Mathematisches Praktikum

Grundlagen Python

M.Sc. Johannes Michael
Universität Rostock

- Einführung
- Basisdatentypen & Variablen
- Operatoren
- Sequentielle Datentypen
- Dictionaries
- Formatierte Ausgaben
- Code–Strukturierung
- Bedingte Anweisungen
- Schleifen
- Funktionen
- Dateien
- Listen-Abstraktion
- Modularisierung
- Numerisches Python
- Graphiken mit Matplotlib

Vorteile & Besonderheiten

- Open-source und plattformunabhängig
- Minimalistische Syntax
- Große Standardbibliothek
- Unterstützt objektorientierte Programmierung
- Flexible Indizierung und Slicing bei Sequenzen
- Automatisches Speichermanagement
- . . .

Interpreter & Interaktive Shell

Interpreter

Ein Interpreter (Softwaretechnik) ist ein Computerprogramm, das einen Programm-Quellcode im Gegensatz zu Assemblern oder Compilern nicht in eine auf dem System direkt ausführbare Datei übersetzt, sondern den Quellcode einliest, analysiert und ausführt.[a]

[a] https://de.wikipedia.org/wiki/Interpreter

Interaktive Shell

Die interaktive Shell steht zwischen dem Anwender und dem Betriebssystem bzw. dem zu interpretierenden Programm (z. B. Python). Eingaben werden dabei direkt von der Kommandozeile gelesen und unverzüglich ausgeführt.

Python–Skripte

- Programme werden normalerweise nicht interaktiv eingetippt sondern in Dateien/Skripten gespeichert.
- Skripte können mit beliebigem Texteditor bearbeitet werden
  $\longrightarrow$ Empfehlung: PyCharm IDE
- Ausführen von Python Skripten:
  - Direkt über IDE: `Run...`
  - Über Kommandozeile `python script.py`

Basisdatentypen & Variablen I

Datentyp

- bezeichnet die Zusammenfassung konkreter Wertebereiche und die darauf definierten Operationen zu einer Einheit
- Basisdatentypen: Ganzzahlen (Integer), Fließkommazahlen (Float), Zeichenketten (Strings), Wahrheitswerte (Boolean)

- in Programmiersprachen wie C/C++ oder Java muss der Datentyp explizit angegeben werden (Bindung an Datentyp)

```
int x = 50;          float y = 1.2;
String s = "Test";   boolean b = true;
```

Basisdatentypen & Variablen II

- Variablen halten keinen bestimmten Typ sondern referenzieren Objekte $\rightarrow$ keine Typdeklaration

```
x = 50                    y = 1.2
s = "Example Text"        b = True
```

- Zuweisung von Wert an Variable durch "="

```
z = x + 10
```

- Umwandlung von Datentypen (Casting):

```
x = 3.1 # Float
int(x)  # Integer, liefert 3
```

Basisdatentypen & Variablen III

- Typ und Wert einer Variablen kann zur Laufzeit geändert werden (d.h. ein neues Objekt eines beliebigen Typs wird der Variablen zugewiesen)
- Variablen referenzieren Objekte und Objekte können einen beliebigen Datentyp haben $\rightarrow$ Variablen können nicht mit Datentypen "verknüpft" werden
- Typabfrage einer Variable:

```
x = 10
print(type(x)) # <type 'int'>
```

Operatoren

Operatoren

- `+, -` : Addition, Subtraktion
- `*` : Multiplikation
- `/` : Division (liefert *float* Python3)
- `//` : Ganzzahldivision (Ganzzahliger Anteil)
- `**` : Exponentiation
- `%` : Modulo (Rest)
- `or, and, not` : Boolsches *Oder*, *Und*, *Nicht*
- `is` : Vergleichsoperator (Identität/Speicherort)
- `<,<=,>,>=,!=,==` : (Standard-)Vergleichsoperatoren

Sequentielle Datentypen I

- Datentyp, der eine Folge von Elementen beinhaltet
- Elemente haben definierte Reihenfolge $\rightarrow$ Zugriff über Indizes möglich
- Python stellt *Strings*, *Listen* und *Tupel* zur Verfügung
- Strings und Tupel sind nach Erzeugung *unveränderlich* (immutable)
- Listen sind nach Erzeugung *veränderlich* (mutable)

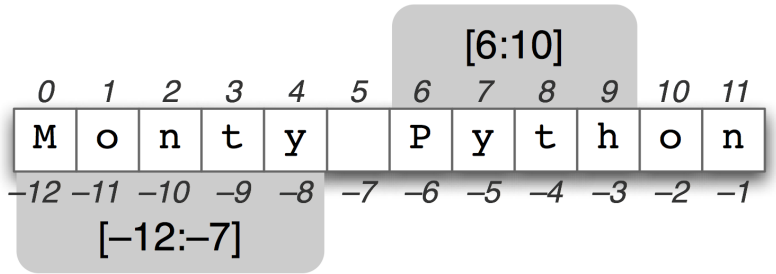
Sequentielle Datentypen II

Operationen auf Sequenzen

- `x in s` : prüft ob sich `x` in `s` befindet
- `s + t` : Verkettung von `s` und `t` als neue Sequenz
- `s += t` : hängt das Element `t` an die Sequenz `s` an
- `s * n` : liefert `n`–fache Kopie der Sequenz `s`
- `s[i]` : liefert das `i`-te Element der Sequenz `s`
- `s[i:j]` : liefert Teilsequenz von Index `i` bis Index `j-1` von `s`
- `s[i:j:k]` : wie `s[i:j]`, nur jedes `k`-te Element wird extrahiert
- `len(s)` : liefert die Anzahl der Elemente in `s`
- `min(s)` : liefert das kleinste Element von `s`
- `max(s)` : liefert das größte Element von `s`

Sequentielle Datentypen – Strings I

- Sequenz von einzelnen Zeichen, die indiziert sind
- Indizierung beginnt bei 0 und ist von hinten möglich




```
s = "Monty Python"
print(s[0]) # Ausgabe: M
```

- Strings sind unveränderlich:

```
s[4] = "Y" # liefert Fehler
```

Sequentielle Datentypen – Strings II

String-Funktionen

- Konkatenation:
  `"Monty" + " Python"` → `"Monty Python"`
- Wiederholung:
  `"Python" * 3` → `"PythonPythonPython"`
- Indexing:
  `"Monty Python"[-1]` → `"n"`
- Slicing:
  `"Monty Python"[6:10]` → `"Pyth"`
- Länge eines Strings:
  `len("Monty Python")` → `12`
- Aufspalten von Strings:
  `"Monty Python".split()` → `["Monty", "Python"]`

Sequentielle Datentypen – Listen I

- Eine Liste speichert eine Folge beliebiger Objekte
- Definiert über eckige Klammern und Elemente mit Kommas getrennt:

```
liste = ["Python", 4, 3.1]
```

- Listen sind veränderlich
  `liste[0] = 1` $\rightarrow$ `[1, 4, 3.1]`
- `liste[:]` legt eine Kopie von `liste` an
- `[]` erzeugt leere Liste
- Liste von Listen ebenfalls möglich:

```
liste = [[1, 2, 3], [4, 5, 6]]
```

Sequentielle Datentypen – Listen II

- Operationen wie Konkatenation, Wiederholung, Indexing, Slicing, Länge . . . übertragen sich analog
- Listeninhalt prüfen mittels `in` und `not in` Operatoren:

```
list = ["a", "b", "c", "d", "e"]
"a" in list     # gibt Wert True zurück
"d" not in list # gibt Wert False zurück
s = "Python"
"y" in s        # gibt Wert True zurück
```

- *Tupel* entspricht unveränderlicher Liste (runde Klammern):

```
tuple = ("a", "b", "c", "d", "e")
```

Sequentielle Datentypen – Listen III

Eine Liste kann als ein Stapelspeicher (Stack) angesehen werden.

Operationen auf einem Stack

- `push`: legt neues Objekt auf den Stack
  → `append` als Äquivalent für Listen
- `pop`: gibt oberstes Objekt des Stacks zurück und entfernt es
  → `liste.pop(i)` wendet `pop` auf das i-te Element an
- `peek`: gibt oberstes Objekt des Stacks zurück ohne es zu entfernen
  → `liste[-1]` als Äquivalent für Listen

Sequentielle Datentypen – Listen IV

append vs extend

- Hinzufügen von mehr als einem Element:

```
l = [1, 2, 3]
l.append([4, 5]) # ergibt [1, 2, 3, [4, 5]]
```

- verwende stattdessen `extend`:

```
l.extend([4, 5]) # ergibt [1, 2, 3, 4, 5]
```

- das Argument von `extend` muss ein iterierbares Objekt sein:

```
l.extend("Hello")
# ergibt [1, 2, 3, 'H', 'e', 'l', 'l', 'o']
```

Sequentielle Datentypen – Listen V

remove

- Entfernen eines Wertes ohne Kenntnis des Indexes:

```
abc = ["a", "b", "c", "d", "e"]
abc.remove("b") # abc = ["a", "c", "d", "e"]
```

- Fehler falls das Element nicht in der Liste vorkommt:

```
abc.remove("f")
ValueError: list.remove(x): x not in list
```

index

- Finden der Position eines Elementes:

```
abc = ["a", "b", "c", "a", "e"]
abc.index("a")          # gibt 0 zurück
abc.index("a", 1)       # gibt 3 zurück
abc.index("a", 1, 2)    # ValueError
```

Sequentielle Datentypen – Listen VI

insert

- Es ist nicht möglich mittels `append` ein Element an beliebiger Stelle einzufügen → `insert`:

```
abc = ["a", "b", "c", "e"]
abc.insert(3, "d")
# abc = ["a", "b", "c", "d", "e"]
```

Dictionaries I

Assoziatives Datenfeld

Das assoziative Datenfeld (englisch map, dictionary) ist eine Datenstruktur, die – anders als ein gewöhnliches Feld (engl. array) – nichtnumerische (oder nicht fortlaufende) Schlüssel (zumeist Zeichenketten) verwendet, um enthaltene Elemente zu adressieren. [a]

[a] https://de.wikipedia.org/wiki/Assoziatives_Datenfeld

- besteht aus Schlüssel-Objekt-Paaren (key-value pairs)
- zu einem Schlüssel gehört immer ein Objekt (Mapping)
- Nicht sequentiell $\rightarrow$ keine Anordnung, keine Indizierung
- Zugriff über Schlüssel

Dictionaries II

Beispiele

- Leeres Dictionary: `d = {}`
- Englisch-Deutsch Wörterbuch:
  ```
  ed = {"red":"rot", "green":"grün"}
  ```
- Deutsch-Französisch Wörterbuch:
  ```
  df = {"rot":"rouge", "grün":"vert"}
  ```
- Englisch-Französisch Wörterbuch über Transitiviät:
  ```
  df[ed["red"]]   # gibt "rouge" zurück
  ```

- als Werte können beliebige Typen verwendet werden
- bei Schlüsseln können nur Instanzen **unveränderlicher** Datentypen verwendet werden (z. B. Strings)

Dictionaries III

Operatoren auf Dictionaries

- `len(d)` : liefert die Anzahl der Schlüssel-Werte-Paare
- `del d[k]` : löscht Eintrag zum Schlüssel `k`
- `k (not) in d` : True, wenn es in `d` (k)einen Schlüssel `k` gibt

pop

`pop` existiert in abgewandelter Form auch für Dictionaries:

```
d = {"a":1, "b":2, "c":3}
d.pop("a")     # gibt Wert 1 zurück
               # d = {"b":2, "c":3}
d.pop("d")     # KeyError: 'd'
d.pop("d", 4) # gibt Default-Wert 4 zurück
d.pop("b", 4) # gibt Wert 2 zurück
```

Dictionaries IV

popitem

`popitem` benötigt keine Parameter und liefert beliebiges Schlüssel-Wert-Paar als Tupel zurück:

```
d = {"a":1, "b":2, "c":3}
d.popitem()  # gibt z. B. ("a", 1) zurück
             # d = {"b":2, "c":3}
```

get

`get` als weitere Methode um auf die Werte über die Schlüssel zuzugreifen:

```
d = {"a":1, "b":2, "c":3}
d.get("c")  # gibt Wert 3 zurück
d["c"]      # äquivalent
```

Dictionaries V

copy

- `copy` ermöglicht das Kopieren von Dictionaries:

```
d1 = {"a":1, "b":2, "c":3}
d2 = d1.copy()    # d2 = {"a":1, "b":2, "c":3}
d1["a"] = 4       # d1 = {"a":4, "b":2, "c":3}
                  # d2 = {"a":1, "b":2, "c":3}
```

- falls es sich bei dem Wert um einen komplexen Datentyp handelt, wirken sich Änderungen innerhalb eines solchen Wertes auf Original und Kopie aus:

```
d1 = {"a":[1,2], "b":[3,4]}
d2 = d1.copy() # d2 = {"a":[1,2], "b":[3,4]}
d1["a"][0] = 5 # d1=d2 = {"a":[5,2], "b":[3,4]}
```

Dictionaries VI

clear

`clear` leert den Inhalt eines Dictionaries (das Dictionary wird dabei nicht gelöscht):

```
d = {"a":1, "b":2, "c":3}
d.clear  # d = {}
```

update

`update` ermöglicht das Updaten über weitere Dictionaries:

```
d1 = {"a":1, "b":2}
d2 = {"b":3, "c":4}
d1.update(d2)  # d1 = {"a":1, "b":3, "c":4}
```

Dictionaries VII

Iteration über Dictionaries

- Iteration über die Schlüssel:

```
d = {"a":1, "b":2}
for key in d:  # liefert nacheinander 'a' und 'b'
  print(key)
```

oder

```
for key in d.keys():
  print(key)
```

- Iteration über die Werte:

```
d = {"a":1, "b":2}
for val in d.values():  # liefert nacheinander
  print(val)            # 1 und 2
```

Dictionaries VIII

Casting zwischen Listen und Dictionaries

- Dictionary $\rightarrow$ Liste:

```
d = {"a":1, "b":2, "c":3}
l_keys = list(d) # oder l_keys = d.keys()
l_values = d.values()
```

- Liste $\rightarrow$ Dictionary:

```
l1 = ["a", "b", "c"]
l2 = [1, 2, 3]
d = dict(zip(l1,l2))
# zip fasst die Komponenten zu Tupeln zusammen
# d = {"a":1, "b":2, "c":3}
```

Formatierte Ausgaben I

- Einfachste Ausgabe: Kommaseparierte Liste von Werten.

```
a, b, c = 1, 2.2, "test"
print(a, a+b, c)
```

- Ausgabeseparator und Endzeichen kann explizit gesetzt werden.

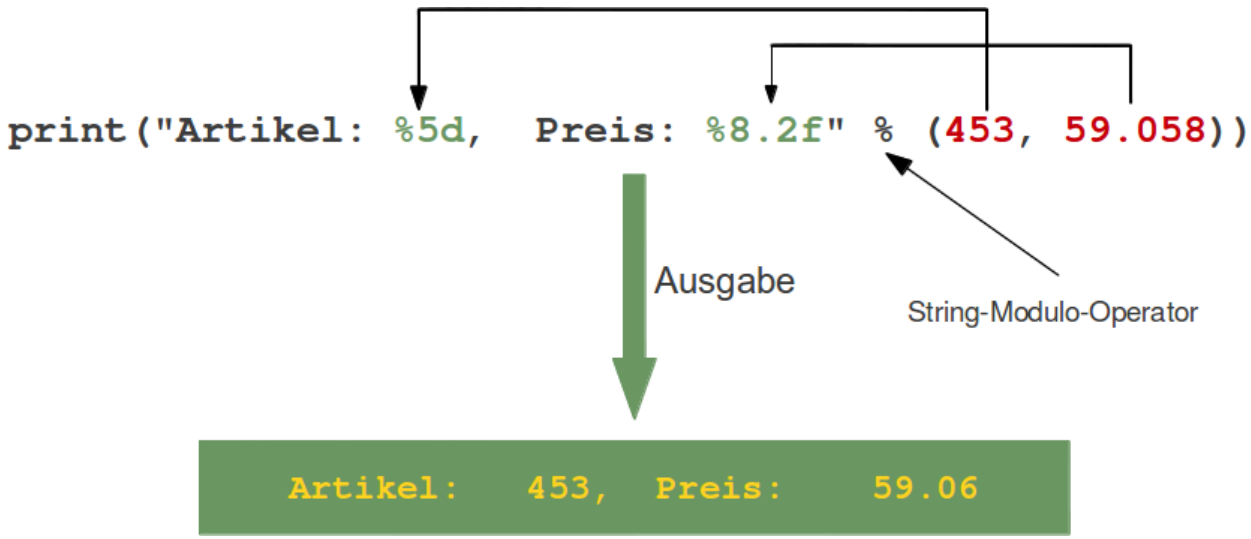
```
print(a, a+b, c, sep=",", end="|")
```

- Alternativ: Neuen String mittels String-Konkatenation ausgeben (beachte dass Variablen in Strings umgewandelt werden müssen).

```
print(str(a) + "," + str(a+b) + "," + c)
```

Formatierte Ausgaben II

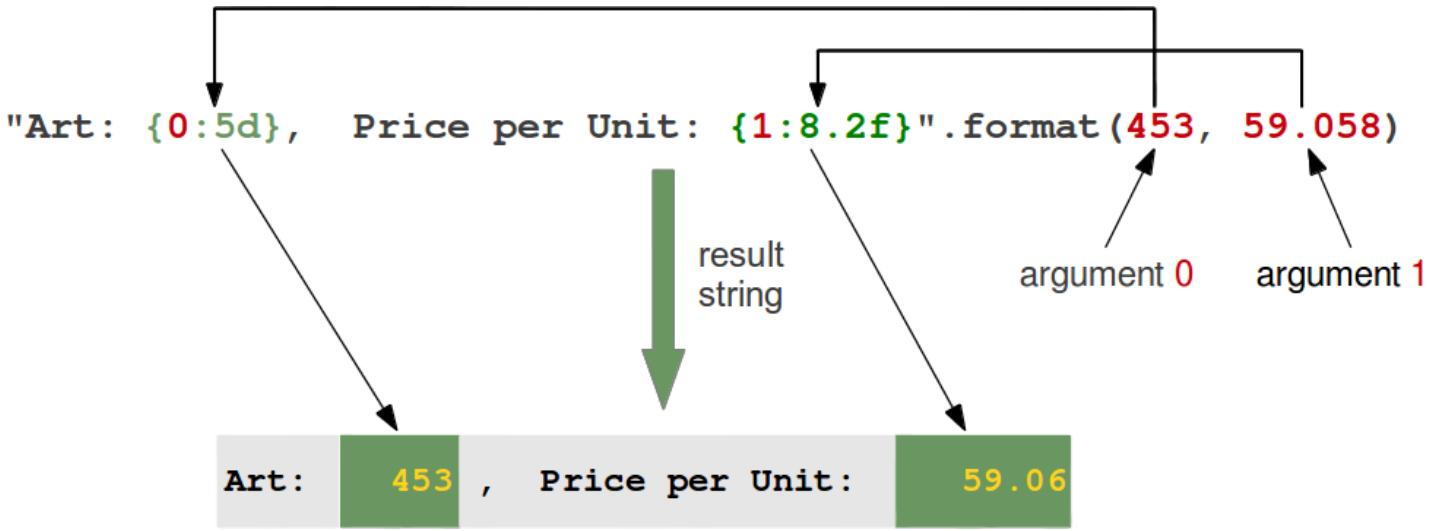
- "Veraltet": **String-Modulo-Operator**.
- Syntax: "Format-String mit Platzhaltern" % (Füllwerte)
- Platzhalter: %[flags][width][.precision]type



https://www.python-kurs.eu/python3_formatierte_ausgabe.php

Formatierte Ausgaben III

- "Klassisch": String-Methode **format**
- Syntax: "Format-String mit Format-Codes".format(Füllwerte)
- Format-Codes: {[index]:[[fill]align][flag][width][.precision][type]}



https://www.python-kurs.eu/python3_formatierte_ausgabe.php

Formatierte Ausgaben IV

- "Neu": **f-Strings**
- Analog zu format-Strings, wobei Variablen in Format-Codes integriert werden und String mit einem ’f’ deklariert wird

```
x = 30.8
p = 3.141592
print("First value = {:.2f}, Second value = {:.4f}".format(x, p))
print(f"First value = {x:.2f}, Second value = {p:.4f}")
```

Formatierte Ausgaben V

type	Bedeutung
'b'	Integer, binär.
'd'	Integer, dezimal.
'o'	Integer, oktal.
'x', 'X'	Integer, hexadezimal (Klein- bzw. Großbuchstaben).
'c'	Zum Integer gehörendes Unicode-Zeichen.
'f'	Float.
'e', 'E'	Float, Exponentialformat (Klein- bzw. Großbuchstaben).
'g', 'G'	adaptiv 'f' oder 'e' bzw. 'E'.
's'	Umwandlung in einen String mittels str()-Methode.
'%'	Umwandlung in Prozent und zusätzliches %-Zeichen.

Formatierte Ausgaben VI

- index: Positions- oder Schlüsselwortparameter
- fill: beliebiges Füllzeichen

align	**Bedeutung**
'<'	Feld wird linksbündig ausgegeben (Standard für Strings).
'>'	Feld wird rechtsbündig ausgegeben (Standard für numerische Werte).
'^'	Feld wird zentriert ausgegeben.
'='	Füllzeichen werden zwischen dem Vorzeichen (falls existent) und der Zahl ausgegeben. Kann nur auf numerische Werte angewendet werden.

Formatierte Ausgaben VII

- width: totale Anzahl an auszugebenden Zeichen
- precision: Präzision des Dezimalanteils

flag	**Bedeutung**
'+'	Ausgabe von positiven und negativen Vorzeichen.
'-'	Ausgabe von negativen Vorzeichen (Standard).
' '	Bei keinem Vorzeichen wird ein Leerzeichen vorangestellt.
'0'	Auffüllen mit Nullen unter Beachtung eines möglichen Vorzeichens (entspricht align '=' und fill '0').
','	Tausender Gruppierungen durch Komma getrennt.
'#'	Binär-, Oktal- und Hexadezimalsystem werden mit einem Präfix versehen.

Code-Strukturierung

- Anstelle von Schlüsselwörtern oder speziellen Klammern, dient die **Einrückung von Zeilen** als Strukturierungselement
- Codeblöcke setzen sich häufig aus einem Anweisungskopf und einem Anweisungskörper zusammen

```
Anweisungskopf:
    Anweisung
        ...
    Anweisung
```

- Wesentlich ist der Doppelpunkt am Ende des Kopfes und die gleichmäßige Einrückung des Anweisungskörpers
- Codeblöcke können geschachtelt werden

Bedingte Anweisungen I

- Eine **bedingte Anweisung** oder **Verzweigung** ist ein Codeteil, der nur unter bestimmten Bedingungen ausgeführt wird.
- Bedingte Anweisungen können geschachtelt werden.

`if`-Anweisung

```
if bedingung:
    anweisung_1
    anweisung_2
        ...
```

- Der eingerückte Block wird nur dann ausgeführt wenn die Bedingung *bedingung* zutrifft, also logisch *true* liefert.

Bedingte Anweisungen II

`elif`- und `else`-Anweisung

- Optionale Erweiterungen der `if`-Anweisung.

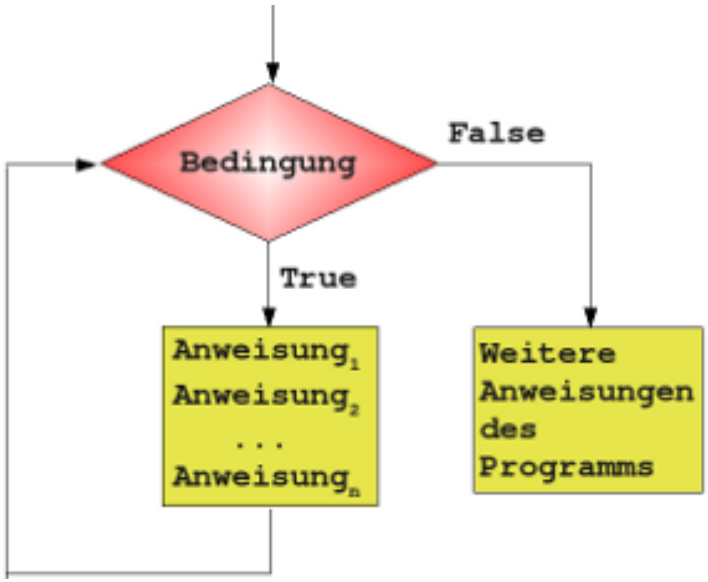
```
if bedingung_1:
    anweisung_1
elif bedingung_2:
    anweisung_2
# weitere elif–Anweisungen
else:
    anweisung_n
```

- Die `elif`-Anweisung hat die gleiche Funktionalität wie die `if`-Anweisung, wird aber nur geprüft wenn die Bedingung des vorherigen `elif` oder `if` *false* war.
- Die `else`-Anweisung wird am Ende ausgeführt wenn keine vorherige Bedingung eingetroffen ist.

Schleifen I

- **Schleifen** ermöglichen es einen Codeblock wiederholt auszuführen, solange das *Schleifenkriterium* erfüllt ist.

- das Schleifenkriterium kann bspw. durch einen Zähler, eine explizite Bedingung oder durch das Durchlaufen einer Sammlung definiert sein.



https://www.python-kurs.eu/python3_schleifen.php

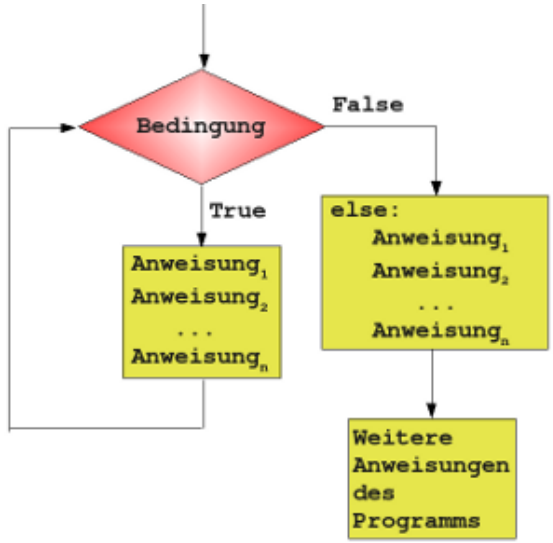
- Schleifen können geschachtelt werden.

Schleifen II

`while`-Schleife

- Die `while`-Schleife wiederholt den eingerückten Codeblock solange die Bedingung *true* liefert.
- Sobald die Bedingung *false* liefert wird die Schleife verlassen bzw. der optionale `else`-Zweig aufgerufen

```
while bedingung:
    anweisung_1
        ...
    anweisung_n
else:
    anweisung_else
```



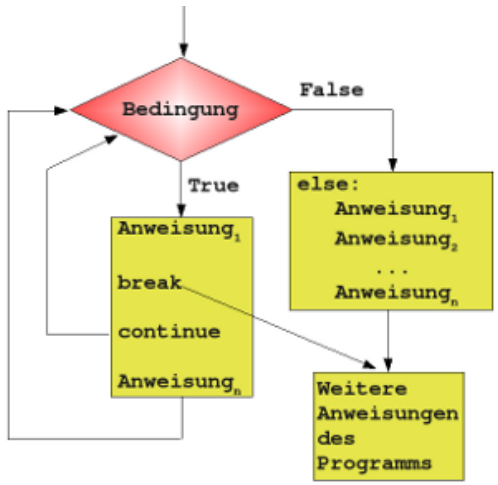
https://www.python-kurs.eu/python3_schleifen.php

Schleifen III

`break`- und `continue`-Anweisung

- Die `break`-Anweisung terminiert sofort die Schleife in der sie enthalten ist (bei geschachtelten Schleifen nur die innerste).
- Die `continue`-Anweisung beendet lediglich den aktuellen Durchlauf und kehrt zum Schleifenkopf zurück.

```
while bedingung_1:
    anweisung_1
    if bedingung_2:
        break
    elif bedingung_3:
        continue
    else:
    anweisung_2
else:
    anweisung_3
```



https://www.python-kurs.eu/python3_schleifen.php

Schleifen IV

`for`-Schleife

- Pythons Art von `for`-Schleife entspricht einer **for-each**-Schleife: Iteration über eine Sequenz von Objekten.

```
for var in sequenz:
    anweisung_1
        ...
    anweisung_n
else:
    anweisung_else
```

- Bei jedem Schleifendurchlauf wird das nächste Element der Sequenz der Variablen *var* zugewiesen.
- Die Schleife terminiert, sobald das letzte Element der Sequenz durchlaufen wurde.

Schleifen V

`range`-Funktion

- Simulation von Zählschleifen mittels der `range`-Funktion.
- `range([begin,]end[,step])`
  liefert einen Iterator über Zahlen im Bereich von 0 bzw. *begin* (inklusive) bis *end* (exklusive) mit Schrittweite 1 bzw. *step*.
- Zählschleife von 1 bis 100:

```
for counter in range(1, 101):
    anweisung_1
        ...
    anweisung_n
```

Funktionen I

- Strukturelement zur Gruppierung von Anweisungen $\rightarrow$ Codewiederverwendung und verbesserte Verständlichkeit

```
def funktionsname(parameterliste):
    """docstring"""
    anweisung(en)
    [return objekt(e)]
```

- Parameterliste kann beliebig viele Argumente enthalten.
- Parameter können obligatorisch und optional sein (letztere werden bei fehlender Übergabe durch Default-Werte ersetzt).
- (optionale) **return**-Anweisung beendet Funktionsaufruf und liefert das(die) zugehörige(n) Objekt(e) bzw. *None* zurück.

Funktionen II

Parameter

- Obligatorische Parameter:

```
def greet(name):
    print(f"Hello␣{name}!")
```

- Optionale Parameter mit Default-Werten:

```
def greet(name="everybody"):
    print(f"Hello␣{name}!")
```

- Funktionenaufruf mittels Schlüsselwortparameter:

```
def my_function(a, b, c=0, d=0):
    return a + b – c – d
print(my_function(5, 10, d=2))
```

Funktionen III

- Funktionen können mit globalen und lokalen Objekten arbeiten

→ Lokal definierte Objekte sind für den globalen Kontext nicht sichtbar

→ Zum Ändern globaler Objekte (anstatt nur zu referenzieren), müssen diese mit `global` gekennzeichnet werden

```
global_var = 0
def func(integer):
    global global_var
    local_var = integer ** 2
    print(global_var)
    global_var += local_var # needs 'global' keyword
func(10)
print(global_var)
# print(local_var) # Error
```

Funktionen IV

Variable Parameteranzahl

- Tupel-Referenz "*" fasst Parameter zu einem Tupel zusammen:

```
def func(*params):
    print(params)
func("My", "answer", "is", 42)
```

- Beliebige Schlüsselwortparameter mittels "**":

```
def func(**key_vals):
    print(key_vals)
func(de="German", en="English", fr="French")
```

- Beachte Semantik von "*" und "**" in Funktionsdefinitionen bzw. -aufrufen: Dienen dem Packen bei Funktionsdefinitionen und Entpacken bei Funktionsaufrufen.

Dateien I

Text aus Datei lesen

- `open`-Funktion erzeugt Dateiobjekt und liefert Referenz auf dieses Objekt.

open(filename[, mode][, encoding])

- Datei zum Lesen öffnen: *mode* = "r" (read, default)

txtfile = **open**("input.txt", "r")

- Zeilenweises Einlesen mittels `for`-Schleife:

```
for line in txtfile:
    print(line.rstrip())
```

- Kodierung beachten (z.B. für Umlaute)!

Dateien II

Datei schließen

- `close`-Funktion schließt Datei nach ihrer Bearbeitung:

```
txtfile.close()
```

- Alternativ benutzt man die Datei innerhalb eines `with`-Blocks:

```
with open("input.txt", "r") as txtfile:
    for line in txtfile:
        print(line.rstrip())
```

- Nach Verlassen des `with`-Blocks wird die Datei automatisch geschlossen.

Dateien III

Schreiben in Datei

- Datei zum Schreiben öffnen: *mode* = "w" (write).
- `write`-Funktion schreibt Daten in Datei.

```
in_file = open("input.txt", "r")
out_file = open("output.txt", "w")
i = 1
for line in in_file:
    out_file.write(str(i) + ": " + line)
    i += 1
in_file.close()
out_file.close()
```

- Achtung: *mode* = "w" löscht die Datei beim Öffnen! Nutze *mode* = "a" zum Anhängen von Daten.

Dateien IV

Komplettes Einlesen

- Datei in eine komplette Datenstruktur einlesen.
  - `readlines`-Funktion liefert eine Liste zurück.
  - `read`-Funktion liefert einen String zurück.

```
text_list = open("input.txt").readlines()
text_string = open("input.txt").read()

print(text_list)
print(text_string)
```

Listen-Abstraktion I

- Listen-Abstraktion (List Comprehension) ist eine elegante Methode um Listen zu erzeugen.
  - Listen dessen Elemente das Ergebnis eines Ausdrucks angewandt auf die Elemente einer anderen Sequenz sind.
  - Listen dessen Elemente eine bestimmte Bedingung erfüllen.
- Aufbau: Eckige Klammern $\rightarrow$ anzuwendender Ausdruck $\rightarrow$ eine `for`-Anweisung $\rightarrow$ optionale `for`- bzw. `if`-Anweisungen.
- Ähnelt der mathematischen Notation von Mengen z.B. Quadratzahlen: $\{x^2 \mid x \in \mathbb{N}\}$)

```
squares = [x**2 for x in range(10)]
# hier nur die ersten 10 Zahlen
```

Listen-Abstraktion II

- Tupel müssen eingeklammert werden, z.B. Elemente zweier Listen kombinieren wenn sie ungleich sind:

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

- Reihenfolge der `for`- und `if`-Anweisungen entspricht sequentieller Ausführung.
- Generatoren-Abstraktion analog mit runden Klammern $\rightarrow$ liefert *Generator*.
- Mengen-Abstraktion analog mit geschweiften Klammern $\rightarrow$ liefert *Menge*.

Modularisierung I

Modulares Design

- Zerlegung eines komplexen Systems in kleinere selbstständige Einheiten oder Komponenten (Module).
- Anstatt eine Funktion zu kopieren, wenn man sie in einem anderen Programm verwenden möchte, sollte man sie in einem Modul speichern.
- Aufteilung eines Quelltextes in Module bezeichnet man als **Modularisierung**.
- Verwendung eigener Module, Module von Drittanbietern oder der Standardbibliothek.

Modularisierung II

`import`-Anweisung

- Einbindung von Modulen mittels der `import`-Anweisung:

```
import math
print(math.sin(math.pi))
```

- Selektives Importieren mittels der `from`-Anweisung:

```
from math import sin, pi
print(sin(pi))
```

- Umbenennen des Namensraumes mittels der `as`-Anweisung:

```
import math as m
from math import sin as sinus
print(sinus(m.pi))
```

Modularisierung III

Suchpfad für Module

- Wenn man ein Modul *module* importiert, sucht der Interpreter nach *module.py* in der folgenden Reihenfolge:
  1. Im aktuellen Verzeichnis.
  2. PYTHONPATH (Umgebungsvariable)
  3. Falls PYTHONPATH nicht gesetzt ist, wird im Default-Pfad gesucht (z.B. /usr/lib/python3.6 [Linux] oder C:\Python\Lib\ [Windows]).
- `sys.path` enthält die Verzeichnisse, in denen Module gesucht werden:

```
import sys
for directory in sys.path:
    print(directory)
```

Modularisierung IV

Inhalt eines Moduls

- Der Inhalt eines Moduls kann mittels der `dir`-Methode abgefragt werden:

```
import math
dir(math)
```

- Ohne Argumente liefer `dir()` die definierten Namen des aktuellen Geltungsbereichs:

```
import math
cities = ["rostock", "berlin", "hamburg"]
dir()
# ['__builtins__', '__doc__', '__name__',
# '__package__', 'cities', 'math']
```

Modularisierung V

Dokumentation eigener Module

- Jedes Modul sollte ausreichend kommentiert sein.

```
# Einzeiliger Kommentar
"""Mehrzeiliger
Kommentar"""
```

- Allgemeine Beschreibung des Moduls durch *Docstring* zu Beginn der Moduldatei.
- Funktionen und Quellcode werden wie üblich dokumentiert.

Modularisierung VI

Packages

- Zusammenfassen mehrerer Module in einem Paket (package).
- Zusätzlich muss es noch eine Datei mit dem Namen __init __.py enthalten.
- Kann leer sein oder Python-Code enthalten, der beim Import des Paketes ausgeführt werden soll.
- Pakete werden wie normale Module importiert:

```
from package import module_a, module_b
module_a.func()
module_b.func()
```

Numerisches Python I

NumPy

- Python Modul für "numerisches" Programmieren

→ Grundlegende Datenstrukturen und wesentliche mathematische Funktionalitäten

→ Sehr schnell und effizient im Vergleich zu nativen Python Listen

- Mehrdimensionale arrays (Skalare, Vektoren, Matrizen, . . .)

```
import numpy as np
values = [1,2,3,4,5,6,7,8]
x = np.array(values)
print(x) # [1 2 3 4 5 6 7 8]
print(type(x)) # <class 'numpy.ndarray'>
```

- Überblick über alle NumPy Routinen:

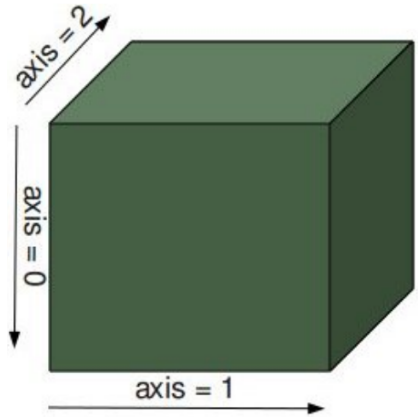
  `https://numpy.org/doc/stable/reference/routines.html`

Numerisches Python II

Array Dimensionen

- Rang = Anzahl der Dimensionen (Skalar)
- Shape = Größe der Dimensionen (Tupel)

```
y = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(y.ndim) # 3
print(y.shape) # (2, 2, 2)
```



https://www.python-kurs.eu/numpy_arrays_erzeugen.php

Numerisches Python III

Indizierung und Slicing

- Für jede Dimension: "[start:stop:step]"

```
print(x[0]) # 1
print(x[2:–1]) # [3 4 5 6 7]
print(x[:4]) # [1 2 3 4]
print(x[::2]) # [1 3 5 7]
```

Siehe `https://numpy.org/doc/stable/user/basics.indexing.html`

2D Beispiel

```
z = np.array([
    [11, 12, 13, 14, 15],
    [21, 22, 23, 24, 25],
    [31, 32, 33, 34, 35],
    [41, 42, 43, 44, 45],
    [51, 52, 53, 54, 55]])
print(z[2, 2]) # 33
print(z[3:, :–1]) # [[41 42 43 44]
                  # [51 52 53 54]]
print(z[:, 3]) # [14 24 34 44 54]
print(z[::3, ::–1]) # [[15 14 13 12 11]
                    # [45 44 43 42 41]]
```

Numerisches Python IV

Array Initialisierungen

```
np.array() # direkt
np.ones(shape) # Einsen
np.zeros(shape) # Nullen
np.full(shape, value) # Array voll mit 'value'
# gleichverteilte Werte aus [start, stop), analog zu range()
np.arange([start,] stop[, step])
# 'num' viele gleichverteilte Werte aus [start, stop] bzw. [start, stop)
np.linspace(start, stop[, num=50][, endpoint=True])
np.random.rand(*shape) # Zufaellige floats aus [0,1]
np.random.randint([start,] stop, shape) # Zufaellige integer aus [start, stop)
```

. . . und viele weitere (siehe auch `https://numpy.org/doc/stable/reference/routines.array-creation.html`)

Numerisches Python V

Array Operationen

- Generell **elementweise**

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a + 10) # [[11 12 13]
              # [14 15 16]]
print(a ** 2) # [[ 1 4 9]
              # [16 25 36]]
```

```
b = np.full((2, 3), 3)
print(a – b) # [[–2 –1 0]
             # [ 1 2 3]]
print(a * b) # [[ 3 6 9]
             # [12 15 18]]
```

- Oder spezielle algebraische Methoden (z.B. Matrixmultiplikation)

```
c = np.transpose(a) # Transponierte Matrix
print(np.matmul(a, c)) # [[14 32]
                       # [32 77]]
```

Numerisches Python VI

- Lassen sich Arrays unterschiedlicher shape miteinander kombinieren?

Broadcasting - Idee

- Beschreibt wie Arrays unterschiedlicher Form während arithmetischen Operationen behandelt werden.
- Unter bestimmten Einschränkungen wird das kleinere Array auf das größere Array "übertragen" (broadcast), sodass ihre Formen kompatibel sind.
- Skalare werden immer auf das gesamte Array übertragen

Numerisches Python VII

Broadcasting - Regeln

- Beim Operieren auf zwei Arrays werden ihre shapes elementweise von hinten nach vorne verglichen.
- Zwei Dimensionen sind kompatibel, wenn
  1) sie gleich sind.
  2) eine von ihr 1 ist.
- Die Größe des resultierenden Arrays ist die maximale Größe entlang jeder Dimension der Eingabearrays.
- Arrays müssen nicht den gleichen Rang haben um broadcast-kompatibel zu sein, es müssen nur alle elementweisen Vergleiche der Dimensionen kompatibel sein.

Numerisches Python VIII

Broadcasting - Beispiele

A (3d):	$2 \times 3 \times 5$
B (1d):	1
$A \bullet B$ (3d):	$2 \times 3 \times 5$

A (2d):	5×4
B (1d):	4
$A \bullet B$ (2d):	5×4

A (3d):	$15 \times 3 \times 5$
B (3d):	$15 \times 1 \times 5$
$A \bullet B$ (3d):	$15 \times 3 \times 5$

A (3d):	$15 \times 3 \times 5$
B (2d):	3×5
$A \bullet B$ (3d):	$15 \times 3 \times 5$

A (2d):	2×1
B (2d):	2×3
$A \bullet B$ (2d):	2×3

A (2d):	6×1
B (3d):	$7 \times 1 \times 5$
$A \bullet B$ (3d):	$7 \times 6 \times 5$

A (1d):	3
B (1d):	5
$A \bullet B$ (—):	*error*

A (2d):	2×1
B (3d):	$8 \times 4 \times 3$
$A \bullet B$ (—):	*error*

Graphiken mit `Matplotlib`

- Python-Bibliothek zum Plotten (Daten, Graphen, Bilder etc.)
- Untermodel `pyplot` als Schnittstelle zur Plot-Bibliothek von Matplotlib

  import matplotlib.pyplot as plt

- Alle `pyplot`-Funktionen beziehen sich auf eine Abbildung (figure) und Plot-Bereich (axes)
- `pyplot` legt automatisch *eine* figure im Hintergrund an, sodass man sich bei einfachen Plots darum nicht kümmern muss
- Abbildungen werden angezeigt sobald `plt.show()` aufgerufen wird

Plotten mit `pyplot`

`plt.plot`

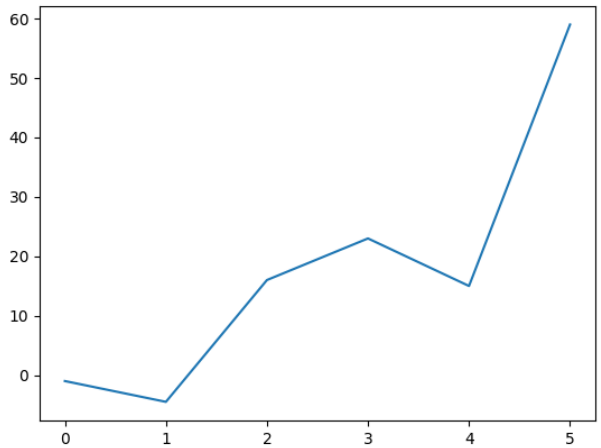
- Zum plotten von (x, y)-Datenpunkten
- Arbeitet auf Listen
  - Eine einzelne Liste wird als y-Werte interpretiert, mit ihren Indizes als x-Werte
  - Zwei Listen werden als y- und x-Werte interpretiert
- Unterstützt Format-Parameter zum Anpassen der Darstellung
  - Linienstil
  - Darstellung der diskreten Werte
  - Farbe

`plt.plot` – Beispiele

- Einzelne Liste als y-Werte (implizite x-Werte)

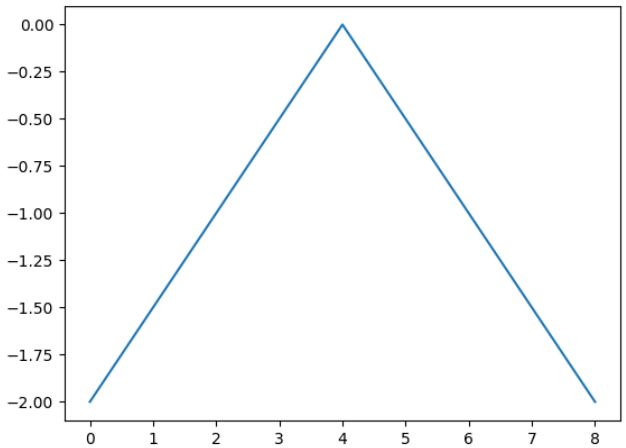
```
import matplotlib.pyplot as plt

plt.plot([–1, –4.5, 16, 23, 15, 59])
plt.show()
```



- Zwei Listen als explizite x- und y-Werte

```
xs = [0, 2, 4, 6, 8]
ys = [–2, –1, 0, –1, –2]
plt.plot(xs, ys)
plt.show()
```

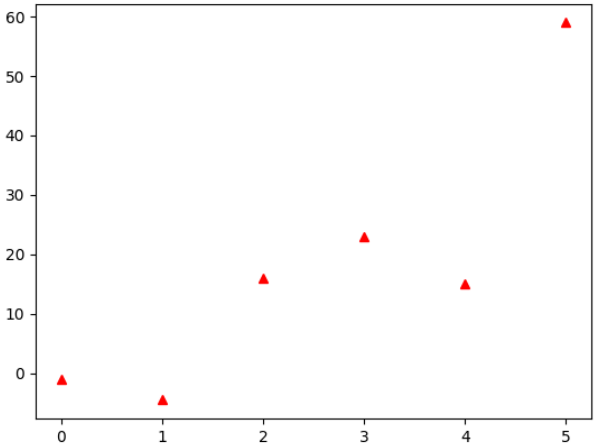


`plt.plot` – Formatierung I

- Format-Parameter enthält `[marker][line][color]`

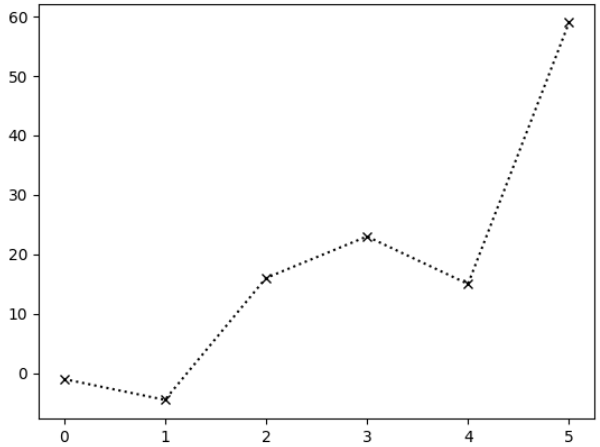
- `'^r'` → rote Dreiecke ohne Linie

```
plt.plot([–1, –4.5, 16, 23, 15, 59], '^r')
plt.show()
```



- `'x:k'` → schwarze Kreuze mit gepunkteter Linie

```
plt.plot([–1, –4.5, 16, 23, 15, 59], 'x:k')
plt.show()
```



`plt.plot` – Formatierung II

character	marker
`.`	point
`,`	pixel
`o`	circle
`v`	triangle down
`^`	triangle up
`<`	triangle left
`>`	triangle right
`1`	tri down
`2`	tri up
`3`	tri left
`4`	tri right

character	marker
`s`	square
`p`	pentagon
`*`	star
`h`	hexagon1
`H`	hexagon2
`+`	plus
`x`	cross
`D`	diamond
`d`	thin diamond
`|`	vertical line
`_`	horizontal line

`plt.plot` – Formatierung III

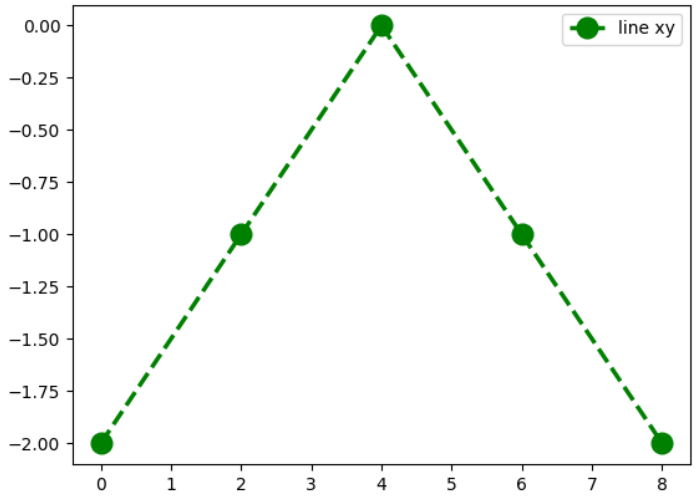
character	line style
`-`	solid
`--`	dashed
`-.`	dash-dot
`:`	dotted

character	color
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

`plt.plot` – Formatierung IV

- (Weitere) Argumente zur Formatierung können auch direkt an die `plot`-Funktion übergeben werden:

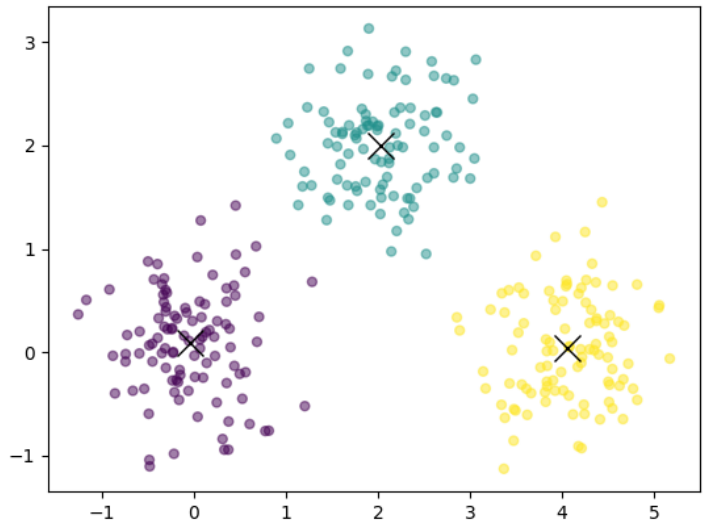
```
plt.plot(xs, ys,
        color='green',
        marker='o',
        linestyle='dashed',
        linewidth=2.5,
        markersize=12,
        label='line xy', ...)
plt.legend() # for the label
plt.show()
```



Streudiagramme (Punkt-Plots)

- Mittels `plt.plot`, indem die Verbindungslinien ausgeblendet werden
- Flexibler: `plt.scatter`
  $\rightarrow$ Farbe und Größe der Markierungen (und weiteres) können *individuell* gesetzt werden

```
plt.scatter(points_x, points_y, s=25
            c=points_color, alpha=0.5)
plt.plot(centroids_x, centroids_y,
         marker='x', markersize=15,
         markeredgecolor="black",
         linestyle="None")
plt.show()
```



`figure` – Formatierung I

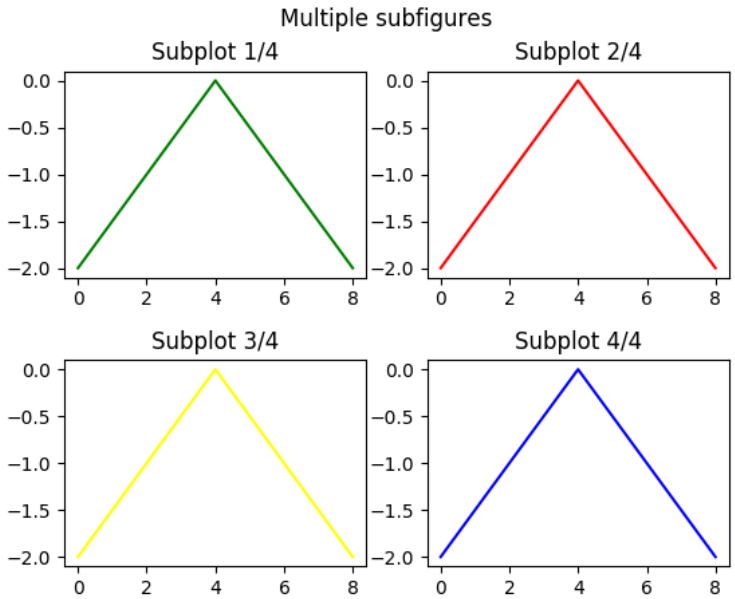
- Achsenbeschriftungen mittels
  `plt.xlabel('x-Achse')` und `plt.ylabel('y-Achse')`
- Wertebereich der Achsen anpassen mittels
  `plt.axis([xmin, xmax, ymin, ymax])`
  oder Achsen ausblenden mittels
  `plt.axis("off")`
- Titel der Abbildung mittels `plt.title('Titel')`

`figure` – Formatierung II

`plt.subplot`

- Mehrere Abbildungen in einer `figure`
- Argumente: `plt.subplot(rows, cols, pos)`
  → Aufteilung in *rows* Zeilen und *cols* Spalten
  → Untergraph an der Position *pos* (Index von 1 bis *rows*∗*cols*)

```
colors = ['green', 'red', 'yellow', 'blue']
plt.suptitle("Multiple subfigures")
for i, c in enumerate(colors):
    plt.subplot(2, 2, i+1)
    plt.plot(xs, ys, color=c)
    plt.title("Subplot {}/4".format(i+1))
plt.subplots_adjust(hspace=0.4)
plt.show()
```



Bilder

plt.imshow

- Darstellen von Bildern auf einem 2D-Raster
- Input Daten 2D $\rightarrow$ Zeilen und Spalten des Bildes
- Optionale dritte Dimension für RGB(A) Werte

```
# 'imgs' enthaelt sechs 28x28 Bilder
for i in range(6):
    plt.subplot(2, 3, i+1)
    # mit grauer colormap
    plt.imshow(imgs[i], cmap="gray")
    # ohne Achsen
    plt.axis("off")
plt.show()
```