Innføring til Python #3 Sammensatte eksempler

I denne notebooken går vi igjennom noen mer sammensatte Python eksempler. Målet er å begynne å få mer forståelse for hvordan vi kan bruke Python til å løse mer komplekse problemstillinger.

Programstruktur og lesbarhet

Før vi går videre på eksemplene kan vi snakke litt om programstruktur og lesbarhet. En ting som er viktig å huske på når man koder, er at man bør strukturere koden så den er så ryddig så mulig. For det første gjør man mindre slurvefeil når man har en ryddig kode, men det å holde en ryddig struktur gjør også at man tenker bedre på problemet. Kokker har begrepet *a cluttered counter gives a cluttered mind*, og det samme er sant for rotete kode.

Samtidig bør koden vi skriver være forståelig for andre som skal lese den. Kode er jo egentlig skrevet for at en datamaskin skal forstå den, men andre skal gjerne bruke, lese eller endre på den senere. Skriv derfor kode med baktanken om at den ikke bare skal kunne kjøres av datamaskinen, men at den skal kunne leses av andre. Selv om du er sikker på at du aldri skal dele koden din med noen, vil det være mye lettere for degselv å gå tilbake til koden din senere og gjøre endringer eller videreutvikle den senere, hvis koden din er ryddig i utgangspunktet. La oss se på et par viktige triks for å gjøre koden ryddig og leselig.

Kommentarer

Det første du kan gjøre, er å skrive kommentarer i koden. Kommentarer er deler av programmet ditt som Python ikke tolker som kode, og som ikke påvirker programmet ditt på noen måte. Det eneste kommentarer gjør, er å forklare hva som skjer til de som leser koden din. Du skriver kommentarer med `#` -symbolet. Alt som står bak en `#` på en kode-linje tolkes som en kommentar, og vil ikke endre noe på programmet ditt. Du kan anta at de som leser koden din kan grunnleggende Python, kommentarer bør derfor fokusere mer på baktanken ved koden. Et vanlig sted å ha kommentarer, er i begynnelsen av funksjoner, for å forklare funksjonen gjør

In [1]:

```
def C2F(C):
    # Ta en temperatur oppgitt i grader Celsius, returner temperaturen oppgitt
    return 9/5*C + 32
```

Variabelnavn

Et annet viktig verktøy for å gjøre programmer leselige er å velge gode variabelnavn. Vi kan velge variabelnavn helt fritt, så det gjelder noe som er beskrivende for det variabelen faktisk brukes til, dette gjelder også funksjonsnavn. I eksempelet over heter funksjonen vår `C2F`, som leses ut "C to F", altså "celsius til fahrenheit". Dette kan være et bedre variabelnavn enn for eksempel `temperaturkonvertering` eller lignende, for isåfall mister vi informasjon på hvilken retningen konverteringen skjer. Her kunne vi kanskje valgt et enda bedre navn med `celsius2fahrenheit`, men samtidig blir det kanskje litt langt. En ting er sikkert, alle disse forslagene er bedre enn om vi kaller funksjonen vår `f`, eller `funksjon` eller noe slikt. Slike helt generelle variabelnavn gjør det veldig vanskelig for noen andre å lese koden.

Whitespace

En annen ting som er viktig for strukturen på et program, er det som kalles whitespace. Whitespace er ganske enkelt forklart alt vi ikke kan se, så det betyr mellomrom, tabs og tomme linjer. I Python går det ikke ann å legge ekstra tomrom på starten av linjer, for innrykket er viktig for fuksjoner og løkker - men ellers står vi ganske fritt til å legge inn whitespace der vi ønsker. Et program kan fort bli veldig mye finere og lettere å lese om vi legger inn litt ekstra tomrom så koden får "puste" litt.

I kode er det noen steder det går fint å legge inn ekstra mellomrom, og andre steder det ikke går. Hvis du lærer deg hvor det er greit å bruke litt ekstra 'tomrom', så kan programmet ditt fort bli veldig mye finere. Du kan for eksempel alltid legge inn blanke linjer i et program. Python bryr seg ikke om blanke linjer. Ved å lage litt pusterom mellom forskjellige deler av et program, så blir det for mye mer leselig. Her er tomme linjer spesielt viktig, for de kan vi bruke til å del opp bolker med kodelinjer som logisk hører sammen.

Lesbarhet i Jupyter

I Jupyter er det litt andre prinsipper som råder over hva som er leselig sammenlignet med kode skrevet i en tradisjonell teksteditor. Med tanke på at vi kan dele opp koden over mange celler og ha bekrivende tekst før og/eller etter kodesnutter gjør at vi kan lage programmer som er leselige uten å måtte bruke mye kommentarer og lignende.

In [2]:

```
1 from pylab import *
```

Primtall

Det først eksempelet vi skal se på handler om å finne primtall. Primtall er som kjent tall som kun kan deles perfekt på seg selv og på 1. Hva betyr det at noe kan deles perfekt? Det betyr at vi ikke får noen *rest* når vi deler. Dette er tydelig når vi skriver det ut som brøk:

$$\frac{7}{5} = 1 + \frac{2}{5}.$$

Her ser vi at 7 går opp i 5 én gang, og vi får en rest på 2, dette er fordi 7 kan skrives som $5 + 2$. I Python kan vi finne disse to talene ved å bruke de to matematiske operasjonene heltallsdivisjon `//` og modulus `%`. Heltallsdivisjon gir antall ganger ett tall går opp i et annet, og det vil alltid være et helt tall. Modulus gir resten, det som er igjen.

In [3]:

```
1 print(7//5)
2 print(7%5)
```

```
1
2
```

Vi kan sjekke om et tall er primtall ved å se på resten etter å ha delt på alle tall fra og med 2, opp til (men ikke med) tallet selv. Om det er en rest på 0 er det perfekt delbart, og dermed er tallet *ikke* ett primtall. Om løkka blir ferdig og vi ikke har funnet noen kandidater, så må tallet følgelig være et primtall. Funksjonen vår vil ikke fungere som den skal om tallet vi sjekker er 1 eller 2, for da vil ikke løkka kunne kjøre (det finnes ingen tall fra og med 2 til (men ikke med) 2. For disse bruker vi en if-test først.

In [4]:

```python
def er_primtall(n):
    if n == 1:
        return False

    if n == 2:
        return True

    # Løkk over alle muligheter
    for i in range(2, n):
        # Sjekk om n er perfekt delbart med tallet
        if n % i == 0:
            return False

    # Fant ingen tall, ergo er n et primtall
    return True
```

La oss teste funksjonen vår

In [5]:

```python
print("Skal være primtall:")
print("==========================")
print(er_primtall(2))
print(er_primtall(3))
print(er_primtall(5))
print(er_primtall(7))
print(er_primtall(11))

print("\nSkal ikke være primtall:")
print("==========================")
print(er_primtall(1))
print(er_primtall(4))
print(er_primtall(8))
print(er_primtall(9))
print(er_primtall(10))
```

```
Skal være primtall:
==========================
True
True
True
True
True

Skal ikke være primtall:
==========================
False
False
False
False
False
```

Vi ser at funksjonen vår fungerer som forventet. La oss bruke funksjonen vår til å finne alle primtall mindre enn 100.

In [6]:

```
for tall in range(1, 101):
    if er_primtall(tall):
        print(tall)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

Siden programmet vårt sjekker hvert eneste primtall vil det gå ganske tregt om vi skal for eksempel finne hvert primtall under én million, det vil nok ta minst et par minutter. Om vi skal gå enda lenger opp begynner det å bli umulig. Vi sier at løsningen vår *skalerer* ikke.

Det er et par måter vi kan forbedre `er_primtall` funksjonen vår på. For eksempel vet vi jo at partall aldri er primtall (med unntak av 2), siden partall per definisjon er perfekt delbare på 2. Så vi trenger ikke å prøve *alle* tall, vi trenger bare å prøve oddetallene. Vi sjekker først om `n` er et partall, i hvilket tilfelle er den ikke et primtall. Om `n` er et oddetall trenger vi bare prøve oddetall i løkka: `for i in range(3, n, 2)`.

In [7]:

```python
def er_primtall(n):
    if n == 1:
        return False

    if n == 2:
        return True

    # Sjekk om n er et partall
    if n % 2 == 0:
        return False

    # Løkk over alle muligheter (bare oddetall)
    for i in range(3, n, 2):
        # Sjekk om n er perfekt delbart med tallet
        if n % i == 0:
            return False

    # Fant ingen tall, ergo er n et primtall
    return True
```

En enda bedre forenkling vi kan se er at vi egentlig ikke trenger å sjekke tall opp til n, men til $\sqrt{n}$. Hvis du ikke ser hvorfor kan du ta en titt på denne wikipedia artikkelen (https://en.wikipedia.org/wiki/Primality_test) som viser hvorfor. Artikkelen går igjennom er par til forenklinger og ender opp med funksonen vi viser under. Målet er ikke at dere skal skjønne denne, men se hvordan man kan gå i steg fra en enkel idé til en bedre og raskere én.

In [8]:

```python
def er_primtall(n):
    if n == 1:
       return False
    elif n <= 3:
       return True

    if n % 2 == 0 or n % 3 == 0:
        return False

    i = 5
    while i**2 <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True
```

Primtallssil

En helt annen måte å finne primtall på, som ikke baserer seg på å faktisk sjekke delbarhet er en *primtallssil*, også kalt Eratosthenes' sil (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) etter den gamle greske matematikeren som fant den opp. Det er en veldig elegant algoritme for å finne primtall. Gå inn på lenken for å lese detaljer.

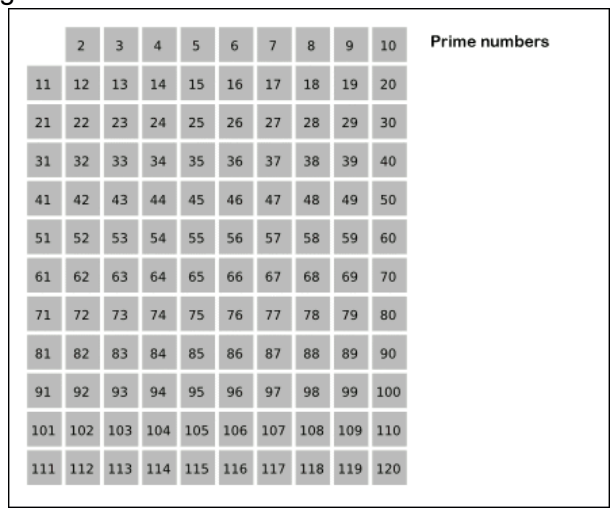
Primtallssilen fungerer ved at vi tar alle tall opp til et eller annet stort tall n, la oss si en million. Så siler den ut alle primtallene fra denne mengden med tall. Måten den gjør det på er som følger:

1. Vi starter med å anta at alle tallene i lista er primtall
2. Vi begynner så å jobbe oss oppover i tallene fra og med 2 (fordi 1 er litt sær, og ikke et primtall)
3. For hvert tall vi møter på som vi fortsatt mener er et primtall, så markerer vi alle *multipler* av det tallet som *ikke* primtall.
4. Vi hopper over alle tall i lista som er markert som ikke primtall.

La oss se på et eksempel der vi skal finne alle primtall opp til og med 10 med en primtallssil. Da går det som følger:

- Vi starer på 2, dette er et primtall. Da markerer vi resten av 2-gangen som *ikke primtal* (det vil si 4, 6, 8, 10)
- Vi går så til 3, denne er fortsatt regnet som et primtall, fordi den ikke er i 2-gangen. Derfor marker vi alle tall oppover i 3-gangen som ikke primtall (dvs 6 og 9).
- Vi går til 4, men denne er markert som ikke primtall fordi den er i 2-gangen.
- Vi går til 5, dette er et primtall, så vi markerer alle tall oppover i 5-gangen som ikke primtall.
- Vi hopper over 6 fordi den er i 2-gangen (og 3-gangen)
- Vi går til 7, dette er et primtall
- Vi hopper over 8, 9 og 10 siden disse er markert som ikke primtall.

Denne prosessen er lett å gjøre med penn og papir når det er få antall tall. Se denne visualiseringen fra Wikipedia som til og med er fargekodet:



Kilde: Wikipedia (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)

Lage en primtallssil i Python

Det er forholdsvis enkelt å lage en primtallssil i Python. Vi starter med å lage en lang liste med N elementer. Hvert element i lista svarer til et tall mellom 1 og N. Vi lar hvert element i lista være `True` for å indikere at vi tror det er et primtall.

Lista kan lages som følger (Vi legger til ett ekstra element, slik at vi kan bruke indeksen til tallet til å bety tallet selv, dvs, `primtallssil[4]` betyr tallet 4.

In [9]:

```
N = 10
primtallssil = [True]*(N+1)

# Vi vet at 0 og 1 ikke er primtall
primtallssil[0] = False
```

Så kan vi løkke over lista. For hvert element sjekker vi om verdien er `True`, i hvilket tilfelle tallet er et primtall,

eller `False`, i hvilket tilfelle det ikke er det.

In [10]:

```
primtall = []
for i in range(2, N+1):
    if primtallssil[i] == True:
        # Vi har funnet et primtall!
        primtall.append(i)
        # Vi må nå utlekke alle multipler
        k = 2
        while k*i <= N:
            primtallssil[k*i] = False
            k += 1

print(primtallssil)
print(primtall)
```

```
[False, True, True, True, False, True, False, True, False, False, Fals
e]
[2, 3, 5, 7]
```

Vi ser at primtallssilen fungerer fint for eksempelet vårt med $N = 10$. La oss prøve med et større tall.

In [11]:

```
N = 1000000
primtallssil = [True]*(N+1)

# Vi vet at 0 og 1 ikke er primtall
primtallssil[0] = False

primtall = []
for i in range(2, N+1):
    if primtallssil[i] == True:
        # Vi har funnet et primtall!
        primtall.append(i)
        # Vi må nå utlekke alle multipler
        k = 2
        while k*i <= N:
            primtallssil[k*i] = False
            k += 1

# Print ut de første 100 primtallene
print(primtall[:100])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 6
7, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 13
9, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 22
3, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 29
3, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 38
3, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 46
3, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
```

Vi ser at primtallssilen er superrask, mye raskere en delemetoden vi brukte først.

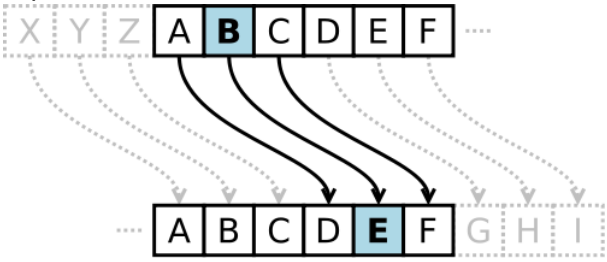
Hemmelige beskjeder

Det andre eksempelet vi ser på handler om kryptografi, som er kunsten av å lage hemmelige beskjeder. Dette eksempelet er delvis basert på et opplegg fra KidsaKoder (http://oppgaver.kidsakoder.no/python/hemmelige_koder/hemmelige_koder.html).

Når vi skal sende en hemmelig beskjed må vi først ta beskjeden vi ønsker å sende, å gjøre den uleselig for andre. Det å gjøre beskjeden uleselig kaller vi å *kryptere* den. Når beskjeden kommer frem til mottakeren, må de gjøre den leselig igjen, dette kalles å *dekryptere*. Målet er at det er kun den som beskjeden *egentlig* er ment for som skal kunne dekryptere den, og dette gjør man ved at sender og mottaker har blitt enig om en *hemmelig nøkkel* på forhånd. Kun den som har nøkkelen kan dekryptere meldingen.

Cæsarschiffer

En av de enkleste måtene å lage en hemmelig beskjed på som kan dekrypteres med en hemmelig nøkkel er å flytte alle bokstavene et gitt antall plasser fremover eller bakover i alfabetet



La oss implementere et Cæsarschiffer. Vi har lyst til å lage en funksjon som krypterer meldingen, og en som dekrypterer. Men før vi kan få til det må vi forstå hvordan vi kan flytte en bokstav fremover eller bakover i alfabetet.

Å flytte én bokstav

Det vi ønsker å gjøre er at vi kan si at for eksempel $B + 3 = F$, men dette går ikke i Python, for vi kan ikke legge sammen tekst og tall. Så det vi isteden vil gjøre, er å lage en funksjon, slik at `flytt_bokstav("B", 3)` gir F tilbake.

For å gjøre dette trenger vi at hver bokstav i alfabetet hører til et tall.

In [12]:

```
alfabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZÆØÅ'
print(len(alfabet))
```

```
29
```

Tallet til hver bokstav er nå hvor i alfabetet den ligger. Vi kan finne dette talled med `find` -funksjonen

In [13]:

```
print(alfabet.find('A'))
print(alfabet.find('F'))
```

```
0
5
```

Vi kan altså bruke `alfabet.find` til å finne posisjonen til en hvilket som helst bokstav. Merk at vi begynner å telle på 0. Tilsvarende, om vi har en posisjon og ønsker å finne hvilken bokstav det svarer til kan vi bruke vanlig indeksering

In [14]:

```
print(alfabet[8])
print(alfabet[9])
print(alfabet[10])
```

```
I
J
K
```

Vi kan nå flytte en bokstav ved å først gjøre den om til en posisjon, så legge til forflytningen til posisjonen, så konverte tilbake til en bokstav. La oss prøve

In [15]:

```
def flytt_bokstav(bokstav, antall_steg):
    """Flytt en bokstav n hakk bakover i alfabetet."""
    # Konverter bokstav til posisjon i alfabetet
    posisjon = alfabet.find(bokstav)

    # Legg til forflytningen for å finne den nye posisjonen
    ny_posisjon = posisjon + antall_steg
    # Konverter den nye posisjonen til tilhørende bokstav
    ny_bokstav = alfabet[ny_posisjon]

    # Send den nye bokstaven tilbake
    return ny_bokstav
```

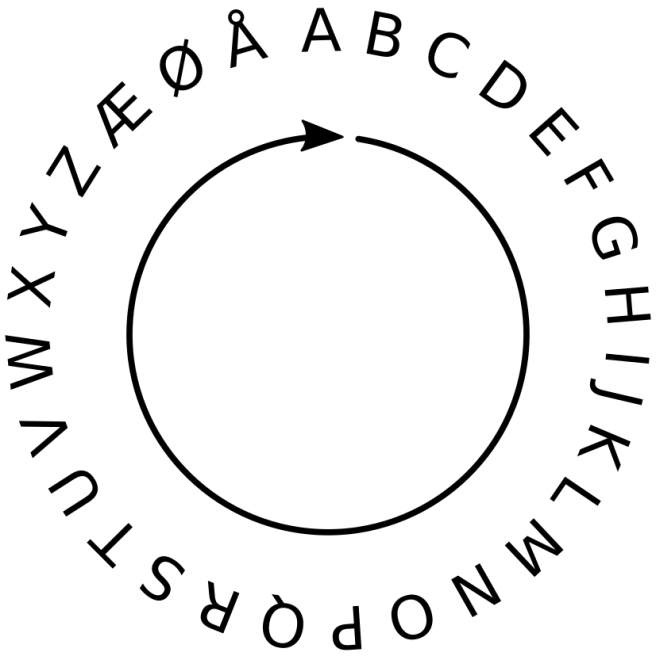
In [16]:

```
flytt_bokstav("B", 3)
```

Out[16]:

```
'E'
```

Flott, vi ser at `flytt_bokstaven` funksjonen fungerer som forventet. Men hva skjer dersom vi er på endene av alfabetet? Si vi er på 'Å' og vil flytte et hakk lenger ned i alfabetet, eller at vi er på 'A' og vil flytte ett hakk oppover i alfabetet. Det vi *ønsker* at skal skje, er at vi løkker rundt til andre siden. Så hvis vi er på 'Å' og skal gå tre plasser lenger ned i alfabetet ender vi på 'C'. Vi kan illustrere dette ved å tegne opp alfabetet i en sirkel



Så når vi kommer til "Å" og ønsker å gå videre bare starter vi på "A" igjen, og tilsvarende motsatt vei.

Å gå mot klokka i denne figuren tilsvarer å bruke en negativ n i funksjonen vår. Og dette fungerer faktisk helt fint "rett ut av boksen", selv om vi er på "kanten".

In [17]:

```
1 print(flytt_bokstav("F", -3))
2 print(flytt_bokstav("A", -1))
```

```
C
Å
```

For 'A' til 'Å' overgangen fungerer dette fordi den nye posisjonen blir $0 - 1 = -1$, og å indeksere med negative tall betyr at man teller fra høyre. Helt til høyre i lista vår ligger jo 'Å', så da treffer vi helt riktig. Å gå med klokka på den andre "kanten" derimot, er verre, la oss prøve:

In [18]:

```
1 flytt_bokstav("Å", 1)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
 last)
<ipython-input-18-2ded50fd15d5> in <module>()
----> 1 flytt_bokstav("Å", 1)

<ipython-input-15-c197cb6875c9> in flytt_bokstav(bokstav, antall_steg)
      7     ny_posisjon = posisjon + antall_steg
      8     # Konverter den nye posisjonen til tilhørende bokstav
----> 9     ny_bokstav = alfabet[ny_posisjon]
     10
     11     # Send den nye bokstaven tilbake

IndexError: string index out of range
```

Her får vi en `IndexError` fordi vi bruker en indeks som er for stor. Posisjonen til Å er 28 (siden vi begynner å telle på 0), så når vi legger til 1 får vi 29. Men strengen vår har bare indekser fra 0 til og med 28, det finnes ingen `alfabet[29]`, så programmet vårt kræsjer.

Det vi vil her, er at dersom den nye posisjonen blir 29 (eller større), så trekker vi fra 29. På denne måten blir 29 til 0, slik at 'Å' blir til 'A'. Om vi går to steg "over grensen" får vi $28 + 2 = 30$, trekker vi fra 29 blir det 1, som er 'B', perfekt!

Det finnes to måter vi kan gjøre dette på nå. Den mest intuitive er kanskje å bruke en if-test:

```
if ny_posisjon >= 29:
    ny_posisjon -= 29
```

Dette fungerer fint og er lett å kode. Men det finnes en enda enklere måte som er litt mer matematisk, vi kan bruke *modulus* operatoren

```
ny_posisjon = (posisjon + antall_steg) % 29
```

Her er `%` modulus, som gir resten etter en divisjon. Så når vi skriver `% 29` vil vi trekke fra 29 så mange ganger som mulig. Om den nye posisjonen for eksempel blir 33 så blir 33, fordi etter vi trekker fra 29 én gang sitter vi igjen med 4.

In [19]:

```python
def flytt_bokstav(bokstav, antall_steg):
    """Flytt en bokstav n hakk bakover i alfabetet."""
    # Konverter bokstav til posisjon i alfabetet
    posisjon = alfabet.find(bokstav)

    # Legg til forflytningen for å finne den nye posisjonen
    ny_posisjon = (posisjon + antall_steg) % 29
    # Konverter den nye posisjonen til tilhørende bokstav
    ny_bokstav = alfabet[ny_posisjon]

    # Send den nye bokstaven tilbake
    return ny_bokstav
```

Kryptere en hel beskjed

Vi er nå klare til å kryptere en hel beskjed. Vi lager først en helt tom tekststreng som skal være den nye, krypterte beskjeden vår. Deretter løkker vi igjennom den originale beskjeden bokstav for bokstav. For hver bokstav flytter vi den n hakk, bokstaven vi ender opp på legger vi til i den nye tekststrengen vår. Alle bokstaver flyttes like mange hakk, og det er dette som er *nøkkelen* vår. Tekststrenger har ikke en `append` -metode på samme måte som lister har, så isteden bruker vi bare `+=` for å legge til bokstaver.

Til slutt kan det være det er mellomrom og andre spesielle tegn i beskjeden vår, og disse klarer vi ikke å flytte på samme måte, fordi de er jo ikke i alfabetet til å begynne med. Disse tegnene lar vi bare være som de er. Vi kan sjekke om et tegn er en vanlig bokstav eller utenfor alfabetet med en if-test: `if bokstav in alfabet:` .

Måten vi gjør dette på er å løkke igjennom beskjeden, og flytte hver bokstav like mange hakk. Vi lager først en

Mellomrom og andre spesielle tegn i beskjeden vår klarer vi ikke å flytte, for de er ikke i alfabetet - så vi lar de tegnene være som de er.

In [20]:

```python
def krypter(beskjed, nøkkel):
    """Ta en beskjed og en nøkkel og krypter den med et cæsarschiffer."""

    # Starter med en tom beskjed
    kryptert_beskjed = ''

    # Løkk over den originale beskjeden
    for bokstav in beskjed.upper():
        # Sjekk om bokstaven er vanlig eller spesiell
        if bokstav in alfabet:
            test = flytt_bokstav(bokstav, nøkkel)
            kryptert_beskjed += flytt_bokstav(bokstav, nøkkel)
        else:
            kryptert_beskjed += bokstav

    return kryptert_beskjed
```

La oss prøve på en hemmelig beskjed

In [21]:

```
beskjed = "VI ANGRIPER IMORGEN KLOKKEN ÅTTE-FEMTEN"
print(krypter(beskjed, 17))
```

```
JZ RBXFZDVF ZACFXVB ØÅCØØVB QHHV-WVAHVB
```

Merk at vi skriver beskjeden i store bokstaver, ettersom at vi lagde alfabetet med store bokstaver. Det beste hadde vært å bruke `beskjed.upper()` i funksjonen vår, slik at selv om brukeren skrev små bokstaver ble de tolket som store.

Dekryptere en beskjed

Å dekryptere en beskjed er egentlig helt likt som å kryptere den, vi må bare "kryptere" den på nytt, men med motsatt antall steg. Så om nøkkelen er 5, så må vi flytte alle bokstavene -5. Istedenfor å lage en helt ny funksjon, så gjør vi nå et smart triks, vi "wrapper" `krypter` funksjonen vår

In [22]:

```
def dekrypter(beskjed, nøkkel):
    return krypter(beskjed, -nøkkel)
```

På denne sparer vi masse arbeid. Det er også veldig lurt at dersom vi finner en feil og endrer på `krypter`, så blir `dekrypter` automatisk fikset også. La oss se om det funker.

In [23]:

```
print(beskjed)
kryptert_beskjed = krypter(beskjed, 7)
print(kryptert_beskjed)
dekryptert_beskjed = dekrypter(kryptert_beskjed, 7)
print(dekryptert_beskjed)
```

```
VI ANGRIPER IMORGEN KLOKKEN ÅTTE-FEMTEN
ÅP HUNYPWLY PTVYNLU RSVRRLU GÆÆL-MLTÆLU
VI ANGRIPER IMORGEN KLOKKEN ÅTTE-FEMTEN
```

Så lenge begge parter er enige om hvilken nøkkel de skal kryptere beskjeden med, og hvordan de skal gjøre det, så kan de nå sende hemmelige beskjeder.

Forbedre på Cæsarschifferet

Det er to hovedproblem med metoden vi nettopp har brukt. Det ene er at spesielle tegn består, og man kan skjønne en del av beskjeden basert på mellomrom, tall, osv. Dette var grunnen til at vi skrev ut "åtte-femten", istedenfor å skrive 8:15 i beskjeden. For å unngå dette kan vi kreve at alle beskjeder vi sener består av *kun* bokstaver. Dette er det som gjøres i praksis, man dropper mellomrom, utropstegn og alt annet, så lar man den som mottar beskjeden tolke seg frem til dette. La oss lag en funksjon som tar en vanlig beskjed og fjerner alt unntat de vanlige bokstavene

In [24]:

```
def kun_bokstaver(beskjed):
    ny_beskjed = ""
    for tegn in beskjed.upper():
        if tegn in alfabet:
            ny_beskjed += tegn
    return ny_beskjed

kun_bokstaver("Vi angriper imorgen åtte-femten")
```

Out[24]:

```
'VIANGRIPERIMORGENÅTTEFEMTEN'
```

Det andre problemet vi har er mye verre. Og det er at det finnes egentlig bare like mange mulige nøkler som det er antall bokstaver i alfabetet, det vil si 29. Dette er fordi alle større tall enn 29 betyr bare at vi tar en hel runde først, og så litt, og det tjener vi ingenting på. Det gjør at noen som vil *knekke koden* vår kan bare prøve alle nøklene. Til slutt finner de noe som ser ut som en beskjed. La oss prøve: Vi lar datamaskinen treffe en tilfeldig nøkkel og kryptere beskjeden uten at vi får vite nøkkelen, så prøver vi alle nøklene etterhverandre, klarer du å finne beskjeden?

In [25]:

```
beskjed = kun_bokstaver("Vi angriper imorgen åtte-femten")
kryptert_beskjed = krypter(beskjed, randint(0, 29))

for gjett in range(0, 29):
    print(gjett, dekrypter(kryptert_beskjed, gjett))
```

```
0 MÅUEÆIÅGYIÅDFIÆYETKKYZYDKYE
1 LØTDZHØFXHØCEHZXDSJJXYXCJXD
2 KÆSCYGÆEWGÆBDGYWCRIIWXWBIWC
3 JZRBXFZDVFZACFXVBQHHVWVAHVB
4 IYQAWEYCUEYÅBEWUAPGGUVUÅGUA
5 HXPÅVDXBTDXØADVTÅOFFTUTØFTÅ
6 GWOØUCWASCWÆÅCUSØNEESTSÆESØ
7 FVNÆTBVÅRBVZØBTRÆMDDRSRZDRÆ
8 EUMZSAUØQAUYÆASQZLCCQRQYCQZ
9 DTLYRÅTÆPÅTXZÅRPYKBBPQPXBPY
10 CSKXQØSZOØSWYØQOXJAAOPOWAOX
11 BRJWPÆRYNÆRVXÆPNWIÅÅNONVÅNW
12 AQIVOZQXMZQUWZOMVHØØMNMUØMV
13 ÅPHUNYPWLYPTVYNLUGÆÆLMLTÆLU
14 ØOGTMXOVKXOSUXMKTFZZKLKSZKT
15 ÆNFSLWNUJWNRTWLJSEYYJKJRYJS
16 ZMERKVMTIVMQSVKIRDXXIJIQXIR
17 YLDQJULSHULPRUJHQCWWHIHPWHQ
18 XKCPITKRGTKOQTIGPBVVGHGOVGP
19 WJBOHSJQFSJNPSHFOAUUFGFNUFO
20 VIANGRIPERIMORGENÅTTEFEMTEN
21 UHÅMFQHODQHLNQFDMØSSDEDLSDM
22 TGØLEPGNCPGKMPECLÆRRCDCKRCL
23 SFÆKDOFMBOFJLODBKZQQBCBJQBK
24 REZJCNELANEIKNCAJYPPABAIPAJ
25 QDYIBMDKÅMDHJMBÅIXOOÅAÅHOÅI
26 PCXHALCJØLCGILAØHWNNØÅØGNØH
27 OBWGÅKBIÆKBFHKÅÆGVMMÆØÆFMÆG
28 NAVFØJAHZJAEGJØZFULLZÆZELZF
```

Det er kanskje litt slitsomt å knekke koden ved å sjekke alle muligheter her, men om alle beskjeder sendes med samme nøkkel så trenger vi bare å knekke koden én gang, så kan vi bare bruke den samme nøkkelen på alle fremtidige beskjeder.

Cæsarschiffer med passord

Nå skal vi gå vekk fra å ha ett enkelt tall som nøkkel og isteden se hvordan vi kan gjøre nesten det samme, men å bruke et passord isteden. La oss først velge oss et eksempelnøkkel: `HUBRO` (en hubro er en type ugle).

Det vi nå gjør er å oversette passordet vårt til en rekke tall, på samme måte som vi har funnet posisjonen tidligere. Tallene vi finner kaller vi for *keypadden* vår

In [26]:

```
passord = "HUBRO"
keypad = []

for bokstav in passord:
    keypad.append(alfabet.find(bokstav))

print(keypad)
```

```
[7, 20, 1, 17, 14]
```

Det vi gjør nå er å kryptere beskjeden vår, men for hver bokstav ser vi på det neste tallet i keypadden. Så i beskjeden vår flytter vi den første bokstaven 7 hakk, den neste bokstaven 20 hakk, den nest 1 hakk, osv. Men, her har vi kanskje et problem, for passordet vårt er jo mye kortere enn beskjeden. Det vi gjør for å fikse dette er at vi bare løkker igjennom passordet vårt når vi har brukt det opp, dette kan vi gjøre på samme måte som for alfabetet vårt, ved å bruke en if-test, eller ved å bruke modulus-operatoren (`%`).

In [27]:

```python
def krypter_med_passord(beskjed, passord):
    ny_beskjed = ""

    # Sørg for at beskjeden er klar for kryptering
    beskjed = kun_bokstaver(beskjed)

    # Lag Keypadden
    keypad = []
    for bokstav in passord:
        keypad.append(alfabet.find(bokstav))

    # Tellevariabel for keypadden
    teller = 0

    # Løkk over beskjeden og krypter
    for bokstav in beskjed:
        ny_beskjed += flytt_bokstav(bokstav, keypad[teller])
        teller += 1
        if teller >= len(keypad):
            teller -= len(keypad)

    return ny_beskjed
```

In [28]:

```python
print(beskjed)
kryptert_beskjed = krypter_med_passord(beskjed, "HUBRO")
print(kryptert_beskjed)
```

```
VIANGRIPERIMORGENÅTTEFEMTEN
ÅÅBBUYÅQVCPDPFULEAHELZFAELE
```

For denne metoden så er det ikke like lett å lage en "wrapper" for krypter funksjonen, så vi lager en tilsvarende funksjon for den andre veien

In [29]:

```python
def dekrypter_med_passord(beskjed, passord):
    ny_beskjed = ""

    # Sørg for at beskjeden er klar for kryptering
    beskjed = kun_bokstaver(beskjed)

    # Lag Keypadden
    keypad = []
    for bokstav in passord:
        keypad.append(alfabet.find(bokstav))

    # Tellevariabel for keypadden
    teller = 0

    # Løkk over beskjeden og krypter
    for bokstav in beskjed:
        ny_beskjed += flytt_bokstav(bokstav, -keypad[teller])
        teller += 1
        if teller >= len(keypad):
            teller -= len(keypad)

    return ny_beskjed
```

In [30]:

```python
print(dekrypter_med_passord(kryptert_beskjed, "HUBRO"))
```

```
VIANGRIPERIMORGENÅTTEFEMTEN
```

Mens om vi prøver feil passord får vi bare nonsense

In [31]:

```python
print(dekrypter_med_passord(kryptert_beskjed, "UGLE"))
```

```
IWTÆASRMBZEÅYÅJHNXZAUTXZNFW
```

Det var lett å gjette på nøkkelen i Cæsarschifferet, men å gjette et vilkårlig passord er mye vanskeligere. Jo lengre passordet er jo bedre. Det finnes måter man kan gå frem for å begynne å knekke en slik kode også, men de blir fort veldig tekniske.