Sichere Programmierung
Projekt 1
Einführung in Python

Nils Klein
(79373)

Alexander Krause
(79878)

Abdullah Yildiz
(79669)

Dozentin: Corina Hampel

Inhaltsverzeichnis

1 Teil 1: Mechanismus der affinen Chiffre

1.1 Aufgabe 1 - decode(text)

In der ersten Aufgabe wird eine Konvertierung der User Eingabe durchgeführt.
Der einzugebende String kann aus einem oder mehreren Chars bestehen. Während Zahlen und Sonderzeichen nicht beachtet werden, wird der Textinput (sofern korrekt eingegeben) durch die Funktion ”decode” in Zahlen konvertiert und als Liste mit einem return zurückgegeben. Die auszugebenden Zahlen spielen hierbei eine wichtige Rolle. Das Alphabet soll mit den Zahlen 0-25 dargestellt werden, das mittels einer Funktion in Python (”dict”) interpretiert werden kann.

1.2 Aufgabe 2 - encode(char_list)

In der zweiten Aufgabe soll die Eingabe nun in Zahlenform erfolgen, interpretiert und als String übersetzt werden.
Wird z.B. die Zahlenfolge [7, 0, 11, 11, 14, 22, 4, 11, 19] eingegeben, wird für jede Zahl die ganze Liste iteriert und mit dem Wörterbuch (”dict”) überprüft, ob der Buchstabe interpretiert werden kann. Falls ein zugehöriger Wert gefunden wurde, wird dieser in eine neue Liste **”encoded_input”** gespeichert.
Sobald alle Werte einmal überprüft wurden, wird die Liste mit einem return zurückgegeben.

```
return "".join(encoded_input)
```

1.3 Aufgabe 3 - key_table

In dieser Aufgabe soll ein Wörterbuch **key_table** für alle erlaubten Werte des Teilschlüssels **”a”** mit dem zugehörigen inversen Element ertellt werden.
Da die Werte bereits angegeben sind, wurde zu jedem Wert von **”a”** das inverse Element hinzugefügt:

```
key_table = {

    1: 1,
    3: 9
    5: 21,
    ..
    ..
}
```

1.4 Aufgabe 4 - acEncrypt(a, b, plain_text)

Der String **plain_text** wird in eine neue Liste geschrieben, mit der Funktion **decode** in Zahlen konvertiert und in der Variable **orig_input** gespeichert.
Mit Hilfe der Funktion **broken_key** wird geprüft, ob der Schlüssel **(a, b)** fehlerhaft oder gültig ist. Die Funktion prüft, ob der Wert des Schlüssels **a** nicht in der "Key_table" oder der von **b** außerhalb des Wertebereichs 0 bis 25 liegt. Bei "True" ist der Schlüssel fehlerhaft, und **acEncrypt** gibt einen leeren String zurück.
Als nächstes wird in einer for-Schleife jedes Element von **orig_input** als Integer Wert gecastet.

Die eigentliche Verschlüsselung erfolgt durch die folgende Formel:

```
1    return ((a * x) + b) mod 26
```

Die Ergebnisse aus der Verschlüsselung werden in eine neue leere Liste **y** gespeichert. Da in der Aufgabenstellung die Ausgabe in Großbuchstaben gefordert ist, müssen die Werte durch ein zuvor erstelltes Wörterbuch "lowercaseToUppercase" von kleinen Buchstaben zu Großbuchstaben umgewandelt werden. Die Großbuchstaben werden dafür in die zuvor erstelle Liste "encrypted_input" gespeichert:

```
1    encrypted_input.append(lowercaseToUppercase.get(encode(y)))
```

Als letztes wird die Liste mithilfe der join Methode als String ausgegeben:

```
1    return "".join(encrypted_input)
```

1.5 Aufgabe 5 - acDecrypt(a, b, cipher_text)

Bei Eingabe von **a=11, b=23** und **cipher_text=OVYNTWXAY** soll durch eine Entschlüsselung folgender Output entstehen:
"botschaft"
Auch hier wird wie in 1.4 eine Variable **orig_input** benötigt, die den cipher_text als Liste speichert. Nach Überprüfung der Schlüssel **a** und **b** durch die Hilfsfunktion **broken_key**, gibt die Funktion entweder einen leeren String zurück (bei "True") oder oder läuft weiter (bei "False").
Zum Entschlüsseln wird das inverse Element zum ersten Schlüsselteil **"a"** benötigt. Dieses wird aus dem in 1.3 erstellten Wörterbuch **key_table** gelesen und in **a_inv** abgespeichert.
Als nächstes muss für jeden Char von **plain_text** die Entschlüsselung durchgeführt werden. Dazu wird - wie in 1.4 auch - in einer for-Schleife der komplette String **plain_text** durchlaufen. Jedes Element wird dann zunächst als Integer gecastet, verschlüsselt und an eine Liste für die Rückgabe angehängt.
Die eigentliche Entschlüsselung erfolgt durch folgende Formel:

```
1    return ((y - b) * a_inv mod 26).
```

Die Ergebnisse aus der Entschlüsselung werden in eine neue leere Liste **tmp** gespeichert. Als letztes wird die Liste mit Hilfe der join-Methode als String zurückgegeben:

```
1    return "".join(decrypted_input)
```

1.6 Aufgabe 6 - Testen der ver- und entschlüsselung

1.6.1 Verschlüsseln des Klartexts ”strenggeheim” mit dem Schlüssel ”db”.

Es soll der Klartext **strenggeheim** mit dem Schlüssel **”db”** verschlüsselt werden. Dafür wird in der main()-Funktion ein print-Befehl verwendet:

```
1    print(acEncrypt(3, 1, "strenggeheim"))
2    # die Werte 3 und 1 entsprechen "d" und "b".
```

Dieser Befehl ruft die Funktion **acEncrypt** mit zwei Integer-Werten (dem Schlüssel) und einem String (dem zu verschlüsselnden Text) auf. Das Ergebnis (verschlüsselter String ”DGANOTTNWNZL”) wird auf der Konsole ausgegeben.

1.6.2 Entschlüsseln des Geheimtexts ”IFFYVQMJYFFDQ” mit dem Schlüssel ”pi”.

Um den Geheimschlüssel **IFFYVQMJYFFDQ** mit dem Schlüssel **”pi”** zu entschlüsseln, kann wie bei 1.6.1 vorgegangen werden:

```
1    print(acDecrypt(15, 8, "IFFYVQMJYFFDQ"))
2    # die Werte 15 und 8 entsprechen "p" und "i".
```

Dieser Befehl ruft die Funktion **acDecrypt** mit zwei Integer-Werten (dem Schlüssel) und einem String (dem zu entschlüsselnden Text) auf. Das Ergebnis (entschlüsselter String ”affinechiffre”) wird auf der Konsole ausgegeben.

1.7 Aufgabe 7 - Zusammenfassen im Modul aclib.py

Das bisher erstellte Skript wurde umbenannt zu **”aclib.py”** und somit als Modul abgespeichert.

1.8 Aufgabe 8 - Ver- / Entschlüsselung von .txt-Dateien in affinecipher.py

Für diese Aufgabe wurde ein neues Skript **"affinecipher.py"** erstellt.
Mit diesem Skript ist der User in der Lage, Dateien durch Kommandozeilen-Eingaben zu ver- oder entschlüsseln. Insgesamt werden drei Parameter benötigt:

- Angabe des Modus: "e" = verschlüsseln / "d" = entschlüsseln
- "xy" = Schlüssel als String mit zwei Buchstaben (Beispiel: "db")
- Pfad zur Datei

Um im Programm einen User-Input verarbeiten zu können, muss die Bibliothek **sys** importiert werden. Der User-Input muss zuvor auf Fehler zu überprüft werden, dafür gibt es die Funktion **"broken_input(arguments)"**. Diese bekommt als Parameter die übergebenen Kommandozeilenargumente, und liefert "True" falls der Input broken ist. Andernfalls "False".
Die erste If-Abfrage prüft, ob die Anzahl der Kommandozeilenargumente stimmt. Der erste Parameter muss hierbei der Name des Skripts ("affinecipher.py") sein. Die anderen drei Parameter wurden bereits definiert.
Die zweite If-else Abfrage prüft, ob der Modus richtig gewählt ist ("e" oder "d"). Sind hierbei keine Fehler aufgetreten, wird als nächstes die Länge des übergebenen Schlüssels geprüft. Dieser muss die Länge zwei haben.
Die letzte If-Abfrage prüft, ob der vorgegebene Wertebereich des Schlüssels eingehalten wurde. Hierfür werden die Funktion **decode** und **broken_key** benötigt. Durch einen Import des **aclib.py** Skripts kann die Funktion im neuen Skript verwendet werden: **import aclib**.
Der übergebene Schlüssel wird von Buchstaben in Zahlen umgewandelt, und anschließend in eine neue Variable **key_decoded** gespeichert:

```
1    key_decoded = aclib.decode(key)
```

Nun kann der Schlüssel durch die Hilfsfunktion **broken_key** geprüft werden, ob der Schlüssel gültig ist oder nicht:

```
1    aclib.broken_key(key_decoded[0], key_decoded[1])
```

Falls der Schlüssel nicht gültig ist, wird auch hier "True" zurückgegeben.
In der main() Funktion wird folglich die Funktion **broken_input** verwendet, um die erwähnten Überprüfungen durchzuführen und Fehler von Beginn an abzufangen. Der Befehl **"sys.argv"** ist eine Liste in Python, welche die in der Kommandozeile übergebenen Argumente enthält.

Die Variable **mode** steht für den Modus, wobei "e" = **verschlüsseln** und "d" = **entschlüsselt**:

```
1    mode = sys.argv[1]
```

Da die nächsten zwei Variablen (Die Schlüsselteile **a** und **b**) ohne Leerzeichen als Parameter in der Kommandozeile übergeben werden, muss hier durch Angabe der Indizes auf die benötigten Werte zugegriffen werden. Konkret heißt das: [0] = "erster Wert" und [1] = "zweiter Wert". Die Werte werden in den Variablen **a** und **b** gespeichert:

```
1     a = aclib.decode(sys.argv[2])[0]
2     b = aclib.decode(sys.argv[2])[1]
```

Die letzte Variable **path** enthält den Pfad zur Datei:

```
1     path = sys.argv[3]
```

Um die Datei schlussendlich zu ver- oder entschlüsseln, muss auf diese eingelesen werden. Durch den folgenden Befehl wird die Datei geöffnet und zeilenweise in der Liste **lines** gespeichert:

```
1     lines = open(path).read().splitlines()
```

Als letztes werden die einzelnen Zeilen in **lines** in einer for-Schleife ver- oder entschlüsselt und gemäß Aufgabenstellung mit einem print-Befehl auf der Konsole ausgegeben:

- Zum verschlüsseln:

```
1     print(aclib.acEncrypt(a, b, element))
```

- Zum entschlüsseln:

```
1     print(aclib.acDecrypt(a, b, element))
```

1.9 Aufgabe 9 - Verschlüsseln der Datei "klartext.txt" mit Schlüssel "pn"

Die Datei "**klartext.txt**" soll mit dem Schlüssel "**pn**" verschlüsselt werden.
Dazu muss nun das Skript "affinecipher.py" mit folgenden Parametern aufgerufen werden (Es wird davon ausgegangen, dass als Betriebssystem Linux verwendet wird und die Datei im selben Verzeichnis wie das Skript affinecipher.py liegt):

```
1     python affinecipher.py e pn ./klartext.txt
```

Dieser Befehl liefert folgenden Konsolen-Output:

```
1     EJMOPADXMVDAVMPWWVEIPZINLLDVIXEINROV
```

1.10 Aufgabe 10 - Entschlüsseln der Datei ”geheimtext.txt” mit Schlüssel ”pn”

Die Datei ”**geheimtext.txt**” soll mit dem Schlüssel ”**pn**” entschlüsselt werden.
Dazu muss nun das Skript ”affinecipher.py” mit folgenden Parametern aufgerufen werden (Es wird davon ausgegangen, dass als Betriebssystem Linux verwendet wird und die Datei im selben Verzeichnis wie das Skript affinecipher.py liegt):

```
1    python affinecipher.py d pn ./geheimtext.txt
```

Dieser Befehl liefert folgenden Konsolen-Output:

```
1    diesisteinstrenggeheimergeheimtextbittevertraulich
2    behandeln
```

2 Teil 2: Knacken der affinen Chiffre

2.1 Aufgabe 11 - computeFrequencyTable(char_list)

Die Funktion **computeFrequencyTable(char_list)** erhält als Parameter eine Liste von Zeichen in Zahlenform, und gibt eine sortierte Tabelle mit der Häufigkeit der einzelnen Zeichen zurück.
Dazu wird ein neues leeres Dictionary **tmp_frequency_table** erstellt, um die Häufigkeiten abzuspeichern. Um die Häufigkeit der Buchstaben zu zählen, läuft eine for-Schleife über die komplette **char_list**. Innerhalb der Schleife ist eine If-Abfrage, die - wenn der Buchstabe bereits gezählt wurde - an der Stelle die Häufigkeit um eins erhöht. Andernfalls wird der der entsprechende char als Key im Dicitonary mit der Häufigkeit eins angelegt.
Nun wird ein neues Wörterbuch **frequency_table** erstellt:

```
1     frequency_table = dict(zip(sorted(tmp_frequency_table),
2                                tmp_frequency_table.values()))
```

Hierbei werden die chars, ergo die Keys des Dictionarys durch die Funktion **sorted** aufsteigend sortiert - dabei geht allerdings die Zuordnung zwischen Keys und Values verloren - und anschließend mit der Funktion **zip** wieder in das neue Dictionary **frequency_table** eingetragen.
Um nun die Zuordnung von Zeichen und Häufigkeit wiederherzustellen, wird in der zweiten for-Schleife für jeden Key aus **tmp_frequency_table** der zugehörige Häufigkeitswert an entsprechender Stelle in **frequency_table** eingetragen:

```
1     for element in tmp_frequency_table:
2         frequency_table[element] = tmp_frequency_table[element]
```

Abschließend wird das neue, sortierte Dictionary **frequency_table** mit den entsprechenden Häufigkeiten zurückgegeben.

2.2 Aufgabe 12 - printFrequencyTable(freq_table)

Um eine in 2.1 erstellte Häufigkeitstabelle auszugeben wird die Funktion **printFrequencyTable(freq_table)** definiert, welche als Parameter eine Häufigkeitstabelle übergeben bekommt.
Über diese Tabelle läuft eine for-Schleife, die bei jedem Schleifendurchlauf das aktuelle Element aus Compiler-Gründen (Ansonsten: Fehlermeldung "'int' object is not iterable") an eine leere Liste anhängt. Dieses Element wird dann durch die Funktion **print** auf der Kommandozeile ausgegeben, wobei der Key noch mit der Funktion **encode** (siehe 1.1) aus dem Modul aclib.py von einer Zahl zu einem Buchstaben umgewandelt wird:

```
1         for element in freq_table:
2             tmp = list()
3             tmp.append(element)
4             print(aclib.encode(tmp), " : ", freq_table[element])
```

Diese Funktion besitzt keinen Rückgabewert.

2.3 Aufgabe 13 - computeMostFrequentChars(freq_table, n)

Diese Funktion soll die n häufigsten chars aus einer Häufigkeitstabelle berechnen, und bekommt dafür eine Häufigkeitstabelle **freq_table** und die gewünschte Anzahl **n** der zurückzugebenden chars.
Zunächst wird eine neue Liste **tmp** erstellt, welche die chars gemessen nach ihrer Häufigkeit absteigend (Funktion **reversed**) sortiert (Funktion **sorted**) speichert. D.h. der char der am häufigsten vorkommt, steht jetzt am Anfang der Liste.

```
1     tmp = reversed(sorted(freq_table, key=freq_table.__getitem__))
```

Aus Compiler-Gründen kann nicht über den return-Wert der "reversed"-Funktion in einer for-Schleife iteriert werden (Fehlermeldung "'list_reverseiterator' object is not subscriptable"). Deshalb wird eine neue leere Liste **freq_list** erstellt, und die Werte werden in einer for-Schleife von **tmp** nach **freq_list** kopiert.
Abschließend werden die ersten **n** Elemente der Liste zurückgegeben, was den **n** häufigsten Buchstaben entspricht:

```
1     return freq_list[:n]
```

Beispiel: Beim Aufruf mit den folgenden Parametern ...

```
1     freq_table = {0: 1, 4: 4, 6: 1, 7: 1, 8: 2, 11: 1,
2                   13: 5, 14: 1, 17: 1, 18: 1, 19: 2, 23: 1}
3     n = 6
```

... wird folgende Liste zurückgegeben:

```
1     [13, 4, 19, 8, 23, 18]
```

2.4 Aufgabe 14 - computeKeyPairs(char_list)

Diese Funktion bekommt eine Liste an chars übergeben, aus denen sie mögliche Schlüsselpaare bilden und zurückgeben soll.
Damit überhaupt ein Schlüsselpaar gebildet werden kann, wird zu Anfang der Funktion überprüft, ob mindestens zwei chars in **char_list** enthalten sind. Falls nicht, wird das Programm mit einer Fehlermeldung abgebrochen.
Es wird eine neue Liste **pair_list** erstellt, die anschließend mit den Schlüsselpaaren gefüllt und zurückgegeben wird.

```
1     pair_list = list()
```

Dafür wird die **char_list** mit zwei for-Schleifen durchlaufen. Die äußere Schleife stellt praktisch die linke Seite eines Tupels da, die innere Schleife die rechte. Jedes Element wird somit also zweimal mit jedem anderen Element kombiniert.
Falls die beiden Elemente aus **char_list** in einem Schleifendurchlauf ungleich sind, die Schleifeniteratoren sich also nicht an der gleichen Stelle befinden, werden beide Elemente als Tupel an die **pair_list** angehängt:

```
        for element1 in char_list:
            for element2 in char_list:
                if element1 != element2:
                    pair_list.append((element1, element2))
```

Wenn die **char_list** komplett abgearbeitet wurde, wird das Endergebnis - also die **pair_list** zurückgegeben.

2.5 Aufgabe 15 - analyzeCipherText(cipher_text, char_pairs)

Die Funktion **analyzeCipherText(cipher_text, char_pairs)** erhält einen verschlüsselten Text und eine Liste mit dem de facto Rückgabewert von **computeKeyPairs** als Übergabeparameter. Mit jedem möglichen char-pair wird dann versucht, den Geheimtext **cipher_text** zu entschlüsseln. Dafür wird eine for-Schleife verwendet.
Analog zur Aufgabenstellung werden dann bei jedem Schleifendurchlauf die für die Entschlüsselung benötigten Schlüsselteile **a** und **b** aus **char_pairs** berechnet:

```
    key_a = 3 * (element[1] - element[0]) % 26
    key_b = element[0] - (4 * key_a) % 26
```

Nach erfolgreicher Überprüfung des Schlüsselkandidaten durch die Funktion **broken_key** aus dem Modul **aclib.py** werden die ersten 50 Zeichen des Geheimtexts mit dem aktuellen Schlüsselpaar in der Funktion **acDecrypt** aus dem Modul **aclib.py** entschlüsselt und auf der Kommandozeile ausgegeben:

```
    print(acDecrypt(key_a, key_b, cipher_text[:50]),
        " --- Schluessel: a = ", key_a, ", b = ", key_b
```

Außerdem wird jedes Mal der verwendete Schlüssel hinter dem entschlüsselten Text ausgegeben.
Diese Funktion hat keinen Rückgabewert.

2.6 Aufgabe 16 - Zusammenfassen im Modul ablib.py

Die folgenden Funktionen wurden in dem neuen Modul **ablib.py** gespeichert:

- computeFrequencyTable(freq_table, n)
- printFrequencyTable(freq_table)
- computeMostFrequencyChars(freq_table, n)
- computeKeyPairs(char_list)
- analyzeCipherText(cipher_text, char_pairs)

2.7 Aufgabe 17 - Verwenden von ablib.py in affinebreaker.py

Die neue Datei **affinebreaker.py** importiert folgende Module:

```
1     import aclib  # Um freq_table zu decoden
2     import ablib  # Import des eigentlichen Mechanismus
3     import sys  # Für die Verarbeitung von Kommandozeilenargumenten
```

Laut Aufgabenstellung soll an das Skript lediglich der Dateipfad der zu knackenden Datei übergeben werden, d.h. die Anzahl der übergebenen Argumente muss zwei sein, was uns zu folgender Abbruchbedingung führt, die eben genau das überprüft:

```
1     if len(sys.argv) != 2:
```

Sollten es ungleich zwei Werte sein, wird das Programm mit einer Fehlermeldung beendet. Die folgenden Schritte wurden bereits in 1.8 zuvor detailliert erklärt, deshalb wird das in dieser Aufgabe abgekürzt:

Damit die Datei entschlüsselt werden kann, wird der Inhalt zeilenweise in der Variable **lines** gespeichert.

```
1     lines = open(sys.argv[1]).read().splitlines()
```

Um die einzelnen Zeilenstrings in einer Variable zu speichern, wird die **join** Funktion verwendet. Dadurch wird der komplette Inhalt von **lines** in der neuen Variable **cipher_text** gespeichert:

```
1     cipher_text = "".join(lines)
```

Daraus wird dann die Häufigkeit der einzelnen Buchstaben mit der Funktion **computeFrequencyTable** aus dem Modul **ablib.py** ermittelt und in **freq_table** gespeichert:

```
1     freq_table = ablib.computeFrequencyTable(aclib.decode
2                  (cipher_text))
```

Die sechs häufigsten Buchstaben aus dem Text werden mit der Funktion **computeMostFrequentChars** aus dem Modul **ablib.py** berechnet und in **freq_chars** gespeichert:

```
1     freq_chars =  ablib.computeMostFrequentChars(freq_table, 6)
```

Jetzt müssen alle möglichen Schlüsselkandidaten gebildet werden. Dafür wird die Funktion **computeKeyPairs** aus dem Modul **ablib.py** ausgeführt und der Rückgabewert in **chair_pairs** gespeichert:

```
1     char_pairs = ablib.computeKeyPairs(freq_chars)
```

Zum Schluss werden die fünf ersten, ergo wahrscheinlichsten Schlüsselpaare mit den ersten 50 Zeichen (Empfehlung der Aufgabenstellung) des **cipher_text** ausprobiert:

```
1      ablib.analyzeCipherText(cipher_text[:50], char_pairs[:5])
```