ZF Informatik II D-MAVT

Julian Lotzer – jlotzer@student.ethz.ch
Daniel Steinhauser – dsteinhauser@student.ethz.ch
Version: 06.08.2023

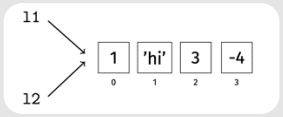
Diese Zusammenfassung basiert auf den D-MAVT Informatik II Vorlesungen von Dr. Ralf Sasse und Dr. Carlos Cotrini. Die ZF wird laufend geupdated, da die Vorlesung zum ersten Mal in dieser Form stattfindet. Es kann keine Gewähr auf Richtigkeit oder Vollständigkeit gegeben werden. Verbesserungsvorschläge sind immer erwünscht (Mail)! Die neueste geupdatete Version findet ihr auf https://n.ethz.ch/~jlotzer/ oder https://n.ethz.ch/~dsteinhauser/

1 General Python

1.1 Reference Semantics and Aliasing

Alles ist ein Pointer:

```
l1 = [1,'hi', 3, -4] #l1 -> [1] ['hi'] [3] [-4]
l2 = l1
```



Wenn eine Kopie benötigt wird:

```
import copy
l2 = copy.copy(l2) #shallow copy (1D-Array)
l2 = copy.deepcopy(l2) #deep copy (Multi Dimensional Arrays)
```

1.2 Datentypen

Python typisiert Variabeln dynamisch, der Variabeltyp kann sich also im Laufe des Programmes auch ändern:

```
s = 5
print(type(s)) #output: <class 'int'>
s = False
print(type(s)) #output: <class 'bool'>
s = "Hello World"
print(type(s)) #output: <class 'str'>
```

- Um den Typ zu konvertieren:

```
s = 5.9 #type(s) = <class 'float'>
x = int(s) #type(x) = <class 'int'>
y = str(s) #type(y) = <class 'str'>
```

1.2.1 Type Hints

- Are not enforced by python, but help make code more legible:

```
def add_integers(li: list[int]) -> int
    …
#the type annotation after the colon (:) indicates the
#expected type. The type annotation after the arrow
#indicates the expected return type
```

1.3 Input and Output

- Output

```
print("Hello World") #output: Hello World
print("Hello", "World") #output: Hello World
print("Hello", "World", sep = "--")
#output: Hello--World
print("Hello", "World", sep = "--", end = "!")
#output: Hello--World!
```

- Input

```
name = input("Enter name: ") #input returns a string
print("Hello", name)
```

- Input eines Integers (str->int konvertierung):

```
number = int(input("Enter your number: "))
print("Number:", number)
```

1.4 Blöcke (if/else, while, for)

- if, elif (else if), else Block

```
x = int(input("Enter a number: "))
if x < 5 and x >= 0:
  print("too small") #if x between 0 and 5
elif x == 69 or x == 420:
  print("nice") #if x is equal to 69 or 420
elif x < 0:
  pass #do nothing if x negative
else:
  print("big number") #x is positive and the
ifs/elifs conditions don't hold
```

- while-Schleife

```
x = 0
while x <= 3:
  print(x)
  x += 1
```

- for-Schleife über Wertebereiche (siehe Ranges Kap 2.2)

```
for i in range(0,4,1): #range(start, stop, step)
  print(i, end = " ") #output: 0 1 2 3

for i in reversed(range(0,4,1)):
  print(i, end = " ") #output: 3 2 1 0
```

- for-Schleife über Listen

```
l=[3,5,25]
for i in l:
  print(i, end = " ") #output: 3 5 25

for i in reversed(l):
  print(i, end = " ") #output: 25 5 3
```

1.5 Funktionen

Funktionen müssen nicht in einer bestimmten Reihenfolge deklariert werden im Gegensatz zu C++ (forward declarations). Folgendes ist also valid:

```
def foo():
  return bar()

def bar():
  return 0
```

1.5.1 Funktion Deklaration

```
def funktion1(arg1, arg2):
  …
  return value
```

1.5.2 Default Argumente

Man kann zusätzliche Argumente haben mit sogenannten «default argumenten»:

```
def specialprint(data = "hello world")
  print(data)
specialprint() #hello world
```

1.5.3 Globale und lokale Variabeln

Variabeln, die ausserhalb von einer Funktion definiert werden, sind global und können auch in der Funktion verwendet werden, wenn sie vor Funktionsaufruf definiert wird (sollte aber wenn möglich vermieden werden!):

```
def scaled(x):
  return x*scale #ok, da scale global ist

scale = 1.1 #muss vor Aufruf von scaled(10)!
print(scaled(10)) #Output: 11
```

- Eine lokale Variable in einer Funktion global zugänglich machen:

```
def double(x):
  global result #result wird global definiert
  result = x*2

double(7) #Funktionsaufruf, muss vor print(result)!
print(result) #Output: 14
```

2 Python Containers

Python Container können in geordnete (Sequenzen) und ungeordnete (Kollektionen) Container eingeteilt werden.
Sequenzen sind z.B. tuple (alle Typen), list (alle Typen), range (integers) und str (Zeichen)
Kollektionen sind z.B: set (nicht-assoziativ) und dict (assoziativ)

2.1 Container-Operationen

- Anzahl Elemente:

```
len(c)
```

- Element x enthalten?:

```
x in c
```

- Iterieren über alle Elemente:

```
for x in c:
 print(x)
```

2.2 Sequenzen (geordnete Container)

- Tuple mit 4 Elementen:

```
t = ("a", 0, -6, 3.3) #t -> ["a"] [0] [-6] [3.3]
#Tuple mit 1 Element: t = ('a',)
#empty tuple: t = ()
```

- Liste mit 4 Elementen:

```
l = [1, 3, "hi", -4] #l -> [1] [3] ['hi'] [-4]
```

- Range mit 4 Elementen:

```
r = range(0, 8, 2) #r -> [0] [2] [4] [6]
#Syntax:
#range(start, stop, step)
#range(start,stop) -> step = 1
#range(stop) -> start = 0, step = 1
```

- String mit Länge 5:

```
s = "hello" # s -> ['h'] ['e'] ['l'] ['l'] ['o']
#es ist egal, ob man " oder ' verwendet
```

WICHTIG: nur List ist veränderlich; tuple, range und string sind unveränderlich!

2.2.1 Allgemeine Sequenzoperationen

- Subskript-Operator `l[i]`:

```
l = [1, 3, 'hi', -4]
print(l[2]) #output: hi
print(l[-1]) #output: -4
```

- Enumeration:

```
enumerate(iterable, start)
#iterable = iterierbarer Container (Sequenz)
#start (optional) = (optional) enumerate beginnt bei
#dieser Nummer mit zählen, beim Weglassen von start->
#enumerate(iterable) beginnt bei 0
#Die enumerate(iterable, start) funktion fügt einen
#Counter zu einem iterable hinzu und gibt ein enumerate
#Objekt zurück
```

- Enumeration Beispiel:

```
for index, value in enumerate(l):
 print(index, value)
#output:
# 0 1
# 1 3
# 2 hi
# 3 -4
```

- Sequenzen `s1` und `s2` kombinieren (zip):

```
z = zip(s1,s2)
#Beispiel:
#s1 -> ["Lea"] ["Tim"] ["Mortis"]
#s2 -> [22] [19] [69]
#z -> [("Lea",22)] [("Tim",19)] [("Mortis",69)]
```

- Ausgabe mit einer for loop:

```
for name, age in z:
 print(name,"->",age)
#output:
# Lea -> 22
# Tim -> 19
# Mortis -> 69
```

- Slicing (Teilsequenz) einer Sequenz s:

```
teilseq = s[start:stop:step]
teilseq = s[start:stop] #step = 1
teilseq = s[:stop:step] #start = 0
teilseq = s[start::step] #stop = len(s)
```

2.2.2 Listenoperationen

- Element ändern:

```
l[i] = value
```

• Element hinzufügen am Ende:
```
l.append(value)
```
• Element entfernen an der Stelle i:
```
del l[i]
```
• Liste umkehren:
```
l.reverse()
```
• Liste mit k Elementen mit Wert v erzeugen:
```
l=[v]*k
```
• Ein String s in eine Liste umwandeln:
```
s.split(seperator, maxsplit)
#seperator und maxsplit sind optional
#s.split() -> wird bei jedem Whitespace gesplittet
#s.split(", ") -> wird bei jedem ", " gesplittet
#s.split(maxsplit = 10) -> wird 10 mal bei jedem
#Whitespace gesplittet -> Liste hat 11 Einträge
```

2.2.3 List Comprehension

• Eine Funktion `f(x)` auf alle Elemente der Liste `l` anwenden:
```
l2 = [f(x) for x in l] #z.B. 2*x für f(x) einsetzen
```
• Eine Funktion `f(x)` auf eine range anwenden:
```
r2 = [f(x) for x in range(1,6)]
```
• Eine Funktion `f(x)` nur auf Elemente der Liste `l` anwenden, die `g(x)` erfüllen (Filter):
```
l3 = [f(x) for x in l if g(x)]
```
• Beispiel: Eine Sequenz von Zahlen einlesen:
```
l = [int(x) for x in input("Input: ").split()]
```

2.2.4 Common String Operations

• Zugriff auf Element:
```
s[i]
```
• Zwei Strings zusammenführen (concatenating):
```
s1 = "hello"
s2 = "world"
s3 = s1 + s2 #s3 = "hello world"
```
• Leerzeichen (whitespaces) entfernen am Anfang und Ende
```
s = "   banana "
s=s.strip() #s="banana"
```
• Einen String in eine Liste von chars umwandeln:
```
s = list(s)
```
• Beispiel: check, ob s ein String mit Inhalt ist:
```
type(s) == str and len(s.strip())
```

2.3 Kollektionen(ungeordnete Container)

• Menge (Set) mit 3 Items:
```
s = {1, 29, 12}
```
• Wörterbuch (Dictionary) mit 3 Items:
```
d = {"Lea":22, "Tim":19, "Mortis":69} #key:value
```

2.3.1 Mengenoperationen (Set Operations)

```
s = {1, 29, 12} #set erstellen
```
• Item hinzufügen:
```
s.add(69)
```
• Item entfernen:
```
s.remove(29)
```
• Nach einem Item suchen:
```
12 in s #gibt bool zurück
```

2.3.2 Dictionary Operationen

```
d={"Lea":22, "Tim":19, "Mortis":69} #dict erstellen
```
• Item ändern:
```
d["Lea"] = 23
```
• Item hinzufügen:
```
d["Peter"] = 24 #einen unbenutzten Key verwenden
```
• Item löschen:
```
del d["Mortis"] #item löschen
```
• Nach einem Key suchen:
```
"Tim" in d #gibt bool zurück
```
• Auf Value bei einem Key zugreifen:
```
d["Tim"] #hat Wert 19
```
• Zwei Listen zu einem Dictionary machen:
```
staedte = ["Zurich", "Basel", "Bern"] #Liste 1
plz = [8000, 4000, 3000] #Liste 2
d2 = dict(zip(staedte,plz)) #Wörterbuch d2 erstellen
```

2.3.3 Über ein Dictionary iterieren

• Über alle Keys einer Dictionary iterieren:
```
for key in d.keys():
  print(key) #Lea Tim Mortis
```
• Über alle Items einer Dictionary iterieren:
```
for item in d.items():
  print(item) #('Lea',22) ('Tim',19) ('Mortis', 69)
```
• Über alle Values einer Dictionary iterieren:
```
for value in d.values():
  print(value) #22 19 69
```
• Über alle Items iterieren und Keys & Values separiert ausgeben:
```
for key, value in d.items():
  print(key+" "+value) #Lea 22 Tim 19 Mortis 69
```

2.3.4 Dictionary/Set Comprehension

• Ein Set in eine Dictionary umwandeln, für jedes Element `x` im Set wird ein Item mit dem Key `f(x)` (Resultat = Key) und Value `g(x)` (Resultat = Value) generiert:
```
d3 = {f(x):g(x) for x in s} #s being a set
```
• Ein Set in eine Dictionary umwandeln, nur die Items im Set, die `h(x)`erfüllen werden genommen:
```
d4 = {f(x):g(x) for x in s if h(x)}
```
• Dictionary comprehension mit mehreren Variabeln
```
d5 = {f(x):g(y) for x, y in h(z)}
#h muss eine List of tuples returnen, z.B. e.g.: zip,
#d.items. Im Falle von einer Dictionary d.items wenden
#wir f(x) auf den Keys und and g(y) auf den Values an.
```
• Beispiel: Value von jedem ungeradem Key in der Dictionary d mit 2 multiplizieren:
```
d6 = {k:2*v for k, v in d.items() if k % 2 == 1}
```

3 Numpy

Numpy ist eine Python package (äquivalent zu einer C++ library) welche operationen mit n-dimensionalen arrays und verschiedene Rechenmethoden unterstützt

• Numpy package importieren:
```
import numpy as np
```

Jetzt kannst du mit "np" auf Funktionen/Klassen von Numpy zugreifen.

3.1 Numpy Arrays

Numpy arrays sind ähnlich wie Python Listen. Wichtige Unterschiede sind in dieser Tabelle:

Lists	Numpy Arrays
Variable Grösse	Fixe Grösse
Beliebige Elementtypen	Ein Elementtyp für alle
Operationen nur auf einzelne Elemente	Operationen auf den ganzen Array/Spalten/Zeilen
Hauptsächlich 1D	Multi-dimensional

3.1.1 Numpy Arrays erstellen

• Mit Sequenzen (geordnete Container):
```
l = [1, 2, 3, 4]
a = np.array(l)
#array([1,2,3,4])
b = np.array(range(2,10,3))
#array([2,5,8])
c = np.array([[1,2],[3,4]])
#array([[1,2],
#       [3,4]])
```
• Mit zufälligen float Zahlen zwischen [0,1)
```
R = np.random.random(10)
#a numpy array with 10 random values which each have a
value between [0,1)
```
• Mit random float Zahlen:
```
R = np.random.uniform(-1,1,5)
#a numpy array with 5 random values in [-1,1]
```
• Mit random integer Zahlen:
```
R = np.random.randint(1,7,10)
#a numpy array with 10 random values in [1,6]
```
• Mithilfe von np.arange (stop ist nicht **inklusiv**):
```
R = np.arange(2,10,3) #array([2,5,8])
#the same output as np.array(range(2,10,3))
#np.arange(start, stop, step)
#np.arange(start,stop) -> step = 1
#n.arange(stop) -> start = 0, step = 1
```
• Mit linspace (stop ist **inklusiv**):
```
R = np.linspace(start,stop,num)
#a numpy array with num equally spaced elements
#between start and stop.
#Step size = (stop-start)/(num-1)
```

3.1.2 Numpy Array Operationen

• Die Anzahl an Elementen zurückgeben:
```
a = np.arange(10) #array([0,1,2,3,4,5,6,7,8,9])
a.size() #10
```
• Zugriff auf Element (1D array)
```
a[5] #5
```
• Zugriff auf Elemente (2D array):
```
A = np.array([[1,2,3],[4,5,6]])
A[1,2] #=A[1][2]=6
A[:,2] #array([3,6])
A[1,:] #array([4,5,6])
```

3.1.3 Numpy Array Slicing

• Slicing hängt von den Dimensionen des Arrays ab. Für 1D Arrays:
```
A = np.arange(10) #[0,1,2,3,4,5,6,7,8,9]
A[2:5:2] #array([2,4])
```
• For 2D arrays (matrices):
```
A = np.array([[1,2,3],
              [4,5,6],
               [7,8,9]])
A[0:2,1:3] #array([2,3],
                  [5,6])
```

3.1.4 Numpy Array Statistiken

```
a = np.linspace(-4,-2,3) #array([-4,-3,-2])
```
• Minimum of all elements:
```
a.min() #-4
```
• Maximum of all elements:
```
a.max() #-2
```
• Sum of all elements:
```
a.sum() #-9
```
• Average of all elements:
```
np.mean(a) #-3
```
• Standard deviation of all elements:
```
np.std(a) #0.81
```

3.1.5 Mathematische Operationen Numpy Arrays

• Generally mathematical operations are carried out element-wise:
```
import numpy as np
A = np.array([[2,3,4],[6,7,6]])
B = np.array([[1,9,1],[2,3,9]])
A+1
#array([[3,4,5],
#       [7,8,7]])
A * 2
#array([[4,6,8],
#       [12,14,12]])
A ** 4
#array([[16,81,256],
#       [1296,2401,1296]])
np.sin(A)
#array([[0.909,0.141,-0.756],
#       [-0.279,0.657,-0.279]])
A + B
#array([[3,12,5],
#       [8,10,15]])
A * B
#array([[2,27,4],
#       [12.21,54]])
np.sum(A, axis = 0)
#= A.sum(axis = 0) -> array([8,10,10])
np.sum(A, axis = 1)
#= A.sum(axis = 1) -> array([9,19])
```

3.1.6 Matrix Operationen auf Numpy Arrays

• Multiplikation zweier Matrizen mit @ (Dimensionen müssen stimmen) :

```
import numpy as np
A = np.array([[2,3,4],[6,7,6]])
A @ np.array([[1, 4], [3, 4], [4,6]])
#array([[27, 44],
#       [51, 88]])
```

• Die dot() Funktion verwenden, um zwei Matrizen zu multiplizieren:

```
A.dot(np.array([[1,4],[3,4],[4,6]])
#array([[27, 44],
#       [51, 88]])
```

• Die dot() verwenden, um das Skalarprodukt zwischen zwei Vektoren zu finden:

```
a = np.array([1,2,3])
b = np.array([3,4,6])
a.dot(b) #=29
```

3.1.7 Filtering Numpy Arrays

• Ein Numpy Array mithilfe des Subskriptoperators filtern:

```
a = np.arange(7) #array([0,1,2,3,4,5,6])
f = a % 2 == 0
a[f] #array([0,2,4,6])
```

4 Pandas

Pandas ist ein Python package, welches sehr nützlich für das Arbeiten mit Tabellen ist.

Pandas package importieren (nach Installation):

```
import pandas as pd
```

4.1 CSV Datei einlesen mit Pandas

• Eine CSV Datei (im gleichen Ordner gespeichert wie das Programm) einlesen:

```
climate = pd.read_csv("climate.csv", sep=",",
index_col=0, usecols=["time", …])
#"sep" -> what characters values in the csv #file are
#separated by. "index_col" -> what the index column will
#be. "usecols" -> what columns of the csv data will be
#selected.
```

4.2 Pandas Dataframe

Man kann sich eine Dataframe als eine 2D-Liste vorstellen (Liste in einer Liste). Ein Dataframe sieht könnte folgendermassen aussehen:

	Unnamed: 0	time	jan	feb	mar	apr	may	jun
0	0	1864	-7.10	-4.52	0.04	2.11	7.43	9.48
1	1	1865	-3.47	-6.25	-5.91	7.03	10.09	10.98
2	2	1866	-1.31	-0.42	-1.00	4.11	4.95	12.02
3	3	1867	-3.87	0.56	-0.13	3.49	7.74	10.57
4	4	1868	-5.46	-1.53	-2.30	2.33	12.04	11.97
...	...	...	...	...	...	...	...	...
153	153	2017	-5.15	0.46	4.11	4.42	9.80	15.18
154	154	2018	0.48	-5.21	-0.21	7.81	10.43	13.81
155	155	2019	-4.37	0.73	2.27	4.47	6.08	15.25
156	156	2020	-0.28	1.62	1.53	7.62	9.53	11.82
157	157	2021	-3.56	NaN	NaN	NaN	NaN	NaN

climate

Die Spalte ganz links ist die "Index Spalte".

4.2.1 Die Index Spalte verändern

• Die Index Spalte ändern (erstellt eine Kopie):

```
climate2 = climate.set_index("time")
```

time	Unnamed: 0	jan	feb	mar	apr	may	jun
1864	0	-7.10	-4.52	0.04	2.11	7.43	9.48
1865	1	-3.47	-6.25	-5.91	7.03	10.09	10.98
1866	2	-1.31	-0.42	-1.00	4.11	4.95	12.02
1867	3	-3.87	0.56	-0.13	3.49	7.74	10.57
1868	4	-5.46	-1.53	-2.30	2.33	12.04	11.97
...	...	...	...	...	...	...	...
2017	153	-5.15	0.46	4.11	4.42	9.80	15.18
2018	154	0.48	-5.21	-0.21	7.81	10.43	13.81
2019	155	-4.37	0.73	2.27	4.47	6.08	15.25
2020	156	-0.28	1.62	1.53	7.62	9.53	11.82
2021	157	-3.56	NaN	NaN	NaN	NaN	NaN

climate

4.2.2 Spalten umbenennen

• Mithilfe der "rename" Funktion:

```
climate = climate.rename(columns={"time":"date",
…}
#renames the time column as "date". Add any entries in
#the form of: "old_index_name":"new_index_name"
```

• Die Spaltennamen direkt setzen:

```
data.columns = ["Date", "January", "February",
…]
#needs to be the same length as the number of columns
```

4.2.3 Auf Dataframe Elemente zugreifen

• Auf eine einzelne Spalte zugreifen (Typ: Series):

```
climate["feb"]
#gets the column "feb"
```

• Auf mehrere Spalten zugreifen (Typ: Dataframe):

```
climate[["jan", "mar"]]
#gets the columns "jan" and "mar"
```

• Auf eine einzelne Zeile mit Index zugreifen (Typ: Series):

```
climate.iloc[3]
#gets row 3
```

• Verschiedene Zeilen (Typ: Dataframe):

```
Climate.iloc[1:4]
#gets rows 1 to 3
```

• Auf eine Subtabelle zugreifen mit Indizes (Typ: Dataframe):

```
climate.iloc[4:7,1:2]
#gets rows 4,7 with data only from column 1
```

• Auf eine Subtabelle zugreifen mit Spaltenindexwerten und Spaltennamen (Typ: Dataframe):

```
climate2 = climate.set_index("time")
climate2.loc[1864:1868,"jan":"mar"]
#includes the rows labeled with 1864 until and including
#1868, the columns from "jan" until and including "mar"
```

• Auf ein einzelnes Element zugreifen:

```
climate["jan"][3]
#gets the element in column "jan" in row 3
```

4.2.4 Dataframes filtern

• Zeilen filtern:

```
climate[climate["jan"]>2]
#filters out the rows with values in the "jan" column
#less than 2
```

• Beispiel: Alle Einträge in "jan" und Werten über 2:

```
climate["jan"][climate["jan"]>2]
```

4.2.5 Mit Invalid Data umgehen

• Alle Werte in einer Spalte zu numeric konvertieren:

```
data[column] = pd.to_numeric(data[column],
errors="coerce")
#converts all the values to numeric values.
#errors="coerce" -> converts values which cannot be
#converted to NaN.
```

• Alle Zeilen mit NaN Werten löschen:

```
data.dropna(axis = 0, how="any")
#how="any" -> delete row if any value is NaN.
#how="all" -> delete row if all values are NaN
#axis = 1 -> delete column instead of row
```

• Alle Einträge mit NaN Werten mit anderem Wert ersetzen:

```
data.fillna(0) #fill any NaN entries with 0
```

4.2.6 Dataframes verändern

• Spalte hinzufügen:

```
climate["new_col"] = climate["time"] +
climate["jan"]
#"new_col" is a new column who's values are #those of the
"time" and "jan" column added
```

• Spalte löschen:

```
climate = climate.drop(columns=["time"])
#delete the "time" column
```

• Zeile hinzufügen:

```
d = {"mar":34, "jan":23}
climate.append(d, ignore_index=True)
#adds another row with the values 34 for "mar" and 23 for
#"jan". Other entries are NaN
```

• Zeile löschen:

```
climate = climate.drop(climate.index[0])
#deletes row 0
```

• Dataframe transponieren (Zeilen und Spalten vertauschen):

```
climate = climate.T
```

4.2.7 Daten in Dataframes analysieren

• Alle Einträge in jeder Spalte aufaddieren (Typ: Series):

```
climate.sum()
```

• Das Maximum aller Einträge in jeder Spalte (Typ: Series):

```
climate.max()
```

• Ein Dataframe erstellen, welches das Maximum und Summe für jede Spalte zusammenfasst:

```
climate.agg(["max","sum"])
#A dataframe containing the same columns as climate
#with row 0 containing the max of the column and row 1
#containing the sum of the column. The strings in the
#list should be names of valid pandas Series functions.
```

• Statistische Informationen über jede Spalte bekommen (Typ: Dataframe):

```
climate.describe()
#includes a variety of statistical measures
```

• Ein Dataframe sortieren anhand von Einträgen in spezifischen Spalte(n):

```
climate = climate.sort_values(["time", "jan"],
ascending=False)
#sorts the rows by "time" in descending order. If two
#entries for "time" are equal, then the rows are sorted
#by "jan"
```

• Ein Dataframe aufteilen in Gruppen anhand von spezifischen Spalten und mathematische Operationen auf jede Gruppe anwenden :

```
data.groupby("column").sum()
#groups data based on the entries for "column" and
#calculates the sum for each group.
data.groupby("column").max()
#groups data based on the entries for "column" and
#calculates the max for each group.
```

5 Matplotlib

Matplotlib ist eine Python Package, mit der man viele Sachen visualisieren kann (Funktionen, Daten, Animationen).

Matplotlib importieren:

```
import matplotlib.pyplot as plt
```

Jetzt kannst du auf Klassen und Funktionen des Packages mit "plt" zugreifen.

5.1 Line Plots

• Zwei Numpy Arrays grafisch darstellen, X -> Wert x-Achse, Y->Wert y-Achse:

```
X = np.linspace(0,2*np.pi,100)
Y = np.sin(X)

fig, ax = plt.subplots()
ax.plot(X,Y)
```

5.2 Scatter Plots

• Zwei Numpy Arrays grafisch darstellen, X -> Wert x-Achse:

```
X = np.arange(1,9)
Y = np.array([1,2,3,1,2,3,1,2])

fig, ax = plt.subplots()
ax.scatter(X,Y)
```

5.3 Histogramm Plots

• Um ein Histogramm zu erstellen:

```
fig, ax = plt.subplots()
X = np.random.randint(0, 100, 500)
# low: 0, high: 100, size: 500
ax.hist(X, bins=10)
plt.show()
```

5.4 Graph Styling

```
fig, ax = plt.subplots()
```

• Um einen Titel hinzuzufügen:

```
ax.set_title("title")
```

• x-Achse beschriften:

```
ax.set_xlabel("x label name")
```

• y-Achse beschriften:

```
ax.set_ylabel("y label name")
```

• Eine Legende hinzufügen:

```
ax.legend()
#requires that you labeled your plots, i.e.: when calling
#ax.plot(X,Y, label="name of function").
```

6 Algorithmen

Algorithmen sind eine Reihe von Anweisungen zur Lösung bestimmter Probleme. Die häufigsten Probleme, mit denen wir uns in der Informatik befassen, sind das Sortieren und Suchen nach Elementen in Daten.

6.1 Performance messen

6.1.1 Big O Notation

Um die Leistung eines Algorithmus zu messen, verwenden wir die sogenannte Big-O-Notation.

Sei g die Beziehung zwischen Zeit und Eingabegröße für einen Algorithmus:

• Falls g nicht schneller wächst als c*f:
$g = \mathcal{O}(f)$
• Falls g etwa gleich schnell wächst wie c*f:
$g = \Theta(f)$ `#g~c*f, where c is a constant`
• Falls g nicht langsamer wächst als c*f:
$g = \Omega(f)$

Mathematische Definitionen:

$$\mathcal{O}(g) = \{ f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \}$$

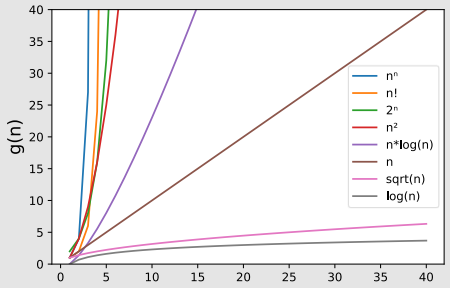
$$\Theta(g) = \{ f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq \frac{1}{c} \cdot g(n) \leq f(n) \leq c \cdot g(n) \}$$

$$\Omega(g) = \{ f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n) \}$$

6.1.2 Asymptotisches Wachstum von Funktionen

• Funktionen in **zunehmender Reihenfolge** (von asymptotischen Wachstum):

1. $\log(n)$
2. $\sqrt{n}$
3. n
4. $n \cdot \log(n)$
5. n^2
6. 2^n
7. $n!$
8. n^n



6.1.3 Laufzeitanalyse

Um die Laufzeit von Code analysieren zu können, müssen wir bestimmte Annahmen treffen:

• Vergleiche haben 'Zeitkosten' von 1:

```
if x==1 #has a cost of 1
if x>1 #has a cost of 1
```

• Mathematische operationen haben Zeitkosten von 1:

```
6 + 4 #has a cost of 1
```

• Zuweisungen haben Zeitkosten von 1:

```
x = 69 #has a cost of 1
```

• Generell haben operationen auf fundamentale typen Zeitkosten von 1.
• Beispiel Laufzaitanalyse (selection sort):

```
def sort(a):
  n = len(a)
  for i in range(n):
    mini = i
    for j in range(i+1,n):
      if a[j] < a[min]:
        mini = j
    a[mini], a[i] = a[i], a[mini]
```

$$time(n) = 1 + \sum_{i=0}^{n-1} (1 + (\sum_{j=i+1}^{n-1} 2) + 1)$$

$$= 1 + \sum_{i=0}^{n-1} (1 + (n-1-i) + 1) = 1 + \sum_{i=0}^{n-1} (2n - 2i)$$

$$= 1 + 2n^2 - 2\left(\frac{n(n+1)}{2} - n\right) = 1 + n^2 + \mathrm{n}$$

$$= \Theta(n^2)$$

6.1.4 Nützliche Formeln

$$\sum_{i=0}^{n-1} 1 = n$$

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6}$$

6.1.5 Teleskopieren

Beispiel:

$$T(n) = \begin{cases} d, & n = 1 \\ T\left(\frac{n}{2}\right) + c, & n > 1 \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$

$$= T\left(\frac{n}{2^i}\right) + i \cdot c$$

$$= T\left(\frac{n}{n}\right) + \log_2 n \cdot c$$

$$= d + \log_2 n \cdot c = \Theta(\log(n))$$

6.1.6 Recursive Runtime Trick

Step 1: Rewrite recursive function T(n) into following form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \theta(n^k \cdot log^p(n))$$

where $\theta(n^k \cdot log^p(n))$ is the additional Runtime in each call of T(n)

Step 2: Runtime is one of these cases:

1. if $a > b^k$, then $T(n) = \theta(n^{log_b(a)})$
2. if $a = b^k$, and
   a) if $p > -1$, then $T(n) = \theta(n^{log_b(a)} \cdot log^{p+1}(n))$
   b) if $p = -1$, then $T(n) = \theta(n^{log_b(a)} \cdot log(log\ (n)))$
   c) if $p < -1$, then $T(n) = \theta(n^{log_b(a)})$
3. if $a < b^k$, and
   a) if $p >= 0$, then $T(n) = \theta(n^k \cdot log^p(n))$
   b) if $p < 0$, then $T(n) = \theta(n^k)$

6.2 Sortieralgorithmen

6.2.1 Invarianten

Bei **Algorithmen** mit **Schleifen** gibt es normalerweise eine sogenannte "**Invariante**". Diese Invariante erfüllt folgende Bedingungen:

1. Initialization: diese Bedingung ist vor Beginn der Loop erfüllt
2. Continuation: die Bedingung ist bei jeder Iteration (einzelner Schleifendurchlauf) wahr
3. Termination: die Bedingung gilt auch am Ende der Loop

Ein Algorithmus, der eine Schleife und eine Invariante hat, gilt als korrekt, wenn die Invariante das oben Genannte erfüllt.

6.2.2 Divide and Conquer

"Divide and Conquer" ist ein Algorithmustyp, bei dem man das Problem rekursiv immer wieder in zwei oder mehrere gleich grosse Teilprobleme vom gleichen Typ teilt. Beispiele hierfür sind Mergesort und Quicksort.

6.2.3 Selection Sort

Ein Array sortieren, indem man jeweils das minimum von links aus wählt und swappt. Pseudocode:

```
Input: Array A = (A[0],...,A[n]), n ≥ 0
Output: Sorted Array A
for i ← 1 to n - 1 do
    p←i
    for j ← i + 1 to n do
        if A[j] < A[p] then p ← j;
            swap(A[i], A[p])
```

Case	Beschreibung	Laufzeit
Worst-case	A is reverse sorted.	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$

6.2.4 Insertion Sort

Ein Array sortieren, indem man jeweils den nächsten Wert von links nach rechts nimmt und an die richtige Position swappt.

```
Input: Array A = (A[0],...,A[n]), n ≥ 0
Output: Sorted Array A
for i ← 1 to n do
    key ← arr[i]
    j ← i - 1
    while j >= 0 and arr[j] > key do
        arr[j+1] ← arr[j]
        j ← j - 1
    arr[j+1] ← key
```

Case	Beschreibung	Laufzeit
Worst-case	A is reverse sorted.	$\Theta(n^2)$
Average-case	-	$\Theta(n^2)$
Best-case	A is already sorted	$\Theta(n)$

6.2.5 Merge Sort

Ein Array sortieren, indem man es in kleinere Subarray teilt (divide and conquer) und dann diese Subarrays sortiereny.
Pseudocode:

```
Input: Array A with length n. 1≤l≤r≤n
Output: A[l,...,r]sorted.
if l < r then
    m ← ⌊(l + r)/2⌋
    Mergesort(A, l, m)
    Mergesort(A, m + 1, r)
    Merge(A, l, m, r)
```

Braucht zusätzlich $\Theta(n)$ Speicher für die Subarrays.

Case	Beschreibung	Laufzeit
All cases	-	$\Theta(n \cdot \log(n))$

6.2.6 Quick Sort

Ein Array sortieren, indem man das Array rekursiv immer wieder halbiert (divide and conquer) in zwei Teile: eine Hälfte, die Elemente grösser als den Pivot enthält und eine Hälfte, die Elemente A kleiner als den Pivot enthält.
Pseudocode:

```
Input: Array A with length n. 1≤l≤r≤n
Output: A[l,...,r]sorted.
if l < r then
    k ← partition(A, l, r)
    quicksort(A, l, k-1)
    quicksort(A, k+1, l)
```

Case	Beschreibung	Laufzeit
Worst-case	Pivot is the min/max value.	$\Theta(n^2)$
Average-case	Pivot is chosen randomly.	$\Theta(n \cdot \log(n))$
Best-case	Pivot is always the median of the array.	$\Theta(n \cdot \log(n))$

Pivot is often the median of three elements:

```
Pivot = Median3(A[l],A[(l+r)//2],A[r])
```

6.2.7 Heapsort

Ein Array sortieren. indem man es in ein Heap (siehe 7.2) umwandelt und dann ein sortierter Array aus dem Heap erstellen. Pseudocode:

```
Input: Array A with length n.
Output: A sorted.
for i ← n/2 downto 1 do
    SiftDown(A, i, n) #create a heap
for i ← n downto 2 do #sort the heap
    Swap(A[1], A[i])
    SiftDown(A, 1, i - 1)
```

Case	Beschreibung	Laufzeit
All	-	$\Theta(n \cdot \log(n))$

6.3 Suchalgorithmen

6.3.1 Linear Search

Für den Index eines spezifischen Elementes in einem **unsortiertem** Array suchen. Code:

```
def LinearSearch(a, b):
  for i, x in enumerate(a):
    if x == b:
     return i
return "not found"
```

Case	Beschreibung	Laufzeit
Worst-case	b is at the end of the array.	$\Theta(n)$
Average-case	-	$\Theta(n)$
Best-case	b is at the beginning.	$\Theta(1)$

6.3.2 Binary Search

Für den Index eines spezifischen Elementes in einem **sortiertem** Array suchen. Code:

```
#a: array
#l: left index
#r: right index
#b: element to search for
def bin_search(a, l, r, b):
  if r < l:
    return None
  else:
    m = (l + r) // 2
    if a[m] == b:
     return m
    elif b < a[m]:
     return bin_search(a, l, m-1, b)
    else: #a[m]>b
     return bin_search(a, m+1, r, b)
```

Case	Beschreibung	Laufzeit
Worst-case	b is the max/min value.	$\Theta(\log(n))$
Average-case	-	$\Theta(\log(n))$
Best-case	b is exactly the median value.	$\Theta(1)$

7 Datenstrukturen

Datenstrukturen sind Methoden zur Organisation und Speicherung von Daten für effizienten Zugriff und Manipulation.

Anmerkung zu Binary Trees (BSTs): Ein Binary Tree ist ein Baum mit höchstens 2 Kindern. Verschiedene Typen von BSTs sind:

- Full BST: jeder Node hat 0 oder 2 Kinder.
- Complete BST: jedes Level ausser das Tiefste ist komplett gefüllt. Das tiefste Level ist von links nach rechts gefüllt
- Perfect BST: jedes Level ist komplett gefüllt.
-

7.1 FIFO/LIFO

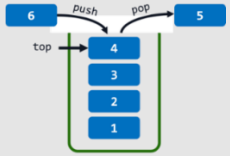
Manche Datenstrukturen können als FIFO (First-In-First-Out) oder LIFO (Last-In-First-Out) kategorisiert werden.

FIFO-Datenstrukturen bieten schnellen Zugriff auf das erste Element, das eingefügt wurde.

LIFO-Datenstrukturen bieten schnellen Zugriff auf das zuletzt eingefügte Element.

Beispiel für FIFO: Queue

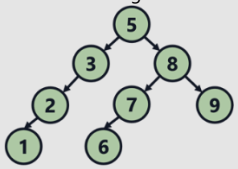
Beispiel für LIFO: Stack



7.2 Binary Search Trees (BSTs)

Ein Binary Search Tree ist ein Binary Tree, der zusätzlich folgende Bedingungen erfüllt:

1. Jeder Node v enthält einen Key
2. Keys im linken Subtree sind kleiner als `v.key`
3. Keys im rechten Subtree sind grösser als `v.key`



A binary search tree

7.2.1 Höhe eines BSTs

Die Höhe eines binären Suchbaums (BST) wird definiert als die maximale Tiefe, die man durch Rekursion erreichen kann, um einen Knoten ohne Kinder zu erreichen. Code-Beispiel:

```
def height(node):
    if node == None:
        return 0
    else:
        return 1 + max(height(node.left), \
                       height(node.right))
```

7.2.2 Node (key) suchen

Einen Node mit einem spezifischen Key "key" returnen:

```
def findNode(root, key):
    n = root
    while n != None and n.key != key:
       if key < n.key:
           n = n.left
       else:
           n = n.right
    return n
```

Case	Beschreibung	Laufzeit
Worst-case	The tree is degenerated	$\Theta(n)$
Average-case	The tree is balanced	$\Theta(\log(n))$
Best-case	Key is the root of the tree	$\Theta(1)$

7.2.3 Node einfügen

Einen Node mit einem spezifischen Key "key" einfügen:

```
def addNode(root, key):
  if root == None:
    root = Node(key)
  n = root
  while n.key != key:
    if key < n.key:
     if n.left == None:
       n.left = Node(key)
     n = n.left
    else:
     if n.right == None:
       n.right = Node(key)
     n = n.right
  return root
```

Case	Beschreibung	Laufzeit
Worst-case	The tree is degenerated	$\Theta(n)$
Average-case	The tree is balanced	$\Theta(\log(n))$
Best-case	The tree is empty	$\Theta(1)$

7.2.4 Node entfernen

1. Falls der Node keine Kinder hat, einfach löschen, indem man die Variable auf `None` setzt.
2. Falls der Node nur ein Kind hat, den Node einfach mit diesem Kind ersetzen.
3. Falls der Node zwei Kinder hat, ersetzten wir ihn mit seinem ***symmetric successor*** – dem nächstgrösseren Node (Key). Code:

```
def deleteNode(root, key):
    if root is None:
        return None
    # Find the node to be deleted
    if key < root.val:
        root.left = deleteNode(root.left, key)
    elif key > root.val:
        root.right = deleteNode(root.right, key)
    else:
        # Case 1: Node has no children
        if root.left is None and root.right is \
            None:
            root = None
        # Case 2: Node has one child
        elif root.left is None:
            root = root.right
        elif root.right is None:
            root = root.left
        # Case 3: Node has two children
        else:
            successor = findSuccessor(root)
            root.val = successor.val
            root.right = deleteNode(root.right,\
               successor.val)
    return root

def findSuccessor(start_node)
    parent = None
    node = start_node.right
    while node.left is not None:
        parent = node
        node = node.left
    return node
```

Case	Beschreibung	Laufzeit
All	Runtime is dominated by finding the successor.	$\mathcal{O}(h)$ h is the height of the tree

7.2.5 Traversal

Es gibt verschiedene Wege, um über einen BST zu traversieren (printen):

1. Inorder traversal (**aufsteigende Reihenfolge**):

```
def traverse(node):
    traverse(node.left)
    print(node.key)
    traverse(node.right)
```

2. Preorder traversal:

```
def traverse(node):
    print(node.key)
    traverse(node.left)
    traverse(node.right)
```

3. Postorder traversal:

```
def traverse(node):
    traverse(node.left)
    traverse(node.right)
    print(node.key)
```

7.2.6 Traversal Regeln

- Inorder traversal: Es gibt keine mögliche Repräsentation des Baums, wenn die Sequenz nicht in aufsteigender Reihenfolge vorliegt. **Repräsentation ist nicht eindeutig**

Beispiel: 1 2 4 3 hat keine Repräsentation als BST.

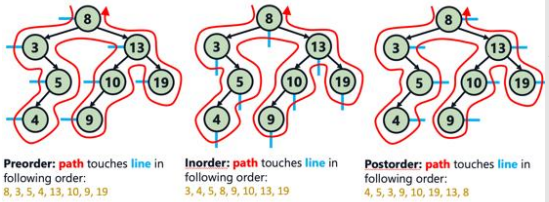
- Preorder traversal: Es gibt keine mögliche Repräsentation des Baums, wenn es keine Möglichkeit gibt, die erste Zahl in der Sequenz so zu platzieren (rekursiv auch für die neuen Teilsequenzen), dass die Zahlen links davon kleiner und die Zahlen rechts davon größer sind. **Repräsentation ist eindeutig**

Beispiel: 4 3 1 2 8 6 5 7
3 1 2 4 8 6 5 7
1 2 3, 6 5 7 8
1 2, 5 6 7 -> valid

- Postorder traversal: Es gibt keine mögliche Repräsentation des Baums, wenn es keine Möglichkeit gibt, die letzte Zahl in der Sequenz so zu platzieren (rekursiv auch für die neuen Teilsequenzen), dass die Zahlen links davon kleiner und die Zahlen rechts davon größer sind. **Repräsentation ist eindeutig**

Beispiel: 1 3 2 5 6 8 7 4
1 3 2 4 5 6 8 7
1 2 3, 5 6 7 8
5 6 -> valid

7.2.7 Tree Traversal - Trick

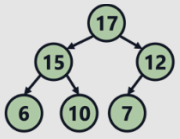


7.3 (Max) Heaps

Ein Heap ist ein Binary Tree, welcher zusätzlich folgende Bedingungen erfüllt::

1. Der Binary Tree ist complete (siehe 7).
2. Das tiefste Level ist von Links nach Rechts aufgefüllt (entspricht der Definition von complete, siehe Kapitel 7)
3. Der Key vom Parent ist immer grösser als die Keys der Children (bei Min-Heap kleiner)

Heaps sind nützlich, wenn man jederzeit effizienten Zugriff auf das Maximum/Minimum haben möchte.



A max heap

7.3.1 Array Darstellung eines Heaps

Man kann einen Heap als Array darstellen, indem man Stufenweise (ganz oben bei root beginnen) von links nach rechts geht und die Keys entnimmt.
Beispiel: Max Heap (vom Bild oben Kapitel 7.2):
22,20,18,16,12,15,17,3,2,8,11,14

Gegeben ein Element mit Index i (erster index = 0):

- Children von Element i: `{2i+1, 2i+2}`
- Parent von Element i: `i-1//2`

Mit erstem Index = 1:

- Children von Element i: `{2i, 2i+1}`
- Parent von Element i: `i//2`

7.3.2 Höhe eines Heaps

- Gegeben seien n Elemente:

$$H(n) = \lceil \log_2(n+1) \rceil$$

7.3.3 Element hinzufügen

Um ein Element in den Heap einzufügen, platziert man es an den ersten freien Platz (erster freier Platz im tiefsten Level, links nach rechts gehend). Dann wird das Element iterativ mit seinem Elternknoten getauscht, bis die Heap-Bedingungen (Siehe 7.2) erfüllt sind.

```
def insert(heap, value)
    heap.append(value)
    SiftUp(heap, len(heap)-1)


#a is the array,m is the end of the array.
def SiftUp(a, m)
    v = a[m]
    c = m
    p = c//2
    while c > 0 and v > a[p]:
        a[c] = a[p]
        c = p
        p = c//2
    a[c] = v
```

Case	Beschreibung	Laufzeit
All	In the worst case, inserting an element will involve $\log(n)$ swaps.	$\mathcal{O}(\log(n))$

7.3.4 Das Maximum entfernen (Max-Heap)

Ersetze das Maximum mit dem Element ganz rechts im tiefsten Level und swappe diesen iterativ nach unten in Richtung des grösseren Kindes (bei Min-Heap in Richtung kleineren Kindes).

```
def removeMax(heap):
    if len(heap) == 0:
        return None
    max_val = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    SiftDown(heap, 0, len(heap)-1)
    return max_val

#i is the index of the element that needs to be sifted
#down. a is the array. m is the end of the array
def SiftDown(a, i, m):
    while 2i ≤ m:
        j = 2i+1
        if j < m and a[j] < a[j+1]:
            j = j + 1
        elif a[i] < a[j]:
            swap(a[i],a[j])
            i = j
        else:
            i = m
```

Case	Beschreibung	Laufzeit
All	In the worst case, removing an element will involve $\log(n)$ swaps.	$\mathcal{O}(\log(n))$

7.3.5 "Heapify" eines Arrays

Eigenschaft: Die Blätter eines Heaps erfüllen die Heap-Bedingung trivially -> Es ist nur notwendig, die ersten n/2 Elemente "umzuwandeln" (heapify).

```
def heapify(a):
    n = len(a)
    for i in range(n//2 - 1, -1, -1):
        SiftDown(a, i, n)

def SiftDown(a, i, m):
    while 2*i + 1 < m:
        j = 2*i + 1
        if j + 1 < m and a[j] < a[j + 1]:
            j = j + 1
        if a[i] >= a[j]:
            break
        a[i], a[j] = a[j], a[i]
        i = j
```

Case	Description	Runtime
All	-	$\Theta(n)$

7.3.6 Heap sortieren

Einen Heap "a" effizient sortieren:

```
#a is a heap
def SortHeap(a)
    n = len(a)-1
    while n > 0:
        swap(a[0],a[n])
        SiftDown(a, 1, n-1)
        n = n - 1
```

Case	Beschreibung	Laufzeit
All	SiftDown traverses at most $\log(n)$ nodes. Sorting the array requires n calls to SiftDown.	$\Theta(n \cdot \log(n))$

7.4 Vectors/Lists

- geordnete Datenstruktur
- Schneller Zugriff via Index
- Langsam für updates (Element ändern).

Operation	Time Complexity
Index Access	$\mathcal{O}(1)$
Search (`in`)	$\mathcal{O}(n)$
Sorted Search	$\mathcal{O}(\log(n))$
Insertion	$\mathcal{O}(n)$
Removal	$\mathcal{O}(n)$

7.5 Linked Lists

- geordnete Datenstruktur
- Schnell für updates am anfang der Linked List

Operation	Time Complexity
Access	$\mathcal{O}(n)$
Search (`in`)	$\mathcal{O}(n)$
Insertion (head)	$\mathcal{O}(1)$
Removal (head)	$\mathcal{O}(1)$
Insertion (after value)	$\mathcal{O}(n)$
Insertion (after value)	$\mathcal{O}(n)$

7.6 Hash Tabellen

Hash-Tabellen sind eine Datenstruktur, die schnellen Zugriff auf Elemente ermöglicht.

- Unsortierte, ungeordnete Daten.
- Schnelle Suche.

Die Idee hinter der Implementierung besteht darin, eine "Hash-Funktion" zu verwenden, um einen Index/Adresse aus dem Element zu erhalten und das Element dort zu speichern.

Operation	Best Case Complexity	Worst Case Complexity
Search	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insertion	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Removal	$\mathcal{O}(1)$	$\mathcal{O}(n)$

7.6.1 Collision Handling

Problem: Der Eintrag am berechneten Index enthält möglicherweise schon ein Element. Lösung:

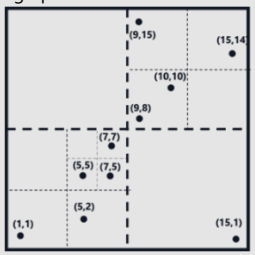
- Mit Probing: Der nächste verfügbare Index wird gewählt.
- Mit Chaining: Eine Linked List an jedem Eintrag.

7.6.2 Eigenschaften einer Hash Funktion

- Konsistent (liefert immer die gleiche Ausgabe für eine bestimmte Eingabe).
- Möglichst kollisionsfrei.

7.7 Quadtrees

Ein Quadtree ist ein Baum mit 4 Kindern. Wir können Quadtrees auch graphisch darstellen:

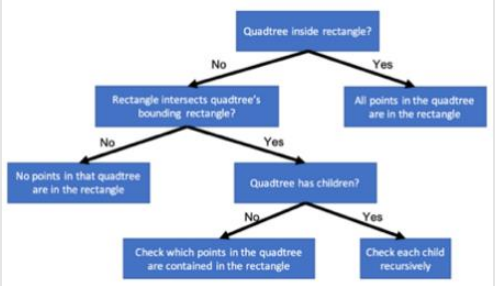


QuadTree mit max_cap = 2

Beispiel einer Quadtree Implementation:

```
class QuadTree:
  def __init__(self, l, u, max_cap)
    self.l = l #coordinate of lower left corner
    self.u = u #coordinate of upper right corner
    self.m = max_cap
    self.points = []
    self.children = None
    self.count = 0 #total number of points within
                   #this quadtree
  #we insert points into the quadtree until the
  #quadtree's capacity is exceeded, in which case we
  #add 4 children to the quadtree and put the points in
  #respective correct child
  def insert(self, point):
    if self.children is not None:
      index = self.get_index(point)
        if index is not None:
            self.children[index].insert(point)
            self.count += 1
            return
     self.points.append(point)
     if len(self.points) > self.m:
       self.subdivide()
       for point in self.points:
         index = self.get_index(point)
         if index is not None:
           self.children[index].insert(point)
           self.count += 1
       self.points = []
```

7.7.1 Suche nach Anzahl von Punkten im Rechteck



Time complexity: $\mathcal{O}(\log(n))$

8 Programmierkonzepte

8.1 Klassen und Objekte (OOP)

Klassen:

- Bündelung von Daten, die inhaltlich zusammengehören.
- Definition eines neuen Typs.

Daten:

- Gespeichert in Variablen der Klasse (Attribute).
- Standardwerte können in der Klasse deklariert werden (siehe 1.5.2).

Objekt:

- Instanz einer Klasse.

8.1.1 Example Implementation

```
class Earthquake:
     #Constructor (member variables: location, magnitude)
     def __init__(self, location, magnitude):
          self.loc = location
          self.mag = magnitude
          self.__id = 5 #hidden (private) attribute

     #Print operator overload -> what is shown when
     #print(Earthquake) is called
     def __str__(self):
          return "earthquake with mag: "  +
                  str(self.mag)

     #Overloaded ==
     def __eq__(self,other):
          return self.mag == other.mag and
                 self.loc == other.loc

     #method to access hidden/private attribute
     def get_id(self):
          return self.__id

     #A class method
     def bar(self, x):
          …

a = Earthquake(Indonesia, 5)
print(a) #earthquake with mag: 5
a.bar(5) #call class method bar
print(a.__id) #AttributeError
```

8.1.2 Magical Methods

Magical methods erlauben es, Operatoren in Python zu überladen. Hier ist eine Tabelle mit den magical methods, die definiert werden können:

Comparisons:

Operation	Meaning	Magical Method
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__
==	Equal to	__eq__
!=	Not equal to	__ne__

Relational Operations:

Operation	Meaning	Magical Method
+, +=	Addition	__add__, __iadd__
-	Subtraction	__sub__
*	Multiplication	__mul__
/	Division	__truediv__
//	Integer division	__floordiv__
%	Modulo	__modulo__
**	Exponentiation	__pow__

Sonstiges

Operation	Meaning	Magical Method
print()	overload print()	__str__
-	Negation	__neg__

Beispiel:

```
class Student:
    def __init__(self, name):
        self.name= name
    def __str__(self): #overload print()
        return self.name
```

8.1.3 Inheritance

Durch Inheritance (Vererbung) kann eine Klasse alle Attribute und Methoden einer anderen Klasse "erben".

Beispiel:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("The animal makes a sound.")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print("The dog barks.")

#Create an instance of the Dog class
dog = Dog("Fido", "Golden Retriever")

#Access the attributes of the Dog instance
print(dog.name)   #Output: Fido
print(dog.breed)  #Output: Golden Retriever

#Call the speak method on the Dog instance
dog.speak()  #Output: The dog barks.
```

8.2 Compiled vs Interpreted

Compiled (C++):

- Der Programmcode wird in Assembler übersetzt.
- Assembler wird ausgeführt.
- Einzelne Übersetzung mit Optimierungen.
- In der Regel höhere Leistung.

Interpreted (Python):

- Der Programmcode wird zusammen mit der Übersetzung ausgeführt.
- Die Übersetzung wird bei jeder Ausführung wiederholt.
- Schnell und einfach kleine Änderungen vorzunehmen.

8.3 Static vs Dynamically Typed

C++ ist statically typed (statisch typisiert):

- jedes Element hat einen vom Programmierer definierten Typ.
- Die verwendeten Typen werden während der Übersetzung überprüft, was bei Fehlern zu Kompilierungsfehlern (die während des Programms selbst auftreten) führt.

Python ist dynamically typed (dynamisch typisiert):

- Elemente haben im Voraus keinen Typ.
- Der Typ wird zur Laufzeit ausgewählt.
- Typen können zur Laufzeit geändert werden.
- Abhängig von dem Typ bei der Ausführung können Laufzeitfehler auftreten.
- Fehler sind schwieriger zu debuggen und treten nicht immer auf.

8.4 Generic Programming

Das Ziel von generic Programming besteht darin, den Code so weit wie möglich verwendbar zu machen (keine Notwendigkeit für neue Funktionen für verschiedene Typen). Dies kann mit Vorlagen (Templates) in C++ erreicht werden. In Python ist dies dank der dynamischen Typisierung nicht erforderlich.

8.5 Functional Programming

Die zentrale Idee ist es, Funktionen als Parameter an Funktionen zu übergeben. Ein Beispiel, bei dem wir eine Funktion an "map" übergeben:

```
#Define a list of numbers
numbers = [1, 2, 3, 4, 5]

#Define a function that squares a number
def square(x):
    return x ** 2

# Use map() to apply the square function to each element
#in the numbers list
squared_numbers = list(map(square, numbers))

#Print the result
print(squared_numbers) #Output: [1,4,9,16,25]
```

8.5.1 Lambda Funktionen

Lambda-Funktionen sind kleine Funktionen ohne einen spezifischen Namen, die nützlich sind, um sie als Parameter an eine Funktion zu übergeben.

```
lambda arguments : expression
```

- Lambda with one argument:

```
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers) #[1,4,9,16,25]
```

- Lambda with multiple arguments:

```
y = lambda x,y: x*y
y(5,3) #15
```

8.5.2 Examples of Functions that Accept Functions

- `map(func, it)` – applies a function on each element of a container.

```
n = [1,2,3,4,5]
sqrd_numbers = map(lambda x : x**2, n)
print(list(sqrd_numbers) #[1,4,9,16,25]
```

- `filter(func, it)` – removes any elements that don't fulfil a condition.

```
n = [1,2,3,4,5]
even_numbers = filter(lambda x : x%2==0, n)
print(list(even_numbers) #[2,4]
```

- `reduce(func,it)` – recursively reduce a container to a single value by applying a function to two elements.

```
from functools import reduce
n = [1,2,3,4,5]
sum_numbers = reduce(lambda x,y: x + y, n) #15
```

9 Dynamic Programming (DP)

DP (Dynamic Programming) ist eine Problem-Lösungsstrategie:

- DP ist im allgemeinen eine "bottom-up" Strategie - wir lösen iterativ kleinere Probleme, um allmählich grössere Probleme zu lösen.
- DP ist schneller als Rekursion, da wir bekannte Lösungen nicht erneut berechnen müssen.
- Wir speichern die Antworten auf kleinere Probleme in einer Tabelle. Dies wird als Memoisierung bezeichnet.

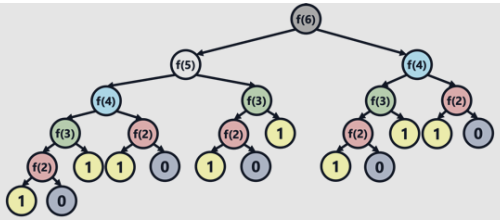
9.1 Wo kann man DP verwenden?

Probleme haben folgende Eigenschaften:

- Optimale Teilstruktur - die Antwort des Problems hängt von der Antwort einiger kleinerer Teilprobleme ab.
- Überlappende Teilprobleme - bei der Berechnung der Antwort auf ein Problem berechnen wir oft die Antwort auf dieselben Teilprobleme erneut.

Beispiel: Fibonacci

```
fib(n) = fib(n-1) + fib(n-2) #Optimal substructure
```



Rekursionsbaum für fib(6)

```
#Overlapping subproblems. Notice the repeating subtrees
#such as
#                       2
#                      / \
#                     1   0
```

9.2 DP Implementation of Fibonacci

```
def fib_DP(n):
  #create table
  F = [None] * (n+1)
  #border cases
  F[0] = 1
  F[1] = 1
  #Bottom-Up for loop
  for i in range(2, n+1):
    F[i] = F[i-1] + F[i-2]
  #return last value
return F[n]
```

9.3 Lösungsstrategie

1. Die erste Lösung mit Brute-Force oder der rekursiven Implementierung finden. Baum zeichnen oder den Prozess visualisieren.
2. Lösung analysieren - nach sich wiederholenden Teilproblemen suchen. Dann eine optimalen Teilstruktur für die Lösung finden - wie hängt die Antwort des Problems von der Antwort der Teilprobleme ab?
3. Überlegen, wie man die Antworten der Teilprobleme speichern könnte. Sollte es eine Liste oder eine Tabelle sein?
4. Die rekursive Implementierung umdrehen, um eine bottom-up, iterative Lösung zu implementieren.

10 Machine Learning (ML)

Machine learning ist die Verwendung von "Modellen", um Funktionen zu erstellen, welche Inputs auf gewünschte outputs abbilden. Generell unterscheidet man zwischen:

1. Regression: gegeben einige Inputwerte, wie lautet der Outputwert? Beispiel: ein Haus hat 5 Schlafzimmer, 1000 Quadratmeter, 3 Badezimmer und ist 50 Meter vom nächsten Bahnhof entfernt, was ist sein Preis (Zahl)?
2. Classification: gegeben einige Inputwerte, zu welcher Gruppe (Kategorie) gehören die Inputwerte? Beispiel: ein Haus hat 5 Schlafzimmer, 1000 Quadratmeter, 3 Badezimmer und ist 50 Meter vom nächsten Bahnhof entfernt, ist es ein teures Haus oder ein günstiges? (Kategorie)

Ein beliebtes Package für viele ML Algorithmen ist sklearn.

10.1 Generelle Prozedur

- Modell auswählen
- Daten einlesesn mit pandas
- Daten in ein Train-set und ein Test-set aufteilen

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.3)
#0.3 of the data will be used in validation
```

- Modell trainieren:

```
from sklearn import linear_model
model = linear_model.LinearRegression()
model.fit(X_train,y_train)
```

- Modell validieren (bewerten, wie gut das Modell performt)

```
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
score = accuracy_score(y_test, y_pred)
```

10.2 Validation Metrics

- Accuracy Score $\frac{\#\ correctly\ classified}{total\ number}$. 1 ist der beste Score, 0 ist der schlechteste.

```
from sklearn.metrics import accuracy_score
```

- R^2 score. 1 is the best score, the more negative the worse.

```
from sklearn.metrics import r2_score
```

- Mean squared error. 0 is the best score. The bigger the worse.

```
from sklearn.metrics import mean_squared_loss
```

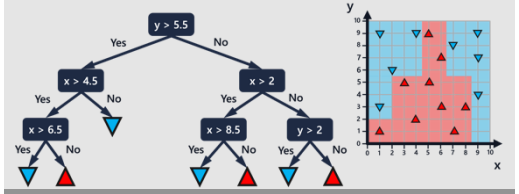
10.3 Classification

10.3.1 Decision Tree Classifier

Gegeben: data X, und labels y. Erstellt einen Decision Tree, um Daten anhand ihrer Merkmale (Features) aufzuteilen.

```
from sklearn.tree import DecisionTreeClassifier
#max_depth specifies the maximum height of the decision
#tree
tree = DecisionTreeClassifier(max_depth=4)
#train the model
tree.fit(X,y)
#calculate the prediction of the model
y_pred = tree.predict(x)
```

Beispiel:



10.4 Regression

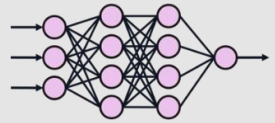
10.4.1 Linear Regression

Gegeben: data X, und labels y. Erstellt ein lineares Modell $w^T x + b_0$ so dass die Loss Function (Unterschied zwischen den Vorhergesagten Werten und den tatsächlichen Werten aufsummieren) für die gegebenen Daten minimiert wird.

```
from sklearn import linear_model
model = linear_model.LinearRegression()
#train the model
model.fit(X,y)
#calculate the prediction of the model
y_pred = tree.predict(x)
```

10.5 Neural Networks

Ein Neural Network ist ein ML-Algorithmus, welcher sehr powerful ist, weil er für Regression und Klassifikation verwendet werden kann. Er besteht aus Layers (Schichten) von Neuronen. Der Output von jedem Neuron wird durch eine Activation Function (Funktion angewendet auf das Resultat der Linearen Funktion in jedem Neuron) non-linear gemacht.



Beispiel Klassifikation:

```
from sklearn.neural_network import MLPClassifier
#hidden_layer_sizes is an array indicating the #number of
neurons in each of the hidden layers,
#activation is the activation function
nn = MLPClassifier(hidden_layer_sizes = [4,4],
activation='relu')
#train the regression model
nn.fit(X,y)
#calculate the prediction of the model
y_pred = nn.predict(x)
```

Beispiel Regression:

```
from sklearn.neural_network import MLPRegressor
nn = MLPRegressor(hidden_layer_sizes = [4,4],
activation='relu')
#train the regression model
nn.fit(X,y)
#calculate the prediction of the model
y_pred = nn.predict(x)
```

10.6 K-Fold Cross Validation

Technik um Machine Learning Modelle zu evaluieren

```
from sklearn.model_selection import
train_test_split, GridSearchCV
from sklearn.neural_network import MLPRegressor
#split data into training and testing data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2)
#create grid
param_grid = {'hidden_layer_sizes':
[(128,),(64,32),(64,128,32)],'activation':
['relu','logistic']}
model = MLPRegressor() #select model
grid_search = GridSearchCV(model, param_grid, cv
= 5) #define GridSearchCV model
grid_search.fit(X_train,y_train) #perform grid
search with cross-validation
y_pred = grid_search.predict(X_test) #predict
using best parameters
```