Bibliotheken für Data Science

Wir betrachten im Folgenden:

- **NumPy**: Handling multidimensionaller Arrays
    - Hier: Fokus auf anspruchsvollere Aspekte
- **Pandas**: Werkzeuge für in Spalten organisierte Daten
    - Überblick über die wichtigste Strukturen; Vertiefung in der Übung

Numpy - Numerical Python

1. Numpy Arrays
2. Shapes und Reshaping
3. Broadcasting
4. Advanced Indexing

1. Numpy Arrays

- enthalten nur Elemente desselben Datentyps
- erlauben effiziente Handhabung (Laden, Speichern, Manipulieren)
- Daten werden als Zahlen-Arrays gedacht

Import des Numpy Moduls

In [ ]:
```
import numpy as np

print(np.__version__) # Numpy Versionsnummer
```

Import des Numpy Moduls

In [ ]:
```
import numpy as np

print(np.__version__) # Numpy Versionsnummer
```

Hilfe erhalten

Sie erinnern sich:

```
help(np)
np?
```

Erzeugen von Numpy Arrays

... mithilfe von Python-Listen:

In [ ]:
```
import numpy as np

# integer array:
np.array([1, 4, 2, 5, 3])
```

In [ ]:
```
np.array([3.14, 4, 2, 3]) # Implizites Upcasting: Alle Elemente werden zu floats
```

In [ ]:
```
np.array([1, 2, 3, 4], dtype='float32') # Explizites Casting
```

Eigenschaften von Numpy Arrays

In [ ]:
```
import numpy as np
np.random.seed(0)  # Fixieren des Zufallszahlengenerators (nur für Vorlesungszwecke)
x3 = np.random.randint(10, size=(3, 4, 5))  # 3-dimensionales Array mit Zufallszahlen
```

Eigenschaften von Numpy Arrays

In [ ]:
```
import numpy as np
np.random.seed(0)  # Fixieren des Zufallszahlengenerators (nur für Vorlesungszwecke)
x3 = np.random.randint(10, size=(3, 4, 5))  # 3-dimensionales Array mit Zufallszahlen
```

In [ ]:
```
print("x3 ndim: ", x3.ndim)  # ndim: Anzahl Dimensionen
print("x3 shape:", x3.shape) # shape: Größe jeder Dimension
print("x3 size: ", x3.size)  # size: Totale Größe des Arrays
print("dtype:", x3.dtype)    # dtype: Datentyp
print("itemsize:", x3.itemsize, "bytes") # itemize: Größe eines Elements in Bytes
print("nbytes:", x3.nbytes, "bytes") # nbytes: Gesamtgröße des Arrays in Bytes
```

Zugriff auf Elemente

Zugriff auf Elemente geschieht wie bei Python Listen:

In [ ]:

```
np.random.seed(0)
x1 = np.random.randint(10, size=6)        # eindimensionales Array
print(x1)
print(x1[0], x1[4], x1[-1], x1[-2])       # Zugriff auf einzelne Elemente
```

Array Slicing (Subarrays)

Wir können Subarrays mithilfe der folgenden Slice-Notation erhalten:

```
x[start:stop:step]
```

Wenn `start`, `stop` oder `step` nicht angegeben wird, werden die Standardwerte `start=0`, `stop=Dimensionsgöße`, `step=1` genutzt.

In [ ]:
```
x2 = np.random.randint(10, size=(10,3))  # Matrix mit 10 Zeilen, 3 Spalten
```

In [ ]:
```
print(x2[:, 0]) # erste Spalte von x2
print(x2[0, :]) # erste Zeile von x2
print(x2[0])    # äquivalent zu x2[0, :]
```

Subarrays sind Views, keine Kopien

Vorsicht: Nützlich (weil effizient) aber potentielle Fehlerquelle.

Slices liefern Views und keine Kopien der Array-Daten zurück. Das ist ein wichtiger Unterschied zu Python-Listen!

In [ ]:
```
print(x2)
x2_sub = x2[:2, :2]
print(x2_sub)
```

Subarrays sind Views, keine Kopien

Vorsicht: Nützlich (weil effizient) aber potentielle Fehlerquelle.

Slices liefern Views und keine Kopien der Array-Daten zurück. Das ist ein wichtiger Unterschied zu Python-Listen!

In [ ]:
```
print(x2)
x2_sub = x2[:2, :2]
print(x2_sub)
```

In [ ]:
```
x2_sub[0, 0] = 99 # Modifiziere Subarray
print(x2_sub)
```

In [ ]:
```
print(x2)
```

2. Reshaping von Arrays

Mit der Methode `reshape` lassen sich Arrays in neue Formen bringen. Größe des Originalarrays sowie des neuen Arrays muss dieselbe sein:

In [ ]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

2. Reshaping von Arrays

Mit der Methode `reshape` lassen sich Arrays in neue Formen bringen. Größe des Originalarrays sowie des neuen Arrays muss dieselbe sein:

In [ ]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

Typisch Umwandlung eindimensionales Array in zweidimensionale Spalte oder Zeile (geht auch via `newaxis` Schlüsselwort):

In [ ]:
```
x = np.array([1, 2, 3])  # ! eindimensionales Array, kein (Zeilen-/Spalten-)Vektor !
x.reshape((1, 3)) # Zeilenvektor via reshape
x[np.newaxis, :]  # Zeilenvektor via newaxis
x[:, np.newaxis]  # Spaltenvektor via newaxis
```

3. Broadcasting

- ist eine Menge von Regeln zur Anwendung binärer Funktionen zwischen Arrays (z.B. +, -, /, *, etc)
- erlaubt das Ausführen von binären Funktionen auf Numpy-Arrays unterschiedlicher Größe

3. Broadcasting

- ist eine Menge von Regeln zur Anwendung binärer Funktionen zwischen Arrays (z.B. +, -, /, *, etc)
- erlaubt das Ausführen von binären Funktionen auf Numpy-Arrays unterschiedlicher Größe

Mißverstandenes Broadcasting ist eine häufige Fehlerquelle!

Broadcasting Einführung

Für Numpy-Arrays derselben Größe werden binäre Operationen elementweise durchgeführt:

In [ ]:
```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Broadcasting erlaubt Ausführen binärer Operationen auf Numpy-Arrays **verschiedener** Größe.

Beispiel: Addiere Skalar (= 0-dimensionales Array) zu einem eindimensionalen Array:

In [ ]:
```
a + 5
```

Broadcasting erlaubt Ausführen binärer Operationen auf Numpy-Arrays **verschiedener** Größe.

Beispiel: Addiere Skalar (= 0-dimensionales Array) zu einem eindimensionalen Array:

In [ ]:
```
a + 5
```

Denkmodell

Operation vervielfältigt den Integer `5` in ein Array `[5, 5, 5]`, bevor Operation durchgeführt wird. (*)

(*) Intern wird natürlich nichts vervielfältigt. Aber dieses Denkmodell hilft beim Verständnis von Broadcasting.

Dies geht auch mit mehreren Dimensionen:

In [ ]:

```
a = np.array([0, 1, 2])  # 1-dimensionales Array
M = np.ones((3, 3))      # 3x3 Matrix voller Einsen
print('M: ', M, '--- a:', a, '\n')
print('M + a:\n', M + a)
```

Dies geht auch mit mehreren Dimensionen:

In [ ]:
```
a = np.array([0, 1, 2])  # 1-dimensionales Array
M = np.ones((3, 3))      # 3x3 Matrix voller Einsen
print('M: ', M, '--- a:', a, '\n')
print('M + a:\n', M + a)
```

Hier wird das eindimensionale Array `a` erweitert (gebroadcastet) in der zweiten Dimension, um dieselbe Form (shape) wie `M` zu erhalten.

Komplizierterer Fall: Beide Arrays müssen gebroadcastet werden.

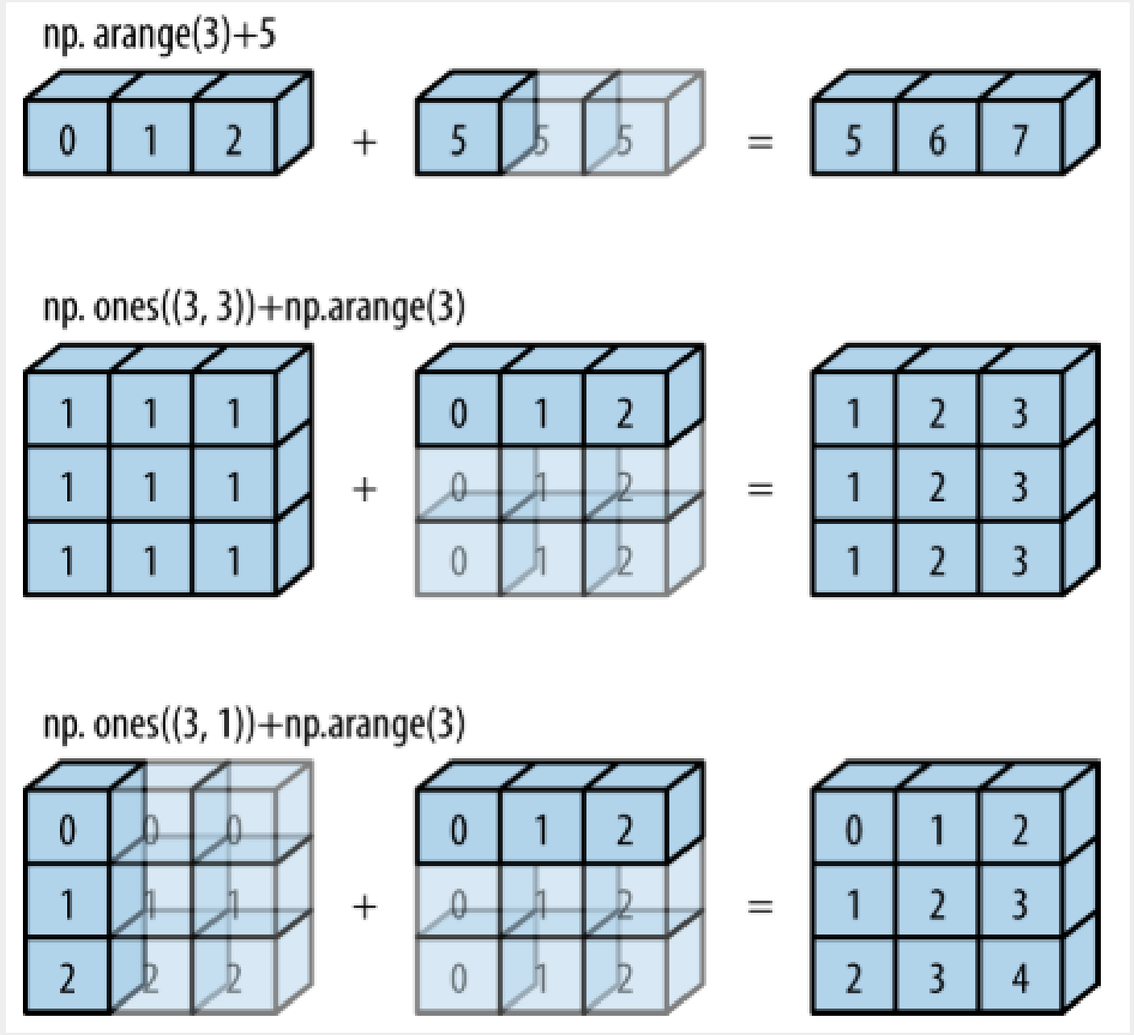
Beispiel:

In [ ]:

```
a = np.arange(3)                    # 1-dimensionales Array: [0, 1, 2]
b = np.arange(3)[:, np.newaxis]  # 2-dimensionales Array

print(a, ' shape:', a.shape, '\n')
print(b, ' shape:', b.shape, '\n')
print(a+b)
```

Visualisierung der vorherigen Beispiele



Broadcasting Regeln (wichtig)

Broadcasting folgt drei strikten Regeln:

1. Wenn zwei Arrays sich in der Anzahl ihrer Dimensionen unterscheiden, wird das Array mit *weniger* Dimensionen mit zusätzlichen Dimensionen auf der führenden linken Seite aufgefüllt (*padding*).
2. Wenn die beiden Arrays in einer Dimension nicht übereinstimmen, wird das Array, dessen Dimension 1 ist, erweitert, bis es die Dimension des anderen Arrays erreicht hat.
3. Wenn die beiden Arrays in mindestens einer Dimension nicht übereinstimmen und keine dieser beiden Dimensionen 1 ist, dann wird ein Fehler geworfen.

Beispiel 1

In [ ]:
```
M = np.ones((2, 3)) # shape: (2, 3)
a = np.arange(3)    # shape:(3,)
```

Beispiel 1

In [ ]:
```
M = np.ones((2, 3)) # shape: (2, 3)
a = np.arange(3)    # shape:(3,)
```

Regel 1: `a` hat weniger Dimensionen, also wird links mit Einsen aufgefüllt:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

Beispiel 1

In [ ]:
```
M = np.ones((2, 3)) # shape: (2, 3)
a = np.arange(3)    # shape:(3,)
```

Regel 1: `a` hat weniger Dimensionen, also wird links mit Einsen aufgefüllt:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

Regel 2: Erste Dimension stimmt nicht überein, also wird die Dimension bis zur Übereinstimmung erweitert:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

Beispiel 2

In [ ]:
```
a = np.arange(3).reshape((3, 1)) # shape: (3, 1)
b = np.arange(3)                 # shape: (3,)
```

Beispiel 2

In [ ]:
```
a = np.arange(3).reshape((3, 1)) # shape: (3, 1)
b = np.arange(3)                 # shape: (3,)
```

Regel 1: Auffüllen der Dimensionen von `b` mit Einsen von links:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

Beispiel 2

In [ ]:
```
a = np.arange(3).reshape((3, 1)) # shape: (3, 1)
b = np.arange(3)                 # shape: (3,)
```

Regel 1: Auffüllen der Dimensionen von `b` mit Einsen von links:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

Regel 2: Erweitern der 1-er Dimensionen, so dass die Dimensionen gleich groß werden.

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Beispiel 3

In [ ]:
```
M = np.ones((3, 2)) # shape: (3, 2)
a = np.arange(3)    # shape: (3,)
```

Beispiel 3

In [ ]:
```
M = np.ones((3, 2)) # shape: (3, 2)
a = np.arange(3)    # shape: (3,)
```

Regel 1: Padding von `a` auf der linken Seite:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

Beispiel 3

In [ ]:
```
M = np.ones((3, 2)) # shape: (3, 2)
a = np.arange(3)    # shape: (3,)
```

Regel 1: Padding von `a` auf der linken Seite:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

Regel 2: Erste Dimension von `a` erweitern:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Beispiel 3

In [ ]:
```
M = np.ones((3, 2)) # shape: (3, 2)
a = np.arange(3)    # shape: (3,)
```

Regel 1: Padding von `a` auf der linken Seite:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

Regel 2: Erste Dimension von `a` erweitern:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Regel 3: Wenn die Dimensionen nicht übereinstimmen und keine Dimension 1 vorhanden ist, werfe einen Fehler. Die Arrays sind nicht kompatibel.

Beispiel aus der Praxis

Zentrieren von Datenreihen.

= Mittelwerte von Datenreihen werden auf Null transformiert.

In [ ]:

```
# 3 Datenreihen (Spalten) mit je 10 Datenpunkte (Zeilen)
X = np.random.random((10, 3))

# Berechnung der Mittelwerte
Xmean = np.mean(X, axis=0)
print('Xmean: ', Xmean)

# Zentrieren
X_centered = X - Xmean
print('X_centered.shape: ', X_centered.shape)
print(np.mean(X_centered, axis=0))
```

4. Advanced Indexing

Betrachten Sie

```
a = np.array([1, 2, 3])
a[indices]
```

Defini on

- `indices` ist Tupel ohne Array-Elemente: **Basic Indexing** (kennen Sie bereits)

4. Advanced Indexing

Betrachten Sie

```
a = np.array([1, 2, 3])
a[indices]
```

Defini on

- `indices` ist Tupel ohne Array-Elemente: **Basic Indexing** (kennen Sie bereits)
- `indices` ist kein Tupel oder Tupel mit Array-Elementen: **Advanced Indexing**

Basic Indexing erzeugt Views. Advanced Indexing erzeugt Kopien.

Advanced Indexing gibt es in zwei Formen: Boolsche Ausdrücke und Integer Arrays

1. Boolsche Ausdrücken

In [ ]:
```
x = np.arange(1, 6)
x[x<4] # x<4 erzeugt ein Boolsches Array
```

2. Integer Arrays

In [ ]:
```
np.random.seed(42)
x = np.random.randint(100, size=10)
print(x)
# Wir wollen Elemente 3, 7 und 2:
[x[3], x[7], x[2]]
```

Alternativ können wir diese Elemente auch über Advanced Indexing anfordern:

In [ ]:
```
ind = [3, 7, 4]
x[ind]
```

Wich `g`

- Shape des Ergebnisarrays wird bestimmt durch Index-Arrays (`ind`), nicht durch `x`!

In [ ]:
```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

Advanced Indexing funktioniert auch bei mehreren Dimensionen:

In [ ]:
```
X = np.arange(12).reshape((3, 4))
print(X)
```

In [ ]:
```
zeilen = np.array([0, 1, 2])
spalten = np.array([2, 1, 3])
X[zeilen, spalten]
```

Advanced Indexing funktioniert auch bei mehreren Dimensionen:

In [ ]:
```
X = np.arange(12).reshape((3, 4))
print(X)
```

In [ ]:
```
zeilen = np.array([0, 1, 2])
spalten = np.array([2, 1, 3])
X[zeilen, spalten]
```

Erster Wert entspricht `X[0, 2]`, zweiter ist `X[1, 1]` und dritter: `X[2, 3]`.

Auch hier gelten die Broadcasting-Regeln, wenn die Shapes von `zeilen` und `spalten` nicht überein stimmen.

Beispiel:

In [ ]:
```
print(zeilen[:, np.newaxis].shape)
print(spalten.shape)
```

In [ ]:
```
X[zeilen[:, np.newaxis], spalten] # Welchen Shape hat das zurückgegebene Array?
```

Auch hier gelten die Broadcasting-Regeln, wenn die Shapes von `zeilen` und `spalten` nicht überein stimmen.

Beispiel:

In [ ]:
```
print(zeilen[:, np.newaxis].shape)
print(spalten.shape)
```

In [ ]:
```
X[zeilen[:, np.newaxis], spalten] # Welchen Shape hat das zurückgegebene Array?
```

Merkregel Bei Advanced Indexing bestimmt der gebroadcastete Shape der Indizes den Shape des zurückgegebenen Arrays.

Bibliotheken für Data Science

Wir betrachten im Folgenden:

- **NumPy**: Handling multidimensionaller Arrays
    - Hier: Fokus auf anspruchsvollere Aspekte
- **Pandas**: Werkzeuge für in Spalten organisierte Daten
    - Überblick über die wichtigste Strukturen; Vertiefung in der Übung

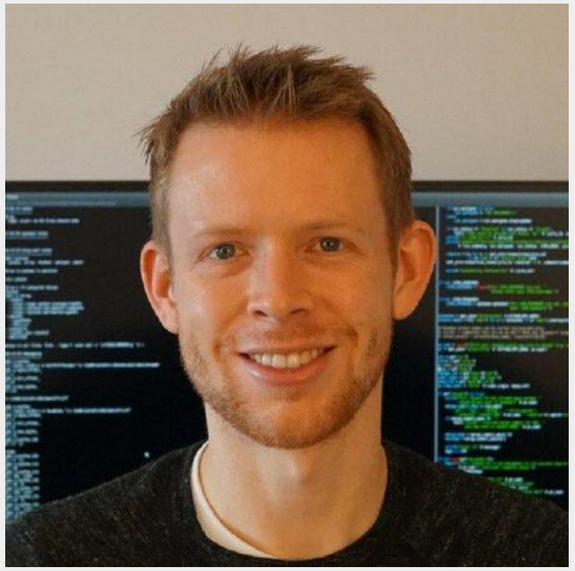
Pandas

1. Einführung
2. Pandas Objekte
3. Indexing und Datenauswahl

1 Einführung

Pandas

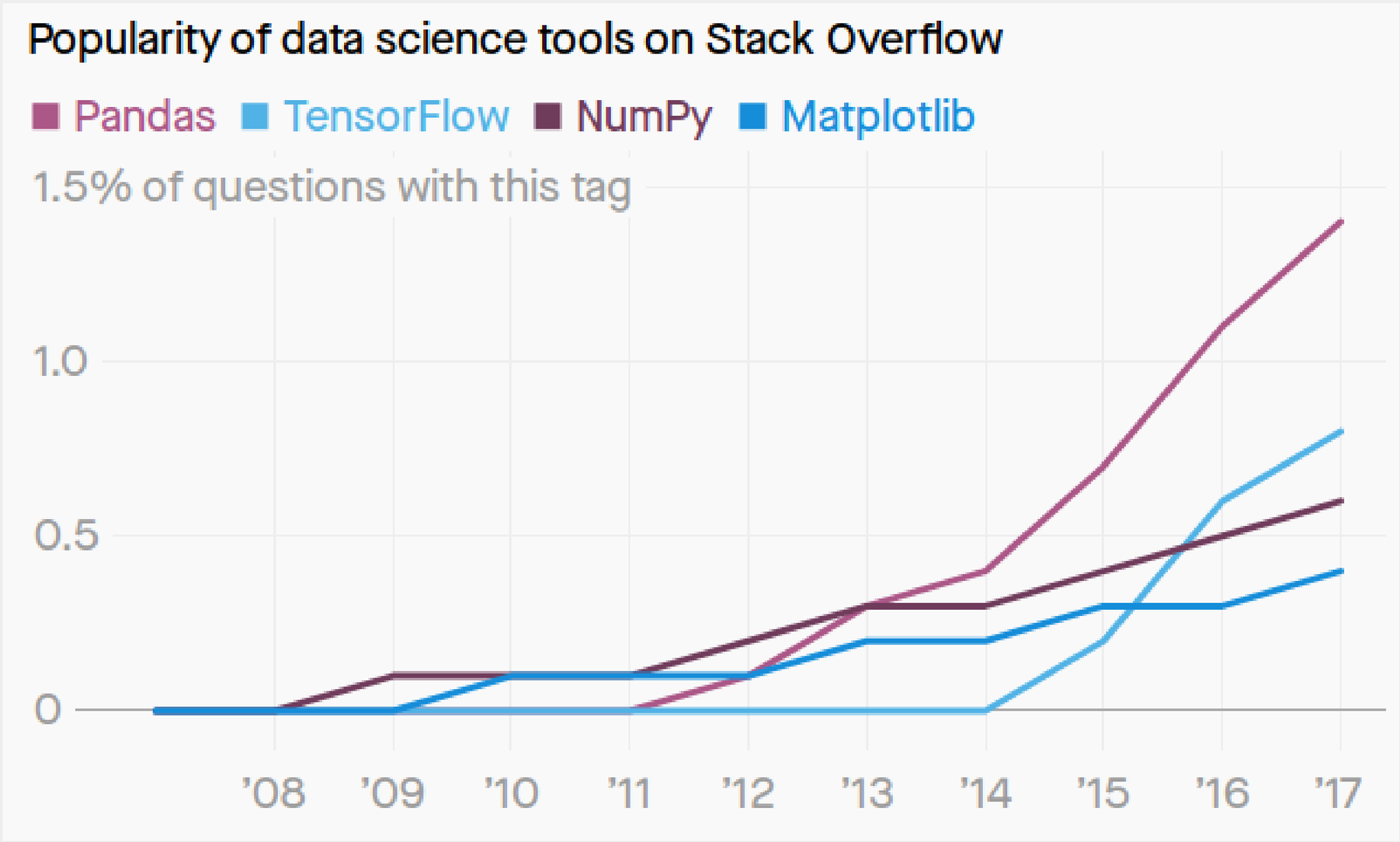
- Python Bibliothek für Datenanalyse
- Pandas - Abkürzung für PAnel DAta. Ökonometrischer Begriff für multidimensional strukturierte Datensätze.
- entwickelt von Wes McKinney seit dem Jahr 2008; Open Source seit Ende 2009



Wes McKinney

McKinney wollte ursprünglich mit Pandas eigene Probleme lösen:

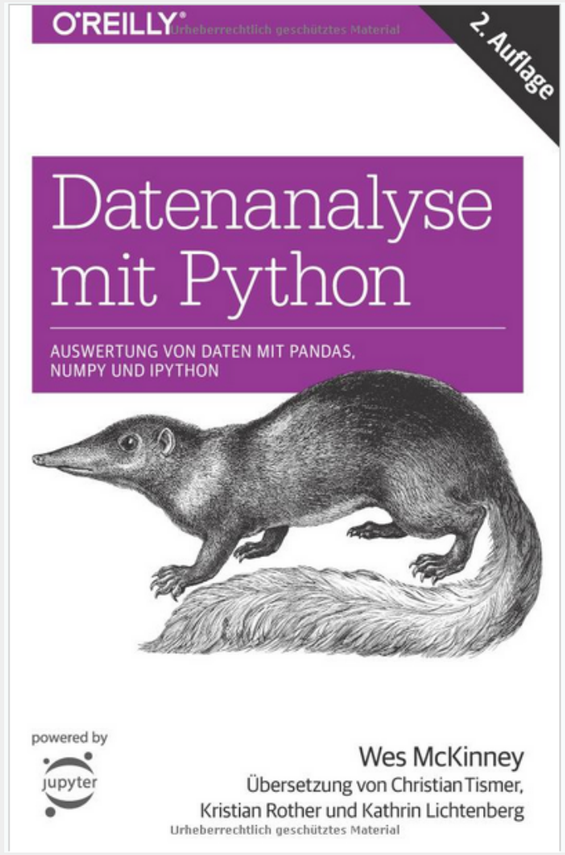
- leichterer Umgang mit fehlenden Daten
- bessere Werkzeuge zur Analyse von Zeitreihen
- gleiche Datenstrukturen zur Verarbeitung von Zeitreihen und Nicht-Zeitreihen
- Datenstrukturen mit benannten Achsen und automatischem Mitführen von Meta-Daten
- vereinfachtes Mergen und Splitten von Daten



Quelle: Quartz Magazine | Stack Overflow Oktober 2018, https://qz.com/1408660/

Buch "Datenanalyse mit Python"

- für anwendungsorientierte Leser
- ausführliche Einführung in die Python Pandas Bibliothek
- Autor ist Hauptentwickler der Pandas Bibliothek
- für diesen Kurs elektronisch angeschafft (schauen Sie bei ILIAS nach)



2 Pandas Objekte

drei fundamentale Datenstrukturen:

1. Series
2. DataFrame
3. Index

2 Pandas Objekte

drei fundamentale Datenstrukturen:

1. Series
2. DataFrame
3. Index

In [ ]:
```
# Standardimporte zum Arbeiten mit Pandas
import numpy as np
import pandas as pd
```

Pandas Series Objekt

- Pandas `Series` ist ein eindimensionales Datenarray mit Index
- `Series` enthält Werte und einen Index.
- Zugriff per `.values` bzw `.index`.

In [ ]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print((data.values))
type(data.values)
```

In [ ]:
```
print((data.index)) # vom Typ pd.Index
```

In [ ]:
```
# Zugriff wie über ein NumPy Array über den entsprechenden Index:
print((data[1]))
print((data[1:3]))
```

`Series` als generalisiertes Numpy Array

- `Series` und 1-dimensionales Numpy Array verhalten sich ähnlich

Wichtigster Unterschied:

- Numpy Array enthält implizit definierten Integer Index
- `Series` enthält expliziten Index

`Series` hat daher zusätzliche Fähigkeiten.

In [ ]:
```
# Beispiel 1: Characters als Index (anstatt Integer Werten)
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

In [ ]:
```
# Zugriff
data['b']
```

In [ ]:
```
# Beispiel 1: Characters als Index (anstatt Integer Werten)
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

In [ ]:
```
# Zugriff
data['b']
```

In [ ]:
```
# Beispiel 2: Nichtkontinuierliche Indexwerte
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

In [ ]:
```
data[5]
```

`Series` als spezialisiertes Dictionary

- `Dictionary` bildet beliebige Schlüssel (keys) auf Werte (values) ab
- `Series` bildet typisierte Schlüssel auf typisierte Werte ab

Dies macht `Series` effizienter als klassische Python Dictionaries.

In [ ]:

```
# Python Dictionary
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127, 'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict) # Pandas Series Objekt
population # Index erzeugt über sortierte Schlüssel !
```

Zugriff auf Elemente

In [ ]:
```
# Zugriff auf Elemente ähnlich wie bei Dictionary:
population['California']
```

Zugriff auf Elemente

In [ ]:
```
# Zugriff auf Elemente ähnlich wie bei Dictionary:
population['California']
```

In [ ]:
```
# ABER: Series erlaubt Slicing (Dictionary nicht!)
population['California':'New York']
```

Zugriff auf Elemente

In [ ]:
```
# Zugriff auf Elemente ähnlich wie bei Dictionary:
population['California']
```

In [ ]:
```
# ABER: Series erlaubt Slicing (Dictionary nicht!)
population['California':'New York']
```

Beachten Sie, dass der letzte Index im Slice mit enthalten ist (anders als bei Numpy!)

Erzeugen von `Series` Objekten

Eine Möglichkeit:

```
>>> pd.Series(data, index=index)
```

mit optionalem Argument `index`.

`data` kann z.B. Python Liste oder Numpy Array sein.

In [ ]:
```
# Ist data eine Python Liste oder Numpy Array,
# dann wird Index implizit als Integer Sequenz angenommen.
pd.Series([2, 4, 6])
```

Erzeugen von `Series` Objekten

Eine Möglichkeit:

```
>>> pd.Series(data, index=index)
```

mit optionalem Argument `index`.

`data` kann z.B. Python Liste oder Numpy Array sein.

In [ ]:
```
# Ist data eine Python Liste oder Numpy Array,
# dann wird Index implizit als Integer Sequenz angenommen.
pd.Series([2, 4, 6])
```

In [ ]:
```
# Ist data ein Skalar, wird er einfach gemäß des Indexes vervielfältigt:
pd.Series(5, index=[100, 200, 300])
```

In [ ]:
```
# Wenn data ein Dictionary ist, wird aus den Keys ein Index erzeugt:
pd.Series({2:'a', 1:'b', 3:'c'})
```

In [ ]:
```
# Wenn data ein Dictionary ist, wird aus den Keys ein Index erzeugt:
pd.Series({2:'a', 1:'b', 3:'c'})
```

In [ ]:
```
# Alternativ kann der Index explizit auch definiert werden:
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

In diesem Fall wird `Series` nur mit den explizit definierten Schlüsseln initialisiert.

Pandas DataFrame Objekt

- `DataFrame` Objekt ist neben `Series` das zweite fundamentale Pandas Objekt.
- kann als Generalisierung eines Numpy Arrays oder eines Dictionaries gedacht werden.

`DataFrame` als generalisiertes Numpy Array

`DataFrame` kann interpretiert werden

- als Analogon eines zwei-dimensionalen Numpy Arrays mit flexiblen Zeilen- und Spaltenindizes
- als Sequenz von `Series` Objekte mit demselbem Index

In [ ]:
```
# Beispiel
# Erzeuge ein Series Objekt
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995} # in Quadratkilometern
area = pd.Series(area_dict)
# Erzeuge DataFrame Objekt aus Dictionary mit zwei Series Objekten
states = pd.DataFrame({'population': population, 'area': area})
states
```

In [ ]:
```
# DataFrame besitzt (wie Series) ein Index-Attribut:
print((states.index))
```

In [ ]:
```
# DataFrame besitzt (wie Series) ein Index-Attribut:
print((states.index))
```

In [ ]:
```
# Zusätzlich besitzt DataFrame ein columns (Spalten) Attribut,
# welches einen Index der Spalten enthält:
states.columns
```

- `DataFrame` sieht wie eine Generalisierung eines zweidimensionlen Numpy Arrays aus, nur bietet es zusätzlich Zeilen- und Spaltenindizes, um auf Daten zuzugreifen.

DataFrame als spezialisiertes Dictionary

`DataFrame` kann interpretiert werden

- als Dictionary: `DataFrame` bildet Schlüssel (=Spaltenname) auf ein `Series` Objekt ab.

DataFrame als spezialisiertes Dictionary

`DataFrame` kann interpretiert werden

- als Dictionary: `DataFrame` bildet Schlüssel (=Spaltenname) auf ein `Series` Objekt ab.

In [ ]:
```
# Beispiel
# Anfordern des "Area" Attributes liefert Series Objekt zurück:
states['area']
```

DataFrame als spezialisiertes Dictionary

`DataFrame` kann interpretiert werden

- als Dictionary: `DataFrame` bildet Schlüssel (=Spaltenname) auf ein `Series` Objekt ab.

In [ ]:
```
# Beispiel
# Anfordern des "Area" Attributes liefert Series Objekt zurück:
states['area']
```

Vorsicht:

```
data[0] # -> liefert erste *Zeile*
data['col0'] # -> liefert erste *Spalte*
```

Erzeugen von DataFrame Objekten

In [ ]:
```
# Beispiele
# DataFrame als Sammlung von Series Objekten
pd.DataFrame(population, columns=['population'])
```

In [ ]:
```
# DataFrame aus einer Liste von Dictionaries
data = [{'a': i, 'b': 2 * i} for i in range(3)] # Dictionary Comprehension
pd.DataFrame(data)
```

In [ ]:
```
# Not-a-Number (NaN) autocompletion
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

In [ ]:
```
# DataFrame aus einem Dictionary von Series Objekten
pd.DataFrame({'population': population,
              'area': area})
```

DataFrame aus einem zweidimensionalen Numpy Array

In [ ]:
```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

- Wenn kein Index angegeben wird, wird implizit ein Integer Index angelegt.

Das Pandas Index Objekt

- `Series` und `DataFrame` enthalten Indizes -> Index Objekte.
- Index-Objekt kann als immutable Array oder geordnete Multimenge interpretiert werden.

Index-Objekt als immutable Array

In [ ]:
```
# Beispiel
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

In [ ]:
```
print((ind[1]))   # Wir können Indexelemente anfordern...
print((ind[::2])) # ... und Slices
```

Index-Objekt als immutable Array

In [ ]:
```
# Beispiel
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

In [ ]:
```
print((ind[1]))   # Wir können Indexelemente anfordern...
print((ind[::2])) # ... und Slices
```

In [ ]:
```
# Index Objekte haben einige von Numpy uns bekannte Attribute:
print((ind.size, ind.shape, ind.ndim, ind.dtype))
```

In [ ]:
```
# Unterschied zu Numpy Arrays: Index-Objekte sind immutable.
ind[1] = 0
```

Indexing und Datenauswahl

Von Numpy kennen wir:

1. Indexing (`array[2, 1]`)
2. Slicing (`array[:, 1:5]`)
3. Advanced Indexing: Masking (`array[array > 0]`), Integer Indexing `array[:, [1, 5]]`

Pandas `Series` und `DataFrame` Objekte unterstützen ähnliche Operationen.

Datenauswahl mit `Series`

Indexing für `Series` analog zu:

1. eindimensionalen Numpy Array
2. Python Dictionary

Index-Attribute loc und iloc

Slicing und Indexing können häufige Fehlerquelle sein.

Beispiel: Expliziter Index besteht aus Integern.

```
    data[1]   # Nutzt expliziten Index
    data[1:3] # Nutzt impliziten Integer Index
```

Index-Attribute loc und iloc

Slicing und Indexing können häufige Fehlerquelle sein.

Beispiel: Expliziter Index besteht aus Integern.

```
    data[1]   # Nutzt expliziten Index
    data[1:3] # Nutzt impliziten Integer Index
```

In [ ]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

In [ ]:
```
# Nutzung des expliziter Index beim normalen Indexing
print((data[1], '\n'))

# Nutzung des impliziten Index beim Slicing
print((data[1:3]))
```

Zur Verminderung dieser Fehlerquelle:

Explizites Spezifizieren der Indizes durch

1. `loc`-Attribut: Nutzung des expliziten Index
2. `iloc`-Attribut: Nutzung des impliziten Index

In [ ]:
```
data.loc[1] # indexing mit explizitem Index
```

In [ ]:
```
data.loc[1:3] # Indexing mit explizitem Index
```

Zur Verminderung dieser Fehlerquelle:

Explizites Spezifizieren der Indizes durch

1. `loc`-Attribut: Nutzung des expliziten Index
2. `iloc`-Attribut: Nutzung des impliziten Index

In [ ]:
```
data.loc[1] # indexing mit explizitem Index
```

In [ ]:
```
data.loc[1:3] # Indexing mit explizitem Index
```

In [ ]:
```
data.iloc[1] # Indexing mit implizitem Index
```

In [ ]:
```
data.iloc[1:3] # Slicing mit implizitem Index
```

Datenauswahl mit DataFrame

Indexing mit DataFrame analog zu

1. Python Dictionary von `Series` Objekten, die denselben Index haben
2. zweidimensionalem Array

DataFrame als Dictionary

Beispiele:

In [ ]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

In [ ]:
```
# Individuelle Spalten auswählbar via Dictionary-artigen Indexing mit Spaltenname:
data['area']
```

Spalten sind auch per Attribut-Zugriff auswählbar, sofern Spaltenname ein String ist, der nicht Methodennamen von DataFrame übereinstimmt:

In [ ]:
```
data.area
```

Spalten sind auch per Attribut-Zugriff auswählbar, sofern Spaltenname ein String ist, der nicht Methodennamen von DataFrame übereinstimmt:

In [ ]:
```
data.area
```

Hinzufügen von Spalten:

In [ ]:
```
# Hinzufügen einer weiteren Spalten namens 'density':
data['density'] = data['pop'] / data['area']
data
```

DataFrame als zweidimensionales Array

- `.values` Attribut erlaubt Zugriff auf 2-d Datenarray:

In [ ]:
```
data.values # liefert Numpy Array zurück
```

DataFrame als zweidimensionales Array

- `.values` Attribut erlaubt Zugriff auf 2-d Datenarray:

In [ ]:
```
data.values # liefert Numpy Array zurück
```

Typische Array-artige Operationen sind erlaubt:

In [ ]:
```
data.T # Transponieren der Daten: Vertauschen von Zeilen und Spalten
```

Vorsicht beim Indexing:

In [ ]:
```
# Operieren auf Numpy Array (via values Attribut)
data.values[0] # liefert Zeile 0 zurück (wie bei Numpy!)
```

In [ ]:
```
# Operieren auf DataFrame Objekt
data['area'] # liefert Spalte "Area" zurück
```

Nutzung von `iloc` und `loc`

In [ ]:
```
data.iloc[:3, :2] # Slicing mit impliziten (Integer-) Indizes
```

In [ ]:
```
data.loc[:'Illinois', :'pop'] # Slicing mit expliziten Indizes
```

Zusätzliche Index-Konventionen

1. Indexing bezieht sich auf Spalten
2. Slicing bezieht sich auf Zeilen.
3. Masking bezieht sich auf Zeilen

Beispiele:

In [ ]:
```
data['area']                # Indexing bezieht sich auf Spalten
```

Zusätzliche Index-Konventionen

1. Indexing bezieht sich auf Spalten
2. Slicing bezieht sich auf Zeilen.
3. Masking bezieht sich auf Zeilen

Beispiele:

In [ ]:
```
data['area']                # Indexing bezieht sich auf Spalten
```

In [ ]:
```
data['Florida':'Illinois'] # Slicing auf Zeilen; alternativ: data[1:3]
```

Zusätzliche Index-Konventionen

1. Indexing bezieht sich auf Spalten
2. Slicing bezieht sich auf Zeilen.
3. Masking bezieht sich auf Zeilen

Beispiele:

In [ ]:
```
data['area']                # Indexing bezieht sich auf Spalten
```

In [ ]:
```
data['Florida':'Illinois'] # Slicing auf Zeilen; alternativ: data[1:3]
```

In [ ]:
```
data[data.density > 100]   # Masking auf Zeilen
```