4.3 Mengen

Mengen (*sets*) sind ungeordnete Kollektionen ohne Duplikate, die iterierbar und veränderbar sind. Damit bezeichnet man sie auch als *iterable* und *mutable*. Die Elemente einer Menge haben keine Reihenfolge, damit auch keine Indizes und zählen somit auch nicht zu den sequentiellen Datentypen.

Um Mengen unveränderbar (*immutable*) zu machen gibt es den Datentyp `frozenset`.

Zum Erzeugen einer Menge verwendet man wie in der Mathematik geschweifte Klammern (alternativ mit dem Schlüsselwort `set(...)`)

In [1]:
```
# TODO Beispiel Menge
menge = {1,2,3,4}
type(menge)
leereMenge = set()
print(leereMenge)
set([1,2, 1, 1, 2])
set("abaacceef")
frozen = frozenset([1, 2, 1, 1, 2])
print(frozen)
#frozen.add(4) # frozenset ist immutable
set((("Python", "Perl"), ("Paris", "Berlin")))
# es dürfen nur immutables in untere Ebenen der Menge
# set((["Python", "Perl"], ["Paris", "Berlin"])) # geht nicht
```

```
set()
frozenset({1, 2})
```

Out[1]:
```
{('Paris', 'Berlin'), ('Python', 'Perl')}
```

Übliche Operatoren aus der Mathematik: Schnittmenge `&`, Vereinigung `|` und Differenz `-`

In [11]:
```
# Mengen-Operatoren aus der Mathematik
m1 = set("Einstein")
m2 = set("Relativitaet")
print(m1 & m2)  # Schnittmenge
print(m1 | m2)  # Vereinigung
print(m1 - m2)  # Differenz
m1
```

```
{'i', 'e', 't'}
{'i', 'a', 't', 'e', 'E', 'v', 'R', 'l', 'n', 's'}
{'E', 'n', 's'}
```

Out[11]:
```
{'E', 'e', 'i', 'n', 's', 't'}
```

Methoden für weitere Operationen auf Mengen:

Methoden	Beschreibung
`menge.add(e)`	Fügt das Element e in die Menge menge als neues Element ein
`menge.clear()`	Entfernt alle elemente aus der Menge menge

Methoden	Beschreibung
`menge.discard(e)`	Das Element e wird aus der Menge menge entfernt.
`menge.copy()`	(Flache) Kopie der Menge menge
`menge.difference(andereMenge)`	= `menge - andereMenge`
`menge.intersection(andereMenge)`	= `menge & andereMenge`
`menge.union(andereMenge)`	= `menge | andereMenge`

Mengenabstraktion (*set comprehension*)

Vergleich von Listen- und Mengenabstraktion am Algorithmus "Sieb des Eratosthenes" zur Ermittlung der Primzahlen von 2 bis n.

In [12]:
```
# Via list comprehension
from math import sqrt
n = 75
sqrt_n = int(sqrt(n))
no_primes = [j for i in range(2, sqrt_n) for j in range(i*2, n, i)]
print(no_primes)
primes = [i for i in range(2, n) if i not in no_primes]
print(primes)
```

```
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44,
46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 6, 9, 12, 15, 18, 21,
24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 8, 12, 16, 2
0, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 10, 15, 20, 25, 30, 35, 4
0, 45, 50, 55, 60, 65, 70, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 14, 21, 2
8, 35, 42, 49, 56, 63, 70]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73]
```

In [13]:
```
# Via set comprehension
from math import sqrt
n = 75
sqrt_n = int(sqrt(n))
no_primes = {j for i in range(2, sqrt_n) for j in range(i*2, n, i)}
print(no_primes)
primes = {i for i in range(2, n) if i not in no_primes}
print(primes)
```

```
{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33,
34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58, 6
0, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74}
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73}
```

4.4 Dictionary

Neben Listen sind *Dictionaries* eines der bedeutendsten Datenstrukturen in Python. Es handelt sich dabei um eine *ungeordnete Sammlung von Schlüssel-Wert-Paaren*. In den Programmiersprachen spricht man dann auch von einem *assoziativen Feld*.

Einschub: Was ist ein *assoziatives Array (Feld)*?

> Ein assoziatives Array ist eine Datenstruktur, die – anders als ein gewöhnliches Feld – nichtnumerische (oder nicht fortlaufende) Schlüssel (zumeist Zeichenketten) verwendet, um die enthaltenen Elemente zu adressieren. Diese sind in keiner festgelegten Reihenfolge abgespeichert. Idealerweise werden die Schlüssel so gewählt, dass eine für die Programmierer nachvollziehbare Verbindung zwischen Schlüssel und Datenwert besteht.

Die Schlüssel (*keys*) dürfen nur unveränderliche (*immutable*) Datentypen sein. Die Dictionaries selbst sind *mutable*.

Die wichtigsten Methoden für Operationen auf Dictionaries:

Methoden	Beschreibung
`d.keys()`	Gibt die Schlüssel der Dictionaries d zurück
`d.values()`	Gibt die Werte des Dictionaries d zurück
`d.items()`	Gibt eine Liste mit Tupeln zurück. Jedes Tupel enthält ein Schlüssel-Wert-Paar aus dem Dictionary d
`d.has_key(k)`	Überprüft, ob der Schlüssel k im Dictionary d enthalten ist
`del d[k]`	Löscht das Schlüssel-Wert-Paar mit dem Schlüssel k aus dem Dictionary d
`k in d`	Überprüft, ob k ein Schlüssel des Dictionarys d ist

In [27]:
```
# Beispiele zu Dictionaries
# In {}-Klammern wie Mengen, einzelne Paare durch "," getrennt, ":" unterscheide
waehrungen = {"Deutschland" : "Euro", "Indien" : "Indische Rupie",
              "Grossbritannien" : "Pfund Sterling", "Japan" : "Yen",
              "Frankreich" : "Euro"}

# TODO Zugriff auf ein Element über Schlüssel
#print(waehrungen["Deutschland"])
#print(waehrungen["Japan"])

# TODO "Ist Schlüssel in Dictionary?"
#print("Drin" if "Italien" in waehrungen else "Nicht drin")

# TODO gib Schlüssel aus
for schluessel in waehrungen:
    print(schluessel)
# dasselbe wie:
for schluessel in waehrungen.keys():
    print(schluessel)

# TODO gib Werte aus
for wert in waehrungen.values():
    print(wert)

# TODO gib Paare aus
for w1, w2 in
    #print(paare)
```

```
    #w1, w2 = paare
    print(w1, w2)
```

```
Deutschland
Indien
Grossbritannien
Japan
Frankreich
Deutschland
Indien
Grossbritannien
Japan
Frankreich
Euro
Indische Rupie
Pfund Sterling
Yen
Euro
Deutschland Euro
Indien Indische Rupie
Grossbritannien Pfund Sterling
Japan Yen
Frankreich Euro
```

- Dictionary aus Listen erzeugen: die `zip` -Funktion

Die `zip()` -Funktion kann auf eine beliebige Anzahl an iterierbaren Objekten angewendet werden und gibt ein zip-Objekt zurück, bei dem es sich um einen Tupel-Iterator handelt. Zuerst liefert sie ein Tupel mit den ersten Elementen der Eingabeobjekte, dann die zweiten, dritten und stoppt, sobald eines der iterierbaren Objekte aufgebraucht ist.

In [28]:
```
# Beispiel mit Zahlen und Buchstaben
einige_buchstaben = ["a", "b", "c", "d", "e", "f"]
einige_zahlen = [5, 3, 7, 9, 11, 2]
print(zip(einige_buchstaben, einige_zahlen))
print(type(zip(einige_buchstaben, einige_zahlen)))
for t in zip(einige_buchstaben, einige_zahlen):
    print(t)
```

```
<zip object at 0x000002D9165365C0>
<class 'zip'>
('a', 5)
('b', 3)
('c', 7)
('d', 9)
('e', 11)
('f', 2)
```

In [29]:
```
# Beispiel für unterschiedlich lange Eingabeobjekte
ort = ["Helgoland", "Kiel", "Berlin-Tegel"]
luftdruck = (1021.2, 1019.9, 1023.7, 1023.1, 1027.7)
for ort, ld in zip(ort, luftdruck):
    print(f"Der Luftdruck in {ort} beträgt: {ld:7.1f}")
```

```
Der Luftdruck in Helgoland beträgt:  1021.2
Der Luftdruck in Kiel beträgt:  1019.9
Der Luftdruck in Berlin-Tegel beträgt:  1023.7
```

In [31]:
```
# TODO Dictionary aus Listen, Beispiel Währungen
l = ["Deutschland", "Indien", "Großbritannien", "Japan", "Frankreich"]
w = [ "Euro", "Indische Rupie","Pfund Sterling", "Yen", "Euro" ]
d = dict(zip(l, w))
print(d)
```

```
{'Deutschland': 'Euro', 'Indien': 'Indische Rupie', 'Großbritannien': 'Pfund Ster
ling', 'Japan': 'Yen', 'Frankreich': 'Euro'}
```

4.5 Generatoren und Iteratoren

Neben Sequenzen (*sequentielle Datentypen*) und Mengen gibt es auch *Generatoren*. Im Gegensatz zu den ersten beiden werden Daten nicht explizit gespeichert, sondern erst bei Bedarf erzeugt. Generatoren werden daher auch als *virtuelle Kollektionen* bezeichnet.

Die Vorteile sind:

- weniger Speicherplatz
- schneller

Betrachten Sie folgende Liste mit zehn Elementen:

In [32]:
```
s = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Jedes Element ist im Arbeitsspeicher abgelegt und kann bei Bedarf abgerufen werden. Wenn es niemals abgerufen wird, ist es trotzdem da und beansprucht Speicherplatz. Abstrakt könnte man diese Liste auch wie folgt definieren:

> *Die Folge aller ganzen Quadratzahlen, die kleiner als 100 sind*

Damit ist die Sequenz exakt festgelegt, ohne ein einziges Element explizit zu erzeugen oder zu nennen.

Generatoren entsprechen damit einer Konstruktionsvorschrift, mit der bei Bedarf jedes Element generiert werden kann. Sie können auf zweierlei Weise definiert werden: *Generatorausdrücke* oder *Generatorfunktionen*.

4.5.1 Generatorausdrücke

Generatorausdrücke ähneln sehr der *list comprehension*. Statt eckigen verwenden Sie einfach *runde* Klammern. Ein Generatorausdrück für die obigen Liste könnte wie folgt aussehen:

In [34]:
```
# TODO Generatorausdruck für Liste
s_gen = (i**2 for i in range(10))
print(s_gen)
for n in s_gen:
    print(n,  )
```

```
<generator object <genexpr> at 0x000002D916476E90>
0 1 4 9 16 25 36 49 64 81
```

Generatorausdrücke werden auch gerne verwendet, um Mengen zu definieren und können direkt als Argument in die `set()` -Funktion übernommen werden:

In [36]:
```
# TODO Mengendefinition mit Generatorausdruck
menge = set(i**2 for i in range(10))
print(menge)
```

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

4.5.2 Generatorfunktionen

Generatorfunktionen sind Funktionen, die ein Generator-Objekt zurückgeben. Sie unterscheiden sich von normalen Funktionen durch die `yield` -Anweisung. Eine Generatorfunktion zur Erzeugung von Quadratzahlen von 0 bis $n-1$ lautet daher:

In [38]:
```
# TODO Generatorfunktion
def generiereQuadrate(n):
    for i in range(n):
        yield i*i

obj = generiereQuadrate(10)
print(obj)
for n in obj:
    print(n, end = " ")
```

```
<generator object generiereQuadrate at 0x000002D916475B10>
0 1 4 9 16 25 36 49 64 81
```

Folgendes passiert bei der Ausführung von `yield` :

- Ausdruck hinter `yield` wird *zurückgegeben* (wie bei `return`)
- aktuelle Funktionsausführung wird *unterbrochen*
- aktueller Zustand des zur Funktion gehörenden Prozesses wird *gemerkt*
- wenn das nächste Generator-Objekt verlangt wird, wird die Funktion *nach* dem `yield` *fortgesetzt*

→ Man kann also die zu erzeugenden Elemente im Gegensatz zu Listen nicht in beliebiger Reihenfolge lesen, sondern nur vorne beginnend in der vorgegebenen Reihenfolge. Die Funktion `next()` liefert das nächste Element der vom Generator erzeugten Folge.

In [44]:
```
# TODO weiteres Beispiel zu Generatoren
def simpleGenerator():
    x = 1
    print("Vor dem ersten yield")
    yield x
    x = 2
    yield x
    x = 3
    yield x
    print("Kein weiteres yield")

gen_obj = simpleGenerator()
```

In [45]:
```
# Erzeugen und Ausgabe der Generatorobjekte durch wiederholtes Ausführen dieser
next(gen_obj)
```

```
Vor dem ersten yield
```

Out[45]: 1

Das Besondere an Generatoren ist damit auch, dass man mit ihnen unendliche (virtuelle) Kollektionen erzeugen kann, wie z.B. eine unendliche Folge der Quadratezahlen:

In [47]:
```
def unendlicheQuadrate():
    i = 1
    while True:
        yield i*i
        i += 1

quad = unendlicheQuadrate()
```

In [80]:
```
next(quad)
```

Out[80]: 1089

Die `islice`-Methode aus dem Modul `itertools`:

In [81]:
```
import itertools
erstenFuenf = itertools.islice(unendlicheQuadrate(), 0, 5)      # liefert iterto
list(erstenFuenf)
```

Out[81]: [1, 4, 9, 16, 25]

4.5.3 Iteratoren

Iteratoren sind spezielle Generatoren, die den Zugriff auf die Elemente einer Kollektion oder während einer Iteration kontrollieren, z.B. liefert eine Iterator zu einer Menge nach und nach alle Elemente der Menge.

Beim Funktionsaufruf `next(iterator)` gibt der Iterator `iterator` einer Kollektion ein Element zurück. Die Standardfunktion `iter()` liefert zu einer Sequenz oder einem anderen iterierbaren Objekt einen Iterator. Man kann auch mehrere Iteratoren zu einer Sequenz erzeugen:

In [93]:
```
# TODO Iteratoren
l = [1, 2, 3, 4]
i1 = iter(l)
i2 = iter(l)
i1
```

Out[93]:
```
<list_iterator at 0x2d916a15420>
```

In [100…
```
# TODO Aufruf Iterator 1
next(i1)
```

Out[100…
```
4
```

In [101…
```
# TODO Aufruf Iterator 2
next(i2)
```

Out[101…
```
4
```

Achtung: *Iterator* vs. *Iterable* vs. *Iteration*

4.5.4 Anwendungen von Generatoren

Generell kann man sagen, dass bei Programmen mit sehr großen Datenmengen durch die Verwendung von Generatoren viel Speicherplatz gespart werden kann, da die Objekte *just in time* generiert werden. Auch einige Operationen für Sequenzen sind auf Generator-Objekte anwendbar (z.B. `min(), max(), in, not in`). Jedoch nur einmal, da nach Abruf der Elemente einer Kollektion diese "verbraucht" sind.

In [102…
```
# Beispiel 1 für Zufallszahlen
import random
zufall = (random.randint(0,100) for i in range(10))     # Generatorausdruck für
print("1. Verwendung")
for i in zufall:
    print( i, end = " ")    # 1. Verwendung
print("\n2. Verwendung")
for i in zufall:
    print( i, end = " ")    # 2. Verwendung
```

```
1. Verwendung
86 86 60 31 45 38 0 7 24 81
2. Verwendung
```

In [103…
```
# Beispiel 2 für Funktionswerte einer Parabel
parabel = (x**2 - 2*x + 3 for x in range (-10,10))
print(f"Minimum = {min(parabel)}")
print(f"Maximum = {max(parabel)}")      # Fehler, da parabel leere Sequenz
```

```
Minimum = 2
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[103], line 4
      2 parabel = (x**2 - 2*x + 3 for x in range (-10,10))
      3 print(f"Minimum = {min(parabel)}")
----> 4 print(f"Maximum = {max(parabel)}")

ValueError: max() arg is an empty sequence
```

In [105…

```python
# rekursive Funktion zur Berechnung der Summe 1 bis n
def rekSum(n):
    if n == 1:
        return n
    else:
        return n + rekSum(n-1)

rekSum(2973)
rekSum(2974)       # geht schon nicht mehr
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[105], line 9
      6         return n + rekSum(n-1)
      8 rekSum(2973)
----> 9 rekSum(2974)

Cell In[105], line 6, in rekSum(n)
      4     return n
      5 else:
----> 6     return n + rekSum(n-1)

Cell In[105], line 6, in rekSum(n)
      4     return n
      5 else:
----> 6     return n + rekSum(n-1)

    [... skipping similar frames: rekSum at line 6 (2970 times)]

Cell In[105], line 6, in rekSum(n)
      4     return n
      5 else:
----> 6     return n + rekSum(n-1)

RecursionError: maximum recursion depth exceeded
```

In [108…

```python
# Beispiel 3: unendlicher Summen-Generator
def genSum():
    n = 1
    erg = n
    while True:
        yield erg
        n += 1
        erg += n

def ausgabeSumme(n):
    #gen = genSum()
    counter = 0
    for x in genSum():
        counter += 1
        if counter == n:
            print(x)
            break

#g = genSum()
ausgabeSumme(10000)
```

```
50005000
```

4.6 Vertiefung: Rekursive Funktionen für Sequenzen

4.6.1 Summe aller Elemente einer Liste

Aufgabe: die Summe der Elemente einer Liste von Zahlen berechnen

Lösungsidee:

- Eine leere Zahlenliste hat die Summe null.
- Ansonsten ist die Summe gleich der ersten Zahl in der Liste plus die Summe der restlichen Liste (Beispiel: `summe([1, 2, 3]) = 1 + summe([2, 3])`)

In [109…

```
# TODO Rekursives Summieren einer Liste
def summe(liste):
    if len(liste) == 0:
        return 0
    else:
        return liste[0]+summe(liste[1:])

summe([2, 4, 6, 8, 10])
```

Out[109… 30

4.6.2 Rekursive Suche

Bekannt: `in`-Operator, um zu überprüfen, ob ein Element in einer Datenstruktur enthalten ist.

Jetzt: Suche in einer Sequenz `s` alle Elemente, die eine bestimmte Eigenschaft haben

Grundidee für rekursiven Suchalgorithmus:

- Wenn die Sequenz `s` nur aus einem Element besteht, prüfe, ob dieses eine Element den geforderten Eigenschaften entspricht
- Wenn die Sequenz `s` aus mehreren Element besteht, zerteile `s` in zwei etwa gleich große Teile `s1` und `s2`

In der Informatik bezeichnet man diese algorithmische Idee auch als *teile und herrsche* (*divide and conquer*), nach dem berühmten Ausspruch des römischen Feldherrn Julius Caesar.

Beispiel mit Liste von Telefonnummern mit bestimmter Vorwahl:

In [110…

```
# Rekursive Vorwahl-Suche
nummernliste = ['0223 788834', '0201 566722', '0224 66898', '0201 899933', '0208
def suche(num, vorwahl):
    if len(num) == 1:
        if num[0][:len(vorwahl)] == vorwahl:    # 1
            return num
        else:
            return []

    else:
```

```
        return suche(num[:len(num)//2], vorwahl) + suche(num[len(num)//2:], vorw

print(suche(nummernliste, '0201'))
```

```
['0201 566722', '0201 899933']
```

Erklärung:

- 1. Elementarer Fall: die Liste `num` besteht nur aus einem Element `num[0]`, welcher wiederrum aus einem String besteht. Der Slice `num[0][:len(vorwahl)]` wird mit der gesuchten Vorwahl verglichen.
- 2. Bei mehr als einem Element in der Liste `num`, wird diese aufgeteilt und es werden zwei Slices gebildet. Die Grenze ist der Index `len(num)/2`. Es folgt der rekursive Aufruf der Funktion auf beide Slices.

In [ ]: