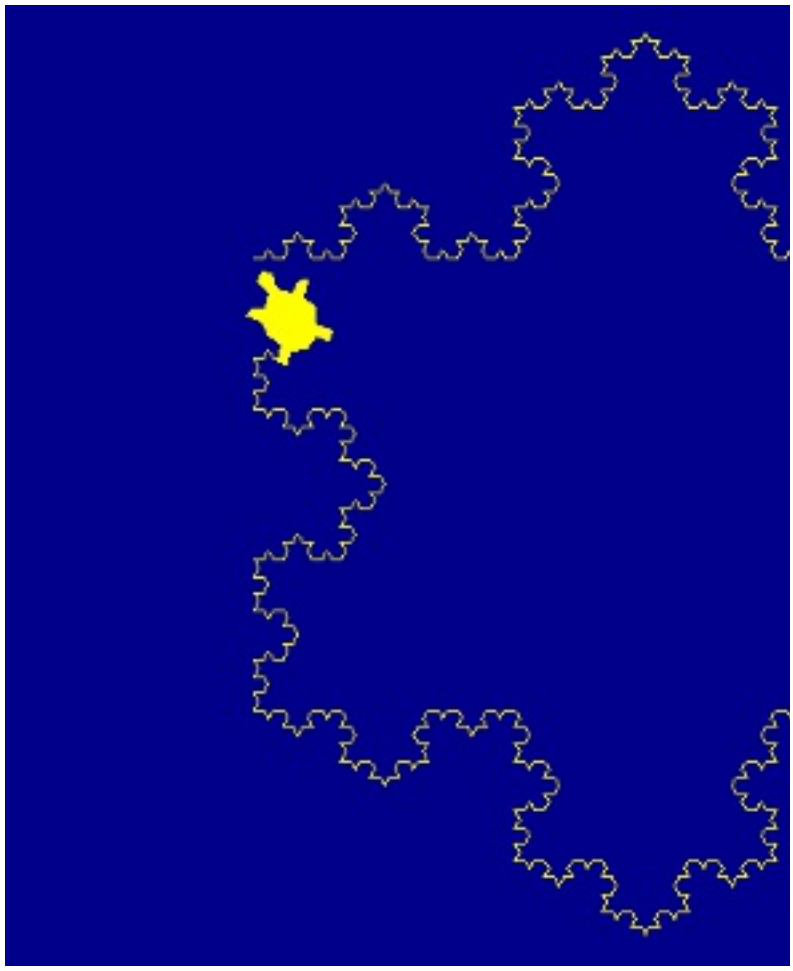
Skilpaddefraktaler

Introduksjon

Vi vil nå jobbe videre med skilpaddekunsten fra tidligere. Denne gang
fraktaler. Fraktaler er figurer som bygges opp av små kopier av seg se
funksjoner og rekursjon.



Steg 1: Husker du skilpadd

Vi har brukt skilpaddebiblioteket `turtle` tidligere. Du husker kanskje

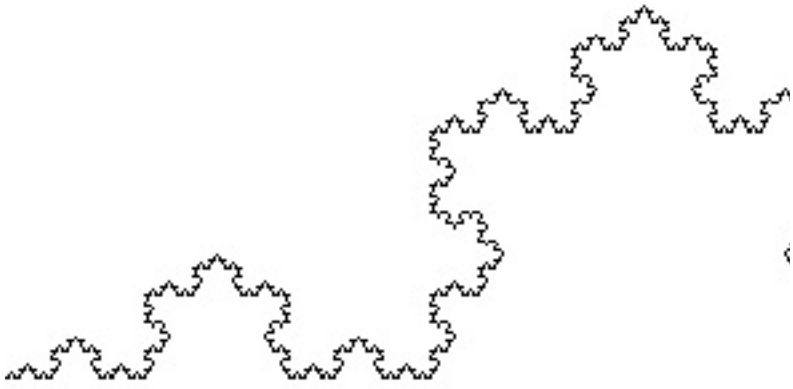
```
from turtle import *

shape('turtle')
shapesize(2)
bgcolor('darkblue')
color('yellow')
speed(3)

forward(270)
```

Sjekkliste

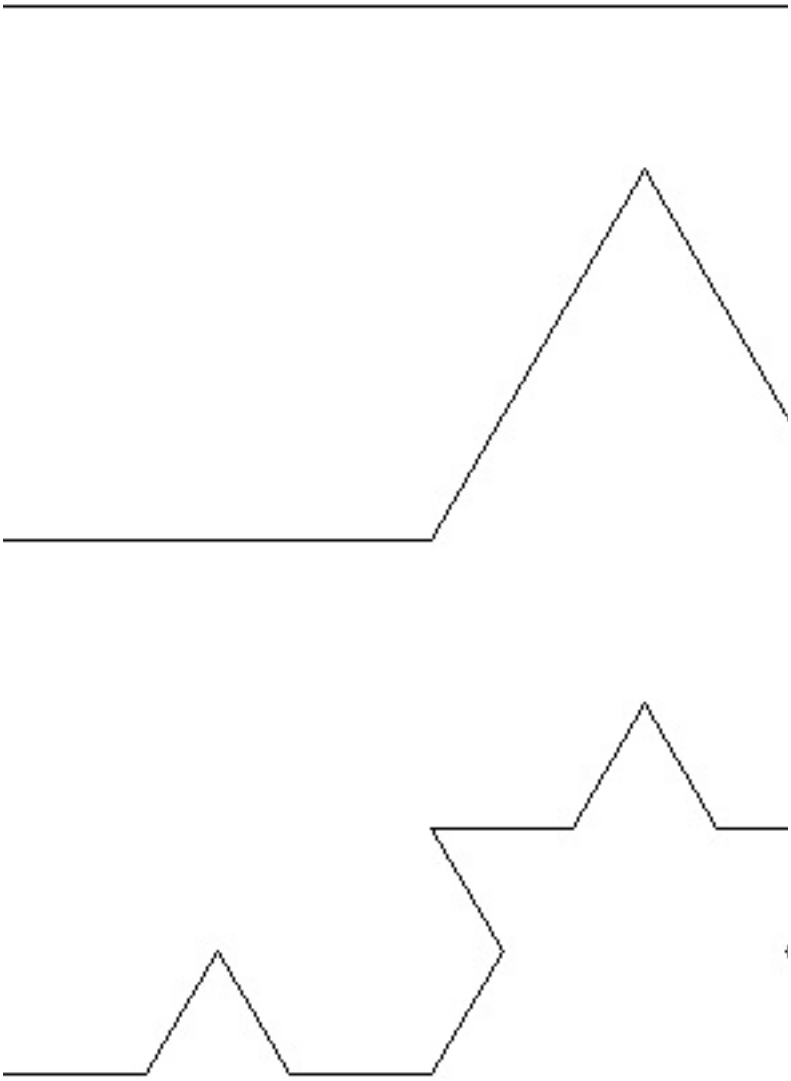
- [ ] Skriv inn programmet over. Lagre det med navnet `snoflak.py`
- [ ] Endre litt på tallene og fargene i koden slik at du husker hva de fargekombinasjon du liker?

Steg 2: En fraktal

En fraktal er en figur som er bygd opp av mindre kopier av seg selv. V



Dette er en fraktal, klarer du å se at den består av mange små kopier
figurene?



Den øverste figuren er bare en rett strek. Den neste figuren består av
ser nærmere på den tredje figuren ser du at den består av fire kopier
mer komplisert fjell.

Hvis du nå ser tilbake på den første figuren, ser du hvordan den bestå

Steg 3: Vi tar det stegvis

Vi skal se på hvordan vi kan lage funksjoner som kan tegne figurene c

Sjekkliste

- [ ] Den rette streken har vi jo allerede tegnet. La oss bare endre lit

```
from turtle import *

shape('turtle')
shapesize(2)
bgcolor('darkblue')
color('yellow')
speed(3)

def en():
    forward(270)

en()
```

Husk at vi må kalle funksjonen for at den skal bli gjort.

- [ ] La oss nå legge til en funksjon `to()` som tegner den andre figu
samme filen.

```
def to():
    forward(90)
    left(60)
    forward(90)
    right(120)
    forward(90)
    left(60)
    forward(90)

to()
```

Ser du sammenhengen mellom figuren og koden?

- [ ] Kjør programmet ditt. Husk at du kan styre hvilke figurer som te

`en` og `to` er definert trenger du ikke kalle begge funksjonene.

I `to` har vi brukt `forward(90)`, mens i `en` brukte vi `forward(2` ganger mindre. Men vi har brukt vinkler slik at de fire strekene i `en`.

La oss endre litt i funksjonene slik at vi bruker `en` i stedet for `f`

```
def en(lengde):
    forward(lengde)

def to(lengde):
    en(lengde / 3)
    left(60)
    en(lengde / 3)
    right(120)
    en(lengde / 3)
    left(60)
    en(lengde / 3)

to(270)
```

Kjør programmet igjen. Tegnes fortsatt de samme figurene?

Vi vil nå tegne den tredje figuren. En måte å gjøre dette på kan opprinnelig gjorde for `to`. Du trenger ikke skrive inn denne kod at det stemmer?

```
def tre():
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
    left(60)
```

```
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
    left(60)
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
```

Dette er en kjedelig måte å programmere på: Vi må skrive kjem
å gjøre endringer i koden.

Hvis du ser litt nærmere på koden vil du se at linjene

```
    forward(30)
    left(60)
    forward(30)
    right(120)
    forward(30)
    left(60)
    forward(30)
```

går igjen flere ganger. Sammenlign disse linjene med funksjone

koden vår?

- [ ] Vi har sett at koden til `tre` består av flere kopier av koden til `t` tidligere, vi bare kaller `to`. Skriv inn følgende kode i den samm

```
def tre(lengde):
    to(lengde / 3)
    left(60)
    to(lengde / 3)
    right(120)
    to(lengde / 3)
    left(60)
    to(lengde / 3)

tre()
```

- [ ] Klarer du å tegne alle tre figurene nå?

Steg 4: Her kan vi kombine

Nå skal vi lage en funksjon som kan tegne alle tre figurene!

Sjekkliste

Nå kommer det morsomste. Før vi kaster bort tid på å lage flere funks
lage en funksjon som kan lage alle disse for oss!

- [ ] Sammenlign funksjonene `to` og `tre`. Ser du at de er nesten he

Vi skal nå bruke noe som kalles rekursjon for å lage en funksjon som g
du kanskje fra tidligere. Det betyr at vi lager en funksjon som kaller se

Med rekursjon ser man gjerne på det enkle tilfellet og det generelle til

vi bare trenger å tegne en rett strek.

- [ ] Legg til denne funksjonen. Dette er det enkle tilfellet:

```
def fjell(lengde, dybde):
    if dybde == 1:
        forward(lengde)
        return
```

Her bruker vi `return` for å si at vi ikke vil gjøre mer for det enkl

- [ ] Det generelle tilfellet er det vi har sett tidligere i `to` og `tre`. M `tre` med samme kode. Utvid funksjonen `fjell` slik at den ser

```
def fjell(lengde, dybde):
    if dybde == 1:
        forward(lengde)
        return

    fjell(lengde / 3, dybde - 1)
    left(60)
    fjell(lengde / 3, dybde - 1)
    right(120)
    fjell(lengde / 3, dybde - 1)
    left(60)
    fjell(lengde / 3, dybde - 1)
```

Kjenner du igjen koden fra tidligere?

- [ ] Prøv å tegn

```
fjell(270, 2)
```

og

```
fjell(270, 3)
```

Gir dette samme resultat som `to(270)` og `tre(270)`?

- [ ] Den nye funksjonen gjør enda mer enn `to` og `tre`. Vi kan bruk
`6)`. Denne vil bruke litt tid. Bruk `speed(11)` for at skilpadden sk

Steg 5: Et snøflak

Vi skal nå kombinere flere slike fjell til et fint snøflak.

Sjekkliste

- [ ] Til sist skal vi kombinere flere kall til `fjell`-funksjonen vår for å
Ser du hvordan snøflaket består av tre fjell?

  Legg til denne funksjonen:

  ```
  def snoflak(lengde, dybde):
      for i in range(3):
          fjell(lengde, dybde)
          right(120)
  ```

- [ ] Prøv å kall denne `snoflak`-funksjonen med forskjellige lengder

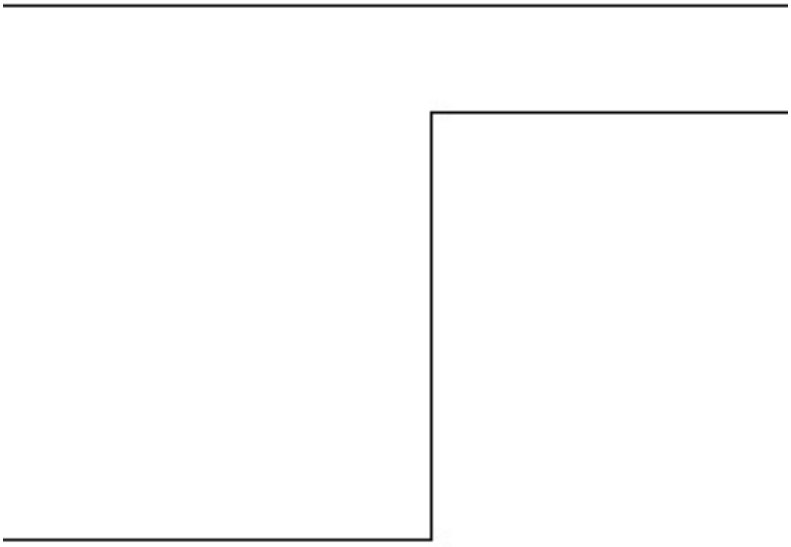
Dette snøflaket er en av de mest kjente fraktalene. Det har fått navne
denne figuren het Helge von Koch.

Steg 6: Firkantede fjell

La oss se på en variant av snøflaket.

Sjekkliste

Vi vil nå lage en fraktal på samme måte som Koch-figuren men med e
vi bruke en firkant som i figuren under:



- [ ] Lag en ny fil som du kaller `firkantfjell.py`.
- [ ] Som tidligere så kan vi prøve å tegne dette nye fjellet med funk

```
from turtle import *

shape('turtle')
shapesize(2)
bgcolor('darkblue')
color('yellow')
speed(3)

def en(lengde):
    forward(lengde)

def to(lengde):
    en(lengde / 3)
    left(90)
```

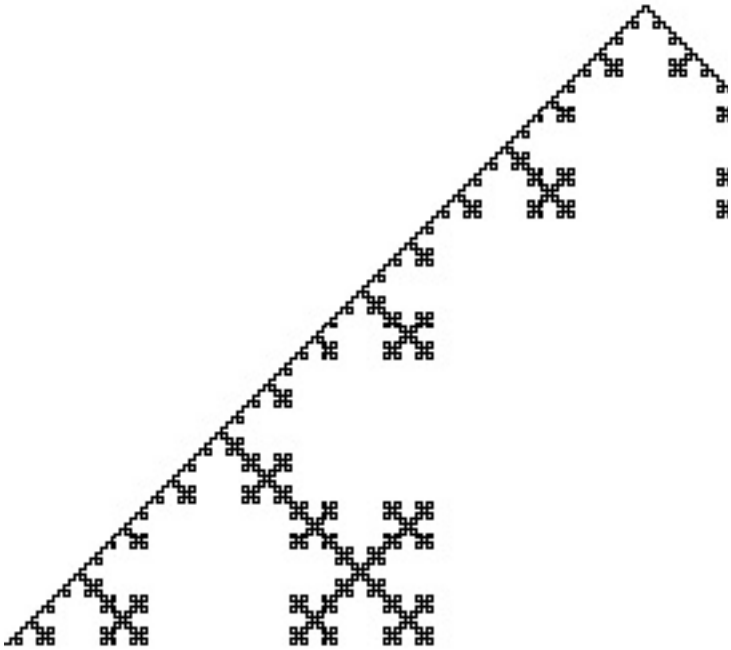
```
    en(lengde / 3)
    right(90)
    en(lengde / 3)
    right(90)
    en(lengde / 3)
    left(90)
    en(lengde / 3)

to(270)
```

- [ ] Men vi har jo lært at det er mye bedre å bruke rekursjon. Vi vil l
firkantfjell ved at den kaller seg selv.

  Prøv selv om du kan skrive denne. Se på hvordan vi laget `fjell`

```
def firkantfjell(lengde, dybde):
    if dybde == 1:
        # Her må du programmere det enkle tilfellet
        return

    # Her må du programmere det generelle tilfellet
```

- [ ] Test koden din. Blir det riktig? Nedenfor ser du et eksempel hvo

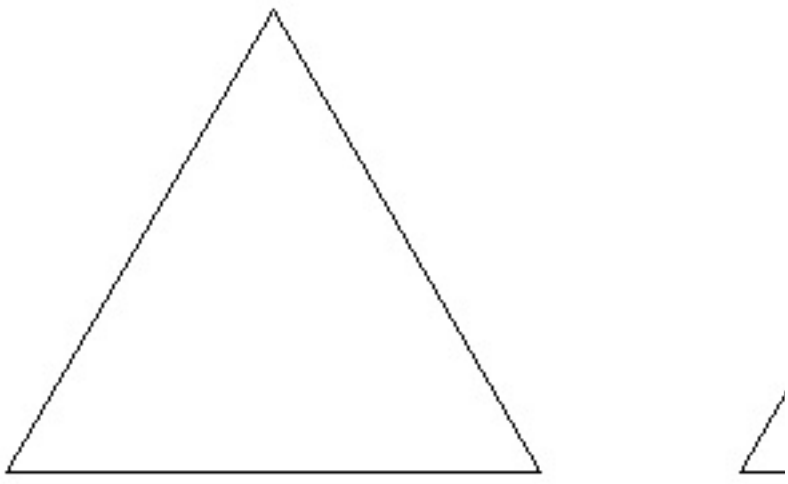


Steg 7: Trekanter

Vi trenger ikke bare bruke rette streker for det enkle tilfellet.

Sjekkliste

Vi skal nå lage en fraktal basert på trekanter. La oss se på de første st



Her ser vi at vi har en trekant som byttes ut med tre mindre trekanter

- [ ] Lag en ny fil `trekant.py` og legg til de vanlige kommandoene p

- [ ] I det enkle tilfellet vil vi nå tegne en trekant. Det kan vi gjøre på

```
def trekant(lengde, dybde):
    if dybde <= 1:
        pendown()
        for i in range(3):
            forward(lengde)
            left(120)
        penup()
        return
```

- [ ] For det generelle tilfellet må vi stable tre trekanter. Det kan vi g
koden med figuren. Ser du sammenhengen?

```
    trekant(lengde / 2, dybde - 1)
    forward(lengde / 2)
    trekant(lengde / 2, dybde - 1)
    left(120)
    forward(lengde / 2)
    right(120)
    trekant(lengde / 2, dybde - 1)
    right(120)
    forward(lengde / 2)
    left(120)
```

- [ ] Tegn noen trekanter med forskjellig dybde og størrelse. Denne f
navnet Sierpinski-trekanten.

Prøv selv

Det finnes mange fraktaler, og du kan lage dine helt egne også!

Prøv for eksempel å endre litt på vinklene og lengdene i `fjell` - ell

Eller kanskje du kan lage en helt annen figur? Prøv og tegn dine eg
Python.

Her er et forslag til en figur du kan prøve, men prøv også å lage dir