



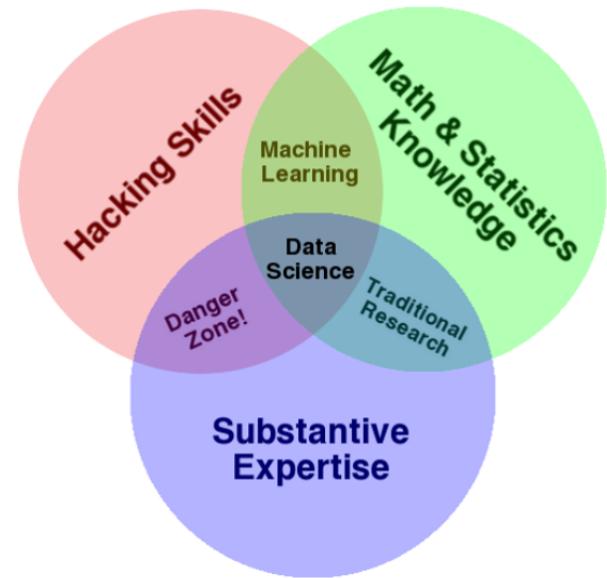
## 1. Grundbegriffe Data Science / Data Analytics

## 2. Python-Grundlagen

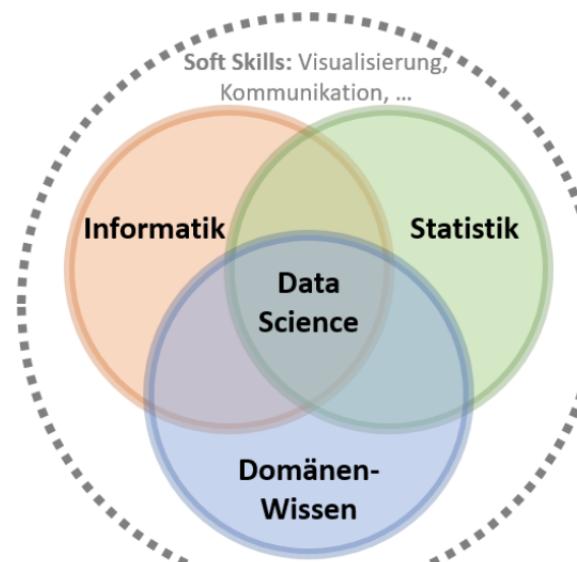
## 3. Einfache Visualisierungen

# 1. Grundbegriffe Data Science / Data Analytics

## 1.1 Das Data Science Venn-Diagramm

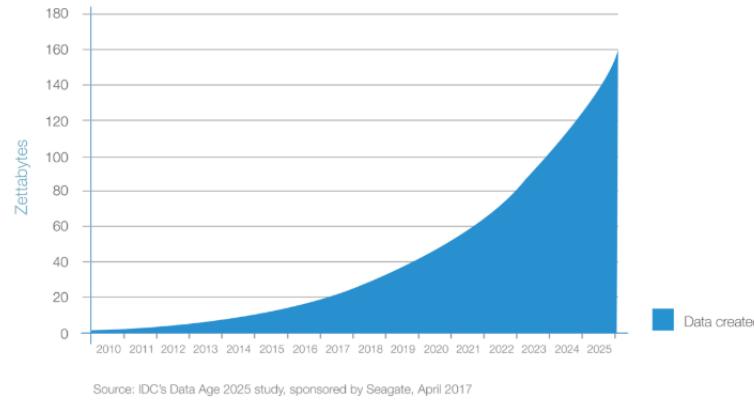


<http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>



## 1.2 Daten

Figure 2. Annual Size of the Global Datasphere



Source: IDC's Data Age 2025 study, sponsored by Seagate, April 2017

1 Zettabyte =  $1024^4$  GB = 1.099.511.627.776 GB

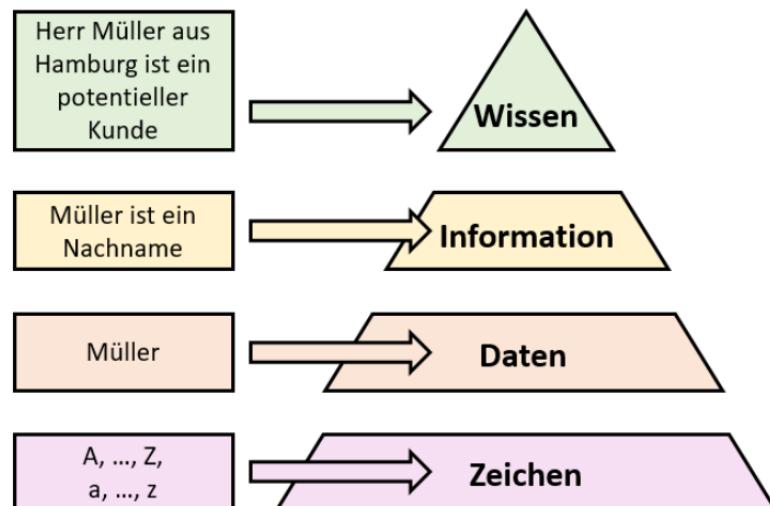
### Definition:

- (durch Beobachtungen, Messungen, statistische Erhebungen u. a. gewonnene) [Zahlen]werte,
- (auf Beobachtungen, Messungen, statistischen Erhebungen u. a. beruhende) Angaben, formulierbare Befunde,
- elektronisch gespeicherte Zeichen, Angaben, Informationen,
- zur Lösung oder Durchrechnung einer Aufgabe vorgegebene Zahlenwerte, Größen

Duden



## Von Zeichen zu Wissen:



Nach: F. Bodendorf: Daten- und Wissensmanagement. 2. Auflage, Springer Verlag, 2006, Abb. 1.1

## Unterscheidungen:

**Syntax** (=Regelsystem für Zeichen) vs.  
**Semantik** (=Bedeutung der Zeichen/Wörter im Kontext)

**Unstrukturierte Daten** (z.B. Video, Bild, Text) vs.  
**Strukturierte Daten** (z.B. Datenbank, xml, csv)

**Ungelabelte Daten** vs.  
**Gelabelte Daten** (Zielgrößen für eine  
Vorhersage sind vorhanden, diese nennt man *Ground Truth*)

## 1.3 Machine Learning: Grundbegriffe



### Unsupervised Learning:

- keine Labels notwendig
- Clustering oder Dimensionsreduktion

### Supervised Learning:

- Labels sind notwendig
- **Regression** für kontinuierliche Zielgröße,  
**Klassifikation** für diskrete Zielgröße

### Reinforcement Learning:

- Training eines handlungsfähigen Agenten

## Beispieldatensatz für die Anwendung von Unsupervised und Supervised Learning:

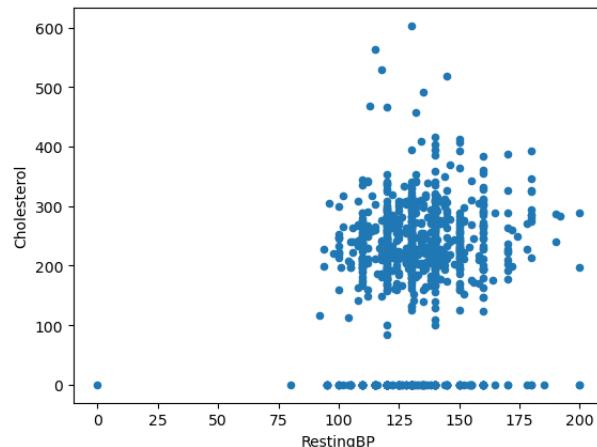
```
In [1]: 1 import pandas as pd # (pandas noch nicht in dieser VL)
2 df = pd.read_csv ( "VL02_Material/heart.csv" )
3
4 df # pandas DataFrame
5
```

```
Out[1]:   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
...
913   45    M       TA       110      264         0     Normal     132        N      1.2      Flat      1
914   68    M       ASY       144      193         1     Normal     141        N      3.4      Flat      1
915   57    M       ASY       130      131         0     Normal     115        Y      1.2      Flat      1
916   57    F        ATA       130      236         0      LVH      174        N      0.0      Flat      1
917   38    M       NAP       138      175         0     Normal     173        N      0.0       Up      0
```

918 rows × 12 columns

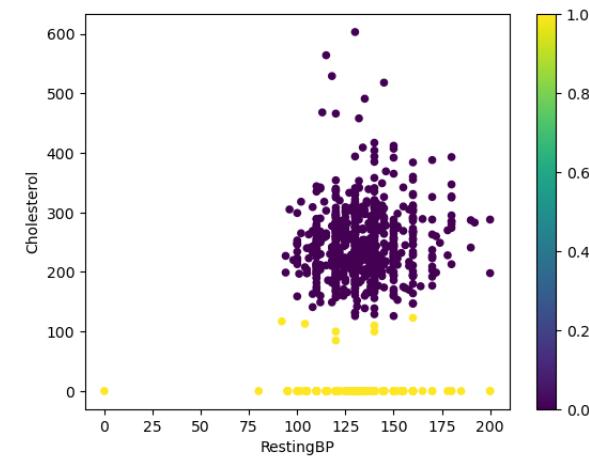
## Unsupervised Learning (Clustering):

```
In [2]: 1 # Visualisiere zwei ausgewählte Spalten
2 # in einem sog. Scatterplot
3 df.plot.scatter(x="RestingBP",y="Cholesterol",
4                  cmap=None);
5
6
7
8
9
10
```



```
In [27]: 1 # Wende das sog. Kmeans-Clustering an
2 from sklearn.cluster import KMeans
3 kmeans = KMeans(n_clusters=2,n_init=10)
4 kmeans.fit(df[["RestingBP","Cholesterol"]])
5
6 # Visualisiere erneut, nun mit Einfärbung
7 df.plot.scatter(x="RestingBP",y="Cholesterol",
8                  c=kmeans.labels_, cmap="viridis");
9
```

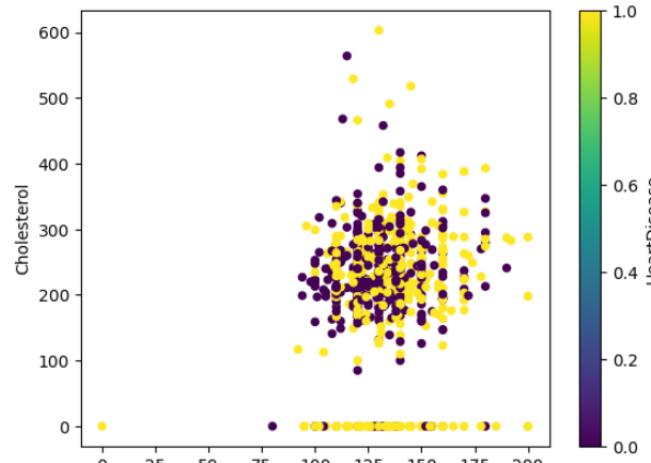
C:\Users\schirmef\Anaconda3\envs\ttz\_base\lib\site-packages\scklearn\cluster\\_kmeans.py:1382: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP\_NUM\_THREADS=4.  
warnings.warn(



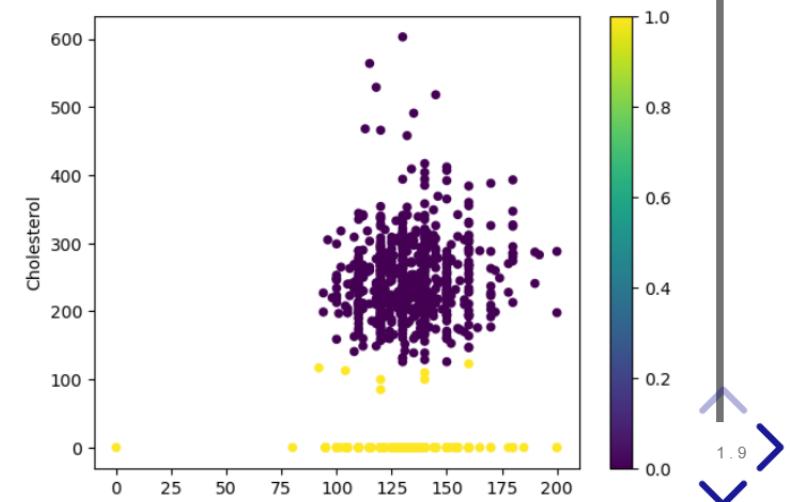
```
In [4]: 1 # Die Variable kmeans.labels_ wurde im Schritt kmeans.fit() gefüllt. Für jeden der 918 Patienten wird eine 0 oder 1  
2 # gesetzt, je nachdem, welchem Cluster der Patient zugeordnet wird. Dieses Verfahren ist "unsupervised", weil keine  
3 # a priori vorhandene Information, was denn die "wahren" Cluster sind, verwendet wird.  
4 kmeans.labels_
```

## Supervised Learning (Klassifikation):

```
In [27]: 1 # Visualisiere zwei ausgewählte Spalten
2 # in einem sog. Scatterplot
3 df.plot.scatter(x="RestingBP",y="Cholesterol",
4                  c="HeartDisease", cmap="viridis");
5
6 # Hierbei wird mit c="HeartDisease" gemäß der
7 # Zielgröße "HeartDisease" eingefärbt.
8 #
9 # Diese Zielgröße nimmt nur den Wert 0 oder 1 an,
10 # daher spricht man von einer (binären)
11 # Klassifikation, die wir nachfolgend durchführen
12 # möchten.
13
```

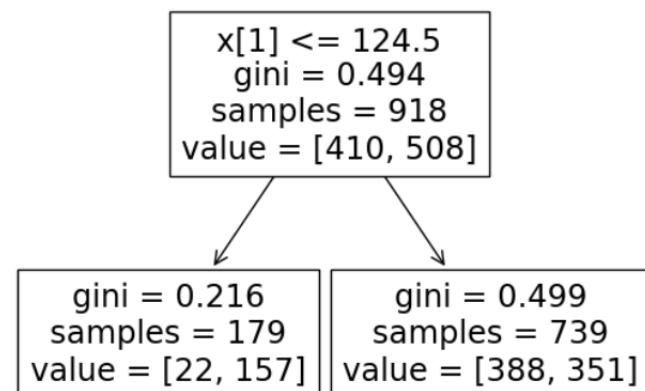


```
In [39]: 1 # Wende ein einfaches Klassifizierungsverfahren an
2 from sklearn.tree import DecisionTreeClassifier
3 clf = DecisionTreeClassifier(max_depth=1)
4 clf.fit( X=df[["RestingBP","Cholesterol"]],
5          y=df["HeartDisease"] )
6
7 # Visualisiere die Vorentscheidung auf den
8 # Trainingsdaten (sinnvoller wäre: auf neuen Daten)
9 predictions = clf.predict(
10    df[["RestingBP","Cholesterol"]])
11 df.plot.scatter(x="RestingBP",y="Cholesterol",
12                  c=predictions, cmap="viridis");
```

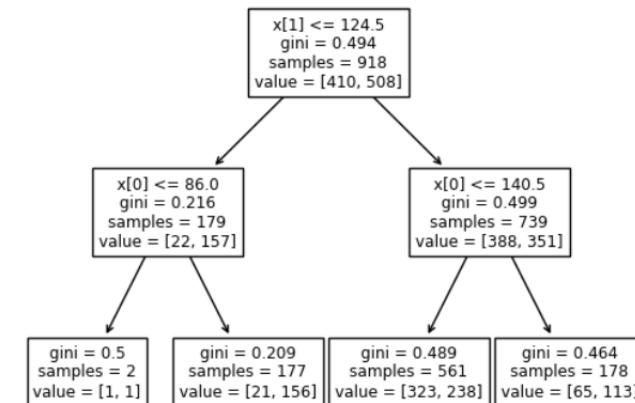


## Analyse der Klassifikation (d.h. der Entscheidungsregeln im Entscheidungsbaum):

```
In [42]: 1 from sklearn import tree
2 clf = DecisionTreeClassifier(max_depth=1
3                               ).fit(
4       X=df[["RestingBP", "Cholesterol"]], 
5       y=df["HeartDisease"])
6 tree.plot_tree(clf);
7
8 # Bei Feature 1 (=Cholesterol) wird beim
9 # Wert 124.5 gesplittet (Links: x[1] <= 124.5).
10 # Linker Zweig ist "ja", rechter Zweig ist "nein".
```



```
In [45]: 1 # Wiederhole den Fit, erlaube tiefere Bäume
2 clf = DecisionTreeClassifier(max_depth=2).fit(
3   X=df[["RestingBP", "Cholesterol"]],
4   y=df["HeartDisease"])
5 tree.plot_tree(clf);
6
7 # Danach wird bei Feature 0 (=RestingBP) weiter
8 # gesplittet (später dazu mehr)
```



## Erklärung der Zeilen in den Entscheidungsknoten im Baum:

```
x[1] <= 124.5
gini = 0.494
samples = 918
value = [410, 508]
```

Zeile	Inhalt	Erklärung
1	x[1] <= 124.5	Entscheidungsregel: Patienten, für die "Feature 1" <= 124.5 ist, landen im linken Zweig, sonst im rechten.
2	gini = 0.494	Der Baumerzeugungsalgorithmus probiert alle möglichen Features und alle möglichen Split-Niveaus durch. Jede Split-Möglichkeit wird mit der <i>gini-Impurity</i> bewertet. Die Split-Möglichkeit mit der niedrigsten gini-Impurity wird gewählt und der Algorithmus wird rekursiv in den Teilzweigen fortgesetzt.
3	samples = 918	Der gini-Wert gibt das Maß der Durchmischtheit an: Der gini-Wert ist hoch (0.5 bei binärer Klassifikation), wenn alle Klassen gleich häufig vertreten ist. Der gini-Wert ist 0, wenn die Teilpopulation zu 100% aus einer Klasse besteht. Es wird jeweils die Durchmischtheit in den beiden durch den Split entstehenden Teilmengen berechnet und ein gewichtetes Mittel gebildet.
4	value = [410, 508]	Die Anzahl der Patienten mit HeartDisease=0 bzw. HeartDisease=1 vor dem Split.

## Eine Spalte vs. zwei Spalten:

```
In [17]: 1 # Zugriff auf eine Spalte;  
2 # Ergebnis ist eine Pandas Series  
3 df["RestingBP"]
```

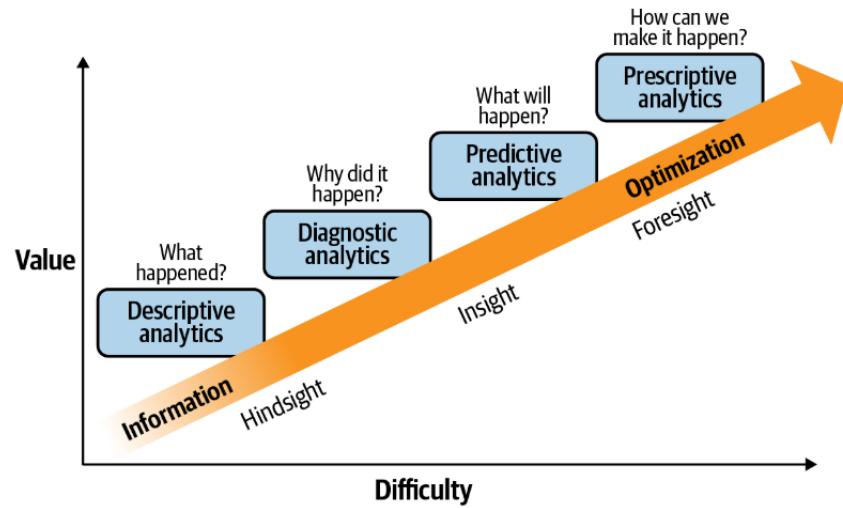
```
Out[17]: 0    140  
1    160  
2    130  
3    138  
4    150  
...  
913   110  
914   144  
915   130  
916   130  
917   138  
Name: RestingBP, Length: 918, dtype: int64
```

```
In [18]: 1 # Zugriff auf zwei Spalten;  
2 # Ergebnis ist ein Pandas DataFrame  
3 df[["RestingBP", "Cholesterol"]]
```

```
Out[18]:   RestingBP  Cholesterol  
0        140        289  
1        160        180  
2        130        283  
3        138        214  
4        150        195  
...      ...      ...  
913     110        264  
914     144        193  
915     130        131  
916     130        236  
917     138        175
```

918 rows × 2 columns

## 1.4 Data Analytics: Das Gartner Analytics Ascendancy Model



J. Finn, G. Troughton: Predictive Analytics for Healthcare, O'Reilly 2020

**1. Descriptive Analytics:** Daten Beschreiben, Hypothesen generieren

**2. Diagnostic Analytics:** Erklärungen finden, Hypothesen testen

**3. Predictive Analytics:** Vorhersagemodelle erstellen, Eintrittswahrscheinlichkeiten berechnen (Klassifikation/Regression)

**4. Prescriptive Analytics:** Das Gesamtsystem optimieren, Handlungen bewerten und Empfehlungen ableiten

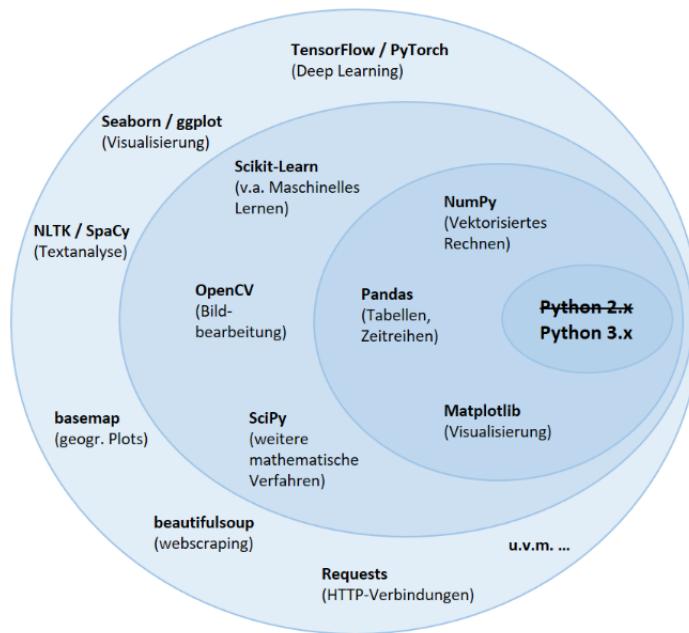


## 1. Grundbegriffe Data Science / Data Analytics

## 2. Python-Grundlagen

## 3. Einfache Visualisierungen

## 2. Python-Grundlagen



### Tutorial:

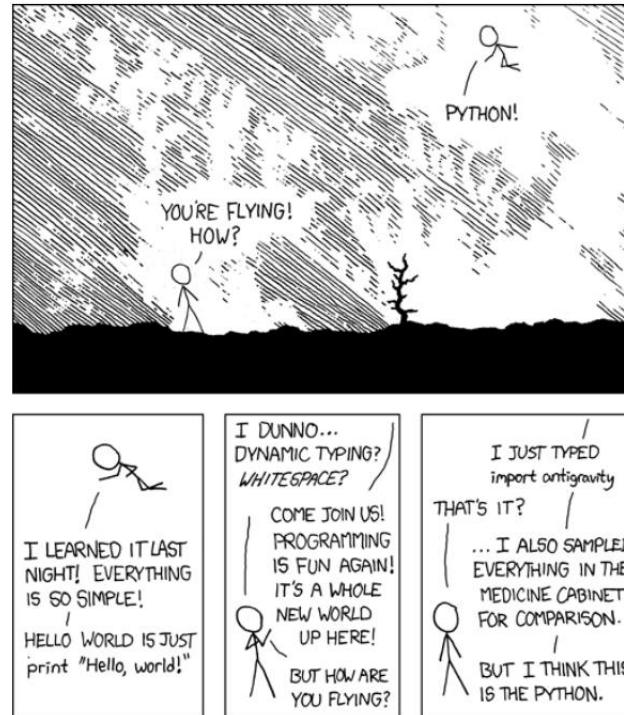
- Erstellt von Justin Johnson, Universität Stanford (Kurs cs231n)
- Angepasst als Jupyter Notebook von Volodymyr Kuleshov und Isaac Caswell (Kurs cs228)
- Erneute Anpassung von Kevin Zakka (Kurs cs231n)
- Anpassung für die HS Kempten von Michael Strobel im SoSe22
- Nochmalige Anpassung für diesen Kurs

Dieses Tutorial beinhaltet folgende Themen:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Funktionen, Klassen
- IPython: Creating notebooks, Typical workflows



## 2.1 Grundlegende Datentypen: Strings



<https://xkcd.com/353/>

```
In [16]: 1 hello = "HELLO" # String Literale werden von einfachen oder doppelten Anführungszeichen umschlossen
          2 world = 'WORLD'
          3 print(hello, len(hello))
          4
```

HELLO 5

```
In [54]: 1 hw = hello + "\n" + world # String concatenation mit +
          2 print(hw)
          3
```

HELLO  
WORLD

```
In [55]: 1 hello + 42 # Es können nur String-Objekte verbunden werden; hier wäre ein Cast zu String notwendig via str(42)
          2
```

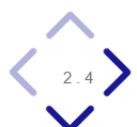
**TypeError**

Traceback (most recent call last)

Cell In[55], line 1

----> 1 hello + 42 # Es können nur String-Objekte verbunden werden (hier wäre ein Cast zu String notwendig via str)

**TypeError:** can only concatenate str (not "int") to str



Verschiedene Möglichkeiten zum Formatieren von Strings, u.a.:

- "printf-style String formatting"
- .format (definiert in [PEP3101](#), seit Python 3.0)
- f-Strings (definiert in [PEP498](#), seit Python 3.6)

```
In [31]: 1 # "printf-style String formatting"  
2 "Hallo %s" % "Welt"  
3
```

```
Out[31]: 'Hallo Welt'
```

```
In [25]: 1 # PEP3101, seit Python 3.0:  
2 # String-Formatierung mit .format  
3 "{0} {1} {2:08d} {3:.3f}".format("Hallo",  
4 27, 28, 3.141592)  
5
```

```
Out[25]: 'Hallo 27 00000028 3.142'
```

```
In [32]: 1 # ... mit mehreren Argumenten  
2 "Hallo %s %d" % ("Welt", 123)  
3
```

```
Out[32]: 'Hallo Welt 123'
```

```
In [26]: 1 # PEP498, seit Python 3.6:  
2 # String-Formatierung mit f-Strings  
3 hw12 = f"{hello} {world:12} {12}"  
4
```

HELLO WORLD 12

## String-Methoden ([Dokumentation](#)) (geben eine Kopie des Strings zurück):

```
In [33]: 1 s = "hallo"
2 print(s.capitalize())          # Anfangsbuchstaben groß schreiben
3 print(s.upper())              # Konvertiert eine Zeichenkette in Großbuchstaben; printet "HELLO"
4 print(s.rjust(7))             # Eine Zeichenkette rechtsbündig ausrichten, mit Leerzeichen auffüllen
5 print(s.center(7))            # Eine Zeichenkette zentrieren, mit Leerzeichen auffüllen
6 print(s.replace("l", "(ell)")) # Ersetze alle vorkommenden Teile einer Teilzeichenkette durch eine andere
7 print(" world ".strip())      # White Space (führend/nachfolgend) wird entfernt
8

Hallo
HALLO
    hallo
    hallo
ha(ell)(ell)o
world
```



## Konvertierung (Type casting)

```
In [119]: 1 print(1.0+2)                      # automatischer Type Cast
2 print(2+True)                       # Konvention False=0, True=1
3 print(int("5"), type(int("5")))     # von String nach Int
4 print(float("5"), type(float("5"))) # von String nach Float
5 print("Hello " + str(5.5))          # von Float nach String
6

3.0
3
5 <class 'int'>
5.0 <class 'float'>
Hello 5.5
```

```
In [120]: 1 print("Hello" + 5)             # keine implizite Konvertierung von Int nach String
2
```

```
-----
TypeError                                     Traceback (most recent call last)
Cell In[120], line 1
----> 1 print("Hello" + 5)                  # keine implizite Konvertierung von Int nach String

TypeError: can only concatenate str (not "int") to str
```

## 2.2 Listen und Slicing

Eine Liste ist das Python-Äquivalent zu einem Array; sie kann verschiedene Typen enthalten und die Größe ändern.



*"If you don't reveal some insights soon, I'm going  
to be forced to slice, dice, and drill!"*

## Grundlegende Funktionalität der Liste:

```
In [80]: 1 L = [3, 1, 2]                                     # Länge der Liste
          2 print(len(L))                                # Negative Indizes zählen vom Ende der Liste
          3 print(L[-1])
          4 print(L[2] == L[len(L) - 1] == L[-1])        # Verschiedene Möglichkeiten, das gleiche Element zu extrahieren
          5
```

```
3
2
True
```

```
In [81]: 1 L[2] = "foo" # Listen können beliebige Typen enthalten
          2 print(L)
          3
```

```
[3, 1, 'foo']
```

```
In [82]: 1 L.append("bar") # Anhängen an die Liste (Entfernen via .pop)
          2 print(L)
          3
```

```
[3, 1, 'foo', 'bar']
```

```
In [83]: 1 L + [97,98,99] # Concatenation von Listen (erzeugt eine Kopie; die Alternative .extend arbeitet "inPlace")
          2
```

```
Out[83]: [3, 1, 'foo', 'bar', 97, 98, 99]
```



## Slicing: Syntax für den Zugriff auf Unterlisten

```
In [85]: 1 nums = list( range(5) ) # range erstellt eine Liste (genauer: einen Generator) von Ganzzahlen
2 print(nums)
3 print(nums[2:4])      # die Teilliste beginnend bei Index 2 (inklusive) und endend bei Index 4 (exklusive)
4
```

```
[0, 1, 2, 3, 4]
[2, 3]
```

```
In [86]: 1 print(nums[2:]) # Slice von Index 2 bis zum Ende
2 print(nums[:2]) # Slice vom Anfang bis zum Index 2 (exklusiv)
3
```

```
[2, 3, 4]
[0, 1]
```

```
In [88]: 1 print(nums[:])      # Holt einen Ausschnitt der gesamten Liste (=Kopie!)
2 print(nums[:-1])      # Slice-Indizes können negativ sein
3 nums[2:4] = [8, 9]    # Zuweisung einer neuen Teilliste zu einem Slice
4 print(nums)
5
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

## Advanced Slicing

```
In [19]: 1 nums = list(range(10))
2 print(nums)
3 print(nums[::2])      # jedes zweite Element
4
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
```

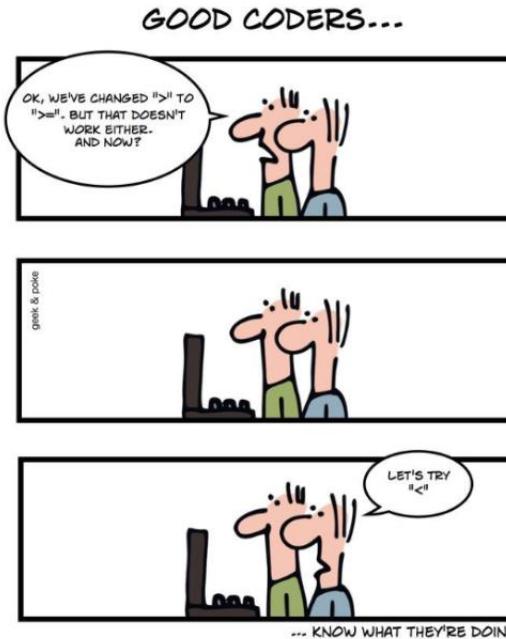
```
In [20]: 1 print(nums[::-1]) # negative Schrittweite: Rückwärts Laufen (d.h. Liste umdrehen)
2
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
In [21]: 1 start = 1
2 stop   = 7
3 step   = 3
4
5 print(nums[start:stop:step]) # Ergebnis?
6
```

```
[1, 4]
```

## 2.3 Kontrollstrukturen und Funktionen



Python Funktionen werden mit dem `def` keyword definiert. Hier gleich mit einem `if`, `elif` und `else` Kontrollfluss:

```
In [337]: 1 def sign(x):
2     if x > 0:
3         return "positive"
4     elif x < 0:
5         return "negative"
6     else:
7         return "zero"
8
9 sign(42)
10
```

Out[337]: 'positive'

## Optionale "keyword" (oder "named") Argumente und Default-Argumente:

```
In [114]: 1 def rufe(name, loud=False):
2     if loud:
3         print(f"HALLO, {name.upper()}!")
4     else:
5         print(f"Hello, {name}!")
6
7 rufe("Bob")
8 rufe("Fred", True)
9
```

```
Hello, Bob!
HALLO, FRED!
```

## Explizite Nennung der Argumente beim Aufruf erhöht die Lesbarkeit:

```
In [115]: 1 rufe("Bob",      loud=True )
2 rufe(name="Bob",  loud=True )
3 rufe(loud=True,   name="Bob")
4
```

```
HALLO, BOB!
HALLO, BOB!
HALLO, BOB!
```

Allerdings müssen "keyword"-Argumente immer nach unbenannten ("positional") Argumenten kommen:

```
In [116]: 1 rufe(name="Bob", True)
2
Cell In[116], line 1
rufe(name="Bob", True)
^
SyntaxError: positional argument follows keyword argument
```

Allerdings müssen "keyword"-Argumente immer nach unbenannten ("positional") Argumenten kommen:

```
In [116]: 1 rufe(name="Bob", True)  
2
```

```
Cell In[116], line 1  
rufe(name="Bob", True)
```

```
SyntaxError: positional argument follows keyword argument
```

```
In [129]:  
  1 # Ausführung der sign-Funktion im Rahmen einer for-Schleife  
  2 for x in [-1, 0, 1]:  
  3     print(sign(x))  
  4
```

```
negative  
zero  
positive
```

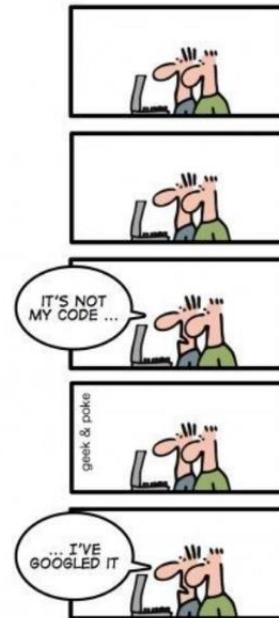
```
In [131]:  
  1 # enumerate zum gleichzeitigen Zugriff auf Index und Wert  
  2 animals = ["cat", "dog", "monkey"]  
  3 for idx, animal in enumerate(animals):  
  4     print(f"# {idx + 1}: {animal}")  
  5
```

```
#1: cat  
#2: dog  
#3: monkey
```

```
In [130]:  
  1 # while-Schleife  
  2 i=1  
  3 while i < 1000:  
  4     i*=2  
  5  
  6     if i == 15: continue  
  7  
  8     if i == 512:  
  9         break  
10  
11 print(i)  
12
```

```
512
```

*CODERS' TOP EXCUSES*



**List comprehensions:** Ein häufiges Einsatzszenario ist, aus einer gegebenen Liste (allgemeiner: "Iterable") eine neue zu erzeugen. Dies ist eine der mächtigsten Funktionen von Python und ersetzt funktionale Konzepte wie `map`, `filter` und `reduce`.

Ein einfaches Beispiel ist der folgende Code zur Berechnung von Quadratzahlen:

```
In [132]: 1 nums = [0, 1, 2, 3, 4]
2 squares = []
3 for x in nums:
4     squares.append(x**2)
5 print(squares)
6
[0, 1, 4, 9, 16]
```

Wir können diesen Code vereinfachen, indem wir eine List Comprehension verwenden:

```
In [133]: 1 nums = [0, 1, 2, 3, 4]
2 squares = [x**2 for x in nums]
3 print(squares)
4
[0, 1, 4, 9, 16]
```

Die List Comprehensions können auch Bedingungen enthalten:

```
In [7]: 1 nums = [0, 1, 2, 3, 4]
2 even_squares = [x**2 for x in nums if x % 2 == 0]
3
4 print(even_squares)
5
```

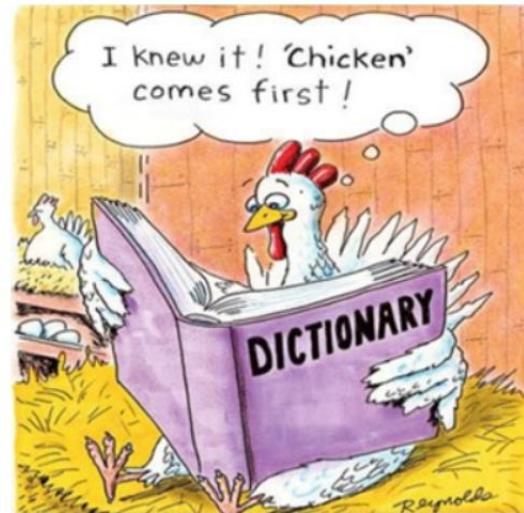
```
[0, 4, 16]
```

In Kombination mit `enumerate`:

```
In [8]: 1 "{0}: {1}".format(i, x**2) for i,x in enumerate(nums) if x % 2 == 0]
2
3 # Ergebnis?
4
```

## 2.4 Weitere Container-Typen

Python enthält mehrere eingebaute Containertypen: Listen, Dictionaries, Sets und Tuples.



**Dictionary:** Ein Dictionary speichert Paare (Key, Value), ähnlich wie eine "Map" in Java oder ein Objekt in Javascript.

```
In [139]: 1 d = {"cat": "cute", "dog": "furry"} # Dict erstellen
2 print(d["cat"])                      # Zugriff
3 print("cat" in d)                     # Key vorhanden?
```

cute  
True

```
In [140]: 1 d["fish"] = "wet"  # Zuweisung
2 print(d["fish"])
3
```

wet

Zugriff auf fehlende Schlüssel:

```
In [141]: 1 # Fehler, weil der Schlüssel fehlt
2 print(d["monkey"])
3
```

```
-----  
-----  
KeyError  
ceback (most recent call last)  
Cell In[141], line 1  
----> 1 print(d["monkey"])

KeyError: 'monkey'
```

```
In [142]: 1 # Zugriff mit Default-Wert
2 print(d.get("monkey", "N/A"))
3 print(d.get("fish", "N/A"))
4
```

N/A  
wet

## Iteration über ein Dictionary:

```
In [143]: 1 for animal, prop in d.items():
2     print(f"The {animal} is {prop}.")
3
The cat is cute.
The dog is furry.
The fish is wet.
```

## Dictionary comprehensions:

```
In [144]: 1 nums = [0, 1, 2, 3, 4]
2 even_num_to_square = {x: x**2 for x in nums if x % 2 == 0}
3 print(even_num_to_square)
4
{0: 0, 2: 4, 4: 16}
```

**Set (Menge)** ist eine ungeordnete Sammlung von unterschiedlichen Elementen. Ein einfaches Beispiel:

```
In [145]: 1 animals = {"cat", "dog"}  
2 print("cat" in animals)  
3 print("fish" in animals)  
4
```

```
True  
False
```

```
In [146]: 1 animals.add("fish")  
2 print("fish" in animals)  
3 print(len(animals))  
4
```

```
True  
3
```

```
In [147]: 1 animals.add("cat")      # Hinzufügen eines bereits vorhandenen Elements ändert nichts  
2 print(len(animals))  
3 animals.remove("cat")  
4 print(len(animals))  
5
```

```
3  
2
```

Iterieren wie zuvor, allerdings ist eine Menge ungeordnet. Sie sollten keine Annahme über die Reihenfolge treffen:

```
In [148]: 1 animals = {"cat", "dog", "fish"}  
          2 for idx, animal in enumerate(animals):  
          3     print(f"# {idx + 1}: {animal}")  
          4  
  
#1: cat  
#2: fish  
#3: dog
```

Set comprehensions:

```
In [150]: 1 print({(x//10)*10 for x in range(30)})  
          2  
  
{0, 10, 20}
```

**Tupel:** Ein Tupel ist eine unveränderbare geordnete Liste von Werten. Ein Tupel ähnelt einer Liste; einer der wichtigsten Unterschiede ist, dass Tupel als Schlüssel in Dictionary und als Elemente von Mengen verwendet werden können (sie sind hashbar), während dies bei Listen nicht möglich ist.

```
In [159]: 1 t = (5, 6)      # Runde Klammern für Tupel  
2 print(t)  
3
```

(5, 6)

```
In [160]: 1 t[0] = 4      # Fehler, weil "immutable"  
2
```

---

-----  
**TypeError** Tra  
ceback (most recent call last)  
Cell In[160], line 1  
----> 1 t[0] = 4 # Liefert einen Fehler,  
weil Tupel unveränderlich sind ("immutable")  
  
**TypeError**: 'tuple' object does not support it  
em assignment

Die Objekte, auf die die Tupel-Elemente verweisen, können sich durchaus ändern:

```
In [167]: 1 p = Person(age=42)
2 t = (p,p)
3 print(t)
4
(Person(42), Person(42))
```

```
In [165]: 1 t[0].age = 43
2 print(t)
3
(Person(43), Person(43))
```

Tupel können als Dictionary Keys verwendet werden (weil sie "immutable" und daher hashable sind):

```
In [166]: 1 # A dictionary with tuple keys
2 d = {(x, x + 1): x for x in range(10)}
3 print(d[(1, 2)])
4
1
```

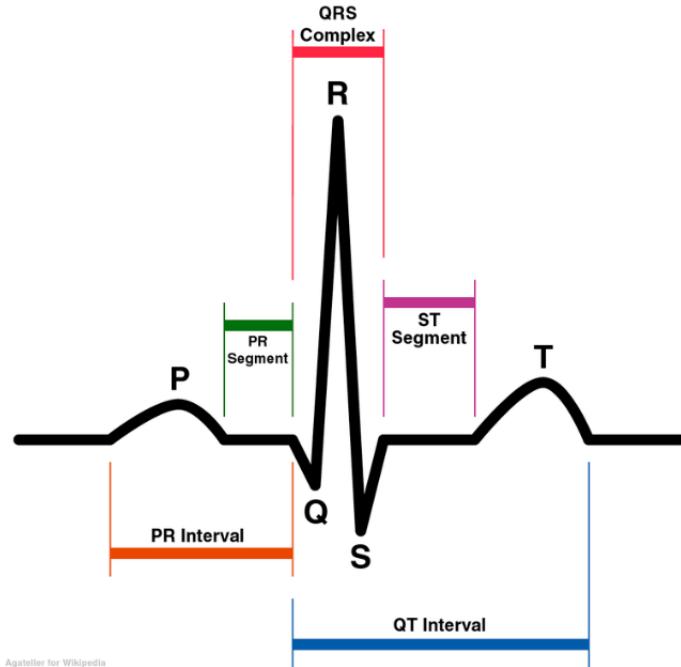


## 1. Grundbegriffe Data Science / Data Analytics

## 2. Python-Grundlagen

## 3. Einfache Visualisierungen

## 3. Visualisierungen



In [205]:

```
1 import pandas as pd # (pandas noch nicht in dieser VL)
2 df = pd.read_csv ( "VL02_Material/heart.csv" )
3
4 df # pandas DataFrame
5
```

Out[205]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG
0	40	M	ATA	140	289	0	Normal
1	49	F	NAP	160	180	0	Normal
2	37	M	ATA	130	283	0	ST
3	48	F	ASY	138	214	0	Normal
4	54	M	NAP	150	195	0	Normal
...	...	...	...	...	...	...	...
913	45	M	TA	110	264	0	Normal
914	68	M	ASY	144	193	1	Normal
915	57	M	ASY	130	131	0	Normal
916	57	F	ATA	130	236	0	LVH
917	38	M	NAP	138	175	0	Normal

918 rows × 12 columns

Basiert auf <https://archive.ics.uci.edu/ml/datasets/Heart+Disease>



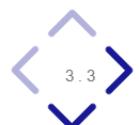
## 3.1 Datenverständnis

```
In [280]: 1 # object bedeutet meistens "String"
2 df.info()
3

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   Age          918 non-null    int64  
 1   Sex          918 non-null    object  
 2   ChestPainType 918 non-null  object  
 3   RestingBP     918 non-null    int64  
 4   Cholesterol   918 non-null    int64  
 5   FastingBS     918 non-null    int64  
 6   RestingECG    918 non-null    object  
 7   MaxHR         918 non-null    int64  
 8   ExerciseAngina 918 non-null  object  
 9   Oldpeak       918 non-null    float64 
 10  ST_Slope       918 non-null    object  
 11  HeartDisease  918 non-null    int64  
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

Einige nicht-offensichtliche Features:

Feature	Beschreibung	Einheit / Wertebereich
ChestPainType	Art der Brustschmerzen	{TA, ATA, NAP, ASY}
RestingBP	Ruheblutdruck	[mm Hg]
RestingECG	Ruhe-EKG	{Normal, ST, LVH}
ExerciseAngina	Angina bei Belastung	{Y, N}
ST_Slope	Steigung im ST Wert	{Up, Flat, Down}
OldPeak	Abweichung im ST Wert bei Belastung	[-10, 10]
HeartDisease	Output	{1, 0}



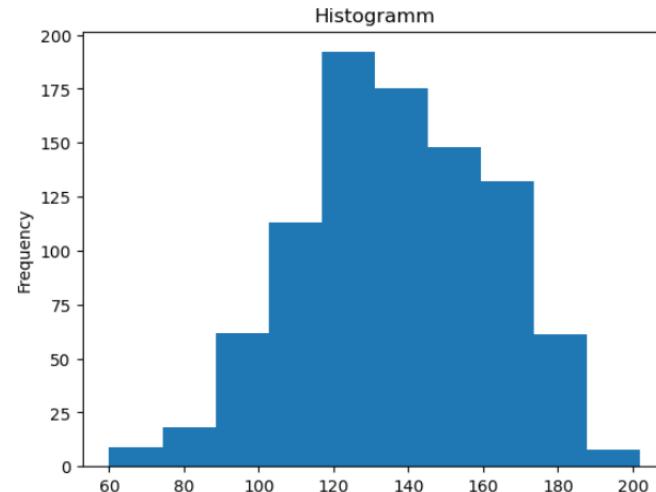
## 3.2 Visualisierung: Wie ist ein numerischer Datenvektor verteilt?

```
In [10]: 1 vec = df["MaxHR"] # eine Spalte eines DataFrames ist eine Pandas Series.  
2 vec  
3 # Rechts sieht man die tatsächlichen Datenwerte, Links sieht man den Index der jeweiligen Zeile.  
# Dieser Index ist standardmäßig eine fortlaufende Nummer, kann aber auch verändert werden.
```

```
Out[10]: 0    172  
1    156  
2    98  
3   108  
4   122  
...  
913   132  
914   141  
915   115  
916   174  
917   173  
Name: MaxHR, Length: 918, dtype: int64
```

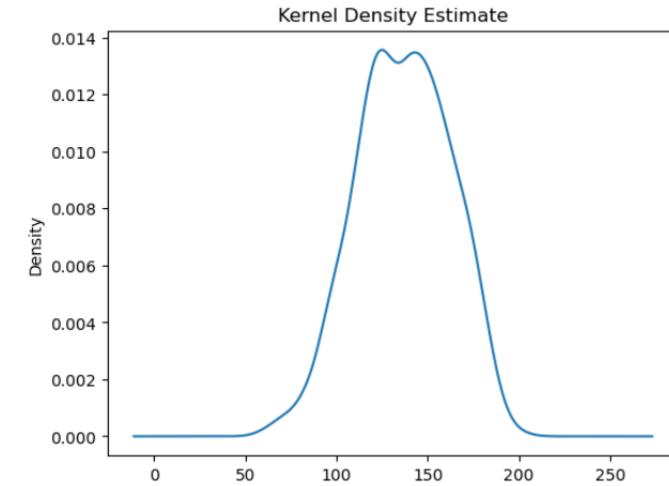
In [13]:

```
1 vec.plot.hist ( title="Histogramm" );
2
```

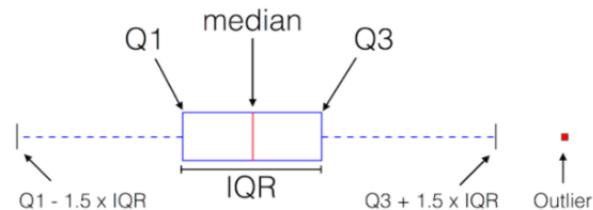


In [228]:

```
1 vec.plot.kde ( title="Kernel Density Estimate" );
2
```



Box plots provide insight into distribution properties of the data. However, they can be challenging to interpret for the unfamiliar reader. The figure below illustrates the different visual features of a box plot.



**Q1:** Quartile 1, or median of the *left* data subset after dividing the original data set into 2 subsets via the median (25% of the data points fall below this threshold)

**Q3:** Quartile 3, median of the *right* data subset (75% of the data points fall below this threshold)

**IQR:** Interquartile-range,  $Q_3 - Q_1$

**Outliers:** Data points are considered to be outliers if  $\text{value} < Q_1 - 1.5 \times \text{IQR}$  or  $\text{value} > Q_3 + 1.5 \times \text{IQR}$

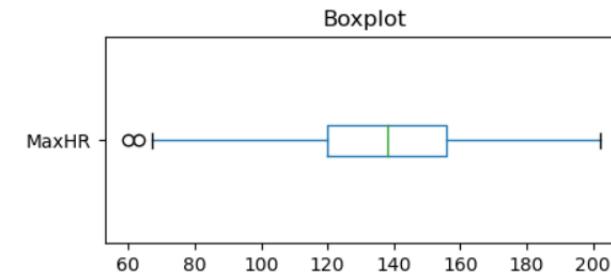


Sebastian Raschka, 2016  
This work is licensed under a Creative Commons Attribution 4.0 International License

The whiskers mark the range of the non-outlier data. The most common definition of non-outlier is  $[Q_1 - 1.5 \times \text{IQR}, Q_3 + 1.5 \times \text{IQR}]$ , which is also the default in this function. Other whisker meanings can be applied via the *whis* parameter.

In [241]:

```
1 vec.plot.box ( title="Boxplot", vert=False,
2                         figsize=(5,2) );
3
```



**Inter Quartile Range (IQR):** Die Größe des zentrierten Bereichs, der 50% der Daten beinhaltet.

Die Box beginnt beim 0,25-Quantil und endet beim 0,75-Quantil, der Strich in der Mitte ist der Median.

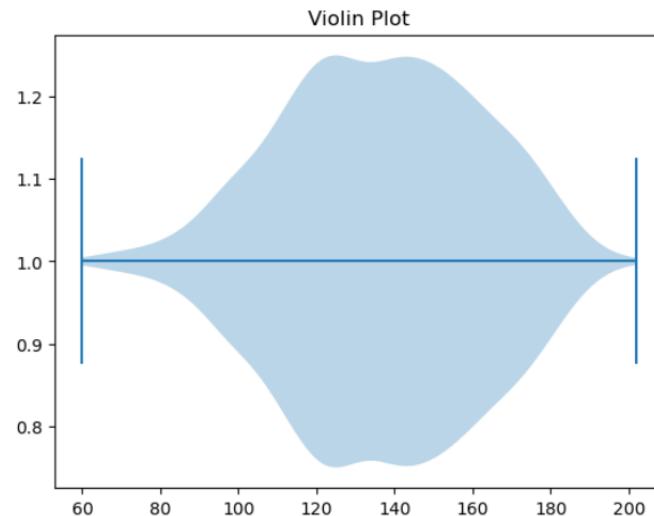
**Achtung:** Die Whisker haben nicht immer die Länge  $1.5 \times \text{IQR}$ , sondern enden am letzten Punkt in diesem Bereich.



Weitere Plot-Funktionalität wird durch Matplotlib und Seaborn angeboten:

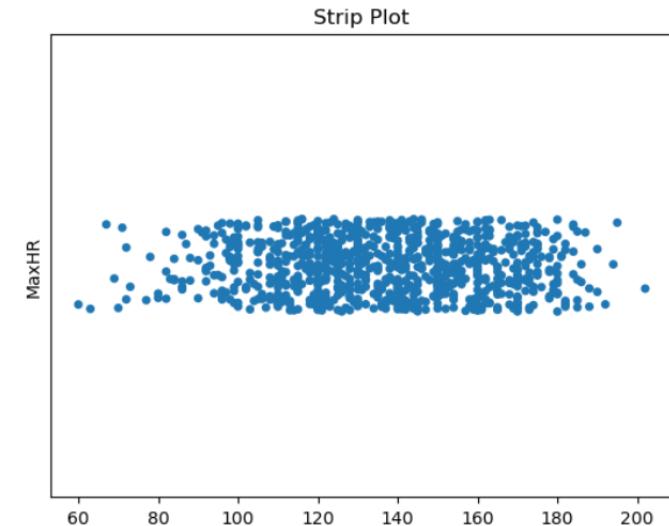
In [265]:

```
1 import matplotlib.pyplot as plt
2 plt.violinplot ( vec, vert=False )
3 plt.title( "Violin Plot" );
4
```



In [332]:

```
1 import seaborn as sns
2 sns.stripplot ( data=vec, orient="h").set(
3     title="Strip Plot", ylabel="MaxHR", yticks=[] );
```



### 3.3 Vergleich verschiedener Subpopulationen

Die Population oder Grundgesamtheit ist die Menge aller Individuen oder Objekte, über die eine Aussage getroffen werden soll.

Göran Kauermann, Helmut Küchenhoff: Stichproben. Methoden und praktische Umsetzung mit R. 1. Auflage. Springer, Berlin Heidelberg 2011, S. 5

In [48]:

```
1 # Untersuche jeden Index, ob hier die Bedingung
2 # "HeartDisease"==0 erfüllt ist
3 idx0 = df["HeartDisease"] == 0
4 idx0
```

Out[48]:

```
0      True
1     False
2      True
3     False
4      True
...
913    False
914    False
915    False
916    False
917    True
Name: HeartDisease, Length: 918, dtype: bool
```

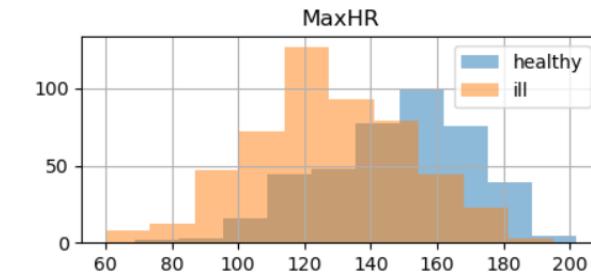
In [49]:

```
1 # Verwende "boolean filtering", um je eine Series
2 # je Teilpopulation zu erhalten (~ = Verneinung)
3
4 vec_healthy = df["MaxHR"][ idx0]
5 vec_ill     = df["MaxHR"][-idx0]
6
7 len(vec_healthy), len(vec_ill)
8
```

Out[49]: (410, 508)

```
In [50]: 1 # Boolean Filtering
2 idx0 = df["HeartDisease"] == 0
3 vec_healthy = df["MaxHR"][ idx0]
4 vec_ill     = df["MaxHR"][-idx0]
```

```
In [279]: 1 # Visualize
2 vec_healthy.hist ( alpha=0.5, label="healthy",
                      figsize=(5,2) );
3 vec_ill.hist ( alpha=0.5, label="ill" );
4 plt.legend ( );
5 plt.title ( "MaxHR" );
```





1. Grundbegriffe Data Science / Data Analytics
2. Python-Grundlagen
3. Einfache Visualisierungen

**Vielen Dank für Ihre Aufmerksamkeit!**