Introduktion til Kryptering i Python - informatik b

Af Henrik Sterner

Introduktion

Kryptering er processen med at omdanne information, så den bliver uforståelig for uvedkommende. Det kan være afgørende for at beskytte følsomme data, især i en tid hvor digital kommunikation er udbredt. Kryptering finder anvendelse i mange områder, herunder sikre e-mails, online banktransaktioner og beskyttelse af personlige oplysninger.

I denne artikel vil vi se på nogle grundlæggende metoder til kryptering i Python ved hjælp af indbyggede biblioteker.

Fagbegreber

Før vi dykker ned i metoder til at kryptere data i Python, er det vigtigt at forstå nogle grundlæggende fagbegreber inden for kryptering:

- **Kryptering (Encryption)**: Processen med at omdanne læsbar information (klartekst) til en uforståelig form (chiffertekst) ved hjælp af en algoritme og en nøgle.
- **Dekryptering (Decryption)**: Processen med at omdanne chiffertekst tilbage til klartekst ved hjælp af den samme eller en anden nøgle.
- **Symmetrisk kryptering**: En krypteringsmetode, hvor den samme nøgle bruges til både kryptering og dekryptering af data.
- **Asymmetrisk kryptering**: En krypteringsmetode, hvor to forskellige nøgler bruges - en offentlig nøgle til kryptering og en privat nøgle til dekryptering.
- **Hashing**: En proces, hvor data omdannes til en fast længde streng (hash) ved hjælp af en hash-funktion. Hashing er envejskryptering, hvilket betyder, at det ikke er muligt at gendanne den oprindelige data fra hash-værdien.

En simpel krypteringsalgoritme: Caesar-chiffer

En af de mest grundlæggende krypteringsmetoder er Caesar-chifferet, hvor hvert bogstav i teksten flyttes et fast antal pladser i alfabetet. Den har sine historiske rødder i Julius Cæsars brug af denne metode til at beskytte militære beskeder.

Grundlæggende så virker Caesar-chifferet ved at tage hvert bogstav i den oprindelige tekst og erstatte det med et bogstav, der er et bestemt antal pladser længere fremme i alfabetet. For eksempel, hvis vi bruger en forskydning på 3, vil 'A' blive til 'D', 'B' til 'E', og så videre.

Herunder en tabel, der illustrerer forskydningen:

Klartekst	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Chiffertekst (forskydning 3)	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Hvis vi ønsker at kryptere teksten "Hej med dig" med en forskydning på 3, vil resultatet være "Kho phg glj".

Lad os prøve at skrive pseudokode for en simpel funktion, der implementerer Caesar-chifferet:

```
Funktion caesar_krypter(tekst, forskydning):
    initialiser tom streng krypteret_tekst
    For hvert tegn i tekst:
        Hvis tegn er et bogstav:
            find den tilsvarende position i alfabetet
            beregn ny position ved at tilføje forskydning (med wrap-around)
            tilføj det nye bogstav til krypteret_tekst
        Ellers:
            tilføj tegnet uændret til krypteret_tekst
    Returner krypteret_tekst
```

Denne pseudokode beskriver en funktion, der tager en tekst og en forskydning som input og returnerer den krypterede tekst ved hjælp af Caesar-chifferet.

Når vi skal dekryptere teksten, kan vi bruge den samme funktion, men med en negativ forskydning, hvorved vi flytter bogstaverne tilbage i alfabetet. Herunder pseudokoden for dekryptering:

```
Funktion caesar_dekrypter(krypteret_tekst, forskydning):
    initialiser tom streng dekrypteret_tekst
    For hvert tegn i krypteret_tekst:
        Hvis tegn er et bogstav:
            find den tilsvarende position i alfabetet
            beregn ny position ved at trække forskydning (med wrap-around)
            tilføj det nye bogstav til dekrypteret_tekst
        Ellers:
            tilføj tegnet uændret til dekrypteret_tekst
    Returner dekrypteret_tekst
```

Lad os nu implementere denne funktion i Python:

```python
def caesar_krypter(tekst, forskydning):
    krypteret_tekst = ""
    for tegn in tekst:
        if tegn.isalpha():
            start = ord('A') if tegn.isupper() else ord('a')
            ny_position = (ord(tegn) - start + forskydning) % 26 + start
            krypteret_tekst += chr(ny_position)
        else:
            krypteret_tekst += tegn
    return krypteret_tekst

def caesar_dekrypter(krypteret_tekst, forskydning):
    dekrypteret_tekst = ""
    for tegn in krypteret_tekst:
        if tegn.isalpha():
            start = ord('A') if tegn.isupper() else ord('a')
            ny_position = (ord(tegn) - start - forskydning) % 26 + start
            dekrypteret_tekst += chr(ny_position)
        else:
            dekrypteret_tekst += tegn
    return dekrypteret_tekst

# Eksempel på brug
klartekst = "Hej med dig"
forskydning = 3
krypteret = caesar_krypter(klartekst, forskydning)
print("Krypteret tekst:", krypteret)
dekrypteret = caesar_dekrypter(krypteret, forskydning)
print("Dekrypteret tekst:", dekrypteret)
```

Denne kode definerer to funktioner: `caesar_krypter` og `caesar_dekrypter`, som henholdsvis krypterer og dekrypterer en given tekst ved hjælp af Caesar-chifferet med en specificeret forskydning. Eksemplet viser, hvordan man kan bruge disse funktioner til at kryptere og dekryptere en tekst.

Caesar-chifferet er en simpel metode til kryptering og er ikke egnet til sikker kommunikation i moderne sammenhænge, men det tjener som en god introduktion til grundlæggende krypteringsprincipper. Vi kan bryde mere komplekse krypteringsmetoder ned i lignende trin og forstå de underliggende koncepter.

Nogle af grundene til, at Caesar-chifferet ikke er sikkert, inkluderer:

1. **Begrænset nøgleplads**: Der er kun 25 mulige forskydninger (1-25), hvilket gør det let at bryde ved hjælp af brute-force angreb.
2. **Frekvensanalyse**: Da visse bogstaver forekommer hyppigere i sproget, kan en angriber analysere frekvensen af bogstaver i den krypterede tekst for at gætte forskydningen.

Opgave til Caesar-chifferet

Herunder nogle opgaver, du kan prøve at løse for at øve dig på Caesar-chifferet:

0. Prøv følgende af i hånden:
    - at kryptere "Python er sjovt!" med en forskydning på 5
    - at dekryptere "Udymts jw xmtzy!" med en forskydning på 5
    - at finde forskydningen, hvis du ved, at "Khoor Zruog" dekrypteret skal være "Hello World"
1. Forklar , hvordan Caesar-chifferet fungerer, og hvorfor det ikke er sikkert til moderne kryptering.
2. Implementer en funktion i Python, der kan bryde en Caesar-krypteret tekst ved hjælp af frekvensanalyse. Dvs. find den mest sandsynlige forskydning baseret på bogstavfrekvenser i det danske sprog. Vi kan antage, at de mest almindelige bogstaver i dansk er E, A, R, N og T.
3. Lav en brute-force funktion, der prøver alle mulige forskydninger for at dekryptere en given tekst og returnerer alle mulige resultater.
4. Prøv at gøre Caesar-chifferet ved brug af substitution, hvor hvert bogstav erstattes med et andet bogstav baseret på en tilfældig permutation af alfabetet.
5. Prøv at overvej din egen krypteringsalgoritme, der kombinerer flere metoder, såsom Caesar-chifferet. Blandt andet kan du prøve:
    - At anvende forskellige forskydninger for vokaler og konsonanter.
    - At inkludere tal og specialtegn i krypteringsprocessen.
    - At implementere en simpel form for nøglebaseret kryptering, hvor nøglen bestemmer forskydningen for hvert bogstav.

Vigenere-chifferet

En anden klassisk krypteringsmetode er Vigenere-chifferet, som er en form for polyalfabetisk substitution. Det bruger en nøgle til at bestemme forskydningen for hvert bogstav i teksten, hvilket gør det mere sikkert end Caesar-chifferet.

I Vigenere-chifferet gentages nøglen over teksten, og hver bogstav i teksten forskydes baseret på den tilsvarende bogstav i nøglen.

For eksempel, hvis vi har teksten "ATTACKATDAWN" og nøglen "LEMON", gentages nøglen som "LEMONLEMONLE". Hvert bogstav i teksten forskydes derefter baseret på den tilsvarende bogstav i nøglen.

Vi kan opstille en tabel for Vigenere-chifferet:

Klartekst	**A**	**T**	**T**	**A**	**C**	**K**	**A**	**T**	**D**	**A**	**W**	**N**
Nøgle	L	E	M	O	N	L	E	M	O	N	L	E
Chiffertekst	L	X	F	O	P	V	E	F	R	N	H	R

Det fremgår, at 'A' forskydes med 'L' (11 pladser) til 'L', 'T' forskydes med 'E' (4 pladser) til 'X', og så videre.

Når vi skal dekryptere teksten, trækker vi forskydningen baseret på nøglen i stedet for at lægge den til. Herunder en tabel for dekryptering:

Chiffertekst	**L**	**X**	**F**	**O**	**P**	**V**	**E**	**F**	**R**	**N**	**H**	**R**
Nøgle	L	E	M	O	N	L	E	M	O	N	L	E
Klartekst	A	T	T	A	C	K	A	T	D	A	W	N

Dvs. tager vi for eksempel 'L' og trækker 'L' (11 pladser) for at få 'A', 'X' trækker 'E' (4 pladser) for at få 'T', og så videre.

Lad os prøve at skrive pseudokode for Vigenere-kryptering:

```
Funktion vigenere_krypter(tekst, nøgle):
    initialiser tom streng krypteret_tekst
    nøgle_længde = længde af nøgle
    nøgle_index = 0
    For hvert tegn i tekst:
        Hvis tegn er et bogstav:
            find den tilsvarende position i alfabetet
            find forskydning baseret på nøgle[nøgle_index % nøgle_længde]
            beregn ny position ved at tilføje forskydning (med wrap-around)
            tilføj det nye bogstav til krypteret_tekst
            nøgle_index += 1
        Ellers:
            tilføj tegnet uændret til krypteret_tekst
    Returner krypteret_tekst
```

Denne pseudokode beskriver en funktion, der tager en tekst og en nøgle som input og returnerer den krypterede tekst ved hjælp af Vigenere-chifferet.

På samme måde kan vi skrive pseudokode for dekryptering:

```
Funktion vigenere_dekrypter(krypteret_tekst, nøgle):
    initialiser tom streng dekrypteret_tekst
    nøgle_længde = længde af nøgle
    nøgle_index = 0
    For hvert tegn i krypteret_tekst:
        Hvis tegn er et bogstav:
            find den tilsvarende position i alfabetet
            find forskydning baseret på nøgle[nøgle_index % nøgle_længde]
            beregn ny position ved at trække forskydning (med wrap-around)
            tilføj det nye bogstav til dekrypteret_tekst
            nøgle_index += 1
        Ellers:
            tilføj tegnet uændret til dekrypteret_tekst
    Returner dekrypteret_tekst
```

Lad os nu implementere denne funktion i Python:

```python
def vigenere_krypter(tekst, nøgle):
    krypteret_tekst = ""
    nøgle_længde = len(nøgle)
    nøgle_index = 0
    for tegn in tekst:
        if tegn.isalpha():
            start = ord('A') if tegn.isupper() else ord('a')
            forskydning = ord(nøgle[nøgle_index % nøgle_længde].upper()) - ord('A')
            ny_position = (ord(tegn) - start + forskydning) % 26 + start
            krypteret_tekst += chr(ny_position)
            nøgle_index += 1
        else:
            krypteret_tekst += tegn
    return krypteret_tekst

def vigenere_dekrypter(krypteret_tekst, nøgle):
    dekrypteret_tekst = ""
    nøgle_længde = len(nøgle)
    nøgle_index = 0
    for tegn in krypteret_tekst:
        if tegn.isalpha():
            start = ord('A') if tegn.isupper() else ord('a')
            forskydning = ord(nøgle[nøgle_index % nøgle_længde].upper()) - ord('A')
            ny_position = (ord(tegn) - start - forskydning) % 26 + start
            dekrypteret_tekst += chr(ny_position)
            nøgle_index += 1
        else:
            dekrypteret_tekst += tegn
    return dekrypteret_tekst
```

Vi kan nu bruge disse funktioner til at kryptere og dekryptere en tekst ved hjælp af Vigenere-chifferet:

```python
# Eksempel på brug
klartekst = "ATTACKATDAWN"
nøgle = "LEMON"
krypteret = vigenere_krypter(klartekst, nøgle)
print("Krypteret tekst:", krypteret)
dekrypteret = vigenere_dekrypter(krypteret, nøgle)
print("Dekrypteret tekst:", dekrypteret)
```

Prøv at køre koden for at se, hvordan Vigenere-chifferet fungerer i praksis.

Vi kan se, at Vigenere-chifferet er mere komplekst end Caesar-chifferet, da det bruger en nøgle til at bestemme forskydningen for hvert bogstav. Dette gør det sværere at bryde ved hjælp af brute-force angreb eller frekvensanalyse, hvilket øger sikkerheden ved krypteringen.

Metoder som Vigenere-chifferet illustrerer, hvordan kryptering kan gøres mere robust ved at introducere variation og kompleksitet i processen. Det er vigtigt at forstå disse grundlæggende principper, da de danner grundlaget for mere avancerede krypteringsmetoder, der anvendes i moderne sikkerhedssystemer.

Eksempel på metoder til at bryde Vigenere-chifferet inkluderer Kasiski-undersøgelsen og Friedman-testen, som begge analyserer gentagelser i den krypterede tekst for at bestemme nøglens længde og derefter anvende frekvensanalyse på de enkelte segmenter.

Herunder formuleres først Friedman-testen i pseudokode:

```
Funktion friedman_test(krypteret_tekst):
    initialiser variabel I = 0
    initialiser variabel N = længde af krypteret_tekst
    For hvert bogstav i alfabetet:
        tælle forekomster af bogstav i krypteret_tekst som f
        I += f * (f - 1)
    I = I / (N * (N - 1))
    returner I
```

Funktionen beregner indeks for tilfældighed (I) for den krypterede tekst, hvilket kan hjælpe med at estimere nøglens længde. Dvs. jo højere I er, desto mere sandsynligt er det, at teksten er krypteret med en kortere nøgle. Vi overlader implementeringen af denne funktion i Python som en øvelse.

Alternativt kan du undersøge Kasiski-undersøgelsen, som involverer at finde gentagne sekvenser af bogstaver i den krypterede tekst og måle afstanden mellem deres forekomster for at estimere nøglens længde. Det er en mere kompleks metode, som også kan implementeres som en øvelse. Herunder er en simpel pseudokode for Kasiski-undersøgelsen:

```
Funktion kasiski_undersøgelse(krypteret_tekst):
    initialiser tom liste gentagelser
    For længde i 3 til 5:
        For start_index i 0 til (længde af krypteret_tekst - længde):
            sekvens = substring af krypteret_tekst fra start_index til start_index + længde
            find alle forekomster af sekvens i krypteret_tekst
            hvis antallet af forekomster > 1:
                beregn afstande mellem forekomster og tilføj til gentagelser
    returner gentagelser
```

For at give en bedre ide om hvordan Kasiski-undersøgelsen virker i praksis kan vi se på et eksempel. Antag, at vi har den krypterede tekst "LXFOPVEFRNHR" og vi leder efter gentagne sekvenser af længde 3. Vi finder sekvensen "LXF" to gange i teksten, og afstanden mellem deres startpositioner kan hjælpe os med at estimere nøglens længde.

Opgave til Vigenere-chifferet

Herunder nogle opgaver, du kan prøve at løse for at øve dig på Vigenere-chifferet:

0. Prøv følgende af i hånden:
    - at kryptere "HELLO WORLD" med nøglen "KEY"
    - at dekryptere "RIJVS UYVJN" med nøglen "KEY"
    - at finde nøglen, hvis du ved, at "LXFOPVEFRNHR" dekrypteret skal være "ATTACKATDAWN"
1. Forklar, hvordan Vigenere-chifferet fungerer, og hvorfor det er mere sikkert end Caesar-chifferet.
2. Implementer en funktion i Python, der kan bryde en Vigenere-krypteret tekst ved hjælp af Friedman-testen for at estimere nøglens længde.
3. Implementer Kasiski-undersøgelsen i Python for at finde gentagne sekvenser i en krypteret tekst og beregne afstande mellem deres forekomster.

RSA-kryptering

RSA (Rivest-Shamir-Adleman) er en af de mest anvendte asymmetriske krypteringsmetoder i dag. Den bygger på matematiske principper inden for talteori, især brugen af store primtal og modulær aritmetik. RSA-kryptering involverer to nøgler: en offentlig nøgle, som kan deles med alle, og en privat nøgle, som holdes hemmelig.

Konceptet bag RSA er baseret på det faktum, at det er let at multiplicere to store primtal sammen, men det er meget svært at faktorisere produktet tilbage til de oprindelige primtal. Dette asymmetriske forhold gør RSA velegnet til sikker kommunikation.

Herunder er en forenklet oversigt over, hvordan RSA-kryptering fungerer:

1. **Nøglegenerering**:
   - Vælg to store primtal, p og q.
   - Beregn n = p * q. Værdien n bruges som modulus for både den offentlige og private nøgle.
   - Beregn φ(n) = (p-1)(q-1), hvor φ er Eulers totientfunktion.
   - Vælg et heltal e, sådan at 1 < e < φ(n) og e er coprime med φ(n). Værdien e bliver den offentlige eksponent.
   - Beregn d, den private eksponent, sådan at d * e ≡ 1 (mod φ(n)). Dette kan gøres ved hjælp af den udvidede Euklidiske algoritme.
2. **Kryptering**:
   - For at kryptere en besked m (hvor m er et heltal i intervallet 0 til n-1), beregn chifferteksten c ved hjælp af formlen: c ≡ m^e (mod n). Dvs. hæv m til potensen e og tag modulus n.
3. **Dekryptering**:
   - For at dekryptere chifferteksten c, beregn den oprindelige besked m ved hjælp af formlen: m ≡ c^d (mod n). Dvs. hæv c til potensen d og tag modulus n.

Dette er en forenklet version af RSA-kryptering, og i praksis anvendes meget større primtal for at sikre sikkerheden.

Herunder implementeres en simpel version af RSA-kryptering i Python fra bunden ved hjælp af indbyggede funktioner. Man skal være velkommen til at skippe denne del, da det kan være lidt avanceret:

```
import random
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
def mod_inverse(e, phi):
    d_old, d_new = 0, 1
    r_old, r_new = phi, e
    while r_new != 0:
        quotient = r_old // r_new
        r_old, r_new = r_new, r_old - quotient * r_new
        d_old, d_new = d_new, d_old - quotient * d_new
    if r_old > 1:
        raise Exception("e is not invertible")
    if d_old < 0:
        d_old += phi
    return d_old
def generate_keypair(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 3
    while gcd(e, phi) != 1:
        e += 2
    d = mod_inverse(e, phi)
    return ((e, n), (d, n))  # public and private keys
def encrypt(public_key, plaintext):
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in plaintext]
    return cipher
def decrypt(private_key, ciphertext):
    d, n = private_key
    plain = [chr(pow(char, d, n)) for char in ciphertext]
    return ''.join(plain)
# Eksempel på brug
p = 61  # et primtal
q = 53  # et andet primtal
public_key, private_key = generate_keypair(p, q)
```

```
message = "HELLO"
ciphertext = encrypt(public_key, message)
print("Krypteret besked:", ciphertext)
decrypted_message = decrypt(private_key, ciphertext)
print("Dekrypteret besked:", decrypted_message)
```

Herunder er et eksempel på, hvordan RSA-kryptering kan implementeres i Python ved hjælp af biblioteket `cryptography`:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes

# Generer private og offentlige nøgler
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)

public_key = private_key.public_key()
# Krypter en besked
message = b"Dette er en hemmelig besked."
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
# Dekrypter beskeden
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
print("Krypteret besked:", ciphertext)
print("Dekrypteret besked:", plaintext)
```

I dette eksempel genererer vi et par nøgler (en privat og en offentlig nøgle), krypterer en besked ved hjælp af den offentlige nøgle og dekrypterer den igen med den private nøgle. Vi bruger OAEP-padding og SHA-256 hashing for at sikre krypteringen.

Opgaver til RSA-kryptering

Herunder nogle opgaver, du kan prøve at løse for at øve dig på RSA-kryptering:

0. Forklar i dine egne ord, hvordan asymmetrisk kryptering adskiller sig fra symmetrisk kryptering. Og vis hvordan RSA kan bruges til at sikre kommunikation mellem to parter. Tag udgangspunkt i eksemplet ovenfor.
1. Forklar, hvordan RSA-kryptering fungerer, og hvorfor det er mere sikkert end symmetriske krypteringsmetoder.
2. En vigtig del af RSA er nøglegenerering. Implementer en funktion i Python, der afgør om et tal er et primtal.
3. På baggrund af 2. implementer en funktion, der genererer to store primtal og bruger dem til at skabe et RSA-nøglepar.
4. Implementer en simpel RSA-krypterings- og dekrypteringsfunktion i Python uden brug af eksterne biblioteker.