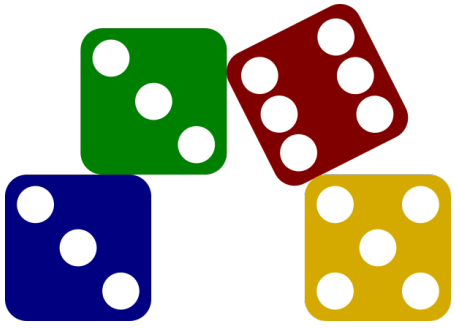
Kræsjkurs i Programmering

Opplegg i Matematikk: Tilfeldighet og simuleringer

kodeskolen simula

Tilfeldighet er kjempeviktig i programmering. Tenk deg for eksempel om du skal lage et spill, det hadde vært ganske begrenset hva slags spill du kunne laget om du ikke kan inkludere tilfeldige elementer. Det er en grunn til at brettspill gjerne inkluderer terninger og kort man stokker.

Men det er ikke bare for lek og morro at tilfeldighet er viktig, i programmeringsverden brukes tilfeldighet til en lang rekke bruksområder. For eksempel trenger man tilfeldighet for å *kryptere* informasjon. Kryptering handler om å kommunisere konfidensielt, og er utrolig viktig for å gjøre kommunikasjon over internett og mobilnettverk sikkert. I vitenskap og matematikk bruker vi også tilfeldighet mye, blant annet for å simulere det som skjer i naturen. Slike *datasimuleringer* kan også brukes for å hjelpe oss å forstå sannsynlighet og statistikk bedre.



Motivasjon

Når man jobber med sannsynlighet, tilfeldighet og statistikk i matematikken er det ofte at resultater er lite intuitive og vanskelige å utforske. Ved hjelp av programmering og simuleringer kan vi derimot rett og slett trekke tilfeldige utfall og *sjekke* hvordan ting utspiller seg. Dette kan man jo selvfølgelig gjøre for hånd og, man kan rulle terninger, flippe mynter, og gjøre eksperimenter. Derimot lar programmering oss gjøre dette i helt andre størrelsesordener. Å rulle en terning 1000 ganger for hånd tar timesvis, og det er lett å gjøre feil. I Python tar dette et par kodelinjer og fullfører på millisekunder—dette lar oss utforske problemstillinger på en helt ny måte.

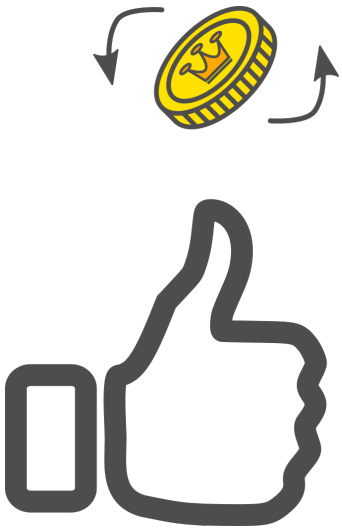
Innhold

1 Generere tilfeldige tall

Det første vi skal se på, er hvordan vi kan lage tilfeldige tall i programmene våres. Dette gjør vi ved å bruke et tileggbiblotek til Python som heter "random", som er engelsk og betyr *tilfeldig*. Fra dette bibloteket kommer vi til å importere et par forskjellige funksjoner, som gjør litt forskjellige ting.

1.1 Å flippe en mynt

Den første funksjonen vi skal se på heter `randint()`. Navnet er litt vanskelig, men *randint* er en sammensetning av ordene *random* og *integer*, som altså betyr *tilfeldig* og *heltall*. Funksjonen `randint` gir altså et tilfeldig heltall. Når vi kaller på randint må vi si *hvilke* tall som skal være mulig.



La oss som et eksempel se på myntkast, som for eksempel for å se hvem som starter en fotballkamp. Vi kaster en mynt i været, tar den imot, og ser hvilken side som havnet opp. Mynten har to sider, så det er to mulige utfall, som vi kaller *kron* eller *mynt*. For å gjenskape et myntkast på datamaskinen kan vi gjøre følgende

```
1 from random import randint
2
3 resultat = randint(1, 2)
4 print(resultat)
```

Her må vi først importere `randint` funksjonen. Deretter kaster vi en mynt og lagrer resultatet av kaster i en variabel vi kaller `resultat`, og så skriver vi den ut.

Obs: Merk at vi kommer til å måtte importere `randint` (og lignende funksjoner) for hvert eksempel vi skriver nedover. I våre eksempler dropper vi dette, for å spare litt plass.

Når vi kjører denne koden vil du se at du får skrevet ut enten `1` eller `2`. Hvilken av disse det blir er helt tilfeldig, og om du kjører koden på nytt flippes en ny mynt, der resultatet er helt uavhengig av forrige kjøring.

Vi sier at koden vår er et eksempel på myntkast, men resultatet vi får er enten 1 eller 2, ikke *kron* eller *mynt*. Det viktige er at vi har to mulige utfall, som begge er like sannsynlige, akkurat som i myntkast. Om vi ønsker kan vi bestemme oss for en tolkning av resultatet, for eksempel at 1 skal bety *kron* og 2 skal bety *mynt*.

Om vi ønsker at programmet skal faktisk skrive ut kron/mynt kan vi gjøre dette med en test som følger:

```
1 kast = randint(1, 2)
2
3 if kast == 1:
4     print("kron")
5 elif kast == 2:
6     print("mynt")
```

Her bruker vi rett og slett en test for å sjekke resultatet og "oversette" svaret fra tall til tekst.

Om vi nå kjører programmet vårt mange ganger på rad, så får vi et tilfeldig utfall hver gang

```
mynt
kron
kron
kron
kron
mynt
kron
kron
mynt
```

```
mynt
...
```

Nå kan man jo lure på hvorfor vi lærer å kaste mynt på datamaskinen, dette er jo noe vi fint klarer å gjøre for hånd? Vel, om vi skal kaste en mynt, eller ti, så klarer vi det fint for hånd. Men si at vi har lyst å kaste hundre mynter, eller tusen, eller en million. Da blir det fort kjedelig og masse jobb. I tilegg er det større sannsynlighet for feiltelling. På datamaskin derimot, så kan vi enkelt bruke en løkke for å repetere prosessen mange, mange ganger, og det går lynraskt. På den måten har vi mulighet til å gjøre flere eksperimenter.

1.2 Mange Myntkast

La oss si at vi ønsker å kaste 10 mynter på rad. Hvordan kan vi repetere koden vi skrev over 10 ganger? Å repetere noe gjør vi ved å skrive en løkke. Vi kan da legge hele koden over på innsiden av en løkke, og på den måten gjenta den:

```
for kast in range(10):
    kast = randint(1, 2)
    if kast == 1:
        print("kron")
    elif kast == 2:
        print("mynt")
```

Fordi hele koden har fått et innrykk hører den til løkka og vil repeteres. Merk spesielt at testene inne i løkka får et dobbelt innrykk.

Når programmet kjøres vil koden repetere, og vi får en rekke med mynt eller kron. Hver gang programmet kjøres får vi en ny, tilfeldig, rekke:

```
mynt
kron
mynt
mynt
...
```

Løkka gjentar seg nå 10 ganger, men dette kan vi enkelt øke ved å endre på tallet i `range()`, så vi kan nå enkelt flippe en mynt tusen ganger.

Derimot hjelper det oss ikke mye å få skrevet ut *mynt* eller *kron* tusen ganger til skjermen, da vil det forsatt ta en halv evighet å gå igjennom og telle opp hvor mange vi fikk av hver. Vi bør derfor la programmet holde tellingen for oss.

Istedenfor å skrive ut hva hvert enkelt kast ble, så ønsker vi at bare det totale antallet mynt og kron skal skrives ut til slutt. Dette gjør vi ved å istedenfor å skrive ut inne i if-testene, øke en *tellevariabel*, som øker hver gang det kommer en kron eller en mynt:

```
antall_kron = 0
antall_mynt = 0

for i in range(100):
    kast = randint(1, 2)
    if kast == 1:
        antall_kron += 1
    elif kast == 2:
        antall_mynt += 1

print(f"Kron: {antall_kron}")
print(f"Mynt: {antall_mynt}")
```

```
Kron:  41
Mynt:  59
```

Her er det en god idé å sjekke at det totalet antallet til slutt ble riktig: $41+59 = 100$. Dette blir en liten kontroll på at programmet vi har skrevet blir riktig.

Det vi kan gjøre i tilegg til å skrive ut antallet av hvert utfall, er å skrive ut *andelen* vi har fått. Isåfall bør vi huske på antall kast vi har utført totalt, så vi oppretter dette som en variabel. I tilegg til å skrive ut totalene regner vi nå ut andelene som en prosent:

```
from random import randint
antall_kast = 1000

# Tellevariabler
antall_kron = 0
antall_mynt = 0

```

```
for i in range(antall_kast):
    kast = randint(1, 2)
    if kast == 1:
        antall_kron += 1
    elif kast == 2:
        antall_mynt += 1

andel_kron = antall_kron/antall_kast
andel_mynt = antall_mynt/antall_kast

print(f"Antall kast: {antall_kast}")
print(f"Kron: {antall_kron} ({andel_kron:.1%})")
print(f"Mynt: {antall_mynt} ({andel_mynt:.1%})")
```

```
Antall kast: 1000
Kron: 488 (48.8%)
Mynt: 512 (51.2%)
```

Her så lager vi en variabel for antall kast. I løkken vår gjentar vi nå prosessen for det antallet vi har satt i variabelen ved å skrive `range(antall_kast)`. Dette gjør vi fordi vi skal regne ut andelen senere, om vi skriver inn et gitt tall begge steder, f.eks, 1000, så må vi isåfall huske å endre alle tilfeller av det tallet senere om vi skal endre det ett sted, ellers blir alt feil. Det er da bedre å bruke en variabel, så vi bare må sette, og endre den, et annet sted.

I tilegg regner vi ut andelen kron ved utregningene

```
andel_kron = antall_kron/antall_kast
andel_mynt = antall_mynt/antall_kast
```

Merk at dette vil bli tall mellom 0 og 1. Disse må vi vanligvis regne om til prosent selv ved å gange med 100%. Her kan vi alternativt skrive `:%` i strengen vår når vi fletter inn variabelen, da sier vi til Python at vi ønsker det skrevet ut som en prosent. Da ganges automatisk tallet med 100% før det skrives ut.

1.3 Rulle terning

Istedenfor å flippe en mynt, kan vi istedet for eksempel rulle en terning. Det finnes mange forskjellige typer terninger, men her mener vi altså en vanlig, seks-sidet

terning.

For å trille en terning i koden vår, kan vi igjen bruke `randint`, men denne gangen må vi ha seks mulige utfall. For å rulle en terning gjør vi da

```
terningkast = randint(1, 6)
print(terningkast)
```

Som før så vil `randint` nå gi oss et tilfeldig tall fra og med 1 til og med 6. I motsetning til myntkastet er det derimot helt rimelig å få tall som utfall.

Som før kan vi nå også rulle mange terninger på rad med en løkke. Vi kan for eksempel finne *summen* av 5 terningkast som følger:

```
total = 0
for kast in range(5):
    total += randint(1, 6)
print(total)
```

1.4 Estimere Sannsynlighet

Når vi så på myntkast skrev vi ut *andelen* kron, og *andelen* mynt vi fikk totalt. Dette ble veldig nært 50 % på hver av dem. Det er selvfølgelig ingen tilfeldighet, men det blir tilfellet fordi sannsynligheten for hvert av utfallene er 50 %.

Hvis man gjentar en tilfeldig prosess mange ganger. For eksempel å flippe en mynt, eller å rulle en terning. Så vil hvert utfall forekomme like mye som sannsynligheten for det utfallet. Dette er kun sant hvis vi gjentar prosessen mange, mange ganger. Om vi ruller en terning seks ganger vil vi ikke få en av hvert tall.

Vi kan snu dette konseptet på hodet, og bruke det til å *estimere* sannsynlighet. Om vi har en prosess der vi ikke kjenner til sannsynligheten for et utfall, så kan vi rett og slett gjenta øvelsen mange ganger og telle opp. Da vil andelen av utfallet være en estimert sannsynlighet. Jo fler gjentagelser vi gjør, jo bedre. Dette kan vi kalle for en *simulering*.

Hvis man kjenner til alle utfallene i en situasjon, og de er like sannsynlige, så sier vi at man regner ut sannsynligheten med formelen

$$\text{sannsynlighet} = \frac{\text{antall gunstige utfall}}{\text{antall mulige utfall}}.$$

Dette er en av de mest grunnleggende formlene i sannsynlighetsregningen.

Om vi ønsker å estimere en sannsynlighet gjennom eksperimenter og simuleringer, kan vi erstatte denne formelen med

$$\text{estimert sannsynlighet} = \frac{\text{antall gunstige simuleringer}}{\text{antall simuleringer totalt}}.$$

Estimatet vårt vil være bedre jo fler simuleringer vi gjør.

1.5 Eksempel: 20 eller mer

La oss gå tilbake til eksempelet vårt der vi regnet ut summen av øynene på 5 terninger. La oss nå stille spørsmålet:

- Hvis du ruller 5 terninger og summerer dem sammen, hva er sannsynligheten for å få 20 eller mer?

Dette er et forholdsvis enkelt spørsmål, det er ihvertfall enkelt å skjønne hva det blir spurt om. For å gå frem for å svare på det derimot, er litt mer vrient. For to terninger er det enkelt å skrive opp en tabell av alle mulige utfall, men med 5 terninger får vi plutselig:

$$6^5 = 7776,$$

mulige utfall! Det blir fort slitsomt og rotete å skulle gå igjennom og finne ut hvilke av disse som summerer til 20 eller mer. Det er langt fra umulig, men det er mye jobb.

La oss istedet *estimere* sannsynligheten med datasimuleringer. Vi har allerede sett hvordan vi kan kaste 5 terninger og regne ut summen:

```
total = 0
for kast in range(5):
    total += randint(1, 6)
print(total)
```

Det vi nå ønsker å gjøre er å gjenta denne prosessen mange, mange ganger. Derfor lager vi en ny løkke som gjentar prosessen med å kaste 5 terninger og legge sammen:

```
from random import randint

antall_simuleringer = 1000
antall_gunstige = 0

for simulering in range(antall_simuleringer):
    # Kast 5 terninger og regn ut summen
    total = 0
    for kast in range(5):
        total += randint(1, 6)

    # Sjekk om summen er over 20
    if total >= 20:
        antall_gunstige += 1

sannsynlighet = antall_gunstige/antall_simuleringer

print(f"Antall simuleringer: {antall_simuleringer}")
print(f"Antall >= 20: {antall_gunstige}")
print(f"Estimert sannsynlighet: {sannsynlighet:.1%}")
```

Når vi kjører programmet vårt gjentar Python eksperimentet vårt 1000 ganger, og forteller oss hvor ofte vi fikk 20 eller mer:

```
Antall simuleringer: 1000
Antall >= 20: 320
Estimert sannsynlighet: 32.0%
```

Om vi kjører dette programmet flere ganger ser vi at sannsynligheten blir litt ulik hver gang. Dette er fordi vi gjør *tilfeldige* eksperimenter og *estimerer* sannsynligheten, det er ikke en perfekt prosess. Vi ser derimot at den holder seg rundt 30 %, og varier et prosent poeng eller to opp eller ned fra denne verdien.

Det at den varierer såpass for hver kjøring forteller oss at vi kanskje gjør litt få simuleringer. Vi kan derfor øke til litt fler, og få et bedre estimat. Datamaskinen gjør dette veldig raskt, så la oss prøve en million

```
Antall simuleringer: 1000000
Antall >= 20: 305397
Estimert sannsynlighet: 30.5%

Antall simuleringer: 1000000
Antall >= 20: 304814
Estimert sannsynlighet: 30.5%

Antall simuleringer: 1000000
Antall >= 20: 305065
Estimert sannsynlighet: 30.5%
```

Nå kjører vi programmet tre ganger. Det bruker nå ca 5 sekunder å kjøre seg ferdig, så vi merker at selv datamaskinen har en del arbeid å få unnagjort her. I hvert tilfelle er det nøyaktige antallet som blir ≥ 20 ganske forskjellig, men *andelen*, som gir den estimerte sannsynligheten er nå stabil ned til første desimal. Merk at vi her skriver ut kun én desimal, fordi vi har skrevet `:.1%` i flettingen vår.

Vi har altså klart å finne ut at sannsynligheten for å få 20 eller mer når vi ruller 5 terninger, er omtrent 30.5 %. I akkurat dette tilfelle kunne vi klart å finne ut dette for hånd også, det hadde bare tatt mye mer tid. I andre tilfeller er det ikke mulig å finne en sannsynlighet med penn og papir, men å estimere det gjennom eksperimenter og sannsynlighet er fortsatt mulig.

1.6 Diskusjon: Partibarometer

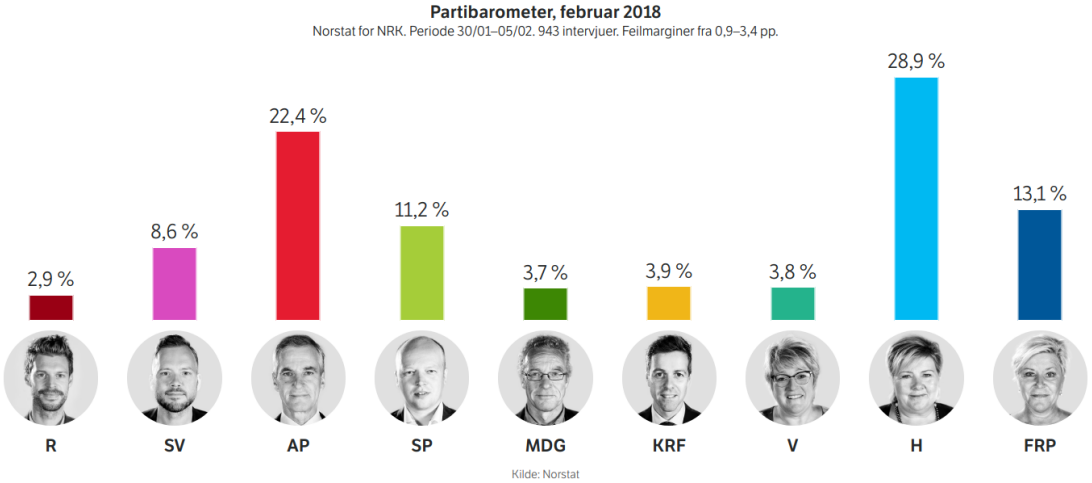
Når det er valgår blir det stadig publisert nye partibarometre som prøver å si hva folket kommer til å stemme. Disse er basert på spørreundersøkelser, der de har spurt tilfeldige mennesker om hva de tenker å stemme.

Et partibarometer prøver å estimere andelen av befolkningen som skal stemme hvert parti, og dette er egentlig akkurat det samme som vi har gjort for å estimere sannsynlighet. Hver person man spør, blir det vi over kaller *én* simulering.

Partibarometre er som oftest basert på å spurt omtrent 1000 personer. Derimot er Norge et demokrati, der antall med stemmerett til stortingsvalg nærmer seg 4 millioner (Kilde: SSB). Ett tusen ut av 4 million svarer til 0.025%! Når man spør så få av de stemmeberettige, hvordan i all verden kan vi forvente å få et godt estimat?

I eksempelet vårt over, så vi at ved å gjennomføre 1000 simuleringer, fikk vi variasjoner i estimatet vårt på et par prosentpoeng. Det viser seg at man i spørreundersøkelser kan forvente omtrent de samme variasjonene i estimatene når man spør ca 1000 mennesker.

I en god spørreundersøkelse eller partibarometer bør det alltid oppgis hvor mange som er spurt, og det de kaller *feilmarginer*. Se for eksempel resultatene under, som kommer fra NRKs Partibarometer, utført av Norstat:



I toppen av bildet står det at de har gjennomført 934 intervjuer, og har feilmarginer på 0.9-3.4 prosentpoeng.

For å regne seg frem til feilmarginene må man bruke litt mer fancy matematikk og statistikk enn vi har sett på her, men idéen er helt lik som for eksperimentet vårt tidligere. Basert på hvor mange simuleringer eller målinger vi gjør, jo sikrere kan vi være på svaret vi finner.

I partibarometrene kunne de vært sikrere på estimatene sine, om de spurte flere mennesker. Så hvorfor gjør de ikke det? Dette er det vi kaller en nytte-kostnadsanalyse. Å intervju 2000 mennesker er dobbelt så mye jobb som å intervjue 1000, men feilmarginene vil ikke krympe så voldsomt mye. Derfor sier de fleste som gjennomfører spørreundersøkelser seg fornøyd med en feilmargin på et par prosent. Derimot har vi en sperregrense på 4% ved stortingsvalg, og i akkurat denne undersøkelsen er det 3 partier som ligger 0.1–0.3 % under denne grensen. Her vil en feilmåling på rundt 1 prosentpoeng være enormt viktig!

1.7 Trekke desimaltall

Så langt har vi bare sett på det å trekke heltall, det var derfor funksjonen het `randint`, fordi *int* står for integer, som betyr heltall.

Derimot har vi av og til lyst å trekke desimaltall. Her finnes det flere måter å trekke tall på, man kan f.eks trekke *normalfordelte* tall, men dette er langt over ungdomsskolematematikken, så det trenger vi ikke å tenke på her.

Istedet bruker vi funksjonen `uniform`. Denne funksjonen tar en minimumsverdi, og en maksimumsverdi, og gir oss en tilfeldig verdi mellom de to ytterpunktene. Det blir litt som å velge et tilfeldig punkt på tallinja mellom to ytterpunkter:

```
1 from random import uniform
2
3 punkt = uniform(-2, 2)
4 print(uniform)
```

```
1.5174108539579518
```

Funksjonen heter *uniform* fordi det er en uniform sannsynlighet over hele intervallet. Det betyr at det er like sannsynlig å få et hvilket som helst tall mellom de to ytterpunktene vi velger.

Vi skal bruke `uniform` til et matematisk eksperiment i slutten av dette kompendiet. For nå viser vi et litt enklere eksempel.

Eksempel: Dyrehage

Vi ønsker å lage et spill der man skal drive en dyrehagene. Alle dyrene i dyrehagen har en gitt vekt, som vil øke etterhvert som dyrene vokser og spiser. Når det blir født nye dyr, må de få en gitt startvekt, og vi ønsker at denne startvekten skal bli tilfeldig generert.

Dette kan være spennende, for startvekten kan for eksempel bestemme hvor mye stell dyrene trenger og hvor fort de vokser. La oss si vi skal finne startvekten til en løvebaby. Et kjapt nettsøk forteller oss at en løvebaby veier typisk 1.5 kg når den blir født. Vi vil ha litt variabilitet i spillet vårt, så vi bestemmer oss for å trekke en vekt uniformt mellom 0.75 og 3 kg i spillet vårt. Da gjør vi som følger:

```
fødselsvekt = uniform(0.75, 3.0)
print("Gratulerer! En ny løve er blitt født.")
print(f"Fødselsvekten er {fødselsvekt:.2f} kg.")
```

```
Gratulerer! En ny løve er blitt født.
Fødselsvekten er 2.68 kg.
```

Akkurat denne løvebabyen havnet altså litt på den tyngre siden! Dette kan ha konsekvenser senere i spillet vårt.

2 Andre former for tilfeldighet

Så langt har vi sett på hvordan vi kan generere tilfeldige heltall med `randint`. La oss nå se kjapt på et par andre former for tilfeldighet vi kan generere med bibloteket `random`.

2.1 Choice

Ordet "choice" betyr *valg*, og funksjonen `choice` gjør et tilfeldig valg for oss. Si for eksempel at vi har 5 personer som er med i trekningen for et par kinobiletter. Her må vi *velge* én person tilfeldig. Det kan vi gjøre med `choice`.

For å bruke `choice` må vi bruke en *listevariabel*, disse har vi ikke dekket tidligere i dette kurset, men de er forholdsvis greie å jobbe med. Vi viser først eksempelet, så forklarer vi:

```
from random import choice
personer = ["Anders", "Beate", "Christine",
            "Daniel", "Erika"]
vinner = choice(personer)
print(vinner)
```

Første linje importerer `choice`-funksjonen. Den andre kodelinjen definerer en *liste* med personer. Her bruker vi firkantparenteser (`[]`) for å si at vi skal lage en liste,

inne i lista deler vi hver ting i lista med komma. Her har vi en liste med 5 personer, så da bruker vi komma mellom hvert navn.

På den tredje linja skriver vi `choice(personer)`, det betyr at vi skal plukke ut én person tilfeldig fra list. Vi lagrer den personen som ble trukket i en variabel vi kaller `vinner`. Til slutt skriver vi ut vinneren.

Alle personene i lista har samme sannsynlighet for å bli valgt, og hver gang vi kjører programmet vil valget gjøres tilfeldig, og uavhengig av de andre kjøringene.

2.2 Sample

Funksjonen `choice` trekker én ting fra en liste med ting. Funksjonen `sample` gjør det samme, men vi kan nå trekke flere elementer. Ordet *sample* betyr å hente inn prøver. Dette virker kanskje som et litt forvirrende navn, men om vi f.eks tenker på spørreundersøkelsene vi diskuterte tidligere, så er det å velge ut og stille tusen tilfeldige mennesker hva de skal stemme et eksempel på en *sampling*, eller å ta tusen prøver.

Men, uavhengig av hva navnet betyr, så kan vi altså bruke `sample` til å gjøre et utvalg med mer enn ett element. Vi kan for eksempel utvide forrige eksempel til å trekke to vinnere:

```
from random import vinnere
personer = ["Anders", "Beate", "Christine",
            "Daniel", "Erika"]
vinnere = sample(personer, 2)
print(vinnere)
```

```
['Daniel', 'Erika']
```

Merk at funksjonen `sample` gjør et utvalg *uten tilbakelegging*. Vi kan altså være sikker på at én person ikke vil få begge premiene. Om vi ønsker å gjøre utvalg *med* tilbakelegging må du bruke `choice` flere ganger på rad.

Vi kan for eksempel trekke lottonummere. I Vikinglotto trekkes det 6 tall fra 48 tall. Det kan vi gjøre som følger

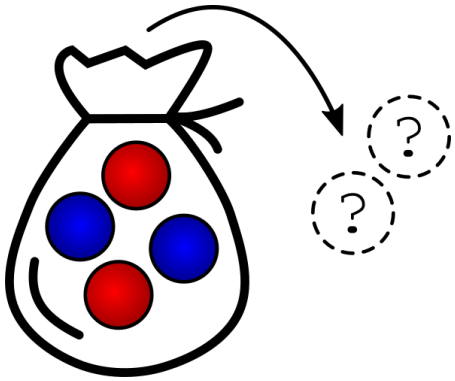
```
lottotall = sample(range(1, 49), 6)
print(lottotall)
```

Her spesifiserer vi at vi skal trekke 6 tall, fra tellrekka $1, \ldots 48$. Resultatet blir 6 lottotall:

```
[39, 42, 23, 41, 3, 32]
```

2.3 Eksempel: To røde, to blå

La oss si vi har en sekk med to røde og to blå baller i seg. Hvis du stikker hånden din inn, og trekker ut to baller uten å se. Hva er sannsynligheten for å få én rød, og én blå ball? Altså i motsetning til å trekke begge de røde, eller begge de blå ballene.



Dette er en morsom oppgave å diskutere, for det er litt lett å snuble å møte på feil svar. Her kan man jo for eksempel skrive opp de fire utfallene våres: rød-rød, rød-blå, blå-rød, blå-blå. Om vi bruker formelen antall gunstige på antall mulige vil vi da tro at svaret er 50 %. Men det er faktisk feil i dette tilfellet, fordi de fire utfallene er ikke like sannsynlige.

La oss lage et kort program som estimerer sannsynligheten ved å simulere prosessen. Da bruker vi `sample` for å trekke to baller fra sekken. Så vi kan lage en sekk og trekke to baller ved å gjøre:

```
1 pose = ["rød", "rød", "blå", "blå"]
2 utvalg = sample(pose, 2)
3 print(utvalg)
```

Dette gjennomfører én simulering, men for å estimere sannsynlighet må vi gjenta dette mange ganger, og holde tellingen må antallet der vi får én rød og én blå ball. Programmet kan da for eksempel bli som dette:

```python
from random import sample
antall_simuleringer = 100000
antall_gunstige = 0
pose = ["rød", "rød", "blå", "blå"]

for simulering in range(antall_simuleringer):
    utvalg = sample(pose, 2)
    if utvalg == ["rød", "blå"] or utvalg == ["blå", "rød"]:
        antall_gunstige += 1

sannsynlighet = antall_gunstige/antall_simuleringer
print(f"Antall simuleringer: {antall_simuleringer}")
print(f"Antall gunstige: {antall_gunstige}")
print(f"Estimert sannsynlighet: {sannsynlighet:.1%}")
```

En kjøring av programmet vårt gir oss nå:

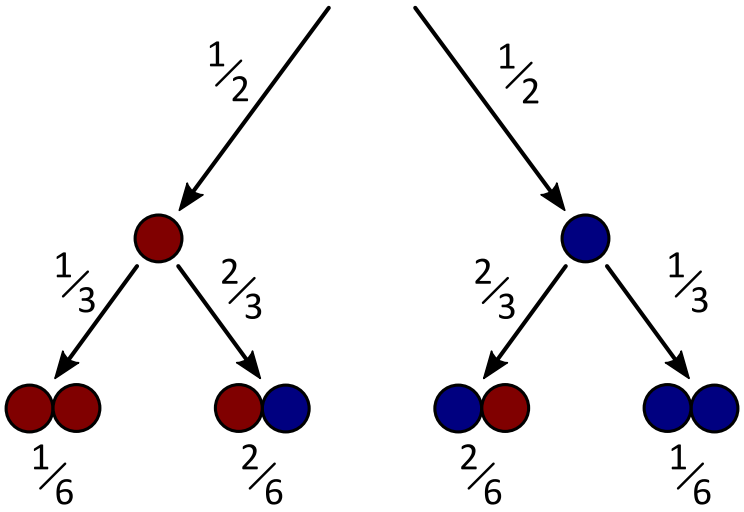
```
Antall simuleringer: 100000
Antall gunstige: 66678
Estimert sannsynlighet: 66.7%
```

Altså estimerer vi sannsynligheten til å være 66.7 %, som betyr 2/3, ikke 50% slik det er lett å tro.

Grunnen til at sannsynligheten ikke blir 50/50 er at man trekker to baller *uten tilbakelegging*. Om man ser for seg at man først trekker 1 ball, og så en til, så blir det kanskje litt mer tydelig.

På den første ballen man trekker er det like sannsynlig å trekke en rød eller en blå ball. Men si at man først trekker en rød ball, da er det jo to blå igjen, men bare én rød igjen. Slik at da blir sannsynligheten for å trekke en blå større. Tilsvarende er det om vi trekker en blå først, da vil den andre ha større sjans for å være rød.

Dette kan illustreres som et utfallstre



2.4 Shuffle

Den neste funksjonen vi ser på er `shuffle`. Ordet *shuffle* betyr å stokke om på, som for eksempel å stokke om på en kortstokk. Om vi bruker shuffle på en liste, så vil lista havne i en tilfeldig rekkefølge.

Si for eksempel at vi skal velge en tilfeldig rekkefølge på elevene i klassen som skal fremføre, eller kanskje tilfeldige sitteplasser. Da kan vi først lage en liste over alle elevene, så stokke om på lista med `shuffle`, og så til slutt skrive ut lista:

```
1 from random import shuffle
2
3 klasse = ["Andreas", "Beate", "Christine", "Daniel", "Egil
    ", "Fredrik", "Grethe", "Henriette", "Ida", "Joakim", "
    Lise", "Maria", "Nils", "Olivia", "Pål"]
4
5 shuffle(klasse)
6
7 print(klasse)
```

```
['Nils', 'Olivia', 'Henriette', 'Joakim', 'Lise', 'Andreas',
```

```
'Beate', 'Fredrik', 'Daniel', 'Egil', 'Pål', 'Ida', 'Grethe',
'Maria', 'Christine']
```

Vi har ikke vist det tidligere, men her kan vi også nevne at man kan skrive ut listen på en litt finere måte med en for-løkke:

```
1 for elev in klasse:
2     print(elev)
```

```
Nils
Olivia
Henriette
...
```

Her bruker vi rett og slett en for-løkke til og iterere over en liste med navn, istedenfor en tallrekke. Vi skriver dermed ut et navn per linje, istedenfor å skrive dem ut som en stor bolk.

3 Spill og Morro

Så langt har vi sett på en del forskjellige funksjoner fra `random`-bibloteket for å generere tilfeldighet i programmene våres. Vi har også sett litt på hvordan man kan bruke dette til å gjøre eksperimenter og estimere sannsynlighet. Dette skal vi komme tilbake til mot slutten av kompendiet, men først skal vi bruke tilfeldighet til noe litt annet, nemlig litt spill og morro.

3.1 Eksempel: Gangetabellquiz

Sist uke så vi hvordan vi kunne lage en enkel quiz. Nå kan vi bruke det vi lærte da, kombinert med det vi har lært om tilfeldighet nå, til å lage en gangetabell quiz.

La oss begynne med å lage ett spørsmål. Først bruker vi `randint` for å trekke to tilfeldige tall mellom 1 og 10:

```
1 from random import randint
2
```

```
a = randint(1, 10)
b = randint(1, 10)
```

Vi må nå stille brukeren spørsmålet hva $a \cdot b$ er. Siden vi skal sammenligne svaret med en tallvariabel, må vi huske å gjøre om svaret fra brukeren til et tall.

```
svar = int(input(f"Hva er {a}x{b}? "))
```

Nå kan vi sjekke om svaret brukeren har oppgitt stemmer:

```
if svar == a*b:
    print("Riktig!")
else:
    print(f"Feil. ({a} x {b} = {a*b}))
```

Nå kan vi sjekke at programmet vårt fungerer som det skal. Da bør vi sjekke begge mulighetene:

```
Hva er 8 x 1? 8
Riktig!

Hva er 8 x 7? 52
Feil. (8 x 7 = 56)
```

Nå som vi vet at ett spørsmål fungerer kan vi legge til flere runder. Si for eksempel at vi skal spille 10 runder og holde styr på antall riktige:

```
from random import randint

riktige = 0

for spørsmål in range(10):
        a = randint(1, 10)
        b = randint(1, 10)

        svar = int(input(f"Hva er {a} x {b}? "))

        if svar == a*b:
            print("Riktig!")
            riktige += 1
```

```
        else:
            print(f"Feil. ({a}x{b} = {a*b})")

print(f"Du fikk {riktige} av 10 rette.")
```

I tilegg kan vi nå skrive ut en ekstra beskjed på slutten basert hvor mange riktige brukeren fikk.

Utvidelser

Dette programmet kan være et fint verktøy om man prøver å lære seg gangetabellen, men på ungdomsskolen kan forhåpentligvis de fleste av elevene gangetabellen fra før. Men man kan her utvide eksempelet for å gjøre det mer spennende. Kanskje trekke tall fra 1 til 20 isteden, så man kan få spørsmål som f.eks $13 \cdot 17$, som kan være langt mer utfordrende. En annen mulighet er å legge inn forskjellige regneoperasjoner. Da kan vi f.eks bruke `choice` og if-tester slik at datamaskinen velger tilfeldig mellom addisjon, subtraksjon, gange, dele og potensregning.

Tidtaking

Vi kan også bygge inn at spillet tar tiden på brukeren, for å gjøre det mer spennende. Dette har vi ikke dekket i kurset, men idéen er ganske grei. Fra bibloteket *time* kan vi importere funksjonen *time*. Denne funksjonen gir den nåværende tiden i sekunder etter 1. januar 1970. Om vi sjekker tiden før og etter brukeren er ferdig med å svaret på alle spørsmålene vil differansen være tiden de brukte

```
from time import time

start = time()

# Resten av koden

slutt = time()
print(f"Du brukte {slutt-start:.1f} sekunder")
```

Etter vi har lagt til dette kan vi nå ha konkurranser. To personer kjører koden likt. Det er om å gjøre å få flest rette, men dersom begge klarer like mange er det den som var raskest som vinner.

```
Du fikk 10 av 10 rette.
Du brukte 13.4 sekunder.
```

3.2 Eksempel: Gjettespillet Over/Under

Over/under er en enkel gjettelek to personer kan leke, som også egner seg godt til å programmere.

Frakoblet

Over/under lekes ved at man settes sammen i par. Person A tenker på et tilfeldig tall mellom 1 og 1000 og skriver det ned på et ark, så den andre personen ikke kan se det.

Deretter skal person B prøve å gjette seg frem til tillet. Etter hvert forslag B gir, så skal person A si om gjettet var over, under, eller helt riktig. På denne måten får B mer og mer informasjon for hvert gjett, og kan peile seg inn til riktig svar.

Etter at person B gjetter helt riktig tall, bytter man roller. Denne gangen skal B finne på et tall, og A gjette seg frem til det. Målet er å komme frem til riktig tall på færrest mulig gjett.

Om man leker dette i klasserommet kan man for eksempel la elevene leke tre runder hver. For hver runde bør de skrive ned hvor mange gjett de brukte. Den eleven som klarte å få til alle tre rundene på færrest gjett totalt er vinneren.

På datamaskin

La oss nå prøve å kode opp dette spillet. Først må vi la datamaskinen nå trekke et tall mellom 1 og 1000 tilfeldig. Dette kan vi gjøre som følger:

```
1 fasitsvar = randint(1, 1000)
2 print("Jeg har tenkt på et tall mellom 1 og 1000.")
3 print("Prøv å gjett det!")
4 print()
```

Merk datamaskinen her vil trekke et tilfeldig tall, men ikke skrive det ut, så det

er hemmelig, akkurat som når man skriver ned tallet man velger på lappen som man ikke viser frem. Etter dette skriver vi ut instrukser til brukeren. Når vi skriver `print()` uten nå beskjed i parentesene så printer vi en blank linje.

Vi skal nå gjenta den samme prosessen mange ganger: nemlig la brukeren gjette. For å gjenta noe mange ganger bruker vi en løkke. Når vi bruker en for-løkke må vi si hvor mange ganger vi løkker, så la oss for eksempel gi brukeren 100 gjett. For hver løkke må vi spørre brukeren om et gjett, og huske å konvertere svaret fra en tekstvariabel til en tallvariabel:

```
for gjett in range(1, 101):
    svar = int(input("Gjett: "))
```

Etter brukeren gir oss et svar ønsker vi nå å sjekke om det er: for lavt, for høyt, eller helt riktig. Dette kan vi gjøre med en if-test. Først kan vi sjekke mindre-enn og større-enn:

```
if svar > fasitsvar:
    print(f"Ditt gjett på {svar} er for høyt.")
elif svar < fasitsvar:
    print(f"Ditt gjett på {svar} er for lavt.")
```

Til slutt vil vi sjekke om de treffer helt riktig. I såfall ønsker vi å skrive ut hvor mange gjett de har brukt totalt sett. Vi ønsker også isåfall å bruke `break`, for da bryter vi løkka:

```
elif svar == fasitsvar:
    print("Helt riktig!")
    print(f"Du brukte {gjett} gjettinger!")
    break
```

Hva skjer om brukeren bruker opp alle 100 gjettene sine? Isåfall er løkka ferdig, og de får ikke lov til å gjette mer. Isåfall bør vi kanskje skrive ut en beskjed. Det kan vi gjøre ved å ha en elseblokk som hører til løkka vår:

```
else:
    print(f"Der har du prøvd 100 ganger og gått tom for
       gjett! Riktig svar er {fasitsvar}")
```

Her bruker vi altså en for-else, eller på Norsk, en for-ellers løkke. Det som står i else skjer bare om brukeren *ikke* finner riktig svar.

Hele programmet blir nå

```
from random import randint

fasitsvar = randint(1, 1000)
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
print()

for gjett in range(1, 101):
    svar = int(input("Gjett: "))

    if svar > fasitsvar:
        print(f"Ditt gjett på {svar} er for høyt.")
    elif svar < fasitsvar:
        print(f"Ditt gjett på {svar} er for lavt.")
    elif svar == fasitsvar:
        print("Helt riktig!")
        print(f"Du brukte {gjett} gjettinger!")
        break
else:
    print(f"Der har du prøvd 100 ganger og gått tom for
       gjett! Riktig svar er {fasitsvar}")
```

En kjøring av programmet gir nå for eksempel

```
Jeg har tenkt på et tall mellom 1 og 1000.
Prøv å gjett det!

Gjett: 400
Ditt gjett på 400 er for lavt.
Gjett: 700
Ditt gjett på 700 er for høyt.
Gjett: 600
Ditt gjett på 600 er for høyt.
Gjett: 500
Ditt gjett på 500 er for høyt.
Gjett: 450
Ditt gjett på 450 er for lavt.
Gjett: 475
Ditt gjett på 475 er for høyt.
```

```
Gjett: 465
Ditt gjett på 465 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 462
Ditt gjett på 462 er for høyt.
Gjett: 459
Ditt gjett på 459 er for høyt.
Gjett: 457
Ditt gjett på 457 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 456
Helt riktig!
Du brukte 13 gjettinger!
```

Strategi

Bare det å kode opp over/under er en god øvelse for å lære programmering og algoritmisk tankegang, det kan også være morsomt. Derimot kan vi gå et steg lenger, og nå tenke på hvordan vi bør *spille* dette spillet for å gjøre det best mulig. Altså, hvilken strategi skal vi velge?

Her kan vi for eksempel velge å bruke *midtpunktsmetoden*, der vi alltid velger midtpunktet av det intervallet vi har igjen. Da starter vi altså med å gjette 500. Om vi er for lave går vi opp til 750, om vi er for høye går vi ned til 250.

Midtpunktsmetoden er en god strategi, fordi uansett om vi er for høye eller lave kan vi eliminere halve det intervallet som er igjen. Et annet godt navn for denne strategien er *halveringsmetoden*.

Her kan det for eksempel være en god øvelse å regne seg frem til hvor mange gjett man trenger for å *garantert* komme frem til riktig svar med midtpunktsmetoden, som blir

$$\log_2(1000) \simeq 10.$$

Her kan det nevnes at midtpunktsmetoden ikke bare er en god strategi for over-/under leken, men en god matematisk metode for å løse matematiske ligninger. Men dette er mer VGS pensum, så vi lar det ligge.

3.3 Eksempel: Stein, saks, papir

Et klassisk spill å programmere opp er stein, saks, papir. Her tenker man seg kanskje at dette spiller ikke har noen *tilfeldighet* i seg. Men poenget er at vi skal lage spillet slik at man spiller mot datamaskinen, og da vil vi at datamaskinen skal velge stein, saks, og papir, tilfeldig.

Vi kan få datamaskinen til å velge ett av de tre alternativene med `choice`-funksjonen:

```
from random import choice
datamaskin = choice(["stein", "saks", "papir"])
```

Så må vi spørre brukeren om hva de velger. Da bruker vi `input`. Her er det litt viktig at de svarer et av de gyldige svarene nøyaktig, så vi skriver dem ut i spørsmålet:

```
bruker = input("Velg stein, saks, eller papir: ")
```

Etter spilleren har gjort valget sitt kan vi skrive ut de to valgene:

```
print(f"Du valgte: {bruker}")
print(f"Datamaskinen valgte: {datamaskin}")
```

Det som gjenstår da er å bruke if-tester til å finne ut hvem som vant. Dette blir rotete, siden man må sjekke alle valg. Om man kunne en del mer Python finnes det måter å få til dette mye lettere, men med det vi har lært skriver vi det rett og slett ut:

```
if bruker == "stein":
    if datamaskin == "saks":
        print("Du vinner!")
    elif datamaskin == "papir":
        print("Du taper!")
    else:
        print("Uavgjort!")

elif bruker == "saks":
    if datamaskin == "papir":
        print("Du vinner!")
    elif datamaskin == "stein":
        print("Du taper!")
    else:
```

```
15          print("Uavgjort!")
16
17  elif bruker == "papir":
18      if datamaskin == "stein":
19          print("Du vinner!")
20      elif datamaskin == "saks":
21          print("Du taper!")
22      else:
23          print("Uavgjort!")
24
25  else:
26      print("Jeg skjønner ikke valget ditt, så jeg kan ikke
           kåre en vinner.")
```

Den siste `else`-blokken slår inn om brukeren ikke skrev inn et av valgene: stein, saks, eller papir, helt nøyaktig.

```
Velg stein, saks, eller papir: saks
Du valgte: saks
Datamaskinen valgte: saks
Uavgjort!

Velg stein, saks, eller papir: saks
Du valgte: saks
Datamaskinen valgte: papir
Du vinner!
```

Dette er et fint eksempel, fordi alle vet hvordan man spiller stein, saks, papir, men det kan fortsatt være en utfordring å programmere det opp. Derimot kan vi utvide spillet vårt. Vi kan for eksempel gjør slik at om det blir uavgjort, så spiller man på nytt med en gang, istedenfor å kjøre programmet på nytt. Det kunne man for eksempel gjort med en løkke.

4 Kaste Dart for å finne tallet π

I mattetimen på skolen lærer vi at tallet π er omtrent 3.14, og man lærer hva dette tallet er og hvordan man skal bruke det. Men det kan også være interessant å tenke litt på *hvordan* man kan vite at π er 3.14, og kanskje litt på hvordan man går frem for å finne enda flere desimaler ned i rekka. Her skal vi vise hvordan man kan komme frem til verdien av π, helt uten å vite hva den er, ved å kaste dartpiler.

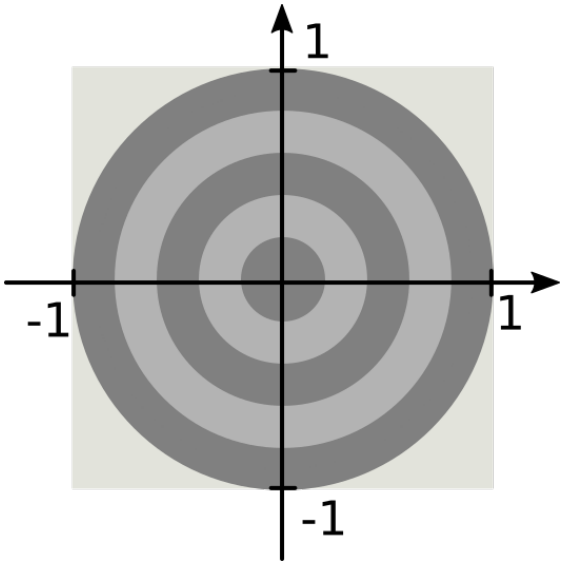
Dette kan man gjøre i person, i skolegården for eksempel. Derimot vil vi se at for å få et godt estimat av π må vi isåfall kaste veldig mange piler! Vi skal lære hvordan vi kan "kaste" millioner av dartpiler på datamaskinen på et par sekunder.

4.1 Finne en formel for π

Vi begynner med å forklare selve eksperimentet. Her er målet å komme frem til tallet π, og vi antar at vi ikke kjenner til verdien av dette tallet fra før.

Det første vi trenger å gjøre, er å bestemme oss for et blink. Her velger vi å bruke en rund skive som blink, denne tenger vi opp inne i et kvadrat slik at skiva akkurat rører kantene av kvadratet. Her kan vi for eksempel skrive skiva ut på et ark, og så klippe til, eller vi kan tegne den opp på tavla, eller på bakken med kritt. Se på figuren under for en tegning av linken.

Vi ser nå for oss at vi begynner å kaste dartpiler på blinken, men vi er ikke så veldig flinke, så dartpilene havner tilfeldige steder på arket. Da kan vi lure på: Hvor mange av pilene som treffer arket, vil havne innenfor skiva?



Når vi spør om andelen som treffer blinken, så spør vi egentlig om *sannsynligheten* for å treffe blinken. For å komme frem til hvor mange dartpiler som treffer skiva, og hvor mange som bommer på den, kan vi tenke sannsynligheten for at én enkelt pil treffer skiva, og dette kan vi finne med formelen:

$$\text{sannsynlighet} = \frac{\text{antall gunstige utfall}}{\text{antall mulige utfall}}.$$

I vårt tilfelle er jo de gunstige utfallene"å treffe blinken, mens de mulige utfallene er å treffe innenfor arket/kvadraten. Men hvor mange "mulige utfall"har vi? Vi klarer ikke å telle akkurat hvor mange utfall vi har, men det vil avhenge av hvor stort blinket er, jo større blinken er, jo større sjans for å treffe. Det er altså *arealet* av skiva som bestemmer sannsynligheten for å treffe blinken. Vi kan altså si at sannsynligheten for å treffe blinken er gitt ved

$$\text{sannsynlighet for å treffe blink} = \frac{\text{areal av blinket}}{\text{totalt areal}}.$$

Vi vet at arealet av en sirkelskive er gitt ved formelen $A = \pi r^2$. Vi later nå som at vi ikke vet hva π er, det er det som er vår *ukjente*, slik som for eksempel x pleier å være. Arealet til hele arket er bredde ganger høyde. Fra figuren ser vi at bredden er $2r$, og høyden er $2r$, slik at det totale arealet av arket er $2r \cdot 2r = 4r^2$. Om vi setter disse to inn i formelen får vi

$$\text{sannsynlighet for å treffe blink} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}.$$

Siden π er vår ukjente, så løser vi for π ved å gange med 4, og bytte sier

$$\pi = 4 \cdot \text{sannsynlighet for å treffe blink.}$$

Nå har vi altså funnet en formel, der vi kan regne ut verdien til π, gitt at vi vet hva sannsynligheten for å treffe blink er. Det eneste problemet er at vi ikke *vet* denne sannsynligheten.

For å løse dette problemet skal vi *estimere* sannsynligheten ved å kaste dartpiler.

4.2 Estimere sannsynlighet ved å kaste dart

Vi kan estimere sannsynligheten for å treffe blinket, rett og slett ved å kaste mange dartpiler, og telle hvor mange som treffer blinket. Dette blir tilsvarende slik vi estimerte sannsynlighet gjennom eksperimenter tidligere.

Vi estimerer sannsynligheten ved formelen

$$\text{estimert sannsynlighet} = \frac{\text{antall piler som treffer blink}}{\text{antall kast}}.$$

Merk her at vi kun teller kast som faktisk treffer arket, piler som havner helt utenfor kan vi ikke inkludere.

Om vi setter inn uttrykket for den estimerte sannsynligheten får vi en endelig formel for π basert på hvordan vi kastet

$$\pi = 4 \cdot \frac{\text{antall piler som treffer blink}}{\text{antall kast}}.$$

Så vi kan nå finne en estimering av π ved å kaste dartpiler, telle opp hvor mange som treffer, og sette inn i formelen. Jo flere piler vi kaster, jo bedre blir estimatet.

4.3 Utføre eksperimentet frakoblet

Dette eksperimentet har så langt ingen direkte kobling til programmering, og det er fullt mulig å utføre eksperimentet i person, og så utføre utregningene i person.

Om man skal utføre eksperimentet fysisk trenger det ikke nødvendigvis å gå ut på å kaste dartpiler. Det som er viktig er at vi får plukket ut punkter tilfeldig innenfor kvadratet, slik at noen treffer blinken, og noen ikke.

Her kan man for eksempel tegne et blink i bunnen av en pappeske, fylle med tørkede erter, riste på boksen, og så telle erter innenfor uten blinken. Eller man kan skrive ut et blink på papir, legge det på gulvet, og så helle erter over.

Det som er viktig å huske på, uavhengig om man kaster dart eller teller erter, er at pilene/ertene skal havne *tilfeldige* steder på arket. Dette er faktisk litt utfordrende, da de fleste er litt for flinke til å faktisk treffe blinket, og da vil vi overestimere π.

Det finnes en god youtube video hvor de utfører eksperimentet for å finne π. Videoen heter Calculating Pi with Darts og er lagt ut av kanalen *Physics Girl* (Klikk her for å komme rett til videoen.). I videoen bruker de flere lure triks for å sørge for at resultatet blir så tilfeldig som mulig.

Uansett hva slags eksperiment man velger å gjøre er det viktig å tenke på to ting: (1) Man ønsker en så tilfeldig fordeling av punkter som mulig, og (2) jo flere punkter man får, jo bedre estimat av π får vi.

4.4 Utføre eksperimentet på Datamaskin

Å gjøre eksperimentet på datamaskin har to store fordeler: (1) Datamaskinen er utrolig flink til å være helt tilfeldig, mye bedre enn vi får til, og (2) datamaskinen vil kunne kaste millioner av dartpiler på sekunder, og dermed lage et godt estimat av π.

Å kaste en dartpil er det samme som å finne ut hvor på arket pilen havner, og vi kan oppgi denne posisjonen i koordinater. Å kaste én dartpil er altså det samme som å finne én x og én y koordinat. For hver pil vi kaster får vi nye koordinater. I koden vår kan vi dermed kaste en digital dartpil ved å generere en tilfeldig x og en tilfeldig y koordinat.

Fra figuren vi tegnet av arket og blinken vår så vi at koordinatsystemet vårt går fra -1 til 1 for både x og y. Det er like sannsynlig å havne hvor som helst på arket, så vi ønsker at x og y skal bli valgt helt tilfeldig mellom -1 og 1. Dette kan vi gjøre med funksjonen `uniform(-1, 1)`, som trekker et uniformt fordelt desimaltall mellom de to grensene vi oppgir. Vi kan altså kaste en dartpil og se hvor den havnet med følgende kodesnutt:

```
1 from random import uniform
2 x = uniform(-1, 1)
3 y = uniform(-1, 1)
4 print(x, y)
```

Hver gang vi kjører kodesnutten kaster vi en ny pil og får nye koordinater. Tre eksempeler er:

```
0.4078123012391546 0.5378875778598815
0.34450903805618216 0.9202934023269749
-0.8149136953112757 -0.5906311058915095
```

Disse tallene er kanskje litt lite håndfast. Så her kan det være en god idé å tegne dem inn i et koordinatsystem, og se hvor de havnet.

4.5 Tegne blinket i programmet vårt

Vi ønsker kanskje at koden vår skal tegne opp, og vise frem punktene vi treffer med dartpilene, og ikke bare skrive dem ut. Dette kan fint gjøres med Python, men det er ikke nødvendig for å regne ut π. Samtidig har ikke vi fokusert på tegning og grafikk i dette kurset, og å skulle tegne blinket gjør derfor ting mer komplisert enn det trenger å være.

Derimot kan det gjøre opplegget mer spennende og håndfast ved å tegne blink. Derfor gir vi koden for hvordan dette gjøres slik at dere kan kopiere den. Her anbefaler vi at læreren viser dette på storskjerm og forklarer, så lager elevene simuleringer som ikke tegner, for å holde det enkelt. Alternativet er å dele koden som tegner blinkene i showbie, slik at elevene bare laster ned koden og kjører den.

For å tegne en tom blink bruker vi følgende kode:

```
1  from pylab import *
2
3  theta = linspace(0, 2*pi, 1001)
4  scatter(cos(theta), sin(theta), s=5)
5  plot((-1, -1, 1, 1, -1), (-1, 1, 1, -1, -1),
6       color='black')
7  axis('equal')
8  axis('off')
9
10 # Kode som kaster dart
11
```

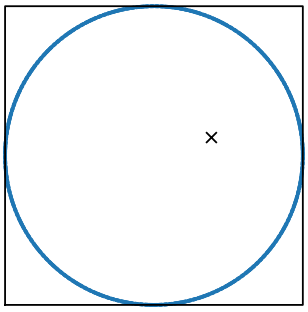
```
12 show()
```

Merk at denne koden ligger som en fil som heter `tegn_blink.py` i Showbie. Det er ikke viktig at dere eller elevene skjønner alle detaljene i denne koden, men om dere er nysgjerrige forklarer vi kjapt. Først importerer vi alt av plotting fra pakken *pylab*. Deretter bruker vi parameterfremstilling for å tegne sirkelen (de to første linjene). Så tegner vi ut "arketsom en firkant ved å skrive opp hjørnene (tredje linje), så skrur vi av alt annet av akser og slikt som skal vises frem og sørger for at aksene er like store, slik at dartskiva ikke vises som en oval. Til slutt brukes vi `show()` for å vise figuren. Om vi bruker show med en gang, så får vi et tomt blink, så om vi også vil tegne inn dartpiler må vi gjøre dette før vi bruker show.

For å tegne en dartpil kan vi bruke følgende kode

```
1 # Kast en pil
2 x = uniform(-1, 1)
3 y = uniform(-1, 1)
4
5 # Tegn inn der pilen traff
6 scatter(x, y, marker='x', s=60, color='black')
```

Her sier vi at vi tegner inn treffet som en `x`, som skal være svart og en viss størrelse (`s` står for size"). Du kan justere størrelsen og symbolet om ønskelig. Vi kan nå lage en figur som dette:

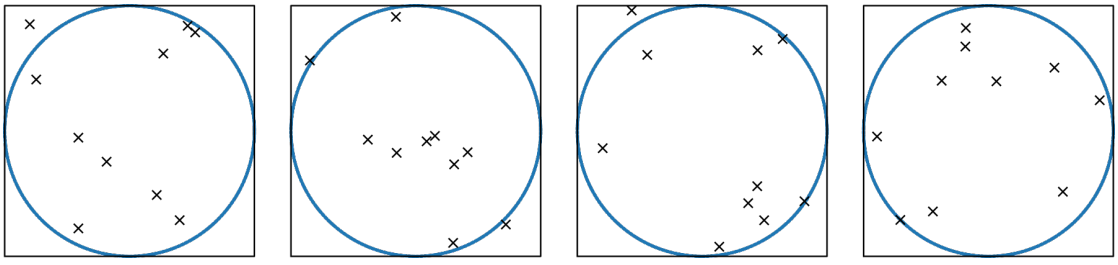


Vi ser at blinken er tegnet som en blå sirkel, siden av arket er tegnet som en tynn svart linje, og dartpilen vår er tegnet inn med et svart kryss.

La oss øke antall kast. Istedenfor å kaste én pil, la oss kaste 10. Da bruker vi en løkke for å gjenta prosessen mange ganger. Vi trenger bare å tegne blinken én gang, men vi å må kaste pilen på nytt mange ganger, og tegne inn hvor den traff mange ganger. Hele programmet blir som følger

```
from pylab import *
from random import uniform

# Tegn tomt blink
theta = linspace(0, 2*pi, 1001)
scatter(cos(theta), sin(theta), s=5)
plot((-1, -1, 1, 1, -1), (-1, 1, 1, -1, -1),
      color='black')
axis('equal')
axis('off')

# Gjenta 10 ganger
for kast in range(10):
        # Kast en pil
        x = uniform(-1, 1)
        y = uniform(-1, 1)

        # Tegn inn pil
        scatter(x, y, marker="x", s=80, color="black")

# Vis figuren
show()
```

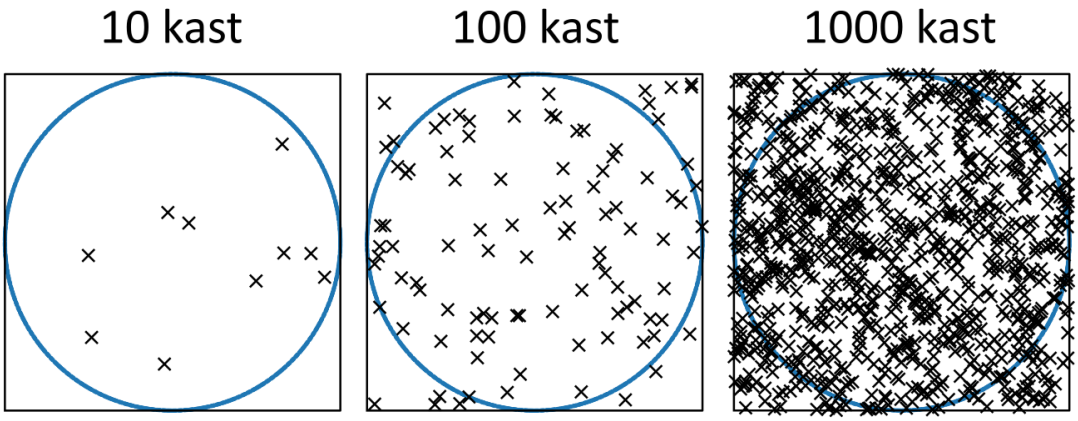
Om du nå kjører koden får du en figur med 10 helt tilfeldig fordelte punkter på blinken. Hver gang du kjører havner punktene andre steder. Vi viser en figur under av 4 slike kjøringer



Programmet vi nå har laget har allerede et viktig læringsmoment. Mange av fi-

gurene vi får ut vil man synes ikke nødvendigvis *ser* så veldig tilfeldige ut, men dette er bare fordi vi er veldig dårlig på å skjønne tilfeldighet. Det mange vil *mene* ser tilfeldig ut, er egentlig jevnt fordelt utover, men det er jo faktisk *ikke* så veldig tilfeldig. Programmet Siffer tar for seg nøyaktig dette i et av programmene sine. Produsenten har selv delt episodene på Youtube, du kan finne klippet i videoen *Siffer: Helse (4:10)"*, segmentet begynner ved 13-minutters merket. (Klikk her for å komme rett til klippet). I segmentet på Siffer ber programleder Jo Røislien to gjester om å fordele en kasse med badeender utover en flate helt tilfeldig. Når man ber noen fordele noe tilfeldig så blir det gjerne veldig jevnt og regelmessig fordelt, men om man faktisk gjør det tilfeldig vil man ofte se at ting klumper seg litt sammen.

Programmet vi har skrevet gir faktisk *helt tilfeldige* punkter. Python er kort fortalt, ekstremt god på å generere tilfeldighet, langt bedre enn oss mennesker. Her kan det også være morsomt å gå tilbake til programmet vårt og øke antall dartpiler. Dette er så enkelt som å endre argumentet i løkka vår, og kjøre på nytt. Under er 3 kjøringer med økende antall kast



Hvis man utforsker litt hvordan disse figurene blir med fler og fler kast vil vi se at vi får en jevnere og jevnere fordeling med flere kast. Dette er et eksempel på *store talls lov*, og det er dette som garanterer at vi kommer til å ende opp med et godt estimat av π når vi teller antall treff innenfor og utenfor sirkelen.

4.6 Bedømme om en dart har truffet skiva

Det å tegne opp blinket er egentlig bare en liten digresjon. For å estimere π trenger vi kun å kunne si om en dartpil har truffet blinken eller ikke. Slik at vi kan telle

antall treff og bom. I virkeligheten, eller på figurene vi tegnet, så er det så enkelt som å telle. Men å telle tusenvis, eller millioner, av piler for hånd er ekstremt slitsomt. Så vi må finne en måte datamaskinen kan finne ut selv om pilen har truffet eller ikke.

Det datamaskinen har å jobbe med er koordinatene til dartpilen, x og y. Så det vi må finne er en måte å *regne* ut, fra koordinatene, om vi er innenfor sirkelen eller ikke. Trikset for å gjøre dette er å regne ut avstanden fra dartpilen til sentrum av sirkelen, i dette tilfellet er det origo. Hvis avstanden fra darten til sentrum er mindre enn radiusen til sirkelen, så er vi innafor sirkelen, mens om avstanden er større enn radiusen, så er vi utenfor sirkelen. Husk at vi i programmet vårt har sagt at radiusen til sirkelen er 1.

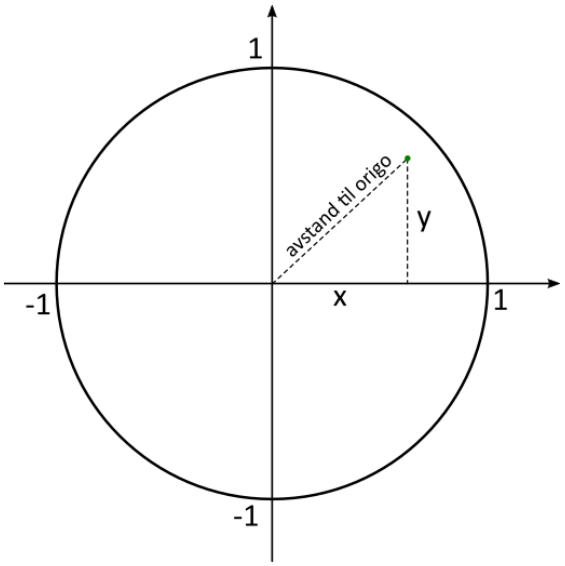
Men hvordan regner vi ut avstanden til sentrum basert på koordinatene x og y? Det viser seg at vi kan gjøre dette med Pytagoras. Uansett hvor dartpilen havner, kan vi tegne en rettsidet trekant med en av hjørnene i origo, en på dartpilen og den siste langs x-aksen. De to langsiden i trekanten blir x og y lange, og langsiden er avstanden fra origo til punktet. Dermed vet vi at

$$x^2 + y^2 = d^2,$$

der d er avstanden fra origo. Vi kan da regne ut avstanden ved å ta kvadratrot på begge sider, slik at

$$d = \sqrt{x^2 + y^2}.$$

Vi kan tegne opp en figur som viser situasjonen som følger



4.7 Kaste mange piler og telle treff

Nå som vi har sett hvordan vi kan sjekke om en pil har truffet eller ikke, la oss prøve å kode det. Vi starter med å gjøre det med bare én pil:

```
from random import uniform
from math import sqrt

x = uniform(-1, 1)
y = uniform(-1, 1)

avstand = sqrt(x**2 + y**2)

if avstand <= 1:
    print("Treff!")
else:
    print("Bom!")
```

Hver gang denne koden kjøres kaster vi en ny pil, og ser om den treffer eller ikke med Pytagoras.

Vi ønsker nå å kaste mange piler, så da bruker vi en løkke

```
from random import uniform

antall_kast = 10
antall_treff = 0

for kast in range(antall_kast):
    x = uniform(-1, 1)
    y = uniform(-1, 1)

    avstand = sqrt(x**2 + y**2)

    if avstand <= 1:
        antall_treff += 1

print(f"Antall kast: {antall_kast}")
print(f"Antall treff: {antall_treff}")
```

```
Antall kast: 10
```

```
Antall treff: 7
```

4.8 Estimere π

Nå som vi klarer å kaste mange piler og telle treff er vi veldig nærme å finne π. Om du husker tilbake til formelen vi lagde var den

$$\pi = 4 \cdot \frac{\text{Antall treff}}{\text{Antall kast}}.$$

La oss legge til dette i programmet vårt.

Hele programmet vi trenger for å finne π er nå:

```python
from random import uniform
from math import sqrt

antall_kast = 10
antall_treff = 0

for kast in range(antall_kast):
    # Kast en pil
    x = uniform(-1, 1)
    y = uniform(-1, 1)

    # Sjekk om den traff
    avstand = sqrt(x**2 + y**2)
    if avstand <= 1:
        antall_treff += 1


# Estimer pi basert på kastene
pi = 4*antall_treff/antall_kast

# Skriv ut resultater
print(f"Antall kast: {antall_kast}")
print(f"Antall treff: {antall_treff}")
print(f"Estimert pi: {pi}")
```

```
Antall kast: 10
```

```
Antall treff: 9
Estimert pi: 3.6
```

Hver gang programmet kjører får vi et litt annet estimat av π. Akkurat nå får vi et ganske dårlig estimat på 3.6! Men dette er jo ikke så rart, siden vi her kaster bare 10 piler, og det er ikke nok.

Om vi isteden prøver med 1000 piler, så blir ting bedre:

```
Antall kast: 1000
Antall treff: 789
Estimert pi: 3.156
```

Det er fortsatt ikke helt 3.14, men den varierer litt over og under. Om du husker tilbake til eksempelet vi hadde tidligere så vi at vi kunne forvente feilmarginer på et par prosent med 1000 simuleringer. Så dette er å forvente.

La oss gå enda litt lengre og kaste en million darts. Denne gangen bruker maskinen en del tid, for dette er litt vrient.

```
Antall kast: 10000000
Antall treff: 7854057
Estimert pi: 3.1416228
```

Dette er et meget godt estimat av π!

4.9 Hva er det vi har gjort?

Det vi her har gjort er en kombinasjon av geometri og sannsynlighetsregning.

Først brukte vi geometri til å argumenterer for hvorfor sannsynligheten av å treffe et blink med en dartpil er gitt av π. Den formelen vi fant, snudde vi rundt på, slik at π ble den ukjente.

Deretter brukte vi et eksperiment til å estimere hva denne sannsynligheten er, og slik fant vi π. Dette er et eksempel på sannsynlighetsregning og statistikk.

Vi brukte her datamaskinen og programmering kun for å utføre selve eksperimentet med å kaste mange piler. Ingenting i programmet vårt kjenner til hva π er for noe eller bruker dette tallet til noe.

Hvorfor fungerer dette?

Den første delen av prosjektet er bare å si at π og arealet til sirkelen henger sammen. Altså kan man regne ut arealet om man kjenner π og radiusen, men om man kjenner radiusen og arealet kan man jo regne ut π.

Selve dartpil-kastingen er egentlig bare en komplisert måte å komme frem til arealet av en sirkel. Det vi egentlig gjør når vi kaster piler er å estimere sirkelens areal, og derfra regner vi ut π.

Det å finne arealet ved å se på tilfeldige punkter, er en form for *integrasjon* som heter Monte Carlo-integrasjon. Denne metoden er navngitt etter det berømte Monte Carlo casionet.

Integrasjon er et tema elever gjerne ikke møter på før i R2 matematikken på slutten av VGS, og der dekkes ikke Monte Carlo integrasjon, som er et tema man gjerne ikke møter på før universitetet. Det trenger man selvfølgelig ikke å si til elevene, for dem har man bare gjort et morsomt eksperiment, som ser ut til å fungere litt på magisk vis.

Det å kaste dartpiler for å finne π er ikke den mest effektive måten å gjøre det på, og det finnes langt bedre måter å finne π på. Derimot er dette et kult eksempel, som kan brukes til å øke forståelse for geometri, sannsynlighetsregning og programmering på en og samme gang.

5 Ekstraopplegg: Finne formelen for volumet til en kule

For de aller fleste på ungdomsskolen så er nok eksemplene vi har vist frem til nå nok, og spesielt om man bygger opp til og gjennomfører π-prosjektet har man lært mye programmering og brukt det til interessante matematiske problemstillinger. Derimot finnes det en liten utvidelse av π-prosjektet vi kan gjøre, som gjør at vi går et lite steg lenger. Dette presenterer vi her.

Dette opplegget kan være fint å gi til elever som er veldig flinke og har lyst på enda mer, eller det kan eventuelt brukes på en klasse som har vært igjennom π-prosjektet året før for eksempel. Eller kanskje det viser seg at π-prosjektet fenger veldig, og du tror klassen ønsker å ta det hele ett steg lenger.

5.1 Motivasjon

Formelen for volumet til en kule er

$$V = \frac{4\pi R^3}{3}.$$

Denne formelen lærer man tidlig i matematikken, men det er gjerne ikke før langt senere, gjerne på universitetet, at man lærer hvordan man kan utlede den. Vi skal nå se på hvordan vi kan bruke kombinasjonen av geometrisk tankegang og datasimuleringer til å komme frem til formelen.

Både konseptuelt og programmeringsmessig bygger dette eksperimentet i stor grad videre på dartkastingen vi dekket tidligere. I programmeringen er det kun et par linjer kode som må legges til programmet vårt for å komme frem til svaret. Konseptuelt er det et litt lenger steg å ta. Det lønner seg altså her å fokusere mye på matematikken og den geometriske tankegangen etterhvert som man jobber seg igjennom eksperimentet.

5.2 Volumbegrepet

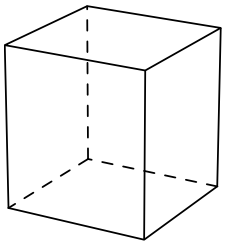
Å gå fra å diskutere, tenke på, og regne med geometri i to dimensjoner, til tre dimensjoner, er ofte rimelig vrient. I to dimensjoner snakker vi om omkrets og areal, mens i tre dimensjoner snakker vi om overflate og volum. Når det kommer

til volumet av en kube, eller et sylinder, er de på ingen måte *trivielle*, men det kan være oppnåelig å forstå utledningen av dem.

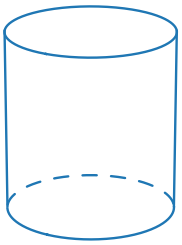
En kube er nok det letteste, der man går fra å ha arealet til et rektangel som bredde ganger høyde, til nå å ha volum som bredde ganger høyde ganger lengde.

For et sylinder er det litt mer vrient, men man kan argumentere med grunnflate ganger høyde for å finne formelen $2\pi rh$. Her finnes det gode gjennomganger og beskrivelser for eksempel i mattebøker eller nettsider som matematikk.org. Det er fint om elevene har en forholdvis god forståelse av dette før vi begynner på volumet til en kule.

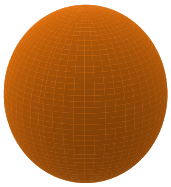
Når det kommer til formelen for volumet av en kule er det mindre forklaringer og ta av. Matematikk.org har en forklaring der de snakker om å skrelle en kule og bygge et parallellogram ut av " skrellet", men denne forklaring er ikke noe matematisk bevis, og ikke er den så veldig intuitiv heller.



V = bredde·lengde·høyde



V = grunnflate·høyde



V = ???

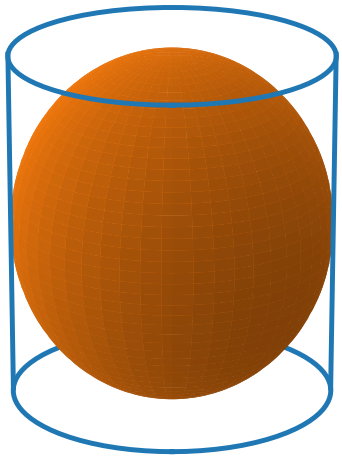
5.3 Litt historisk perspektiv

Ikke overraskende var det en gresk matematiker som først fant formelen for volumet til en kule, og dette var Arkimedes. Arkimedes var et multitalent og gjorde store oppdagelser innen matematikk, fysikk, ingeniørkunst og astronomi. Til tross for hans utrolig mange oppdagelser og matematiske teorem var det formelen for volumet til en kule han var mest stolt over, og han spurte om å få nettopp denne oppdagelsen plassert på sin gravsten.

Arkimedes laget et langt geometrisk bevis for formelen, et bevis som er altfor langt og komplisert til å gjentas i stor grad idag. Idag brukes mer moderne matematikk for å finne formelen, nemlig kalkulus og integralregning. I dette eksperimentet gjentar vi hovedargumentet til Arkimedes ved hjelp av programmering, og det går ut på å finne andelen av et sylinder som rommes av en kule.

5.4 En kule i et sylinder

Se for deg at du har en ball, og et rør som er akkurat like bred og høy som ballen, slik at vi akkurat får plass til å plassere ballen inne i rører. Det vi har er da et sylinder, med en kule som akkurat får plass i sylinderet, både med tanke på høyde og bredde. Vi tegner en figur av situasjonen



Hvis vi sier at kula har en radius R, så vet vi at sylinderet har den samme radiusen. Sylinderet er akkurat like høy som hele ballen, og høyden av sylinderet blir derfor lik ballens diameter, det vil si $2R$. Det betyr at grunnflaten til sylinderet er πR^2, og volumet blir da

$$\text{volum av sylinder} = G \cdot h = \pi R^2 \cdot 2R = 2\pi R^3.$$

Men hva er så volumet av kula? Om vi ser på figuren ser vi ihvertfall at volumet av kula må være mindre enn volumet av sylinderet, ellers ville ikke hele kula fått plass inne i sylinderet. Vi kan se at volumet til kula kan skrives som *andelen* av sylinderet kula fyller, ganger volumet til sylinderet. Si for eksempel om kula rommer nøyaktig halve sylinderet, da må volumet vært halvparten av sylinderet. Vi kan altså skrive formelen som

$$\text{volum av kule} = (\text{andel av sylinderet som er fylt av kule}) \cdot 2\pi R^3.$$

Det vi må gjøre nå, er å finne andelen av sylinderet som kula fyller. Dette kan i teorien gjøres på flere ulike vis. Arkimedes gjorde det ved hjelp av et geometrisk bevis, men som nevnt tidligere er det rimelig vanskelig og tar veldig mye arbeid.

Vi kunne utført et fysisk eksperiment: Hvis vi har en sylinderformet beholder kan vi fylle den med vann og målt vekten. Så dytter vi en kule ned i sylinderet så en del av vannet flyter over kanten og forsvinner. Tar vi kula ut igjen og måler vannet som er igjen har vi nok informasjon til å finne ut hvor mye av sylinderet kulen rommet. Utfordringene er å finne en sylinder og en kule av riktig størrelse, og man trenger en nøyaktig nok vekt. Samtidig kan det være vanskelig å utføre eksperimentet nøyaktig nok.

Vi velger istedet å finne andelen ved hjelp av datasimuleringer, på samme måte som vi estimerte π tidligere.

5.5 Finne andelen kula med datasimulering

For å finne andelen kula dekker av sylinderet gjør vi nesten nøyaktig det samme vi gjorde da vi kastet dart. Å kaste dart svarte til å finne to koordinater x og y tilfeldig. Nå "kaster vi dart" i 3D, og velger *tre* tilfeldige koordinater: x, y og z.

Sannsynligheten for at et punkt som havner inne i sylinderet også havner inne i kula blir nå gitt ved forholdet mellom volumene

$$\frac{\text{antall treff i kule}}{\text{antall treff i sylinder}} = \frac{\text{volum av kule}}{\text{volum av sylinder}}.$$

Men forholdet mellom volumene er jo egentlig bare andelen kula fyller av sylinderet, så vi kan skrive dette som

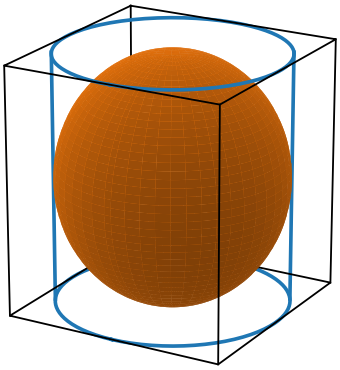
$$\text{andel av sylinderet som er fylt av kule} = \frac{\text{antall treff i kule}}{\text{antall treff i sylinder}}.$$

Da gjenstår det bare å skrive et program som trekker punkter tilfeldig, sjekker om de treffer innefor sylinderet og kula, og holder tellingen på antall treff.

Når vi trekker koordinater trekker vi nå tre koordinater. Vi lar igjen sentrum av kula og sylinderet ligge i origo, så vi trekker punkter mellom -1 og 1.

```
1 x = uniform(-1, 1)
2 y = uniform(-1, 1)
3 z = uniform(-1, 1)
```

Å trekke tre koordinater uniformt mellom -1 og 1 svarer til å plukke et tilfeldig punkt i en tredimensjonal kvadratisk kube. Så situasjonen er nå vist som i figuren under. Vi trekker tilfeldige punkter innenfor den svarte kuben, og så skal vi sjekke om de ligger innenfor det blå sylinderet og inenfor den oransje kula.



For å sjekke om et punkt er innenfor sylinderet gjør vi nøyaktig det samme som for dartskiva. Vi regner ut Pytagoras med x og y. Nå er ikke dette lenger avstanden til origo, men til senterlinja av sylinderet. Kort fortalt har ikke z-verdien noe å si for om punktet er innenfor sylinderet. Om dette er forvirrende kan det lønne seg å tenke seg situasjonen sett rett ovenifra, da ser z-aksen ut som den er klemt flat som en pannekake, og sylinderet ser bare ut som en sirkel. Da har vi en situasjonen som kan behandles som om den var 2D.

```
# Gjennomfør mange "kast" med en løkke
for kast in range(1000):
        # Trekk posisjon
        x = uniform(-1, 1)
        y = uniform(-1, 1)
        z = uniform(-1, 1)

        # Sjekk om punktet er i sylinderet
        if sqrt(x**2 + y**2) < 1:
                antall_treff_i_sylinder += 1
```

Men hvordan sjekker vi om punktet er innafor kula? Her må vi bruke samme triks som istad og regne avstanden til origo. Om avstanden til origo er mindre enn 1, er vi innafor kula, er den større er vi utafor origo.

Istad brukte vi Pytagoras for å regne avstanden til origo, og det vi kan gjør nå er å bruke Pytagoras i 3D! Dette har elevene kanskje aldri sett, så her må man vurdere om man vil bruke litt tid på å forklare dem hvordan det funker, det kan nok være en god læringsopplevelse. Det finnes mange fine forklaringer av Pytagoras i 3D på internet med forklarende bilder. Vi bruker ikke tid på utlede det her. Kort fortalt sier Pytagoras i 3D at avstanden til origo er gitt ved formelen

$$x^2 + y^2 + z^2 = d^2,$$

der d er avstanden. For å finne d tar vi kvadratroten på begge sider og flytter sidene slik at vi får

$$d = \sqrt{x^2 + y^2 + z^2}.$$

Denne formelen er altså ekstremt lik som Pytagoras i 2D, vi bare legger til den tredje koordinaten.

Med dette uttrykket er vi klare for å skrive et komplett program

5.6 Komplett program

```
from random import uniform
from math import sqrt

antall_kast = 1000000
antall_treff_i_sylinder = 0
antall_treff_i_kule = 0

for kast in range(antall_kast):
        x = uniform(-1, 1)
        y = uniform(-1, 1)
        z = uniform(-1, 1)

        # Sjekk om punktet er i sylinderet
        if sqrt(x**2 + y**2) < 1:
                antall_treff_i_sylinder += 1

        # Sjekk om punktet er i kula
        if sqrt(x**2 + y**2 + z**2) < 1:
                antall_treff_i_kule += 1


# Regn ut andelen av sylinderet som er fyllt av kula
andel = antall_treff_i_kule/antall_treff_i_sylinder

# Skriv ut resultater
print(f"Antall kast: {antall_kast}")
print(f"Antall treff i sylinderet: {
   antall_treff_i_sylinder}")
print(f"Antall treff i kula: {antall_treff_i_kule}")
print(f"Andel av sylinderet fyllt av kula: {andel:.2%}")
```

Igjen er programmet vårt basert på tilfeldighet, så vi får forskjellige svar hver gang vi kjører det. Det er også igjen slik at vi ønsker å lene oss på store talls lov, så vi kan være sikre på svaret vårt. Vi velger derfor å bruke en million kast.

Da tar nok programmet en liten stund å kjøre, men etter litt tid får vi ut et svar. En kjøring av programmet gir meg følgende

```
Antall kast: 1000000
Antall treff i sylinderet: 785024
Antall treff i kula: 523414
Andel av sylinderet fyllt av kula:  66.67%
```

Fra dette resultatet ser vi at kula fyller omtrent 66.67 % av sylinderet, det vil si 2/3.

5.7 Volumet av en kule

Hvis vi nå går helt tilbake til starten av eksperimentet vårt hadde vi satt opp:

$$\text{volum av kule} = (\text{andel av sylinderet som er fylt av kule}) \cdot 2\pi R^3.$$

Og vi har nå funnet at denne andelen er to tredjedeler, så vi setter inn 2/3.

$$\text{volum av kule} = \frac{2}{3} \cdot 2\pi R^3.$$

Og når vi ganger inn med brøken vår vi

$$\text{volum av kule} = \frac{4\pi R^3}{3}.$$

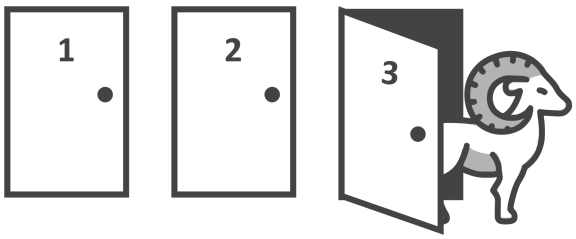
Hva er det vi har lært?

I dette opplegget har vi gått igjennom og lagd et argument for hva volumet til en kule må være, avhengig av hvor mye av et sylinder det fyller. Deretter viste vi hvor mye av et sylinder en kule fyller ved hjelp av programmering og sannsynlighetsregning.

Merk at dette er en ganske lik fremgangsmåte Arkimedes brukte, han brukte bare geometri for å komme frem til 2/3 istedenfor sannsynlighetsregning, men det er bare to forskjellige veier frem til samme mål.

6 Prosjekt: Monty-Hall Problemet

Monty Hall problemet er en veldig kjent mattenøtt med et veldig overraskende svar. Det er et problem som fører til mye engasjement og diskusjon, og har selv en spennende historie der diskusjon rundt matematikk ble satt på dagsorden i hele USA.



Monty Hall problemet er et flott problem å bruke i Matematikkundervisningen, da det fører til god diskusjon rundt sannsynlighet og tilfeldighet. Det er også fullt mulig å simulere spillet i klasserommet for å teste om intuisjonen til elevene stemmer, noe den ofte ikke gjør. Ved å simulere *manuelt* kan man enkelt gjennomføre spillet et par hundre ganger, men vi kan også simulere på datamaskin for å få langt flere simuleringer. Dette gjør Monty Hall problemet til en mulig prosjektoppgave, gruppearbeid eller kanskje et tema for en fagdag.

Vi starter her med å presentere problemet og hvor det kommer fra. Deretter går vi gjennom noen mulige forklaringer av løsningen før vi til slutt dekker hvordan spillet kan simuleres både manuelt og ved hjelp av programmering.

6.0.1 Problemets historie

Problemet ble formulert og løst allerede i 1975, men ble ikke allment kjent på dette tidspunktet. Det var først når problemet ble formulert på nytt og sendt til *Ask Marilyn* i 1990 at ting virkelig tok av. *Ask Marilyn* er en kolonne i *Parade magazine* der Marilyn vos Savant besvarer diverse matematiske nøtter og logiske utfordringer. Vos Savant er kjent for å ha blitt kåret til den høyeste målte IQ-en av Guinnes rekordbok.

6.1 Problemets formulering

Se for deg at du er på et gameshow, og du får velge mellom tre dører: Bak én av dem er en bil; bak de to andre, geiter. Du velger en dør, for eksempel nummer 1. Verten, som vet hva som er bak dørene, åpner en av de andre dørene, for eksempel nummer 3, som har en geit. Han sier nå til deg, "Har du lyst å bytte til dør 2?". Er det til din fordel å bytte dør?

(Dette er den originale formuleringen av problemet publisert i *Parade* i 1990, oversatt til norsk.)

Problemet er kjent som *Monty Hall* problemet da Monty Hall var en kjent gameshowvert på denne tiden. Derimot var det aldri noe game show som faktisk hadde denne oppgaven i programmet sitt.

6.2 Debatten

Marilyn besvarte spørsmålet helt korrekt i sin kolonne. Men kort tid etter at problemet og svarer ble publisert fikk *Parade* en enorm mengde leserbrev i protest og både lekfolk og fagfolk var skråsikre på at hun tok feil—noen gikk så langt som å beskylde henne for å drive med vranglære og for å aktivt ødelegge for matematikkundervisning i USA.

Dette førte til at Marilyn måtte returnere til det samme spørsmålet ved flere anledninger og komme med mer detaljerte forklaringer. På Marilyn's egne nettside (`http://marilynvossavant.com/game-show-problem/)` kan dere lese de originale svarene Marilyn skrev, samt noen av brevene *Parade* mottok i protest og støtte.

6.3 Trumfkortet

Debatten rundt spørsmålet fortsatte helt til Marilyn til slutt spurte om hjelp fra skoleklasser over hele USA til å rett og slett *gjør e simuleringer* for å komme frem til svaret. Kort tid etter var det gjort nok forsøk og empirisk innsamling til å bekrefte at Marilyns svar var helt rett hele tiden.

6.4 Å simulere én runde

Vi skal nå gjenta eksperimentet til Marilyn. Først gjør vi dette ved å simulere Monty Hall problemet for hånd. Etter vi har gjort dette går vi over til å gjøre det ved hjelp av programmering. Målet er å gjennomføre en million eller flere eksperimenter, så vi kan være sikre på at vi har funnet riktig svar.

6.4.1 Simulere Monty Hall i klasserommet med terning

Vi kan simulere Monty Hall problemet med 3 pappkopper og et lite objekt som kan legges under en kopp, for eksempel en liten ball, en sammenrullet papirbit, en liten stein, eller hva som helst annet.

To personer samarbeider om simuleringen, den ene er verten og den andre deltakerene. Først skjuler verten objektet under en av de tre koppene uten at deltakeren ser det. Deretter velger deltakeren en av de tre koppene. Verten løfter nå koppen og viser at objektet ikke er der. Deltakeren kan nå velge om de vil bytte eller ikke.

Når vi skal simulere Monty Hall problemet mange ganger for å finne sannsynligheter, er det viktig at vi er objektive. For eksempel må vi passe på at vi ikke faller inn i mønstre fordi vi begynner å kjede oss og slikt. Vi skal derfor bruke terninger.

Kall de tre koppene 'A', 'B' og 'C'. Først ruller verten en terning, om det blir 1 eller 2 er premien under A, om det blir 3 eller 4 er det under B og om det blir 5 eller 6 legges premien under C. Deretter ruller deltakeren for hvilken han eller hun gjetter på etter samme system. Etter gjettet viser verten en kopp uten en premie under. Spill 10 ganger der man velger å bli, og 10 ganger der man velger å bytte. Hva er resultatet?

6.4.2 Simulere Monty Hall med programmering

Vi skal nå lage et program som kan simulere Monty Hall problemet. Vi starter med å lage et program som spiller én runde. Vi må være sikre på at denne versjonen fungerer som den skal før vi går videre til å simulere en lang rekke spill etterhverandre. Vi skriver ut informasjon for hvert steg i prosessen for å se at alt går riktig for seg.

```
# Hvilken strategi bruker vi, bytte eller bli?
strategi = "bytte"

# Hvilken dør er premien bak?
dører = ["A", "B", "C"]
fasit = choice(dører)
print("Premien er bak dør: {}".format(fasit))

# Plukk en dør tilfeldig
førstevalg = choice(dører)
print("Dør {} blir valgt.".format(førstevalg))

# Verten viser frem en av dørene vi ikke har valgt, og
   viser at det er en geit
shuffle(dører)
for dør in dører:
    if dør != førstevalg:
        if dør != fasit:
            geit_dør = dør
            print("Verten viser frem en geit bak dør",
               geit_dør)
            break

# Vil vi bytte dør?
if strategi == "bytte":
    for dør in dører:
        if dør != førstevalg and dør != geit_dør:
            endelig_valg = dør
            print("Du byttet til dør {}.".format(
               endelig_valg))
else:
    endelig_valg = førstevalg
    print("Du valgte å bli på dør {}".format(endelig_valg)
       )

# Sjekk om vi vinner
if endelig_valg == fasit:
    print("Du vant bilen!")
else:
    print("Du fikk en geit denne gangen")
```

```
Premien er bak dør: A
Dør C blir valgt.
Verten viser frem en geit bak dør B
Du byttet til dør A.
Du vant bilen!
```

6.5 Simulere mange spill og estimere sannsynlighet

Nå som vi har fått til å simulere en runde med Monty Hall problemet gjenstår det kun å skrive om litt slik at vi kan gjennomføre andelen spill, og finne sannsynligheten for å vinne med de ulike strategiene. Vi lager en funksjon og bruker en løkke.

```
1  def montyhall(strategi):
2      dører = ["A", "B", "C"]
3      fasit = choice(dører)
4
5      førstevalg = choice(dører)
6
7      shuffle(dører)
8      for dør in dører:
9          if dør != førstevalg:
10             if dør != fasit:
11                 geit_dør = dør
12                 break
13
14     if strategi == "bytte":
15         for dør in dører:
16             if dør != førstevalg and dør != geit_dør:
17                 endelig_valg = dør
18     else:
19         endelig_valg = førstevalg
20
21     if endelig_valg == fasit:
22         return 1
23     else:
24         return 0
```

Hver gang vi kaller på funksjonen får vi enten 0 tilbake om vi taper, eller 1 tilbake om vi vinner:

```
print(montyhall('bytte'))
print(montyhall('bytte'))
print(montyhall('bytte'))
```

```
1
1
0
```

Med funksjonen vår i boks er det egentlig bare å bruke en løkke til å spille mange ganger og holde telling på antall seire. Da kan vi til slutt estimere sannsynlighetenfor å vinne med de to ulike strategiene.

```
strategi = 'bytte'
antall_simuleringer = 100000
antall_seire = 0

for runde in range(antall_simuleringer):
    antall_seire += montyhall(strategi)

# For hver runde så enten vinner vi, eller så taper vi så vi vet at
antall_tap = antall_simuleringer - antall_seire
andel_seire = antall_seire/antall_simuleringer
andel_tap = antall_tap/antall_simuleringer

print(f"Du spiller med strategien: {strategi}")
print(f"Du spilte {antall_simuleringer} antall runder.")
print(f"Du vant {antall_seire} runder ({andel_seire:.1%})")
print(f"Du tapte {antall_tap} runder ({andel_tap:.1%})")
```

```
Du spiller med strategien: bytte
Du spilte 100000 antall runder.
Du vant 66707 runder (66.7%)
Du tapte 33293 runder (33.3%)
```

Og med det har vi laget et program som simulerer Monty-Hall problemet for oss, og lar oss etterprøve vår intuisjon.