Python Handbuch

für die Veranstaltungen *Experimentieren und Evaluieren*, *Digitale Codierung* und *Simulation*

Marc Sommerhalder, Andreas Rinkel

Aktuelle Version: 30. Mai 2022

Inhaltsverzeichnis

1 Grundlagen: Programmieren mit Python **6**

1.1 Einleitung 6

1.1.1 Warum Python? 6

1.1.2 Warum (noch) ein Handbuch zu Python? 6

1.1.3 Ressourcen 7

1.2 Python ausführen 8

1.2.1 Lokale Installation 8

1.2.2 Paketinstallationen und virtuelle Umgebungen 8

1.2.3 Jupyter Notebook Server 9

1.2.4 Lokale Jupyter Notebook Installation 10

1.3 Syntax: Eine Kurzreferenz 11

1.3.1 Kommentare 11

1.3.2 Variablen 11

1.3.3 Struktur 11

1.3.4 Datentypen 12

1.3.5 Operatoren 15

1.3.6 Ablaufsteuerung 16

1.3.7 Funktionen 17

1.3.8 Klassen 17

1.3.9 Module verwenden 18

1.4 Dateien lesen und schreiben 19

2 Experimentieren und Evaluieren **20**

2.1 Bibliotheken und Ressourcen 20
2.2 Dateien lesen und Schreiben 22
2.3 Fehlerrechnung 23
2.4 Einfache Datenvisualisierung 24
2.4.1 Balkendiagramm 24
2.4.2 Kurven 25
2.5 Lage- und Streuparameter 27
2.5.1 Parameter berechnen 27
2.5.2 Boxplot 28
2.6 Regression 30
2.6.1 Streudiagramme 30
2.6.2 Kovarianz berechnen 31
2.6.3 Korrelation berechnen 31
2.6.4 Autokorrelation berechnen 32
2.6.5 Regression berechnen 32
2.6.6 Newton Algorithmus 35
2.7 Verteilungen 36
2.7.1 Zufallszahlen erzeugen 36
2.7.2 Testwerte berechnen 36
2.7.3 Konfidenzintervalle berechnen 37
2.7.4 Konfidenzintervalle darstellen 37
2.7.5 Verteilungen darstellen 38
2.7.6 Verteilungen ermitteln 40
2.8 Laplace-Experiment 42

2.9 Hypothesentests . . . 43
2.9.1 Modelle vergleichen mit t-Tests . . . 43
2.9.2 Anteilshypothesen testen . . . 44
2.9.3 Verteilungen testen . . . 44

3 Digitale Codierung **46**
3.1 Ressourcen . . . 46
3.2 Vektor- und Matrizenrechnung . . . 48
3.3 Bits und Bytes . . . 49
3.4 Komprimierung . . . 50
3.4.1 Huffman-Codierung . . . 50
3.4.2 Lempel-Ziv . . . 50
3.5 Verschlüsselung . . . 52
3.5.1 RSA . . . 52
3.6 Blockcodes . . . 53
3.6.1 Hamming-Distanz . . . 53
3.6.2 Hamming-Blockcodes . . . 53
3.6.3 Zyklische Blockcodes . . . 54
3.7 Faltungscodes . . . 55

4 Simulation **56**
4.1 Ressourcen . . . 56
4.2 Zeitdiskrete Variablen . . . 57
4.3 SimPy . . . 58
4.3.1 Statistische Auswertung . . . 59

4.3.2 Animation 60

4.4 Salabim 61

4.4.1 Statistische Auswertung 62

4.4.2 Animation 62

4.5 Mesa 63

1 Grundlagen: Programmieren mit Python

1.1 Einleitung

Das erste Kapitel richtet sich an die, die mit Python programmieren wollen und eine kompakte Darstellung der Python Syntax benötigen. Das Kapitel gliedert sich in Installation, Syntaxübersicht und Dateizugriff.

1.1.1 Warum Python?

Python bietet einige Vorteile:

1. Einfachheit
   (a) Schlanke Syntax
   (b) Leserlichkeit
2. Vielseitigkeit
   (a) prozedural, objektorientiert, funktional
   (b) Interpreter, Just-in-time Compiler, Cython, Jython
3. Verbreitung
   (a) Free und Open Source
   (b) Implementierungen für alle gängigen Betriebssysteme
   (c) Umfangreiche Bibliotheken und Module
   (d) Grosse Community
4. Relevanz
   (a) Data Science
   (b) Machine Learning

1.1.2 Warum (noch) ein Handbuch zu Python?

Das vorliegende Handbuch soll neben einem groben Überblick über die Grundlagen der Python Programmiersprache insbesondere als eine Rezeptsammlungfür die behandelten Themen in den Veranstaltungen Ëxperimentieren und Evaluieren", "Digitale Codierungßowie SSimulation"dienen.

1.1.3 Ressourcen

Da Python mittlerweile eine der beliebtesten Programmiersprachen ist, gibt es eine Vielzahl sehr guter Bücher und Online Ressourcen. Hier eine kleine Auswahl

1. Learn Python 3 the Hard Way von Zed Shaw, ISBN 978-0134692883
2. Automate the Boring Stuff with Python von Albert Sweigart ISBN 978-1593275990, auch online verfügbar
3. The Python Tutorial
4. The Python Documentation
5. Learn Python Programming
6. Real Python Tutorials
7. Python @ Stackoverflow
8. Wissenschaftliches Programmieren in Python, Grundlage für dieses Handbuch

1.2 Python ausführen

1.2.1 Lokale Installation

Python kann lokal über den Software Manager des Betriebssystems (z.B. Homebrew auf Mac, Apt auf Linux) oder als Download von der Python Webseite installiert werden. Alternativ kann auch Anaconda installiert werden, welches bereits viele nützliche Pakete beinhaltet.

Nachdem Python installiert ist, wird es über die Kommandozeile aufgerufen. Python kann entweder interaktiv als Shell verwendet werden:

```
python3

    Python 3.7.3 (default, Jun 17 2019, 15:02:19)
    [GCC 7.4.0] on linux
    Type "help", "copyright", "credits" or "license" for more information.
    >>> 1 + 2
    3
    >>>
```

... oder ein angegebenes Python Skript ausführen:

```
python3 myfile.py
```

Je nach Betriebsystem und Installationsart kann es sein, dass statt *python3* der Befehl *python* oder *py* verwendet werden muss.

Für Python gibt es auch verschiedene IDEs (*Integrated development environment*), z.B. Spyder, PyCharm oder Visual Studio Code. IDE's haben im Vergleich zu den Jupyter Notebooks insbesondere den Vorteil, dass sie Code-Vervollständigung haben. Visual Studio Code unterstützt auch direkt Jupyter Notebooks!

1.2.2 Paketinstallationen und virtuelle Umgebungen

Zusätzliche Pakete können mit *pip* (je nach Installation auch *pip3*) installiert werden. Es hat sich bewährt, für jedes Projekt eine virtuelle Umgebung zu verwenden. Diese können mit dem eingebauten Modul *venv* im aktuellen Verezeichnis so erzeugt werden:

```
python3 -m venv .
```

Anschliessend wird einfach jeweils *python3* und *pip* der virtuellen Umgebung verwendet:

```
env/bin/pip
env/bin/python
```

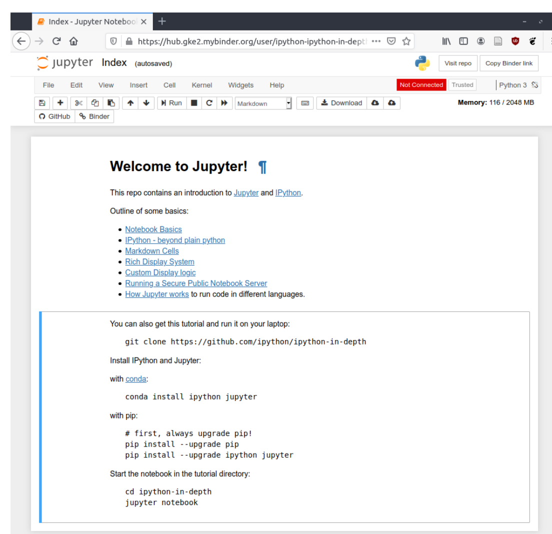
Je nach Betriebsystem und Installationsart kann es sein, dass statt *python3* der Befehl *python* oder *py* verwendet werden muss.

Nun können Pakete mit *pip install …* installiert, mit *pip uninstall …* deinstalliert und mit *pip list* eine Liste installierter Pakete angezeigt werden.

1.2.3 Jupyter Notebook Server

Eine weitere Möglichkeit, um Python Code auszuführen ist die Verwendung von Jupyter Notebooks. Jupyter Notebooks werden im Browser dargestellt und benötigen daher einen Jupyter Server. Ein Vorteil von Jupyter Notebooks ist, dass sie keine lokale Installation von Python benötigen.

Ein Jupyter Notebook besteht aus Blöcken von Text (Markdown) oder Code (Python), wobei jeder Block einzeln ausgeführt werden kann.



Es gibt einige öffentliche Jupyter Notebook Server, z.B.:

1. Jupyter
2. Binder

3. Google Collab

An der OST gibt es die Möglichkeit, den Jupyter Server des IFS zu verwenden: https://jupyter.tschiera.ch (erfordert ein Login für https://gitlab.ost.ch/).

1.2.4 Lokale Jupyter Notebook Installation

Der Jupyter Notebook Server kann auch lokal installiert werden, dazu wird das Paket jupyter (sowie weitere, optionale Pakete wie matplotlib , scipy, pandas etc.) mittels pip installiert. Anschliessend wird der Server mit jupter notebook gestarted und über mittels Browser auf localhost:8888 zugegriffen:

```
python3 -m venv env
env/bin/pip install jupyter
env/bin/pip install matplotlib scipy pandas
env/bin/jupyter notebook
```

1.3 Syntax: Eine Kurzreferenz

Python hat einen eigenen Style Guide. Dieser ist zwar optional, erhöht aber die Lesbarkeit des Codes und wird hier wärmstens empfohlen. Das Skript folgt daher diesen Empfehlungen.

1.3.1 Kommentare

Kommentare beginnen mit einem # und enden am Zeilenende.

```
1 + 2 * 4              # Ich bin ein Kommentar
```

1.3.2 Variablen

Variablen werden mit einem einfachen Gleichheitszeichen definiert. Es können auch mehrere Variablen gleichzeitig definiert werden, indem sie mit einem Komma getrennt werden.

Namen von Variablen, Funktionen, Klassen etc. können beliebig lang sein und aus Buchstaben (UTF-8), Ziffern und Unterstrichen bestehen. Namen dürfen nicht mit einer Ziffer beginnen. Gross- und Kleinbuchstaben werden unterschieden. Namen sollten zugunsten einer hohen Lesbarkeit möglichst aussagekräftig sein!

```
my_variable = 1        # gültig
π = 3.14               # gültig
2_chars = 'ab'         # ungültig
a, b = 1, 2            # gültig
```

1.3.3 Struktur

Python ist zeilenorientiert, d.h. jede Zeile enthält genau eine Anweisung. Eine Anweisung kann aber über mehrere Zeilen gestreckt werden durch das Verwenden von Klammern.

```
summe = a + b + c
summe = (
  a
  + b
  + c
)
```

Blöcke werden durch Einrücken (4 Leerschläge) erzeugt. Anweisungen, welchen einen Block einleiten, enden in der Regel mit einem Doppelpunkt. Blöcke können verschachtelt werden.

```
def abs_diff(a, b):
    # Gib die absolute Differenz zurück
    if a > b:
        return  a - b
    return b - a
```

1.3.4 Datentypen

Alle Objekte haben in Python einen Typ (String, Integer, Float, etc.). Variablen sind hingegen nur typlose Referenzen auf Objekte und können verschiedene Typen referenzieren, die Typen werden erst bei der Verwendung des Objekts geprüft (dynamisches Typsystem).

```
x = 1               # x referenziert einen Integer
x = 'abcd'          # x referenziert nun einen String
x = x + 3           # wird erst zur Laufzeit zu einem Fehler führen (TypError)
type(x)             # <class 'str'>
```

Numerische Typen, Booleans, None In Python gibt es je einen Typ für Ganz- und Kommazahlen (*int*, *float*) sowie einen Typ für komplexe Zahlen (*complex*). Die Standardbibliothek definiert zudem noch Kommazahlen mit einer spezifischen Genauigkeit (*decimal.Decimal*). Zudem gibt es noch einen speziellen Typ mit nur einem Wert, welcher verwendet wird, falls eine Variable keinen Wert hat (*None*), sowie einen Typ für wahr/falsch (*bool*).

```
type(1)             # <class 'int'>
type(1.0)           # <class 'float'>
type(1j+1)          # <class 'complex'>
type(None)          # <class 'NoneType'>
type(True)          # <class 'bool'>

from decimal import Decimal
Decimal('10.01')
```

Zeichenketten Zeichenketten (*str*) enthalten Zeichen und werden mit (einfachem oder doppeltem) Hochkomma definiert. Die Standardbibliothek enthält eine Vielzahl von String-Funktionen.

Zeichenketten können mit *f-Strings* formatiert werden. Alle gefundenen Ausdrücke innerhalb von geschweiften Klammern werden dabei interpretiert.

```
a = 'Ein String'
b = "Auch ein String"
c = """
  Ein String über
  mehrere Zeilen
"""

'abcd'.startswith('a')        # True

a = 4
b = 3
f'{a} * {b} = {a*b}'          # '4 * 3 = 12'
```

Listen Listen (*list*) enthalten beliebige Objekte gleichen oder unterschiedlichen Typs, sie werden mit eckigen Klammern definiert. Listen können auch geschachtelt werden.

```
a = [1, 2, '3']
b = [[1, 2], [a, a]]
```

Tupel Tupel (*tuple*) sind wie Listen, aber unveränderbar und werden mit runden Klammern definiert.

```
a = (1, 2, (3, 4))
b = (,)                    # leeres Tuple
```

Indexierung Auf die Elemente von Zeichenketten, Listen und Tupeln kann mittels eckigen Klammern zugegriffen werden. Dabei kann entweder ein einzelner Index oder ein Bereich von / bis, getrennt mit einem Doppeltpunkt angegeben werden.

```
'Hallo Welt'[0]            # 'H'
'Hallo Welt'[1]            # 'a'
'Hallo Welt'[-1]           # 't'
'Hallo Welt'[2:-1]         # 'llo We'
```

Sets Sets (*set*) entsprechen Mengen, sie enthalten keine Duplikate und werden ohne Reihenfolge gespeichert und sind damit auch nicht indizierbar. Sets werden mit geschweiften Klammern definiert. Python definiert spezielle Mengenoperationen wie Schnittmenge oder Vereinigungsmenge.

```
a = {1, 2}
b = set()
{1, 2} & {2, 3}         # Schnittmenge {2}
{1, 2} | {2, 3}         # Vereinigungsmenge {1, 2, 3}
```

Dictionaries Dictionaries (*dict*) beinhalten Schlüssel-Werte-Paare und werden mit geschweiften Klammern definiert. Als Schlüssel kann jeder Typ verwendet werden, von welchem ein Hash erzeugt werden kann (mittels eingebauter hash-Funktion). Werte können von unterschiedlichem Typ sein. Auf die Werte von Dictionaries kann mit dem einer eckigen Klammer zugegriffen werden. Die Reihenfolge wird erst mit Python 3.8 gespeichert.

Achtung: Sowohl Dictionaries als auch Sets benutzen geschweifte Klammern! Bei dicts werden Schlüssel-Werte-Paare mit einem Doppelpunkt definiert, bei Sets nur Werte. Leere geschweifte Klammen definieren ein leeres Dictionary.

```
a = {
  '1': (1, 2, 3),       # Schlüssel '1' enthält ein Tupel
  2: {1.0: 2.0}         # Schlüssel 2 enthält ein geschachteltes Dictionary
}
a['1']                  # Zugriff auf Schlüssel '1': (1, 2, 3)
a[2][1.0]               # Zugriff auf verschachteltes Dictionary: 2.0
b = {}                  # leeres Dictionary
```

Comprehension Listen, Tupel, Sets und Dictionaries können auch mittels For-Schleifen-ähnlicher Syntax (vergleiche unten) erzeugt werden. Es kann ein *if* verwendet werden, um Elemente nur unter bestimmten Bedingungen hinzuzufügen.

```
text = 'Hallo Welt'

# Alle Zeichen als Liste
# ['H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't']
[char for char in text]

# Alle Kleinbuchstaben ohne Duplikate
# {'o', 't', 'e', 'a', 'l'}
{char for char in text if char.islower()}

# Dictionary mit erstem Auftreten der Kleinbuchstaben
# {1: 'a', 2: 'l', 4: 'o', 7: 'e', 9: 't'}
{text.index(char): char for char in text if char.islower()}
```

1.3.5 Operatoren

Mathematischen Operatoren Neben den typischen mathematischen (*+*, *-*, ***, */*) gibt es auch einen für Ganzzahldivisionen (*//*), Modulo (*%*) sowie Potenz (****).

```
2 + 3                  # Addition          = 5
2 - 3                  # Subtraktion       = -1
2 * 3                  # Multiplikation    = 6
2 / 3                  # Division          = 0.6666666666666666
2 // 3                 # Ganzzahldivision  = 0
2 % 3                  # Modulo            = 2
2 ** 3                 # Potenz            = 8
```

Logische Operatoren Logische Operatoren umfassen und (*and*), oder (*or*), nicht (*not*), sowie exklusiv oder (*ˆ*). Zudem gibt es auch noch logische Operatoren auf Bit-Ebene (*&*, *|*, *˜*, *<<*, *>>*).

```
True and False         # und             = False
2 & 3                  # und             = 2
True or False          # oder            = True
2 | 4                  # oder            = 6
not True               # nicht           = False
~2                     # nicht           = -3
True ^ True            # eklusiv oder    = False
2 ^ 6                  # eklusiv oder    = 4
2 << 1                 # links schieben  = 4
2 >> 4                 # rechts schieben = 0
```

Vergleichsoperatoren Vergleichsoperatoren umfassen Gleichheit (*==*, *is*), Ungleichheit (*!=*, *is not*), kleiner ¡(*<*, *<=*), sowie grösser (*>*, *>=*). Der Operator *is* wird für *None* und *bool*-Typen verwendet.

```
1 == 2                 # Gleichheit      = False
True is False          # Gleichheit      = False
1 != 2                 # Ungleichheit    = True
1 is not None          # Ungleichheit    = True
1 > 2                  # grösser         = False
3 >= 2                 # grösser gleich  = True
2 > 2                  # kleiner         = False
2 <= 3                 # kleiner gleich  = True
```

Mitgliedsschaftsoperatoren Mitgliedsschaftsoperatoren umfassen in (*in*) und nicht in (*not in*).

```
1 in [1, 2]      # = True
```

1.3.6 Ablaufsteuerung

if In Python bestehen *if* aus mindestens einem *if*-Block, mehreren optionalen *elif*-Blöcken sowie einem optionalen *else*-Block.

```
if x > 0:
    # Block, falls Bedingung 'x > 0' zutrifft
    print('x ist positiv')
elif x < 0:
    # Block, falls Bedingung 'x < 0' zutrifft
    print('x ist negativ')
else:
    # Block, falls keine der Bedingungen zutrifft.
    print('x ist 0')
```

while Bei einem *while* wird der Block so lange ausgeführt, wie die Bedingung wahr ist oder ein *break* gefunden wird. Bei einem *continue* wird sofort der nächste Durchlauf begonnen.

```
while x > a:
    x = x - b
    if x < a:
      break          # Schleife abbrechen
    if x < c:
      continue       # Rest des Blockes überspringen
    x = x - d
```

for Bei einem *for* wird der Block für jedes Element im gegebenen iterierbaren Objekt wiederholt. Iterierbare Objekte sind zum Beispiel die *range*-Funktion, Strings, Listen, Tuples, Sets oder die Schlüssel von Dictionaries.

Bei einem *break* wird abgebrochen, bei einem *continue* sofort der nächste Durchlauf begonnen.

```
for x in (1, 2, 3, 4):
    print(x)
    if x > 2:
      continue       # Rest des Blockes überspringen
    if x < 3:
      break          # Schleife abbrechen
```

1.3.7 Funktionen

Funktionen werden mit einem *def* definiert und mit runden Klammern aufgerufen.

Funktionen geben den mit *return* definierten Wert zurück oder *None*, falls klein *return* definiert ist.

Python kennt positionelle (*args*) und Schlüsselwort-Argumente (*kwarg*). Schlüsselwort-Argumente müssen beim Funktionsaufruf immer nach den positionellen Argumenten angegeben werden. Schliesslich können Argumente auch einen Default-Wert haben (=), Argumente mit Default-Werten müssen nach Argumenten ohne Default-Werte definiert werden.

Globale Variablen können innerhalb von Funktionen gelesen werden, sollte aber nicht verändert werden.

```
e = 4

def sum4(a, b, c=0, d=0):
    return a + b + c + d + e

summe(1, 2)
summe(1, 2, 3)
summe(1, 2, d=3)
summe(d=1, b=2, a=1)
```

1.3.8 Klassen

Klassen werden mit einem *class* definiert und mit einer runden Klammer angelegt.

Funktionen innerhalb Klassen heissen in Python Methoden und besitzen als ersten Argument immer *self*: die Instanz / das Objekt.

```
class Person:
    def __init__(self, second_name, first_name):
      self.second_name = second_name
      self.first_name = first_name

    def hallo(self):
      return f'Hallo {self.first_name} {self.second_name}!'

jan = Person('Rothstein', 'Jan')
jan.hallo()       # 'Hallo Jan Rothenstein!'

sophia = Person('Kuhn', 'Sophia')
sophia.hallo()    # 'Hallo Sophia Kuhn!'
```

Klassen können von anderen Klassen erben, indem die Basisklasse nach dem Klassennamen in Klammern gesetzt wird. Eine Klasse kann von mehreren anderen Klassen erben - dazu werden einfach alle Basisklassen kommagetrennt in den Klammern aufgeführt. Um eine Funktion der Basisklasse aufzurufen, kann *super* verwendet werden.

```
class AlbertEinstein(Person):
    def __init__(self):
      super().__init__('Einstein', 'Albert')

AlbertEinstein().hallo()  # Hallo Albert Einstein!
```

1.3.9 Module verwenden

Module stellen Variablen, Funktionen, Klassen, Objekte etc. zur Verfügung. Diese müssen vor der Verwendung importiert werden.

```
from random import randint

a = randint(1, 10)  # Erzeuge eine zufällige Zahl zwischen 1 und 10
```

Python besitzt eine Vielzahl eingebauter Module in der Standardbibliothek. Daneben gibt es noch eine Vielzahl Pakete mit Modulen von Drittanbietern auf pypi, welche über das Kommandozeilentool pip installiert werden können. Da man oft verschiedene Projekte mit unterschiedlichen Paketen hat und diese Pakte selbst wieder unterschiedliche Abhängigkeiten haben, hat sich die Verwendung von virtuellen Umgebungen bewährt. Eine virtuelle Umgebung installiert alle Pakete innerhalb eines Verzeichnisses, anstatt global.

```
python3 -m venv env          # Umgebung erstellen
env/bin/pip install numpy    # Numpy lokal installieren
env/bin/python               # Lokale Python Version mit Numpy ausführen
```

1.4 Dateien lesen und schreiben

Um Dateien zu lesen und zu schreiben, müssen diese zuerst geöffnet werden mittels *open* unter Angabe des Modus ('r' oder keine Angabe: Lesen, 'w': Lesen und Schreiben). Damit die Datei nach dem Zugriff auch in Fehlerfällen wieder freigegeben wird, wird ein *with* verwendet und der Zugriff innerhalb des Blockes gemacht.

Text Text kann direkt mittels *write* geschrieben lesen, Zeilen lesen ist möglich, indem über das erhaltene Dateiobjekt iteriert wird.

```
# Schreiben
with open('text.txt', 'w') as file:
    file.write('Hallo\n')
    file.write('Welt!')

# Lesen
with open('text.txt') as file:
    for line in file:
        print(line)
```

CSV Eine einfache Möglichkeit, Daten zu speichern sind Comma-seperated Values (CSV) Dateien. Dazu gibt es in Python das CSV Modul. Achtung, da CSV ein Textformat ist, müssen Integer, Float, etc. zuerst konvertiert werden.

```
from csv import reader, writer

# Schreiben
with open('personen.csv', 'w') as file:
    csv = writer(file)
    csv.writerow(('Rothenstein', 'Jan', 22))
    csv.writerow(('Kuhn', 'Sophia', 48))

# Lesen
with open('personen.csv') as file:
    csv = reader(file)
    for second_name, first_name, age in csv:
        print(f'{first_name} {second_name}, {int(age)}')
```

2 Experimentieren und Evaluieren

Für die Veranstaltung Experimentieren und Evaluieren werden viele Formeln der Statistik verwendet. Bisher haben Sie in kleinen Beispielen die Anwendung der Formeln geübt. Bei grossen Datenmengen ist das manuelle Bearbeiten äusserst mühsam, deshalb bieten wir Ihnen im Folgenden einen Weg an, wie Sie auch grössere Datenmenge mit Python bearbeiten und analysieren können.

Als nächste stellen wir die Bibliotheken kurz vor, die Sie zur Bearbeitung einbinden müssen.

2.1 Bibliotheken und Ressourcen

Matplotlib Matplotlib ist eine Visualisierungsbibliothek, mit der verschiedene Graphen gezeichnet werden können.

Installation: `pip install matplotlib`

Mehr Informationen: https://matplotlib.org

seaborn seaborn ist eine Visualisierungsbibliothek, welche auf *Matplotlib aufbaut*.

Installation: `pip install matplotlib`

Mehr Informationen: https://matplotlib.org

NumPy NumPy ist Bibliothek für die Manipulation numerischer Vektoren und Matrizen.

Installation: `pip install numpy`

Mehr Informationen: https://numpy.org

Pandas Pandas ist Bibliothek für Datenanalyse und Manipulation.

Installation: `pip install pandas`

Mehr Informationen: https://pandas.pydata.org

SciPy SciPy enthält eine Vielzahl wissenschaftlicher Funktionen, unter anderem eine grosse Auswahl statistischer Funktionen.

Installation: `pip install scipy`

Mehr Informationen: https://www.scipy.org

statsmodels statsmodels enthält eine Vielzahl statistischer Funktionen.

Installation: `pip install statsmodels`

Mehr Informationen: https://www.statsmodels.org

distfit distfit ist eine kleine Bibliothek für das Testen von Verteilungen.

Installation: `pip install distfit`

Mehr Informationen: https://erdogant.github.io/distfit

error-solver error-solver ist eine kleine Bibliothek für Fehlerrechnungen.

Installation: `pip install error-solver[all]`

Mehr Informationen: https://error-solver.readthedocs.io

error-limits error-limits ist eine kleine Bibliothek, um beliebige Berechnungen mit Fehlergrenzen zu berechnen.

Installation: `pip install error-limits`

Mehr Informationen: https://pypi.org/project/newton-polynomial/

newton-polynomial newton-polynomial ist eine kleine Newton-Algorithmus-Bibliothek.

Installation: `pip install newton-polynomial`

Mehr Informationen: https://pypi.org/project/newton-polynomial/

2.2 Dateien lesen und Schreiben

Grosse Mengen von Daten können mit *NumPy* z.B. mittels *loadtxt* und *savetxt* gelesen und geschrieben werden.

```
import numpy as np

random = np.random.default_rng()
x = random.uniform(0, 1, 1000)
y = random.triangular(0, 0.5, 1, 1000)

# Daten schreiben
np.savetxt('data.txt', (x, y))

# Daten schreiben
x, y = np.loadtxt('data.txt')
```

Auch mit *Pandas* können grosse Mengen an Daten sehr einfach mittels *to_csv* und *read_csv* gelesen und geschrieben werden.

```
import pandas as pd

data = pd.DataFrame({
    'x': random.uniform(0, 1, 1000),
    'y': random.triangular(0, 0.5, 1, 1000)

})

# Daten schreiben
data.to_csv('data.csv', index=False)

# Daten lesen
data = pd.read_csv('data.csv')
data.head()
```

2.3 Fehlerrechnung

Absolute Fehler können mittels partieller Differentiation mit dem Paket *error-solver* berechnet werden.

```
from error_solver import ErrorSolver

equations = [
  'W = m * g * h / 2 + D * h**2 / 8'
]
values = {
    'm': 0.3,
    'h': 0.3,
    'D': 10.0,
    'g': 9.81,
    'W': 0.3 * 9.81 * 0.3 / 2 + 10.0 * 0.3**2 / 8
}
errors = {
    'm': 0.002,
    'h': 0.002,
    'D': 0.5,
    'g': 0.01,
}

solver = ErrorSolver(equations)
solver.solve(values, errors)['error']['W']   # 0.013461
```

Fehler können auch numerisch mit dem Paket *error-limits* berechnet werden. Dazu wird die Berechnung in einer mit *imprecise*-dekorierten Funktion gemacht und dieser optional *Ranges* statt Zahlen übergeben.

```
from error_limits import Range, imprecise

@imprecise
def calculate(m, g, h, d):
    return m * g * h / 2 + d * h**2 / 8

w = calculate(
    m=Range(0.3 - 0.002, 0.3 + 0.002),
    g=Range(9.81 - 0.01, 9.81 + 0.01),
    h=Range(0.3 - 0.002, 0.3 + 0.002),
    d=Range(10.0 - 0.5, 10.0 + 0.5),
)
w                         # [0.5405943499999999, 0.56751689]
(w.upper - w.lower) / 2   # 0.013461270000000008
```

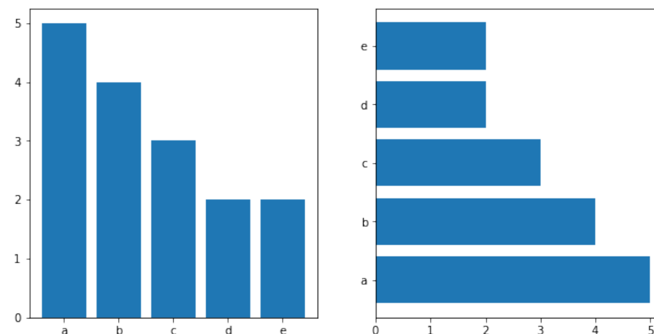
2.4 Einfache Datenvisualisierung

Neben den hier aufgeführten Visualisierungen finden sich weiter unten zum Beispiel noch Boxplots, Streudiagramme und Histogramme.

2.4.1 Balkendiagramm

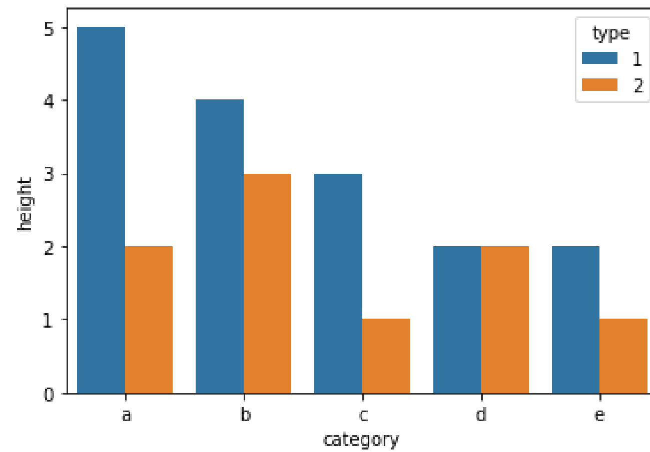
Balkendiagramme können mit *matplotlib* entweder mittels *bar* (vertikal) oder *hbar* (horizontal) erstellt werden.

```
import matplotlib.pyplot as plt

category = ('a', 'b', 'c', 'd', 'e')
height = (5, 4, 3, 2, 2)

plt.subplots(figsize=(10, 5))
plt.subplot(1, 2, 1).bar(category, height)
plt.subplot(1, 2, 2).barh(category, height)
```



Mit *seaborn* können zudem Balkendiagramme sehr einfach für mehere Variablen mittels *barplot* erstellt werden. Dazu werden die Daten in einer *Pandas*-Tabelle erfasst und mittels *hue*-Parameter die Variable für die Gruppierung gesetzt.

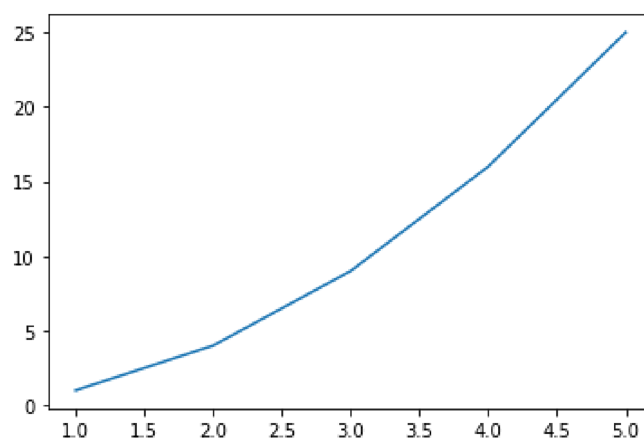
```
import pandas as pd
import seaborn as sns

data = pd.DataFrame({
    'category': 2 * ('a', 'b', 'c', 'd', 'e'),
    'height': (5, 4, 3, 2, 2) + (2, 3, 1, 2, 1),
    'type': (1, 1, 1, 1, 1) + (2, 2, 2, 2, 2)
})
sns.barplot(x='category', y='height', hue='type', data=data)
```



2.4.2 Kurven

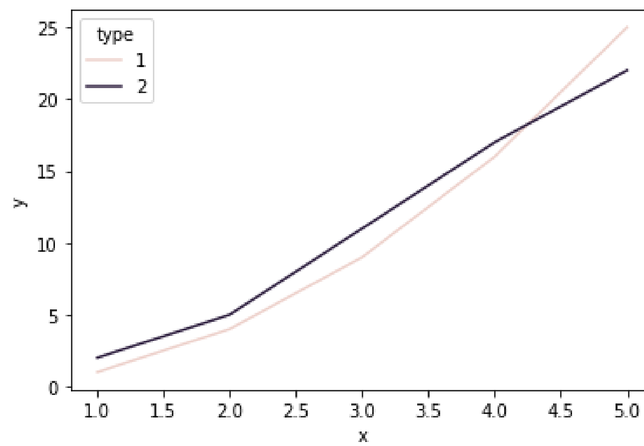
Kurven können mit *matplotlib* mittels *plot* erstellt werden.

```
import matplotlib.pyplot as plt

x = (1, 2, 3, 4, 5)
y = (1, 4, 9, 16, 25)

plt.plot(x, y)
```



Auch hier kann wieder mit *seaborn* mittels *lineplot* Kurven sehr einfach für mehrere Variablen erstellt werden. Wie oben werden dazu die Daten in einer *Pandas*-Tabelle erfasst und mittels *hue*-Parameter die Variable für die Gruppierung gesetzt.

```
import pandas as pd
import seaborn as sns

data = pd.DataFrame({
    'x': 2 * (1, 2, 3, 4, 5),
    'y': (1, 4, 9, 16, 25) + (2, 5, 11, 17, 22),
    'type': (1, 1, 1, 1, 1) + (2, 2, 2, 2, 2)
})
sns.lineplot(x='x', y='y', hue='type', data=data)
```



2.5 Lage- und Streuparameter

2.5.1 Parameter berechnen

Parameter können entweder mit *NumPy* oder *SciPy* berechnet werden, einige Parameter auch direkt mit der Standardbibliothek von Python.

Achtung: Es gibt verschiedene Arten, die Quartile und den Quartilsabstand zu berechnen[1], *NumPy* und *SciPy* verwenden standardmässig unterschiedliche und beide unterscheiden sich von der in der Vorlesung verwendeten.

```
import numpy as np
from scipy import stats

x = (1, 2, 3, 4, 5, 6)

# Lageparameter
stats.mode(x)                                   # (kleinster) Modus
np.median(x)                                    # Median
stats.mstats.mquantiles(x)                      # 1.-3. Quartile
np.quantile(x, 0.25)                            # 1. Quartil, andere Methode
np.median(sorted(x)[0:len(x) // 2])             # 1. Quartil, unsere Methode
np.mean(x)                                      # arithmetisches Mittel
stats.hmean(x)                                  # harmonisches Mittel
stats.gmean(x)                                  # geometrisches Mittel

# Streuparameter
min(x)                                          # Minimum
max(x)                                          # Maximum
max(x) - min(x)                                 # Spannweite
np.quantile(x, 0.75) - np.quantile(x, 0.25)     # zentraler Quartilsabstand
stats.median_abs_deviation(x)                   # mittlere absolute Abweichung
np.var(x)                                       # Varianz
np.std(x)                                       # Standardabweichung
np.var(x, ddof=1)                               # Stichprobenvarianz
stats.variation(x)                              # Variationskoeffizient
```

[1] Siehe https://mathworld.wolfram.com/Quartile.html für einen Vergleich.

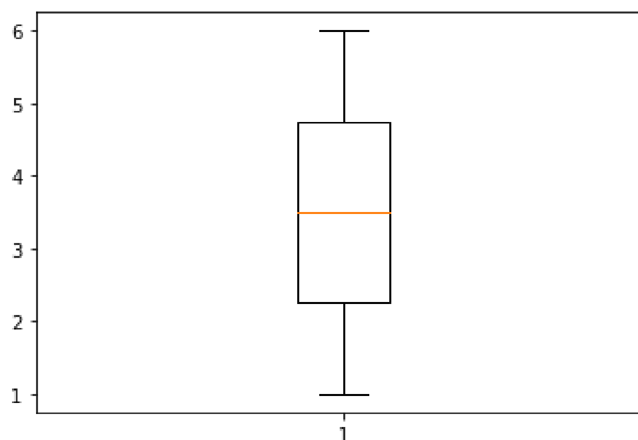
Eine einfache Möglichkeit, die wichtigsten Parameter für eine Menge von Daten zu erhalten ist *describe* von *Pandas*:

```
import pandas as pd

x = pd.Series((1, 2, 3, 4, 5, 6))
x.describe()

#count    6.000000
#mean     3.500000
#std      1.870829
#min      1.000000
#25%      2.250000
#50%      3.500000
#75%      4.750000
#max      6.000000
```

2.5.2 Boxplot

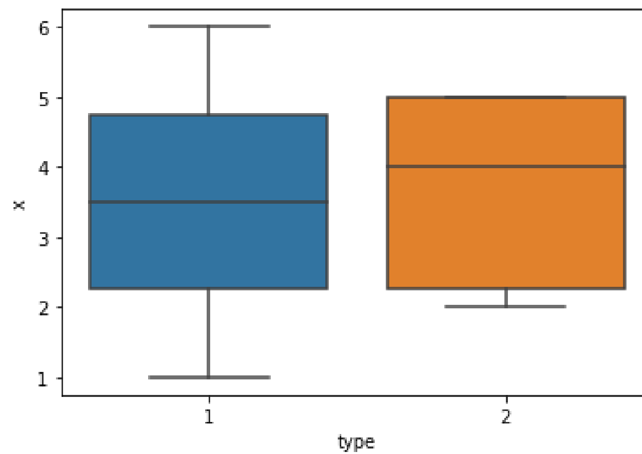
Boxplots können mit *Matplotlib* mittels *boxplot* gemacht werden. Indem *whis* auf Unendlich gesetzt wird, kann verhindert werden, dass das Minimum und Maximum als Ausreisser dargestellt werden.

```
import matplotlib.pyplot as plt

x = (1, 2, 3, 4, 5, 6)

plt.subplots()
plt.boxplot(x, whis=float('inf'))
```



Auch bei Boxplots kann wieder mit *seaborn* mittels *boxplot* sehr einfach gruppiert werden. Wie oben werden dazu die Daten in einer *Pandas*-Tabelle erfasst und mittels x-Parameter die Variable für die Gruppierung gesetzt.

```
import pandas as pd
import seaborn as sns

data = pd.DataFrame({
    'x': (1, 2, 3, 4, 5, 6) + (2, 2, 3, 5, 5, 5),
    'type': (1, 1, 1, 1, 1, 1) + (2, 2, 2, 2, 2, 2)
})
sns.boxplot(x='type', y='x', data=data)
```

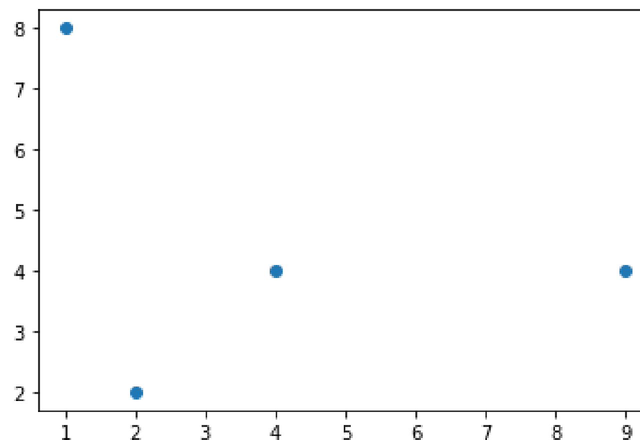


2.6 Regression

2.6.1 Streudiagramme

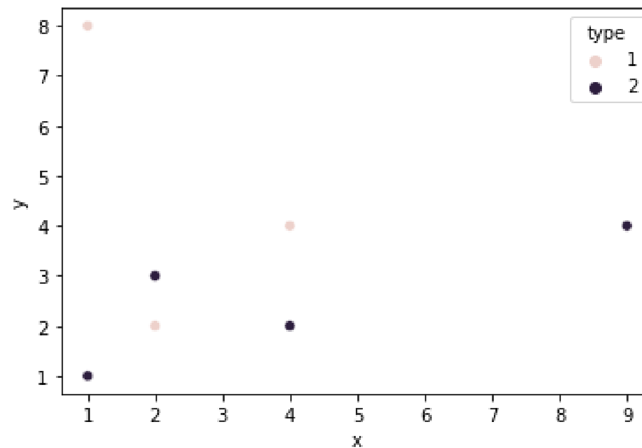
Streudiagramme können mit *matplotlib* mittels *scatter* erzeugt werden.

```
from matplotlib import pyplot as plt

x = (1, 4, 2, 9)
y = (8, 4, 2, 4)
plt.scatter(x, y)
```



Auch bei Streudiagrammen kann wieder mit *seaborn* mittels *scatterplot* sehr einfach gruppiert werden. Wie oben werden dazu die Daten in einer *Pandas*-Tabelle erfasst und mittels *hue*-Parameter die Variable für die Gruppierung gesetzt.

```
import pandas as pd
import seaborn as sns

data = pd.DataFrame({
    'x': 2 * (1, 4, 2, 9),
    'y': (8, 4, 2, 4) + (1, 2, 3, 4),
    'type': (1, 1, 1, 1) + (2, 2, 2, 2)
})
sns.scatterplot(x='x', y='y', hue='type', data=data)
```



2.6.2 Kovarianz berechnen

Die Kovarianz kann mit *numpy* mittels *cov* berechnet werden. Das Resultat ist eine 2x2-Matrix mit den einzelnen Varianzen in der Hauptdiagonalen und der Kovarianz in der Gegendiagonalen.

Achtung: der Parameter *bias* muss auf *True* gesetzt werden.

```
import numpy as np

x = np.array((1, 2, 3, 4, 5, 6, 7))
y = np.array((7, 6, 6, 4, 2, 2, 1))

cov = np.cov(x, y, bias=True)

cov[0][0]   # Varianz x == np.var(x)
cov[1][1]   # Varianz y == np.var(y)
cov[0][1]   # Kovarianz == cov[1][0]
```

2.6.3 Korrelation berechnen

Der Korrelationskoeffizient kann mit *numpy* mittels *corrcoef* berechnet werden. Das Resultat ist eine 2x2-Matrix mit dem Korrelationskoeffizient in der Gegendiagonalen.

```
import numpy as np

x = np.array((1, 2, 3, 4, 5, 6, 7))
y = np.array((7, 6, 6, 4, 2, 2, 1))

np.corrcoef(x, y)[0][1] # == np.corrcoef(x, y)[1][0]
```

2.6.4 Autokorrelation berechnen

Die (normierte) Autokorrelation kann mit *statsmodels* mittels *acf* berechnet werden.

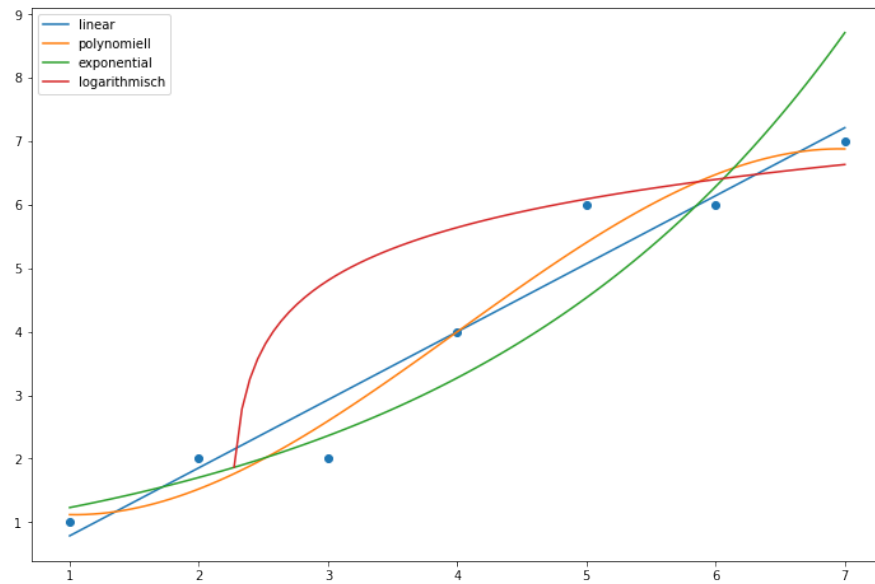
```
from statsmodels.tsa.stattools import acf

acf((1, 2, 3, 4, 5, 6), nlags=2)  # [1, 0.5, 0.057]
```

2.6.5 Regression berechnen

Regressionen können sehr einfach mit *NumPy* mittels *polyfit* berechnet werden. Die berechneten Koeffizienten können direkt verwendet werden, um ein *Polynomial* zu erstellen, welches konkrete y-Werte berechnet, falls man es mit einem x-Wert aufruft.

Nicht-lineare Regressionsmodelle können erstellt werden, indem wie gewohnt die y-Werte vor und nach der Berechnung entsprechend transformiert werden.

```
import numpy as np
import numpy.polynomial.polynomial as polynomial
import matplotlib.pyplot as plt

x = np.array((1, 2, 3, 4, 5, 6, 7))
y = np.array((1, 2, 2, 4, 6, 6, 7))

lin_model = polynomial.Polynomial(polynomial.polyfit(x, y, 1))
pol_model = polynomial.Polynomial(polynomial.polyfit(x, y, 3))
exp_model = polynomial.Polynomial(polynomial.polyfit(x, np.log(y), 1))
log_model = polynomial.Polynomial(polynomial.polyfit(x, np.exp(y), 1))

x_test = np.linspace(1, 7, 100)

plt.subplots(figsize=(12, 8))
plt.scatter(x, y)
plt.plot(x_test, lin_model(x_test), label='linear')
plt.plot(x_test, pol_model(x_test), label='polynomiell')
plt.plot(x_test, np.exp(exp_model(x_test)), label='exponential')
plt.plot(x_test, np.log(log_model(x_test)), label='logarithmisch')
plt.legend()
np.polynomial.Polynomial(np.polyfit(x, y, 1))
```



Auch mit *statsmodels* können Regressionen einfach durchgeführt werden, wobei hier noch viele weitere Details ersichtlich sind.

```
import pandas as pd
import statsmodels.api as sm

data = pd.DataFrame({
    'x': (1, 2, 3, 4, 5, 6, 7),
    'y': (7, 6, 6, 4, 2, 2, 1)
})

# Lineare Regression mit (Ordinary Least Squares)
model = sm.OLS.from_formula('y ~ x', data=data)
result = model.fit()
result.summary()
```

OLS Regression Results

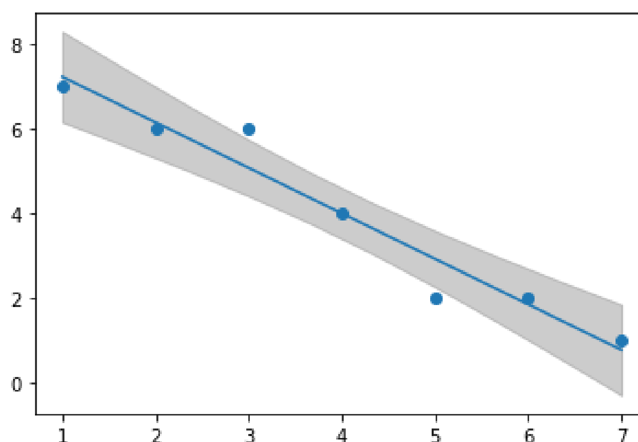
Dep. Variable:	y	**R-squared:**	0.945
Model:	OLS	**Adj. R-squared:**	0.934
Method:	Least Squares	**F-statistic:**	86.54
Date:	Sun, 09 May 2021	**Prob (F-statistic):**	0.000242
Time:	10:36:09	**Log-Likelihood:**	-5.2885
No. Observations:	7	**AIC:**	14.58
Df Residuals:	5	**BIC:**	14.47
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	8.2857	0.515	16.086	0.000	6.962	9.610
x	-1.0714	0.115	-9.303	0.000	-1.367	-0.775

Omnibus:	nan	**Durbin-Watson:**	2.170
Prob(Omnibus):	nan	**Jarque-Bera (JB):**	0.000
Skew:	-0.000	**Prob(JB):**	1.00
Kurtosis:	3.028	**Cond. No.**	10.4

Zusätzlich können mit *statsmodels* auch das Konfidenzband der Regression dargestellt werden!

```
import seaborn as sns
from statsmodels.sandbox.predict_functional import predict_functional

y_test, co, x_test = predict_functional(result, 'x', alpha=0.05)
plot = sns.lineplot(x_test, y_test)
plot.fill_between(x_test, co[:, 0], co[:, 1], color='grey', alpha=0.4)
plot.scatter(data['x'], data['y'])
```



2.6.6 Newton Algorithmus

Ein Polynom kann mit *netwon-polynomial* mittels *polynomial* erstellt werden.

```
import numpy as np
from newton_polynomial import polynomial
from sympy import lambdify, symbols

x = np.array([1, 2, 3, 4, 5, 6])
y = np.array([0, 5, 22, 57, 116, 205])

model = polynomial(x, y)                          # 1.0 x^3 - 2.0 x + 1.0
predict = lambdify(symbols('x'), model, 'numpy')
predict(0)                                        # = 1
```

2.7 Verteilungen

2.7.1 Zufallszahlen erzeugen

Zufallszahlen können zum Beispiel entweder mit der Standardbibliothek *random*, mit *NumPy* oder mit *SciPy* erzeugt werden. Mit *NumPy* und *SciPy* hat man insbesondere mehr Verteilungen zur Auswahl sowie die Möglichkeit, mehrere Zufallszahlen einer Verteilung gleichzeitig zu erzeugen.

```
import random

# Ganzzahlen
random.randint(1, 10)             # Uniform zwischen 1 und 10

# Auswahl
random.choice([True, False])      # Zufällige Auswahl
random.sample(range(1, 100), 4)   # Zufällig 4 auswählen

# Kommazahlen
random.uniform(0, 1)              # Uniform zwischen 0 und 1
random.triangular(0, 1, 0.5)      # Triangular zwischen 0 und 1, Modus bei 0.5
random.expovariat(0.1)            # Exponential mit lambda = 0.1
random.gauss(0, 1)                # Normalverteilt mit mu=0 und sigma=1
```

```
from numpy.random import default_rng
random = default_rng()

# Ganzzahlen
random.integers(1, 10)            # Uniform zwischen 1 und 9
random.poisson(10)                # Poisson mit lambda = 10
random.binomial(1000, 0.5)        # Binomial mit n = 1000 und p=0.5

# Auswahl
random.choice([True, False])      # Zufällige Auswahl
random.choice(range(1, 100), 4)   # Zufällig 4 auswählen

# Kommazahlen
random.uniform(0, 1)              # Uniform zwischen 0 und 1
random.triangular(0, 1, 0.5)      # Triangular zwischen 0 und 1, Modus bei 0.5
random.exponential(0.1)           # Exponential mit lambda = 0.1
random.normal(0, 1)               # Normalverteilt mit mu=0 und sigma=1
random.chisquare(2)               # Chiquadrat mit 2 Freiheitsgraden
```

2.7.2 Testwerte berechnen

z-, t- und χ^2-Werte können mit *SciPy* mittels *ppf* erzeugt werden.

```
from scipy import stats

stats.norm.ppf(0.99)        # z
stats.t.ppf(0.99, 1)        # t
stats.chi2.ppf(0.99, 1)     # chi^2
```

2.7.3 Konfidenzintervalle berechnen

Beidseitige Konfidenzintervalle können mit *SciPy* mittels *interval* der entsprechenden Verteilung berechnet werden, einzelne Werte können wie oben mittels *ppf*.

Wahrscheinlichkeiten für bestimmte Verteilungswerte können mittels der kumulativen Verteilungsfunktion (*cdf*) berechnet werden mit der optionalen Angabe des Mittelwertes (*loc*) und der Varianz (*scale*). Da t-Werte beliebig genau und schnell berechnet werden können, empfiehlt sich, wenn immer möglich diese zu verwenden.

```
from scipy import stats

stats.norm.interval(0.99)     # 99%, beidseitig
stats.t.interval(0.99, 10)    # 99%, beidseitig, 10 Freiheitsgrade
stats.chi2.interval(0.99, 10) # 99%, beidseitig, 10 Freiheitsgrade

# p: a < x < b
stats.norm.cdf(0.5) - stats.norm.cdf(-0.5)
stats.t.cdf(1.8, 10) - stats.t.cdf(-1.4, 10)
stats.chi2.cdf(20, 10) - stats.chi2.cdf(15, 10)

# p: x > a
1 - stats.norm.cdf(0.5)
1 - stats.t.cdf(2.2, 10)
1 - stats.chi2.cdf(9, 10)

# p: x < b
stats.norm.cdf(0.5)
stats.t.cdf(-0.9, 10)
stats.chi2.cdf(9, 10)

# p: x < 0.95, mu=10, sigma=1.5, n=50
stats.t.cdf(9.5, df=50-1, loc=10, scale=1.5/(50**0.5))
```

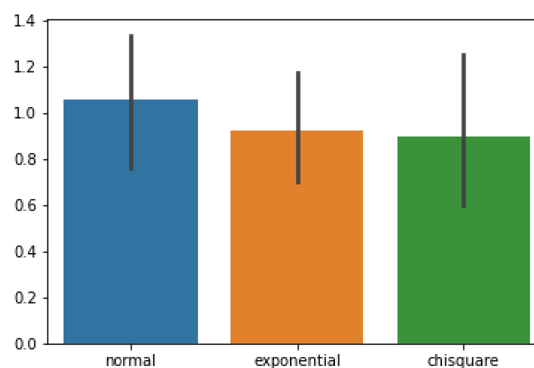
2.7.4 Konfidenzintervalle darstellen

seaborn erlaubt, Balkendiagramme für einen Parameter mit Konfidenzintervallen darzustellen. Hierzu wird mit *estimator* eine Funktion für die Parameterberechnung (Default: Mittwert) und mit *ci* das Konfidenzintervall übergeben.

Achtung: *seaborn* berechnet hierbei die Konfidenzintervalle nur auf Basis des

Datensatzes (mittels Resampling/Bootstrapping), nicht wie gewohnt mit einem gegebenen Standardfehler!

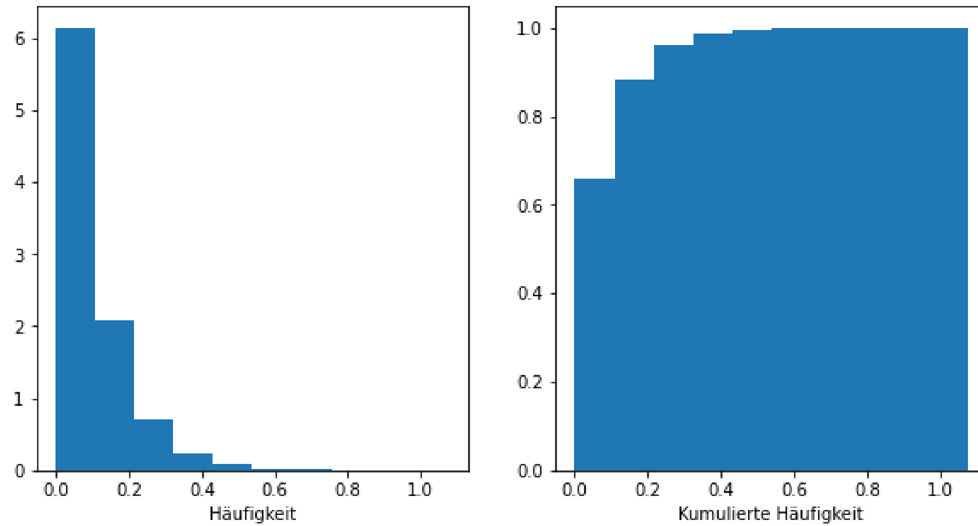
```
import numpy as np
import pandas as pd

random = np.random.default_rng()
data = pd.DataFrame({
    'normal': random.normal(1, 1, 100),
    'exponential': random.exponential(1, 100),
    'chisquare': random.chisquare(1, 100)
})

sns.barplot(data=data, estimator=np.mean, ci=99)
```



2.7.5 Verteilungen darstellen

Eine einfache Möglichkeit, empirische Verteilungen darzustellen ist mittels *hist* von *matplotlib*. Neben den Werten kann optional noch die Anzahl Klassen (*bins*) angegeben werden und ob die Dichte (*density*) und/oder kumulierte Häufigkeit (*cumulative*) angezeigt werden soll.

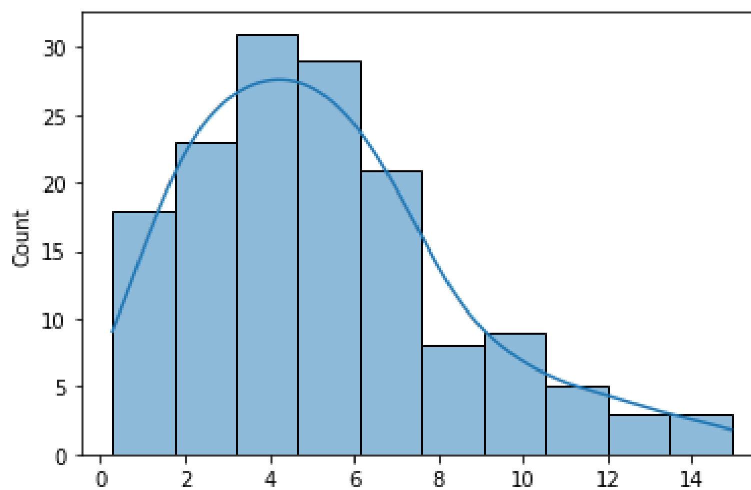
```
import maplotlib.pyplot as plt
from numpy.random import default_rng

random = default_rng()
x = random.exponential(0.1, 100000)

plt.subplots(figsize=(10, 5))
plt.subplot(1, 2, 1).hist(x, bins=10, density=True)
plt.xlabel('Häufigkeit')
plt.subplot(1, 2, 2).hist(x, bins=10, density=True, cumulative=True)
plt.xlabel('Kumulierte Häufigkeit')
```



Mit *Pandas* können zudem Histogramme direkt aus Reihen und Tabellen erstellt werden, indem die Methode *hist* aufgerufen wird.

seaborn erlaubt es zudem, auf einfache Weise eine Schätzung der Wahrscheinlichkeitsverteilung anzuzeigen mittels dem Parameter *kde*:

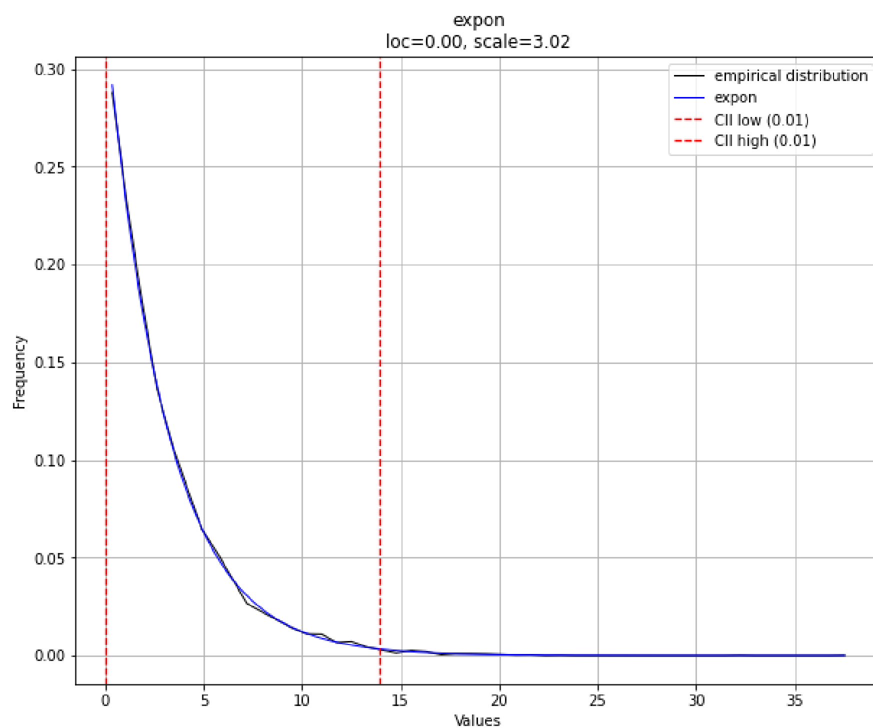
```
import numpy as np
import seaborn as sns

random = np.random.default_rng()
x = random.chisquare(5, 150)

sns.histplot(x, kde=True)
```



2.7.6 Verteilungen ermitteln

Eine sehr einfache Möglichkeit, eine Verteilung herauszufinden ist *distfit*. Optional kann man einn spezifischen Signifikanzwert angeben (*alpha*) oder aus einer Liste von Distributionen auswählen (*distr*). *distfit* erzeugt einen Plot mit den wahrscheinlichen Verteilungen

```
from distfit import distfit
from numpy.random import default_rng

random = default_rng()
x = random.exponential(3, 10000)

dist = distfit(alpha=0.01, distr=['norm', 'expon', 't', 'uniform', 'triang'])
dist.fit_transform(x)
dist.plot()
```



Verteilungsparameter ermitteln Falls man die Verteilung kennt, kann man mit *SciPy* (für einige Verteilungen) die Parameter schätzen lassen mittels *fit*.

```
from numpy.random import default_rng
from scipy import stats

random = default_rng()

x = random.normal(1, 2, 100000)
a, b = stats.norm.fit(x)              # 1.0, 2.0

x = random.exponential(0.1, 100000)
_, lambda_ = stats.expon.fit(x)       # 0.1

x = random.triangular(1, 1.5, 2, 100000)
md, a, b = stats.triang.fit(x)        # 1.5, 1, 2

x = random.chisquare(8, 100000)
df, _, _ = stats.chi2.fit(x)          # 8
```

2.8 Laplace-Experiment

Mit Hilfe der Zufallsfunktionen lassen sich sehr einfach Laplace-Experimente durchführen.

```
from random import choice

dice = (1, 2, 3, 4, 5, 6)
good = 0
total = 100000
for experiment in range(total):
    if choice(dice) == 6:
        good += 1
good / total  # ~ 1/6
```

2.9 Hypothesentests

Hypothesentests mit Python haben den Vorteil, dass direkt die Wahrscheinlichkeit berechnet werden kann, ohne den Umweg über Konfidenzintervalle nehmen zu müssen!

2.9.1 Modelle vergleichen mit t-Tests

T-Tests können sehr einfach mit *SciPy* mittels *ttest_rel* (unabhängige Stichproben) und *ttest_ind* (abhängige Stichproben) berechnet werden. Zudem kann auch ein t-Test für unabhängige Stichproben mittels *ttest_ind_from_stats* berechnet werden, wenn die Varianzen und Mittelwerte bekannt sind.

Bei allen Tests kann die Art der Hypothese angegeben werden:

1. *two-sided*: beide Stichproben haben denselben Mittelwert
2. *less*: Mittelwert ist kleinster
3. *greater*: Mittelwert ist grösser

```
from scipy import stats

a = (1.0, 2.0, 3.0, 4.0, 5.0)
b = (1.1, 1.9, 2.9, 4.1, 5.1)

# Abhängige Stichproben
t, p = stats.ttest_rel(a, b, alternative='less')
t, p = stats.ttest_rel(a, b, alternative='greater')
t, p = stats.ttest_rel(a, b, alternative='two-sided')

# Unabhängige Stichproben
t, p = stats.ttest_ind(a, b, alternative='less')
t, p = stats.ttest_ind(a, b, alternative='greater')
t, p = stats.ttest_ind(a, b, alternative='two-sided')

t, p = stats.ttest_ind_from_stats(
    3.0, 1.41, 5,      # Mittelwert 1, Standardabweichung 1, Stichprobengösse 1
    3.02, 1.45, 5,     # Mittelwert 2, Standardabweichung 2, Stichprobengösse 2
    alternative='less' # or 'greater', 'two-sided'
)
```

Mittels *ztest* von *statsmodels* gibt es zudem auch noch die Möglichkeit, den Test mit der Normalverteilung durchzuführen.

```
import statsmodels.api as sm

a = (1.0, 2.0, 3.0, 4.0, 5.0)
b = (1.1, 1.9, 2.9, 4.1, 5.1)

# Wahrscheinlichkeit, dass beide Stichproben denselben Mittelwert haben
t, p = sm.stats.ztest(a, b, alternative='two-sided')
t, p = sm.stats.ztest(a, b, alternative='smaller')
```

2.9.2 Anteilshypothesen testen

Anteilshypothesen können mit *statsmodels* mittels *binom_test* (exakt) berechnet werden. Es können einseitige oder beidseitige Intervalle getestet werden. Es wird jeweils die Wahrscheinlichkeit berechnet, dass bei gegebener Wahrscheinlichkeit eine Anzahl Erfolge bei einer bestimmten Anzahl Versuche erzielt werden.

```
import statsmodels.api as sm

# Wahrscheinlichkeit, 16 Erfolge bei 20 Versuchen falls p=0.5
p = sm.stats.binom_test(16, 20, 0.5, alternative='smaller')
p = sm.stats.binom_test(16, 20, 0.5, alternative='larger')
p = sm.stats.binom_test(16, 20, 0.5, alternative='two-sided')
```

Mit *proportions_ztest* kann zudem die Wahrscheinlichkeit berechnet werden, wie wahrscheinlich ein gefundener empirischer Wert im Vergleich zur Nullhypothese ist.

```
import statsmodels.api as sm

# Wahrscheinlicheit 18 Erfolge bei 20 Versuchen zu erhalten,
# wenn 10 Erfolge erwartet werden
z, p = sm.stats.proportions_ztest(18, 20, 10/20, alternative='smaller')
z, p = sm.stats.proportions_ztest(18, 20, 10/20, alternative='larger')
z, p = sm.stats.proportions_ztest(18, 20, 10/20, alternative='two-sided')
```

2.9.3 Verteilungen testen

Zwei (kategoriale) Verteilungen können mit *SciPy* auf Gleichheit geprüft werden mittels *chisquare*. Zurückgeben wird der χ^2-Wert sowie die dazugehörige Wahrscheinlichkeit, dass die beiden Verteilungen gleich sind.

```
from scipy import stats

a = (1, 2, 3, 9)
b = (5, 6, 7, 8)

chi_2, p = stats.chisquare(a, b)
```

3 Digitale Codierung

3.1 Ressourcen

NumPy NumPy ist Bibliothek für die Manipulation numerischer Vektoren und Matrizen.

Installation: `pip install numpy`

Mehr Informationen: https://numpy.org

SciPy SciPy enthält eine Vielzahl wissenschaftlicher Funktionen, unter anderem eine grosse Auswahl statistischer Funktionen.

Installation: `pip install scipy`

Mehr Informationen: https://www.scipy.org

scikit-dsp-comm scikit-dsp-comm enthält eine Vielzahl Funktionen und Klassen für Signalverarbeitung und Kommunikationstheorie.

Installation: `pip install scikit-dsp-comm`

Mehr Informationen: https://scikit-dsp-comm.readthedocs.io

bitstring bitstring ist eine kleine Bibliothek für die Erstellung von binären Daten.

Installation: `pip install bitstring`

Mehr Informationen: https://pypi.org/project/bitstring

dahuffman dahuffman ist eine kleine Bibliothek für die Erstellung von Huffman-Codes.

Installation: `pip install dahuffman`

Mehr Informationen: https://pypi.org/project/dahuffman

lzw3 lzw3 ist eine kleine Bibliothek zur LZW-Codierung.

Installation: `pip install lzw3`

Mehr Informationen: https://pypi.org/project/lzw3

PyFlocker PyFlocker ist ein Wrapper für die beiden populären Crypto-Bibliotheken *pycryptodome* und *cryptography*.

Installation: `pip install pyflocker`

Mehr Informationen: https://pyflocker.readthedocs.io

3.2 Vektor- und Matrizenrechnung

Numerische Vektor- und Matrizenrechnungen können mit *NumPy* berechnet werden, Vektoren und Matrizen werden wie gewohnt mit *array* definiert. Neben den üblichen Operationen gibt es auch spezifische Operationen wie das Skalarprodukt (*dot*).

Hinweis: Vektoren sind in *NumPy* eindimensionale Matrizen, Transponieren ist daher nicht nötig.

```
import numpy as np

# Auftrittswahrscheinlichkeit Quelle
px = np.array((0.3, 0.7))

# Informationsgehalt
ix = -np.log2(px)

# Kanalmatrix
pyx = np.array(((0.9, 0.1), (0.1, 0.9)))

# Entropie der Quelle
hx = -np.dot(px, np.log2(px))

# Auftrittswahrscheinlichkeit am Kanalausgang
py = np.dot(px, pyx)

# Entropie am Kanalausgang
hy = -np.dot(py, np.log2(py))

# Verbundentropie
hyx = sum(sum(-px * pyx * np.log2(pyx)))

# Transinformation
t = hy - hyx
```

3.3 Bits und Bytes

In Python können Byte-Folgen als Strings definiert werden. Dazu wird dem ersten Hochkomma ein b vorangestellt. Einzelne Bytes können innerhalb des Strings auch hexadezimal definiert werden, indem das $\backslash x$ Escape-Zeichen verwendet wird.

Eine Umwandlung von Byte-Strings in Bit-Strings und zurück ist mit dem Paket *bitstring* möglich.

Ein anderer, häufig verwendeter Ansatz ist die Bits als NumPy-Array zu speichern.

```
import numpy as np
from bitstring import BitArray

BitArray(bin='0000111100001111').bytes  # b'\x0f\x0f'
BitArray(bytes=b'\x0f\x0f').bin         # '0000111100001111'

# Conversion zu/von NumPy
np.array(BitArray(bin='01001')).astype(int)  # array([0, 1, 0, 0, 1])
BitArray(np.array([0, 1, 0, 0, 1]))          # BitArray('0b01001')
```

3.4 Komprimierung

3.4.1 Huffman-Codierung

Huffman-Codierung ist mit dem Paket *dahuffmann* möglich. Eine neue Codierung wird mittels *from_frequencies* erstellt, anschliessend kann mit *encode* und *decode* codiert und decodiert werden. Die in diesem Paket verwendete Codierung verwendet ein zusätzliches Zeichen *EOF* mit Wert dem Wert Null, welches das Ende einer codierten Nachricht markiert. Dadurch ist es z.B. möglich, Bitfolgen mit Nullen auf Byte-Grenzen aufzufüllen.

```
from bitstring import BitArray
from dahuffman import HuffmanCodec

codec = HuffmanCodec.from_frequencies({
    'A': 0.5,
    'B': 0.25,
    'C': 0.1,
    'D': 0.1,
    'E': 0.05,
})

codec.print_code_table()
#Bits Code  Value Symbol
#   1 0         0 _EOF
#   2 10        2 'A'
#   3 110       6 'B'
#   4 1110     14 'D'
#   5 11110    30 'E'
#   5 11111    31 'C'

codec.encode('DEAD')
codec.decode(BitArray(bin='1110111101011100').bytes)
```

3.4.2 Lempel-Ziv

Python bietet mittels *zipfile* von Haus aus Unterstützung für *deflate*, *bzip2* und *lzma* Codierungen, die frühren *LZ77*, *LZ78* und *LZW* Codierungen sind heute jedoch praktisch ohne Bedeutung. Codiert wird immer ein gesamtes Archiv, d.h. eine oder mehrere Dateien.

```python
from zipfile import ZipFile, ZIP_DEFLATED, ZIP_BZIP2, ZIP_LZMA

# deflated
with ZipFile('deflated.zip', 'w', compression=ZIP_DEFLATED) as file:
    file.writestr('content.txt', 'Hello world!')
with ZipFile('deflated.zip', 'r') as file:
    file.read('content.txt')                      # b'Hello world!'

# bzip2
with ZipFile('bzip2.zip', 'w', compression=ZIP_BZIP2) as file:
    file.writestr('content.txt', 'Hello world!')
with ZipFile('bzip2.zip', 'r') as file:
    file.read('content.txt')                      # b'Hello world!'

# lzma
with ZipFile('lzma.zip', 'w', compression=ZIP_LZMA) as file:
    file.writestr('content.txt', 'Hello world!')
with ZipFile('lzma.zip', 'r') as file:
    file.read('content.txt')                      # b'Hello world!'
```

Wer trotzdem einen Blick in eine Implementierung der LZW Codierung werfen möchte, kann das zum Beispiel mit dem Paket *lzw3* machen. Das Paket nimmt geöffnete Dateien entgegen, liest Zeichen Byte-weise ein und erstellt das Wörterbuch beim codieren. Das Wörterbuch hat dementsprechend zu Beginn 256 Einträge.

```python
from lzw3.compressor import LZWCompressor

with open('in.data', 'wb') as file:
    file.write(b'\x01\x02\x03\x01\x02\x03\x01\x02\x03\x01\x02\x03')

compressor = LZWCompressor()
compressor.compress('in.data', 'out.data')

compressor._sequence_table
# {(-1, 0): 0,       # Index 0: 0
#  (-1, 1): 1,       # Index 1: 1
#  ...
#  (1, 2): 257,      # Index 257: 12
#  (2, 3): 258,      # Index 258: 23
#  (3, 1): 259,      # Index 259: 31
#  (257, 3): 260,    # Index 260: 123
#  (259, 2): 261,    # Index 261: 312
#  (258, 1): 262}    # Index 262: 231
```

3.5 Verschlüsselung

Zurzeit gibt es in Python zwei populäre Crypo-Bibliotheken: *pycryptodome* und *cryptography*. Mittels *PyFlocker* können beide mit demselben Interface verwendet werden.

3.5.1 RSA

Verschlüsselung und Entschlüsselung ist mit *pyflocker* mittels *RSA* möglich. Dazu wird zum Beispiel zuerst ein Schlüsselpaar generiert, anschliessend kann mit dem öffentlichen Schlüssel verschlüsselt, mit dem privaten Entschlüsselt werden.

```
from pyflocker.ciphers import RSA
from pyflocker.ciphers.backends import Backends

private = RSA.generate(2048, backend=Backends.CRYPTODOME)
public = private.public_key()

message = public.encryptor().encrypt(b'Hello World')
private.decryptor().decrypt(message)
```

3.6 Blockcodes

3.6.1 Hamming-Distanz

Die normierte Hamming-Distanz kann mit *SciPy* mittels *hamming* berechnet werden. Durch das Multiplizieren des berechneten Wertes mit der Länge der Codewörter erhält man die Hamming-Distanz in Bits.

```
from bitstring import BitArray
from scipy.spatial.distance import hamming

# Numpy/Listen
x_1 = (0, 1, 0, 0, 1)
x_2 = (1, 1, 0, 0, 0)
hamming(x_1, x_2) * len(x_1)  # 2.0

# BitArray
x_1 = BitArray(bin='01001')
x_2 = BitArray(bin='11000')
hamming(x_1, x_2) * len(x_1)  # 2.0
(x_1 ^ x_2).count(True)       # 2
```

3.6.2 Hamming-Blockcodes

Hamming-Blockcodes können mit *scikit-dsp-comm* mittels *fec_hamming* erstellt und verwendet werden. Die Bibliothek kennt je einen Blockcode für ein gegebenes k:

k	(n, m)
3	(7, 4)
4	(15, 11)
5	(31, 26)
6	(63, 57)
...	...

Neben dem Codieren (*hamm_encoder*) und Decodieren (*hamm_decoder*) kann auch die Prüfmatrix H ausgegeben werden. Das Decodieren liefert leider keine Information zur Anzahl korrigierter Fehler oder zum Fehlersyndrom.

```
import numpy as np
from sk_dsp_comm.fec_block import fec_hamming

code = fec_hamming(4)
code.hamm_encoder(np.array((1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0)))
code.hamm_decoder(np.array((1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1)))

# Prüfmatrix
hh.H
#[[1 1 0 1 0 0 0 1 1 1 1 1 0 0 0]
# [1 1 0 1 1 1 1 1 0 0 0 0 1 0 0]
# [1 1 1 0 0 1 1 0 0 1 1 0 0 1 0]
# [1 0 1 1 1 0 1 0 1 0 1 0 0 0 1]]
```

3.6.3 Zyklische Blockcodes

Zyklische Blockcodes können mit *scikit-dsp-comm* mittels *fec_cyclic* erstellt und verwendet werden. Es wird das Generatorpolynom als Vektor angegeben. Anschliessend kann mit dem Code codiert (*cyclic_encoder*) und decodiert (*cyclic_decoder*) werden. Das Decodieren liefert leider keine Information zur Anzahl korrigierter Fehler oder zum Fehlersyndrom.

```
import numpy as np
from sk_dsp_comm.fec_block import fec_cyclic

code = fec_cyclic('1011')  # x^3 + x + 1

code.cyclic_encoder(np.array((1, 0, 1, 0)))
code.cyclic_decoder(np.array((1, 0, 1, 0, 0, 1, 1)))
```

3.7 Faltungscodes

Faltungscodes können mit *scikit-dsp-comm* mittels *fec_conv* erstellt und verwendet werden. Es werden die Generatorpolynome als Vektor angegeben. Anschliessend kann mit dem Code codiert (*conv_encoder*) und decodiert (*viterbi_decoder*) werden. Für das Decodieren muss entsprechend der Entscheidungstiefe Bits zum Codewort hinzugefügt werden. Das Decodieren liefert leider keine Information zur Anzahl korrigierter Fehler oder zum Fehlersyndrom.

```
import numpy as np
from sk_dsp_comm.fec_conv import fec_conv

# Codierung
code = fec_conv(('101', '111'))            # x^2 + 1, x^2 + x + 1
x = np.array((1, 0, 0, 1, 1, 0, 1, 0, 0))  # inklusive 2 Tailbits
y, state = code.conv_encoder(x, '00')
y = y.astype(int)                          # [1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 1 1 1]

# Decodierung mit Entscheidungstiefe 10
depth = 10
code = fec_conv(('101', '111'), depth)
y = np.append(y, 2 * (depth-1) * [0])
code.viterbi_decoder(y, 'hard').astype(int) # [1 0 0 1 1 0 1 0 0]
```

4 Simulation

4.1 Ressourcen

Pandas Pandas ist Bibliothek für Datenanalyse und Manipulation.

Installation: `pip install pandas`

Mehr Informationen: https://pandas.pydata.org

Simpy Simpy ist Bibliothek für ereignisorientierte Simulation (DES).

Installation: `pip install simpy`

Mehr Informationen: https://simpy.readthedocs.io

Salabim Salabim ist Bibliothek für ereignisorientierte Simulation (DES).

Installation: `pip install salabim`

Mehr Informationen: https://www.salabim.org

ipywidgets ipywidgets ist Bibliothek für interaktive Widgets in Jupyter Notebooks..

Installation: `pip install ipywidgets`

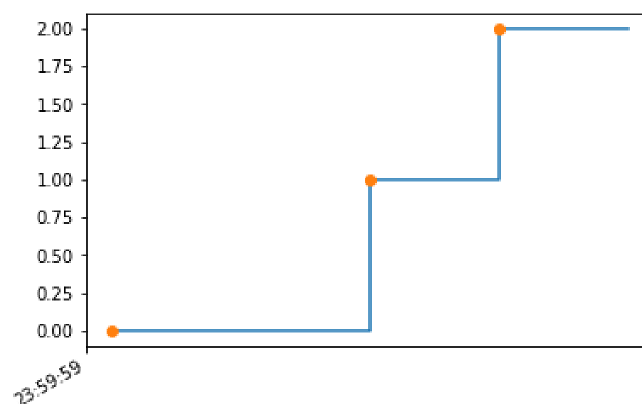
Mehr Informationen: https://ipywidgets.readthedocs.io

4.2 Zeitdiskrete Variablen

Für den Umgang mit zeitdiskreten Variablen eignet sich insbesondere Reihen (*Series*) von *Pandas*, welchen ein zeitbasierter Index hinzugefügt wird. Diese können anschliessend zum Beispiel sehr einfach ausgegeben werden mittels *plot* und mittels *diff* gewichtet werden.

Damit die Gewichtung richtig berechnet werden kann, sollte das Simulationsende auch erfasst werden (zum Beispiel mit *np.nan*). Da *diff* zudem die Differenz zum vorherigen Wert berechnet, muss das Resultat verschoben werden (*shift(-1)*).

```
import pandas as pd
import numpy as np

end = 2
x = pd.Series({
    pd.to_datetime(0, unit='s'): 0,
    pd.to_datetime(1, unit='s'): 1,
    pd.to_datetime(1.5, unit='s'): 2,
    pd.to_datetime(end, unit='s'): np.nan  # Simulationsende
})

# Plotting
x.plot(drawstyle='steps-post')
x.plot(marker='o', linestyle='')

# Gewichtung
w = x.index.to_series().diff().shift(-1).dt.total_seconds()
average = (w * x).sum() / end  # 0.75
```



4.3 SimPy

SimPy ist ein Framework für diskrete Ereignissimulation mit Prozessen, Interprozessinteraktionen und geteilten Ressourcen.

Prozesse werden als Funktionen definiert, welche mittels *yield* pausiert werden können. Diese werden einem Scheduler (*Environment*) übergeben, welcher sie bis zu einem bestimmten Zeitpunkt ausführt (*env.run*). Ein Prozess kann entweder nach Abschluss seiner Aufgabe sich selbst beenden (d.h. die Funktion kehrt zurück) oder endet mit dem Ende der Simulation (d.h. die Funktion ist beinhaltet eine Endlosschlaufe).

Geteilte Ressourcen können mit *request* angefordert und *release* wieder freigegeben werden. Alternativ kann auch ein *with* verwendet werden, um die Ressource automatisch freizugeben. Sobald eine Ressource angefordert wurde, muss mit *yield* gewartet werden, bis sie verfügbar ist.

Im Vergleich zu *Salabim* besitzt *SimpPy* weniger eingebaute Funktionen, ist dafür aber unter Umständen in gewissen Bereichen etwas flexibler.

Hier ein kleines Beispiel mit einer Kaffeemaschine. Die Simulation hat zwei Prozesse: *customer* und *generate_customer*. *customer* modelliert das Verhalten eines Kunden: Anstehen vor der Kaffeemaschine und warten bis diese frei wird, anschliessend Kaffeezubereitung mit einer zufälligen, dreiecksverteilten Wartezeit. *generate_customer* modelliert das exponentiell verteilte Eintreffen von neuen Kunden.

```
from random import expovariate, triangular
from simpy import Environment, Resource

def customer(env):
    with env.coffee_machine.request() as request:
        # waiting for coffee machine
        yield request

        # waiting for coffee
        yield env.timeout(triangular(10, 30, 15))

def generate_customer(env):
    while True:
        # create a new customer
        env.process(customer(env))

        # wait for next customer to arrive
        yield env.timeout(expovariate(50/(60*60)))

env = Environment()
env.coffee_machine = Resource(env)
env.process(generate_customer(env))
env.run(until=60*60)
```

4.3.1 Statistische Auswertung

SimPy besitzt keine eigenen Hilfsmittel für die statistische Auswertung. Hierzu kann aber zum Beispiel einfach relevante Messpunkte in einer Variablen erfasst werden und mit *pandas* oder *NumPy* ausgewertet werden.

Hier ein kleines Beispiel, wie eine Warteschlange umgesetzt werden kann. Bei jedem *request* und *release* wird ein neuer Messwert erfasst, diese können anschliessend zu einer *pandas*-Zeitreihe konvertiert werden.

```
import numpy as np
import pandas as pd

from collections import OrderedDict
from simpy import Environment, Resource


class WaitingLine(Resource):

    def __init__(self, env, capacity=1):
        super().__init__(env, capacity)
        self.data = OrderedDict()

    def request(self):
        now = pd.to_datetime(self._env.now, unit='s')
        self.data[now] = len(self.queue)
        return super().request()

    def release(self, request):
        now = pd.to_datetime(self._env.now, unit='s')
        self.data[now] = len(self.queue)
        return super().release(request)

env = Environment()
env.coffee_machine = WaitingLine(env)
env.process(generate_customer(env))
env.run(60*60)

series = pd.Series(env.coffee_machine.data)
series.plot(drawstyle='steps-post')
series.describe()
# count    108.000000
# mean       0.425926
# std        0.929312
# min        0.000000
# 25%        0.000000
# 50%        0.000000
# 75%        0.000000
# max        4.000000
```

4.3.2 Animation

SimPy besitzt keine eigenen Hilfsmittel für Animation. Hierzu kann aber zum Beispiel für Jupyter Notebooks *ipywidgets* verwendet werden. Dabei werden die Widgets mit Hilfe eines eigenen Prozesses zu bestimmten Zeitpunkten aktualisiert.

Hier ein kleines Beispiel, wie eine Progress-Bar mit *ipywidgets* und *SimPy* umgesetzt werden kann:

```
from ipywidgets.widgets import IntProgress
from simpy import Environment


def progress(env, until, steps=100):
    progress = IntProgress(max=steps)
    progress.value = 0
    display(progress)
    while True:
        yield env.timeout(until / (steps - 1))
        progress.value = env.now

env = Environment()
until = 60 * 60
env.process(progress(env, until))
env.run(until=until)
```

4.4 Salabim

Salabim ist ein weiteres Framework für diskrete Ereignissimulation mit Prozessen, Prozessinteraktionen, geteilten Ressourcen, Warteschlangen, statistischer Probenahme, Überwachung und Animation.

Prozesse werden als Klassen definiert, welche mittels *yield* pausiert werden können. Diese werden einem Scheduler (*Environment*) übergeben, welcher sie bis zu einem bestimmten Zeitpunkt ausführt (*env.run*). Ein Prozess kann entweder nach Abschluss seiner Aufgabe sich selbst beenden (d.h. die Funktion kehrt zurück) oder endet mit dem Ende der Simulation (d.h. die Funktion ist beinhaltet eine Endlosschlaufe).

Geteilte Ressourcen können mit *request* angefordert und *release* wieder freigegeben werden. Sobald eine Ressource angefordert wurde, muss mit *yield* gewartet werden, bis sie verfügbar ist.

Im Vergleich zu *SimPy* besitzt *Salabim* mehr eingebaute Funktionen ("batteries included"), ist aber unter Umständen in gewissen Bereichen etwas weniger flexibel.

Hier ein kleines Beispiel mit einer Kaffeemaschine. Die Simulation hat einen Prozess (*Customer*), welcher das Verhalten eines Kunden modelliert: Anstehen vor der Kaffeemaschine und warten bis diese frei wird, anschliessend Kaffezubereitung mit einer zufälligen, dreiecksverteilten Wartezeit. Kunden werden über eine spezielle Komponente (*ComponentGenerator*) exponentiell-verteilt der Simulation hinzugefügt.

```python
from salabim import Environment, Component, ComponentGenerator, Resource
from salabim import Exponential, Triangular

class Customer(Component):

    def process(self):
        # waiting for coffee machine
        yield self.request(self.env.coffee_machine)

        # Wait for coffee
        yield self.hold(Triangular(10, 60, 15).sample())

        self.release(self.env.coffee_machine)

env = Environment()
env.coffee_machine = Resource()
ComponentGenerator(Customer, iat=Exponential((60*60)/50))
env.run(till=60*60)
```

4.4.1 Statistische Auswertung

Salabim erfasst zum Beispiel für Ressourcen automatisch Messwerte wie Länge der Warteschlange, Länge des Aufenthalts in der Warteschlange usw. Zu diesen Messwerten können bequem die statistischen Parameter ausgelesen werden (*print_statistics*). Eigene Messwerte können mit einem *Monitor* mittels *tally* erfasst werden.

```
import matplotlib.pyplot as plt

env.coffee_machine.requesters().length.print_statistics()
#Statistics of Length of requesters of resource.0 at      3600
#                       all     excl.zero         zero
#-------------- ------------ ------------ ------------
#duration           3600          450.903     3149.097
#mean                  0.171        1.362
#std.deviation         0.518        0.721
#
#minimum               0            1
#median                0            1
#90% percentile        1            3
#95% percentile        1            3
#maximum               4            4

plt.plot(*env.coffee_machine.requesters().length.tx(), drawstyle="steps-post")
```

4.4.2 Animation

Salabim besitzt eine ausführliche Animationsbibliothek, welche auf *tkinter* basiert. Alternativ kann aber auch zum Beispiel für Jupyter Notebooks *ipywidgets* verwendet werden. Dabei werden die Widgets mit Hilfe eines eigenen Prozesses zu bestimmten Zeitpunkten aktualisiert.

```
from ipywidgets.widgets import IntProgress
from salabim import Environment, Component

class Progress(Component):

    def process(self):
        progress = IntProgress(max=self.env.until)
        display(progress)
        while True:
            progress.value = self.env.now()
            yield self.hold(self.env.until / 99)

env = Environment()
env.until = 60 * 60
Progress()
env.run(till=env.until)
```

4.5 Mesa

Mesa ist ein Framework für agentenbasierte Modellierung. Die Simulation erfolgt in Schritten. Agenten und Modelle werden dabei als Klassen definiert und besitzen jeweils eine *step* Funktion, welche die Aktion innerhalb eines Schrittes definiert. Optional kann ein *Scheduler* verwendet werden, um die Aktivierung resp. die Reihenfolge der Aktivierung zu definieren (z.B. zufällig). Mesa bietet zudem Unterstützung für die Sammlung von Daten, das parametrisierte Ausführen von Simulationen sowie die Visualisierung.

Hier ein kleines Beispiel eines ungeordnete Anstehen an einer Kaffeemaschine. Das Modell hat 50 Kunden, welche jeweils überprüfen, ob die Kaffemaschine frei ist und sich gegebenenfalls eine Kaffee entnehmen und die Maschine wieder freigeben. Die Reihenfolge der Kunden bei jedem Schritt ist zufällig. Bei jedem Schritt werden die Zustände der Kunden erfasst. Es werden 50 Schritte simuliert.

```
from mesa import Agent, Model
from mesa.datacollection import DataCollector
from mesa.time import RandomActivation


class Customer(Agent):

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.had_coffee = False
        self.state = 'idle'

    def step(self):
        if self.state == 'idle':
            if self.model.coffee_machine_free:
                self.state = 'waiting'
                self.model.coffee_machine_free = False
        elif self.state == 'waiting':
            self.state = 'done'
            self.model.coffee_machine_free = True

class CoffeeMachineModel(Model):

    def __init__(self):
        self.coffee_machine_free = True
        self.schedule = RandomActivation(self)
        for unique_id in range(50):
            self.schedule.add(Customer(unique_id, self))
        self.collector = DataCollector(agent_reporters={'State': 'state'})

    def step(self):
        self.collector.collect(self)
        self.schedule.step()


model = CoffeeMachineModel()
for x in range(50):
    model.step()

data = model.collector.get_agent_vars_dataframe()
data = data.xs(49, level='Step')['State']
data[data == 'done'].count()
```