Introduction to Programming - Re-exam Project (Part I)

Simon Dradrach Jørgensen
Supervisor: Peter Schneider Kamp
DM550

7. september 2017

Indhold

1 Specification

The given task is to write a program in *Python*, that encrypts a message, based on the **Caesar-cipher**, which is used in **cryptography**. The way it works is, a person is given a **plain text** to encrypt into a coded message, this is done by shifting the letters in the message, by a value of n, e.g.,

shifting w by 6: $w \rightarrow x \rightarrow y \rightarrow z \rightarrow a \rightarrow b \rightarrow c$ *(result is c).*

There's in-built functions in python that can come up with a needed result, rather easily aswell. However, knowledge of the required key is a great threat for the *encrypted* message, it'd become too easy to decipher, and serve no purpose.

With that goal in mind, one can quickly figure out that it's not safe to have a program with a given key to encrypt the message, since the key is known. The second task brings that issue to mind, where an encrypted message is given, without a known key. Here one is supposed to find a way to find the desired key to decrypt the message.

The program should be able to support multiple *sets* of symbols. To help create the range of symbols, a function must be designed, such as the function is capable of receiving a starting entry and an end entry. The starting entry will in this case be the first *character* in both *lists* of the two-dimensional *list*, and the end entry will be the last *character* in the *lists*.

2 Design

2.1 Part 1

If one could use the key to to shift through the *characters* in the message given, the task can be completed in a rather simple way; with the useage of **integers**. Python can convert *characters* to integers, with the **ord()** in-built function. This makes one capable in shifting through the desired n-value of shifts, to encrypt a message, and to convert the integers back to *characters*, Python has another in-built function, the **chr()**, which converts the integers back to *characters*. As one would think, the way to decrypt an encrypted message, is simply to do the shifts backwards, meaning, instead of doing n shifts, one would have to do -n shifts.

The problem with using **ord()** and **chr()** however, is that they represent the *alphabet* in a *homogenize* way, since there's an unnecessary space between the small and capital letters. Instead of converting *characters* into integers, they can be indexed in a *set*, e.g., {0 : a, 1 : b, 2 : c, 3 : d, 4 : e}. Because shifting from small to capital letters would be unwise, these sets are to be two-dimensional, e.g.,

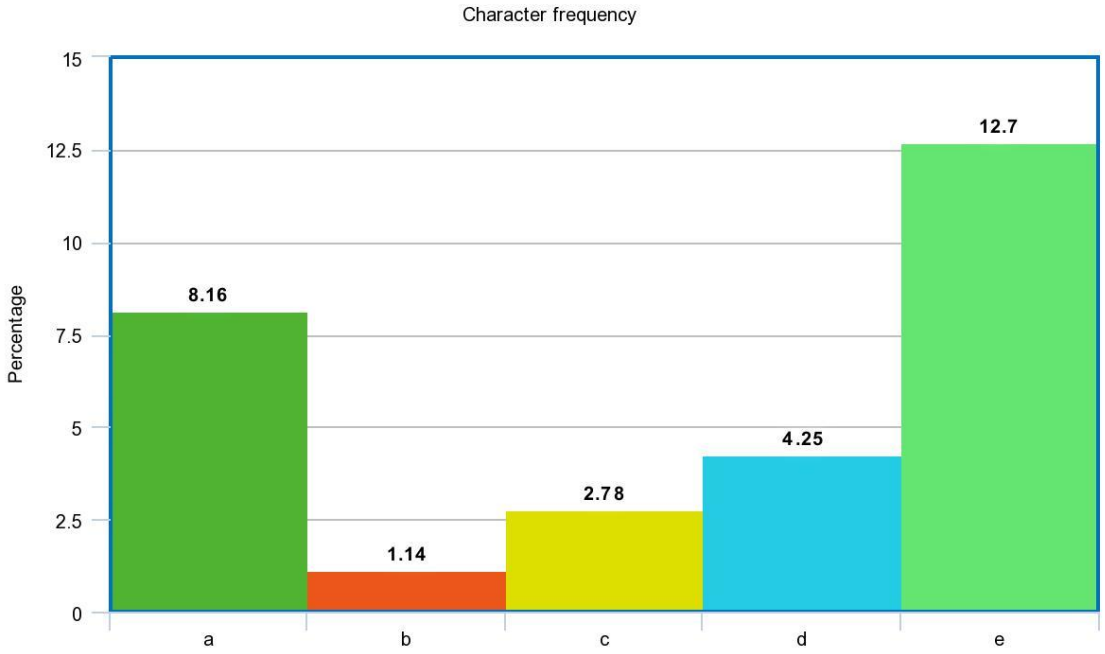
{{0 : a, 1 : b, 2 : c, 3 : d, 4 : e}, {0 : A, 1 : B, 2 : C, 3 : D, 4 : E}}.

2.2 Part 2

The specification requires a *histogram class*, this class requires two different constructors, these constructors are to differentiate between a **.dat** file, and a **.crypt** files. The first constructor, is to load each *character*, with its own occurence and the second constructor is to account each occurence for every occuring *character*. During the counting, the constructor is to filter symbols not inside the two-dimensional list. The class is also to own a function to create

a list, this list is to be sorting with the keys, being the occurence of every symbol, descending. A final function must be implemented to enable statistical abbreviation of the key, using the list of two different *histograms*.

The way *histogram* works in the following task, is by using the **.dat** file, to construct *frequencies*[1]:



Figur 1: Histogram

These *frequencies* were used to plot the overall occurences of a given *character* in the *lists* within the *dictionary*.

3 Implementation

3.1 Part 1

Due to the requirements of the *sets*, a function for creating a range of the symbols that are to be rotated, must be implemented. The way the rotations work, is the following example:

- $\{a, b, ..., x, z, \rightarrow, a, b\}$

The reasoning for this requirement, is because of the **ord()** function converting the *characters* into **unicode**, which are essentially *integers*, e.g.,

```
1 print(ord('a'))
```

The following result would be 97, and *z* would be 122 in the row. However, if one would shift an additional time, one would get the symbol, "{", therefore it's important to have a *set* range between *a* and *z*, and when the shift is made from *z*, the list would rotate back from the start. The exact same idea is used for capital letters.

To convert the *integers* back to *characters*, one must simply use the reverse function of **ord()**, with the in-built function **chr()**:

```
1 print(chr(97))
```

This will result in: *a*, which is the inverse of the already implemented **ord()** function. With these in-built functions from the python library, it's possible to convert a given *character* into its *integer* representation, and shift with that, before converting back to the *character* representation of the now shifted *integer*.

```
1 def srange(s, e = None, step = 1 ,sj = 0, ej = 0):
2     r = range(ord(s) + sj, ord(e) + ej, step) if e else range(0, ord(s) +
          sj, step)
3     return [chr(x) for x in r]
```

Since Python already has an in-built range-function, taking inspiration from that function, would be the most ideal solution for the arguments. However, due to Python's inability to increment *characters*, two arguments for incrementing the *characters* must be implemented to increment the *characters*, hereby the origin of the *sj (start-jump)*, and *ej (end-jump)*.

When creating the cipher, one must realize that the *sets* of *characters* that are to be rotated, is a two-dimensional list, hereby allowing for each list inside the two-dimensional list, to be rotated individually. Due to the fact, that the most idiomatic way is to replace *characters* in a string, is via a dictionary, one must create a dictionary for each list in the two-dimensional list, and *merge* the created dictionaries to obtain one complete dictionary, to be later used in the in-built *string-translate* function.

To do this, two functions must be created locally inside the cypher-function, these functions are the *table* and *merge* functions.

```
1 def cypher(key, message):
2     def table(l):
3         ...
4     def merge(l):
5         ...
```

The *table* function creates a dictionary, with the keys being an integer representation of each of the elements in the list, and the value be each of the rotated elements in the list. This setup fits the arguments of the in-built *string-translate* function. The *merge* function will run *n* times, depending on the length of the first dimension in the two-dimensional list, with each arguments being the output of a new **iteration**, of the *table* function. Thus the in-built *string-translate* function will use the *merged* dictionary to rotate all necessary *characters*.

3.2 Part 2

In python for a class to have multiple constructors, the useage of the @staticmethod *wrapper-class* is required. Since @staticmethod creates a non-member static function, it gives the idea of non-preference, since in multiple languages, static methods are not allowed to change the input *parameters*. Because the function to compare the two lists from the *histograms* has no preference, and the lists are not a variable in the *histogram* themselves.

The function for loading *.dat*-file aptly named *key*, takes every line in the given *string* and uses the first *sub-string* as the key, and the second *sub-string* as the value. Afterwards, calls the *histogram's* default constructor with a list of *dictionaries* and a *boolean*, which tells the constructor whether or not it should fractor the given list of *dictionaries*. However, to load a sample file, the *key* function, needs some help, an example for this, could be:

```
1 eng = histogram.key((filter(lambda s: len(s), x.split(" ")) for x in f))
```

The function for loading a *.crypt* file, aptly named *text*, counts every occurrence, for every character occurring in the given *string*.

The default constructor for the *histogram* class, allows for *fracturing* of the given lists of *dictionaries*. *Fracturing* follows the following **Pseudo-code**:

for $d \in \{\{a:?...z:?\}, \{A:?...Z:?\}...\}$ **do**
  $s := \sum_{i=0}^{|d|-1} (d_i)_1$;
  for $v \in d$ **do**
    $v_1 \leftarrow \frac{v_1}{s}$;
  end
end

This algorithm replaces each keys absolute occurrence, with relational occurrence, compared to the sum of all occurrences.

As the specification demands, the *histogram* class is to have a function, capable of creating a sorted *list* from the *histogram* itself. This is done by converting the *dictionary* in the *histogram* into a list, and sorting every element in the *list* depending on the value of the key.

for $d \in \{\{a:?...z:?\}, \{A:?...Z:?\}...\}$ **do**
  $d \leftarrow \{e_0 | e \in sort\ d\}$
end

To calculate the desired key, one initially calculates a list of the distances between each element with the following function: $f(a,b) = (b-a)\ mod\ 26$. The most occuring element is then chosen as the key.

4 Testing

In the testing phase of part 1, a simple logical mindset was used, converting a *string* into an *integer*, and then adding an n-value to it, before converting it back to a *string*. Then printing the *string* to check if the characters had shifted towards the desired value.

Part 2 was a bit tricky, due to the fact that a *dictionary* does not add elements in a predictable order, thus an ordered *dictionary* was required for this task. A number of statistical functions were also used, however, the function that returned the most occurring element proved to be the most proficient function.

To launch the program, one must use the following command:

4.1 Part 1

```
$ python3 caesar.py ( N ) < hello.txt > result.txt
```

Where N is the value of the shifts, that the program will perform before saving the file at the desired location, with the given result after the shift.

4.2 Part 2

```
$ python3 histogram.py engelsk.dat < engelsk1.crypt.txt > result.txt
```

Where the program will read the encrypted **.crypt** file, and save the decrypted **.txt** file in the desired location.

5 Conclusion

To conclude the project, the useage of *dictionaries* in python, is to be respected, since the order of inserted elements is **inhomogeneous**. Sadly, only 2 out of 3 **.crypt** files were solved, namely: **engelsk1.crypt.txt** and **engelsk3.crypt.txt**. The file that couldn't be solved had too many distances with equal or almost equal occurrences. This can point towards, that the program might not be perfect to solve the required task, even if it did solve 2 out of 3 tasks.

The **python3 interpreter** is required to launch the program, due to the useage of the *system* library, e.g.,

```
print(cypher(args.key, sys.stdin.read()))
```

6 Appendix (source code)

6.1 Part 1

```
def srange(s, e = None, step = 1 ,sj = 0, ej = 0):
    #[s+sj;e+ej;step[
    r = range(ord(s) + sj, ord(e) + ej, step) if e else range(0, ord(s) +
         sj, step)
    return [chr(x) for x in r]
#s = start
#sj = start jump
#e = end
#ej = end jump

hitchars = [srange("a", "z", ej = 1), srange("A", "Z", ej = 1)]
#[a;z+1[, Range of all lowercase characters
#[A;Z+1[, Range of all uppercase characters

def cypher(n, s, tables = hitchars):
    def table(l):
        return {ord(x) : y for x, y in
                   zip(l, l[n:] + l[:n])}

        #l = Original l (list)
        #l[n:] + l[:n] = Rotated l (list)

    def merge(l):
        i = iter(l)
        #create iterator for l (list).
        z = next(i).copy()
        #get the first element from the iterator and copy it
        for x in i:
            #get the remaining elements
            z.update(x)
            #Update the copied element(z) with the remaining elements in
                the iterator

        return z

    s = s.translate(merge((table(l) for l in tables)))
    return s

def decypher(n, s, tables = hitchars):
    return cypher(-n, s, tables)

if __name__ == "__main__":
    import argparse, sys
    parser = argparse.ArgumentParser(description='Run a ceasar cypher.')
    parser.add_argument('key', metavar = 'N', type = int, help = 'The key
         for shifting the charaters')
    args = parser.parse_args()
    print(cypher(args.key, sys.stdin.read()))
```

6.2 Part 2

```
from operator import itemgetter
import collections
from collections import OrderedDict
from caesar import hitchars, decypher

def find(c, ll):
    for i, l in enumerate(ll):
        if c in l:
            return i
    return -1

class histogram:
    def __init__(self, ld, frac=False):
        if frac:
            self.stat = ld
            for d in self.stat:
                s = sum(d.values())
                for k in d:
                    d[k] = d[k]/s
        else:
            self.stat = ld

    def list(self):
        def f(d):
            return [e[0] for e in sorted(d.items(), key = itemgetter(1),
                reverse = True)]
        return [f(d) for d in self.stat if d]

    @staticmethod
    def compare(a, b):
        #Estimate shift.
        def dist(x, y, n):
            return ((hitchars[n].index(y) - hitchars[n].index(x)))%len(
                hitchars[n])
        def mode(l):
            return collections.Counter(l).most_common(1)[0]


        d = [dist(x, y, i) for i, ll in enumerate(zip(a,b)) for x, y in
            zip(ll[0], ll[1])]
        return mode(d)[0]

    @staticmethod
    def text(s, frac=False):
        #to be used as a constructor
        #creates a histogram from a text
        ld = [OrderedDict() for x in hitchars]
        for c in s:
            n = find(c, hitchars)
            if n >= 0:
                ld[n][c] = ld[n].get(c, 0) + 1
```

```
49          return histogram(ld, frac)
50
51      @staticmethod
52      def key(s, frac=False):
53          #to be used as a constructor
54          #creates a histogram from a list of element and occurences
55          ld = [OrderedDict() for x in hitchars]
56          for c, v in s:
57              n = find(c, hitchars)
58              if n >= 0:
59                  ld[n][c] = float(v)
60          return histogram(ld, frac)
61
62      def __repr__(self, digits=3):
63          return repr([{k : round(d[k], digits) for k in d} for d in self.
                stat if d])
64
65  if __name__ == "__main__":
66      import sys
67      with open(sys.argv[1]) as f:
68          eng = histogram.key((filter(lambda s: len(s), x.split(" ")) for x
                in f))
69
70      s = "".join(sys.stdin.readlines())
71      t = histogram.text(s)
72      print(decypher(histogram.compare(eng.list(), t.list()), s))
```

Litteratur

[1] https://www.meta-chart.com/histogram