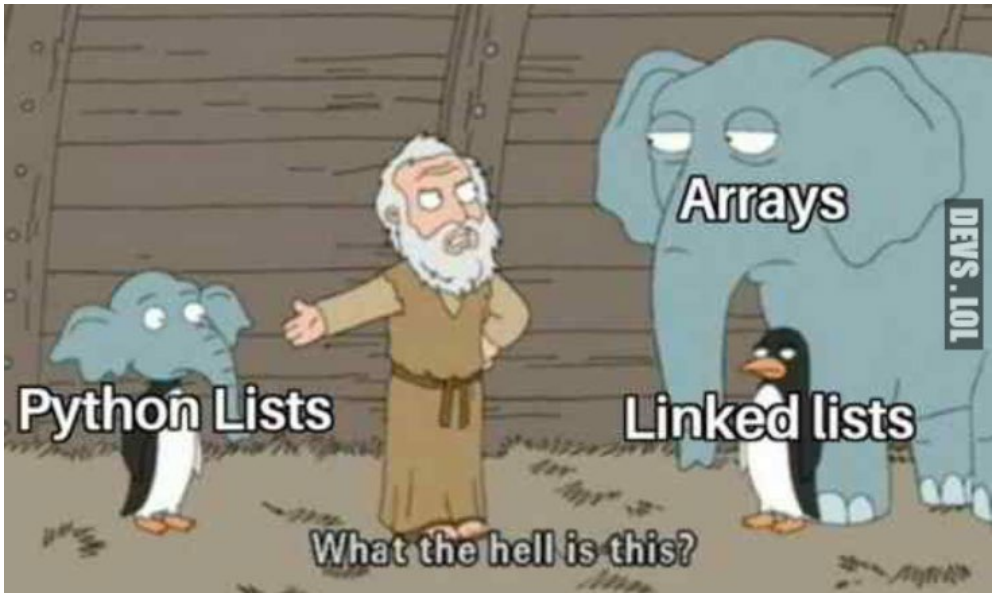
Lösungsalgorithmen & Programmieren

- Block 01 /Abend 03-

Datenstrukturen: Kollektionen in Python



**) Python-Listen sind weder klassische Arrays noch echte verkettete Listen, sondern ein hybrider, dynamischer Datentyp, der von aussen einfach aussieht, aber intern komplex ist*

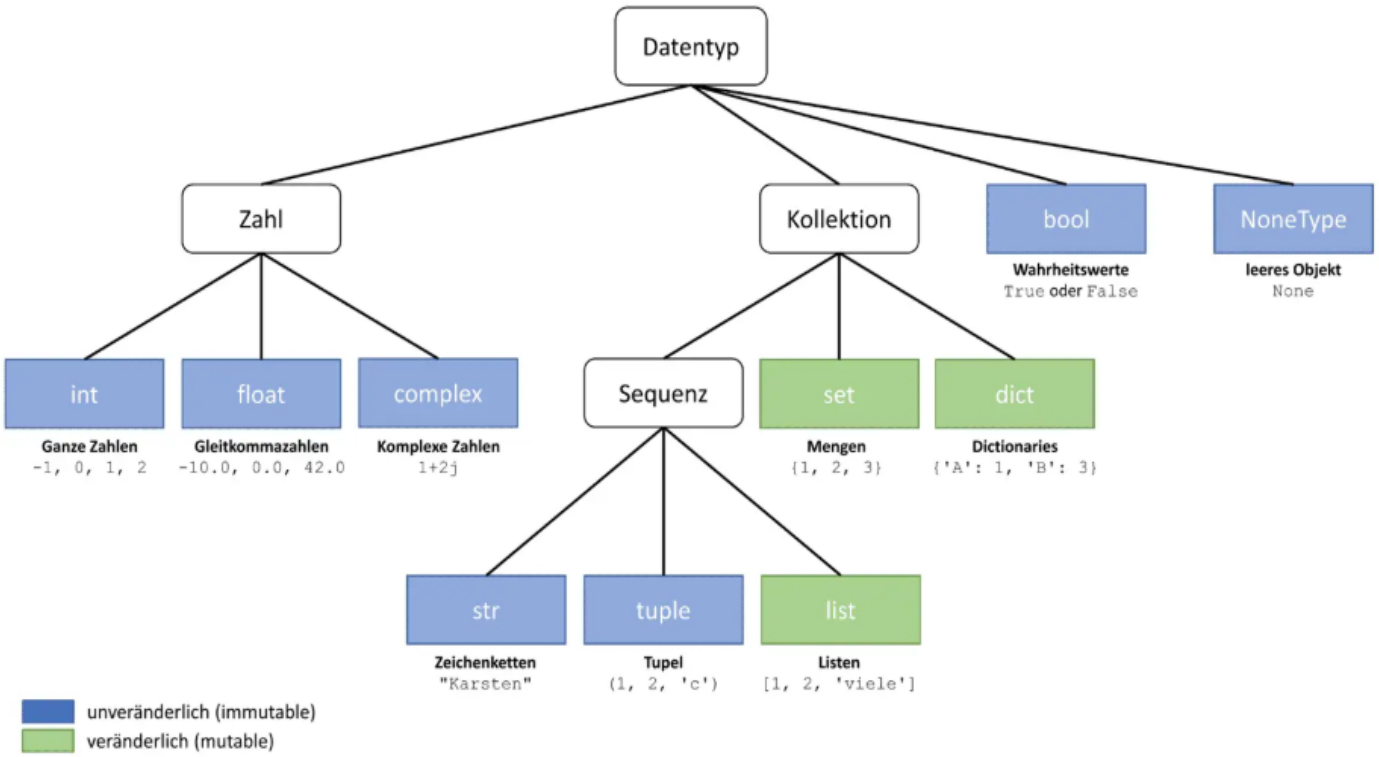
🎯 Leistungs-/Lernziele dieses Blocks:

Die Studierenden …
können Pythonprogramme mit Verzweigungen und Wiederholungen umsetzen (Repetition)
kennen grundlegende Datenstrukturen in Python (Listen, Tupel, Sets, Dictionaries) und können deren Unterschiede beschreiben
können Listen und Dictionaries erstellen, verändern und ausgeben.
können Elemente hinzufügen, ändern und löschen (append(), insert(), remove(), etc.).
können über Datenstrukturen iterieren, um deren Inhalte gezielt zu verarbeiten
können einfache Aufgabenstellungen mit Listen oder Dictionaries zur Datenspeicherung und -auswertung lösen

Inhaltsverzeichnis:

1 Kollektionen

Python kennt eine Vielzahl von Datentypen, die sich grob in Zahlen, logische Werte, leere Objekte und Sammlungen (Kollektionen) unterteilen lassen. Die folgende Grafik zeigt, wie diese Typen miteinander verwandt sind und welche davon veränderlich (mutable) oder unveränderlich (immutable) sind.



Veränderliche und unveränderliche Typen:
Zu den unveränderlichen Typen gehören z. B. ganze Zahlen (int), Gleitkommazahlen (float), Zeichenketten (str) oder Tupel (tuple). Diese Objekte können nach ihrer Erstellung nicht mehr verändert werden. Python erstellt bei jeder Änderung ein neues Objekt.
Die veränderlichen Typen wie Listen (list), Mengen (set) oder Dictionaries (dict) erlauben es dagegen, ihren Inhalt nachträglich anzupassen – etwa Elemente hinzuzufügen, zu löschen oder zu verändern.

Innerhalb der Kollektionen unterscheidet man verschiedene Strukturen:

Sequenzen (z.B. str, tuple, list)	sind geordnete Sammlungen, auf deren Elemente über Indizes zugegriffen werden kann
Mengen (set)	sind ungeordnete Sammlungen ohne doppelte Werte
Mappings (dict)	sind Zuordnungen vom Typ *Schlüssel → Wert*.

1.1 Tupel ( )

Ein Tupel ist eine geordnete Sammlung mehrerer Werte, die – im Gegensatz zu Listen – nicht verändert werden kann. Man bezeichnet Tupel deshalb als unveränderlich (immutable).
Tupel werden häufig verwendet, um mehrere logisch zusammengehörende Werte als eine Einheit zu speichern, zum Beispiel Koordinaten, Farbcodes oder feste Datensätze.

Bei einer *Tupelzuweisung* werden mehrere Werte in einer einzigen Anweisung zugewiesen. Die linke Seite des nebenstehdnen Codes ist gleichbedeutend mit der rechten Seite.

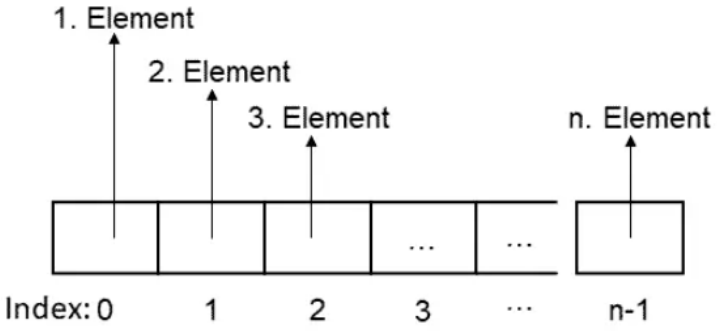
a=1 b=2 c=3	a, b, c = 1, 2, 3

Weil zuerst die rechte Seite vom Zuweisungsoperator ausgewertet wird, bevor die Zuweisung an die links stehenden Variablen erfolgt, kann die Tupelzuweisung dazu verwendet werden, die Wertzuweisungen von Variablen zu tauschen. Ohne Tupelzuweisung ist dazu eine Hilfsvariable erforderlich.

a, b = 1 , 2 a, b = b, a print(a, b)	Ausgabe: `2 1`

Sowohl Tupel als auch Listen sind in Python geordnete Sequenzen von Elementen. Jedes Element hat eine feste Position (Index) und kann über diesen Index angesprochen werden.

In vielen anderen Programmiersprachen (z. B. C, Java) müssen alle Elemente eines Arrays denselben Datentyp besitzen. In Python dagegen dürfen die Datentypen gemischt sein – eine Sequenz kann also Zahlen, Zeichenketten, Wahrheitswerte oder sogar andere Listen enthalten.
Es ist jedoch nicht korrekt, sich vorzustellen, die Daten lägen „im" Tupel oder „in" der Liste.



Tatsächlich speichert jede Sequenz Referenzen (Zeiger) auf eigenständige Datenobjekte im Speicher. Das erklärt, warum auch komplexe Strukturen wie verschachtelte Listen problemlos funktionieren.

Tupel werden mit **runden Klammern ()** erstellt. Die einzelnen Elemente werden **durch Kommas getrennt**.

koord = (47.05, 8.30)
farbe = ("rot", 255, 0, 0)
print(koord)
print(farbe)
	Ausgabe:
	`(47.05, 8.3)`
	`('rot', 255, 0, 0)`

Ein Tupel kann Objekte beliebigen Typs enthalten:

daten = ("Luzern", 6020, 25.5, True) print(daten)	Ausgabe: `('Luzern', 6020, 25.5, True)`

Bei nur einem Element ist das **Komma zwingend erforderlich**, sonst würde Python dies als Wert in Klammern interpretieren:

x = (5)
print(type(x))
y = (5**,**)
print(type(y))
	Ausgabe:
	`<class 'int'>`
	`<class 'tuple'>`

Auf die einzelnen Elemente kann mit dem Index-Operator [ ] zugegriffen werden. Die Indexierung beginnt bei 0. Negative Indizes zählen von hinten.

```
zahlen = (10, 20, 30, 40, 50)
print(zahlen)       # gibt den ganzen Inhalt aus
print(zahlen[0])    # einzelnes Element → 10
print(zahlen[4])    # einzelnes Element → 50
print(zahlen[-1])   # ebenfalls letztes Element → 50
```

Ausgabe:

```
(10, 20, 30, 40, 50)
10
50
50
```

Nach der Erstellung kann man den Inhalt eines Tupels nicht mehr ändern. Versucht man, ein Element zu überschreiben, erhält man einen Fehler:

```
zahlen = (10, 20, 30, 40, 50)
zahlen[1] = 88
```

Ausgabe:

```
Exception has occurred: TypeError ×
'tuple' object does not support item assignment
```

Wenn man dennoch einen neuen Wert möchte, muss man ein neues Tupel erzeugen (sinnvollerweise mit einer while-Schleife):

```
zahlen = (10, 20, 30, 40, 50)
zahlen2 = (zahlen[0], 88, zahlen[2], zahlen[3], zahlen[4])
print(zahlen2)
```

Ausgabe:

```
(10, 88, 30, 40, 50)
```

Tupel sind iterierbar, das heisst man kann sie in Schleifen durchlaufen:

```
zahlen = (10, 20, 30, 40, 50)
for element in zahlen:
    print(element, end=" ; ")
print()
auto = ("Hans Glas GmbH", "Goggomobil T", 1954)
for element in auto:
    print(element)
```

Ausgabe:

```
10 ; 20 ; 30 ; 40 ; 50 ;
Hans Glas GmbH
Goggomobil T
1954
```

Tupel können andere Tupel oder auch Listen enthalten:

```
geo = ("Luzern", (47.05, 8.30) )
print( geo[1][0] )
```

Ausgabe:

```
47.05
```

Tupel eignen sich, um mehrere Werte in einem Schritt zu speichern oder zurückzugeben.

```
# Packing: mehrere Werte zu einem Tupel zusammenfassen
punkt = (10, 20)
# Unpacking: Werte wieder auf einzelne Variablen verteilen
x, y = punkt
print("x =", x)
print("y =", y)
```

Ausgabe:

```
x = 10
y = 20
```

Das Tupel selbst ist unveränderlich, wenn es aber veränderliche Elemente (z. B. **Listen**) enthält, kann deren Inhalt trotzdem geändert werden:

```
daten = ("Temperaturen", [12, 15, 14] )
daten[1].append(16)
print(daten)
```

Ausgabe:

```
('Temperaturen', [12, 15, 14, 16])
```

Nützliche Funktionen bei Tupeln und Listen: len liefert die Anzahl Elemente, count liefert die Anzahl der Vorkommen und index liefert den Index des ersten Vorkommens:

```
autos=("VW", "Toyota", "VW", "BMW", "Audi", "VW", "BMW")
anz= len(autos)
anzRot = autos.count("VW")
pos= autos.index("Audi")
print (anz, anzRot, pos)
```

Ausgabe:

```
7 3 4
```

1.2 Listen [ ]

Eine Liste ist wie ein Tupel eine geordnete Sammlung mehrerer Werte, mit dem entscheidenden Unterschied, dass sie veränderlich (mutable) ist. Das bedeutet: Man kann nach der Erstellung Elemente hinzufügen, ändern oder löschen.
Listen sind in Python eine der wichtigsten Datenstrukturen. Sie dienen als Grundlage für viele komplexere Strukturen (z. B. Sets, Dictionaries, DataFrames, etc.).

Listen werden mit eckigen Klammern [] erstellt. Die einzelnen Elemente werden durch Kommas getrennt.

```
noten = [5.0, 4.5, 5.5, 4.0]
farben = ["rot", "grün", "blau"]
print(noten)
print(farben)
```

Ausgabe:
```
[5.0, 4.5, 5.5, 4.0]
['rot', 'grün', 'blau']
```

Eine Liste kann verschiedene Datentypen enthalten:

```
mix = ["Luzern", 6020, True, 12.5]
print(mix)
```

Ausgabe:
```
['Luzern', 6020, True, 12.5]
```

Der Zugriff erfolgt analog wie bei den Tupeln über den Index-Operator []. Die Indexierung beginnt bei 0. Negative Indizes zählen von hinten. Ein Zugriff auf einen nicht existierenden Index erzeugt einen Fehler.
Da Listen im Gegensatz zu Tupeln veränderlich sind, kann der Inhalt nachträglich geändert werden:

```
zahlen = [10, 20, 30, 40, 50]
zahlen[1] = 88
print(zahlen)
```

Ausgabe:
```
[10, 88, 30, 40, 50]
```

Bei Listen kann man zusätzliche **Elemente hinzufügen**:

Methode	**Beschreibung**	**Beispiel**
append(x)	fügt x am Ende hinzu	zahlen.append(60)
insert(i, x)	fügt x an Position i ein	zahlen.insert(2, 99)
extend(liste2)	Hängt eine weitere Liste an	zahlen.extend([70, 80])

```
zahlen = [10, 20, 30]
zahlen.append(40)
zahlen.insert(1, 15)
print(zahlen)
zahlen2 = [50, 80]
zahlen.extend(zahlen2)
print(zahlen)
```

Ausgabe:
```
[10, 15, 20, 30, 40]
[10, 15, 20, 30, 40, 50, 80]
```

Bei Listen kann man **Elemente entfernen**:

Methode	**Beschreibung**	**Beispiel**
remove(x)	fügt x am Ende hinzu	farben.remove(“rot“)
pop(i)	entfernt Element an Index i (Standard: letztes Element)	farben.pop(1)
clear()	Löscht alle Elemente	Farben.clear()

```
farben = ["rot", "blau", "grün", "rot"]
farben.remove("rot")  # entfernt das erste "rot"
print(farben)
farben.pop()          # entfernt das letzte Element
print(farben)
```

Ausgabe:
```
['blau', 'grün', 'rot']
['blau', 'grün']
```

Wie Tupel sind Listen iterierbar und können bequem mit einer for-Schleife durchlaufen werden:

```
noten = [5.0, 4.5, 5.5, 4.0]
for wert in noten:
    print(wert, end=" ; ")
```

Ausgabe:

```
5.0 ; 4.5 ; 5.5 ; 4.0 ;
```

Mit Hilfe des **Slicing-Operators** (Doppelpunkt innerhalb des Indexoperators) kann man Teilbereiche einer Liste kopieren oder umkehren. Slicing erzeugt immer eine neue Liste, das Original bleibt unverändert.
Der Slicing-Operator hat drei wichtige Parameter: liste[start:stop:step]

```
zahlen = [10, 20, 30, 40, 50]
print( zahlen[1:4] )     # vom (inkl.) 1ten bis (exkl.) 4ten
print( zahlen[:3] )      # vom (inkl.) 0ten bis (exkl.) 3ten
print( zahlen[3:] )      # vom (inkl.) 3ten bis und mit dem letzten
print( zahlen[::-1] )    # alle Elemente aber rückwärts weil step -1
```

Ausgabe:

```
[20, 30, 40]
[10, 20, 30]
[40, 50]
[50, 40, 30, 20, 10]
```

Zusätzlich zu den Funktionen len(), count() und index(), die wir bereits bei den Tupeln kennengelernt haben, verfügen Listen über Funktionen die Tupel wegen der Unveränderbarkeit nicht haben. Solche Funktionen sind z.B. sort(), reverse() und copy()

Methode	**Beschreibung**	**Beispiel**
Sort()	Sortiert Liste (in-place) **	zahlen.sort()
Reverse()	Kehrt die Reihenfolge um	zahlen.reverse()
copy()	Erstellt eine flache Kopie ***	neu = zahlen.copy()

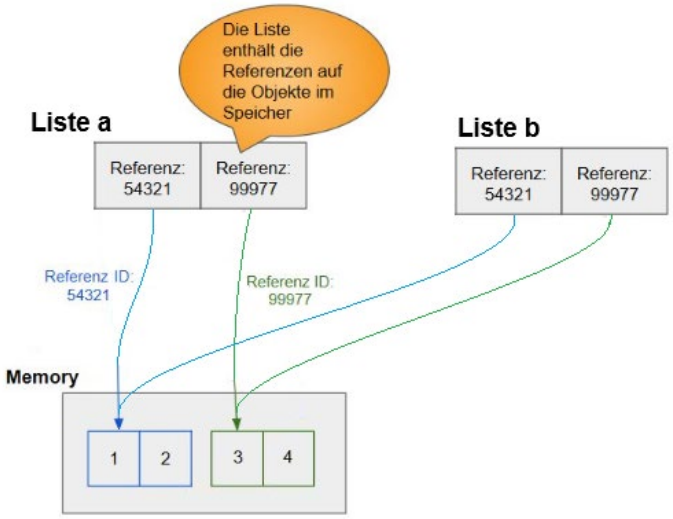
**) "in-place" bedeutet: direkt im bestehenden Speicherobjekt, also ohne eine neue Liste zu erzeugen. Die Methode sort() verändert die Original-Liste selbst. Es gibt keine Rückgabe, sort() liefert None zurück. Die Zuweisung x=zahlen.sort() funktioniert also nicht! Wenn du stattdessen eine neue, sortierte Liste willst, verwende die Funktion sorted(liste).

***) Eine flache Kopie ist eine neue Liste, die auf dieselben Elemente zeigt wie das Original. Sie kopiert also nur die äussere Struktur, nicht die inneren Objekte.

```
a = [ [1, 2], [3, 4] ]
b = a.copy()          # flache Kopie
b[0].append(99)
print(a)
print(b)
```

Ausgabe:

```
[[1, 2, 99], [3, 4]]
[[1, 2, 99], [3, 4]]
```



Alias vs. Kopie

Beim direkten Zuweisen wird nur die Referenz kopiert. beide Variablen zeigen auf dieselbe Liste!

a = [1, 2, 3]
b = a # kein neues Objekt!
b.append(4)
print(a)
	Ausgabe:
	`[1, 2, 3, 4]`

Richtig kopieren:

a = [1, 2, 3] **b = a.copy()** b.append(4) print(a)	Ausgabe: `[1, 2, 3]`

Sortieren und Kopieren:

zahlen = [5, 2, 9, 1]
zahlen.sort()
print(zahlen)
kopie = zahlen.copy()
kopie.append(100)
print(kopie)
print(zahlen)
	Ausgabe:
	`[1, 2, 5, 9]`
	`[1, 2, 5, 9, 100]`
	`[1, 2, 5, 9]`

Verschachtelte Listen - Listen können andere Listen enthalten:

noten = [
  ["Anna", [5.0, 5.5, 5.0]],
  ["Ben", [4.0, 4.5, 5.0]]
]
print(noten[0][1][1])
	Ausgabe:
	`5.5`

Projekt: Gemeinsam das Beispiel Temperaturen zum Thema Listen lösen		
Schwierigkeitsgrad: ⭐	⏳ Zeit: 15 Minuten	Arbeitsform: Plenum 👫
Wir werden gemeinsam ein Beispiel zum Thema Listen. Erstelle die folgende Liste in deiner App: **temperaturen = [18, 21, 19, 22, 20, 18, 17]** Danach möchten wir folgende Aufgaben erledigen: 01) Gib die Temperaturwerte einzeln in einer Zeile aus. 02) Berechne den Durchschnitt 03) Ersetze einen Messfehler: Das zweite Element sollte nicht 21 sondern 20 sein. Ersetze den Wert. 04) Ergänze die zusätzliche Messung (20) am Ende der Liste 05) Gib die höchste und die niedrigste Temperatur aus		
Tipp zu 01: Verwende die for Schleife um durch die Liste zu iterieren Tipp zu 02: Verwende die Funktionen sum() und len() um den Durchschnitt zu berechnen Tipp zu 03: Verwende den Indexoperator [ ] um die richtige Stelle zu überschreiben Tipp zu 04: Verwende die Funktion append() um etwas am Schluss anzufügen Tipp zu 05: Verwende die Funktionen max() und min() zur Ermittlung der gesuchten Werte		

```python
temperaturen = [18, 21, 19, 22, 20, 18, 17]
# 01) Gib die Temperaturwerte einzeln in einer Zeile aus.
for wert in temperaturen:
    print(wert, end=" ")
print()   # Zeilenumbruch
# 02) Berechne den Durchschnitt.
durchschnitt = sum(temperaturen) / len(temperaturen)
print("Durchschnitt:", round(durchschnitt, 1))
# 03) Ersetze einen Messfehler:
# Das zweite Element (Index 1) sollte nicht 21 sondern 20 sein.
temperaturen[1] = 20
print("Nach Korrektur:", temperaturen)
# 04) Ergänze die zusätzliche Messung (20) am Ende der Liste.
temperaturen.append(20)
print("Nach Ergänzung:", temperaturen)
# 05) Gib die höchste und die niedrigste Temperatur aus.
print("Höchste Temperatur:", max(temperaturen))
print("Niedrigste Temperatur:", min(temperaturen))
```

```
18 21 19 22 20 18 17
Durchschnitt: 19.3
Nach Korrektur: [18, 20, 19, 22, 20, 18, 17]
Nach Ergänzung: [18, 20, 19, 22, 20, 18, 17, 20]
Höchste Temperatur: 22
Niedrigste Temperatur: 17
```

1.3 Sets { }

Ein Set (Menge) ist eine ungeordnete Sammlung von einzigartigen Elementen. Es gibt keine doppelten Werte und keine feste Reihenfolge. Sets eignen sich hervorragend, um Mitgliedschaften zu prüfen, doppelte Einträge zu entfernen oder Mengenoperationen wie Vereinigung, Schnittmenge oder Differenz auszuführen.

Listen werden mit geschweiften Klammern { } erstellt oder mit der Funktion set(). Die Ausgabe kann in beliebiger Reihenfolge erscheinen, da Sets ungeordnet sind.

```
noten = {5.0, 4.5, 5.5, 4.0}
farben = {"rot", "grün", "blau"}
print(noten)
print(farben)
```

Ausgabe:

```
{5.5, 4.0, 4.5, 5.0}
{'blau', 'grün', 'rot'}
```

Doppelte Werte werden automatisch entfernt:

```
zahlen = {1, 2, 3, 3, 2}
print(zahlen)
```

Ausgabe:

```
{1, 2, 3}
```

Sets werden oft genutzt, um:

- doppelte Einträge aus einer Liste zu entfernen
- Mitgliedschaften zu prüfen (in)
- Schnittmengen, Vereinigungen und Differenzen zu bilden

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
print(a | b)        # Vereinigung
print(a & b)        # Schnittmenge
print(a - b)        # Differenz
print(a ^ b)        # Symmetrische Differenz
```

Ausgabe:

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{1, 2, 5, 6}
```

Da Sets veränderlich sind, können sie erweitert oder verkleinert

```
tiere = {"Hund", "Katze"}
tiere.add("Pferd")
tiere.remove("Hund")
print(tiere)
```

Ausgabe:

```
{'Katze', 'Pferd'}
```

Wie bei Listen kann man mit einer Schleife über die Elemente iterieren, nur ist die Reihenfolge nicht garantiert:

```
zahlen = {10, 20, 30}
for z in zahlen:
    print(z, end=" ; ")
```

Ausgabe:

```
10 ; 20 ; 30 ;
```

Typische Funktionen

Methode	Beschreibung	Beispiel
len(s)	Liefert Anzahl der Elemente	x=len(zahlen)
add(x)	Fügt ein Element hinzu	zahlen.add(99)
remove(x)	Entfernt ein Element (Fehler bei Nichtvorhandensein)	zahlen.remove(3)
discard(x)	Entfernt ein Element (ohne Fehler)	zahlen.discard(99)
clear()	Leert das Set	zahlen.clear()

Das folgende Programm, entfernt die doppelten Einträge aus einer Liste:

```
namen = ["Anna", "Ben", "Anna", "Clara", "Ben"]   #List
einzigartig = set(namen)                          #Set
print(einzigartig)
```

Ausgabe:

```
{'Clara', 'Anna', 'Ben'}
```

1.4 Mappings (Dictionaries)

Ein Dictionary (Wörterbuch) speichert Paare von Schlüssel und Wert (auch Key-Value-Paare) genannt.
Man spricht daher von einer Mapping-Struktur, weil jedem Schlüssel genau ein Wert zugeordnet ist.

```
noten = {"Anna": 5.0, "Ben": 4.5, "Clara": 5.5}
print(noten["Ben"])
```

Ausgabe:
```
4.5
```

- Schlüssel (key) müssen eindeutig und unveränderlich sein (z. B. str, int, tuple)
- Werte (value) können beliebige Datentypen haben
- Die Reihenfolge ist ab Python 3.7 stabil (Einfügereihenfolge bleibt erhalten)

```
personen = {"Name": "Max", "Alter": 25}
print(personen["Name"])

personen["Ort"] = "Luzern"    # neues Paar hinzufügen
personen["Alter"] = 26        # Wert ändern
del personen["Name"]          # Element löschen
print(personen)
```

Ausgabe:
```
Max

{'Alter': 26, 'Ort': 'Luzern'}
```

Dictionaries werden in Python sehr häufig verwendet:

- zum Speichern strukturierter Daten
- für Konfigurationen, JSON-Daten, API-Resultate, Zähler oder Tabellen
- als Grundlage vieler komplexerer Datentypen (z. B. DataFrames)

Iteration über Dictionaries:

```
noten = {"Anna": 5.0, "Ben": 4.5, "Clara": 5.5}

for schluessel, wert in noten.items():
    print(schluessel, "→", wert)
```

Ausgabe:
```
Anna → 5.0
Ben → 4.5
Clara → 5.5
```

Typische Funktionen

Methode	Beschreibung	Beispiel
keys()	liefert alle Schlüssel	noten.keys()
values()	liefert alle Werte	noten.values()
items(x)	liefert alle Schlüssel-Wert-Paare	noten.items()
get(k, default)	gibt den Wert zu k zurück oder default	noten.get("Anna", 0)
pop(k)	entfernt ein Paar und gibt den Wert zurück	noten.pop("Ben")
clear()	Löscht alle Einträge	noten.clear()

Das folgende Programm, erstellt ein Wörterbuch für Länder und Hauptstädte und gib sie formatiert aus:

- linke Seite = Schlüssel (Key) → eindeutig
- rechte Seite = Wert (Value) → beliebig

```
hauptstädte = {
    "Schweiz": "Bern",
    "Deutschland": "Berlin",
    "Italien": "Rom"
}
for land, stadt in hauptstädte.items():
    print(f"Die Hauptstadt von {land} ist {stadt}.")
```

Ausgabe:
```
Die Hauptstadt von Schweiz ist Bern.
Die Hauptstadt von Deutschland ist Berlin.
Die Hauptstadt von Italien ist Rom.
```

2 Miniprojekte realisieren

Löst im Alleingang möglichst viele der untenstehenden Problemstellungen.

2.1 Projekt 01 (Notenverwaltung):

📝 **Projekt 01: Notenverwaltung**		
Schwierigkeitsgrad: ⭐	⏳ Zeit: 30 Minuten	Arbeitsform: Alleine 👤

Schreiben Sie ein Programm, das den Benutzer zur Eingabe von sechs Noten auffordert. Speichere alle Werte in einer Liste. Erledige danach folgende Detailaufgaben:
a) Berechne den Durchschnitt und gib ihn gerundet auf eine Zehntelnote aus
b) Finde die beste und schlechteste Note und gib diese aus:
c) Sortiere die Liste und gib sie sortiert aus
d) Erzeuge eine Teilliste mit den drei besten Noten (Slicing!) und gib diese Liste aus

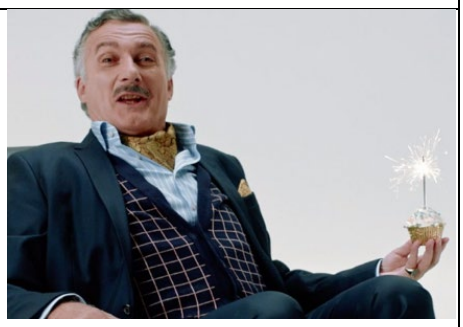
Option 1: ⭐ ⭐
Erweitere das Programm so, dass nicht immer genau 6 Noten eingegeben werden müssen. Die BenutzerIn wird solange zur Eingabe von Noten aufgefordert, bis die Eingabe „-1“ das Ende der Eingabe definiert. Danach läuft es ab wie oben beschrieben.

Option 2: ⭐ ⭐
Versuche den Notendurchschnitt nicht auf ein Zehntel genau, sondern auf eine halbe Note gerundet auszugeben.

2.2 Projekt 02 (Lotto-Tipp):

Projekt 02: Lotto-Tipp

Schwierigkeitsgrad: ⭐ ⭐	⏳ Zeit: 40 Minuten	Arbeitsform: Alleine



Schreiben Sie mit Hilfe von Schleifen, Verzweigungen und Listen ein Programm, welches ihnen einen Lotto-Tipp mit zufälligen Zahlen generiert. Ein Tipp besteht immer aus sechs Zahlen (1-42) und einer Glückszahl (1-6).
Der Lottotipp soll keine doppelten Zahlen enthalten und soll in aufsteigender Reihenfolge ausgegeben werden:
zum Beispiel:

LottoTipp: **2; 3; 11; 26; 28; 37; Glückszahl: 5**

siehe: www.swisslos.ch/de/swisslotto/information/spielanleitung/swisslotto_game_system/einzeltipps.html

Tipp 1:
Die Funktion randint(a, b) gibt eine ganze Zahl zwischen a und b (inklusive) zurück.

```
import random
zahl = random.randint(1, 100)
print(zahl)
```

2.3 Projekt 03 (Secret Intelligence Service):

Projekt 03: Secret Intelligence Service

Schwierigkeitsgrad: ⭐⭐⭐⭐	Zeit: 60 Minuten	Arbeitsform: Gruppenarbeit

Sie sind als GeheimagentIn beim MI6 tätig und arbeiten damit für King Charles III.
Es gelingt dem MI6 immer wieder verschlüsselte Nachrichten abzufangen. 005 hat herausgefunden, dass die Nachrichten mit der **Caesar-Chiffre** verschlüsselt wurden. Allerdings ist unklar, welche Verschiebung angewendet wird, da diese mit jeder Nachricht wechselt. Ihr seid zu zweit als Team tätig und erledigt die Mission gemeinsam. „Person A implementiert die Verschlüsselung (Teil 2), Person B die Häufigkeitsanalyse (Teil 3).
Zum Schluss testet ihr beide Programme gemeinsam, indem ihr eine Nachricht verschlüsselt und mit der Häufigkeitsanalyse versucht, die Verschiebung zu bestimmen."

Ihre Mission:

01) Lest euch als Team in die Funktionsweise der Caesar-Chiffre ein:
02) Um dem Feind schnell mit falschen Informationen antworten zu können benötigt der MI6 ein Programm, bei dem man einen Text und eine Verschiebung eingeben kann und als Resultat den nach der Caesar-Chiffre verschlüsselten Text zurückbekommt (Ausgabe).

    Beispieltext : DIESE NACHRICHT IST STRENG GEHEIM Verschiebung: +7

    Verschlüsselter Text : KPLZL UHJOYPJOA PZA ZAYLUN NLOLPT

    (Durch eine weitere Verschiebung um 26-7=19 bekäme man wieder den ursprünglichen Text)
03) Damit man schneller ermitteln kann, welche Caesar-Chiffre-Verschiebung gewählt wurde, musst du ein Programm schreiben, dass die Häufigkeiten aller Buchstaben eines einzugebenden Textes ermittelt. Leerzeichen sollen nicht berücksichtigt werden. Gross- und Kleinschreibung soll nicht unterschieden werden. Als Ausgabe sollen die häufigsten 5 und die seltensten 5 Zeichen mit der Anzahl der Vorkommnisse aufgelistet werden. Versuche damit abzuleiten, welche Verschiebung gewählt wurde. Übersetze gleich die neuste eingetroffene Nachricht:

    QRE OHF SNRUEG NZ ZBETRA RVA ZVG ENFCHGVA

    → Welche Verschiebung wurde gewählt?

    → Wie lautet die Originalnachricht?

Funktionsweise der Caesar-Chiffre:

In der Kryptografie bezeichnet man eine Caesar-Chiffre als einfache Verschiebung des Alphabets: Jeder Buchstabe wird durch einen anderen ersetzt, der *x* Positionen weiter rechts im Alphabet steht.
Beispiel (Verschiebung +3):

Klartext	**A**	**B**	**C**	**D**	**E**	**F**	**...**
Geheimtext	D	E	F	G	H	I	...

Der Klartext **„HALLO"** wird damit zu **„KDOOR"**.

Falls du weitere Infos zur Caesar-Verschlüsselung benötigst, findest du sie hier:
https://de.wikipedia.org/wiki/Caesar-Verschl%C3%BCsselung)
Caesar Verschlüsselung Online - kryptowissen.de

Buchstabenhäufigkeit

Die Buchstabenhäufigkeit gibt an, wie oft ein bestimmter Buchstabe in einem Text vorkommt.
Siehe https://de.wikipedia.org/wiki/Buchstabenh%C3%A4ufigkeit.
Im Deutschen ist der Buchstabe E mit 17.4 % mit Abstand am häufigsten. Der Buchstabe N mit 9.78% am zweithäufigsten etc. Der Buchstabe Q kommt mit 0.02% am wenigsten häufig vor.
Beachten Sie, dass die Umlaute *ä, ö* und *ü* wurden wie *ae, oe* und *ue gezählt wurden.*