

# **Pythonmatte**

**Programmering i matematikk programfag på vgs**

Torodd F. Ottestad

# Innhald

<b>Om boka</b>	<b>4</b>
Om meg . . . . .	4
 <b>I Sannsyn og simulering (S1/S2)</b>	 <b>5</b>
<b>1 Terningar og intro til simulering</b>	<b>7</b>
1.1 Ein terning . . . . .	7
1.2 Fleire terningar . . . . .	9
1.3 Nøyaktighet . . . . .	10
 <b>2 Samansette forsøk - choice</b>	 <b>12</b>
2.1 Teoretisk sannsyn . . . . .	12
2.2 Simulering av twist-trekket . . . . .	12
2.2.1 Med tilbakelegging . . . . .	13
2.2.2 Utan tilbakelegging . . . . .	14
2.3 Ikkje uniforme sannsynsmodellar . . . . .	14
 <b>3 Simulering av ulike fordelingar</b>	 <b>16</b>
3.1 Binomisk sannsyn . . . . .	16
3.2 Hypergeometrisk sannsyn . . . . .	16
3.3 Normalfordelt sannsyn . . . . .	16
 <b>4 Hypotesetesting</b>	 <b>17</b>
 <b>II Følgjer og rekker (S2/R2)</b>	 <b>18</b>
<b>5 Følgjer</b>	<b>20</b>
5.1 Aritmetiske følgjer . . . . .	20
5.1.1 Rekursiv formel for ledd n . . . . .	20
5.1.2 Eksplisitt formel for ledd n . . . . .	22
 <b>6 Rekker</b>	 <b>23</b>

<b>III Funksjonar (S1/S2/R1/R2)</b>	<b>24</b>
<b>7 Plotting</b>	<b>26</b>
7.1 Funksjonar med delt forskrift . . . . .	26
7.1.1 Alternativ: <code>if/else</code> . . . . .	28
7.1.2 Diskontinuerlege funksjonar . . . . .	29
<b>8 Derivasjon</b>	<b>33</b>
<b>9 Integrasjon</b>	<b>34</b>
<b>Vedlegg</b>	<b>35</b>
<b>A Tips til programmeringa</b>	<b>35</b>
A.1 Jupyter lab . . . . .	35
A.2 Miniconda . . . . .	35

# Om boka

Denne boka inneheld ulike måtar ein kan nytta programmering på i matematikk (programfag på vgs).

Eg nyttar Python som programmeringsspråk gjennom heile boka .

## OBS

Boka er under utvikling og vert oppdatert med ujamne mellomrom. Sist oppdatert: `{{today}}`

For dei spesielt interesserte er boka laga med Quarto. For å lære meir om Quarto-bøker kan ein kikka [her](#).

## Om meg

[Her](#) kan du lesa meir om meg

---

Logo: Programmer icons created by juicy\_fish - Flaticon

## **Part I**

# **Sannsyn og simulering (S1/S2)**

I det følgjande kapitlet skal me sjå på korleis me kan simulera ulike stokastiske forsøk i Python. Me ser på alt frå [enkle simuleringar](#) i uniforme modellar, [samansette forsøk](#), og vidare binomiske, hypergeometriske og normalfordelte forsøk. Me ser òg på hypotesetesting i Python.

# 1 Terningar og intro til simulering

Ein fin stad å starta med simulering er med terningar. Her er sannsynet *uniformt* (det er like sannsynleg å få 2 som 5), og dei ulike utfalla er heiltal.

Det første som må gjerast er å gjera i stand “trekkaren” vår. Eg bruker her ein tilfeldigheits-generator frå NumPy (dokumentasjon [her](#)).

```
1 import numpy as np
2 rng = np.random.default_rng()
```

Når me no har klargjort generatoren kan me bruka den innebygde `integers`-funksjonen for å trilla ein terning.

## ! Merk

Dei to linjene med kode over **må** vera med i programmet for at det skal funka. I mange av døma i boka er ikkje desse to linjene med i alle kodesnuttane.

## 1.1 Ein terning

```
1 terning = rng.integers(1, 7)
2 print(terning)
```

6

## ! Merk

Her er verdien `terning` eit heiltal (*integer*) **større eller lik** 1 og **mindre enn** 7. Sidan det er heiltal me trekk er dermed

$$\text{terning} \in \{1, 2, 3, 4, 5, 6\}$$

For å trilla fleire terningar kan me anten bruka løkker:

```

1 for i in range(10):
2     print(rng.integers(1, 7))

```

```

3
4
1
6
5
2
1
5
1
5

```

eller så kan me leggja inn eit argument `size` i `integers`. Då blir output ein array (ein form for liste) med `size` terningar:

```

1 terningar = rng.integers(1, 7, size=10)
2 print(terningar)

```

```
[2 3 4 6 1 1 1 1 6 6]
```

No har me det me treng for å kunna simulera eit stokastisk forsøk og estimera sannsyn ut frå simuleringa. Til dømes kan me prøva å finna ut av kor sannunleg det er å trilla 5 eller 6 på ein terning:

```

1 N = 1000000 # tal simuleringar
2
3 terningar = rng.integers(1, 7, size=N)
4
5 gunstige = sum(terningar >= 5)
6
7 sannsyn = gunstige / N
8
9 print(f"Sannsynet for 5 eller 6 er {sannsyn:.4f}")

```

Sannsynet for 5 eller 6 er 0.3336



💡 Forklaring: `gunstige = sum(terningar >= 5)`

For å forstå denne ser me på eit døme:

```
1 array = np.array([1, 2, 3, 4, 5, 6])
2
3 større_enn_3 = array > 3
4
5 print(array)
6 print(større_enn_3)
7 print(sum(større_enn_3))
```

```
[1 2 3 4 5 6]
```

```
[False False False  True  True  True]
```

```
3
```

Altså gjer me om verdier til `True` eller `False`. Python reknar `True` som 1 og `False` som 0. Når me då summerer alle elementa i `array` får me antall `True` i arrayen.

## 1.2 Fleire terningar

Spørsmål som “Kva er sannsynet for at produktet av to terningar er 8 eller mindre” er fint å finna svar på ved hjelp av simulering:

```
1 N = 1000000
2
3 terning1 = rng.integers(1, 7, size=N)
4 terning2 = rng.integers(1, 7, size=N)
5
6 produkt = terning1 * terning2
7 gunstige = sum(produkt <= 8)
8 sannsyn = gunstige / N
9
10 print(f"Sannsynet er {sannsyn:.4f}")
```

Sannsynet er 0.4452

💡 Forklaring: `produkt = terning1 * terning2`

Kodelinja finn produktet av element på samme plass i dei to arrayane. Sjekk dømet:

```
1 a = np.array([1, 2, 3, 4, 5])
2 b = np.array([6, 7, 8, 9, 10])
3
4 c = a * b
5
6 print(c)
```

```
[ 6 14 24 36 50]
```

$1 \cdot 6 = 6$  og  $2 \cdot 7 = 14$ ...

## 1.3 Nøyaktighet

Sjekkar kva som skjer når me triller fleire og fleire terningar (eller ein terning fleire gongar). For å visa samanhengen plottar me resultatet. I dømet ser me på sannsynet for å trilla 4 på ein terning.

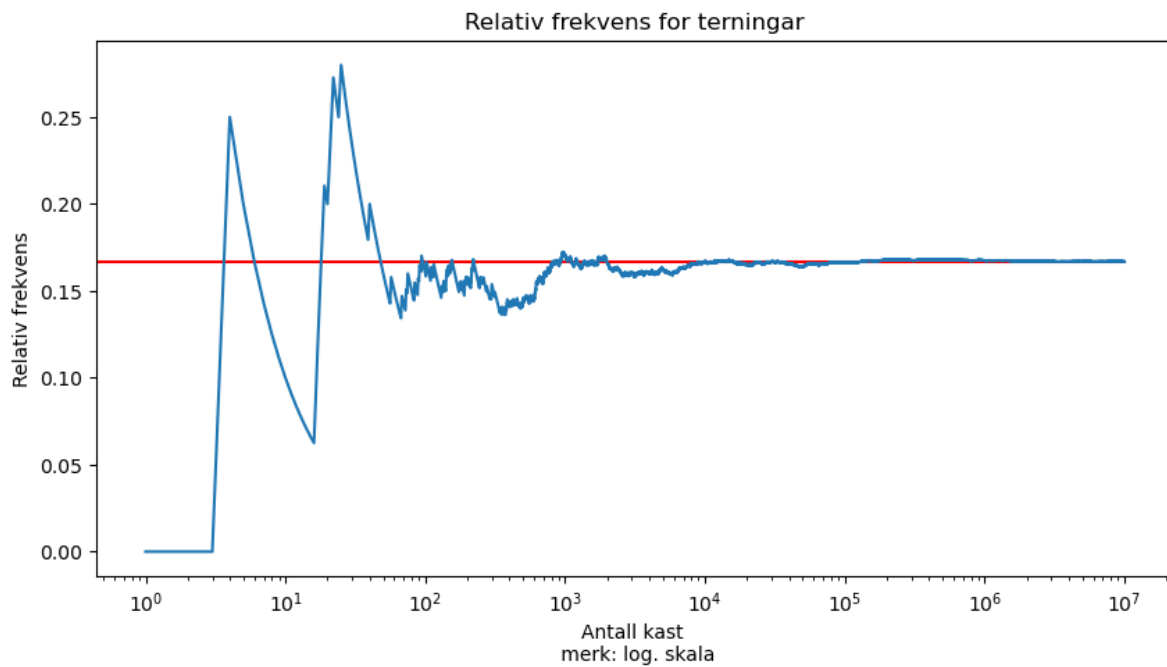
```
1 import matplotlib.pyplot as plt
2
3 # antall kast
4 N = 10000000
5
6 # triller terningar
7 terningar = rng.integers(1, 7, size=N)
8
9 # finn den kumulative summen av terningar som er lik 4
10 kumulativ_sum = np.cumsum(terningar == 4)
11
12 # lager "x-akse" frå 1 til N
13 x = np.arange(1, N + 1)
14
15 # finn relativ frekvens
16 rel_frekvens = kumulativ_sum / x
17
18 plt.figure(figsize=(10, 5))
19 plt.hlines(1/6, 0, N, color="red")
```

# lager ein figur med 10x5 mål  
# teiknar ein linje med farge "red" for d

```

20 plt.plot(x, rel_frekvens)           # plottar x-akse og y-akse
21 plt.xscale("log")                  # logaritmisk x-akse
22 plt.xlabel("Antall kast \n merk: log. skala") # namn på x-aksen
23 plt.ylabel("Relativ frekvens")      # namn på y-aksen
24 plt.title("Relativ frekvens for terningar") # tittel på figur
25 plt.show()

```



Her ser me at di fleire kast me gjennomfører, di nærare kjem den relative frekvensen den teoretiske verdien for å trilla ein firar på vanleg terning.

$$P(\text{firar}) = \frac{1}{6} \approx 0.167$$

## 2 Samansette forsøk - choice

No skal me sjå på korleis me kan simulera eit enkelt samansett forsøk. Oppgåva me skal sjå på er denne:

I ei skål ligg det 7 banan-twist og 3 daim-twist. Kva er sannsynet for at me får banan når me trekk ut to bitar frå skåla. (Både med og utan tilbakelegging)

### 2.1 Teoretisk sannsyn

Først kan me sjå på kva det teoretiske sannsynet er for desse to. Ofte når me bruker simulering er det fordi det er vanskeleg å finna svaret ved rekning, men i dette dømet er det ikkje så vanskeleg.

Med tilbakelegging

$$P(\text{BB}) = \frac{7}{10} \cdot \frac{7}{10} = \frac{49}{100} = 0.490$$

Utan tilbekelegging

$$P(\text{BB}) = \frac{7}{10} \cdot \frac{6}{9} = \frac{42}{90} \approx 0.467$$

### 2.2 Simulering av twist-trekket

```
1 # importerar og lagar ein random generator
2 from numpy.random import default_rng
3 rng = default_rng()
4
5 # antall simuleringar
6 N = 1000000
7
8 # lagar liste med twist-skåla
9 twistskål = ["Banan"]*7 + ["Daim"]*3
10
```

```

11 # skriv ut twisteskåla
12 print(twistskål)

```

```
['Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Daim', 'Daim', 'Daim']
```

### 2.2.1 Med tilbakelegging

```

1  BB = 0
2
3  for i in range(N):
4      twist = rng.choice(twistskål, size = 2)
5      if twist[0] == "Banan" and twist[1] == "Banan":
6          BB += 1
7
8  rel_frekk = BB/N
9
10 print(f"Sannsynet for at me trekk to banantwist er {rel_frekk}")

```

Sannsynet for at me trekk to banantwist er 0.489409

#### Tips

For ein litt meir elegant kode *kan* me droppa if-setningen i løkka vår. Dette kan me gjera ved å gjera ein boolsk variabel (True eller False) om til eit heiltal. Då blir True = 1 og False = 0

```

1  BB = 0
2
3  for i in range(N):
4      twist = rng.choice(twistskål, size = 2)
5      BB += int(twist[0] == "Banan" and twist[1] == "Banan")
6
7  rel_frekk = BB/N
8
9  print(f"Sannsynet for at me trekk to banantwist er {rel_frekk}")

```

Sannsynet for at me trekk to banantwist er 0.489524

Me kan sjå kor langt unna den teoretiske verdien me kjem:

```

1 feil = abs(rel_frekk - 49/100)
2 print(f"Feilen blir {round(feil, 6)} når me gjer {N} simuleringar")

```

Feilen blir 0.000476 når me gjer 1000000 simuleringar

Dersom me vil ha eit enno meir nøyaktig resultat kan me gjera fleire simuleringar, dette kjem me litt attende til seinare. Merk at programmet vil fort ta ganske lang tid å køyra etter kvart som talet på simuleringar aukar.

## 2.2.2 Utan tilbakelegging

Forskjellen blir ikkje stor her. Det einaste me gjer er å leggja til `replace = False` som argument i `choice`-funksjonen

```

1 BB = 0
2
3 for i in range(N):
4     twist = rng.choice(twistskål, size = 2, replace = False)
5     if twist[0] == "Banan" and twist[1] == "Banan":
6         BB += 1
7
8 rel_frekk = BB/N
9
10 print(f"Sannsynet for at me trekk to banantwist er {rel_frekk}")

```

Sannsynet for at me trekk to banantwist er 0.466383

```

1 feil = abs(rel_frekk - 42/90)
2 print(f"Feilen blir {round(feil, 6)} når me gjer {N} simuleringar")

```

Feilen blir 0.000284 når me gjer 1000000 simuleringar

## 2.3 Ikkje uniforme sannsynsmodellar

Dette dømet me har sett på er eit døme på ein ikkje-uniform sannsynsmodell, sidan sannsynet for Banan og Daim ikkje er det same. I starten laga me ei liste med alle twistane i skåla. I

dette dømet er det praktisk, sidan me har eit lite utfallsrom (banan og daim) og kontroll på kor mange det er av kvar.

Av og til kan det vera nyttig å definera ikkje-uniforme sannsynsmodellar på ein litt anna måte.

```
1  twistar = ["Banan", "Daim"]
2  sannsyn = [7/10, 3/10]
3
4  to_twist = rng.choice(twistar, size = 2, p = sannsyn)
5
6  print(f"Me trekk {to_twist}")
```

Me trekk ['Banan' 'Daim']

Dette kan brukast viss me veit utfallsrommet og sannsynet for kvart av utfalla, ikkje nødvendigvis antallet. F.eks. blodtype hos tilfeldige personar i befolkningen.

## **3 Simulering av ulike fordelingar**

### **3.1 Binomisk sannsyn**

### **3.2 Hypergeometrisk sannsyn**

### **3.3 Normalfordelt sannsyn**



## 4 Hypotesetesting

```
1 print("Kjem etterkvart")
```

Kjem etterkvart

## **Part II**

### **Følger og rekker (S2/R2)**

I dette kapitlet skal me sjå på korleis me kan bruka Python til å arbeida med følgjer og rekker.

Ei talfølgje er ein serie tal. Dei kjem på ulike formar. Eit døme på ei talfølgje er:

```
1  for i in range(1, 6):  
2      print(i, end=",")  
3  
4  print("...")
```

1,2,3,4,5,...

Dette kjenner me att som dei 5 første naturlege tala,  $\mathbb{N}$ , og er eit enkelt døme på ei talfølgje. Kvart av tala i følgja kallar me for *ledd*. Det første leddet vert kalla  $a_1$  medan det  $n$ -te leddet vert kalla  $a_n$ .

Vidare i kapitlet skal me sjå på ulike typer talfølgjer, eksplisitte og rekursive funksjonar for å finna ledd i talfølgjer. Vidare ser me på ulike typer rekker. Før me mot slutten ser på nokre døme på praktisk bruk av følgjer og rekker (lån og sparing).

## 5 Følgjer

På førre side såg me eit kjapt døme på ei talfølgje,

$$1, 2, 3, 4, 5, \dots$$

Av denne talfølgja ser me to ting. For det første er ho uendeleg, sidan det ikkje er definert nokon ende, berre "...". Me ser og at det er ein fast differanse mellom kvart av ledda. Talfølgjer med fast differanse mellom ledda kallar me *arismetiske* talfølgjer.

### 5.1 Aritmetiske følgjer

Eit anna døme på ei aritmetisk talfølgje er denne

$$3, 7, 11, 15, \dots$$

Her ser me at det første leddet  $a_1 = 3$  og at differansen  $d = 4$ .

At det er ein fast differanse mellom kvart av ledda betyr at dersom me skal finna eit ledd ( $a_n$ ), må me ta leddet før ( $a_{n-1}$ ) og leggja til differansen ( $d$ ). Dermed får me

$$a_n = a_{n-1} + d$$

Frå dømet over ser me at det stemmer,  $7 = 3 + 4$  og  $11 = 7 + 4$  osv.

#### 5.1.1 Rekursiv formel for ledd $n$

Samanhengen over kan me nytta for å finna  $a_n$  rekursivt. Rekursjon handlar om gjentakning, så tanken er at me kan gjenta formelen for å finna ledd  $n$ . Me kan laga ein funksjon som kun tek utgangspunkt i opplysningen om at  $a_n = a_{n-1} + d$ .

```
1 def a(n):  
2     return a(n-1) + 4  
3
```

```
4 print(a(4))
```

RecursionError: maximum recursion depth exceeded

**Men** om du prøver å køyra denne koden vil du få ein feil:

```
RecursionError: maximum recursion depth exceeded
```

Om du ser på koden, ser du kanskje kva som er problemet?

Det let seg løysa om me definerer ein rekursjonsbotn (i dette tilfellet  $a_1$ ). Me prøver på nytt:

```
1 def a(n):
2     if n == 1:
3         return 3
4     else:
5         return a(n-1) + 4
6
7 print(a(4))
```

15

Denne funksjonen kan me bruka for å t.d. skriva ut dei 10 første ledda i følgja:

```
1 for i in range(1, 11):
2     print(a(i), end = ", ")
3
4 print("...")
```

3, 7, 11, 15, 19, 23, 27, 31, 35, 39, ...

Ulempen med denne rekursive funksjonen er at han må gjennom *alle* dei føregåande ledda for kvart ledd du bruker han for å finna. Så om du skal finna  $a_{1000}$  vil formelen finna alle ledda før. Og på nytt når du ser etter  $a_{1001}$ ... Det kan fort bli både tidkrevjande og tungvint, sjølv for datamaskina.

### 5.1.2 Eksplisitt formel for ledd n

Me kan sjå om me finn ein eksplisitt måte å finna  $a_n$  på (altså finna direkte, utan rekursjon).  
Me ser på dømet igjen.

$$3, 7, 11, 15, \dots$$

Me ser at

$$a_2 = 7 = 3 + 4$$

$$a_3 = 11 = 7 + 4 = 3 + 4 + 4$$

$$a_4 = 15 = 11 + 4 = 7 + 4 + 4 = 3 + 4 + 4 + 4$$

Altså er

$$a_n = a_1 + (n - 1)d$$

som me kan programmera som

```
1 def a(n):  
2     return 3 + (n-1)*4  
3  
4 print(a(4))
```

15

Og igjen kan me skriva ut dei ti første ledda:

```
1 for i in range(1, 11):  
2     print(a(i), end=" ", )  
3  
4 print("...")
```

3, 7, 11, 15, 19, 23, 27, 31, 35, 39, ...

## 6 Rekker

```
1 a = "test"
```

## **Part III**

# **Funksjonar (S1/S2/R1/R2)**



- plotting
- kurvetilpassing / modellering
- derivasjon
- integrasjon

# 7 Plotting

## 7.1 Funksjonar med delt forskrift

Nokre funksjonar kan ha ulik definisjon på ulike intervall. Desse kallar me funksjonar med delt forskrift. Me ser på funksjonen

$$f(x) = \begin{cases} x^2 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Denne kan definerast med `numpy.piecewise()`.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

Startar med å definera funksjonen. Det gjer me med

```
1 numpy.piecewise(x, condlist, funclist)
```

der `condlist` er ei liste med betingelse/intervall og `funclist` funksjonane i same rekkefølge

```
1 def f(x):
2     return np.piecewise(x, [x <= 0, x > 0], [lambda x: x**2, lambda x: x])
```

💡 Kva er `lambda`?

`lambda x:` er kortversjonen av

```
1 def g(x):
2     return x**2
```

Dermed kan me heller definera funksjonen slik:

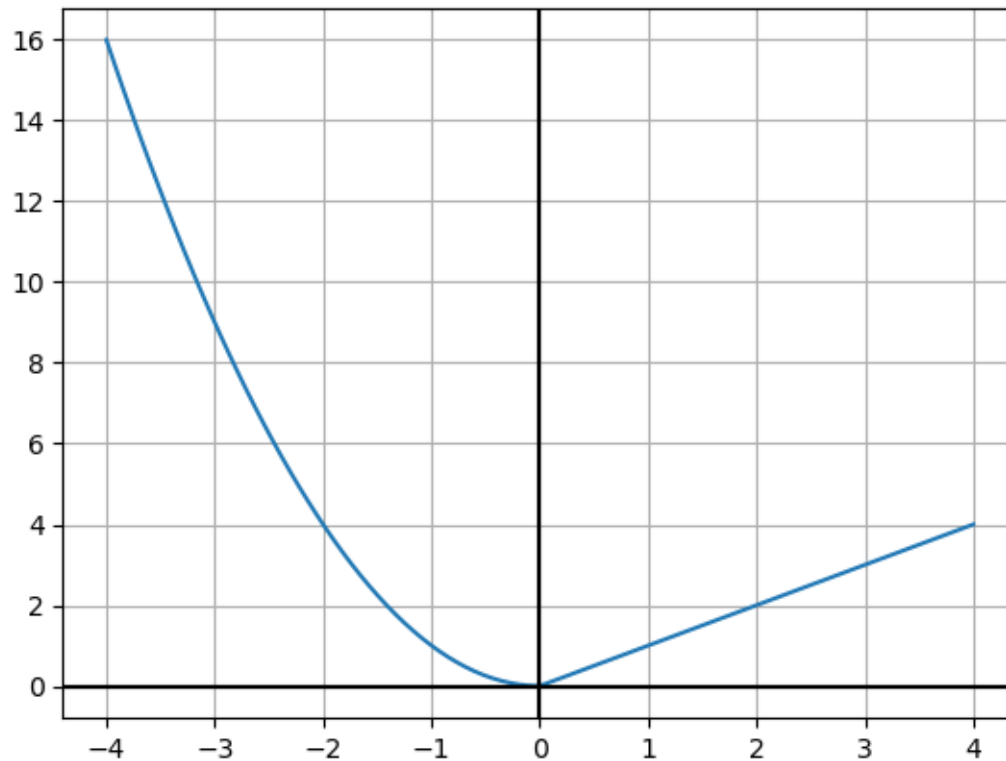
```
1 x = np.linspace(-3,2,5)
2 g = lambda x: x**2
3
4 y = g(x)
```

Lagar ein array med 100  $x$ -verdiar og finn vidare  $y$ -verdiane med funksjonen me definerte.

```
1 x = np.linspace(-4,4,100)
2 y = f(x)
```

Plottar grafen:

```
1 plt.plot(x, y)
2
3 # pynt
4 plt.axhline(0, color="black")
5 plt.axvline(0, color="black")
6 plt.grid()
7 plt.show()
```



### 7.1.1 Alternativ: if/else

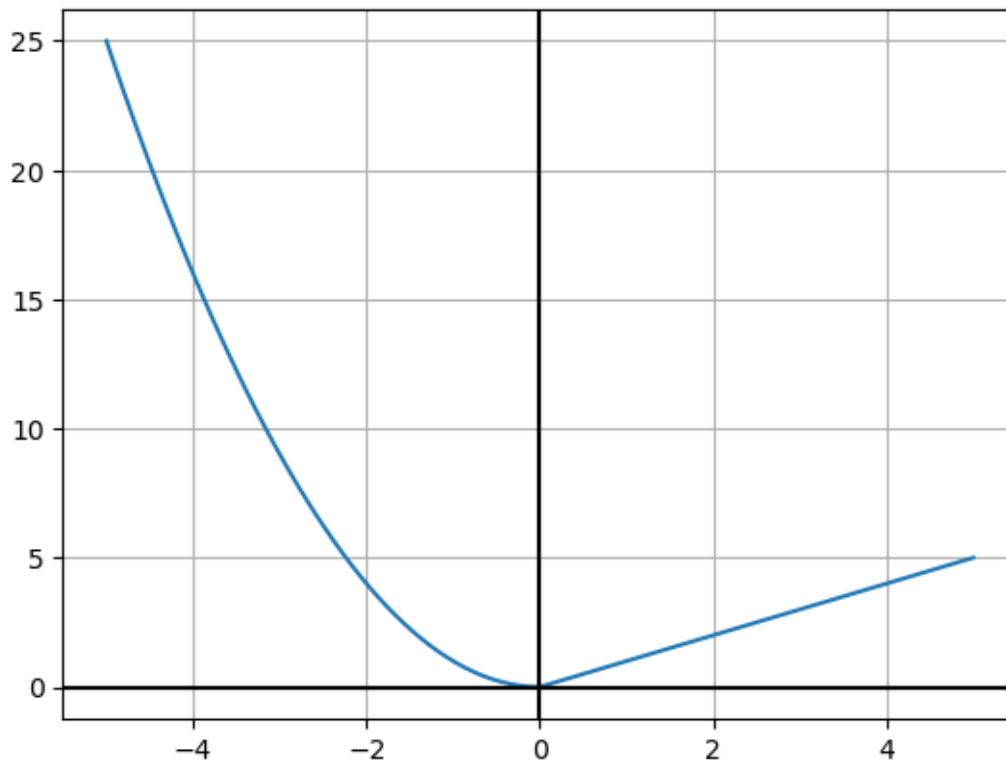
Ein annan måte dette kan gjerast på er å bruka betingelsar og løkker. Dette kan me gjera slik:

```
1 # definerer funksjonen
2 def f(x):
3     if x <= 0:
4         return x**2
5     else:
6         return x
7
8 # lager x-verdiar
9 x_verdiar = np.linspace(-5, 5, 100)
10
11 # rekner ut y-verdiane
12 y_verdiar = [f(x) for x in x_verdiar]
13
```

```

14 # plottar
15 plt.plot(x_verdiar, y_verdiar)
16
17 # pynt
18 plt.axhline(0, color="black")
19 plt.axvline(0, color="black")
20 plt.grid()
21 plt.show()

```



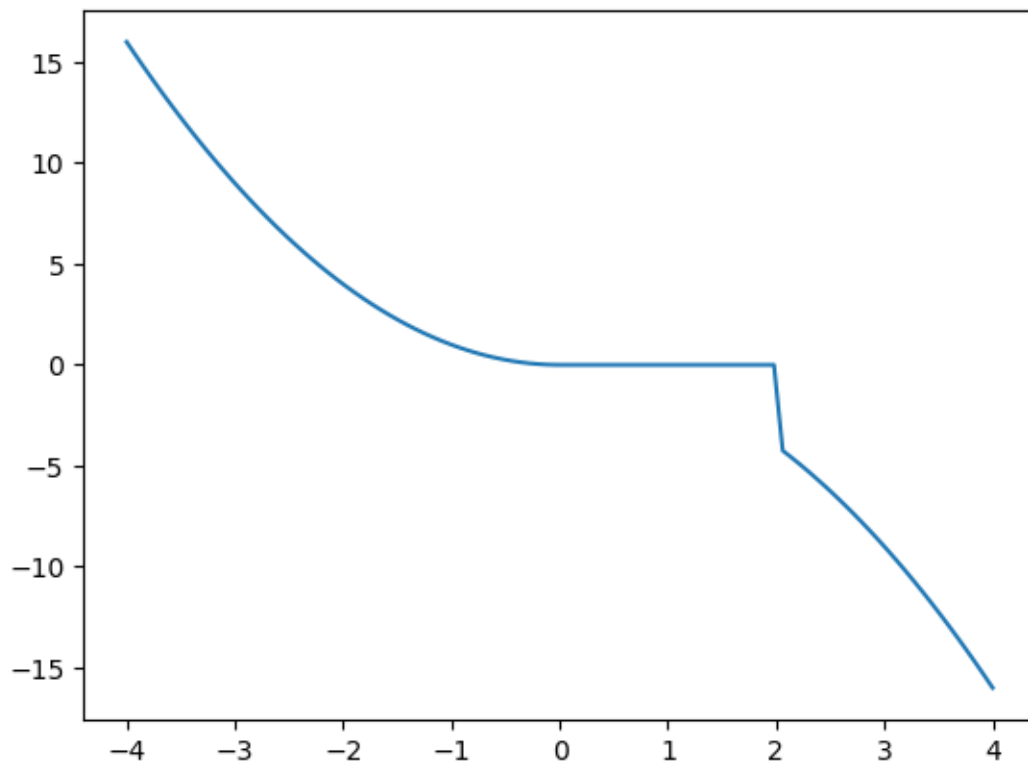
### 7.1.2 Diskontinuerlege funksjonar

Framgangsmåten over med `piecewise` kan brukast for funksjonar som er definert for alle  $x$ -verdiar mellom nederste og øverste del av definisjonsmengda. Om ein har ein funksjon med delt definisjonsmengde som td.

$$h(x) = \begin{cases} x^2 & \text{for } x \leq 0 \\ -x^2 & \text{for } x \geq 2 \end{cases}$$

må ein gjera tilpassingar. Prøver først måten me gjorde det over:

```
1 # definerer funksjonen
2 def h(x):
3     return np.piecewise(x, [x <= 0, x >= 2], [lambda x: x**2, lambda x: -x**2])
4
5
6 # finn x og y
7 x = np.linspace(-4, 4, 100)
8 y = h(x)
9
10 # plottar
11 plt.plot(x, y)
12 plt.show()
```

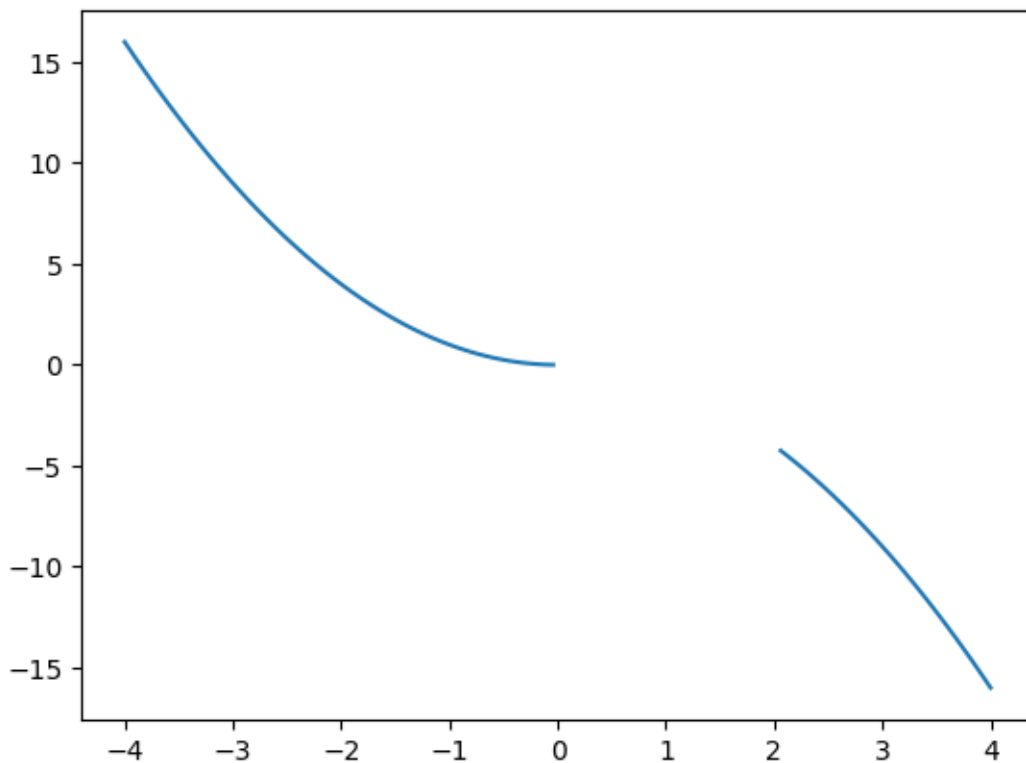


Her viser utfordringa med denne typen funksjonar. I staden for å teikna to kurver som ikkje heng saman, vert funksjonsverdien 0 når  $x \in \langle 0, 2 \rangle$ . I `piecewise` sin [dokumentasjon](#) finn me dette:

The output is the same shape and type as `x` and is found by calling the functions in `funclist` on the appropriate portions of `x`, as defined by the boolean arrays in `condlist`. **Portions not covered by any condition have a default value of 0.**

Måten å løysa det på er å definera kva som skal skje i intervallet der funksjonen ikkje er definert:

```
1 # definerer funksjonen (også mellom 0 og 2)
2 def h(x):
3     return np.piecewise(x, [x <= 0, (x > 0) & (x < 2), x >= 2], [lambda x: x**2, np.nan, 1
4
5
6 # finn x og y
7 x = np.linspace(-4, 4, 100)
8 y = h(x)
9
10 # plottar
11 plt.plot(x, y)
12 plt.show()
```



Her er intervallet  $(x > 0) \ \& \ (x < 2)$  definert ved funksjonen `np.nan` (*not a number*). På denne måten unngår me at funksjonsverdien vert sett til 0 automatisk i mellomrommet mellom dei to intervalla som utgjer definisjonsmengda.



## 8 Derivasjon

```
1 print("...")
```

```
...
```

## 9 Integrasjon

```
1 print("...")
```

```
...
```

# A Tips til programmeringa

Det er mange måtar ein kan skrive og kompilere Pythonkode på. Beste tips for programmering i matematikk er å bruka [Jupyter-notatbøker](#). Dette er filer (.ipynb) der ein kan kombinera små snuttar med Pythonkode og tekst (i markdown-format). Denne boka er stort sett basert på jupyter-filer. [Dette](#) er eit døme på korleis ei slik fil ser ut.

## A.1 Jupyter lab

Eit godt verktøy for å laga, endra og køyra jupyterfiler er Jupyter Lab. Dette programmet kan installerast på mange måtar. Mitt tips er å installera det gjennom *Miniconda*. Dokumentasjonen til programmet finn du [her](#)

## A.2 Miniconda

- 1) Gå inn på <https://docs.conda.io/en/latest/miniconda.html>

### Latest Miniconda Installer Links

Latest - Conda 23.5.2 Python 3.11.3 released July 13, 2023

Platform	Name	SHA256 hash
Windows	Miniconda3 Windows 64-bit	00e8370542836862d4c790aa8966f1d7344a8add4b766004febcb23f40e
	Miniconda3 Windows 32-bit	4fbc26c9c28b88beab16994bfba4829110ea3145baa60bda5344174ab65
macOS	Miniconda3 macOS Intel x86 64-bit bash	1622e7a0fa60a7d3d892c2d8153b54cd6ffe3e6b979d931320ba56bd5258
	Miniconda3 macOS Intel x86 64-bit pkg	2236a243b6cbe6f16ec324ecc9e631102494c031d41791b44612bbb6a7a1
	Miniconda3 macOS Apple M1 64-bit bash	c8f436dbde130f171d39dd7b4fca669c223f130ba7789b83959adc1611a3
	Miniconda3 macOS Apple M1 64-bit pkg	837371f3b6e8ae2b65bdfc8370e6be812b564ff9f40bcd4eb0b22f84bf9b

- 2) Last ned den nyaste installasjonsfila
- 3) Installer miniconda ved å køyra fila.
- 4) Opne **terminal** på Mac og **Anaconda Prompt** på Windows. (På Mac kan du opna spotlight og søka etter programmet, på Windows kan du søka i start-menyen).
- 5) I terminal/Anaconda Prompt skriv du desse kodelinjene (linje for linje)

```
1 conda config --add channels conda-forge
2 conda config --set channel_priority strict
3 conda update -n base -c defaults conda
4 conda install pandas matplotlib jupyterlab ipympl xarray python=3.11
5 conda install scipy sympy
```

6) Når du skal bruke Jupyter lab opnar du terminal/Anaconda Prompt og skriv inn

```
1 jupyter lab
```