Python

H. Fausk

06.03.2023 3. versjon

1 Sett i gang å kode

Her er en introduksjon til programeringsspråket Python. Det er fokus på syntaks og eksempler, de fleste av dem i matematikk.

Følgende lille program tar summen av første til fjerde potens av tallet 3, og skriver deretter ut resultatet

```
print(3+3*3+3**3+3**4)
```

Stjerne $*$ er gangetegnet og dobbel stjerne $**$ er tegnet for potens. Tall eller tekststrenger som plasseres innenfor parentesene i print() skrives ut. Hvis vi ønsker å regne ut de fire første potensene av et annet tall enn 3, så må fem av forekomstene av 3 erstattes. Hvis vi derimot benytter en variabel med verdien 3, så blir det enklere å skifte ut denne verdien. Programmet blir og mer oversiktlig. Her har vi modifisert programmet ved å benytte en variabel som kalles a, og samtidig skrevet inn forklarende kommentarer i koden.

```
a=3      #Vi gir verdien 3 til variabelen a
s=a+a*a+a**3+a**4    #s er satt lik summen av a+a*a+a^3+a^4
print(s)     #verdien til summen s skrives ut
```

Likhetstegn benyttes til å tilordne verdier til navn (variabler).

Skal vi finne summen av potenser opp til a^{10}, eller en enda høyere potens, blir det fort tungvint å skrive opp alle leddene. Vi lar heller programmet selv ta summen av leddene a**n fra n=1 til en oppgitt verdi.

```
''' summen til den geometriske rekken som er summen av grunntallet
opphoeyd i eksponenten 1, opp til antall ledd regnes ut.'''
grunntall = 3
antalledd = 4
summen = 0
for n in range(antalledd):  #For-loekke. Gjentar linjene med fire innrykk
# fra n=0 til n=antalledd -1, totalt  antalledd  ganger.
    summen = summen + grunntall**(n+1)   #Legger til ledd n+1 til summen.
print('summen er lik: ', s)     #verdien til summen s skrives ut
```

Helt i starten har vi lagt inn en liten beskrivelse av hva programmet gjør. Vi har valgt variabelnavn som er selvforklarende. Det hadde kanskje vært mer naturlig at summen

av potensleddene bare het sum, men sum er opptatt—det benyttes i Python. Vi har lagt inn en for-løkke som oppdaterer summen med et nytt ledd hver gang den gjentas. Den gjentas med n=0, n=1, n=2, helt opp til n=antalledd − 1. Merk at løkken gjentas antalledd antall ganger, hvor antalledd er et heltall. Løkken starter med 0, ikke 1. Det er grunnen til at vi legger til 1 i eksponenten.

For-løkken gjentar koden som står nedenfor seg og som er plassert med fire innrykk. Dette er en særegenhet med Python. I andre språk benyttes parenteser etc. for å avgrense hva som skal utføres i løkken. Hadde vi også rykket print(summen) inn fire innrykk, så ville utskriften av summen blit utført i hver gjentakelse av løkken. Delsummene hadde da blitt skrevet ut. Forsøk gjerne det! Vi har og lagt til litt "luft" mellom likhetstegnene for å øke lesbarheten. Vi kunne ha gitt en mer beskrivende tekst sammen med resultatet som skrives ut. For eksempel

```
print{'Summen av ', grunntall,'opphoeyd i eksponentene   1 opp til ',
antalledd,'er lik: ',summen)
```

Vi minner om formelen for summen av en endelig geometrisk rekke

$$a + a^2 + \cdots + a^n = \frac{a(a^n - 1)}{a - 1}$$

når $a \neq 1$ og lik n hvis $a = 1$.

Kode kan skrives inn, eller ferdig kode kan limes inn i en kompilator. Det enkleste er å benytte en av kompilatorene som ligger tilgjengelig på diverse nettsider. For eksempel, World wide web consortium har nettsider med lett tilgjengelige kurs i koding for ulike dataspråk. Blant annet Python. Etter hvert kan man laste ned Python til datamaskinen sammen med en editor hvor koden kan skrives inn og kjøres. For eksempel Spyder og Visual Studio Code Disse kan også lastes ned sammen med diverse pakker ved hjelp av Anaconda. Har du en maskin med Linux tilgjengelig, så anbefales det å installere Python på den.

I disse notatene har vi samlet litt informasjon om syntaksen i Python og noen eksempler på enkle programmer. Vi forsøker ikke å forklare de grunnleggende konseptene eller gi en introduksjon til dem som ikkje har noen forståelse for hvordan kode kan skrives. Er dette er deg og eksemplene vi gir fremstår som uforståelige. Da anbefaler eg at du kontakter noen som kan programere litt og spør dem om hjelp, eller sjekk noen undervisningvideoer. Undersøk gjerne ovenfor nevnte w3school.

2 Datatyper

Vi ser først på noen grunnleggende datatyper: Heltall (integer), desimaltall (float) og strenger (strings). I tillegg har vi Bolske variabler. De kan bare ha to verdier: sannhetsverdiene True (Sann) eller False (Falsk). Det er mange flere datatyper, og vi kan også lage våre egne datatyper.

Strenger er sekvenser av tegn. Tegnene er nummerert, hvor vi starter med 0. Strengnene kan bestå av tall, bokstaver eller andre tegn. De skrives innenfor apostrof eller anførselstegn (ikke dobbel apostrof). Vi kan sjekke typen ved funksjonen *type*().

```
print(type(3.0))  #et desimaltall
print(type(3))     #et heltall
print(type('en streng'))
print(type("Dette er og en streng. 3|  @$#^&@"))
print(type(True))  #Bolsk verdi
print(type(2+5j))  #Komplekst tall
```

Vi kan gjøre om mellom forskjellige typer der hvor det gir mening ved bruk av funksjonene int(), float() og str().

```
print(int(34.0))
print(float(671))
print(float(34+4))  #Skriver ut 38.0
print(str(123))

Print(float(True))  #Tilordner verdien 1.0
Print(int(True))    #Tilordner verdien 1
Print(int(False))   #Tilordner verdien 0
```

Her er eksempler hvor omgjøring av datatypene ikke gir mening og vi får feilmelding.

```
int(3+7j)
int('3+7')
float("34+4")
float("streng")
str(uten apostrof)
```

Oppgave 2.1. Undersøk typen til følgende objekt og forsøk å endre typene deres med funksjonene ovenfor.

$$\text{True},\ \text{TRUE},\ \text{true},\ 'true',\ 3.14,\ 44,\ "44",\ \text{tretti3},\ 50.000,\ '2+3=5',\ 2+3$$

3 Heltall og desimaltall

Hvis vi vet at tallene vi arbeider med er heltall, er det ingen grunn til å huske på at alle desimalene skal være lik 0. Vi bruker da typen heltall. Pyton regner eksakt med heltall.

Desimaltall eller float (flyt) lagres gjerne som et tall på standard form: et tall mellom 1 og 10 ganget med en potens av 10, eller som 0. Vi kan skrive tallet $3.563 \cdot 10^3 = 3563$ som $3.563e3$. Tallet 0.0003357 kan skrives som $3.357e-4$. Et heltall som skal oppfattes som et desimaltall kan etterfølges av .0 slik som 3.0. Dette er og måten Python viser oss at heltallet er av typen flyttall. Merk at vi benytter punktum (eng. point) som desimalskilletegn.

Vi legger sammen og trekker fra tall ved å benytte $+$ og $-$. Multiplikasjon skrives med symbolet $*$ og divisjon ved symbolet $/$. Potenopphøying skrives med $**$. (Hatt $\wedge$ er i Python reservert til XOR, eksklusiv eller, for binære tall.) Her er noen eksempler

```
print(2+3)
print(2-3)
print(2*3)
print(2/3)
print(6/2)
print((2+5)/(5-2))
print((1-2/3)*9)
```

Divisjon av heltall gir et resultat som er av typen flyttall. Dette skjer selv om kvotienten er et heltall. Kvotienten 6/2 resulterer i 3.0. Kvotienten av heltall rundet ned til største heltall er gitt ved $a//b$. Resultatet er av type heltall. Er derimot minst en av a og b av type desimaltall, er $a//b$ også av type desimaltall.

Vi kan runde av tall som følger.

```
A= 123456789.123456789
print(round(A,3))     #Runder av til 3 desimaler
print(round(A,5))     #Runder av til 5 desimaler
print(round(A))     #Runder av til naermeste heltall.
print(round(A,-3))     #Runder av til -3 desimaler (gir 123457000)
```

Ofte så ønsker vi bare at utskriften ikke skal ha med for mange gyldige siffer, for økt lesbarhet. Dette kan oppnås med bruk av formaterte strenger, kort f-strings.

```
a=12345.6789
print(f'skriver ut taller i klammeparentes inne i en streng. Her er a: {a}.
Her er 2 ganget med 7: {2*7}')
print(f'tre gyldige siffer: {a:.3}')
print(f'tre desimaler: {a:.3f}')
```

Vi kan regne ut restklasser av et tall modulo et annet tall. $m\%n$ gir tallet a mellom 0 og n (ekte mindre) slik at m er lik a pluss et heltallsmultiplum av n.

```
5%3
15%6
12.65%5
8.5%3.4
```

Oppgave 3.1. Sjekk at pyton følger regnekonvensjoner. For eksempel regn ut

```
2**3**2
2/3/4
2+3/5
5/2+3
5/(2+3)
```

Oppgave 3.2. Sjekk hvordan Python runder av tall. For eksempel 1.49 og 1.5 rundes av til henholdsvis 1 og 2.

Oppgave 3.3. Python regner med sirka 16 siffers nøyaktighet når den bruker desimaltall. Undersøk resultatet av å kjøre følgend kode:

```
a=12345678901234567890012345     #Type heltall
print(a)
d=a-4
print(d)

b=float(a)  #benytt b= a + 0.0 istedet. Blir c den samme?
print(b)
c=int(b)
print(c)
print(a-c)
```

Oppgave 3.4. Komplekse tall skrives i Python som $a+bj$, hvor $j^2 = -1$. Sjekk typen til tallet $2-3j$ og $2+0j$. Sjekk at $4+j$ og $4+j3$ ikke er gyldige som komplekse tall. Det komplekse tallet $4+j$ bør altså skrives som $4+1j$.

4 Strenger

Vi kan legge sammen strenger til en ny streng. Dette gjøres ved å la en streng følge rett etter en annen streng. Dette kalles å *konkatenere* strengene. Dette gjøres med tegnet pluss +. Mellomrom er en tom streng ' '.

Linjeskift, ved utskrift av strenger, kan legges inn som $\backslash n$. Fire inntrykk—TAB—får vi ved å skrive $\backslash t$. Apostrof kan skrives ved $\backslash'$.

```
p = "Python"    #En streng skrives innenfor hermetegn.
k = 'kurs'      #Eller innenfor apostrofer.
s = p + k   #Konkatenerer strenger, det vil si legger dem sammen.
print(s)
s = k + " i " + p
print(s)

print(m[3])   #plukker ut tegnet i posisjon 4 (Tilsvarnde for 0,1,2,3 etc.)
print(m[0])   #plukker ut det foerste tegnet.
print(m[:3])  #Skriver ut de tre foerste tegnene
print(m[3:])  #Fjerner de tre foerste tegnene og skriver opp resten
```

Vi kan ikke oppdatere et tegn eller deler av en streng. Forsøker vi å oppdatere tegnene får vi en feilmelding.

```
streng = "Uforanderlig"
streng[0] = " "
#streng = "Helt " + streng  #Dette fungerer. Oppdaterer verdien til variabelen.

l=len("123456789")   #lengden til strengen, dvs.  antall tegn.
print(l)
print(len("1 2 3 4 5 6 7 8 9"))    #Ett mellomrom telles som ett tegn.

print('hvordan skrive "hermetegn"?')
print("hvordan skrive 'hermetegn'?")
```

En streng kan kjøres som en Python kode ved bruk av funksjonen exec().

```
y = 0
print('y = 5**2')
print(y)
exec('y = 5**2')
print(y)
```

Oppgave 4.1. Lag et program som tar inn to positive heltall a og b skriver brøken a/b som en sum av et heltall og en ekte brøk. (En ekte brøk er på formen m/n hvor $0 \leq m < n$. (Utvid til at programmet kan ta inn to heltall, hvor b er ulik 0).

5 Input—output

Python skriver ut tekst og tall ved å benytte print(). Benytter vi print() flere ganger så kommer hver utskrift på en ny linje.

```
print(12, end=' ') #12 etterfoelges av et mellomrom og linjeskift droppes.
print(13) #Benyttes end='' kommer tallene rett etterhverandre.

s = 'En'
t = "streng"
print(s , end=' ')
print(t)
print(s + ' \n ' )      # \n  "newline" linjeskift
print('\t' +t)          # 't tab, innrykk med 4 mellomrom.
print(s, end=' test ')
print(s + '\t' + t)

print("Vi maa ha komma mellom print(),"), print("ellers faar vi feilmelding.")
```

Input tar inn en streng. Vi må selv gjøre om til heltall eller desimaltall.

```
i = input('Skriv noe: ')  #tar inn en streng
print(i)
n = input('skriv inn et heltall: ')
print(type(n)), print(n + '3')
n= int(n)
print(type(n + 3)), print(n + 3)
```

Her er et lite program som regner om fra grader til radianer

```
import numpy as np  #Henter en pakke som inneholder tallet pi.
g = float(input(Skriv en vinkel i grader))
r = g * np.pi / 180
print{('Vinkelen i radianer er lik: ' , r)
```

Oppgave 5.1. Lag et program som først spør om etternavn og deretter om fornavn, og som skriver hele navnet ut på én linje.

6 Funksjoner

En funksjon tar inn en eller flere variabler og gir ut et resultat. Vi kan og ha funksjoner som ikke tar inn noen variabler. Vi har allerede møtt flere funksjoner som er innebygd i python. Slik som print(), type(), int(), len(). Vi kan lage egne funksjoner. Her er en funksjon som regner ut kvadratet til et tall, minus tallet selv.

```
def f(x):   #Kolon avlutter navnet paa funksjonen.
#Variabelnavnet brukes bare internt i funksjonen.
    return x**2 -x    #Vi maa bruke 4 innrykk for innholdet i funksjonen
```

Funksjoner kan ta inn og gi ut flere verdier

```
def f(x,y):
    return (x/y, y/x)
#Kjoeres programmet kan de to verdiene lagres som
(a,b) = f(3,7)
print(a)
print(b)
```

Vi kan også angi standardverdier til variablene. De benyttes hvis vi ikke angir verdier til noen eller alle av inn-verdiene.

```
def Annengradsuttrykk(a=1,b=0,c=0):
    """Inn tre variabler, ut en streng med et grad 2 polynom i variabel x"""
    #Det er vanlig aa skrive en 'doc string' rett etter start av en funksjon,
    #med informasjon om funksjonen.
    #Dette er informasjon som vises naar 'help' benyttes for funksjonen.
    U=str(a)+'*x**2 + '+str(b)+'*x + '+str(c)
    return U
print(Annengradsuttrykk)
print(Annengradsuttrykk(3))
print(Annengradsuttrykk(3, ,7))
print(Annengradsuttrykk(3,7,-4))
```

Her er en funksjon som ikke gir noe tilbake.

```
a='Noe blir utfoert'
def skrivUt(t):
    """Denne funksjonen printer det vi gir den som input."""
    print(t)
skrivUt(a)
```

Fakultet til et naturlig tall n er lik produktet av alle naturlige tall mindre enn eller lik n. For eksempel er $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Vi kan regne ut fakultet av naturlige tall ved å lage en funksjon som kaller seg selv.

```
n = 2958  #Tallet vi skal regne ut fakultet av.
def fact(x):
    if x == 0:    # avslutter og returnerer 1 hvis n=0.
```

```
        return(1)
    a =  fact(x-1)*x   #regner ut n! som $ (n-1)! * n.
    return(a )
fac = fact(n)
s=str(fac)   #Gjoer tallet om til en streng, for aa telle antall siffer
print( 'n! = ' , n )
print('antall siffer', len(s))
```

7 Pakker

I Python kan vi laste inn pakker. Bruker vi en funksjon eller kommando fra pakken skriver vi det som pakkenavn.kommando. En pakke er en mappe som består av Python filer. De kan og være organisert i undermapper. Vi kan lage egne pakker med Python kode. Det er vanlig å forkorte navnet til pakker som benyttes mye. Forkortelsene er ofte standardisert. Her er noen pakker med en kort beskrivelse av dem.

```
import numpy as np
```

Numerical Python, numpy, er en pakke med matematiske funksjoner. For eksempel sqrt, kvadratroten. Tallet pi etc.

```
import numpy as np
s=np.pi   # Skriver ut tallet pi=3.14...
print(s)

t=np.sqrt(2)  # Kvadratroten til 2
print(t)
```

Merk at kvadratroten til 4 blir 2.0, et flyt tall. Dette har å gjøre med hvordan kvadratroten regens ut.

Avrundring av et desimaltall a opp til minste heltall større enn a er gitt ved np.ceil(a). Avrunding av et desimaltall a opp til minste heltall mindre enn a er gitt ved np.floor(a).

En fin pakke for å teste effektiviteten til ulike koder er time. Det er eksempler på bruk delkapitlet om for-løkker.

Random er en pakke som gir tilfeldige tall. For eksempel gir Random.random() ut et tilfeldig tall mellom 0 og 1.

8 For-løkker

For-løkker lar oss gjenta en prosedyre et gitt antall ganger. Dette kombineres med at vi gjennomløper hvert element i en liste, eventuelt hvert tegn i en streng.

```
s='En tekststreng.'
l=len(s)    #Antall tegn i strengen s.
```

```
#Foelgende loekke skriver opp hvert av tegnene i strengen s.
for x in range(l):    #x gjennomloeper 0,1, ... , l-1.
    print(s[x])
#Alternativt kan vi gjennomloepe alle tegnenene i strengen ved
for x in s:
    print(x)
```

Her er en kode hvor vi legger inn litt ventetid mellom hver gjennomgang i løkken.

```
import time
M=''
for x in n:
    print(M+x, end='')
    M=' '+M
    time.sleep(0.1)    # venter i 0.1 sekunder
```

Her er eksempler med tall.

```
#Foelgende loekke skriver opp restklassen til x når vi tar av
#heltallsmultipler av 6 til vi får et heltall mellom 0 og 5.
for x in range(20):
    print(x % 6)

#Her regner vi ut 2-er potenser a=2**i
for i in range(21):     #Den gjennomloeper
    a=2**i
    print(a)          # for i = 0 , 1, 2 til og med 21-1=20.
#Skift gjerne 2**i ut med i  ,  2*i ,  2*i +1  etc.
```

Flytter vi print(a) helt til venstre får vi bare skrevet opp $2^{20} = 1024^2 = 1048576$. Grunnen til det er at print(a) da ikke er en del av løkken, så vi skriver bare ut verdien som variabelen a har etter at løkken er avsluttet.

Strengoperasjonene kan implementeres ved bruk av for-løkken. For eksempel er lengden til en streng gitt som følger

```
def lengde(x):
    lengde=0
    for s in x:   #s brukes bare internt i definisjonen.
        lengde = lengde +1
    return lengde
f='En test streng'    #mellomrom er ogsaa tegn og telles med naar lengden gis.
print(lengde(f))
```

Her et program som fjerner alle mellomrom i en streng. Her bruker vi en if-setning, som omhandles mer senere.

```
s="starter med en tom streng som vi legger tegn til."
u=""
for x in s:
     if x !=" "    :  # !=  betyr ikke lik
         u=u+x
print(u)
```

Oppgave 8.1. For $n \geq 1$ så er n fakultet definert som $1 \cdot 2 \cdots n$. I tilleg så er $0! = 1$. Lag et program som benytter en for-løkke til å regne ut n fakultet for $n \geq 0$. Skriv opp svaret som horisontal liste på formen $0! = 0$ $1! = 1$ $2! = 2$ $3! = 6$ $4! = 24$ etc. Innfør gjerne ny linje for hvert tiende tall. Varier hvor mange fakulteter du regner ut og skriver dem opp. Skriv gjerne opp antall siffer til tallene. For eksempel 69! har 99 siffer, 816! har 2024 siffer og 3000! har 9131 siffer.

Oppgave 8.2. Lag et program som skrive ut de 100 første Fibonacci tallene. Det er tallene definert rekursivt som

$$F_{n+2} = F_{n+1} + F_n \qquad F_0 = 0 \quad F_1 = 1$$

for $n \geq 0$.

9 If-setning—Bolske verdier

Bolske verdier tar verdiene True (Sann) eller False (falsk). De er resultatet av å sjekke en relasjon mellom to tall for eksempel

$$< \quad > \quad <= \quad >= \quad == \quad !=$$

Her betyr == lik og ! = betyr ikke lik. Rekkefølgen <= er samme rekkefølge som vi uttaler ulikheten: mindre enn eller lik. Benytter vi =< får vi feilmelding.

Vi kan utføre logiske operasjoner med Bolske størrelser

OG B and C

ELLER B or C

IKKE not C

```
Bolsk1 = 2<3       #Bolske verdier er True eller False
print(Bolsk1)
Bolsk2 = 2>3
print(Bolsk2)
print("og:" , Bolsk1 and Bolsk2 )
print("eller:" , Bolsk1 or Bolsk2)
print("Negering:" , not Bolsk2)
```

Bolske størrelser lar oss gjøre valg basert på om verdien er sann eller falsk. Til dette benyttes if-setninger. Det er to muligheter i hver if-setning, men flere if-setninger kan kombineres til å håntere tilfeller hvor det må gjøres mange valg. Benytter vi fem if-setninger, kan de benyttes til å velge blant opptil $2^5 = 32$ forskjellige muligheter.

```
#Programmet forteller oss hva relasjonen mellom de to tallene er.
a=5   #Forsoek med forskjellige verdier for a og b.
b=5
if a<=b:
    print(f"a={a} er mindre enn eller lik {b}")
if a>b:
    print(f"a={a} er stoerre enn b={b}")
if a==b:
    print(f"a={a} er lik b={b}")

H= a//b  #Heltallsdelen av kvotienten.
R= a % b   #Gir heltallet R mellom 0 og b-1 slik at a = heltall * b  + (a % b)
if R == 0:  #skriver opp a/b som et heltall pluss en ekte brøk.
    print( f'{a}/{b} = {H}')
else:
    print( f'{a}/{b} = {H} + {R}/{b}')
```

Her er et program som erstatter gjentatte mellomrom med bare ett mellomrom.

```
#Fjerner overfloedige mellomrom i strenger.
s="Python    er   nyttig."
u=""
M=False
print(s)
for x in s:
    if x !=" ":
        u=u+x
        M=False
    else:
        if M==False:    #Gir True hvis de har samme sannhetsverdi.
            u=u+x
            M=True
print(u)
```

Oppgave 9.1. Sjekk sannhetsverdiene til følgende utsagn

```
print( 2 == 2.0)
print( 2 == 2.0 + 1e-16)
print( 2 == 2.0 + 1e-15)
print( 100 == 100 + 1e-15)
```

Oppgave 9.2. Lag et program som finner alle positive heltallsløsninger til likningen

$$x + y + xy = 209$$

(Prøv andre tall også som 670.) Ved å teste ut alle mulige tilfeller. Det er lett å se at positive heltallslsløsninger må oppfylle $x, y \leq 209$.

Finn alle heltallige løsningene til likningene. Disse likningene kan også, med litt algebra løses elegant. Forsøk gjerne å finne løsningene ved regning.

Oppgave 9.3. Fermat sitt lille resultat sier at for ethvert primtall p så er differansen

$$x^p - x$$

delelig med p for alle heltall x. Dette kan og formuleres som at $x^{(p-1)} - 1$ er delelig med p for alle x som er relativt primisk til p. Her er et program som sjekker at dette stemmer for et gitt primtall.

```
p = 7  #et primtall
def f(x)
return x**(p-1) - 1

for i in range(p)
    print (f(i) % p )
```

Modifiser programmet slik at det bare skriver ut eventuell x-verdi hvor resten vi får når vi deler $x^p - x$ med p er ulik 0. Skriv gjerne opp resten sammen med x-verdien.

10 3x+1 problemet

Collatz formodning sier at starter vi med et positivt heltall, deler det med 2 til vi får et odde tall, ganger oddetallet med 3 og legger til 1 og gjentar prosessen, så kommer vi etter et endelig antall steg til tallet 1. For eksempel

15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1

Når vi kommer til 1 så får vi deretter: 1 sendes til $3+1=4$, så 2 og deretter 1. Vi sier at vi har en sykel. For naturlige tall er den eneste kjente sykelen den som gir sekvensen $1, 4, 2, 1$.

Her er en funksjon som kaller seg selv og som skriver ut resultatet av å utføre den rekursive prosedyren. Den stopper når vi kommer til 1. Ingen moteksempler til Collatz formodning er kjent, så hvis det skulle finnes et tall hvor vi ikke kommer til 1, etter endelig mange steg, så må det være for et enormt stort tall.

```
def Collatz(x):
"""Vi deler med 2 til vi faar et oddetall.
Dette oddetallet ganges med 3 og 1 legges til.
Prosessen gjentas til vi kommer til 1."""
    print(x, end=' ')
    while  x%2 ==0:
        x = x//2
        print(x, end=' ')  #Fjern 4 innrykk hvis vi bare skal
        #skrive ut resultatet av aa dele x med stoerste potens av 2.
    if(x != 1):
        Collatz(3*x+1)
Collatz(15)
#Collatz(27)
#Collatz(154308811)
```

Oppgave 10.1. Vi kan også anvende algoritmen for negative heltall. Det viser seg at vi får sykler for $-1, -5$ og -17, Sjekk dette. Du bør da oppdatere programmet slik at det stopper opp om vi kommer til -1, -5 eller -17, i tillegg til 1.

11 While løkker

En while løkke kjører så lenge et kriterie er oppfylt. Det er nyttig å kunne avbryte løkken hvis et kriteriumet aldri blir er oppfylt. Til det benyttes break.

```
import math
import random
a=0
while a<10:
    print(a)
    a += 2  # Legger 2 til a helt til a blir 8
b=0
while 1<2:   #altid sant, saa loekken vil gjentas helt til den avbrytes
    b += 1
    print(b)
    if b>20:
        break  #avslutter while-løkken
```

Vi kan for eksempel avbryte etter 1000 løkker ved å starte med $s = 0$ og oppdatere s ved å legge til 1 for hver gang løkken kjører, og legge inn

```
\break{s>999}
```

til starten av løkken. (Sjekk at det dette stemmer ved å teste et eksempel.)

```
"""Vi skriver opp antall tilfeldige tall mellom 0 og 1 foer vi
kommer til et tall stoerre enn g."""
import random
g=0.99
c=0    #Tallet som tildeles tilfeldige verdier
t=0   #antall tall
while c<0.99:
    c = random.random()
    #print(c)
    t = t + 1
print(t)
```

Oppgave 11.1. Modifiser programmet slik at det gjentas flere ganger. For eksempel ved å benytte en for-løkke. Skriv opp verdiene og regn ut gjennomsnittet av dem.

12 Multiplikasjonsoppgaver

Vi lager et program hvor vi spør om hva produktet av to tall mellom 0 og 100 er lik. Det gis tilbakemelding om svarforsøket er for lite eller for stort. Det blir gitt 10 forsøk før

programmet avslutter ved å gi riktig svar. Denne avsluttingen av spørsmålene oppnås ved å legge inn en avslutting i en while løkke. Vi benytter pakken random til å generere tilfeldige tall i et intervall.

```
import random, math
r=random.random()  #velger tilfeldig tall mellom 0 og 1
r=math.floor(101*r)  #Tilfeldig heltall mellom 0 og 100.
s=random.random()  #velger tilfeldig tall mellom 0 og 1
s=math.floor(101*s)  #Tilfeldig heltall mellom 0 og 100.
#print(r), print(s)   #brukt til test
p=r*s
R= format(r)    # gjoer om til strenger
S= format(s)
#Lager tekst for spørsmålet.
sp= "Hva er produktet av " + R + " og " + S +"? "
# Her er noen alternativer:
#sp=format("Hva er " + format(s) +"x" + format(t)+ "= ")
# sp = f"Hva er produktet av {r} og {s}  ?"
#print(sp)
svar= int(input(sp))  #Spør etter svar.
a=10 #Teller ned fra a=10 slik at vi avslutter
#spørsmålene etter 10 forsøk.
while svar != p:  #Kjører løkken så lenge svaret er galt.
    a -= 1    #trekker 1 fra a
    if a==0:
        print("Dette var kanskje litt vanskelig, "+R+"x"+S+" =", p )
        break  #avbryter løkken.
#svar = input("Svaret er galt, forsøk igjen: ")  Vi gir en versjon med hint
    elif svar < p:
        svar = int(input("Svaret er for lite, forsøk igjen: "))
    elif svar > p:
        svar = int(input("Svaret er for stort, forsøk igjen: "))
if a>0:  #Skriver bare dette ut hvis løkken avsluttes
#ved at riktig svar er besvart.
    print("Det stemmer, bra!")
```

13 Time pakken

Dette er en pakke som lar oss foreta pauser i program, og som lar oss måle hvor lang tid det tar å kjøre en del av et program. Sistnevte kan være nyttig til å sjekke effektiviteten til forskjellige kandidater til et program.

```
"""Programmet tar en streng og skriver den ut tegn for tegn, med 0,2
sekunders pause mellom hvert tegn. Ordet skrives nedover og p\aa\ skraa."""

import time
s='En'
```

```
t="streng"
u=s+' '+t
#konkatenering av de to strengene. Legger til et mellomrom mellom dem.
M=' '  #Begge m\aa ter \aa\ skrive strenger p\aa\ kan benyttes...
for x in u:
    print(M+x)    #legger inn et mellomrom foran tegnet.
    #Legger vi til $end=''$ i print, s\aa\ skrives ordet ut horisontalt
    #og med oekend eavstand mellom tegnene.
    M='  '+M  #oeker mellomrommet som benyttes
    time.sleep(0.2)

import time
s=time.time()   #Oppgir tiden i sekunder i UNIX epoch.
print(s)
t=time.ctime(s)    #Skriver ut tiden med klokkeslett, dato og aar.
print('UNIX epoch starter: ', end=''), print( time.ctime(0))
print(t)
```

For å bare måle CPU-tid (tiden det tar å utføre berekningen) kan funksjonen time.process.time() benyttes.

```
import time
s=0
tik = time.process_time()
for i in range(1,int(1e6)):
    s = s + i     #Sum av naturlige tall opp til 10**6 - 1.
tak = time.process_time()
print('tid addisjon: ', tak-tik)   #Differansen i tid i enheten sekunder.
print(s)
s=1
tik = time.process_time()
for i in range(1,int(1e6)):
    s = s * (i**(1/i))    # produktet av n/te rot av n fra 1 til 10**6-1.
tak = time.process_time()
print('tid multiplikasjon: ', tak-tik)
print(s)
```

Resultatet varier mellom hver gang programmet kjøres. Det blir mindre variasjon i resultatene hvis bare time.time() benyttes. Eg får at multiplikasjonen bruker 2 til 3 ganger lengre tid. Dette til tross for at vi både tar n-te roten av n og multipliserer disse røttene sammen.

14 Primtall

Her er et program som sjekker om et naturlig tall er et heltall.

```
n=93757 """Et naturlig tall.  Vi skal sjekke om tallet er et
primtall eller ikke."""
```

```
flag=True  #En sanhetsverdi som oppdateres til falsk hvis n ikke
er et primtall.
M=1
for m in range(2, 1+ int(n**(0.5))):  #Vi behøver bare sjekke om n er
#delelig med tall mindre enn eller lik kvadratroten til n
    #print( m)  #Brukt under testing av koden
    if n % m == 0:     #Sjekker om m deler n
       flag = False    #Hvis n er delelig med et mindre tall endrer vi flagget til
       M=m             #Husker på hvilke tall som delte n.
       break  #Det er ikkje behov for å fortsette løkken så den avsluttes.

K=n//M  #Regner ut kvotienten.  Hvis n er et primtall gir dette bare n selv.
if flag == True:  #n har ingen divisorer, og er derfor et primtall.
    print(f'Det naturlige tallet {n} er et primtall')
else:
    print(f'Det naturlige tallet {n}={M}*{K} er ikke et primtall')
    #Hvis n ikke er et primtall skriver vi det opp som et produkt av to
    #faktorer hvor den ene faktoren er den minste primtallsdivisoren.
```

15 Divisjonsalgoritmen

Å finne største felles multiplum av to positive heltall kan gjøres effektivt. Det er ikke nødvendig å faktorisere tallene.

La tallene være $n_1 \geq n_2 \geq 1$. Vi finner da restklassen til det største tallet modulo det minst. Restklassen er et tall mindre enn det minste av de to. Vi gjentar nå prosedyren helt til vi ender opp med at det minste tallet deler det største. Det minste tallet er da største felles faktor til de to positive heltallene.

Dette kan implementeres av en funksjon som kaller seg selv

```
"""Finner stoerste felles faktor til to heltall."""
#n1 = 3456674
#n2=  45522
n1 = 140
n2=  35
def gcd(x,y):
    while  y!= 0:
        z =  x % y
       #print(z)   benyttet under test
        x=y
        y=z
    return x

print(gcd(n1,n2))
```

Oppgave 15.1. Modifiser programmet til å finne heltall m_1 og m_2 slik at

$$m_1 n_1 + m_2 n_2 = gcd(n_1, n_2)$$

Heltallene kan velges slik at $|m_1| \leq n_2/(2gcd(n_1, n_2))$ og $|m_2| \leq n_1/gcd(n_1, n_2)$.

Oppgave 15.2. Finn største felles faktor til $x^n - x$ for alle tall fra 1 til n. Finn også største felles faktor for alle x som er relativt primisk til n.

16 Halveringsmetoden

Har vi en kontinuerlig funksjon på et intervall $[a, b]$ slik at $f(a)$ og $f(b)$ har motsatt fortegn, så må det finnes en verdi c inni intervallet slik at $f(c) = 0$. Dette er en konsekvens av skjæringssetningen. Ved ådele intervallet i to finner vi enten at midtpunktet $(a+b)/2$ er et nullpunkt, eller at minst en av de to nye intervallene har samme egenskap. Ved å gjenta prosedyren n ganger ender vi op med et intervall med bredde $(b-a)/2^n$ hvor det må være minst ett nullpunkt. Dette lar seg lett inmplementere og gir en effektiv metode for å estimere et nullpunkt til en likning.

```
#Halveringsmetoden for å estimere nullpunkt til kontinuerlige funksjoner.
def f(x):    #En funksjon. Denne kan endres for å løse andre likninger
   return     x**2 -2                        #4-x**2
a=1    # To verdier slik at f(a) og f(b) har motsatt fortegn.
b=2
if f(a) == 0 :
       print("en løsning er ", a)
if f(b) == 0 :
       print("en løsning er ", b)
if f(a)*f(b) >0:
    print('Betingelsen er ikke oppfylt')
N=10  #Antall iterasjoner med halvering av lengden paa intervallet.
#Bredden på intervallet blir da (b-a)*2**(-n)
for i in range(0,N):
    c= (a+b)/2
    if f(c ) == 0 :
          print("en løsning er " , c)
    elif  f(a)*f(c)>=0 :  #Oppdaterer a hvis f(a) og f(c) har samme fortegn.
        a=c
    elif f(b)*f(c)>0:
        b=c
print(f"Det er en løsning mellom {a} og {b}")
#Med innrykk skrives intervallet opp for hvert steg i løkken.
#print(b-a) lengden på intervallet loesningen ligger i.
#Dette angir noeyaktighten til loesningen.
```

17 Kvadratrotfunksjonen

Kvadratroten av et tall a er det positive nullpunktet til funksjonen $x^2 - a$. Halveringsmetoden kan benyttes, men her er en algoritme som er langt mer effektiv.

```
import numpy as np
m=10  # m er her antall iterasjoner.
```

```
a=2  #Verdien vi skal regne ut kvadratroten til.
x=a  #Startverdien for iterasjonen. Her satt lik a.
#x= 2**((len(bin(np.ceil(a)))-3)//2)
#Dette er et estimat for kvadratroten.
for j in range(m):
    x=(x**2 + a)/(2*x)   #Oppdaterer verdien x.
    #Det er ingen referanse til j i loekken.
    print(x)    #skriver ut tilnærmingene til kvadratroten av a.
#print(a**0.5, "er kvadratroten utregnet av maskinen")
#Til sammenlikning: Verdien regnet ut av ved aa skriv roten som a^(1/2))
```

Den rekursive formelen som er benyttet her er

$$x_{n+1} = \frac{x_n^2 + a}{2x_n}$$

Den kommer fra Newtons metode anvendt på funksjonen $y = x^2 - a$. Nullpunktene er her nettopp pluss eller minus kvadratroten til a.

Oppgave 17.1. Modifiser programmet til en funksjon som kan kalles kvad.

Oppgave 17.2. Programmet kan modifiseres til å gi bedre valg av startverdi x enn bruken av a. Lag en modifisering slik at vi får en velfungerende utregning av kvadratroten når a er et veldig stort tall eller et veldig lite tall. Forsøk for eksempel med 10^n hvor n er det største heltallet slik at $100^n = 10^{2n} \leq a$.

Alternativt, kan du runde tallet opp til nærmeste heltall, gjøre det om til et binært tall og telle antall siffer s. Da er 2^{s-1} den største 2-erpotensen mindre enn eller lik tallet. Vi får et estimat for kvadratroten ved å benytte 2 opphøyd i det største heltallet mindre enn eller lik $(s-1)/2$. En kode for dette er gitt i eksempelet.

18 Trigonometriske funksjoner

Vi viser her hvordan de trigonometriske funksjonene kan lages. Metoden benytter Taylor-polynomer til å estimere sin og cos. Dette er polynomer som har de samme deriverte som funksjonene opp til en høy grad i $x = 0$. De gir gode tilnærminger til funksjonene for små verdier av x.

Her lager vi to funksjoner som er gitt ved Tayler polynomer av grad mindre enn eller lik 12. Dette gir en grei tilnærming når $|x|$ er tilstrekkelig liten.

```
def sinus(x):
return x - x**3/6+ x**5/120 - x**7/5040 + x**9/362880 - x**(11)/39916800
def cosinus(x):
return 1 - x**2/2+ x**4/24 - x**6/720 + x**8/40320 -
x**(10)/39916800  + x**{12}/479001600
```

Vi kan avgrense utregning av sin for vilkårlige vinkler til utregning av sin og cos for vinkler som er mellom 0 og $\pi/4 \cong 0.78539\ldots$.

```
###   Implementering av sinus funksjonen
import numpy as np
x= 10     #  Vinkelen
#4*(np.pi)/3 +0*np.pi
#print(np.sin (x))     Verdien av sin x regnet ut av Python

s=0.0     #variabelen som skal faa sinusverdien
omloep = 2*np.pi
x= x % omloep     #reduserer til foerste omloep fra 0 til 2pi.
#Tayler polynomer for sin og cos
def sinus(x):
    s= x -x**3/6 +x**5/120 -x**7/5040 + x**9/362880 -x**11/39916800 +
    x**13 /6227020800 -x**15 / 1307674368000
    return(s)
def cosinus(x):
    s= 1 -x**2/2 +x**4/24 - x**6/720 + x**8/40320 - x**10/3628800 +
    x**12 /479001600 - x**14/ 87178291200
    return(s)
oevrehalvdel =  x <= np.pi     #True hvis x er i oevre halvdel
#print('oevrehalvdel ' ,oevrehalvdel)
if oevrehalvdel == False :
    x = 2*np.pi - x     #Reflekterer om x-aksen
foerstekvadrant = x <= np.pi/2
if foerstekvadrant == False :     #Reflekterer om y-aksen
    x= np.pi - x
#print('foerstekvadrant ' ,foerstekvadrant)
under45 = x <= np.pi / 4
#print('under45 ', under45)
if under45 == True:
  s =  sinus (x)
else:
      s= cosinus (np.pi/2 -x)
if  oevrehalvdel == False:
    s = -s
print(s)
```

Funksjonen cosinus kan regnes ut som $\sin(x) = \cos(\pi/2 - x)$. Funksjonen tangens $\tan(x)$ kan regnes ut som $\sin(x)/\cos(x)$.

Oppgave 18.1. Modifiser programmet slik at vi ikke gjentar flere utregninger med π under reduksjonen av vinkelen til en vinkel mellom 0 og $\pi/4$ radian. For eksempel hvis vinkelen x ligger i nedre del av tredje kvadrant så reduseres x til $x - \pi$. Ligger den i øvre del av tredje kvadrant (mellom 225 og 270 grader), så reduseres vinkelen til $3\pi/2 - x$. Det er cosinus som skal aanvendes på denne vinklen. Hensikten med dette er å få mindre feilledd ved å begrense antall regneoperasjoner.

19 Annengradsformelen

Her er et program som finner løsninger til et annengradsuttrykk. Det hånterer også tilfellet hvor likningen reduseres til en lineær likning.

```
import numpy
print("Skriv inn parametrene til ax^2+bx+c=0")
a=input("a= ")
b=input("b= ")
a = float(a)
b = float(b)
c=float(input("c= "))    #Vi gjoer innput om til desimaltall med en gang.
d = b**2 - 4*a*c    #diskriminanten.
if a == 0:
    if b != 0:
        print("Likningen er lineaer og har løsningen: ", -c/b)
    elif c != 0:
        print("Likningen er lineaer og har ingen løsninger.")
    else:
        print("Likningen er lineaer og har alle reelle tall som løsninger.")
elif d<0:
    print("Likningen har ingen reelle løsninger.")
elif d==0:
    l1 = -b/(2*a)
    print("Likningen har en løsning: ", l1)
elif d>0:
    l1 = (-b+numpy.sqrt(d))/(2*a)
    l2 = (-b-numpy.sqrt(d))/(2*a)
    print("Likningen har to løsninger: ", l1 , " og " , l2)
```

20 Plotting

En pakke som lar oss tegne grafer er pyplot som igjen er en underpakke av biblioteket mathplotlib. Det er vanlig å kalle den importerte pakken for plt.

```
'''Her er noen kommandoer for å tegne opp grafer i Python.
Ekseperimenter gjerne selv med koden, og undersøk flere muligheter.
For eksempel hvordan tegne opp histogram.'''
import numpy as np
import matplotlib.pyplot as plt  #pakke med komandoer for å tegne grafer
#En funksjon.
def f(x):  #Endre uttrykket for å tegne opp grafen til andre funksjoner.
    r=x**2
    return r
#En annen funksjon.
def der_f(x):    #I dette tilfelle den deriverte til den ovenfor.
    r=2*x
```

```
    return r

N=50  #antall punkt vi plotter
plt.close('all')   #fjerner tidligere grafer
plt.xlabel('x'),  plt.ylabel('y')   #navn på aksene
dx=np.linspace(-2,2,N)  #Lager en "array" med N x-koordinater.
dy = f(dx)  #Anvender funksjonen f på hver av elementene i dx
der_dy = der_f(dx)

#print(dy)  #skriver ut dy, slik at vi kan se hvordan den ser ut.
plt.grid()  #Lager til et gitter i koordinatsystemet.  Husk tomme parenteser().
#plt finner optimale koordinater selv
#Alternativt kan vi velge angi dem selv som plt.axis([-3,3,-2,5])

plt.plot(dx,dy,'k.')  #Dette er komandoen for å plotte punktene med
#koordinater hentet fra i-te element i de to arrayene.
#I hermetegn står koder for farge på graf samt hvordan grafen skal plottes
#Default farge er blå og linjer trukket mellom hver av de plotta punktene
#Prøv også  ro , g-- , y-. r. etc.
#Vi kan skrive opp flere grafer i samme koordinatsystem
plt.plot(dx,der_dy,'g-')
plt.legend(['f','derivert av f'])
plt.show()  #Dette er kommandoen som viser oss grafen.
```

21 Primtallsfordelingen

```
def prime(n):  #Vi lager en funksjon som sjekker om et tall er et primtall.
#Funksjonen gir ut en sannhetsverdi.
    flag=True
    for m in range(2, 1+ int(n**(0.5))):
#Sjekke om n er delelig med tall mindre enn eller lik kvadratroten til n.
    if n % m == 0:    #Sjekker om m deler n
          flag = False   #Hvis n er delelig med et tall ekte stoerre enn 1
          #og           mindre enn n endrer vi flagget til falsk.
          break
    return flag
#print(prime(15))  #tester
N=911
for n in range(2, N+1):  #Skriver opp alle primtall mindre enn eller lik N
    if prime(n)==True:
        print(n, end=' ')  #Legger inn et mellomrom mellom tallene.
#Vi lager en funksjon som teller opp antall primtall mindre enn eller lik N

# Gjør dere N stor vil maskinen bruke svært lang tid på beregningene.
def PI(N):
    PI = 0
```

```
    for n in range(2, N+1):
        n = int(n)
        if prime(n)==True:
         PI += 1 #legger til 1 til PI for hvert primtall.
        # print(PI) #tester
    return PI
#print(PI(N))

import numpy as np
import matplotlib.pyplot as plt

heltall = np.arange(1,N,1)
antallp = np.zeros(N-1)
for i in range(N-1):
    antallp[i] = PI(heltall[i])

#plt.axis([0,N,0,N])
plt.xlabel('naturlige tall n')
plt.ylabel('antall primtall <= n')

plt.plot(heltall, antallp)
plt.show()

#Tegner opp forholdet mellom n/ln(n) + n/ln(n)^2 +2n/ln(n)^3
# og antall primtall <= n. Dette er tilnærming til en antiderivert til 1/ln(x)

N=6000   # Gjør dere N stor vil maskinen bruke svært lang tid på beregningene.
def PI(N):
    PI = 0
    for n in range(2, N+1):
        n = int(n)
        if prime(n)==True:
         PI += 1 #legger til 1 til PI for hvert primtall.
        # print(PI) #tester
    return PI
#print(PI(N))

import numpy as np
import matplotlib.pyplot as plt

heltall = np.arange(1,N,1)
antallp = np.zeros(N-1)
for i in range(N-1):
    antallp[i] = PI(heltall[i])/(i/np.log(i)+i/np.log(i)**2+2*i/np.log(i)**3)
linje = np.ones(N-1)
plt.plot(heltall, linje)
```

```
#plt.axis([0,N,0,N])
plt.xlabel('naturlige tall n')
plt.ylabel('antall primtall <= n /(ln(n)/n')

plt.plot(heltall, antallp)
plt.show()
```