

<https://matplotlib.org/stable/tutorials/colors/colormaps.html>

## 1. Die Holdout-Methode

## 2. Indizierung in Pandas

## 1.1 Wiederholung: Matplotlib

### Grundlegende Plotbefehle:

- `plt.scatter / ax.scatter`: Punkte
- `plt.plot / ax.plot`: Linien

### Beschriftungen:

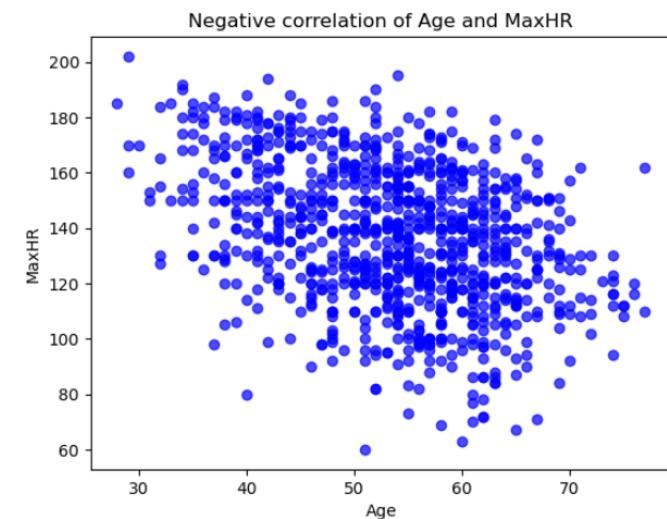
- `plt.xlabel / ax.set_xlabel` (ebenso für y)
- `plt.xticks / ax.set_xticks` (ebenso für y)
- `- / ax.set_xticklabels` (ebenso für y)
- `plt.title / ax.set_title`: Titel je Plot
- `plt.suptitle / -`: Titel über Subplots

### Skalierung und Formatierung:

- `plt.xlim / ax.set_xlim` (ebenso für y)
- `plt.tight_layout / -`: Subplot-Abstände

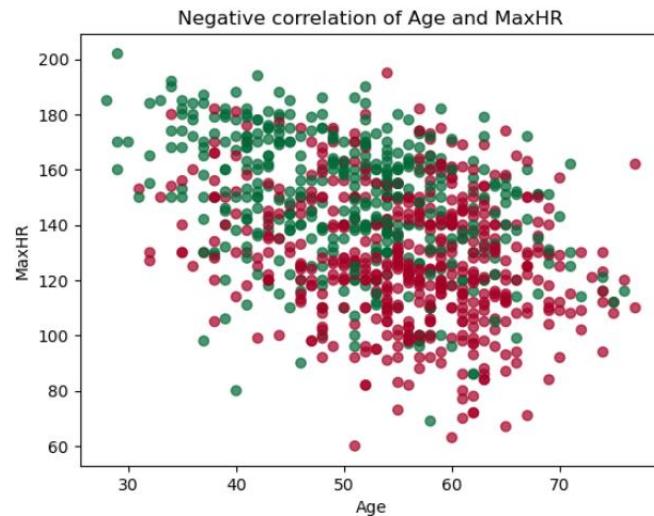
In [1]:

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 df = pd.read_csv("VL02_Material/heart.csv")
6
7 plt.scatter ( df["Age"],df["MaxHR"],
8               alpha=0.7, color="blue" )
9 plt.xlabel("Age")
10 plt.ylabel("MaxHR")
11 plt.title("Negative correlation of Age and MaxHR");
12
```

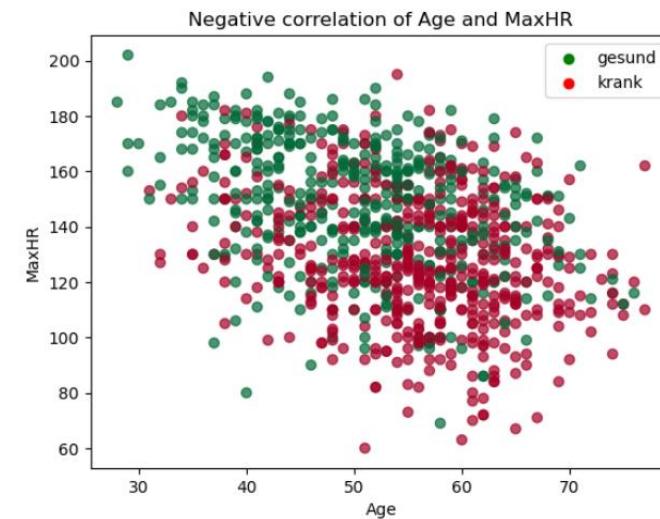


Integration der Zusatzinformation "HeartDisease" (Frage: Können Sie eine Legende hinzufügen?):

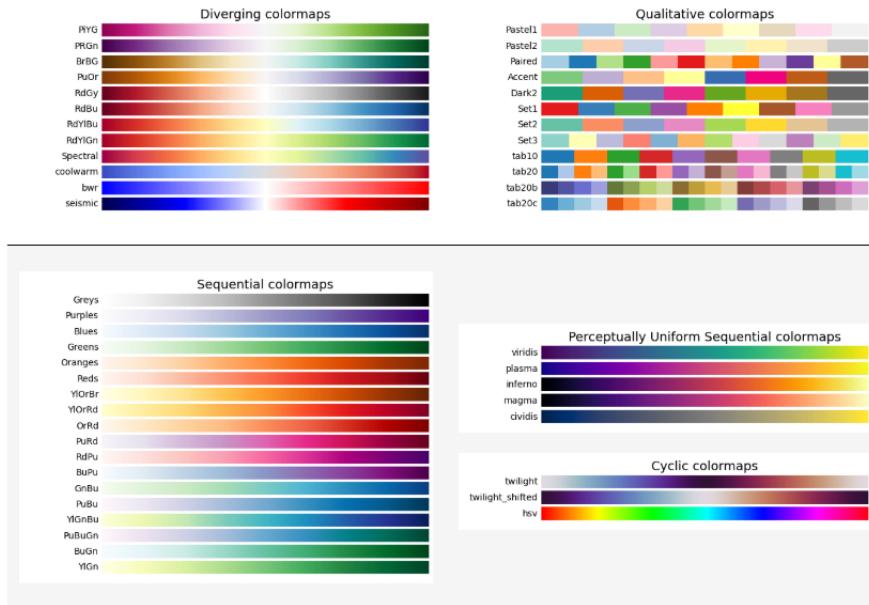
```
In [3]: 1 plt.scatter ( df["Age"],df["MaxHR"], alpha=0.7,  
2                 c=df["HeartDisease"], cmap="RdYlGn_r" )  
3 plt.xlabel("Age")  
4 plt.ylabel("MaxHR")  
5 plt.title("Negative correlation of Age and MaxHR");
```



```
In [9]: 1 plt.scatter ( df["Age"],df["MaxHR"], alpha=0.7,  
2                 c=df["HeartDisease"], cmap="RdYlGn_r" )  
3 plt.xlabel("Age")  
4 plt.ylabel("MaxHR")  
5 plt.title("Negative correlation of Age and MaxHR")  
6  
7 # Legende fügen  
8 plt.scatter([],[],label="gesund", color="green")  
9 plt.scatter([],[],label="krank", color="red")  
10 plt.legend();
```



Der Parameter `cmap` spezifiziert eine ColorMap, z.B.:



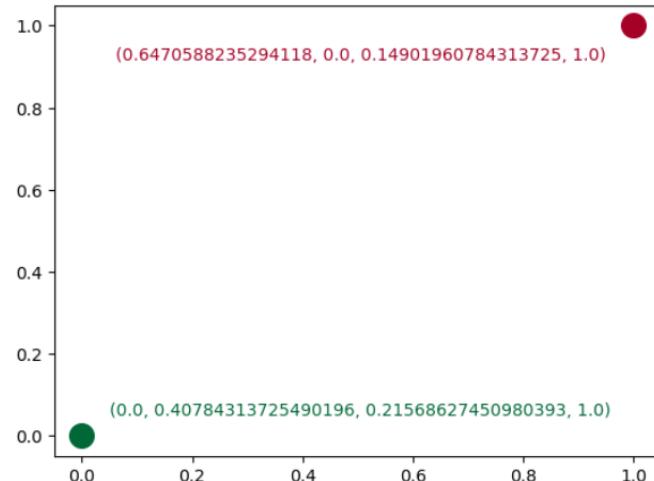
**For many applications, a perceptually uniform colormap is the best choice; i.e. a colormap in which equal steps in data are perceived as equal steps in the color space.**

- Sequential: (...) should be used for representing information that has ordering.
- Diverging: (...) should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
- Cyclic: (...) should be used for values that wrap around at the endpoints, such as phase angle, wind direction, or time of day.
- Qualitative: (...) should be used to represent information which does not have ordering or relationships.

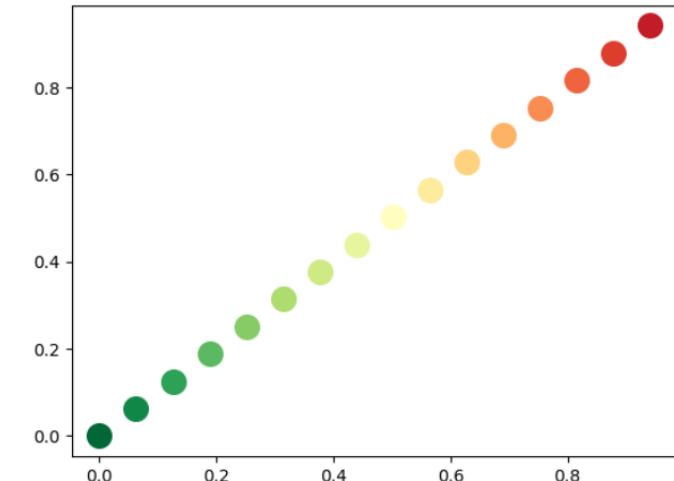
<https://matplotlib.org/stable/tutorials/colors/colormaps.html>

## Verwendung von Colormaps:

```
In [244]: 1 # Die Werte, die via c=... spezifiziert werden,  
2 # werden zu [0,1] normiert und als Input für  
3 # die cmap verwendet.  
4 cmap = plt.get_cmap("RdYlGn_r")  
5 plt.scatter ( [0],[0], color=cmap(0.0), s=200 )  
6 plt.scatter ( [1],[1], color=cmap(1.0), s=200 )  
7 plt.text(x=0.05,y=0.05,s=cmap(0.0), color=cmap(0.0))  
8 plt.text(x=0.95,y=0.95,s=cmap(1.0), color=cmap(1.0),  
9         ha="right", va="top");  
10
```

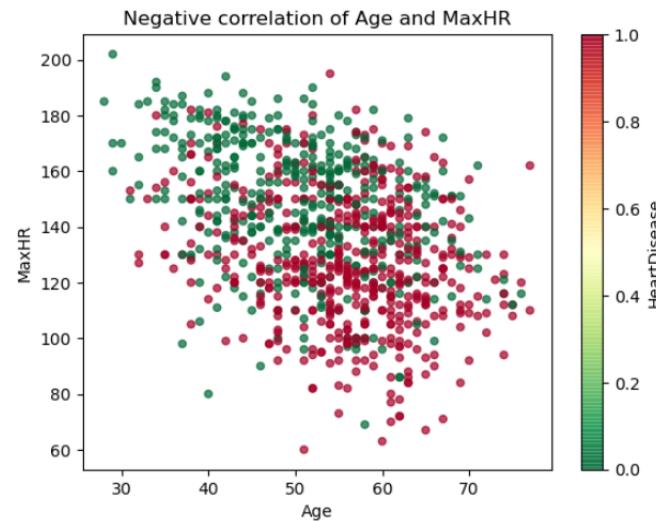


```
In [250]: 1 # Vorsicht: Integers werden anders interpretiert.  
2 # Es gilt hier cmap(255)==cmap(1.0)  
3 # bzw. cmap(1)==cmap(1/255)  
4 # unterschiedlich je cmap)  
5  
6 for i in range(0,256,16): # nur jeder 16. Punkt  
7     plt.scatter ( [i/255],[i/255],  
8                 color=cmap(i), s=200 )  
9  
10
```

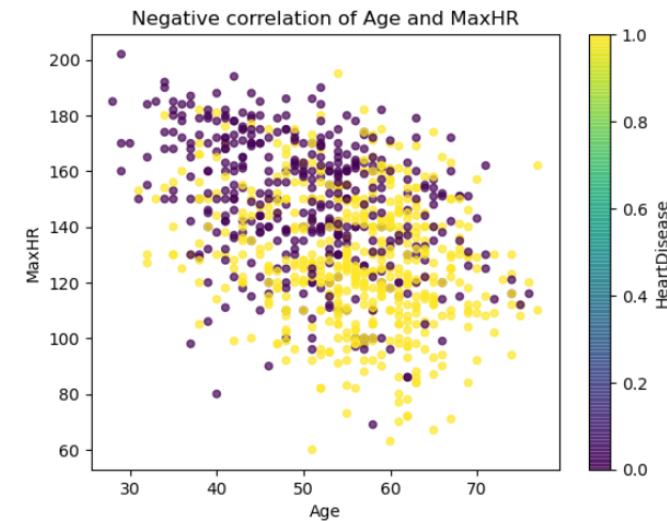


Mit dem Pandas-Wrapper in einer Zeile:

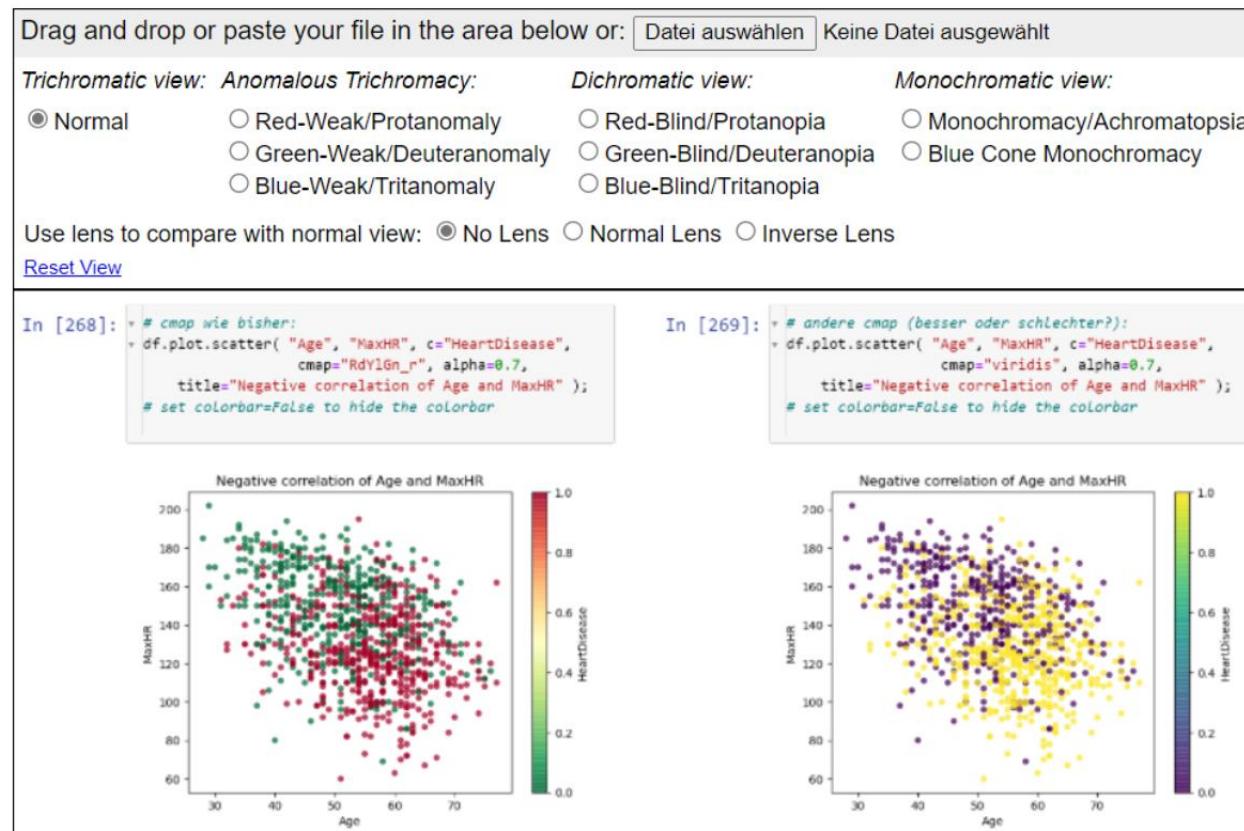
```
In [63]: 1 # cmap wie bisher:  
2 df.plot.scatter( "Age", "MaxHR", c="HeartDisease",  
3                  cmap="RdYlGn_r", alpha=0.7,  
4                  title="Negative correlation of Age and MaxHR" );  
5 # set colorbar=False to hide the colorbar  
6
```



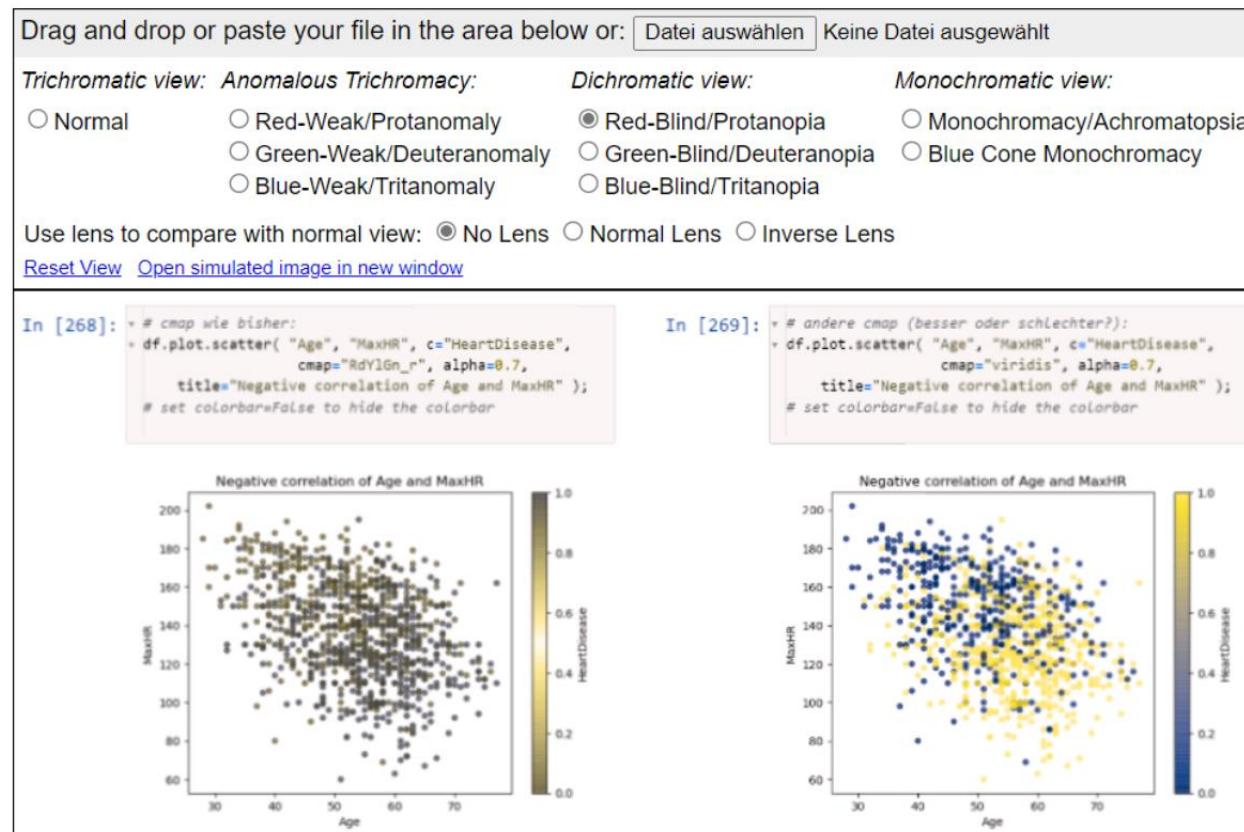
```
In [269]: 1 # andere cmap (besser oder schlechter?):  
2 df.plot.scatter( "Age", "MaxHR", c="HeartDisease",  
3                  cmap="viridis", alpha=0.7,  
4                  title="Negative correlation of Age and MaxHR" );  
5 # set colorbar=False to hide the colorbar  
6
```



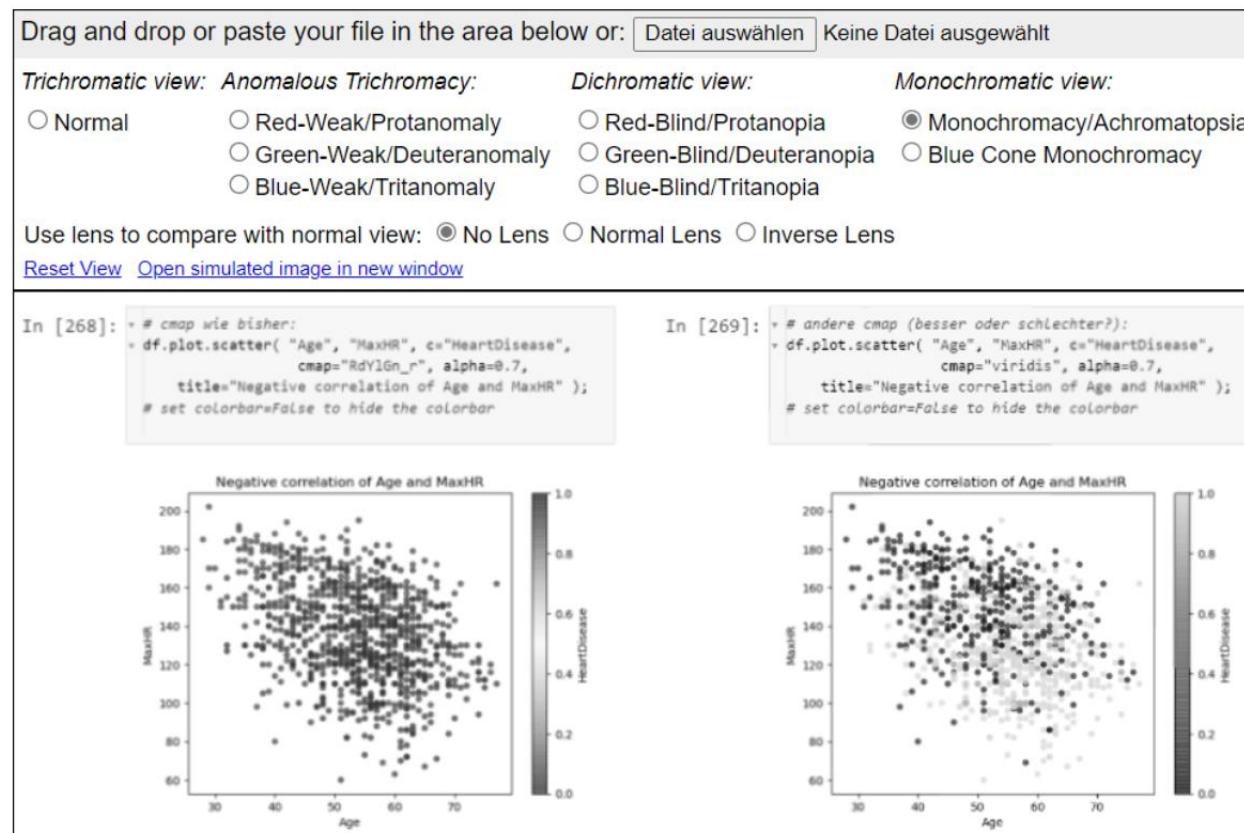
## Relevanz der Farbwahl bzgl. Sehschwächen: <https://www.color-blindness.com/coblis-color-blindness-simulator/>



Relevanz der Farbwahl bzgl. Sehschwächen: <https://www.color-blindness.com/coblis-color-blindness-simulator/>



Relevanz der Farbwahl bzgl. Sehschwächen: <https://www.color-blindness.com/coblis-color-blindness-simulator/>



Etwa 5% der Menschen hat eine Rot-Grün-Schwäche (RGS).

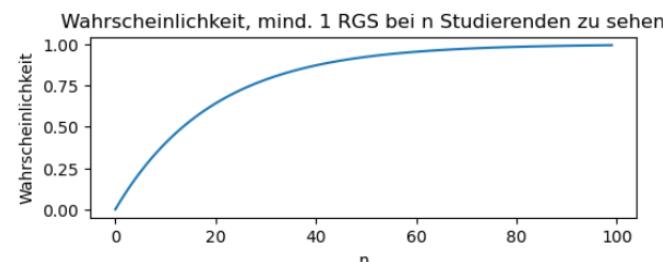
$$P(\text{mind. 1 RGS}) = 1 - P(\text{genau 0 RGS})$$

**Aufgabe:** Was ist die Wahrscheinlichkeit, dass von  $n=30$  Studierenden mindestens eine Person eine RGS hat?

$$= 1 - \prod_{i=1}^n P(\text{Person } i \text{ hat keine RGS})$$

$$= 1 - (1 - 0.05)^n = 78.5\%$$

```
In [12]: 1 x = np.arange(0,100); y = 1-(1-0.05)**x
2 fig, ax = plt.subplots ( figsize=(6,2) )
3 ax.plot ( x, y )
4 ax.set_title ( "Wahrscheinlichkeit, mind. 1 RGS bei "+
5                 "n Studierenden zu sehen" )
6 ax.set_xlabel("n"); ax.set_ylabel("Wahrscheinlichkeit")
7
```



**Best practice:** Sehschwächen sollten berücksichtigt werden.

**Darüberhinaus wichtig:** Unwahrscheinliche Ereignisse werden wahrscheinlich, wenn sie genügend Möglichkeiten haben, einzutreten.

## 1.2 Wiederholung: Konfusionsmatrix und Accuracy-Metrik

```
In [216]: 1 y      = [1,1,1,0,0,0,0,0] # wahr  
2 pred   = [1,1,0,0,0,1,1,1] # vorhergesagt  
3
```

True Positives (TP): ?

False Positives (FP): ?

True Negatives (TN): ?

False Negatives (FN): ?

```
In [217]: 1 TP   = 2  
2 FP   = 4  
3 TN   = 2  
4 FN   = 1  
5  
6 accuracy = (TP+TN)/(TP+TN+FP+FN)  
7 accuracy  
8
```

Out[217]: 0.4444444444444444

```
In [218]: 1 from sklearn.metrics import accuracy_score  
2 accuracy_score ( y, pred )  
3
```

Out[218]: 0.4444444444444444

## 1.3 Wiederholung: Overfitting beim Entscheidungsbaum

Wir haben bereits gesehen, dass ein Machine Learning Modells (konkret: ein Entscheidungsbaum zur binären Klassifikation) auf den Testdaten schlechter performen kann als auf den Trainingsdaten. Man spricht von einem **Overfit**.

### Gründe:

- Das Modell verwendet Unterschiede in den Merkmalen (Features) zwischen den Gruppen\* "krank" und "gesund", die allein aus Zufallsgründen in der verwendeten Trainingsdatenmenge vorliegen.
- Man sagt, **das Modell generalisiert nicht** (oder: "das Modell lernt die Trainingsdaten auswendig") wenn diese verwendeten Unterschiede auf neuen Daten nicht mehr nutzbar sind.

\* bei einer Regression oder einer Multiclass-Klassifikation analog

### Gegenmaßnahmen, um den Zufallseinfluss zu verringern:

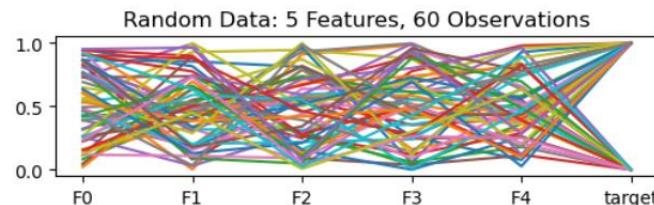
- Erhöhung der zum Training verwendeten Beobachtungen (mehr Zeilen im Dataframe)
- Verringerung der verwendeten Features (weniger Spalten im Dataframe)

### Gegenmaßnahmen, um das Modell robuster zu machen:

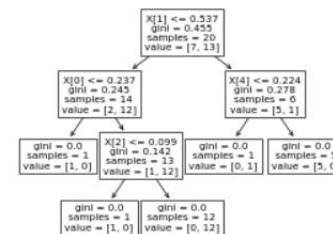
- Verringerung der Parameterzahl (z.B. Verringerung der Baumtiefe bei einem Entscheidungsbaum)
- Regularisierung (hohe Parameterwerte werden durch Gewichtsterme bestraft, z.B. "weight decay" bei neuronalen Netzwerken)
- early stopping bei iterativen Verfahren, ...

## Beispiel für eine Abweichung der Performance auf Trainings- und Testdaten ("Overfit"):

```
In [212]: 1 # best practice: use a seeded random number generator
2 R = np.random.default_rng ( seed=13 )
3
4 # random data: 5 features, 1 target value, 60 rows
5 df = pd.DataFrame ( R.uniform( size=(60,6) ) )
6 df.columns = ["F"+str(i) for i in range(5)]+["target"]
7
8 # make the target variable binary
9 df.iloc[:, -1] = df.iloc[:, -1].round().astype("int")
10 df.T.plot(legend=False, title="Random Data: 5 Features"
11 plt.gcf().set_size_inches( (6,1.4) ) # make smaller
```



```
In [213]: 1 # Split into training and validation set
2 X_train = df.iloc[:20,:-1]; X_val = df.iloc[20:40,:-1]
3 y_train = df.iloc[:20,-1]; y_val = df.iloc[20:40,-1]
4
5 # Train the decision tree
6 from sklearn import tree
7 clf = tree.DecisionTreeClassifier ( random_state=0 )
8 clf.fit ( X_train, y_train )
9 tree.plot_tree ( clf )
10 plt.gcf().set_dpi ( 50 ) # make smaller
```



```
In [219]: 1 # Training accuracy
2 pred_train = clf.predict ( X_train )
3 accuracy_score ( pred_train, y_train )
4
```

Out[219]: 1.0

```
In [220]: 1 # Validation accuracy => Overfit
2 pred_val = clf.predict ( X_val )
3 accuracy_score ( pred_val, y_val )
4
```

Out[220]: 0.5

**Immer wichtig:**

Kontrolle des Overfits auf einem Datensatz, der nicht zum Training verwendet wurde:

- "Testdatensatz" beim einmaligen Trainieren eines Machine Learning Modells
- "Validierungsdatensatz", wenn mehrere Modelle oder mehrere Trainingsdurchgänge betrachtet werden sollen  
(danach EINMALIGE Auswertung auf dem Testdatensatz für das finale Modell)

## 1.4 Wiederholung: Hyperparameter-Tuning

Machine Learning Modelle haben sog. **Hyperparameter**, die vor dem Training vom Anwender gesetzt werden und das Trainingsverhalten bestimmen (z.B. Baumtiefe beim Entscheidungsbaum).

"Hyper" in Abgrenzung zu den eigentlichen Modellparametern, die der Algorithmus während des Trainings bestimmt.

Wenn Hyperparameter versuchsweise gesetzt werden und dann, nach dem Training, angepasst werden, um die Modell-Performance zu verbessern, spricht man von **Hyperparameter-Tuning**.

Hierbei muss aufgepasst werden, dass -- obwohl danach der Overfit auf Testdaten evaluiert wird -- durch **multiples Testen** das gewählte Modell besser aussieht, als es tatsächlich ist (auch das ist Overfit!):

- Aus Zufallsgründen sehen einige Modelle besser aus als andere. Wenn "genügend viele" Modelle betrachtet werden, kann die Performance "beliebig gut" aussehen.
- Daher Modellauswahl (bzw. Hyperparameter-Tuning) auf einem separaten Validierungsdatensatz und finale Bewertung auf einem **nur einmal verwendeten Testdatensatz**.

## Beispiel für eine scheinbar gute Performance auf mehrmals verwendeten Validierungsdaten:

```
In [477]: 1 acc_train = dict()
2 acc_val   = dict()
3
4 for max_depth in [1,2,3,4,5]:
5     for min_samples_leaf in [1,2,3,4,5]:
6         for max_features in [1,2,3,4,5]: # another hyperparameter (usually not used for a single tree)
7             # train a decision tree with the specified hyperparameters
8             clf = tree.DecisionTreeClassifier ( max_depth = max_depth, min_samples_leaf=min_samples_leaf,
9                                              max_features = max_features,
10                                             random_state=0 )
11             clf.fit ( X_train, y_train )
12
13             # calculate the accuracy metric on the training and the validation set
14             pred_train = clf.predict ( X_train )
15             pred_val = clf.predict ( X_val )
16             acc_train[ (max_depth, min_samples_leaf, max_features) ] = accuracy_score ( pred_train, y_train ) # from sklearn
17             acc_val [ (max_depth, min_samples_leaf, max_features) ] = accuracy_score ( pred_val, y_val )
```

```
In [478]: 1 # the best accuracy values on the validation set
2 # (indexed by the hyper-parameters)
3 pd.Series(acc_val).sort_values(ascending=False).head()
4
```

```
Out[478]: 3 5 2 0.7
5 5 2 0.7
4 5 2 0.7
2 5 2 0.7
1 5 2 0.6
dtype: float64
```

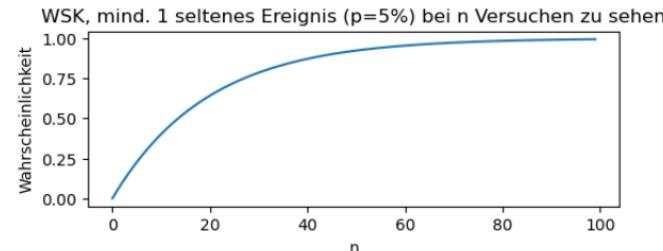
```
In [224]: 1 # re-train with the optimal choice
2 clf = tree.DecisionTreeClassifier ( random_state=0,
3                                   max_depth=3, min_samples_leaf=5, max_features=2 )
4 clf.fit ( X_train, y_train )
5
6 # accuracy on the test set (=> overfit!)
7 X_test = df.iloc[40:,:-1]; y_test = df.iloc[40:,-1]
8 pred_test = clf.predict ( X_test )
9 accuracy_score ( pred_test, y_test )
10
```

```
Out[224]: 0.5
```

## 1.5 Zusammenfassung: Holdout-Methode

In [472]:

```
1 x = np.arange(0,100); y = 1-(1-0.05)**x
2 fig, ax = plt.subplots ( figsize=(6,2) )
3 ax.plot ( x, y )
4 ax.set_title ( "WSK, mind. 1 seltenes Ereignis (p=5%)"  
" bei n Versuchen zu sehen" )
5
6 ax.set_xlabel("n"); ax.set_ylabel("Wahrscheinlichkeit")
7
```



Das Unterteilen der Daten in Trainings-, Validierungs- und Testdatensatz wird **Holdout-Methode** genannt.

- Auf dem **Trainingsdatensatz** wird ein Machine Learning Modell trainiert.
- Der **Validierungsdatensatz** dient zur Überprüfung, ob ein Overfit vorliegt: Wie gut ist das Modell auf Daten, die nicht zum Trainieren verwendet wurden?
- Beim Hyperparameter-Tuning werden viele Modelle trainiert; es besteht die Gefahr, dass beim Vergleich der Validierungsperformance ein Modell nur aus Zufallsgründen "gut aussieht".

Daher wird das finale Modell einmalig auf dem **Testdatensatz** evaluiert, um eine finale Aussage über die Generalisierungsfähigkeit zu erhalten.

(Die Abbildung links zeigt den "worst case", wenn die Ereignisse unabhängig sind, wie bei der RGS. Beim Hyperparameter-Tuning werden Performances der Modelle positiv korrelieren (d.h. recht ähnlich sein). Es wird daher länger dauern, bis man aus Zufallsgründen gute Ergebnisse sieht.)

## 1.6 Der Tree-Split-Algorithmus `tree.DecisionTreeClassifier().fit(X,y)`

Es seien Daten  $Q$  gegeben mit  $(x, y) \in Q$ , wobei  $x \in \mathbb{R}^n$  Features sind und  $y \in \{0, 1\}$  Labels.

Es sei  $p(Q)$  der Anteil der " $y = 1$ "-Labels in  $Q$ . Dann ist die **Gini-Impurity** definiert via  $H(Q) = 2p(Q)(1 - p(Q))$ .

**Algorithmus:** Modifikation des CART-Algorithmus: <https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation>; siehe auch ID3, C4.5 und C5.0 für alternative Implementierungen.

1. Für jedes Feature  $j \in \{1, \dots, n\}$  und jeden möglichen Splitwert  $t$ , zerlege die Daten in Mengen  $Q^{\text{left}}(j, t)$  und  $Q^{\text{right}}(j, t)$ , definiert via

$$\begin{aligned}Q^{\text{left}}(j, t) &= \{(x, y) \in Q \mid x_j \leq t\} \\Q^{\text{right}}(j, t) &= Q \setminus Q^{\text{left}}(j, t)\end{aligned}$$

2. Die Qualität eines Splits  $\theta = (j, t)$  wird bewertet durch eine Impurity-Funktion  $H$  (z.B. Gini-Impurity oder Entropie):

$$\text{gemittelte Impurity nach dem Split: } G(Q, \theta) = \frac{N^{\text{left}}}{N} H(Q^{\text{left}}(\theta)) + \frac{N^{\text{right}}}{N} H(Q^{\text{right}}(\theta))$$

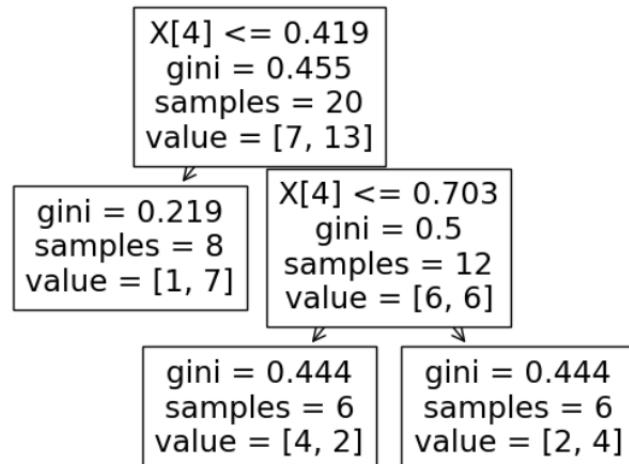
Hierbei ist  $N = |Q|$  und  $N^{\text{left}} = |Q^{\text{left}}(\theta)|$ ,  $N^{\text{right}} = |Q^{\text{right}}(\theta)|$ .

3. Wähle den Split  $\theta^*$ , der die niedrigste gemittelte Impurity  $G(Q, \theta)$  erzeugt:  $\theta^* = \operatorname{argmin}_{\theta} G(Q, \theta)$ .

4. Fahre rekursiv fort auf  $Q^{\text{left}}(\theta^*)$  und  $Q^{\text{right}}(\theta^*)$ , bis eine geeignete Abbruchbedingung erreicht ist: "until max\_depth is reached,  $N < \text{min\_samples}$  or  $N = 1$ ".

## 1.7 Inference: Wie funktioniert `clf.predict?`?

```
In [227]: 1 # Der im Beispiel verwendete Entscheidungsbaum
2 tree.plot_tree ( clf );
3
```



```
In [229]: 1 X_test.head()
2
```

Out[229]:

	F0	F1	F2	F3	F4
40	0.867332	0.336623	0.518845	0.726491	0.561026
41	0.021726	0.696199	0.131128	0.516053	0.392497
42	0.767732	0.090596	0.049037	0.937811	0.486237
43	0.152259	0.503091	0.666234	0.894390	0.390633
44	0.756945	0.304682	0.714374	0.801033	0.963154

```
In [230]: 1 # Predictions
2 clf.predict ( X_test.head() )
3
```

Out[230]: array([0, 1, 0, 1, 1])

### Algorithmus:

- Für jede Observation (=Zeile in `X_test`), ermittle das zugehörige Blatt im Entscheidungsbaum.
- Wähle als Vorhersage die in diesem Blatt überwiegende Klasse.

### clf.predict vs. clf.predict\_proba:

```
In [233]: 1 tree.plot_tree ( clf );  
2
```

X[4] <= 0.419  
gini = 0.455  
samples = 20  
value = [7, 13]

gini = 0.219  
samples = 8  
value = [1, 7]

X[4] <= 0.703  
gini = 0.5  
samples = 12  
value = [6, 6]

gini = 0.444  
samples = 6  
value = [4, 2]

gini = 0.444  
samples = 6  
value = [2, 4]

```
In [229]: 1 X_test.head()  
2
```

Out[229]:

	F0	F1	F2	F3	F4
40	0.867332	0.336623	0.518845	0.726491	0.561026
41	0.021726	0.696199	0.131128	0.516053	0.392497
42	0.767732	0.090596	0.049037	0.937811	0.486237
43	0.152259	0.503091	0.666234	0.894390	0.390633
44	0.756945	0.304682	0.714374	0.801033	0.963154

```
In [235]:
```

```
1 # Die der Klassifikation zugrundeliegenden  
2 # Klassenverhältnisse in den jeweiligen Blättern  
3 # (die zweite Spalte nennt man "Scores")  
4 clf.predict_proba ( X_test.head() )  
5  
6  
7
```

Out[235]: array([[0.66666667, 0.33333333],  
[0.125 , 0.875 ],  
[0.66666667, 0.33333333],  
[0.125 , 0.875 ],  
[0.33333333, 0.66666667]])

**Frage:** Wie können Sie die Predictions aus den Scores berechnen?  
Welchen Zusatznutzen haben die Scores?

Durch Vergleich mit einem Threshold wird eine Prediction zu einer 0/1-Vorhersage. Scores geben Auskunft darüber, wie "sicher" sich das Modell ist.

## Best Practices:

- Für nicht-temporale Daten sollte die Aufteilung in die Teilmengen randomisiert erfolgen.
- Es sollte zumindest nach der Zielgröße  $y$  stratifiziert werden, ggf. auch nach Features wie Geschlecht, ...; bei großen Datenmengen ist die Verteilung der Merkmale in den Teilmengen automatisch sehr ähnlich (Gesetz der großen Zahlen) und eine explizite Stratifizierung ist weniger wichtig.
- Für den Split in Train/Validation/Testset wurden hier vereinfachend die Anteile 1/3, 1/3, 1/3 gewählt. Üblicherweise werden mehr Daten fürs Training reserviert, z.B. 60/20/20 oder 80/10/10.

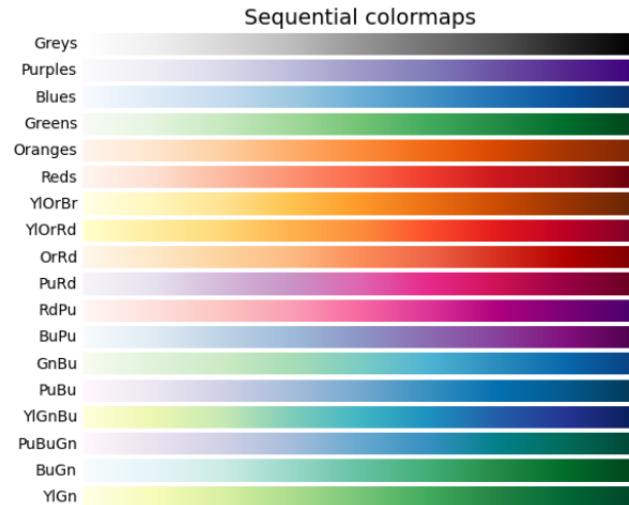
**Frage:** Welcher Trade-off welcher Zielgrößen ist zu beachten, wenn über die Aufteilung der Datenpunkte in die Trainings-, Validierungs- und Testdatensätzen entschieden wird?

Einerseits möchte man möglichst viele Datenpunkte im Trainingsdatensatz haben, damit das Modell möglichst gut lernt und ein Overfit minimiert wird.

Andererseits braucht man genügend viele Daten im Validierungs- und Testdatensatz, um Aussagen über die Generalisierungsfähigkeit auf ungesiehenen Daten und über die Güte des finalen Modells zu ermöglichen.

### Best Practices(ctd.):

- Bei **temporalen Daten** (z.B. Zeitreihen) sollte die zeitliche Struktur beim Split beibehalten werden: kleine Zeitstempel in die Trainings-, mittlere Zeitstempel in die Validierungs- und hohe Zeitstempel in die Testdaten. Denn:
  - Es soll die Generalisierungsfähigkeit überprüft werden, hierbei möchte man möglichst nahe am tatsächlichen Einsatzszenario bleiben: zukünftige Daten sollen basierend auf einem Modell, das auf vergangenen Daten trainiert wurde, vorhergesagt werden.
  - Es ist schwerer, aus einer Zeitreihe zukünftige Datenpunkte zu extrapolieren, als zufällig ausgewählte Punkte zu interpolieren. Bei einem randomisierten Split würde die Generalisierungsfähigkeit zu optimistisch bewertet werden.



<https://matplotlib.org/stable/tutorials/colors/colormaps.html>

## 1. Die Holdout-Methode

## 2. Indizierung in Pandas

## 2.1 Slicing in Pandas

```
In [126]: 1 import pandas as pd
2 df = pd.read_csv("VL02_Material/heart.csv")
3
4
5 # mit .head() zeigt man nur die ersten Zeilen an
6 df.head()
7
8
9 # mit .tail() werden die letzten Zeilen
10 # angezeigt.
11
12
13 # Mit dem Parameter n (default n=5)
14 # wird die Anzahl der Zeilen spezifiziert.
15
```

Out[126]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	I
0	40	M	ATA	140	289	0	Normal	1
1	49	F	NAP	160	180	0	Normal	1
2	37	M	ATA	130	283	0	ST	1
3	48	F	ASY	138	214	0	Normal	1
4	54	M	NAP	150	195	0	Normal	1

```
In [42]: 1 # Einfacher Zugriff auf eine Spalte
2 df["ChestPainType"].head()
3
```

Out[42]:

0	ATA
1	NAP
2	ATA
3	ASY
4	NAP

Name: ChestPainType, dtype: object

```
In [43]: 1 # einfacher Zugriff auf ausgewählte Zeilen
2 df[3:6]
3
```

Out[43]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	I
3	48	F	ASY	138	214	0	Normal	1
4	54	M	NAP	150	195	0	Normal	1
5	39	M	NAP	120	339	0	Normal	1



## Boolean Indexing (wie in Numpy):

```
In [134]: 1 # Bool'scher Vektor
           2 idx = (df["ChestPainType"] == "ASY")
           3 idx
           4
Out[134]: 0    False
           1    False
           2    False
           3    True
           4    False
           ...
          913   False
          914   True
          915   True
          916   False
          917   False
Name: ChestPainType, Length: 918, dtype: bool
```

```
In [46]: 1 # ... mit anschließender Aggregierung
           2 df["Age"][idx].mean()
           3
```

```
Out[46]: 54.95967741935484
```

```
In [132]: 1 # "Boolean Indexing"
           2 df["Age"][ idx ]
Out[132]: 3      48
           8      37
           13     49
           16     38
           18     60
           ..
          909    63
          911    59
          912    57
          914    68
          915    57
Name: Age, Length: 496, dtype: int64
```



In [139]: 1 df.head()

Out[139]:

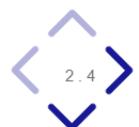
	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
0	40	M	ATA	140	289	0	Normal	172	N	0.0	Up	0
1	49	F	NAP	160	180	0	Normal	156	N	1.0	Flat	1
2	37	M	ATA	130	283	0	ST	98	N	0.0	Up	0
3	48	F	ASY	138	214	0	Normal	108	Y	1.5	Flat	1
4	54	M	NAP	150	195	0	Normal	122	N	0.0	Up	0

Integer Array Indexing (wie in Numpy):

In [138]: 1 # Spalte ChestPainType, Zeile 0 und 1  
2 df["ChestPainType"][[ 0,1 ]]  
3

Out[138]: 0 ATA  
1 NAP  
Name: ChestPainType, dtype: object

Frage: Greifen wir auf Zeilennummern zu, oder auf Zeilennamen (=Index)?



## Mehrdeutigkeiten vermeiden mit `.loc` und `.iloc`:

```
In [49]: 1 # Indizierung mit .loc verwendet Spaltennamen
          2 df.loc[:, "ChestPainType"].head()
          3
```

```
Out[49]: 0    ATA
          1    NAP
          2    ATA
          3    ASY
          4    NAP
Name: ChestPainType, dtype: object
```

```
In [51]: 1 # Zugriff auf Spaltennamen und Index
          2 df.loc[3, "ChestPainType"]
          3
```

```
Out[51]: 'ASY'
```

```
In [50]: 1 # Indizierung mit .iloc verwendet Spaltennummern
          2 df.iloc[:, 2].head()
          3
```

```
Out[50]: 0    ATA
          1    NAP
          2    ATA
          3    ASY
          4    NAP
Name: ChestPainType, dtype: object
```

```
In [52]: 1 # Zugriff auf Spalten- und Zeilennummer
          2 df.iloc[3, 2]
          3
```

```
Out[52]: 'ASY'
```

Mit `.loc` und `.iloc` können Unklarheiten vermieden werden:

In [53]:

```
1 # Zugriff auf mehrere Elemente über deren Namen
2 df.loc[[0,1],"ChestPainType"]
3
```

Out[53]:

0	ATA
1	NAP

Name: ChestPainType, dtype: object

In [54]:

```
1 # Zugriff auf mehrere Elemente über deren Nummern
2 df.iloc[[0,1],2]
3
```

Out[54]:

0	ATA
1	NAP

Name: ChestPainType, dtype: object

In [55]:

```
1 # jetzt stimmen Zeilennamen (=Index) nicht mehr mit der natürlichen Nummerierung überein
2 df = df[::-1]
3 df.head()
```

Out[55]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
917	38	M	NAP	138	175	0	Normal	173	N	0.0	Up	0
916	57	F	ATA	130	236	0	LVH	174	N	0.0	Flat	1
915	57	M	ASY	130	131	0	Normal	115	Y	1.2	Flat	1
914	68	M	ASY	144	193	1	Normal	141	N	3.4	Flat	1
913	45	M	TA	110	264	0	Normal	132	N	1.2	Flat	1

In [56]:

```
1 # Zugriff auf mehrere Elemente über deren Namen
2 df.loc[[0,1],"ChestPainType"]
3
```

Out[56]:

0	ATA
1	NAP

Name: ChestPainType, dtype: object

In [57]:

```
1 # Zugriff auf mehrere Elemente über deren Nummern
2 df.iloc[[0,1],2]
3
```

Out[57]:

917	NAP
916	ATA

Name: ChestPainType, dtype: object

## 2.2 Index in Pandas: "Spaltennamen für die Zeilen"

Neuer Index:

```
In [58]: 1 # mit reset_index wird der Index zu einer Spalte, und ein "generischer" Index wird hinzugefügt
2 # (Rückgabe als Kopie, df bleibt unverändert)
3 df.reset_index().head()
4
```

```
Out[58]:   index  Age  Sex ChestPainType  RestingBP  Cholesterol  FastingBS  RestingECG  MaxHR  ExerciseAngina  Oldpeak  ST_Slope  HeartDisease
0  917    38    M      NAP          138         175           0     Normal       173        N        0.0      Up        0
1  916    57    F     ATA          130         236           0      LVH        174        N        0.0     Flat        1
2  915    57    M     ASY          130         131           0     Normal       115        Y        1.2     Flat        1
3  914    68    M     ASY          144         193           1     Normal       141        N        3.4     Flat        1
4  913    45    M     TA          110         264           0     Normal       132        N        1.2     Flat        1
```

Überschreiben des alten Index durch einen neuen:

```
In [59]: 1 # setze neuen Index (Rückgabewert ist eine Kopie)
2 df.set_index("Age").head()
3
```

Out[59]:

Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
38	M	NAP	138	175	0	Normal	173	N	0.0	Up	0
57	F	ATA	130	236	0	LVH	174	N	0.0	Flat	1
57	M	ASY	130	131	0	Normal	115	Y	1.2	Flat	1
68	M	ASY	144	193	1	Normal	141	N	3.4	Flat	1
45	M	TA	110	264	0	Normal	132	N	1.2	Flat	1

Manuelle Zuweisung eines Index (der nicht eindeutig sein muss):

```
In [60]: 1 import numpy as np  
2 df.index = np.where( df.index % 2 == 0, "A", "B")  
3 df
```

```
Out[60]:
```

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG
B	38	M	NAP	138	175	0	Normal
A	57	F	ATA	130	236	0	LVH
B	57	M	ASY	130	131	0	Normal
A	68	M	ASY	144	193	1	Normal
B	45	M	TA	110	264	0	Normal
...	...	...	...	...	...	...	...
A	54	M	NAP	150	195	0	Normal
B	48	F	ASY	138	214	0	Normal
A	37	M	ATA	130	283	0	ST
B	49	F	NAP	160	180	0	Normal
A	40	M	ATA	140	289	0	Normal

918 rows × 12 columns

```
In [61]: 1 df.loc["A","ChestPainType"]
```

```
Out[61]: A    ATA  
A    ASY  
A    ASY  
A    ATA  
A    ASY  
...  
A    ASY  
A    ATA  
A    NAP  
A    ATA  
A    ATA  
Name: ChestPainType, Length: 459, dtype: object
```

```
In [62]: 1 # Hier wurde np.where verwendet:  
2 # Dies ist ein vektorisiertes "ternary" if/else  
3 np.where( [True, False, True], [1, 2, 3], ["A", "B", "C"] )
```

```
Out[62]: array(['1', 'B', '3'], dtype='<U11')
```

Ausblick: Verwendung mehrerer Spalten zur Indizierung resultiert in einem sog. Multiindex:

In [140]: 1 df.set\_index( ["Sex","Age"] ).sort\_index()

Out[140]:

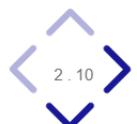
		ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
Sex	Age										
	30	TA	170	237	0	ST	170	N	0.0	Up	0
	31	ATA	100	219	0	ST	150	N	0.0	Up	0
F	32	ATA	105	198	0	Normal	165	N	0.0	Up	0
	33	ASY	100	246	0	Normal	150	Y	1.0	Flat	1
	34	ATA	130	161	0	Normal	190	N	0.0	Up	0
	...	...	...	...	...	...	...	...	...	...	...
	75	ASY	136	225	0	Normal	112	Y	3.0	Flat	1
	75	ASY	160	310	1	Normal	112	Y	2.0	Down	0
M	76	NAP	104	113	0	LVH	120	N	3.5	Down	1
	77	ASY	124	171	0	ST	110	Y	2.0	Up	1
	77	ASY	125	304	0	LVH	162	Y	0.0	Up	1

918 rows × 10 columns

In [8]: 1 # Zugriff mit "xs" auf die verschiedenen Index-Levels ("liefer mir alle 31-jährigen")  
2 df.set\_index( ["Sex","Age"] ).xs(31, axis=0, level=1)

Out[8]:

		ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	ST_Slope	HeartDisease
Sex											
M	ASY		120	270	0	Normal	153	Y	1.5	Flat	1
F	ATA		100	219	0	ST	150	N	0.0	Up	0



## 2.3 train\_test\_split für einen randomisierten Split

```
In [147]: 1 df = pd.read_csv("VL02_Material/heart.csv")
```

```
In [148]: 1 from sklearn.model_selection import train_test_split
2 X = df.iloc[:, :-1]
3 y = df.iloc[:, -1] # Zielgröße HeartDisease
4
```

Verwende `train_test_split`, um Trainingsdaten abzuseparieren:

```
In [149]: 1 # train_size als float (=Anteil der Gesamtmenge)
2 # oder als int (=Anzahl Datenpunkte)
3 X_train, X_rest, y_train, y_rest = train_test_split(
4     X, y, train_size=1/3, random_state=123 )
5
6 len(X_train), len(X_rest), \
7     y_train.mean(), y_rest.mean()
8 # 60% der Personen in den Trainingsdaten sind krank
```

Out[149]: (306, 612, 0.6013071895424836, 0.5294117647058824)

```
In [150]: 1 # Argument "stratify", um die Werte angegebener
2 # Spalten gleichmäßig auf die Zielmengen zu verteilen
3 X_train, X_rest, y_train, y_rest = train_test_split(
4     X, y, train_size=1/3, random_state=123, stratify=y )
5
6 len(X_train), len(X_rest), \
7     y_train.mean(), y_rest.mean()
8 # je 55% der Trainings- UND Testpatienten sind krank
```

Out[150]: (306, 612, 0.5522875816993464, 0.553921568627451)



Erneuter Split, um weiter in Validierungs- und Testdaten aufzuteilen mit `train_test_split(..., train_size=?)`:

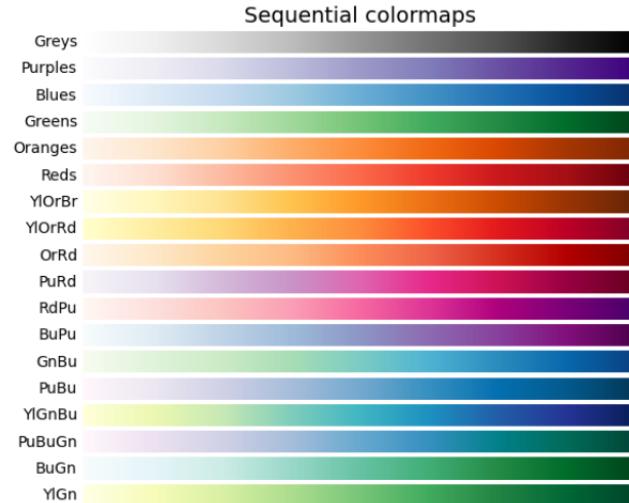
```
In [151]:  
1 X_val, X_test, y_val, y_test = train_test_split (  
2     X_rest,y_rest,train_size=1/2,random_state=456 )  
3  
4  
5 len(X_train), len(X_val), len(X_test),\  
6     y_train.sum(), y_val.sum(), y_test.sum()  
7
```

Out[151]: (306, 306, 306, 169, 178, 161)

```
In [152]:  
1 # erneut mit Stratifizierung:  
2 X_val, X_test, y_val, y_test = train_test_split (  
3     X_rest,y_rest,train_size=1/2,random_state=456,  
4     stratify=y_rest )  
5  
6  
7 len(X_train), len(X_val), len(X_test),\  
8     y_train.sum(), y_val.sum(), y_test.sum()  
9
```

Out[152]: (306, 306, 306, 169, 169, 170)





<https://matplotlib.org/stable/tutorials/colors/colormaps.html>

## 1. Die Holdout-Methode

## 2. Indizierung in Pandas

**Vielen Dank für Ihre  
Aufmerksamkeit!**