000 | 0.000000 | 0.000000 | 14.454200 |
| 75% | 1.000000 | 3.000000 | 38.000000 | 1.000000 | 0.000000 | 31.000000 |
| max | 1.000000 | 3.000000 | 80.000000 | 8.000000 | 6.000000 | 512.329200 |

```
np.mean(titanic["age"])

np.sum(titanic["age"])/714

titanic['age'] = titanic['age'].fillna(titanic['age'].mean())
```

```
29.69911764705882
```

Notice that the age columns contains missing values, which is a big topic by itself in data science.

How should an aggregating react to and handle missing values? The default often is to ignore and exclude them from the computation, e.g.

```
#the following shoudl be equal but is not due to missing values
meanAge =np.mean(titanic.age)
print(meanAge)
print(np.sum(titanic.age)/len(titanic.age))
```

```
29.69911764705882
23.79929292929293
```

In general, it is a good idea to diagnose how many missing values there are in each column. We can use some handy built-in support for this task:

```
titanic.isna().sum()
```

```
survived      0
pclass        0
sex           0
age         177
sibsp         0
parch         0
fare          0
```

```
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
3rdClass         0
male             0
dtype: int64
```

### 1.0.5 Tasks

Dropping or replacing NAs:

---

## 1.1 Plotting

The "plot type of the day" is one of the most popular ones used to display data distributions, the **boxplot**.

Boxplots, also known as **box-and-whisker plots**, are a statistical visualization tool that provides a concise summary of a dataset's distribution. They display key descriptive statistics and provide insights into the central tendency, variability, and skewness of the data. Here's a brief introduction and motivation for using boxplots:

1. Structure of Boxplots: Boxplots consist of a box and whiskers that represent different statistical measures of the data:
    - The box represents the interquartile range (IQR), which spans from the lower quartile (25th percentile) to the upper quartile (75th percentile). The width of the box indicates the spread of the middle 50% of the data.
    - A line (whisker) extends from each end of the box to show the minimum and maximum values within a certain range (often defined as 1.5 times the IQR).
    - Points beyond the whiskers are considered outliers and plotted individually.
2. Motivation for Using Boxplots: Boxplots offer several benefits and are commonly used for the following reasons:

- Visualizing Data Distribution: Boxplots provide a concise overview of the distribution of a dataset. They show the skewness, symmetry, and presence of outliers, allowing for quick identification of key features.
- Comparing Groups: Boxplots enable easy visual comparison of multiple groups or categories. By placing side-by-side boxplots, you can assess differences in central tendency and variability between groups.
- Outlier Detection: Boxplots explicitly mark outliers, aiding in the identification of extreme values or data points that deviate significantly from the overall pattern.
- Data Summary: Boxplots summarize key statistics, including the median, quartiles, and range, providing a quick understanding of the dataset without the need for detailed calculations.
- Robustness: Boxplots are relatively robust to skewed or asymmetric data and can effectively handle datasets with outliers.

Boxplots are widely used in various fields, including data analysis, exploratory data visualization, and statistical reporting. They offer a clear and concise representation of data distribution, making them a valuable tool for understanding and communicating the characteristics of a dataset.

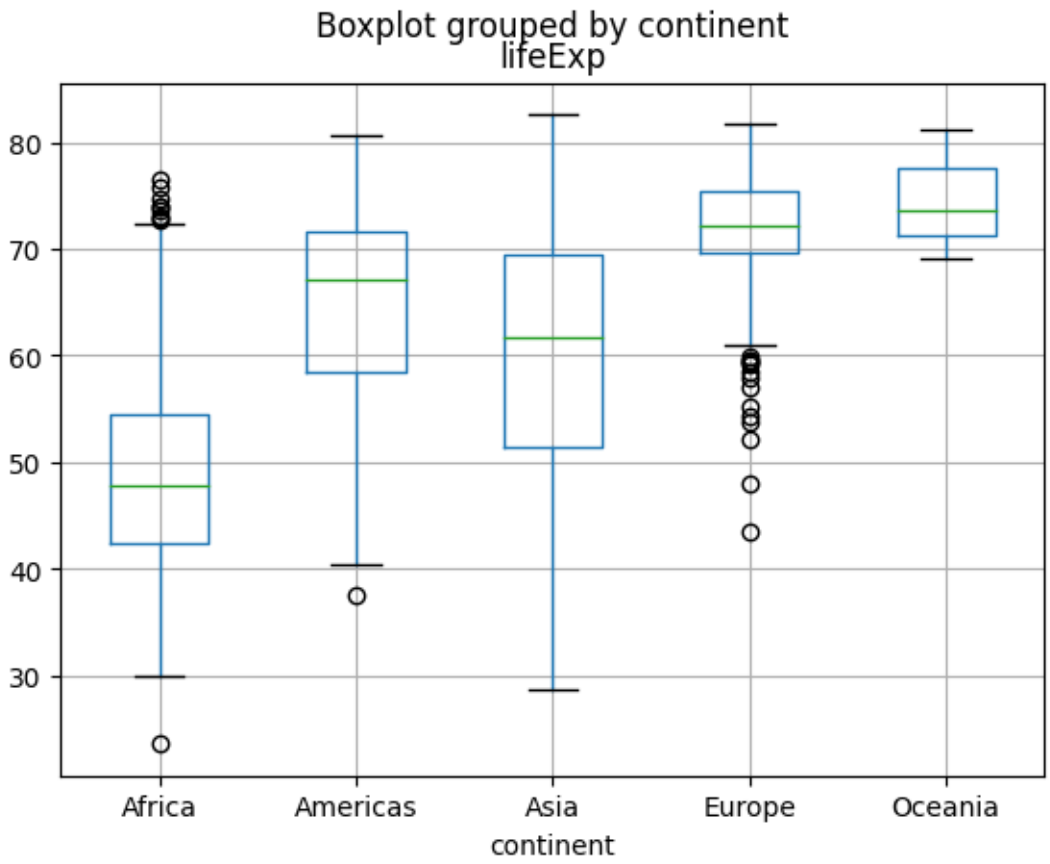
```
!pip install gapminder
from gapminder import gapminder
```

```
gapminder
```

| | country | continent | year | lifeExp | pop | gdpPercap |
|---|---|---|---|---|---|---|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 1 | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| 2 | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| 3 | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| 4 | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |
| ... | ... | ... | ... | ... | ... | ... |
| 1699 | Zimbabwe | Africa | 1987 | 62.351 | 9216418 | 706.157306 |
| 1700 | Zimbabwe | Africa | 1992 | 60.377 | 10704340 | 693.420786 |
| 1701 | Zimbabwe | Africa | 1997 | 46.809 | 11404948 | 792.449960 |
| 1702 | Zimbabwe | Africa | 2002 | 39.989 | 11926563 | 672.038623 |
| 1703 | Zimbabwe | Africa | 2007 | 43.487 | 12311143 | 469.709298 |

The pandas way

```
gapminder.boxplot(column = "lifeExp", by="continent");
```



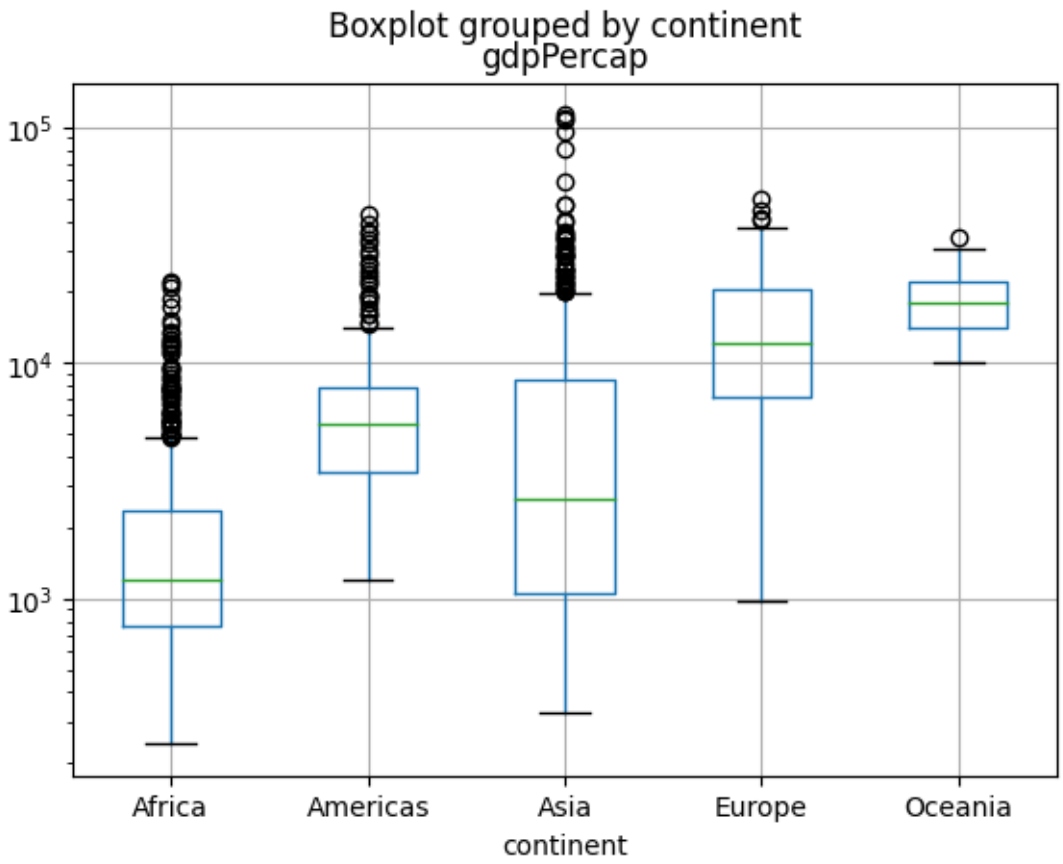

The matplotlib way

```
plt.boxplot(gapminder["continent"], gapminder["lifeExp"]);
```

### 1.1.1 Task

- Create a boxplot for `gdpPercap` instead. What do you notice ? Are you happy with how the plot looks? Any "trick" you can think to make this more readable?
- Advanced: can you create boxplots for `gdpPerCap` and `lifeExp` in one command?

```
gapminder.boxplot(column = "gdpPercap", by="continent");
plt.yscale("log")
```




---

Further Reading:

- Python Plotting With Matplotlib Tutorial.)

```
import numpy as np
from scipy.stats import entropy

p = np.array([1/100, 99/100])
n=2
#p = np.array(np.ones)/n
H = entropy(p, base=2)
H
```

```
0.08079313589591118
```

# 2 Lecture 7

In this lecture we will learn about **modeling** data for the first time. After this lesson, you should know what we generally mean by a "model", what linear regression is and how to interpret the output. But first we need to introduce a new data type: *categorical variables*.

1. Categorical variables
2. Models
    - Tables as models
    - Modeling Missing Values
    - Linear Regression

Online Resources:

Chapter 7.5 of our textbook introduces categorical variables.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from numpy.random import default_rng
```

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
!pip install gapminder
from gapminder import gapminder
```

```
gapminder.head()
```

| | country | continent | year | lifeExp | pop | gdpPercap |
|---|---|---|---|---|---|---|
| 0 | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| 1 | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| 2 | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| 3 | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |

| | country | continent | year | lifeExp | pop | gdpPercap |
|---|---|---|---|---|---|---|
| 4 | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |

## 2.1 Categorical variables

As a motivation, take another look at the gapminder data which contains variables of a **mixed type**: numeric columns along with string type columns which contain repeated instances of a smaller set of distinct or **discrete** values which

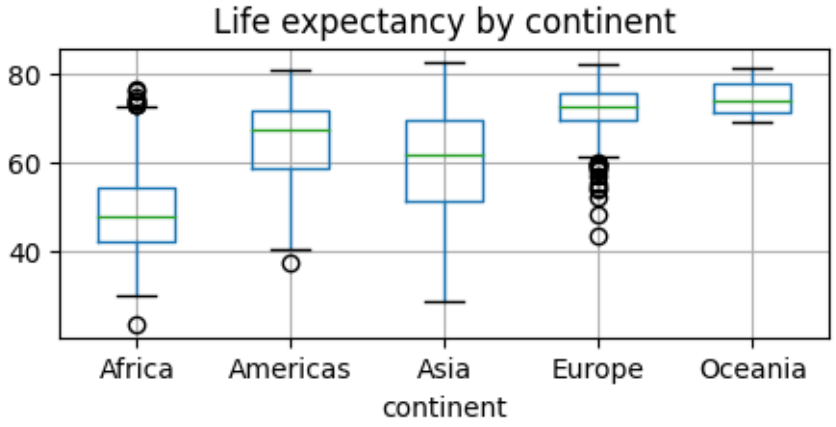
1. are not numeric (but could be represented as numbers)
2. cannot really be ordered
3. typically take on a finite set of values, or *categories*.

We refer to these data types as **categorical**.

We have already seen functions like `unique` and `value_counts`, which enable us to extract the distinct values from an array and compute their frequencies.

Boxplots and grouping operations typically use a categorical variable to compute summaries of a numerical variables for each category separately, e.g.

```
gapminder.boxplot(column = "lifeExp", by="continent",figsize=(5, 2));
plt.title('Life expectancy by continent')
# Remove the default suptitle
plt.suptitle("");
```



pandas has a special `Categorical` extension type for holding data that uses the integer-based categorical representation or encoding. This is a popular data compression technique for data with many occurrences of similar values and can provide significantly faster performance with lower memory use, especially for string data.

```
gapminder.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   object
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

```
gapminder['country'] = gapminder['country'].astype('category')
gapminder.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   category
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: category(1), float64(2), int64(2), object(1)
memory usage: 75.2+ KB
```

We will come back to the usefulness of this later.

## 2.2 Tables as models

For now let us look at our first "model":

```
titanic.head()
```

| | survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male | dec |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | man | True | Na |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | woman | False | C |
| 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | woman | False | Na |
| 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | woman | False | C |
| 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | man | True | Na |

```
titanic = sns.load_dataset('titanic')
titanic["3rdClass"] = (titanic["pclass"]==3)
titanic["male"] = (titanic["sex"]=="male")
vals1, cts1 = np.unique(titanic["3rdClass"], return_counts=True)
print(cts1)
print(vals1)
```

```
[400 491]
[False  True]
```

```
print("The mean survival on the Titanic was", np.mean(titanic.survived))
```

```
The mean survival on the Titanic was 0.3838383838383838
```

```
ConTbl = pd.crosstab(titanic["sex"], titanic["survived"])
ConTbl
```

| survived | 0 | 1 |
|---|---|---|
| sex | | |
| female | 81 | 233 |
| male | 468 | 109 |

What are the estimated survival probabilities?

```
#the good old groupby way:
bySex = titanic.groupby("sex").survived
bySex.mean()
```

```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

```
p3D = pd.crosstab([titanic["sex"], titanic["3rdClass"]], titanic["survived"])
p3D
```

| sex | survived<br>3rdClass | 0 | 1 |
|---|---|---|---|
| female | False | 9 | 161 |
| | True | 72 | 72 |
| male | False | 168 | 62 |
| | True | 300 | 47 |

What are the estimated survival probabilities?

```
#the good old groupby way:
bySex = titanic.groupby(["sex", "3rdClass"]).survived
bySex.mean()
```

```
sex     3rdClass
female  False       0.947059
        True        0.500000
male    False       0.269565
        True        0.135447
Name: survived, dtype: float64
```

The above table can be looked at as a **model**, which is defined as a function which takes *inputs* $\mathbf{x}$ and "spits out" a *prediction*:

$y = f(\mathbf{x})$

In our case, the inputs are $x_1 = \text{sex}$, $x_2 = \text{3rdClass}$, and the output is the estimated survival probability!

It is evident that we could keep adding more *input* variables and make finer and finer grained predictions.

### 2.2.1 Modeling Missing Values

We have already seen how to detect and how to replace missing values. But the latter -until now- was rather crude: we often replaced all values with a "global" average.

Clearly, we can do better than replacing all missing entries in the *survived* column with the average 0.38.

```
rng = default_rng()

missingRows = rng.integers(0,890,20)
print(missingRows)
#introduce missing values
titanic.iloc[missingRows,0] = np.nan
```

```
[409 881 389 717 378 547 134 351 691 134 212  99  69 642 700 861 205   8
 559 864]
```

```
np.sum(titanic.survived.isna())
```

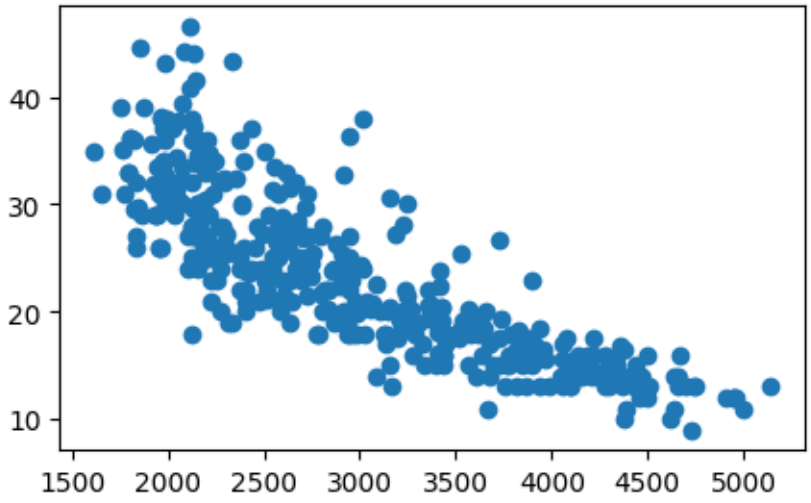
```
36
```

#### 2.2.1.1 From categorical to numerical relations

```
url = "https://drive.google.com/file/d/1UbZy5Ecknpl1GXZBkbhJ_K6GJcIA2Plq/view?usp=share_li
url='https://drive.google.com/uc?id=' + url.split('/')[-2]
auto = pd.read_csv(url)
auto.head()
```

| | mpg | cylinders | displacement | horsepower | weight | acceleration | year | origin | name |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 18.0 | 8 | 307.0 | 130 | 3504 | 12.0 | 70 | 1 | chevrolet chevelle mal |
| 1 | 15.0 | 8 | 350.0 | 165 | 3693 | 11.5 | 70 | 1 | buick skylark 320 |
| 2 | 18.0 | 8 | 318.0 | 150 | 3436 | 11.0 | 70 | 1 | plymouth satellite |
| 3 | 16.0 | 8 | 304.0 | 150 | 3433 | 12.0 | 70 | 1 | amc rebel sst |
| 4 | 17.0 | 8 | 302.0 | 140 | 3449 | 10.5 | 70 | 1 | ford torino |

```
plt.figure(figsize=(5,3))
plt.scatter(x=auto["weight"], y=auto["mpg"]);
```

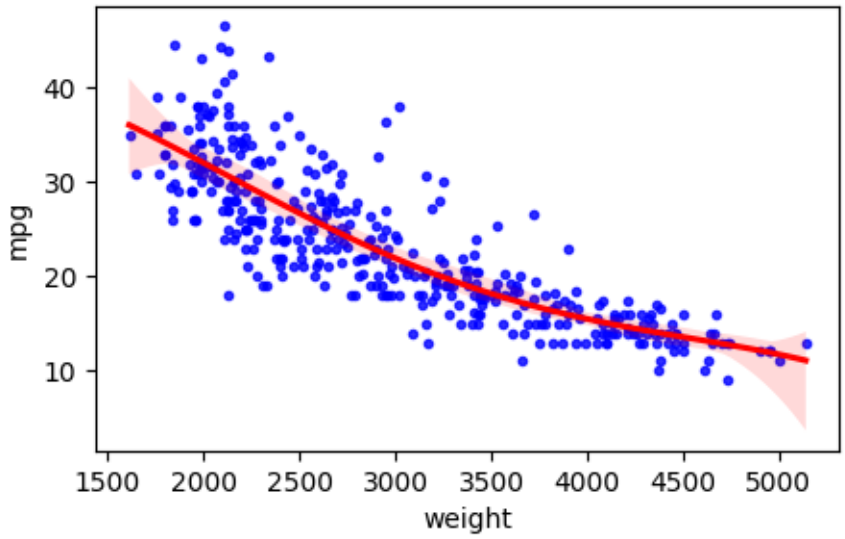



## 2.3 Linear Regression

We can roughly estimate, i.e. "model" this relationship with a straight line:

$$y = \beta_0 + \beta_1 x$$

```
plt.figure(figsize=(5,3))
tmp=sns.regplot(x=auto["weight"], y=auto["mpg"], order=4, ci=95,
                scatter_kws={'color':'b', 's':9}, line_kws={'color':'r'})
```




Remind yourself of the definition of the slope of a straight line

$$\beta_1 = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

```
est = smf.ols('mpg ~ weight', auto).fit()
est.summary().tables[1]
```

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 46.2165 | 0.799 | 57.867 | 0.000 | 44.646 | 47.787 |
| weight | -0.0076 | 0.000 | -29.645 | 0.000 | -0.008 | -0.007 |

```
np.corrcoef(auto["weight"], auto["mpg"])
```

```
array([[ 1.        , -0.83224421],
       [-0.83224421,  1.        ]])
```

```
np.corrcoef(auto[["weight","mpg", "horsepower"]])
```

```
array([[1.        , 0.99996983, 0.99998307, ..., 0.99996685, 0.99993843,
        0.99993167],
```

```
[0.99996983, 1.        , 0.9999981 , ..., 0.99987343, 0.99982207,
 0.9998107 ],
[0.99998307, 0.9999981 , 1.        , ..., 0.99990253, 0.99985692,
 0.99984671],
...,
[0.99996685, 0.99987343, 0.99990253, ..., 1.        , 0.99999564,
 0.99999371],
[0.99993843, 0.99982207, 0.99985692, ..., 0.99999564, 1.        ,
 0.99999982],
[0.99993167, 0.9998107 , 0.99984671, ..., 0.99999371, 0.99999982,
 1.        ]])
```

---

Further Reading:

- 