Laboratorium kryptograficzne dla licealistów 5

Projekt „Matematyka dla ciekawych świata”
Łukasz Mazurek

27.04.2017

1 Najczęstsze słowa

Na poprzednim wykładzie poznaliśmy metodę łamania szyfru podstawieniowego poprzez analizę najczęstszych słów jedno- dwu- i trzyliterowych. Na dzisiejszych zajęciach przećwiczymy tę metodę w praktyce.

1.1 Słowniki

Dotychczas do zliczania wystąpień poszczególnych liter w tekście używaliśmy 32-literowej listy `ile`, w której indeks odpowiadał pozycji litery w alfabecie: `ile[0]` przechowywało liczbę liter `A`, `ile[1]` przechowywało liczbę liter `Ą`, itd. Czy tą samą metodą da się zliczyć wystąpienia słów dwuliterowych? Moglibyśmy analogicznie użyć listy 1024-elementowej ($32 \cdot 32$) na wszystkie potencjalne słowa dwuliterowe. Wówczas `ile[0]` oznaczałoby liczbę słów `AA`, `ile[1]` oznaczałoby liczbę słów `AĄ`, itd. Widzimy jednak, że to raczej słabe rozwiązanie — większość elementów listy byłaby nieużywana (w rzeczywistości słów dwuliterowych jest dużo mniej niż 1024).

Właśnie do takich zadań idealnie nadaje się *słownik* (zwany również czasami *mapą*). Słownik to obiekt bardzo podobny do listy, z tym że jego poszczególne elementy są indeksowane nie liczbami, a dowolnymi innymi obiektami, np. słowami. O słowniku można myśleć jak o zbiorze par (*klucz* → *wartość*), przy czym zarówno *klucze*, jak i *wartości* mogą być dowolnego typu.

```
 > sl = {'Janek': 8, 'Marta': 10, 'Jurek': 8}
 > sl['Krzysiek'] = 9
 > sl
=> {'Marta': 10, 'Janek': 8, 'Jurek': 8, 'Krzysiek': 9}
 > sl['Marta']
=> 10
 > sl['Krysia']
Traceback (most recent call last):
  File "python", line 1, in <module>
KeyError: 'Krysia'
```

Zwróć uwagę, że kolejność elementów nie jest w żaden sposób zachowana: elementy nie są posortowane, nie występują też w kolejności dodawania do słownika.

Tak jak mówiłem, zarówno klucze, jak i wartości mogą być dowolnego typu — mogą być liczbami, słowami, ale również bardziej skomplikowanymi obiektami, np. parami:

```
 > sl[2] = 4
 > sl[17] = 'Jadzia'
 > sl[(2,'Ola')] = -1
 > sl
=> {'Marta': 10, 'Janek': 8, 'Krzysiek': 9, 17: 'Jadzia',
    (2, 'Ola'): -1, 2: 4, 'Jurek': 8}
```

1.2 Projekt na dzisiejsze zajęcia

Na dzisiejszych zajęciach będziemy korzystać z dobrze nam znanego pliku `lalka.txt` oraz z dwóch nowych plików: `tekst4.txt` oraz `precode.py`. Z tej okazji został przygotowany projekt na `repl.it` zawierający wszystkie potrzebne pliki. Aby zacząć pracę w tym projekcie, kliknij na link `PROJEKT DESZYFRATOR` na stronie `ciekawi.icm.edu.pl` → *Dla uczestników* → *Materiały - ćwiczenia* (pod tematem *Lab 5. Łamiemy szyfr podstawieniowy*). Projekt ten zawiera pliki `tekst4.txt`, `lalka.txt`, `precode.py` i `main.py`. Plik `tekst4.txt` zawiera szyfrogram, który będziemy dzisiaj rozszyfrowywać. Plikiem `precode.py` na razie się nie przejmujmy. Cały kod będziemy pisać w pliku `main.py`.

Jeżeli z jakichś powodów nie uda Ci się otworzyć projektu `DESZYFRATOR`, możesz stworzyć pusty projekt na `repl.it` i ręcznie dodać do niego poszczególne pliki. Każdy z plików można osobno pobrać ze strony `ciekawi.icm.edu.pl` → *Dla uczestników* → *Materiały - ćwiczenia*.

1.3 Najczęstsze słowa jednoliterowe

Spróbujmy wypisać wszystkie jednoliterowe słowa występujące w pliku `lalka.txt`. W tym celu użyjemy polecenia `tekst.split()`, które zamienia jeden długi tekst na listę słów występujących w tym tekście. Zmień zawartość pliku `main.py` na następującą:

```
plik = open('lalka.txt', 'r')
lalka = plik.read().upper()
slowa = lalka.split()
plik.close()

for s in slowa:
  if len(s) == 1:
    print(s)
```

i uruchom program przyciskiem *run*. Program powinien wypisać z tekstu wszystkie słowa jednoliterowe, każde w osobnym wierszu.

Zwróć uwagę, jak aplikowane są kolejno po kropce polecenia `read()`, `upper()` i `split()`. W tej konstrukcji najpierw dla pliku `plik` wywołujemy polecenie (*metodę*) `read()`, która czyta tekst z pliku. Następnie dla tekstu `plik.read()` wywołujemy metodę `upper()`, która zamienia w tekście małe litery na wielkie i zapisujemy wynik w zmiennej `lalka`. Na koniec wywołujemy metodę `split()` na tekście `lalka`, aby zamienić jeden, długi tekst na listę pojedynczych słów. Na upartego moglibyśmy wszystkie te polecenia wykonać w jednej linijce:

```
slowa = open('lalka.txt', 'r').read().upper().split()
```

Jednak taki zapis po pierwsze robi się nieczytelny, a po drugie w ten sposób tracimy dostęp do „etapów pośrednich", czyli do zmiennych `plik` i `lalka`, a te zmienne mogą nam się jeszcze przydać. Dlatego pozostaniemy przy wersji trzylinijkowej.

Spróbujmy teraz zliczyć wystąpienia wszystkich jednoliterowych słów:

```
plik = open('lalka.txt', 'r')
lalka = plik.read().upper()
slowa = lalka.split()
plik.close()

ile = {}

for s in slowa:
  if len(s) == 1:
    if s in ile:
      ile[s] += 1
    else:
```

```
        ile[s] = 1

print(ile)
```

W powyższym kodzie:

- tworzymy pusty słownik `ile`,
- dla każdego jednoliterowego słowa `s` sprawdzamy, czy słowo to już występuje jako klucz w słowniku `ile`, przy użyciu polecenia `s in ile`,
- jeśli występuje, to zwiększamy wartość `ile[s]` o 1,
- jeśli nie występuje, to ustawiamy wartość `ile[s]` na 1.

Jeżeli się nie pomyliłeś, program powinien wypisać słownik o następującej zawartości:

```
{'Z': 363, 'W': 546, 'U': 34, 'I': 784, 'A': 288, '-': 1095,
 ':': 1, 'O': 131}
```

Zadanie 1 *Program uznał za słowa m.in. słowa '-' i ':'. Zmodyfikuj program tak, aby dodawał do słownika tylko słowa składające się z liter należących do polskiego alfabetu.*

1.4 Sortowanie

Zawartość słownika wypisywana jest w przypadkowej kolejności. My natomiast chcielibyśmy wypisać słowa od najczęściej do najrzadziej występującego. W tym celu zamienimy najpierw słownik na listę przy użyciu polecenia `items()`, a następnie posortujemy ją przy użyciu polecenia **sorted**`()`:

```
print(ile.items())
```

```
dict_items([('Z', 363), ('I', 784), ('A', 288),
  ('W', 546), ('O', 131), ('U', 34)])
```

```
print(sorted(ile.items()))
```

```
dict_items[('A', 288), ('I', 784), ('O', 131),
  ('U', 34), ('W', 546), ('Z', 363)]
```

Lista rzeczywiście została posortowana, ale niestety nie po częstościach, tylko po literach, alfabetycznie. Na szczęście istnieje dodatkowa opcja, za pomocą możemy podać funkcji **sorted** klucz do sortowania. Aby posortować listę par po drugim elemencie pary należy użyć polecenia:

```
print(sorted(ile.items(),
  key = lambda x: x[1]))
```

```
[('U', 34), ('O', 131), ('A', 288),
  ('Z', 363), ('W', 546), ('I', 784)]
```

Wyrażenie `key = lambda x: x[1]` oznacza dokładnie tyle: dla każdej pary `x`, jako klucza do sortowania użyj drugiego elementu tej pary. Widzimy, że tak sformułowane polecenie posortowało litery rosnąco ze względu na częstość wystąpień. Okazuje się, że funkcja **sorted** ma również opcję do odwracania kolejności sortowania:

```
print(sorted(ile.items(),
  key = lambda x: x[1],
  reverse = True))
```

```
[('I', 784), ('W', 546), ('Z', 363),
  ('A', 288), ('O', 131), ('U', 34)]
```

Na koniec możemy pozbyć się drugiego elementu każdej pary i wypisać same słowa, od najczęstszego do najrzadszego:

```
lista = sorted(ile.items(), key = lambda x: x[1],
  reverse = True)
for x in lista:
  print(x[0], end = ' ')
```

```
I W Z A O U
```

Zadanie 2 *Napisz funkcję* `najczestsze(tekst)`*, która wypisze: wszystkie słowa jednoliterowe, 10 najczęstszych słów dwuliterowych i 10 najczęstszych słów trzyliterowych występujących w danym tekście.*

2 Łamanie szyfru podstawieniowego

Przejdźmy do gwoździa programu, czyli do złamania szyfru podstawieniowego! W pliku `tekst4.txt` znajduje się tekst zaszyfrowany pewnym szyfrem podstawieniowym. W pliku `precode.py` znajduje się kod Deszyfratora, który udostępnia funkcje pomagające w złamaniu szyfru.

Deszyfrator działa w ten sam sposób, w jaki łamaliśmy szyfr podstawieniowy podczas ostatniego wykładu: najpierw obliczane są rozkłady częstości występowania liter w szyfrogramie i pliku `lalka.txt`. Na tej podstawie tworzona jest permutacja przyporządkowująca litery w następujący sposób:

- najczęstsza litera z szyfrogramu rozszyfrowywana jest na najczęstszą literę z Lalki,
- druga najczęstsza litera z szyfrogramu rozszyfrowywana jest na drugą najczęstszą literę z Lalki,
- itd.

Aby korzystać z funkcji z pliku `precode.py`, musimy najpierw wykonać w pliku `main.py` dwie linijki:

```
from precode import Deszyfrator
d = Deszyfrator()
```

Pierwsza z nich importuje Deszyfrator i wszystkie jego funkcje z pliku `precode.py`. Druga linijka tworzy Deszyfrator i zapisuje go w zmiennej `d`. Od tej pory możemy używać wszystkich potrzebnych funkcji z pliku `precode.py`.

Sprawdźmy, jak działa permutacja wygenerowana przez Deszyfrator. W tym celu zmień treść pliku `main.py` na następującą:

```
from precode import Deszyfrator
d = Deszyfrator()
d.sprawdz_permutacje()
```

i uruchom program przyciskiem *run*. W efekcie program powinien wypisać garść informacji na temat aktualnego działania Deszyfratora. Są to kolejno:

1. Aktualna postać permutacji używanej do rozszyfrowywania. Górna linijka odpowiada alfabetowi szyfrogramu, a dolna alfabetowi rozszyfrowanej wiadomości. Tak więc pierwsza litera górnej linijki jest rozszyfrowywana na pierwszą literę dolnej, itd.
2. Najczęstsze słowa jedno- dwu- i trzyliterowe w szyfrogramie i w rozszyfrowanej wiadomości. Tutaj również w każdym przypadku górne słowo jest rozszyfrowywane na dolne.
3. Fragment szyfrogramu rozszyfrowanego aktualną wersją permutacji.

Widzimy, że początkowa postać permutacji, krótko mówiąc, działa źle. Dlatego trzeba będzie ją poprawić.

2.1 Funkcje pomocnicze

Deszyfrator dostarcza dwóch narzędzi ułatwiających poprawianie permutacji. Pierwszym z nich jest funkcja `d.wypisz_jezyk()`. Wypisuje ona słowa jedno- dwu- i trzyliterowe najczęściej występujące w naszym języku (na podstawie fragmentu Lalki). Porównując te słowa i słowa wypisane przez `d.sprawdz_permutacje()` możemy próbować wydedukować, jakie słowa naprawdę występowały w zaszyfrowanym tekście.

Drugą przydatną funkcją jest funkcja `wypisz_kandydatow()`. Funkcja ta dla każdej litery wpisuje listę kandydatów na rozszyfrowanie tej litery: od najbardziej do najmniej prawdopodobnych.

Uruchom funkcje `d.wypisz_jezyk()` i `d.wypisz_kandydatow()`:

```
from precode import Deszyfrator
d = Deszyfrator()
d.wypisz_jezyk()
d.wypisz_kandydatow()
```

Powyższy kod powinien wypisać najczęstsze słowa z naszego języku oraz 32 listy kandydatów na rozszyfrowanie. Skopiuj wypisany tekst (np. do edytora tekstu). Te dane przydadzą nam się podczas rozszyfrowywania, jednak nie będą się one zmieniać wraz z modyfikacją permutacji, więc nie będzie potrzeby ponownego uruchamiania funkcji `d.wypisz_jezyk()` ani `d.wypisz_kandydatow()`.

2.2 Poprawianie permutacji

Wróćmy do programu sprawdzającego aktualną permutację:

```
from precode import Deszyfrator
d = Deszyfrator()
d.sprawdz_permutacje()
```

i zwróćmy uwagę na najczęstsze słowo trzyliterowe. Najczęstszym trzyliterowym słowem występującym w szyfrogramie jest słowo `PST` i zostało one rozszyfrowane na `WIĄ`. Będziemy mówić, że słowo `PST` *przechodzi* w permutacji na `WIĄ` albo, że w permutacji są *przejścia*: `P` → `W`, `S` → `I` i `T` → `Ą`.

Z zapisanych wcześniej wyników wiemy, że najczęstszym trzyliterowym słowem występującym w języku polskim jest słowo `SIĘ`. Może więc w naszym szyfrze słowo `PST` powinno przechodzić właśnie na `SIĘ`? Ale w jaki sposób upewnić się, czy tak jest w rzeczywistości? Co jeśli `PST` przechodzi na inne popularne słowo, np. `NIE`? Aby się o tym przekonać, spójrzmy na listy *kandydatów* dla liter `P`, `S` i `T`:

```
P: SCRWYTKDNMPZŁLJUBĘOGĄŻHÓŚEĆFŃŹIA
S: IEAOZNSCRWYTKDMPŁLJUBĘGĄŻHÓŚĆFŃŹ
T: ĄŻHGÓĘŚBĆFUŃJLŹŁPMDKTYWRCSNZOEIA
```

Istotnie, zgodnie z tymi listami, przejście `P` → `S` jest najbardziej prawdopodobnym przejściem dla litery `P`. Również przejście `S` → `I` jest najbardziej prawdopodobnym przejściem dla litery `S`. Litera `Ę` jest szóstą najprawdopodobniejszą literą dla przejścia z `T`. Wydaje się więc, że przejście `PST` → `SIĘ` jest dobrym przejściem w naszej permutacji (na pewno lepszym niż np. `PST` → `NIE`).

Ustalmy zatem w naszej permutacji przejście `PST` → `SIĘ`. W tym celu wykonaj następujący kod:

```
from precode import Deszyfrator
d = Deszyfrator()
d.ustal('PST', 'SIĘ')
d.sprawdz_permutacje()
```

W ten sposób „na sztywno” ustalamy trzy przejścia w permutacji, a pozostałe generowane są automatycznie przy użyciu list kandydatów. W dalszym ciągu deszyfrowania będziemy szukali kolejnych charakterystycznych słów i będziemy *ustalali* kolejne przejścia w permutacji kolejnymi wywołaniami funkcji `d.ustal()`. Tak więc pod koniec naszych zmagań plik `main.py` będzie miał postać:

```
from precode import Deszyfrator
d = Deszyfrator()
d.ustal(...)
d.ustal(...)
d.ustal(...)
...
d.sprawdz_permutacje()
```

Zauważ, że do kolejnych ustaleń można dochodzić nie tylko na podstawie analizy najczęstszych słów jedno- dwu- i trzyliterowych, ale również na podstawie analizy dłuższego fragmentu tekstu rozszyfrowywanego przez Deszyfrator. Np. aktualnie we fragmencie tym występuje słowo `uesteś`. Domyślamy się,

że powinno tam być `jesteś`. Zatem litera, która aktualnie przechodzi na `U` powinna przechodzić na `J`. Spoglądając na permutację widzimy, że aktualnie `F` przechodzi `U`. Powinniśmy więc dokonać ustalenia:

```
d.ustal('F', 'J')
```

Zwróć uwagę, że po dokonaniu pierwszych ustaleń, w informacjach wypisywanych przez Deszyfrator część rozszyfrowanego tekstu jest wypisana wielkimi literami, a część małymi. Dzieje się tak, ponieważ Deszyfrator wypisuje wielkimi literami fragmenty, które zostały rozszyfrowane przy użyciu *ustalonych* przez nas (przy użyciu funkcji `d.ustal()`) elementów permutacji, natomiast małymi literami wypisuje fragmenty rozszyfrowane przy użyciu pozostałej części permutacji wygenerowanej automatycznie. Dlatego podczas dalszego łamania szyfru będziemy zakładać, że fragmenty zapisane wielkimi literami są raczej dobrze rozszyfrowane i będziemy starali się poprawiać przede wszystkim elementy permutacji zapisane małymi literami. Oczywiście może się okazać, że któraś z naszych poprawek będzie błędna i wtedy trzeba będzie zmienić jedno z wcześniejszych *ustaleń* (program wypisze stosowny komunikat w przypadku, gdy wprowadzimy dwa wykluczające się ustalenia).

Zadanie 3 *Używając opisanych powyżej metod, rozszyfruj tekst zapisany w pliku* ***`tekst4.txt`***

3 Zadania dodatkowe

Zadanie 4 *Zmodyfikuj funkcję* ***`najczestsze(tekst)`*** *napisaną na początku zajęć w taki sposób, aby działała dla angielskiego alfabetu ABCDEFGHIJKLMNOPQRSTUVWXYZ. Wypisz najczęstsze słowa z pliku* ***`eng.txt`****.*

Zadanie 5 *Rozszyfruj plik* ***`tekst_eng.txt`****, wiedząc, że powstał poprzez zaszyfrowanie szyfrem podstawieniowym pewnego tekstu w języku* ***angielskim****.*

4 Praca domowa nr 4

Rozwiązania zadań należy przesłać do czwartku 11 maja do godz. 16^{59} na adres `licealisci.pracownia@icm.edu.pl` wpisując jako temat wiadomości `Lx PD5`, gdzie `x` to numer grupy, np. `L3 PD5` dla grupy L3, itd.

Zadanie domowe 1 (1 pkt) *Jak brzmi najdłuższe słowo w pliku* ***`lalka.txt`****?*

Zadanie domowe 2 (2 pkt) *Wypisz wszystkie słowa 4-literowe z pliku* ***`lalka.txt`*** *(1 pkt). Zwróć uwagę, żeby Twój program wypisywał tylko słowa składające się z 4 liter, a nie np. z trzech liter i jednego znaku interpunkcyjnego (1 pkt).*

Zadanie domowe 3 (3 pkt) *Rozszyfruj plik* ***`tekst5.txt`***[1]*, wiedząc, że powstał przez zaszyfrowanie szyfrem podstawieniowym pewnego tekstu napisanego w języku polskim.*

[1](dostępny do pobrania ze strony `ciekawi.icm.edu.pl`, zakładka *Dla uczestników* → *Materiały - ćwiczenia*)