Inhalt - Python

1 Einleitung

Diese Dokumentation führt in die Programmierung in Python ein. Es werden sowohl elementare Grundlagen wie Variablendeklarationen, Datentypen als auch weiterführende Themen wie Rekursion, Exception Handling und Objektorientierung behandelt.
Darüber hinaus erfolgt eine Beschreibung eines Krypto-Tools, das klassische Verschlüsselungs- und Analyseverfahren in Python implementiert.

2. GRUNDLAGEN DER PROGRAMMIERUNG IN PYTHON

2.1 Variablen und Datentypen

In Python erfolgt die Deklaration einer Variablen durch die direkte Zuweisung eines Wertes:

```
zahl = 42          # Integer
text = "Hallo"     # String
wert = 3.14        # Float
flag = True        # Boolean
```

Integer (ganzzahlige Werte): Eignen sich für Zählvorgänge oder Berechnungen ohne Nachkommastellen.
Float (Fließkommazahlen): Dienen zur Repräsentation von Zahlen mit Dezimalstellen.
String (Zeichenkette): Speichert Textinformationen. In Python können Strings in einfache oder doppelte Anführungszeichen gesetzt werden.
Boolean (True/False): Bildet eine wichtige Grundlage für Verzweigungen und logische Operationen.

Python verwendet **dynamische Typisierung**, sodass eine Variable ihren Typ anhand des aktuell zugewiesenen Wertes erhält. Eine explizite Typangabe wie in statisch typisierten Sprachen entfällt.
Type Casting: Wenn eine Datentyp-Umwandlung erforderlich ist, etwa von String zu Integer, bietet Python eingebaute Funktionen wie `int(), float(), str()`:

```
eingabe = input("Bitte eine Zahl eingeben: ")  # Liefert String
zahl = int(eingabe)          # Konvertiert String in eine ganze
Zahl
```

2.2 Datenstrukturen

2.2.1 Listen

Eine Liste (`list`) ist eine **geordnete, veränderbare** Sequenz beliebiger Python-Objekte. Sie eignet sich, wenn Elemente hinzugefügt, entfernt, sortiert oder insgesamt flexibel verwaltet werden sollen.

```
meine_liste = [10, 20, 30]
meine_liste.append(40)    # Liste wird zu [10, 20, 30, 40]
erstes_element = meine_liste[0]  # 10
```

Typische Methoden einer Liste:

`append(x)`: Hängt das Element x am Ende an.

`remove(x)`: Entfernt das erste Vorkommen von x.

`pop(i)`: Entfernt das Element am Index i und gibt es zurück.

`insert(i, x)`: Fügt x an Indexposition i ein.

`clear()`: Leert die Liste vollständig.

Listen-Slicing

Ein sehr mächtiges Feature in Python ist das **Slicing** einer Liste. Es werden bestimmte Teilbereiche basierend auf Start-, End- und optionalem Schrittindex extrahiert.

```
liste = [0, 1, 2, 3, 4, 5, 6]

# 1) liste[a:b] liefert die Elemente von Index a (inklusive)
#    bis Index b (exklusive).
teil_1 = liste[1:4]  # [1, 2, 3]

# 2) liste[a:] liefert alle Elemente ab Index a bis zum Ende.
teil_2 = liste[3:]   # [3, 4, 5, 6]

# 3) liste[:b] liefert alle Elemente vom Start bis Index b
(exklusive).
teil_3 = liste[:3]   # [0, 1, 2]

# 4) Negative Indizes ermöglichen das Zählen vom Ende her.
teil_4 = liste[-3:]  # [4, 5, 6]

# 5) liste[a:b:c] - Schrittweiten (c). z.B. c=2
teil_5 = liste[0:7:2]  # [0, 2, 4, 6]
```

Randfälle:

- Wenn der Startindex größer als das Listenende ist, resultiert ein leeres Ergebnis.

- Wenn der Endindex hinter dem Listenende liegt, wird automatisch am letzten Element der Liste aufgehört.
- Beim Schrittindex 0 würde ein Fehler auftreten, da ein Schritt von 0 nicht möglich ist.

2.2.2 Dictionary

Ein Dictionary (`dict`) speichert **Schlüssel-Wert-Paare** in einer ungeordneten Form. Darauf kann sehr schnell (in der Regel in O(1)) zugegriffen werden.

```
noten = {
    "Alice": 1,
    "Bob": 2
}
print(noten["Alice"])  # 1
print(noten.items)   # dict_items([('Alice', 1), ('Bob', 2)])
```

- Hinzufügen neuer Schlüssel: `noten["Charlie"] = 3`
- Iteration:

```
for key, value in noten.items():
    print(key, value)# Alice 1; Bob 2; Charlie 3
```

2.2.3 Sets

Ein Set (`set`) ist eine **ungeordnete** Sammlung **einzigartiger** Elemente:

```
elemente = {"A", "B", "C"}
elemente.add("D")     # {"A", "B", "C", "D"}
elemente.discard("B") # {"A", "C", "D"}
```

Wichtige Mengenoperationen: `union()`, `intersection()`, `difference()`. Sets eignen sich, wenn Duplikate ausgeschlossen werden sollen oder große Mengenoperationen effizient durchzuführen sind.

2.2.4 Tupel

Tupel (`tuple`) sind **unveränderliche** Sequenzen:

```
koordinaten = (10, 20)
x, y = koordinaten  # Entpackung, x=10, y=20
```

Tupel finden Anwendung, wenn eine feste Struktur benötigt wird (z. B. zwei Koordinatenwerte, die sich nicht ändern sollen).

2.2.5 Stacks und Queues

Obwohl Python die abstrakten Datenstrukturen Stack und Queue nicht als separate Klassen anbietet, lassen sich Stacks und Queues mithilfe von Listen oder `collections.deque` realisieren:

`Stack (LIFO)`: `append()` zum Ablegen, `pop()` zum Entfernen.

`Queue (FIFO)`: Bei einer Liste müsste `pop(0)` verwendet werden, was ineffizient sein kann. Stattdessen bietet sich `deque` an:

```
from collections import deque

warteschlange = deque()
warteschlange.append("Erstes")
warteschlange.append("Zweites")
erst_entfernt = warteschlange.popleft()  # 'Erstes'
```

2.3 Kontrollstrukturen

2.3.1 Verzweigungen (if-elif-else)

```
wert = 10
if wert < 0:
    print("Negativ")
elif wert == 0:
    print("Null")
else:
    print("Positiv")    # 'Positiv'
```

2.3.2 Schleifen

for-Schleife

```
for i in range(3):
    print("Index:", i)
# index 0
# index 1
# index 2
```

while-Schleife:

```
counter = 0
while counter < 5:
    print("Counter:", counter)
    counter += 1
# Counter 0
# Counter 1
# Counter 2
```

2.3.3 Enumaration

Iterierbare Objekte (Strings, Listen, Sets, …) können mittels der `enumerate` Funktion iteriert werden. Bei Dictionaries wird entweder über die keys oder explizit über `items()` iteriert.

Unter Verwendung von `enumerate` kann gleichzeitig auf den Index und den konkreten Wert zugegriffen werden. Der Startindex kann mit `start=` festgelegt werden, ansonsten ist dieser 0. Wird ein Durchlaufs mittels `continue` übersprungen bleibt der Zähler konsistent.

```
werte = [10, 20, 30]


# Index + Wert (Start 0)
for i, v in enumerate(werte):
    print(f"Index {i}: Wert {v}")
# Index 0: Wert 10
# Index 1: Wert 20
# Index 2: Wert 30

# Index bei 1 beginnen lassen
for pos, v in enumerate(werte, start=1):
    print(f"{pos}. Element = {v}")
# 1. Element = 10
# 2. Element = 20
# 3. Element = 30


text = "ABC"
for i, ch in enumerate(text):
    print(i, ch)
# 0 A
# 1 B
# 2 C


d = {"a": 1, "b": 2}
for i, (k, v) in enumerate(d.items(), start=1):
    print(i, k, v)
# 1 a 1
# 2 b 2
```

3 Exception Handling

3.1 Try-Catch

Bei potentiell fehlerträchtigem Code kann mittels `try-except` ein kontrollierter Umgang mit Ausnahmen erreicht werden:

```
try:
    zahl = int(input("Bitte Zahl eingeben: "))
    ergebnis = 10 / zahl
    print(ergebnis)
except ValueError:
    print("Fehler: Eingabe war keine gültige Zahl!")
except ZeroDivisionError:
    print("Fehler: Division durch Null!")
finally:
    print("Ende der Verarbeitung, ggf. Aufräumoperationen.")
```

4 Funktionen

4.1 Normale Funktionen

Eine Funktion bietet die Möglichkeit ein Teilprogramm auszulagern und ggf. um Parameter zu erweitern. Sie kann lediglich ein Programm ausführen, aber auch einen Rückgabewert ausgeben. Sie werden über das Schlüsselwort `def` definiert.

```
def functionname(param1,…):
    body

functionname(param1,…) #Aufruf der Funktion
```

4.2 Anonyme Funktionen

Eine anonyme Funktion erfüllt den gleichen Zweck wie eine herkömmliche Funktion, nur das diese nicht explizit via `def` definiert wird, sondern wie eine Variable mit modularem Input behandelt werden kann. Die Definition erfolgt über das Schlüsselwort `lambda`, das eine solche Funktion charakterisiert. Die eigentliche Funktion nimmt einen oder mehrere Werte entgegen und verarbeitet diese auf eine bestimmte Art und Weise. Die Trennung zwischen Ein- und Ausgabe erfolgt über „`:`“. Verschachtelungen sind ebenfalls möglich. So sind folgende Inhalte äquivalent:

```
#Einfach Funktion
def regfunc(val1, val2):
    return val1 + val2
anfunc = lambda val1, val2 : val1 + val2
print(regfunc(1,2), anfunc(1,2)) # -> 3
```

```
#Verschachtelung
def funcnest(val1):
    def inner(val2):
        return val1 + val2
    return inner
#funcnest(1) gibt zurück --> inner(val2): return 1 + val2
anfuncnest = lambda val1 : (lambda val2 : val1 + val2)
print(funcnest(1)(2), anfuncnest(1)(2)) # -> 3
```

5 Algorithmen

5.1 Iteration vs. Rekursion

Iteration: Wiederholung eines Codeblocks (for- oder while-Schleife).

Rekursion: Aufruf einer Funktion innerhalb ihrer selbst.

Beispiel: **Fibonacci-Folge**

Iterativ:

```
def fib_iter(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n+1):
        a, b = b, a + b
    return b
```

Rekursiv:

```
def fib_rek(n):
    if n <= 1:
        return n
    return fib_rek(n-1) + fib_rek(n-2)
```

5.2 Suchalgorithmen

5.2.1 Lineare Suche

Durchläuft die Liste von Anfang bis Ende, vergleicht jedes Element mit dem gesuchten Wert. **O(n)** im Worst Case.

```
def linear_search(liste, ziel):
    for i in range(len(liste)):
        if liste[i] == ziel:
            return i
    return -1
```

Randfälle:

- Leere Liste: Schleife wird nicht ausgeführt, -1 wird zurückgegeben.
- Mehrere Vorkommen: Liefert den Index des ersten Vorkommens.
- Gesuchter Wert nicht vorhanden: Gibt -1 zurück.

5.2.2 Binäre Suche

Nutzt eine **sortierte** Liste. Teilt die Liste in zwei Hälften. Vergleicht den mittleren Wert mit dem gesuchten Wert. Falls das Element kleiner ist, wird nur die linke Hälfte betrachtet, andernfalls die rechte. **O(log n)**.

```
def binary_search(liste, ziel):
    links = 0
    rechts = len(liste) - 1
    while links <= rechts:
        mid = (links + rechts) // 2
        if liste[mid] == ziel:
            return mid
        elif liste[mid] < ziel:
            links = mid + 1
        else:
            rechts = mid - 1
    return -1
```

Randfälle:

- Leere Liste (length=0): links=0, rechts=-1 → Schleife läuft nicht, -1 zurück.
- Gesuchter Wert kleiner als alle Elemente oder größer als alle Elemente → Abbruch.
- Gesuchter Wert mehrfach vorhanden: Die obige Implementierung findet irgendein Vorkommen, jedoch nicht zwingend das erste.

5.3 Sortieralgorithmen

Sortieralgorithmen ordnen die Elemente einer Liste, z. B. in aufsteigender Reihenfolge. Jeder Algorithmus besitzt verschiedene Komplexitäts- und Stabilitätseigenschaften.

5.3.1 Bubble Sort

Vergleicht benachbarte Elemente paarweise und vertauscht sie, falls die Reihenfolge nicht stimmt.
Jeder Durchlauf platziert das jeweils größte Element ans Ende.
Zeitkomplexität: O(n²) im Worst Case.
Vorteil: Sehr leicht zu verstehen, aber relativ langsam für große Datenmengen.

```
def bubble_sort(liste):
    n = len(liste)
    for i in range(n):
        for j in range(0, n - i - 1):
            if liste[j] > liste[j + 1]:
                liste[j], liste[j + 1] = liste[j + 1], liste[j]
```

5.3.2 Selection Sort

Findet in jedem Durchlauf das **kleinste** Element im unsortierten Teil und tauscht es mit dem ersten Element dieses unsortierten Teilbereichs.
Zeitkomplexität: O(n²).

```
def selection_sort(liste):
    n = len(liste)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if liste[j] < liste[min_index]:
                min_index = j
        liste[i], liste[min_index] = liste[min_index], liste[i]
```

5.3.3 Insertion Sort

Geht Element für Element durch die Liste und baut sukzessive eine sortierte Teilliste auf. Jeder Wert wird in die passende Position innerhalb dieses bereits sortierten Bereichs geschoben. **Zeitkomplexität**: O(n²) im schlechtesten Fall, aber effizient, wenn die Liste schon fast sortiert ist

```
def insertion_sort(liste):
    for i in range(1, len(liste)):
        key = liste[i]
        j = i - 1
        while j >= 0 and liste[j] > key:
            liste[j + 1] = liste[j]
            j -= 1
        liste[j + 1] = key
```

5.3.4 Mergesort

1. **Teilen**: Die Liste wird in zwei Hälften (left_half, right_half) gesplittet.
2. **Rekursives Sortieren**: Jede Hälfte wird solange weiter aufgespalten, bis nur noch Teilsequenzen mit ≤ 1 Element vorliegen (diese gelten als sortiert).
3. **Zusammenführen**: In einer Merge-Funktion werden die Teillisten wieder **sortiert zusammengefügt**.

Detailbetrachtung:

Teilfunktion: `mergesort(liste)`

Falls `len(liste) > 1`:

Bestimmung des Mittelpunktes: `mid = len(liste) // 2`

Aufspaltung in `left_half = liste[:mid]` und `right_half = liste[mid:]`

Rekursiver Aufruf:

```
mergesort(left_half)
mergesort(right_half)
```

Merge-Prozess (Zusammenführen der beiden sortierten Hälften):

Drei Zeiger: `i` (für `left_half`), `j` (für `right_half`), `k` (für das Wieder-Einfügen in `liste`).

Während sowohl `i < len(left_half)` als auch `j < len(right_half)`:

Vergleich `left_half[i]` und `right_half[j]`: Das kleinere Element wird an `liste[k]` geschrieben. Der entsprechende Zeiger wird erhöht.

Falls Elemente übrig bleiben (entweder links oder rechts), werden diese direkt ans Ende kopiert.

Der gesamte Code gliedert sich wie folgt auf:

```
def mergesort(liste):
    # 1) Abbruchbedingung
    if len(liste) > 1:
        # 2) Liste teilen
        mid = len(liste) // 2
        left_half = liste[:mid]
        right_half = liste[mid:]

        # 3) Rekursiv sortieren
        mergesort(left_half)
        mergesort(right_half)

        # 4) Zusammenführen
        i = j = k = 0

        # 4a) Solange in beiden Listen noch Elemente vorhanden sind
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                liste[k] = left_half[i]
                i += 1
            else:
                liste[k] = right_half[j]
                j += 1
            k += 1

        # 4b) Restliche Elemente der linken Hälfte
        while i < len(left_half):
            liste[k] = left_half[i]
            i += 1
            k += 1

        # 4c) Restliche Elemente der rechten Hälfte
        while j < len(right_half):
            liste[k] = right_half[j]
            j += 1
    k += 1
```

Komplexität:

- Best Case und Average Case: O(n log n).
- Worst Case: Ebenfalls O(n log n).

Des Weiteren ist Mergesort **stabil**, d.h. die relative Reihenfolge gleicher Elemente bleibt erhalten.

Randfälle:

- Leere Liste oder Liste mit nur einem Element: Wird als bereits sortiert betrachtet, die Rekursion endet.
- Ungerade Listenlänge: Eine Teilliste ist um ein Element größer, was das Zusammenführen jedoch nicht beeinträchtigt.

5.3.5 Quicksort

1. Wählt ein **Pivot**-Element.
2. Teilt die Liste in drei Teile: kleiner als `Pivot(left)`, gleich dem `Pivot(middle)` und größer als `Pivot(right)`
3. Sortiert rekursiv `left` und `right`
4. Fügt die Ergebnisse zusammen.

```
def quicksort(list):
  if len(list) <= 1:
       return list
  pivot = list[0]
  left  = []
  right  = []
  for i in range(1, len(list)):
       if list[i] <= pivot:
            left.append(list[i])
       else:
            right.append(list[i])
  return quicksort(left) + [pivot] + quicksort(right)
```

Komplexität

- **Durchschnittlich** O(n log n)
- **Worst Case** O(n²)
- wenn immer ein extremes Pivot gewählt wird.

6. OBJEKTORIENTIERTE PROGRAMMIERUNG (OOP) IN PYTHON

6.1 Klassen und Objekte

Eine Klasse dient als **Bauplan**, ein Objekt ist eine **Instanz** dieser Klasse:

```
class Fahrzeug:
    def __init__(self, farbe, marke):
        self.farbe = farbe
        self.marke = marke

auto1 = Fahrzeug("Rot", "BMW")
```

6.2 Methoden und Konstruktoren

Methoden sind innerhalb der Klasse definierte Funktionen. Der Konstruktor `__init__` wird aufgerufen, sobald ein Objekt aus dieser Klasse erzeugt wird.

```
class Fahrzeug:
    def __init__(self, farbe, marke):
        self.farbe = farbe
        self.marke = marke

    def beschreibung(self):
        return f"Dieses Fahrzeug ist ein {self.farbe}er
{self.marke}."
```

6.3 Punktnotation und Zugriffsarten

Der Aufruf von Methoden oder der Zugriff auf Attribute eines Objekts erfolgt mit **Punktnotation**:

```
auto2 = Fahrzeug("Blau", "Audi")
print(auto2.beschreibung())
auto2.farbe = "Gelb"  # Attributzugriff
```

7. DOKUMENTATION KRYPTO-TOOLS

Der folgende Abschnitt beschreibt ein in Python implementiertes **Krypto-Tool**, das verschiedene Verfahren der **klassischen Kryptographie** sowie deren Analyse bereitstellt. Hierbei kommen die zuvor erläuterten Python-Grundlagen und -Konzepte zur Anwendung. Es wird zunächst die Struktur des Tools beschrieben, um anschließend jede Funktionalität im Detail darzustellen. Abschließend erfolgt eine Zusammenführung, in der verdeutlicht wird, wie die einzelnen Bestandteile zusammenwirken.

7.1 Allgemeine Funktionsweise

Das Krypto-Tool verarbeitet einen Text – entweder Klartext oder chiffrierten Text – und bietet folgende Hauptfunktionen:

1. **Cäsar-Verschlüsselung** (inklusive Entschlüsselung und Bruteforce)
2. **Vigenère-Verschlüsselung** (inklusive Entschlüsselung)
3. **Häufigkeitsanalyse** (absolute und relative Häufigkeiten)
4. **n-Gramme** (Bigramme, Trigramme etc.)
5. **Spracheinschätzung** über Koinzidenzindex
6. **Multi-Replacement** (Zeichenersetzungen)

7.2 Grundlagen der Kryptographie im Tool

7.2.1 Cäsar-Verschlüsselung

Ein einfaches, historisches Verfahren, bei dem jeder Buchstabe um eine feste Anzahl (Key) im Alphabet verschoben wird:

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"


def caesar(klartext, key):
    klartext = klartext.upper()
    chiffre = ""
    for buchstabe in klartext:
        index = alphabet.index(buchstabe)
        zielIndex = (index + key) % 26
        chiffre += alphabet[zielIndex]
    return chiffre
```

Randfälle:

- Umlaute oder Sonderzeichen sind hier nicht abgedeckt.

Das **Entschlüsseln** erfolgt durch die inversen Verschiebung:

```
def de_caesar(chiffre, key):
    return caesar(chiffre, 26 - key)
```

Eine **Bruteforce-Funktion** testet alle Schlüssel von 0 bis 25:

```
def caesar_bruteforce(chiffre):
    for i in range(26):
        potKlartext = de_caesar(chiffre, i)
        print(f"Key {i}: {potKlartext}")
```

7.2.2 Vigenère-Verschlüsselung

Die Vigenère-Chiffre nutzt ein **Passwort**, um verschiedene Verschiebungen durchzuführen. Jeder Buchstabe des Passworts (A=0, B=1, …) bestimmt die Verschiebung des entsprechenden Klartext-Buchstabens:

```
def vigenere(klartext, passwort):
    klartext = klartext.upper()
    passwort = passwort.upper()
    result = ""
    for i, buchstabe in enumerate(klartext):
        keyInt = ord(passwort[i % len(passwort)]) - ord('A')
        result += caesar(buchstabe, keyInt)
    return result
```

Entschlüsselung durch die entsprechende Gegenverschiebung:

```
def de_vigenere(chiffre, passwort):
    chiffre = chiffre.upper()
    passwort = passwort.upper()
    result = ""
    for i, buchstabe in enumerate(chiffre):
        keyInt = ord(passwort[i % len(passwort)]) - ord('A')
        result += de_caesar(buchstabe, keyInt)
    return result
```

7.3 Häufigkeitsanalyse und Dictionary-Einsatz

Die Häufigkeitsanalyse bestimmt, wie oft bestimmte Zeichen innerhalb eines Textes vorkommen. Bei historischen monoalphabetischen Chiffren liefert dies wertvolle Informationen:

```
def absolute_haeufigkeiten(text):
    freq = {}
    for char in text:
        if char in freq:
            freq[char] += 1
        else:
            freq[char] = 1
    return freq


def relative_haeufigkeiten(text):
    freq_abs = absolute_haeufigkeiten(text)
    laenge = len(text)
    freq_rel = {}
    for key in freq_abs:
        freq_rel[key] = freq_abs[key] / laenge * 100
    return freq_rel
```

Darüber hinaus berechnet das Tool **n-Gramme** (z. B. Bigramme, Trigramme). Ein mögliches Vorgehen:

```
def ngram(text, n):
    ntuplesTotal = 0
    ngrams = dict()
    for delta in range(n):
        for i in range(0, len(text), n):
            teil = text[delta+i : delta+i+n]
            if len(teil) == n:
                ntuplesTotal += 1
                if teil in ngrams:
                    ngrams[teil] += 1
                else:
                    ngrams[teil] = 1
    for key in ngrams:
        ngrams[key] = ngrams[key] / ntuplesTotal * 100
    return ngrams
```

7.4 Multi-Replacement (Zeichenersetzungen)

Ein wesentlicher Bestandteil des Tools ermöglicht, gezielt einzelne Zeichen oder Zeichenketten in einem Text auszutauschen. Hierbei werden zwei Listen verwaltet:

`remove`: Liste der zu ersetzenden Zeichen (oder Strings)

`new`: Liste der Ersatzzeichen

```
def multirep(text, remove, new):
    """
    Ersetzt remove[i] durch new[i] im Text.
    """
    result = text
    for i in range(len(remove)):
        result = result.replace(remove[i], new[i])
    return result
```

7.5 Zusammenführung der Bausteine

Im Tool selbst wird ein **Interaktionsmenü** bereitgestellt, das alle beschriebenen Funktionen orchestriert. Zusammengefasst ist der Ablauf folgendermaßen zu sehen:

1. **Eingabe des Textes** (Klar- oder Chiffretext)
2. Auswahl einer Verschlüsselung oder Entschlüsselung (Cäsar, Vigenère)
3. Optional: Ausführung der **Häufigkeitsanalyse** oder n-Gramm-Bestimmung
4. Ggf. **Multi-Replacement**, z. B. um Substitutionstabellen zu testen
5. Anzeige des Ergebnisses

7.6 Praxisbeispiel: Analyse einer unbekannten symmetrischen (monoalphabetischen) Verschlüsselung mit dem Krypto-Tool

In diesem Beispiel wird ein unbekannt verschlüsselter, deutschsprachiger Text vorliegen. Das Ziel besteht darin, mithilfe des Krypto-Tools sämtliche Analyse- und Ersetzungsfunktionen so einzusetzen, dass am Ende der Klartext wiederhergestellt wird. Diese Verschlüsselung ist **weder** eine einfache Cäsar- noch eine Vigenère-Chiffre, sondern eine **monoalphabetische Substitution** (jedes Zeichen A-Z wird eindeutig durch ein anderes Zeichen ersetzt).

7.6.1 Ausgangssituation: Chiffretext

Klartext (original, zur Illustration hier unverschlüsselt gezeigt, weiter unten dann verschlüsselt):

„DIESES PROGRAMM DEMONSTRIERT WIE EINE MONOALPHABETISCHE SUBSTITUTION IN PYTHON GEBROCHEN WERDEN KANN."

Der obige Satz wurde zufällig verändert, um Umlaute oder sehr seltene Zeichen zu vermeiden, und könnte typisch für einen deutschsprachigen Textausschnitt sein. Es kommen Wörter wie „DIESES", „PROGRAMM", „SUBSTITUTION" usw. vor, die sich im Deutschen durch charakteristische Buchstaben (häufiges „E", typische Bigramme wie „CH") charakterisieren.

Angenommen, der Text wird nach einer beliebigen Substitution verschlüsselt. Das Ergebnis, der **Chiffretext**, lautet:

FKLULU PBAGBCXX FLXAQUZBKLBZ WKL LKQL XAQACLPJCBLZKUCJL UUBUZKZUZKAQ KQ PYZJAQ GLBBACJLQ WLBFLQ KCQQ

7.6.2 Einbringen in das Krypto-Tool

Zunächst übernimmt das Tool diesen Text als `chiffre`:

```
chiffre = " FKLULU PBAGBCXX FLXAQUZBKLBZ WKL LKQL XAQACLPJCBLZKUCJL
UUBUZKZUZKAQ KQ PYZJAQ GLBBACJLQ WLBFLQ KCQQ"
```

7.6.3 Schritt 1: Häufigkeitsanalyse auf dem Chiffretext

Der erste Schritt besteht in der **absoluten** und **relativen** Häufigkeitsanalyse:

```
freq_abs = absoluteHaeufigkeiten(chiffre)
freq_rel = relativeHaeufigkeiten(chiffre)
```

Für die relative Häufigkeitsverteilung ergibt sich:

```
Relative Haeufigkeiten:
{0: ('L', 14.0), 1: ('Q', 10.0), 2: ('K', 9.0), 3: ('B', 9.0),
4: ('U', 8.0), 5: ('A', 7.1), […]}
```

Interpretation:
Das Zeichen `L` sticht hier mit 14.0 % hervor. Auch `Q,K,B` erscheinen recht häufig. Deutsch zeichnet sich durch relativ häufige Buchstaben wie `E`, `N`, `I`, `S`, `R`, `A`, `T` aus. Da es sich um ein unbekanntes Substitutionsverfahren handelt, kann vermutet werden, dass z. B. `L` eines der häufigen Klartextzeichen sein könnte (etwa `E`, `N` oder `I`).
Optional lässt sich mit `sprache(chiffre)` der **Koinzidenzindex** bestimmen, um zu sehen, ob sich die Häufigkeiten eher dem Deutschen oder einer anderen Sprache zuordnen lassen. Wenn der Index bei ~0.076 liegt, bestärkt das die Deutsch-Vermutung. Damit diese Methode effizient funktioniert, muss die Chiffre hinreichend lang sein, da das Ergebnis aufgrund von zu wenig Eingabeinformationen ggf. verfälscht wird.

7.6.4 Schritt 2: n-Gramm-Analyse

Um genauere Anhaltspunkte für Buchstabenpaare (Bigramme) und -dreier (Trigramme) zu erhalten, ruft das Tool die `ngram`-Funktion auf:

```
bigrams = ngram(chiffre, 2)
```

Der Aufruf führt zu:

```
Bigramme:
{0: ('AQ', 4.040404040404041), 1: ('KL', 3.0303030303030303),
2: ('ZK', 3.0303030303030303), […]}
```

Im Deutschen finden sich oft Doppelbuchstaben wie „ll“, „ss“, „ee“. Das könnte ein starker Hinweis sein, dass bei einfacher Substitution in der chiffre doppelt auftretende Buchstaben im Klartext ebenfalls ein häufiger Doppelbuchstabe (etwa „SS“ oder „LL“) sein könnte.

7.6.5 Schritt 3: Erstellen einer Multi-Replacement-Ersetzungstabelle

Monoalphabetische Substitutionen kann gebrochen werden, indem hypothesenweise Buchstaben austauscht und beobachtet werden bzw. ob sich sinnvolle Fragmente ergeben. Das Krypto-Tool bietet hierfür die `remove`- und `new`-Listen sowie die Funktion `multirep(...)`.

Erste Hypothese:

- Da `L` extrem häufig ist, könnte `L` → `E` sein (typischerweise der häufigste Buchstabe im Deutschen).
- Da `Q` ebenfalls sehr oft vorkommt, könnte `Q` → `N` sein.
- Dass K oft vorkommt, könnte auf `I`, `S`, `R`, `T` (alles häufige Buchstaben) hindeuten.

Diese Vermutungen können wie folgt umgesetzt werden:

```
remove = ["L", "Q", "K"]
new    = ["E", "N", "I"]
test_1 = multirep(chiffre, remove, new)
print(test_1)
```

An allen Positionen innerhalb der Chiffre an denen vorher L, Q und K standen, stehen nach dem Aufruf entsprechend E, N und I. Während dieser Prozedur entstehende Teilergebnisse können auf gleiche Art und Weise ebenfalls wieder Analysiert werden, und auf die Bildung von Wortfragmente (z.B. Artikel: Der, Die, Das) untersuchen.

7.6.6 Schritt 4: Iteratives Vorgehen mit Korrekturen

Ergibt die erste Hypothese nichts Lesbares, **wird sie korrigiert**.

Beispielhafter Ablauf:

`L` → `E` (beibehalten)

`Q` → `N` (beibehalten)

`K` → `I` (Annäherung: `Z` könnte „S“ sein)

`B` → `R` (Hypothese: Häufiger Konsonant)

`Z` → `T` (Hypothese: Ebenfalls ein recht häufig vorkommender Buchstabe)

[…]

Nach jedem Schritt kann entweder manuell das Zwischenergebnis untersucht werden oder erneut Bigramme abgerufen werden (`ngram(..., 2)` auf dem teilersetzten Text). Fällt auf, dass sich z. B. ein hohes Aufkommen von `ST` bildet, könnte es sich dabei um ein deutsches Bigramm wie „st“ handeln.

Ergeben sich Widersprüche (z. B. sehr viele unsinnige Folgen), werden entsprechende Anpassungen vorgenommen.

Auf diese Weise füllt sich nach und nach die Ersetzungstabelle mit ~26 Einträgen (insofern jeder Buchstabe des Alphabets irgendwo auftritt).

7.6.7 Schritt 5: Erfolgskontrolle und finaler Klartext

Sobald genügend richtige Zuordnungen existieren, beginnt der Text lesbar zu werden. In diesem konkreten Beispiel führt das Tool durch fortgesetztes Raten und Bestätigen letztlich zu:

Finale Ersetzungstabelle (fiktives Mapping):

{l → e, q → n, s → z, k → l, a → o, z → t, u → s, b → r, c → a, x → m, j → h, f → d}

Werden diese Ersetzungsfolgen angewendet, resultiert das in dem Klartext:

```
„DIESES PROGRAMM DEMONSTRIERT, WIE EINE MONOALPHABETISCHE
SUBSTITUTION IN PYTHON GEBROCHEN WERDEN KANN.“
```

Dieser Klartext entspricht dem ursprünglichen, was den Erfolg der Analyse belegt.

8. DOKUMENTATION WordExtraktTool

Der folgende Abschnitt beschreibt ein in Python implementiertes **Web-Tool**, das den HTML-Code einer Webseite herunterlädt und von diesem die Wörter zählt. Das Programm bietet eine Konsolen-Schnittstelle, in der eine beliebige URL eingefügt werden kann.
Hierzu wird zunächst eine detailliertere Einführung zu dem Thema Bibliotheken gegeben, sowie die Grundlagen zum Thema Web, die für die Funktion des Programms benötigt werden.

8.1 Bibliotheken

In Python können fertige Code-Pakete in Form einer Bibliothek freigegeben und Projekt- bzw. Dateiübergreifend verwendet werden. Dazu stellt Python eine Standardbibliothek bereit, welche eine Vielzahl nutzbarer Module für Dateisystem, Textverarbeitung, Zahlen, Zeit, Datenformate, Algorithmen, etc bereitstellt. Des Weiteren lassen sich aber auch Drittanbieter-Module oder eigen geschrieben Module auf diese Art und Weise in den Code einbinden. Der Import erfolgt über `import modul` meist zu Beginn eines Programms bzw. über spezifische Importe wie `from modul import Name`. Durch eine solche Bibliothek bereit gestellte Klassen und Methoden können mit `dir(module)` ausgegeben werden.

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', 'UTC', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'date', 'datetime', 'datetime_CAPI',
'time', 'timedelta', 'timezone', 'tzinfo']
```

8.2 URL und Request Bibliothek

Für die Extrahierung des HTML Codes wird die request-Bibliothek in Python verwendet. Über dieses Modul können HTTP-Anfragen an einen Web-Server gestellt werden.

`get(url, params, args)`	Sends a GET request to the specified url
`head(url, args)`	Sends a HEAD request to the specified url
`post(url, data, json, args)`	Sends a POST request to the specified url

Mit Folgendem Code kann also eine Request an einen Webserver gestellt werden, und der Eigentliche Inhalt bzw. HTML-Code mit `.content.decode()` (gibt den Content

in Bytes zurück und dekodiert diesen) bzw. `.text` (gibt den Content in Unicode zurück) extrahiert werden.

```
import requests

PAGE_URL = 'http://target:port'

resp = requests.get(PAGE_URL)
html_str = resp.content.decode()
print(html_str)
```

8.3 Einbinden von Statusmeldungen

Das Programm soll allerdings nur auf verfügbaren bzw. existierenden Webseiten agieren. Hierzu wird zunächst der Status der Anfrage überprüft und nur fortgefahren, falls dieser 200-OK ist, andernfalls wird der tatsächliche Statuscode ausgegeben und das Programm beendet. Zudem wird die gesamte Logik in eine Funktion ausgelagert, um innerhalb des Programms eine saubere Trennung zu schaffen.

```
import requests

PAGE_URL = 'http://target:port'

def get_html_of(url):
    resp = requests.get(url)
    if resp.status_code != 200:
         print(f'HTTP status code of {resp.status_code} returned,
but 200 was expected. Exiting...')
        exit(1)

    return resp.content.decode()

print(get_html_of(PAGE_URL))
```

8.4 Text aus HTML-Code extrahieren

Als nächstes muss der wirkliche Text extrahiert werden und HTML-Tags und andere Metadaten verworfen werden. Hierzu eignet sich die Bibliothek `BeautifulSoup`. Mit der Funktion `get_text()` kann aus einem HTML Code der reine Text extrahiert werden. Um die einzelnen Wörter zu zählen bzw. in dem Text zu trennen kann die Bibliothek Regex (Regular Expression) verwendet werden. Reguläre Ausdrücke sind ein Teilgebiet der theoretischen Informatik und definieren eine Formale Sprache. D.h. ein Regulärer Ausdruck ist eine Art Regelwerk für die Anordnung und Häufigkeit von Buchstaben zum Beschreiben einer Sprache, also einem Wortschatz der bestimmten

Regeln unterliegt. Für diesen Anwendungsfall wird die Methode `re.findall(pattern, string)` mit dem Pattern bzw. Ausdruck `r'\w+'` verwendet. `r''` ist ein raw String, bedeutet der String wird so gelesen, wie übergeben. Ansonsten würde „\“ einen Fehler verursachen. `\w` beschreibt jede mögliche zusammenhängende Kombination aus den Zeichen `A-Z, a-z, 0-9, _` bzw. jedes einfache Wort und jede einfache Zahl ohne Sonderzeichen. Das `+` gibt an, dass eine solche Zeichen- bzw. Zahlen-Kette nicht leer ist. Die Methode `findall()` gibt eine Liste mit allen Strings zurück, die der Bedingung entsprechen. Hier bedeutet das, dass der gesamte Text in eine Liste aus einzelnen Wörtern umgewandelt werden kann, die dann für die Iteration bzw. den eigentlichen Zählvorgang der Wörter verwendet werden kann. Diese Logik wird im Programm wie folgt zusammengesetzt:

```
import re
from bs4 import BeautifulSoup

def get_text_of_html(html):
    soup = BeautifulSoup(html, 'html.parser')
    return soup.get_text()

def list_of_words(text):
    return re.findall(r"\w+", text)
```

8.5 Anzahl einzelner Wörter ermitteln und verwalten

Bis zu diesem Punkt wird der Text aus dem HTML-Code extrahiert und jedes Wort in einen einzelnen Listeneintrag geschrieben. Als nächstes muss für jedes Wort die Häufigkeit ermittelt werden. Dazu wird ein Dictionary verwendet, bei dem das konkrete Wort als Schlüssel und die Anzahl als Wert fungiert. Anschließend wird die Liste iteriert und das Dictionary gefüllt.

```
def dict_word_count(wordlist):
    worddict = {}
    for word in wordlist:
        if word not in worddict:
            worddict[word] = 1
        else:
            count = worddict.get(word)
            worddict[word] = count + 1
    return worddict
```

8.6 Einträge sortieren – Lamda Funktionen

Im letzten Schritt müssen die Einträge auf Basis der Wertigkeit sortiert werden. Dazu wird die von Python bereitgestellte Funktion `sorted(iterable, key=FUNC, reverse=False)` in Kombination mit einer anonymen Funktion verwendet. `sorted()` nimmt eine iterierbare Struktur sowie ein Sortierkriterium entgegen und gibt eine sortierte Liste zurück. In diesem Fall muss ein Dictionary auf Basis der Wertigkeit der Einträge sortiert werden. Die Fertige Liste speichert Tupel mit `(key, value)` Paaren, die entsprechend nach `value` absteigend sortiert werden soll. Dazu eignet sich folgender Aufruf:

```
sorted(worddict.items(), key=lambda item: item[1], reverse=True)
```

Für jeden Eintrag wird genau einmal die key-Funktion aufgerufen und die Elemente auf Basis ihres erstens Eintrags (hier: value) sortiert. Die resultierende Liste kann dann für weitere Zwecke z.B. die Ausgabe der n häufigsten Elemente verwendet werden. Dadurch entsteht das folgende finale Programm:

6.7 Finales Programm

```
import re
import requests
from bs4 import BeautifulSoup

PAGE_URL = 'https://google.com'

def get_html_of(url):
    resp = requests.get(url)
    if resp.status_code != 200:
         print(f'HTTP status code of {resp.status_code} returned,
but 200 was expected. Exiting...')
        exit(1)
    return resp.text

def get_text_of_html(html):
    soup = BeautifulSoup(html, 'html.parser')
    return soup.get_text()

def list_of_words(text):
    return re.findall(r"\w+", text)

def dict_word_count(wordlist):
    worddict = {}
    for word in wordlist:
        if word not in worddict:
            worddict[word] = 1
        else:
            count = worddict.get(word)
            worddict[word] = count + 1
    return worddict

def get_sorted_tupel_list(worddict):
      return  sorted(worddict.items(),  key=lambda  item:  item[1],
reverse=True)

def count_words():
    html = get_html_of(PAGE_URL)
    text = get_text_of_html(html)
    wordlist = list_of_words(text)
    worddict = dict_word_count(wordlist)
    sortedfinallist = get_sorted_tupel_list(worddict)
    for index, word in enumerate(sortedfinallist):
        if index >= min(5, len(sortedfinallist)):
            break
        print(word[0], "->", word[1])

count_words()
```