INF100 h26 – kursnotater

↑ Innhold

Generert 4.11.2026

Installasjon

Kom i gang (print/input) • Variabler og uttrykk • Typer • Operatorer • Betingelser (if/else) • Boolske uttrykk • Funksjoner • Minne og kodesporing • Feil og debugging • Grafikk

Løkker • Strenger • Lister • Oppslagsverk • Tilfeldighet • Flerdimensjonale lister

Grafiske brukergrensesnitt • Appendiks til grafiske brukergrensesnitt

JSON • Filer og CSV • Unicode og tekstkoding • Håndtere krasj • Mengder (set)

Moduler • Standardbiblioteket • Eksterne pakker

Installasjon

↑ Innhold

I dette kurset bruker vi Python 3.13 som programmeringsspråk og Visual Studio Code som editor.

Installasjon av Python

- Last ned og installer Python 3.13 eller nyere fra https://www.python.org/downloads/
  - Windows: husk å markere «Add to PATH» på første skjerm i veiviseren
  - Mac: husk å kjøre *Install Certificates.command* og *Update Shell Profile.command* etter at veiviseren er ferdig.

Installasjon av Visual Studio Code

> *PS: Visual Studio Code er **ikke** det samme som Visual Studio, selv om begge deler er kodeeditorer laget av Microsoft.*

- Last ned og installer VSCode (Visual Studio Code) fra https://code.visualstudio.com/Download
  - Mac: flytt programmet til «Applications» -mappen slik at du ikke får problemer med begrensede rettigheter senere.

Konfigurasjon av Visual Studio Code

- Åpne Visual Studio Code, og gå til «View -> Extensions».
- Søk etter «Python» og installer Python-utvidelsen publisert av Microsoft.
- Opprett en ny fil (f. eks. *hello.py*) og la den inneholde teksten `print("Hello World")`.
- Sjekk at VSCode finner riktig python-version (nede i høyre hjørne)
- Kjør programmet og se at *Hello World* skrives ut i terminalen.

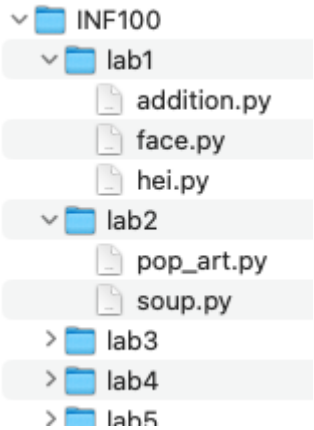
Hvordan organisere arbeidet ditt?

I dette emnet vil du etter hvert skrive *mange* Python-programmer. For å gjøre livet ditt enklest mulig, bør du organisere disse filene på en hensiktsmessig måte. Vi er her noen råd.

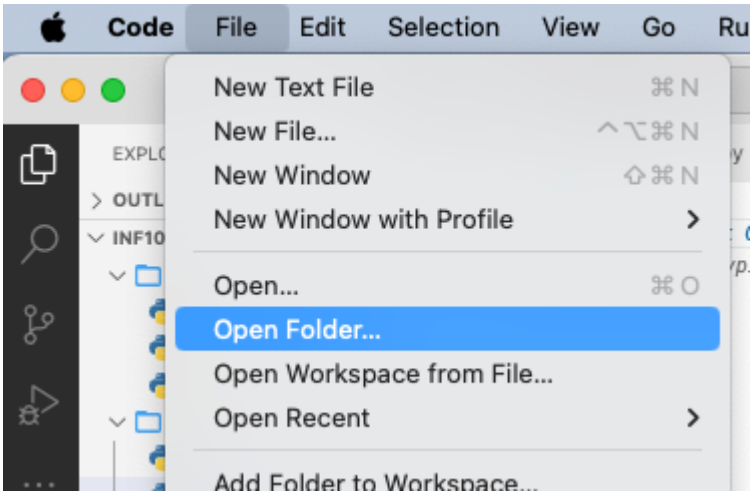
1. Opprett en mappe som heter **INF100** for alt arbeidet ditt i emnet. La mappen være et sted du kjenner godt til; gjerne et sted hvor det tas jevnlig backup (f. eks. iCloud, OneDrive, Dropbox, Dokumenter - mapper eller lignende).

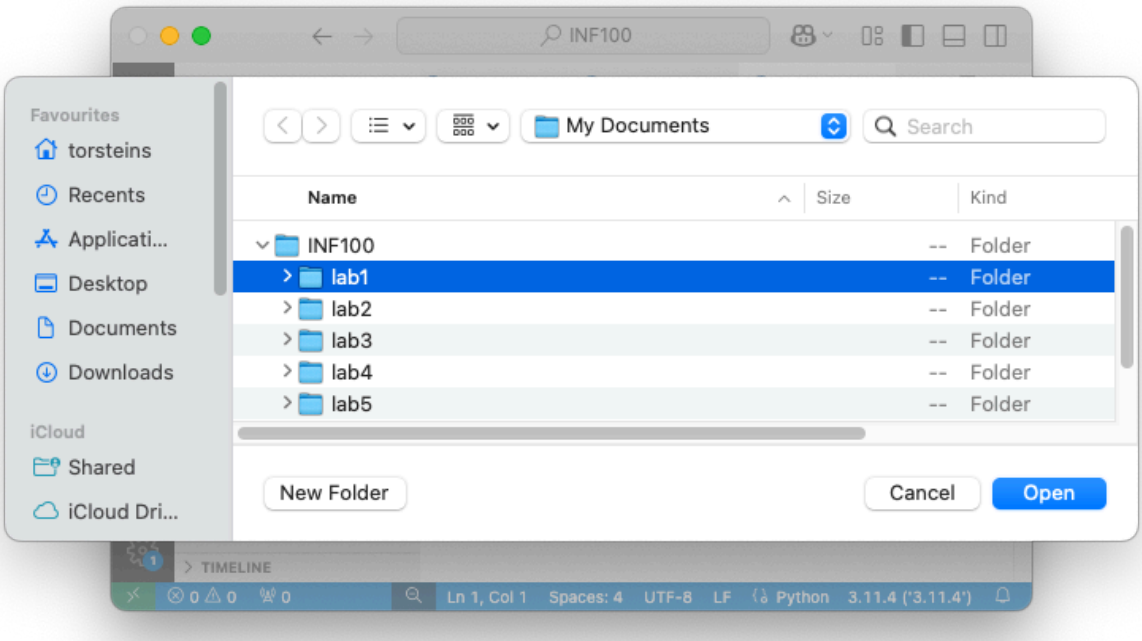
2. Inne i mappen **INF100** er det lurt å ha egne mapper for hver lab.

↑ Innhold



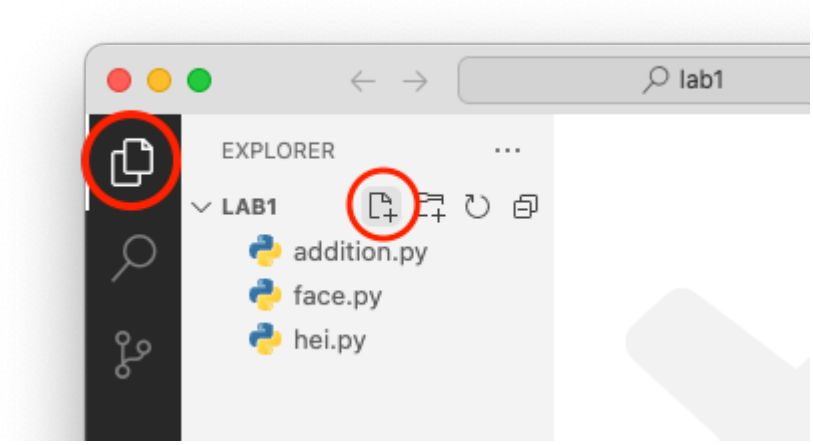
3. Når du skal jobbe med programmering, bør du åpne VSCode i en **arbeidsmappe**. De beste arbeidsmappene er *lab* -mappene (dette vil særlig vise seg når vi senere i emnet skal la programmene vi skriver jobbe med filer).



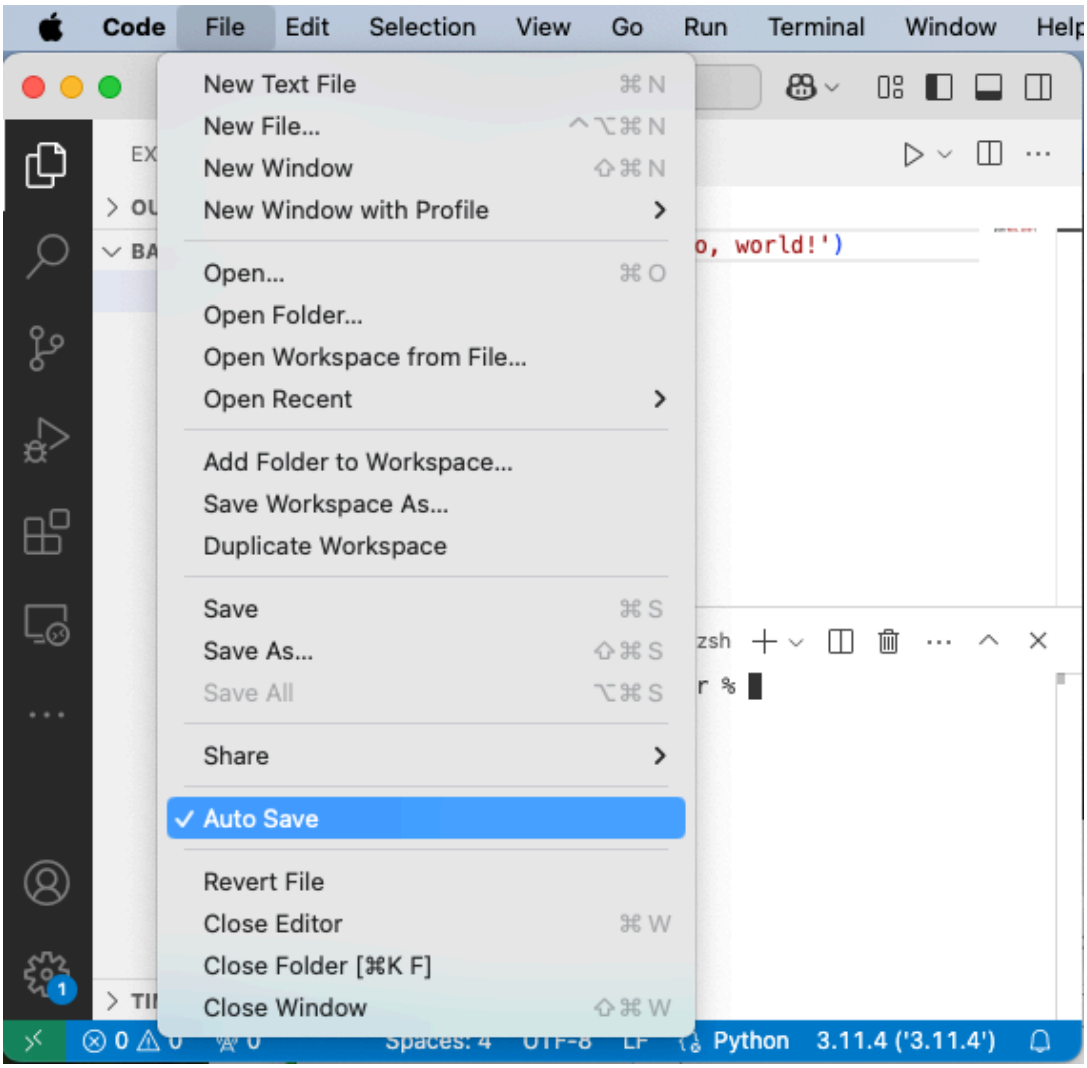


↑ Innhold

4. Når du har åpnet VSCode i en arbeidsmappe, kan du opprette nye filer ved å klikke på «ny fil» -ikonet når «explorer» -vinduet vises. Gi filen et navn (la Python-filer ha filendelsen `.py`) og trykk enter for å opprette.



5. Slå på autosave: i fil-menyen, sørg for å ha autosave aktivert, da unngår du å laste opp filer med «gammelt» innhold som ikke samsvarer med hva som vises i VSCode.



Kom i gang

↑ Innhold

Hvis du ikke har gjort det enda, må du installere Python og Visual Studio Code før du fortsetter.

- Hello World
- Kommentarer
- Utskrift til terminalen
- Python som enkel kalkulator
- Variabler
- Strenger
- Input fra terminalen
- Syntaksfeil, krasj og logiske feil

Hello World

Det er tradisjon for at det første programmet man skriver når man lærer seg et nytt programmeringsspråk er et program som skriver ut ordene «Hello World» til terminalen. I Python ser programmet slik ut:

```
print('Hello World')
```

Ordbok: funksjonskall og argumenter

Et **funksjonskall** er en instruksjon om å utføre en bestemt oppgave. Et funksjonskall består av to deler:

- *Funksjonsnavn*. I eksempelet over er `print` navnet på funksjonen.
- *Argumenter* til funksjonen, omsluttet av paranteser. Dette er verdier vi gir til funksjonen som input. I eksempelet over er `'Hello World'` argumentet som sendes til funksjonen `print`.

Antall argumenter kan variere.

- Hvis det er to eller flere argumenter, må de skilles med komma. For eksempel er `print('Hello', 'Hello again')` et funksjonskall med to argumenter.
- Hvis det er null argumenter, må det likevel være parenteser. For eksempel er `print()` et funksjonskall uten argumenter.

Kommentarer

Kommentarer er tekst i programmet vårt som blir fullstendig ignorert av Python. Alt som kommer etter en hashtag (`#`) på en linje blir ignorert, og er kommentarer.

↑ Innhold

```
print('Hello World') # Her er en kommentar
# print('På denne linjen skjer det ingenting')

print('Her er # i en streng.') # Hashtag mellom 'apostrofer' telles ikke
```

Utskrift til terminalen

Print-funksjonen skriver ut verdier til terminalen. Hver print skriver ut én linje hver.

```
print('Hello')  # Hello
print('World')  # World
print(40 + 2)   # 42
```

Flere ting kan skrives ut på sammen linje. Du kan

- bruke én print-setning med flere argumenter, eller
- legge til `end=''` som argument i print-setningen. Da kommer det ikke et linjeskift på slutten når print er ferdig.

```
# Hello World 42
print('Hello', 'World', 40 + 2)

# HelloWorld42
print('Hello', end='')
print('World', end='')
print(40 + 2)
```

Utskrift av variabler og uttrykk med f-strenger.

```
x = 2
y = 3
print(f'Visste du at {x} + {y} er {x + y}?') # Visste du at 2 + 3 er 5?

# Uten f'en foran strengen tolkes ting i {krøllparenteser} bokstavelig
print('Visste du at {x} + {y} er {x + y}?') # Visste du at {x} + {y} er {x + y}?
```

Python som enkel kalkulator

Man kan bruke Python som en enkel kalkulator for å evaluere matematiske uttrykk.

↑ Innhold

```
print(2 + 2)       # Addisjon                                --> 4
print(5 - 3)       # Subtraksjon                             --> 2
print(2 * 3)       # Multiplikasjon                          --> 6
print()

print(157 / 10)    # Divisjon                                --> 15.7
print(157 // 10)   # Heltallsdivisjon (runder alltid ned)    --> 15
print(157 % 10)    # Modulo (rest etter heltallsdivisjon)    --> 7
print()

print(3 ** 2)      # Potens/eksponentiering                  --> 9
print(max(2, 3))   # Største verdi                           --> 3
print(min(2, 3))   # Minste verdi                            --> 2
print(abs(-3))     # Absoluttverdi                           --> 3
print()

# Man kan også ha kombinerte uttrykk.
print(2 + 3 * 4)                  # 14
print(max(2, 3, 199, 4, 5) + 1)   # 200
print()

# Man kan bruke parenteser for å overstyre presedensen til operasjoner.
# Standard presedens (rekkefølgen operasjoner utføres i) er som i matematikk
print((2 + 3) * 4)                # 20
```

Variabler

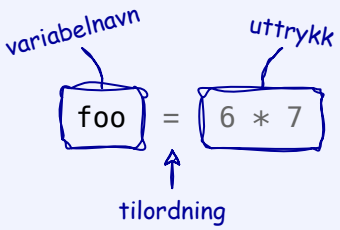
En variabel er et navn som refererer til en verdi. Etter at en variabel er angitt, kan navnet brukes i stedet for verdien den refererer til.

```
x = 2
y = 3
print(x + y)  # 5
```

```
foo = 6 * 7
print(foo)  # 42
```

↑ Innhold

*Symbolet `=` benyttes for å **tilordne verdi** til en variabel. På høyre side av `=` må det være en verdi eller et uttrykk som evaluerer til en verdi, mens venstresiden typisk er et variabelnavn.*



På en slike linje med `=` skjer to ting, i denne rekkefølgen:

1. *Høyresiden evalueres til en verdi (for eksempel vil `6 * 7` evaluere til `42`).*
2. *Variabelen på venstresiden av `=` settes til å fra nå av referere til resultatverdien (i vårt eksempel: `foo` peker fra nå av på `42`)*

Vi velger selv hvilket navn vi vil gi våre variabler. Variabelnavn kan ikke inneholde mellomrom eller spesialtegn, annet enn understrek (`_`).

```
work_hours_per_week = 12
number_of_weeks = 18
hourly_salary = 200

total_hours = work_hours_per_week * number_of_weeks
total_salary = total_hours * hourly_salary

print('Totalt antall timer:', total_hours) # Totalt antall timer: 216
print('Totalt lønn:', total_salary) # Totalt lønn: 43200
```

Strenger

En streng er en verdi som representerer tekst. En streng består av bokstaver, tallsymboler, mellomrom, linjeskift og andre tegn.

```
print('Strenger skrives mellom apostrofer.')
print("Man kan også bruke hermetegn.")
print(Uten noen form for anførselstegn er det ikke en streng, og det krasjer)
```

```
# En streng kan lagres i en variabel.
foo = 'bar'
print('foo')  # 'foo' er en streng (skriver ut: foo)
print(foo)    # foo er en variabel (skriver ut: bar)
```

↑ Innhold

Bakstreken `\` har en spesiell betydning når man angir strenger. Den benyttes for å angi tegn man vil ha i strengen sin, men hvor selve tegnet har spesiell betydning i Python; for eksempel linjeskift og apostrof.

- `\n` linjeskift
- `\'` apostrof
- `\"` hermetegn
- `\\` bakstrek

```
# En streng kan inneholde linjeskift (\n)
print('Denne strengen\ninneholder to\nlinjeskift')

# En streng kan inneholde apostrof (')
print('Her \'er\' det apostrof')
print("Her 'er' det apostrof")  # bedre løsning: bytt typen anførselstegn

# En streng kan inneholde bakstrek (\)
print('C:\\mappe\\script.py')
```

Lengden av en streng

```
# len() finner lengden til en streng
x = 'foo'
print(len(x)) # 3
print(len('foobar')) # 6
print(len('foo bar')) # 7
print(len('foo\nbar')) # 7 (linjeskift \n telles kun som 1)
print(len(' foobar  ')) # 9 (inkluderer mellomrom)
```

```
# En streng kan være «tom»
x = ''
print(len(x)) # 0
print(x) # printer en tom streng (altså ingenting) på denne linjen
print("---")
```

Operasjoner på strenger

```
# Strenger konkateneres (limes sammen) med pluss (+)
print('foo' + 'bar') # foobar

a = 'super'
b = 'duper'
c = a + b
print(c) # superduper
```

Samme operasjon (`+`) fungerer ulikt på strenger og tall:

↑ Innhold

```python
# Pluss med strenger: konkatenasjon
x = '12'
y = '34'
print(x + y) # 1234

# Pluss med tall: addisjon
x = 12
y = 34
print(x + y) # 46
```

```python
# Strenger repeteres flere ganger med gangesymbol (*) og et tall
print('*' * 9) # *********

s = 'bar'
print(2 * s) # barbar
```

f-strenger: å putte verdier inn i en streng

Det er mulig å kombinere tekst og andre verdier med *f-strenger*. Legg merke til `f` før apostrofen.

```python
# Med en f-streng kan vi fleksibelt inkludere verdier i strengen
number_of_students = 550
course_id = 'INF100'
print(f'Vi ønsker {number_of_students} studenter velkommen til {course_id}')

# Uten f får vi en logisk feil -- klarer du å forutsi hva før du trykker kjør?
print('Vi ønsker {number_of_students} studenter velkommen til {course_id}')
print()

# f-strenger kan lagres i en variabel
age = 35
message = f'Jeg er {age} år gammel.'
print(message) # Jeg er 35 år gammel.

# etter opprettelsen er f-strenger helt vanlige strenger (de er ikke magiske)
age = 40
print(message) # fremdeles «Jeg er 35 år gammel.»
```

Input fra terminalen

```python
# Les input og skriv ut en hilsen
print('Skriv ditt navn')
name = input()
print(f'Hei, {name}!')
```

↑ Innhold

Input-funksjonen returnerer alltid en *streng*, uansett hva brukeren skriver inn. Hvis vi skal bruke input fra brukeren som om det var et tall, må det konverteres til et tall med `int` (eller `float`) -funksjonen.

```
# Når input er et tall
print('Skriv et tall')
a = input() # f. eks. 4

print('Skriv et tall til')
b = input() # f. eks. 2

print('Før konvertering er a og b strenger. a + b =', a + b) # 42

# Konverterer til tall
a = int(a)
b = int(b)

print('Etter konvertering er a og b heltall. a + b =', a + b) # 6
```

Syntaksfeil, krasj og logiske feil

Kildekoden til et dataprogram kan inneholde tre typer feil:

- syntaksfeil,
- kjøretidsfeil (feil som fører til krasj), og
- logiske feil.

Syntaksfeil gjør at programmet vårt ikke starter i det hele tatt. Med litt øvelse er feilmeldingene om syntaks-feil som kommer fra Python ganske informative, og forteller oss nøyaktig hvor i programmet feilen er. For eksempel:

```
print('Programmet starter her') # Vi kommer ikke hit engang før krasjen
print('oj") # Syntaksfeil! (hermetegn matcher ikke apostrof rundt strengen)
print('Programmet er ferdig')

# Gir output:
#
#  File "foo.py", line 2
#    print('oj")
#          ^
# SyntaxError: unterminated string literal (detected at line 2)
```

```
print('Programmet starter her') # Vi kommer ikke hit engang før krasjen
print(hello world) # Syntaksfeil!
print('Programmet er ferdig')

# Gir output:
#
#   File "foo.py", line 2
#     print(hello world)
#           ^^^^^^^^^^^
# SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Kjøretidsfeil (krasj) skjer når programmet vårt slutter å kjøre midtveis i programmet fordi man forsøker å utføre en ulovlig operasjon, for eksempel ved å referere til et variabelnavn som ikke er definert eller å dele på 0. Kjøretidsfeil kan være vanskeligere å finne enn syntaksfeil, fordi feilmeldingen ikke alltid forteller oss nøyaktig hvor i programmet selve feilen er. Eksempler på kjøretidsfeil:

```
print('Programmet starter her')
hallo = 42 # Her er feilen (skrivefeil i variabelnavn)
world = 3.14
print(hello, world) # Her krasjer programmet
print('Programmet er ferdig') # Vi kommer ikke hit

# Gir output:
#
# Programmet starter her
# Traceback (most recent call last):
#   File "foo.py", line 4, in <module>
#     print(hello, world)
#           ^^^^^
# NameError: name 'hello' is not defined.
```

```
print('Programmet starter her') # Denne linjen utføres
print('42' + 3) # Krasj! (å legge sammen en streng og et tall er umulig)
print('Programmet er ferdig') # Vi kommer ikke hit

# Gir output:
#   TypeError: can only concatenate str (not "int") to str
```

```
print(1/0) # Krasj! (å dele på 0 er umulig)

# Gir output:
#   ZeroDivisionError: integer division or modulo by zero
```

Logiske feil er den kjipeste formen for feil, fordi den er så vanskelig å oppdage. I verste fall oppdages ikke den logiske feilen før det har oppstått store kostnader et annet sted i verdikjeden (for eksempel hos en

kunde).

↑ Innhold

```
x = 2
y = 3
z = x + y
print(f"{x} + {z} = {y}") # Logisk feil, vi har byttet z og y

# Gir output:
#   2 + 5 = 3
```

Variabler og uttrykk

↑ Innhold

- Variabler
- Tilordning
- Uttrykk og setninger
- Variabler refererer til verdier, ikke til uttrykk
- Variabelnavn

Variabler

En *variabel* er en referanse til en verdi. En variabel kan tilordnes verdi med symbolet `=`. Variabelen vil deretter referere til denne verdien frem til den eventuelt tilordnes en ny verdi.

```
# Vi tilordner verdien 5 til variabelen x
x = 5
print(x)      # 5
print(x + 2)  # 7
print(x)      # fremdeles 5
```

Til forskjell fra matematikk slik vi vanligvis kjenner den, kan en variabel i Python endre verdi over tid.

```
y = 10      # y refererer nå til 10
y = 20      # y refererer nå til 20, forrige verdi (10) er helt glemt
print(y)    # 20
print(y)    # fremdeles 20

# Når en variabel angis, evalueres uttrykket på høyre side først; deretter
# tilordnes resultatet til variabelen angitt på venstre side

y = y + 1   # y refererer nå til 21, forrige verdi (20) er helt glemt
print(y)    # 21
```

*Legg merke til at uttrykket `y = y + 1` er en fullstending meningsfull setning å skrive i Python, mens det ville vært en selvmotsigelse å skrive det samme i matematikk. I Python skal setningen tolkes slik: «regn ut y + 1, og **deretter** tilordne denne verdien til y.»*

Tilordning

En variabel tilordnes verdi ved bruk av tilordningsoperasjonen `=`. Det er også mulig å kombinere tilordningsoperasjonen med operasjonssymboler, slik som f. eks. `+=` og `-=`.

```
x = 5
x += 2 # det samme som x = x + 2
print(x) # 7

y = 5
y + 2 # ingen tilordnings-operasjon her
print(y) # fremdeles 5
```

Tilordningsoperatøren kan kombineres med alle symbol-operasjonene.

```
x = 42

x += 2       # det samme som  x = x + 2
x -= 3 - 1   # det samme som  x = x - (3 - 1)
x //= 10     # det samme som  x = x // 10
x **= 5 // 2 # det samme som  x = x ** (5 // 2)
x *= 2 + 3   # det samme som  x = x * (2 + 3)
x %= 3       # det samme som  x = x % 3
x /= 3       # det samme som  x = x / 3
```

Når man tilordner en verdi, må venstre side av tilordningsoperatøren være variabelnavn.

```
x = 5      # OK
5 = x      # Feil (venstre side kan ikke være en verdi)
x + 2 = 7  # Feil (venstre side er kan ikke være et uttrykk)
```

Det er mulig å tilordne flere variabler verdi på samme linje.

```
x = y = z = 5   # alle tre variablene x, y og z refererer nå til verdien 5
print(x, y, z)  # 5 5 5

a, b = 2, 3     # a refererer nå til verdien 2, b refererer nå til verdien 3
print(a, b)     # 2 3
```

Ikke pensum, men kjekt å vite: fra og med python 3.8 er det også mulig å tilordne en variabel verdi *i et uttrykk* med bruk av `:=` («hvalross»-operatøren).

```
x = 37 + (y := 3 + 2)
print(x, y) # 42 5
```

Uttrykk og setninger

Et **uttrykk** er et «regnestykke» som kan evalueres til en verdi. Det består av én eller flere verdier, variabler, operasjoner og funksjoner. Her er noen eksempler på uttrykk; både svært banale uttrykk og mer komplekse

(vi antar at x, y og num_students er variabelnavn som allrede er tilordnet verdier).

↑ Innhold

Eksempler:

- `42`
- `'Hei verden'` (enkelstående verdier er de enkleste uttrykkene)
- `x`
- `y` (variabelnavn er uttrykk som evaluerer til sin angitte verdi)
- `len(x)`
- `min(2, y)` (funksjonskall er uttrykk)
- `f'Det er {y} grader ute'` (f-strenger er uttrykk)
- `3 + 2`
- `y > 5` (uttrykk kan kombineres med operatorer)
- `(len(x) > 5) and (y < 10)`
- `y + min(0, max(len(x) - y, 100 - y))`

En **setning** er ett steg (ofte én linje) i et Python -program, og representerer en handling og ikke en verdi. For eksempel tilordningen av en verdi til en variabel.

- En linje `x = x + 1` er en setning, mens `x + 1` er et uttrykk.
- En linje `print('Hallo')` er en setning, men om vi ser på kun `'Hallo'` i den samme linjen er det et uttrykk.

Variabler referer til verdier, ikke til uttrykk

Det som står på høyresiden av tilordningsoperatøren når vi tilordner verdi til en variabel er et uttrykk. Variabelen vil være en referanse til den ferdig evaluerte verdien, og vil *ikke* huske hvordan uttrykket så ut.

```
x = 5
y = x + 2
print(x) # 5
print(y) # 7

x += 10
print(x) # 15
print(y) # fremdeles 7
```

Variabelnavn

Vi bestemmer selv hva variablene skal hete, så lenge de begynner med en bokstav og ikke inneholder spesielle tegn (bortsett fra `_`).

```
num_students = 800
num_group_leaders = 42
course_code = 'INF100'
99problems = True  # Krasjer, variabelnavn kan forresten ikke begynne med tall
```

↑ Innhold

Det er viktig å gi variablene våre navn som tydelig beskriver hva slags verdi de representerer (gjerne inkludert hvilken enhet det er). Dette gjør det enklere å lese og forstå koden. Gode variabelnavn gjør at koden vår blir *selvdokumenterende*.

> *INF100 sin stilguide for variabelnavn:*
>
> 1. *Bruk selvdokumenterende variabelnavn.*
> 2. *Bruk engelske variabelnavn.*
> 3. *Bruk snake_case: bare små bokstaver, understrek (_) for å skille mellom ord.*
> 4. *Ikke benytt variabelnavn som er opptatt fra før av i Python.*

1. Bruk selvdokumenterende variabelnavn. Dette er det viktigste punktet ved valg av variabelnavn.

```
# Dårlig
s = 'L'
```

```
# Bra
shirt_size = 'L'
```

2. Bruk engelske variabelnavn. I og med at Python er basert på engelske nøkkelord, er koden ofte lettere å lese ved å benytte engelske variabelnavn. Dessuten gjør det koden mer tilgjengelig for kolleger som kanskje ikke forstår norsk. Til sist er de også lurt å unngå spesielle symoboler som æ, ø og å i variabelnavn; hvorfor kan du lese mer om når vi kommer til unicode og tekstkoding senere i emnet. Selv om vi anbefaler engelske variabelnavn, godtar vi også variabelnavn på norsk; men vær konsekvent.

```
# Nja, vær i så fall konsekvent
antall_mennesker = 42
```

```
# Bra
number_of_people = 42
```

3. Bruk små bokstaver og bruk understrek (`_`) for å skille mellom ord, såkalt `snake_case`. Merk at du i noen eksterne pakker kanskje vil se andre konvensjoner, som `lowerCamelCase` (ansett for å være god stil i en noen andre programmeringsspråk). I INF100 holder vi oss derimot til `snake_case` for variabelnavn vi døper selv, slik den offisielle dokumentasjonen for Python anbefaler.

```
# Nja, vær i så fall konsekvent
numberOfPeople = 42
```

```
# Bra
number_of_people = 42
```

Noen ganger oppretter vi en variabel som får en fast verdi når programmet starter, og som deretter aldri skal endre verdi. Dette er en såkalt *konstant*. Det er god stil å la konstanter være skrevet i `UPPER_CASE`.

```
# Dårlig
minimum_age = 12
```

```
# Bra
MINIMUM_AGE = 12
```

↑ Innhold

> *Du kan også støte på `PascalCase` noen steder, for eksempel i eksterne pakker. Det er god stil å benytte denne navneformen når man skal navngi såkalte «klasser»; men dette er utenfor pensum i INF100, og du bør derfor ikke benytte denne formen når du navngir noe i dette emnet.*

4. Ikke benytt et innebygd nøkkelord eller funksjonsnavn fra Python som variabelnavn.

De mest vanlige tabbene er å bruke `input`, `len`, `sum`, `abs`, `min` eller `max` som variabelnavn. Du kan se at disse ordene er spesielle fordi VSCode fargelegger dem annerledes enn andre variabelnavn. Dette er fordi de er reservert for spesielle formål i Python, og det er derfor en dårlig idé å bruke dem til andre ting.

- Nøkkelord med spesielle betydninger i Python 3 er: `False` `None` `True` `and` `as` `assert` `break` `class` `continue` `def` `del` `elif` `else` `except` `finally` `for` `from` `global` `if` `import` `in` `is` `lambda` `nonlocal` `not` `or` `pass` `raise` `return` `try` `while` `with` `yield`.
- Innebygde funksjoner i Python 3 er f. eks: `abs` `bool` `float` `input` `int` `len` `max` `min` `print` `sum` `str` `type`. For uttømmende liste, se https://docs.python.org/3/library/functions.html.

Typer

↑ Innhold

- Vanlige typer
- Typen avgjør hva en operasjon betyr
- Typekonvertering
- Enkle eksempler på bruk av ulike typer

Vanlige typer

Enhver verdi har en *type* (også kalt *klasse*). Det er mange ulike typer som er innebygget i Python. For å se hvilken type en verdi har, kan vi benytte en funksjon som heter `type`. Her er en oversikt over de aller viktigste typene, som vi hele tiden støter på i Python:

```
print('Noen elementære typer i Python:')
print(type('foo'))        # str       (streng/tekst)
print(type(2))            # int       (heltall)
print(type(2.2))          # float     (flyttall/desimaltall)
print(type(True))         # bool      (boolsk verdi; True eller False)
print(type(None))         # NoneType  («ingenting» -verdien har egen type)
print(type([1, 2, 3, 4])) # list      (en liste av andre verdier)
print()

print('Flere viktige typer vi skal lære om senere')
print(type((1, 2, 3, 4))) # tuple
print(type({1, 2, 3, 4})) # set
print(type({1: 2, 3: 4})) # dict
```

Typen avgjør hva en operasjon betyr

```
# Asterisk (*) betyr forskjellige ting (f.eks. multiplikasjon el. repetisjon)
print(3 * 2)              # 6
print(3 * 'abc')          # 'abcabcabc'
print(3 * [1, 2, 3])      # [1, 2, 3, 1, 2, 3, 1, 2, 3]

# Plusstegn (+) betyr forskjellige ting (f.eks. addisjon eller konkatenasjon)
print(3 + 2)               # 5
print('3' + '2')           # '32'
print([1, 2, 3] + [4, 5])  # [1, 2, 3, 4, 5]
```

```
# Ikke alle operasjoner er definert for alle (kombinasjoner av) typer
print(3 + 'def')          # Krasjer med TypeError
print('abc' * 'def')      # Krasjer med TypeError
```

↑ Innhold

Typekonvertering

Vi kan konvertere mellom typer ved å bruke funksjoner med samme navn som typen.

```
# Konvertering fra streng
s = '42'
i = int(s) # Konverterer fra streng til heltall
f = float(s) # Konverterer fra streng til desimaltall
a = list(s) # Konverterer fra streng til liste

print(s) # 42
print(i) # 42
print(f) # 42.0
print(a) # ['4', '2']
print()

print(type(s)) # <class 'str'>
print(type(i)) # <class 'int'>
print(type(f)) # <class 'float'>
print(type(a)) # <class 'list'>
```

```
# Konvertering flyttall -> heltall -> flyttall
x = 4.9
y = int(x) # runder av nedover (mot negativ uendelig)
z = float(y) # konverterer tilbake til flyttall

print(x) # 4.9
print(y) # 4
print(z) # 4.0

print(type(x)) # <class 'float'>
print(type(y)) # <class 'int'>
print(type(z)) # <class 'float'>
```

Enkle eksempler på bruk av ulike typer

int	`x = 5`
	`y = -3`
	`z = x + y`
	`print(z) # 2`

float	`x = 5.0`
	`y = -3.2`
	`print(x + y) # 1.8`

Alle heltall kan brukes som om de var flyttall.	`x = 5`
	`y = -3.2`
	`print(x + y) # 1.8`

bool	`x = True`
	`y = False`
	
	`print(x and y) # False`
	`print(x or y)  # True`
	`print(not x)  # False`

Når man sammenligner ting med hverandre, får vi alltid en boolsk verdi tilbake.	`x = 2 < 3`
	`y = 2 > 3`
	`print(x)  # True`
	`print(y)  # False`
	
	`s = 'abc'`
	`print(s == 'def')  # False`
	`print(s != 'def')  # True`

→ Innhold

str

```
s = 'abc'
print(s + 'def') # 'abcdef'
print(s * 3)     # 'abcabcabc'

# Hente ut enkelt-tegn fra strengen
x = s[0]
print(x) # 'a'

y = s[2] + s[1]
print(y) # 'cb'
```

list

```
a = ['foo', 42, 3.14]

print(a) # ['foo', 42, 3.14]
print(type(a)) # <class 'list'>

# Hente ut enkelt-element fra listen
x = a[0]
print(x) # foo
print(type(x)) # <class 'str'>

y = a[1] + a[2]
print(y) # 45.14
print(type(y)) # <class 'float'>

# Endre verdi til en posisjon i listen
a[0] = 'bar'
print(a) # ['bar', 42, 3.14]

# Legger til et nytt element i listen
a.append('baz')
print(a) # ['bar', 42, 3.14, 'baz']
```

Operatorer

↑ Innhold

Kategori	Operatorer
Aritmetikk • Artitmetikk med tall • Artitmetikk med strenger • Heltallsdivisjon og modulo	`**`
	`*` `/` `//` `%`
	`+` `-`
Relasjoner • Sammenligning av verdier • Flyttall og avrundingsfeil • Medlemsskap	`==` `!=` `<` `<=` `>` `>=` `in` `not in` `is` `is not`
Logikk	`not`
	`and`
	`or`
Betingelse	`if … else`

- Presedens og assosiativitet
- Eksempler

Aritmetikk med tall

Symbolene `+` `-` `*` `/` `**` utfører henholdsvis addisjon, subtraksjon, multiplikasjon, divisjon og eksponentiering med to tall.

Alle operatorene fungerer likt for både heltall (int) og flyttall (float), men merk at divisjon `/` alltid returnerer et flyttall, selv om resultatet numerisk sett er et heltall.

```
print(6 + 2) # 8
print(6 - 2) # 4
print(6 * 2) # 12
print(6 / 2) # 3.0
print(6 ** 2) # 36
print(36 ** 0.5) # 6.0 (en eksponent på 0.5 er det samme som kvadratrot)
```

↑ Innhold

Aritmetikk med strenger

```
# Strenger repeteres flere ganger med gangesymbol (*) og et heltall
print('bar' * 2) # barbar
```

```
# Strenger konkateneres (limes sammen) med pluss (+)
a = 'foo'
b = 'bar'
c = a + b
print(c) # foobar
```

Heltallsdivisjon og modulo

Heltallsdivisjon med restverdi er den første formen for divisjon vi lærte på barneskolen. La oss si at du skal fordele 14 gullmynter på 4 pirater: da kan hver pirat få 3 gullmynter, og så blir det 2 gullmynter til overs. Vi kan uttrykke regnestykket i Python ved å benytte operatorene for heltallsdivisjon `//` og modulo `%` slik:

```
coins = 14
pirates = 4
coins_per_pirate = coins // pirates
remainder = coins % pirates

print(coins, 'gullmynter skal fordeles på', pirates, 'sjørøvere.')
print('Hver sjørøver får da', coins_per_pirate, 'gullmynter',
      'og det blir', remainder, 'mynter til overs.')
```

Heltallsdivisjon er som vanlig divisjon, men runder alltid *nedover* (mot negativ uendelig).

```
print('Operatøren / utfører vanlig divisjon')
print(' 7 / 4  =', (7/4))  # 1.75
print()
print('Operatøren // utfører heltallsdivisjon:')
print(' 7 // 4 =', ( 7//4)) # 1 (runder nedover)
print('-1 // 4 =', (-1//4)) # -1 (runder også nedover)
print('-7 // 4 =', (-7//4)) # -2 (runder også nedover)
print('Når nevneren er negativ')
print(' 7 // -4 =', (7//-4)) # -2 (runder også nedover)
print('-7 // -4 =', (-7//-4)) # 1 (runder altså alltid nedover uansett)
```

Modulo-operatoren `%` returnerer «resten» etter heltallsdivisjon. Tallet sier noe om hvor langt unna dividenden er å være delelig med divisoren; hvor mye man får til overs. Vi kan bruke dette for å avgjøre om et tall er delelig med et annet (hvis resten er 0, er det delelig).

↑ Innhold

```
print(8 % 3) # 2
print(7 % 3) # 1
print(6 % 3) # 0    6 er delelig med 3, derfor må 6 % 3 være 0
print(5 % 3) # 2
print(4 % 3) # 1
print(3 % 3) # 0    3 er delelig med 3
print(2 % 3) # 2
print(1 % 3) # 1
print(0 % 3) # 0    0 er delelig med 3
print(-1 % 3) # 2
print(-2 % 3) # 1
print(-3 % 3) # 0   -3 er delelig med 3
print(-4 % 3) # 2
print(-5 % 3) # 1
print(-6 % 3) # 0   -6 er delelig med 3
```

En vanlig bruk av modulo er å finne siste siffer i et tall:

```
print(123 % 10) # 3
print(1234 % 10) # 4
print(12345 % 10) # 5
```

Eller avgjøre om et tall er oddetall eller partall. Partall er som kjent delelig med 2:

```
num = 42
is_num_even = (num % 2) == 0
print(f'{num} er et partall: {is_num_even}')
```

En annen vanlig bruk av modulo er for å få en verdi til å «gå i ring» innenfor et visst intervall. For eksempel, la oss si at vi ønsker at x-koordinatet til en ball alltid skal befinne seg innenfor et vindu med bredde på 400 piksler; hvis ballen faller utenfor, vil vi at den skal komme tilbake på den andre siden av vinduet. Da kan vi bruke modulo-operatoren til å «wrappe» verdien tilbake til intervallet dersom den skulle komme utenfor:

```
x = 385

...

x = (x + 10) % 400
print(x) # 395 (som forventet når vi gjør 385 + 10)

...

x = (x + 10) % 400
print(x) # tilbake til 5 i stedet for 405, fordi 405 % 400 blir 5
```

↑ Innhold

Sammenligning av verdier

Relasjons-operatorene benyttes for å sammenligne to verdier, og resulterer alltid i en boolsk verdi (enten `True` eller `False`).

```
print('== sjekker om to verdier er like')
print(2 == 2) # True
print(2 == 3) # False
print(2 == 2.0) # True
print('foo' == 'foo') # True
print('foo' == 'bar') # False
print()

print('!= sjekker om to verdier er ulike')
print(2 != 2) # False
print(2 != 3) # True
print(2 != 2.0) # False
print('foo' != "foo") # False
print('foo' != 'bar') # True
```

```
print('< sjekker om venstre side er «mindre enn» høyre side')
print(2 < 3) # True
print(2 < 2) # False
print(2 < 1) # False
print('foo' < 'barbar') # False (sammenligner «ASCII-alfabetisk»)
print('foo' < 'foo') # False
print('barbar' < 'foo') # True
print()

print('<= sjekker om venstre side er «mindre enn eller lik» høyre side')
print(2 <= 3) # True
print(2 <= 2) # True
print(2 <= 1) # False
print('foo' <= 'barbar') # False (sammenligner «ASCII-alfabetisk»)
print('foo' <= 'foo') # True
print('barbar' <= 'foo') # True
```

Operatorene `>` og `>=` fungerer likt som `<=` og `<` bare med motsatt resultat.

Flyttall og avrundingsfeil

Se også videoen Floating point numbers (https://www.youtube.com/watch?v=PZRI1IfStY0) av Computerphile.

```
print(0.1 + 0.1 == 0.2)        # True, men...
print(0.1 + 0.1 + 0.1 == 0.3)  # False!
print(0.1 + 0.1 + 0.1)         # gir 0.30000000000000004 (oj sann!)
print((0.1 + 0.1 + 0.1) - 0.3) # gir 5.55111512313e-17 (lite, men ikke 0!)
```

↑ Innhold

Derfor: Ikke bruk `==` for å sammenligne flyttall! Sjekk i stedet at de to tallene som sammenlignes er *nesten* like.

```
def almost_equals(a, b):
    epsilon = 0.0000000001
    return abs(a - b) < epsilon # abs()-funksjonen gir absolutt-verdien

print(0.1 + 0.1 + 0.1 == 0.3)  # False
print(almost_equals(0.1 + 0.1 + 0.1, 0.3))  # True
```

Medlemskap

Operatorene `in` og `not in` brukes for å sjekke om en verdi er medlem av en liste, tuple, mengde, streng eller en annen samling av ting.

```
# Sjekk om symboler finnes i strenger
print('a' in 'abc') # True
print('d' in 'abc') # False
print('A' in 'abc') # False ('A' og 'a' er forskjellige symboler)

print('a' not in 'abc') # False
print('d' not in 'abc') # True
print()

print('bc' in 'abc') # True ('bc' utgjør en sammenhengende del av 'abc')
print('ac' in 'abc') # False (selv om både 'a' og 'c' er i 'abc')
```

```
# Sjekk om en verdi finnes i en liste eller ikke
print(1 in [1, 2, 3]) # True
print(4 in [1, 2, 3]) # False
print(1 not in [1, 2, 3]) # False
print(4 not in [1, 2, 3]) # True
```

```
# Sjekk om en verdi finnes i en tuple eller ikke
print(1 in (1, 2, 3)) # True
print(4 in (1, 2, 3)) # False
print(1 not in (1, 2, 3)) # False
print(4 not in (1, 2, 3)) # True
```

```
# Sjekk om en verdi finnes i en mengde eller ikke
print(1 in {1, 2, 3}) # True
print(4 in {1, 2, 3}) # False
print(1 not in {1, 2, 3}) # False
print(4 not in {1, 2, 3}) # True
```

↑ Innhold

Logiske operatorer

For å bruke boolske verdier (True og False) i logiske uttrykk, bruker vi boolske operatorer. Disse operatorene er konjunksjon `and`, disjunksjon `or` og negasjon `not`.

x	not x
False	True
True	False

x	y	x or y	x and y
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Uttrykk som består av boolske uttrykk og logiske operatorer er en del av hva vi kaller «boolsk algebra». Bruken av `and` og `or` i slik logikk minner om det vi er vant til fra dagligtalen. Legg merke til at svaret på et spørsmål som «er du voksen *eller* er du barn?» alltid vil være `True` (med mindre du verken voksen eller barn, så klart; bare da er svaret `False`).

```
print('and returnerer True hvis begge leddene er True')
print(True and True) # True
print(True and False) # False
print(False and True) # False
print(False and False) # False
print()

print('or returnerer True hvis minst ett av leddene er True')
print(True or True) # True
print(True or False) # True
print(False or True) # True
print(False or False) # False
print()

print('not returnerer motsatt boolsk verdi')
print(not True) # False
print(not False) # True
```

Det er vanlig å benytte logiske operatorer for å binde sammen flere uttrykk som hver for seg evaluerer til boolske verdier.

```
age = 2000
if (age < 0) or (age > 130):
    print(f'Litt vanskelig å tro at du er {age} år gammel...')
```

Les mer om hvordan logiske uttrykk evalueres, kortslutning, og hva `and` og `or` egentlig returnerer i kursnotater om boolske uttrykk.

↑ Innhold

Uttrykk med betingelse (if else)

```
# Eksempel på uttrykk med betingelse
age = 20
status = 'et barn' if age < 18 else 'en voksen'
print(f'Alder: {age}. Dette er {status}.')  # Alder: 20. Dette er en voksen.

# Generelt
condition = True  # or False
value_if_true = 'a'
value_if_false = 'b'
result = value_if_true if condition else value_if_false
print(result)  # a
```

Presedens og assosiativitet

En vanlig feil er at man gjør gale antakelser om hvilken rekkefølge operatorer i et større uttrykk utføres i; hvordan de «usynlige parentesene» i uttrykket er plassert.

For aritmetikk gjelder de samme presedens-reglene som er vanlig i matematikk.

```
print('Presedens:')
print(2 + 3 * 4)  # gir 14, ikke 20   ( * har høyere presedens enn + )
print(5 + 4 % 3)  # gir  6, ikke 0    ( % har høyere presedens enn + )
print(2 ** 3 * 4) # gir 32, ikke 4096 (** har høyere presedens enn * )
```

Presedenstabell

Vi gir her en oversikt over noen operatorer rangert fra høyeste til laveste presedens. Parenteser vil alltid overstyre presedens og assosiativitet, og er derfor øverst på listen. Alle operatører untatt `**` og relasjonene assosierer venstre-til-høyre dersom det er aktuelt.

↑ Innhold

Operatorer	
`()`	Parentes. Det som er mellom parentesene evalueres før en operator utenfor parentesen.
`**`	Eksponentiering. Assosierer høyre-til-venstre.
`*` `/` `//` `%`	Multiplikasjon, divisjon og modulo.
`+` `-`	Addisjon og subtraksjon.
`<` `<=` `>` `>=` `==` `!=` `in` `not in` `is` `is not`	Relasjoner. Disse vil *ikke* assosiere verken til høyre eller venstre; dersom man har flere slike etter hverandre vil de *komponeres* som en konjunksjon i stedet. For eksempel, `-1 < 0 == False` gir det samme svaret som `(-1 < 0) and (0 == False)`, og altså ikke det samme som `(-1 < 0) == False` slik man ellers kunne trodd.
`not`	Logisk negasjon.
`and`	Logisk konjunksjon.
`or`	Logisk disjunksjon.
`if … else`	Betinget verdi.

I presedenstabellen over (og også i tabellen øverst på denne siden) er operatorene sortert etter presedens. Det vil si, `**` har høyeste presedens mens `if … else` har den laveste. Operatorer i samme rad har samme presedens (for eksempel har `+` og `-` samme presedens).

Det er operatorene med høyest presedens som utføres «først» med mindre parenteser indikerer noe annet.

Assosiativitet

Dersom flere operasjoner med samme presedens forekommer i samme uttrykk, utføres de som hovedregel fra venstre til høyre.

```
print("Assosiativitet: venstre til høyre")
print(5 - 4 - 3)  # det samme som (5 - 4) - 3, altså -2 (ikke 4)
print(9 // 3 // 3) # det samme som (9 // 3) // 3, altså 1 (ikke 9)
```

Det finnes likevel noen unntak:

- `**` er høyre-assosiativ (det vil si at `2 ** 3 ** 4` er det samme som `2 ** (3 ** 4)`)
- Relasjonene (altså `==` `!=` `<` `<=` `>` `>=` `in` `not in` `is` `is not`) assosierer hverken til høyre eller til venstre; dersom man har flere slike i et uttrykk vil de *komponeres som en konjunksjon* i stedet. For eksempel, `-1 < 0 == False` vil tolkes som `(-1 < 0) and (0 == False)`, og altså ikke som `(-1 < 0) == False`.

↑ Innhold

Parenteser vil alltid overstyre presedens og assosiativitet. Det er god stil å bruke parenteser for å vise hvilken rekkefølge du ønsker at operatorene utføres i, selv om det ikke alltid er nødvendig – det gjør koden din mer lesbar og mindre utsatt for feil som skyldes at du ikke husker presedenstabellen.

Eksempler

Under er det noen eksempler hvor det er fort gjort å feiltolke hvordan uttrykket evalueres fordi det ikke er angitt parenteser. Bruk reglene for presedens og assosiativitet kombinert med presedenstabellen og prøv å forutsi hva hvert uttrykk evaluerer til før du kjører koden og ser fasiten.

```
print(True or True and False)
print(not False or True)
print(not (False or True))
print()
print(2 < 3 < 4)
print(not 3 < 2 < 1)
print(not 3 < 2 and 2 < 1)
print()
print('b' in 'box')
print('b' in 'box' == True)
print('b' in 'box' == False)
print('a' and 'b' in 'box')
print('a' or 'z' in 'box')
```

For å forstå de to siste eksemplene, se hva and og or egentlig returnerer.

Moralen i historien: ***benytt parenteser*** *for å vise hva du mener. Det er fort gjort å huske feil rekkefølge, og du kan heller ikke forvente at dine kolleger og ditt fremtidige jeg (som senere skal vedlikeholde koden) husker den.*

Betingelser

↑ Innhold

- If -setninger
- Betingelser
- If-else
- If-elif-else
- If-else -uttrykk
- Truthy og falsy verdier
- God stil

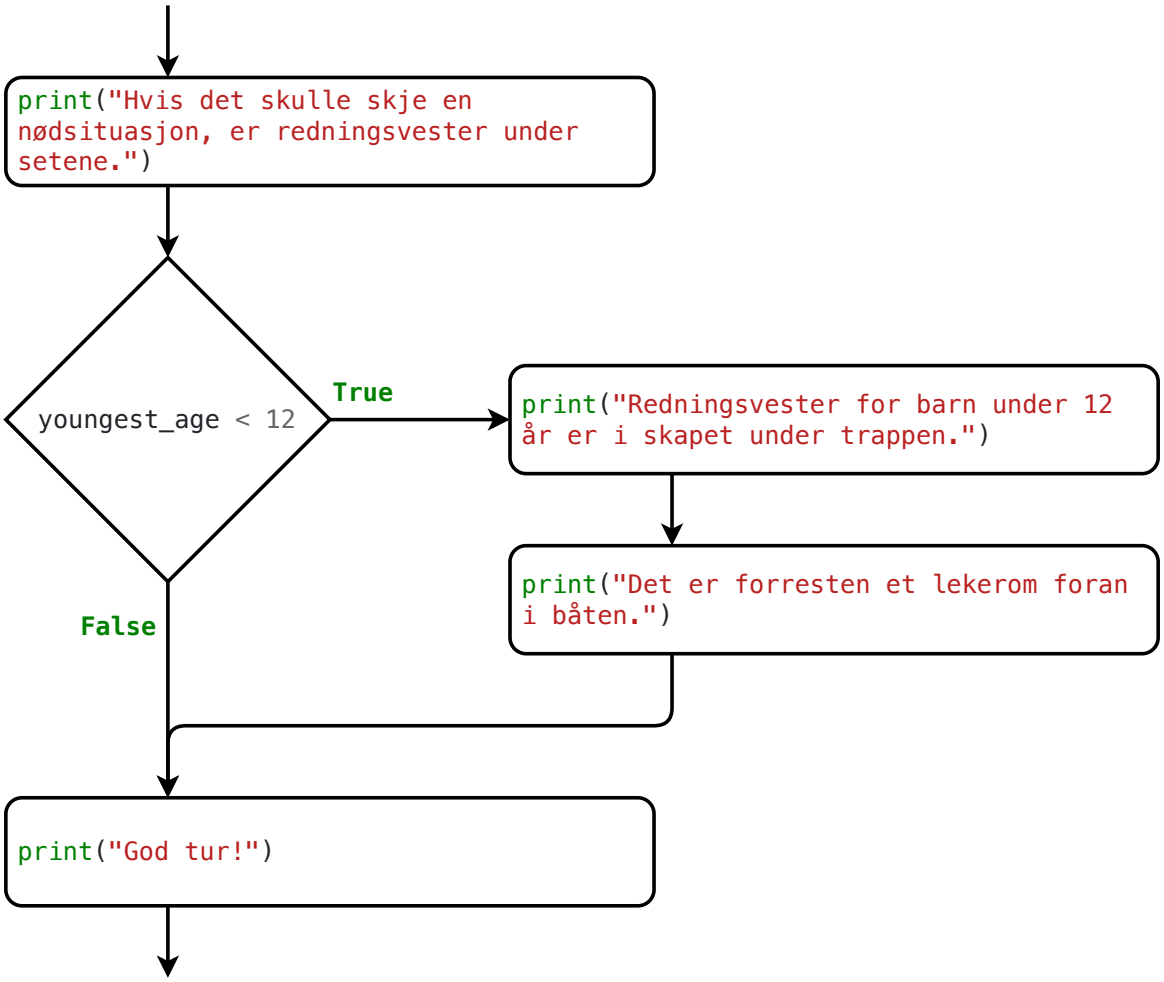
If -setninger

Man kan benytte en if-setning for å utøre en kodeblokk kun i gitte tilfeller.

```
print("Hva er alderen til den yngste reisende?")
youngest_age = int(input())

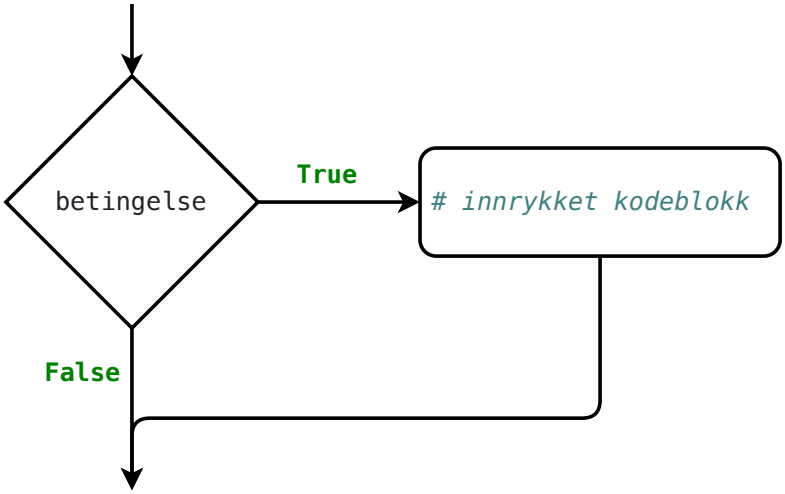
print("Velkommen om bord!")
print("Hvis det skulle skje en nødsituasjon, er redningsvester under setene.")
if youngest_age < 12:
    print("Redningsvester for barn under 12 år er i skapet under trappen.")
    print("Det er forresten et lekerom foran i båten.")
print("God tur!")
```

↑ Innhold



Syntaks. For å benytte en if-setning, må vi benytte kodeordet `if` fulgt av en *betingelse*, etterfulgt av et kolon. Deretter må vi skrive koden som skal utføres dersom betingelsen er oppfylt med et *innrykk*. God stil tilsier at innrykket består av 4 mellomrom.

```
if betingelse:
    # innrykket kodeblokk
```



↑ Innhold

Betingelser

En betingelse er et uttrykk som evaluerer til enten `True` eller `False`. Dersom betingelsen er `True`, vil koden i kodeblokken utføres. Dersom betingelsen er `False`, vil koden i kodeblokken ikke utføres.

Alle *relasjonsoperatorer* evaluerer til `True` eller `False` (aka *boolske* verdier), og er derfor egnet til betingelser. Eksempler på relasjonsoperatorer er `==`, `!=`, `<`, `>`, `<=`, `>=`, `in` og `not in` (les mer i kursnotater om operatorer). Eksempler på betingelser:

```
# Betingelser som evaluerer til True
if True:
    print("A")

x = True
if x:
    print("B")

if 2 + 2 == 4:
    print("C")

x = "yes"
if x == "yes":
    print("D")

x = 2
y = 3
if x < y:
    print("E")

# Betingelser som evaluerer til False
if False:
    print("F")

x = False
if x:
    print("G")

if 2 + 2 == 5:
    print("H")

x = 2
if x == 3:
    print("I")

x = 2
y = 3
if x > y:
    print("J")
```

↑ Innhold

I tillegg er de *logiske operatorene* (`not` , `and` , `or`) nyttige for å kombinere eller negere boolske verdier (les mer i kursnotater om operatorer og boolske uttrykk). Eksempler på betingelser med logiske operatorer:

```
age = 18
residency = "Norway"
if (age >= 18) and (residency == "Norway"):
    print("Du kan stemme ved fylkes- og kommunevalget.")

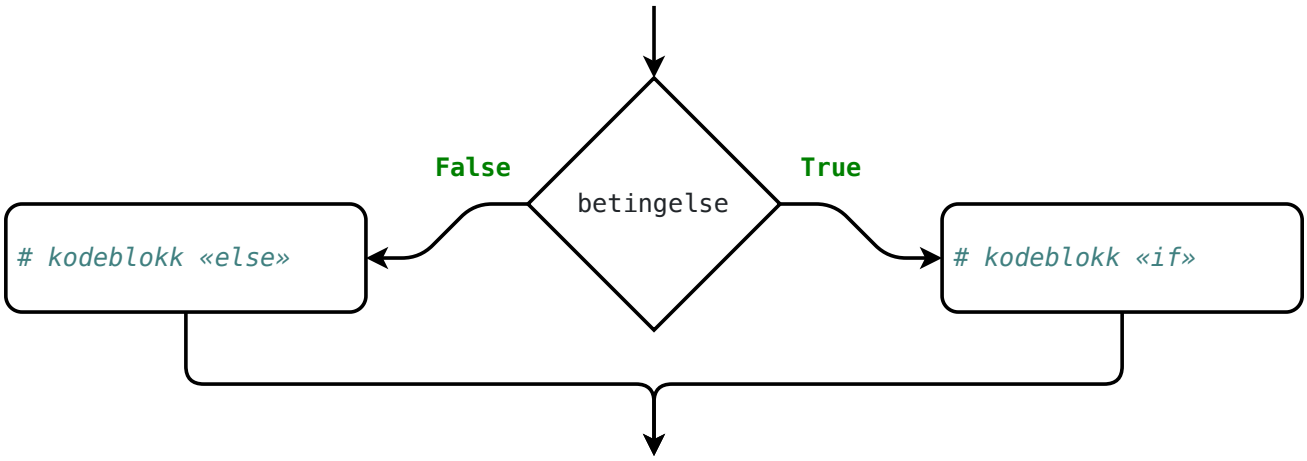
is_criminal = False
physical_test_score = 8
if (not is_criminal) and (physical_test_score >= 6):
    print("Du kan søke på jobb i politiet.")

height = 2.1
if (height < 1.5) or (height > 2.0):
    print("Beklager, du får ikke lov å kjøre berg-og-dal-banen.")
```

If-else

I en if-else vil programflyten velge én av to mulige flyter videre gjennom programmet, før flyten samles igjen. Dersom betingelsen er `True` , vil kodeblokken etter `if` -setningen utføres, men hvis betingelsen er `False` , vil kodeblokken etter `else` -ordet utføres.

```
if betingelse:
    # kodeblokk «if»
else:
    # kodeblokk «else»
```



Eksempel:

↑ Innhold

```
print("I hvilken kommune var du bostedsregistrert 30. juni i år?")
your_municipality = input()

if your_municipality == "Bergen":
    print("På valgdagen 11. september kan du stemme i et valglokale i Bergen.")
else:
    print("For å avgi stemme mens du er i Bergen, må du forhåndsstemme.")

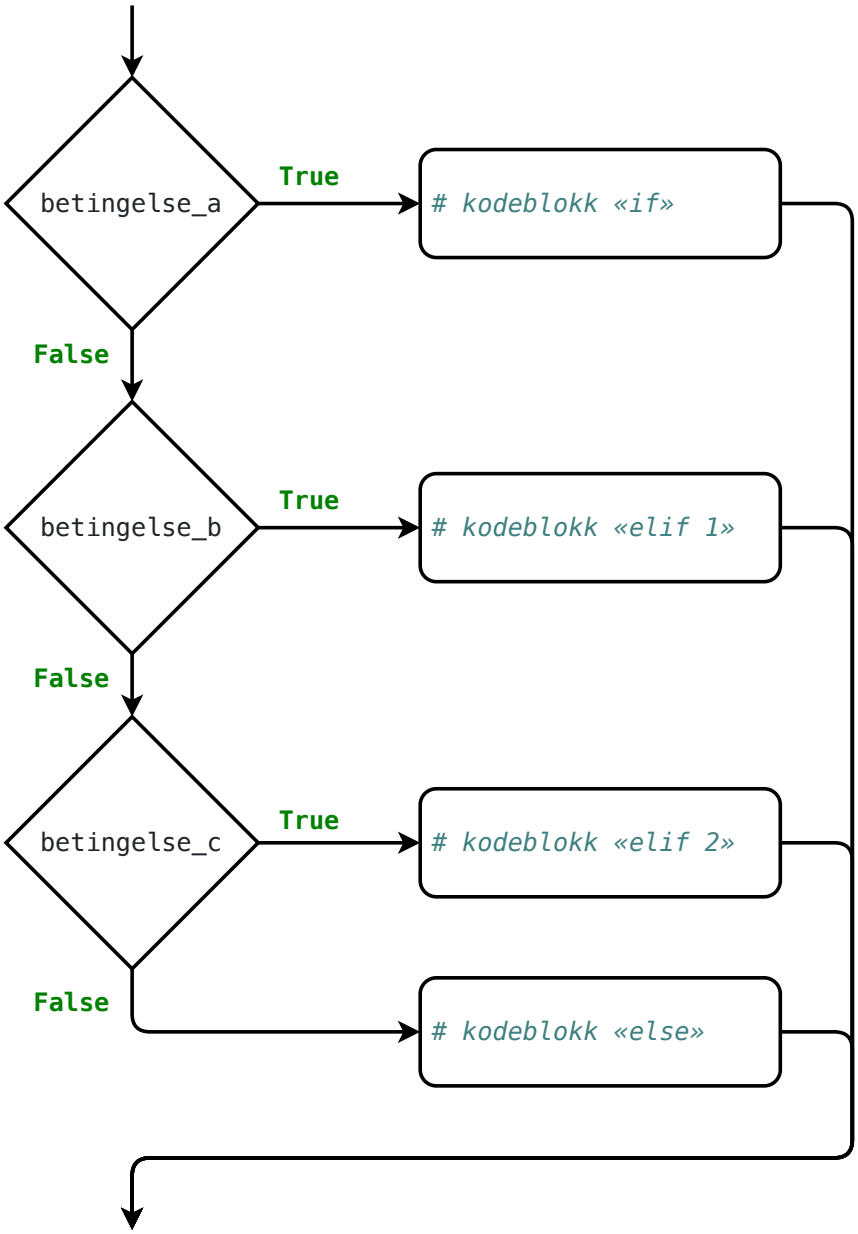
print("NB! Du kan forhåndsstemme på Torgalmenningen frem til 8. september!")
```

If-elif-else

I en sekvens som begynner med en `if` -setning og som fortsetter med et valgfritt antall `elif` -setninger og som eventuelt avsluttes med en `else`, er det nøyaktig én av kodeblokkene som utføres: første gang en betingelse evaluerer til `True`. Hvis ingen av betingelsene evaluerer til `True`, vil kodeblokken etter `else`-ordet utføres.

```
if betingelse_a:
    # kodeblokk «if»
elif betingelse_b:
    # kodeblokk «elif 1»
elif betingelse_c:
    # kodeblokk «elif 2»
else:
    # kodeblokk «else»
```

↑ Innhold



Eksempel:

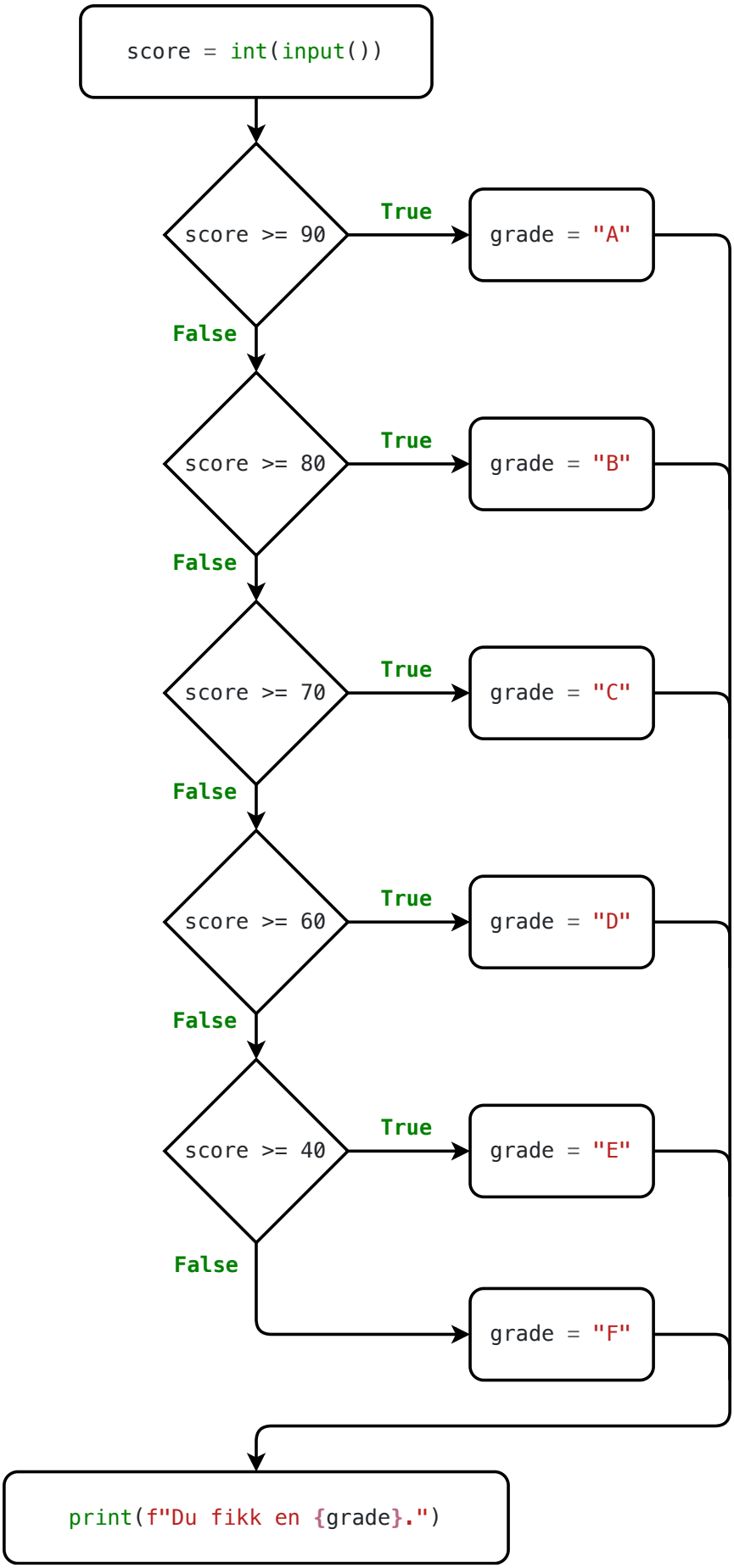
↑ Innhold

```
print("Hvor mange poeng fikk du på prøven?")
score = int(input())

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
elif score >= 40:
    grade = "E"
else:
    grade = "F"

print(f"Du fikk en {grade}.")
```

↑ Innhold


score = int(input())
score >= 90
True
grade = "A"
False
score >= 80
True
grade = "B"
False
score >= 70
True
grade = "C"
False
score >= 60
True
grade = "D"
False
score >= 40
True
grade = "E"
False
grade = "F"
print(f"Du fikk en {grade}.")

↑ Innhold

If-else -uttrykk

```
print("Foo" if True else "Bar") # Foo
print("Foo" if False else "Bar") # Bar
print("Foo" if 1 < 2 else "Bar") # Foo
print()

x = -3
print(0 if x < 0 else x) # 0 (fordi -3 < 0 er True)
```

Se også avsnittet om uttrykk med betingelse i operatorer.

Truthy og falsy verdier

Dersom en betingelse ikke evaluerer til en boolsk verdi, vil den likevel *tolkes* som en enten `True` eller `False`. De fleste verdier tolkes som True, men det er også noen verdier som tolkes som False. Verdier som tolkes som True kalles *truthy*, mens verdier som tolkes som False kalles *falsy*.

```
print("Alt kan tolkes som en boolsk verdi")
print("Sann" if True else "Usann") # Sann
print("Sann" if 1 else "Usann") # Sann
print("Sann" if "Hello" else "Usann") # Sann
print("Sann" if "0" else "Usann") # Sann
print("Sann" if 0 else "Usann") # Usann
print("Sann" if "" else "Usann") # Usann
print("Sann" if None else "Usann") # Usann
```

For å sjekke om en verdi er truthy eller falsy, kan vi benytte funksjonen `bool` som konverterer enhver verdi til en boolsk verdi:

↑ Innhold

```
print('Alle tall bortsett fra 0 er truthy')
print(f'{bool( 2) = }') # True
print(f'{bool( 1) = }') # True
print(f'{bool(-1) = }') # True
print(f'{bool( 0) = }') # False
print()
print(f'{bool( 1.0) = }') # True
print(f'{bool(0.01) = }') # True
print(f'{bool(-0.1) = }') # True
print(f'{bool( 0.0) = }') # False
print(f'{bool(-0.0) = }') # False
print()
print('Alle strenger bortsett fra den tomme strengen er truthy')
print(f'{bool( "True") = }') # True
print(f'{bool("False") = }') # True
print(f'{bool(    " ") = }') # True
print(f'{bool(     "") = }') # False
print()
print('Alle lister bortsett fra den tomme listen er truthy')
print(f'{bool([1, 2, 3]) = }') # True
print(f'{bool([0, 0, 0]) = }') # True
print(f'{bool(  [False]) = }') # True
print(f'{bool(       []) = }') # False
print()
print('Den spesielle verdien None er falsy')
print(f'{bool(None) = }') # False
```

Verdier som er falsy er:

- `False`
- `None`
- `0` `0.0` (tallverdien 0)
- `""` (en tom streng)
- `[]` `()` `{}` `set()` (en tom liste/tuple/dict/mengde)

De fleste andre verdier er truthy. Vi kan alltid dobbeltsjekke med `bool` -funksjonen.

Les mer om hvordan truthy og falsy verdier fungerer sammen med `and` og `or` i kursnotater om boolske uttrykk.

God stil

Merk at dette avsnittet omhandler *stil* og ikke korrekthet.

Negert betingelse:

↑ Innhold

```python
# Dårlig
b = True
if not b:
    print("nei")
else:
    print("ja")
```

```python
# Bra
b = True
if b:
    print("ja")
else:
    print("nei")
```

Tom `if` -setning:

```python
# Dårlig
b = True
if b:
    pass
else:
    print("nei")
```

```python
# Bra
b = True
if not b:
    print("nei")
```

Unødvendig sammenligning med `True` / `False` :

```python
# Dårlig
x = 2
y = 3
if (x < y) == True:
    print("ja")
```

```python
# Bra
x = 2
y = 3
if x < y:
    print("ja")
```

Bruk av `if` i stedet for `and` :

```python
# Mindre foretrukket
b1 = True
b2 = True
if b1:
    if b2:
        print("begge")
```

```python
# Bra
b1 = True
b2 = True
if b1 and b2:
    print("begge")
```

Bruk av ekstra `if` i stedet for `else` :

```python
# Dårlig
b = True
if b:
    print("ja")
if not b:
    print("nei")
```

```python
# Bra
b = True
if b:
    print("ja")
else:
    print("nei")
```

Bruk av ekstra `if` i stedet for `elif` :

↑ Innhold

```python
# Dårlig
x = 10
if x < 5:
    print('small')
if (x >= 5) and (x < 10):
    print('medium')
if (x >= 10) and (x < 15):
    print('large')
if x >= 15:
    print('extra large')
```

```python
# Bra
x = 10
if x < 5:
    print('small')
elif x < 10:
    print('medium')
elif x < 15:
    print('large')
else:
    print('extra large')
```

Fancy bruk av Python sin «aritmetikk»:

```python
# Horribelt
x = 42
y = int((x > 0) and 99)
```

```python
# Bra
x = 42
y = 99 if x > 0 else 0
```

Lurer du på hvorfor koden til venstre i det hele tatt fungerer? Se hva and og or egentlig returnerer og boolske verdier som tall.

PS! Hvis du syntes det er morsomt med uleselig kode, kan du gjerne prøve deg på code golf (https://code.golf/) etterhvert som du føler deg komfortabel med grunnleggende programmering.

Boolske uttrykk

↑ Innhold

- Boolske verdier er også verdier
- Evaluering steg for steg
- Logikk og sannhetstabeller
- Kortslutning
- Truthy og falsy verdier
- Hva and og or egentlig returnerer
- Vanlige feil og misforståelser
- Boolske verdier som tall
- Funksjonene any og all
- God stil
- Eksempler

Boolske verdier er også verdier

Typen `bool` har nøyaktig to verdier: `True` og `False`. Dette er verdier på samme måte som `42` er en verdi av typen `int` og `"hei"` er en verdi av typen `str`. Når vi skriver `True` direkte i koden, er det en *literal*, akkurat som når vi skriver `42` eller `"hei"`.

```
print(type(42))     # <class 'int'>
print(type("hei"))  # <class 'str'>
print(type(True))   # <class 'bool'>
print(type(False))  # <class 'bool'>
```

Merk at `True` og `False` skrives med stor forbokstav og *uten* anførselstegn. `"True"` er en streng, ikke en boolsk verdi, og `true` (med liten t) er ikke noe som finnes, eller eventuelt et veldig forvirrende variabelnavn.

En *sammenligning* som `age >= 18` er et *uttrykk*, på samme måte som `age + 1` er et uttrykk. Forskjellen er bare hvilken type verdi uttrykket evaluerer til: `age + 1` evaluerer til en `int`, mens `age >= 18` evaluerer til en `bool`.

```
age = 20
print(age + 1)    # 21
print(age >= 18)  # True
```

Siden boolske verdier er helt vanlige verdier, kan vi gjøre det samme med dem som med andre verdier. Vi kan for eksempel lagre dem i variabler, skrive dem ut, legge dem i lister, og bruke dem som argument til og returverdi fra funksjoner.

```
age = 20
is_adult = age >= 18      # Variabelen is_adult får verdien True
print(is_adult)           # True

answers = [True, False, age > 100]
print(answers)            # [True, False, False]

if is_adult:              # Betingelsen er variabelen is_adult, som er True
    print("Velkommen inn!")
```

En betingelse i en if-setning trenger altså ikke å være en sammenligning. Det kan være *ethvert uttrykk*, inkludert en variabel, en literal eller et funksjonskall. Python evaluerer uttrykket, og ser deretter på hvilken verdi det ble.

```
def is_even(n):
    return n % 2 == 0    # Returnerer resultatet av uttrykket: True eller False

print(is_even(4))  # True
print(is_even(7))  # False

x = 10
if is_even(x):
    print(x, "er et partall")
```

Evaluering steg for steg

Et boolsk uttrykk evalueres på samme måte som et aritmetisk uttrykk: ett deluttrykk av gangen, og hvert deluttrykk erstattes med verdien det evaluerer til. Når vi regner ut `2 + 3 * 4` regner vi først ut `3 * 4` og erstatter det med `12`, slik at vi får `2 + 12`, og deretter `14`.

Tilsvarende for et boolsk uttrykk:

```
x = 3
(x < 5) and not (x == 3)
# Steg 1: evaluer  x < 5    -> True and not (x == 3)
# Steg 2: evaluer  x == 3   -> True and not True
# Steg 3: evaluer  not True -> True and False
# Steg 4: evaluer  and      -> False
```

Hvilken rekkefølge operatorene utføres i bestemmes av *presedens* (se presedenstabellen). For de logiske operatorene gjelder:

- Aritmetikk (`+`, `*` osv.) utføres før relasjoner (`<`, `==`, `in` osv.)
- Relasjoner utføres før `not`
- `not` utføres før `and`
- `and` utføres før `or`

↑ Innhold

Dette betyr at uttrykket under har «usynlige parenteser» slik som vist i kommentarene:

```
x = 7
print(x > 5 or x < 0 and not x == 7)
# Med usynlige parenteser:
#   (x > 5) or ((x < 0) and (not (x == 7)))
# Steg for steg:
#   True or (False and (not True))
#   True or (False and False)
#   True or False
#   True
```

Legg merke til at `and` binder sterkere enn `or`, på samme måte som `*` binder sterkere enn `+`. Uttrykket `True or True and False` betyr derfor `True or (True and False)`, og ikke `(True or True) and False`.

```
print(True or True and False)    # True
print((True or True) and False)  # False
```

Selv om det ikke alltid er nødvendig, er det god stil å bruke parenteser for å gjøre det tydelig hva man mener.

Kjedede sammenligninger

I Python kan vi kjede sammen flere sammenligninger, slik vi er vant med fra matematikk. Et uttrykk som `0 < x < 10` tolkes som `(0 < x) and (x < 10)`.

```
x = 5
print(0 < x < 10)            # True
print((0 < x) and (x < 10))  # True (det samme)

x = 15
print(0 < x < 10)            # False
```

Logikk og sannhetstabeller

Operatorene `not`, `and` og `or` er i logikken definert ved sine *sannhetstabeller* (se også logiske operatorer). En sannhetstabell viser resultatet for hver mulige kombinasjon av verdier.

↑ Innhold

x	not x
False	True
True	False

x	y	x and y	x or y
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Kort oppsummert:

- `x and y` er `True` bare hvis *begge* er `True`.
- `x or y` er `True` hvis *minst én* er `True`.
- `not x` er det motsatte av `x`.

Sannhetstabeller for sammensatte uttrykk

Vi kan lage sannhetstabeller for større uttrykk ved å regne ut én kolonne for hvert deluttrykk. La oss se på uttrykket `not x or y`. Siden `not` har høyere presedens enn `or`, betyr dette `(not x) or y`.

x	y	not x	(not x) or y
False	False	True	True
False	True	True	True
True	False	False	False
True	True	False	True

De Morgans lover

To uttrykk er *ekvivalente* dersom de har samme sannhetstabell. To nyttige ekvivalenser er *De Morgans lover*:

- `not (x and y)` er det samme som `(not x) or (not y)`
- `not (x or y)` er det samme som `(not x) and (not y)`

Med andre ord: når vi flytter en `not` inn i en parentes, må vi bytte `and` med `or` og omvendt. Dette er nyttig for å gjøre negerte betingelser enklere å lese.

Norsk har et eget ord for den andre loven: «verken … eller». «Jeg drikker *verken* kaffe *eller* te» betyr «det er ikke sant at jeg drikker kaffe eller te», altså `not (coffee or tea)`. Det betyr det samme som «jeg drikker *ikke* kaffe, *og* jeg drikker *ikke* te», altså `(not coffee) and (not tea)`.

↑ Innhold

```
age = 10
# Disse to betingelsene er ekvivalente
if not (age >= 18 and age <= 67):
    print("Utenfor arbeidsfør alder")
if age < 18 or age > 67:
    print("Utenfor arbeidsfør alder")
```

Kortslutning

Når Python evaluerer `x and y`, evalueres `x` først. Hvis `x` er `False`, er svaret uansett `False`, uavhengig av hva `y` er. Python gidder da ikke å evaluere `y` i det hele tatt. Tilsvarende for `x or y`: hvis `x` er `True`, er svaret uansett `True`, og `y` blir ikke evaluert. Dette kalles *kortslutning* (engelsk: *short-circuit evaluation*).

- `False and ...` blir `False` uten at høyresiden evalueres.
- `True or ...` blir `True` uten at høyresiden evalueres.

Vi kan se at høyresiden ikke blir evaluert ved å la høyresiden være noe som ville krasjet programmet, for eksempel en divisjon med null:

```
print(False and 1/0)  # False (1/0 blir aldri evaluert)
print(True or 1/0)    # True (1/0 blir aldri evaluert)
print(True and 1/0)   # Krasjer med ZeroDivisionError
```

Vi kan også se det ved å bruke funksjoner som skriver ut noe når de kalles:

```
def yes():
    print("  yes() ble kalt")
    return True

def no():
    print("  no() ble kalt")
    return False

print("no() and yes():")
print(no() and yes())   # Bare no() blir kalt
print()
print("yes() or no():")
print(yes() or no())    # Bare yes() blir kalt
print()
print("yes() and no():")
print(yes() and no())   # Begge blir kalt
```

Kortslutning som vakt

Kortslutning er nyttig for å beskytte et uttrykk som bare gir mening dersom en annen betingelse er oppfylt. Venstresiden fungerer da som en *vakt* for høyresiden.

↑ Innhold

```
x = 0
y = 5
# Hvis x er 0, blir y / x aldri evaluert, og programmet krasjer ikke
if x != 0 and y / x > 2:
    print("y er mer enn dobbelt så stor som x")
else:
    print("Kan ikke dele på 0, eller y er ikke stor nok")
```

```
names = []
# Hvis listen er tom, blir names[0] aldri evaluert (som ville gitt IndexError)
if len(names) > 0 and names[0] == "Ada":
    print("Ada er først i køen")
else:
    print("Ada er ikke først i køen")
```

Rekkefølgen er viktig! Dersom vi bytter om på venstre- og høyresiden, krasjer programmet:

```
names = []
if names[0] == "Ada" and len(names) > 0:  # Krasjer med IndexError
    print("Ada er først i køen")
```

Truthy og falsy verdier

En betingelse i en if-setning trenger ikke å evaluere til en boolsk verdi. Dersom verdien er av en annen type, vil Python likevel *tolke* den som enten `True` eller `False`. Verdier som tolkes som `True` kalles *truthy*, mens verdier som tolkes som `False` kalles *falsy*.

```
if "Hello":
    print("Strengen 'Hello' er truthy")

if 0:
    print("Dette skrives ikke ut, for 0 er falsy")

if [1, 2, 3]:
    print("En liste med elementer er truthy")

if []:
    print("Dette skrives ikke ut, for en tom liste er falsy")
```

Funksjonen `bool` konverterer en verdi til den boolske verdien den tolkes som:

↑ Innhold

```
print(bool(42))       # True
print(bool(-1))       # True
print(bool(0))        # False
print(bool(0.0))      # False
print()
print(bool("hei"))    # True
print(bool(" "))      # True (et mellomrom er ikke tomt)
print(bool("0"))      # True (en streng med tegnet 0 er ikke tom)
print(bool("False"))  # True (en streng med ordet False er ikke tom)
print(bool(""))       # False
print()
print(bool([1, 2]))   # True
print(bool([0]))      # True (listen har ett element, selv om det er 0)
print(bool([False]))  # True
print(bool([]))       # False
print()
print(bool(None))     # False
```

Verdier som er falsy er:

- `False`
- `None`
- `0` og `0.0` (tallverdien null)
- `""` (den tomme strengen)
- `[]` (den tomme listen), og tomme samlinger av andre typer, for eksempel `{}` og `set()`

Nesten alle andre verdier er truthy. En god tommelfingerregel er at «null» og «tomt» er falsy, mens alt annet er truthy.

Operatoren `not` fungerer også med truthy og falsy verdier, og returnerer alltid en boolsk verdi:

```
print(not 0)       # True
print(not 42)      # False
print(not "")      # True
print(not "hei")   # False
print(not [])      # True
```

En vanlig bruk av truthy og falsy er å sjekke om en liste eller streng er tom:

↑ Innhold

```
names = []
if not names:
    print("Listen er tom")

name = input("Hva heter du? ")
if name:
    print(f"Hei, {name}!")
else:
    print("Du skrev ikke noe navn")
```

Hva and og or egentlig returnerer

Så langt har vi sett på `and` og `or` med boolske verdier. Men hva skjer om vi bruker dem med andre verdier?

```
print(0 or 5)        # 5
print(3 or 5)        # 3
print(0 and 5)       # 0
print(3 and 5)       # 5
print("" or "tom")   # tom
```

Resultatet er verken `True` eller `False`, men en av de opprinnelige verdiene! For å forstå hvorfor, må vi kombinere det vi vet om kortslutning og truthy/falsy verdier. Python evaluerer venstresiden først. Dersom venstresiden alene avgjør svaret, returneres venstresiden. Ellers er det høyresiden som avgjør svaret, og da returneres høyresiden.

- `x and y`: Hvis `x` er falsy, returneres `x`. Ellers returneres `y`.
- `x or y`: Hvis `x` er truthy, returneres `x`. Ellers returneres `y`.

Med andre ord returnerer `and` og `or` alltid *enten venstresiden eller høyresiden*, uendret. Det er ikke nødvendigvis en boolsk verdi. Dersom vi bruker resultatet som en betingelse, vil det tolkes som truthy eller falsy, og da får vi akkurat samme oppførsel som i sannhetstabellen. Derfor merker vi vanligvis ikke dette.

Vi kan uttrykke det samme med if-else -uttrykk:

```
x and y   # gir det samme som:   x if not x else y
x or y    # gir det samme som:   x if x else y
```

La oss se på noen eksempler steg for steg:

↑ Innhold

```
0 and 5
# 0 er falsy, så svaret er avgjort; venstresiden returneres
# -> 0

3 and 5
# 3 er truthy, så høyresiden avgjør svaret; høyresiden returneres
# -> 5

0 or "" or "standard"
# or assosierer venstre til høyre: (0 or "") or "standard"
# Steg 1: evaluer parentesen (0 or "")
#   0 er falsy, så høyresiden i parentesen, "", returneres
#   -> "" or "standard"
# Steg 2: evaluer "" or "standard"
#   "" er falsy, så høyresiden, "standard", returneres
#   -> "standard"
```

```
print(0 and 5)                # 0
print(3 and 5)                # 5
print(0 or "" or "standard")  # standard
print([] or [1, 2])           # [1, 2]
print([1] and [])             # []
print(None or 0)              # 0
print(True and "hei")         # hei
print(False or None)          # None
```

Dette blir noen ganger brukt for å angi en standardverdi:

```
name = input("Hva heter du? ")
name = name or "anonym"  # Hvis name er tom streng, blir name "anonym"
print(f"Hei, {name}!")
```

Selv om dette er lov, er det ofte tydeligere å skrive det som et if-else -uttrykk, slik at leseren ikke trenger å kjenne til disse reglene (se God stil).

Merk at `not` er et unntak: `not` returnerer alltid `True` eller `False`.

Vanlige feil og misforståelser

Sammenligne med flere verdier

En vanlig feil er å skrive en betingelse slik vi ville sagt det i dagligtale: «hvis svaret er ja eller Ja».

```
answer = "nei"
if answer == "ja" or "Ja":   # Feil!
    print("Du svarte ja")    # Skrives ut uansett hva answer er
```

↑ Innhold

Med presedens og parenteser blir betingelsen `(answer == "ja") or "Ja"`. Når `answer` er `"nei"`, blir dette `False or "Ja"`, som evaluerer til `"Ja"`. Det er en streng som ikke er tom, og derfor truthy. Betingelsen er altså truthy uansett hva `answer` er.

```
answer = "nei"
print(answer == "ja" or "Ja")  # Ja
```

Det riktige er å skrive ut begge sammenligningene, eller å bruke `in` med en liste:

```
answer = "nei"
if answer == "ja" or answer == "Ja":
    print("Du svarte ja")

if answer in ["ja", "Ja"]:
    print("Du svarte ja")
```

Sjekke om en verdi er innenfor et intervall

```
if x > 0 and < 10:   # SyntaxError: begge sider av and må være hele uttrykk
    ...
```

Det riktige er `x > 0 and x < 10`, eller kortere `0 < x < 10`.

Strengen “True” er ikke True

`True` og `"True"` er to forskjellige verdier av forskjellige typer. Dette er spesielt lett å blande når man leser input fra brukeren, siden `input()` alltid returnerer en streng.

```
answer = input("Skriv True eller False: ")
print(answer == True)     # False, uansett hva du skrev (en str er aldri lik en
bool)
print(bool(answer))       # True, så lenge du skrev noe (også om du skrev False)
print(answer == "True")   # True bare hvis du skrev nøyaktig True
```

Boolske verdier som tall

I Python er `bool` egentlig en spesialisert variant av `int`: `True` oppfører seg som tallet `1`, og `False` oppfører seg som tallet `0`.

```
print(True + True)        # 2
print(True * 10)          # 10
print(False + 5)          # 5
print(True == 1)          # True
print(False == 0)         # True
```

Dette kan brukes til å telle hvor mange verdier i en liste som er `True`:

↑ Innhold

```
answers = [True, False, True, True, False]
print(sum(answers))  # 3
```

Vær likevel forsiktig med å bruke dette for å være smart (se God stil).

Funksjonene any og all

Funksjonene `any` og `all` tar en liste som argument, og fungerer som `or` og `and` over alle elementene i listen:

- `any(a)` returnerer `True` dersom *minst ett* element i `a` er truthy.
- `all(a)` returnerer `True` dersom *alle* elementene i `a` er truthy.

I motsetning til `and` og `or` returnerer `any` og `all` alltid `True` eller `False`.

```
print(any([False, False, True]))  # True
print(any([False, False, False])) # False
print(all([True, True, True]))    # True
print(all([True, False, True]))   # False
print()
print(any([0, "", "hei"]))        # True ("hei" er truthy)
print(all([1, "a", [0]]))         # True (alle er truthy)
```

```
scores = [7, 9, 4, 10]
passed = []
for score in scores:
    passed.append(score >= 5)
print(passed)        # [True, True, False, True]
print(all(passed))   # False (ikke alle bestod)
print(any(passed))   # True (minst én bestod)
print(sum(passed))   # 3 (antall som bestod)
```

En litt overraskende konsekvens av definisjonen er hva som skjer med en tom liste. `any([])` er `False`, siden det ikke finnes noe element som er truthy. `all([])` er `True`, siden det ikke finnes noe element som *ikke* er truthy.

```
print(any([]))  # False
print(all([]))  # True
```

God stil

Merk at dette avsnittet omhandler *stil* og ikke korrekthet. Koden til venstre fungerer, men koden til høyre er lettere å lese.

↑ Innhold

Unødvendig sammenligning med `True` / `False` : Siden en sammenligning allerede evaluerer til `True` eller `False` , er det unødvendig å sammenligne resultatet med `True` en gang til.

```python
# Dårlig
if is_adult == True:
    print("ja")
if is_adult == False:
    print("nei")
```

```python
# Bra
if is_adult:
    print("ja")
if not is_adult:
    print("nei")
```

Returnere `True` / `False` med en if-setning: Når en funksjon skal returnere om noe er sant eller ikke, er det fristende å bruke en if-setning. Men uttrykket i betingelsen *er* allerede den verdien vi ønsker å returnere.

```python
# Dårlig
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

```python
# Bra
def is_even(n):
    return n % 2 == 0
```

Tilordne `True` / `False` med en if-setning: Det samme gjelder for variabler.

```python
# Dårlig
if age >= 18:
    is_adult = True
else:
    is_adult = False
```

```python
# Bra
is_adult = age >= 18
```

Negert sammenligning: Det finnes en sammenligningsoperator for det motsatte av hver sammenligning, så vi trenger ikke `not` .

```python
# Dårlig
if not x == y:
    print("ulike")
if not x < 5:
    print("stor")
```

```python
# Bra
if x != y:
    print("ulike")
if x >= 5:
    print("stor")
```

Intervall: En kjedet sammenligning ligner på matematisk notasjon og er lettere å lese.

```python
# Mindre foretrukket
if x > 0 and x < 10:
    print("ett siffer")
```

```python
# Bra
if 0 < x < 10:
    print("ett siffer")
```

Bruk av `or` for å velge en verdi: Dette er lov, men leseren må kjenne til hva `or` egentlig returnerer for å forstå koden.

↑ Innhold

```python
# Mindre foretrukket
name = name or "anonym"
```

```python
# Bra
name = name if name != "" else "anonym"
```

Fancy bruk av boolske verdier som tall:

```python
# Horribelt
y = (x > 0) * 99
```

```python
# Bra
y = 99 if x > 0 else 0
```

Eksempler

Prøv å forutsi hva hvert uttrykk evaluerer til før du kjører koden og ser fasiten. Bruk reglene for presedens, kortslutning og truthy/falsy verdier.

```python
print(True and not False)
print(not True or True)
print(not (True or True))
print(3 > 2 > 1)
print(1 < 3 == 3)
print()
print(0 or 7)
print(7 and 0)
print("" or "tom")
print([] and [1, 2])
print(not "")
print(None or 0)
print()
print(1 == 1 or 2)
print(1 == 2 or 2)
print("a" == "b" or "c")
print(True + True + False)
print(all([1, "a", [0]]))
print(any([0, "", []]))
```

Funksjoner

↑ Innhold

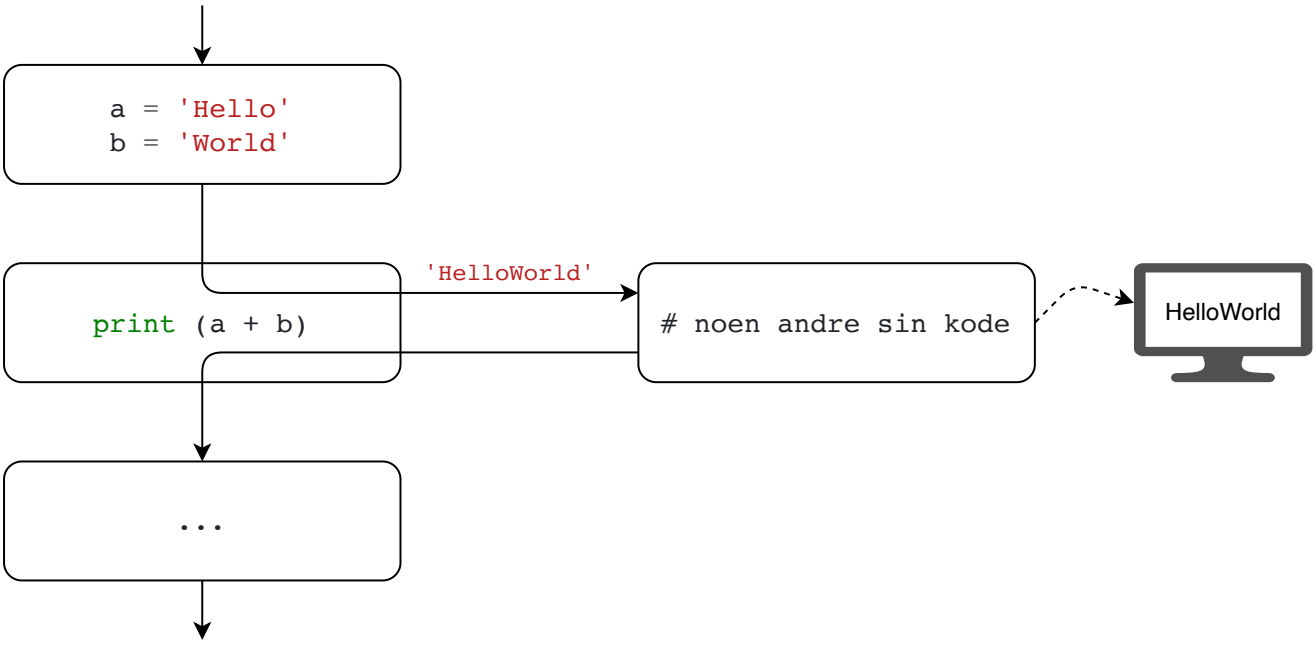
- Funksjonskall
- Funksjonskall med returverdi
- Funksjonskall med både returverdi og sideeffekt
- Hvorfor funksjoner?
- Vår første funksjon
- Definere egne funksjoner
- Retursetninger
- Vanlig feil: forveksle returverdi og sideeffekt
- Mer enn én returverdi
- Ordbok
- Skop

Funksjonskall

Å *kalle* en funksjon betyr at vi instruerer funksjonenen til å kjøres. For eksempel gjør vi et kall til `print` -funksjonen i denne kodesnutten:

```
a = 'Hello'
b = 'World'
print(a + b)
...
```

Når vi kaller en funksjon, gir vi ofte funksjonen noe informasjon som den trenger for å gjøre jobben sin. Denne informasjonen kalles et **argument**. I eksempelet over blir verdien `'HelloWorld'` gitt som argument til `print` -funksjonen (det er denne verdien uttrykket `a + b` evaluerer til).



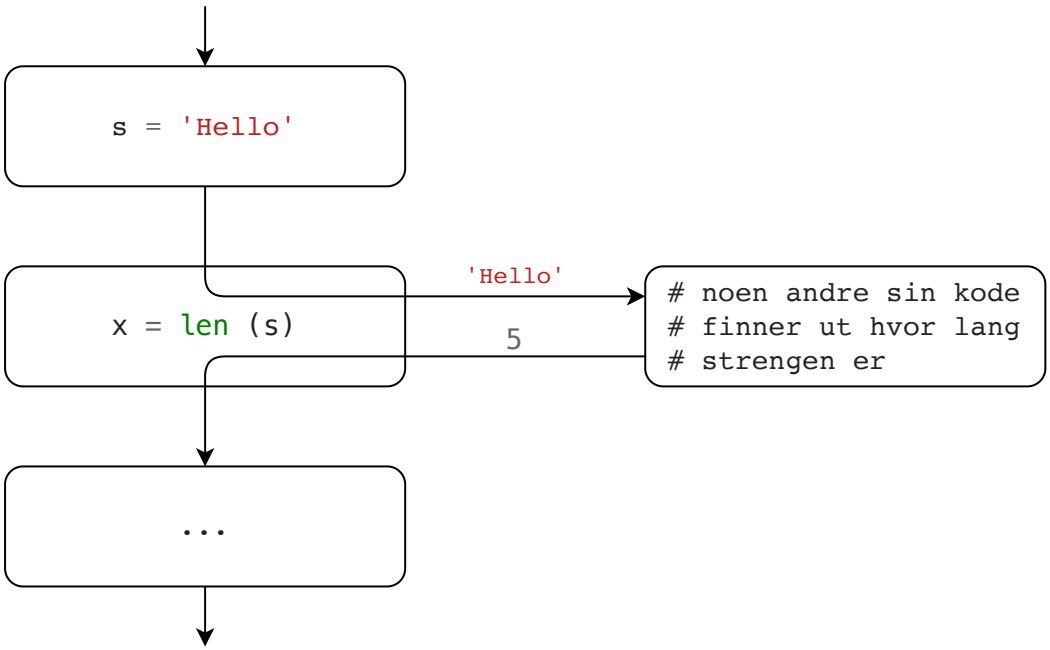
↑ Innhold

Funksjonskall med returverdi

Et annet eksempel på et funksjonskall, er kallet til `len` -funksjonen i kodesnutten under. Dette er en funksjon med en **returverdi**.

```
s = 'Hello'
x = len(s)
...
```

Her blir verdien `'Hello'` gitt som argument til `len` -funksjonen. Funksjonen *returnerer* så verdien `5` , før variabelen x endres til å peke på denne verdien.



📖 Mer om returverdier

↑ Innhold

Alle funksjonskall har en returverdi. For eksempel returnerer `input`-funksjonen den strengen brukeren skriver inn. Vi kan lagre denne strengen i en variabel, og bruke den senere i programmet.

Eksempler på innebygde funksjoner i Python og deres returverdi.

```
x = max(1, 2, 3)   # x = 3
y = min(1, 2, 3)   # y = 1
z = abs(-6)        # z = 6
print(x, y, z)

s = input('Skriv noe: ') # s = 'Hallo'    (hvis brukeren skriver Hallo)
l = len(s)               # l = 5          (hvis brukeren skrev Hallo)
print(s, l)
```

I disse eksemplene er det returverdien fra funksjonskallene som lagres med variablene `x`, `y`, `z`, `s` og `l`.

Print-funksjonen returnerer den spesielle verdien `None`. Det er helt meningsløst å ta vare på den, men det er teknisk sett mulig:

```
x = print('Hello') # Skriver ut Hello. Returverdien fra print lagres i x
print(x)           # None
```

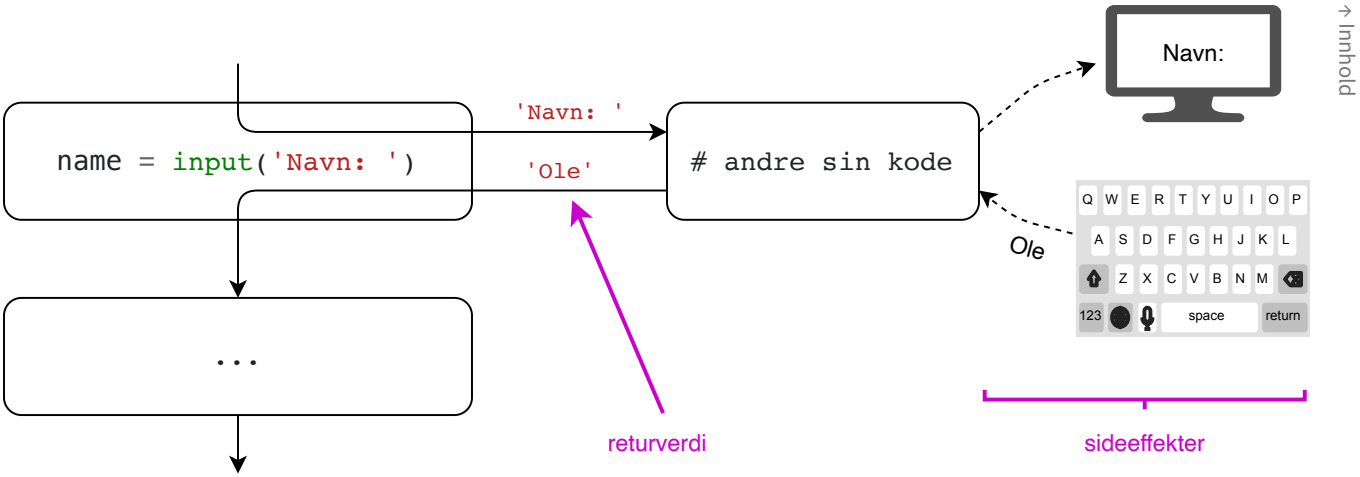
*Alle funksjoner har **teknisk sett** en returverdi; men for noen funksjoner er returverdien alltid den spesielle verdien `None`. Hvis vi litt flåsete hevder at en funksjon ikke har en returverdi, mener vi altså egentlig at returverdien alltid er `None`.*

Funksjonskall med både returverdi og sideeffekt

Et tredje eksempel på et funksjonskall vi har sett før, er input-funksjonen. Denne funksjonen har *både* sideeffekt og en returverdi. Med **sideeffekt** mener vi at noe skjer i «verden forøvrig» som følge av funksjonskallet, som ikke er en returverdi. I dette tilfellet skjer det noe på skjermen: teksten «Navn: » vises.

```
name = input('Navn: ')
...
```

↑ Innhold



Her blir strengen `'Navn: '` gitt som argument til `input`-funksjonen. Som en sideeffekt av input-funksjonen vises denne strengen i terminalen. Brukeren skriver inn f. eks. «Ole». Funksjonen *returnerer* så verdien `'Ole'` før variabelen `name` endres til å peke på denne verdien.

📖 Mer om sideeffekter

Et funksjonskall kan ha to typer effekter:

- Returverdi
- Sideeffekter

En **returverdi** er den verdi som et funksjonskall *evalueres* til. For eksempel evalueres funksjonskallet `max(1, 2, 3)` til `3`. Returverdien kan lagres i en variabel, eller brukes som en del av et større uttrykk. Eksempler på funksjoner med en meningsfylt returverdi er `max`, `min`, `input` og `len`.

En **sideeffekt** er en endring i «verdens tilstand forøvrig» som skjer på grunn av et funksjonskall. For eksempel har `print`-funksjonen en sideeffekt i at det dukker opp tekst på skjermen. Alle tegne-funksjonene som er dokumentert i kursnotatene om grafikk er også eksempler på funksjoner med sideeffekter (de tegner figurerer som blir vist på skjermen).

Det er som regel sideeffekter sluttbrukeren til syvende og sist ønsker seg av et program: noe vises på skjermen, innholdet i en fil endrer seg eller lignende. Samtidig har funksjoner *uten* sideeffekter store fordeler: de er lettere å teste og feilsøke, og det er lettere å modularisere et program med dem.

> *En av de aller vanligste kildene til forvirring for ferske programmerere er forskjellen på returverdi og sideeffekt; nærmere bestemt, forskjellen mellom å returnere en verdi og å skrive ut en verdi til skjermen.*
>
> *Å skrive noe ut på skjermen er en sideeffekt, det innebærer **ikke** å returnere noe.*

Hvorfor funksjoner?

Noen har gjort et kall til `print`-funksjonen: under panseret i datamaskinen skjer det noe greier, og så «vipps!» dukker det opp tekst i terminalen. Heldigvis trenger ikke *vi* å tenke på noe av det. `print`-

funksjonen bare fungerer. Vi kan gjenbruke noen andre sin genialitet uten at det koster oss en kalori. Dette bringer oss til de viktigste hensiktene med funksjoner: å gjenbruke kode og å abstrahere bort detaljer.

↑ Innhold

📖 **Hensikten med funksjoner**

- **Gjenbruk av kode.** Om vi har en samling instruksjoner vi ønsker å kjøre flere ganger, kan vi legge disse instruksjonene i en funksjon, og deretter kalle funksjonen hver gang vi ønsker å kjøre instruksjonene. Da slipper vi å skrive den samme koden flere ganger. Dette gjør det enklere å endre på koden eller rette feil senere.
- **Selvdokumenterende kode.** Funksjoner har *navn* som ideelt sett beskriver hva funksjonen gjør. Dette gjør det enklere å lese koden.
- **Abstraksjon.** Funksjoner lar oss abstrahere bort detaljer som ikke er umiddelbart relevante for det vi holder på med. Ved å dele inn koden i et hierarki av mer og mer spesialiserte funksjoner, kan vi fokusere på det som er viktig for oss på det abstraksjonsnivået vi befinner oss på.

📖 **Abstraksjon**

Abstraksjon er å se bort fra detaljer som ikke er umiddelbart relevante for det vi holder på med.

I en tidlig fase av sommerferie-planleggingen sier vi gjerne til hverandre «først besøker vi bestemor, så drar vi til Sverige, og så drar vi en uke på hytten.» Vi trenger ikke å vite hvordan vi kommer oss til bestemor, eller hvordan vi kommer oss til Sverige – vi bare antar at det finnes en løsning på dette. Vi abstraherer altså bort detaljene, og fokuserer på det som er viktig for oss på dette «abstraksjonsnivået».

Når vi senere detaljplanlegger reisen til bestemor, sier vi til hverandre «først tar vi bussen til byen, så spiser vi lunsj på stasjonen, så tar vi toget videre til Hønefoss, så henter bestemor oss der». Vi er nå på et litt lavere abstraksjonsnivå enn før, men vi bryr oss fremdeles ikke om hvordan bussen eller toget fungerer – vi antar bare at bussen gjør som vi forventer, uten at vi trenger å ofre en tanke på trafikkregler eller bussmotorer. Vi abstraherer altså fremdeles bort detaljene, og fokuserer på det som er viktig for oss.

Når vi programmerer, er det lurt å skille fra hverandre instruksjoner som befinner seg på ulike abstraksjonsnivåer. Dette gjør vi blant annet ved å dele kode inn i **funksjoner**.

Vår første funksjon

En funksjon er en sekvens med kommandoer man kan referere til ved hjelp av et funksjonsnavn. Man kan utføre funksjonen flere ganger ved å kalle den flere ganger.

↑ Innhold

```
# Vi definerer en funksjon som heter `my_sample_function`
def my_sample_function():
    print('A')
    print('B')
    print('C')

# Vi kaller my_sample_function to ganger
my_sample_function()
my_sample_function()
```

Funksjonskroppen (setningene som skal utføres når funksjonen kjører) må ha riktig *innrykk*. God stil tilsier at innrykket består av 4 mellomrom.

```
def hello():
    print('Skriv ditt navn:')
    name = input()
   print(f'Hei {name}') # Krasjer, mangler et mellomrom

hello()
```

Definere egne funksjoner

For å definere vår egen funksjon:

- begynn med det spesielle ordet `def` fulgt av et mellomrom; så
- et valgfritt navn vi ønsker å gi funksjonen; så
- en opplisting av parametre omsluttet av paranteser; så
- et kolon; så
- selve funksjonskroppen med innrykk og eventuelt retursetninger

Funksjonskroppen er instruksjonene som skal utføres når funksjonen kalles, og må ha et innrykk i forhold til `def`-ordet. Standard innrykk er 4 mellomrom.

```
def sum_of_squares(a, b):
    square_of_a = a * a
    square_of_b = b * b
    return square_of_a + square_of_b

x = sum_of_squares(1, 1)
print(x) # 2

y = sum_of_squares(1, x)
print(y) # 5
```

En funksjon må være definert *før* den kalles.

```
sum_of_squares(1, 2) # Krasjer, funksjonen sum_of_squares er ikke definert enda

def sum_of_squares(a, b):
    square_of_a = a * a
    square_of_b = b * b
    return square_of_a + square_of_b
```

En setning som befinner seg inne i en funksjonskropp kan derimot kalle andre funksjoner som defineres senere i koden; så lenge den andre funksjonen er definert *når den kalles* er det tilstrekkelig.

```
def besok_bestemor():
    reis_til_bestemor()
    print('Spis kake')
    reis_hjem_fra_bestemor()

def reis_til_bestemor():
    print('Ta bussen til byen')
    print('Ta toget til Hønefoss')
    print('Bli hentet av bestemor på stasjonen')

def reis_hjem_fra_bestemor():
    print('Bli kjørt til Hønefoss av bestemor')
    print('Ta toget til byen')
    print('Ta bussen hjem')

besok_bestemor()
```

*Hva skjer hvis du gjør kallet til `besok_bestemor()` like **før** du skriver definisjonen av `reis_hjem_fra bestemor` i stedet for etterpå? Krasjer det? Hvis ja; når og hvorfor krasjer det?*

Retursetninger

Dersom en funksjon skal returnere en verdi (som ikke er `None`), må den ha en *retursetning*. Returverdien er den verdien uttrykket i retur-setningen evaluerer til.

```
def square(x):
    return x * x

x2 = square(3)
print(x2) # 9
```

Dersom en funksjon ikke har noen retursetning, eller retur-setningen ikke inneholder et uttrykk, returnerer funksjonen den spesielle verdien `None`.

↑ Innhold

```python
def a():
    print('Denne funksjonen returnerer None')
    return None

return_value_a = a()
print(return_value_a) # None

def b():
    print('Denne funksjonen har en tom return-setning')
    return

return_value_b = b()
print(return_value_b) # også None

def c():
    print('Denne funksjonenen har ikke return-setning')

return_value_c = c()
print(return_value_c) # også None
```

Det er ikke nødvendig at en retursetning er den siste setningen i en funksjon; men når en retursetning utføres, avsluttes funksjonen umiddelbart uten å fortsette videre i funksjonskroppen. Hvis man har kode etter en retursetning, kalles dette gjerne *død kode* (og det er selvfølgelig svært dårlig stil).

```python
def hello():
    print('Hello')
    return
    print('Goodbye') # død kode; utføres aldri

hello() # Hello
```

Det kan ofte være nyttig å ha en tidlig retur-setninger inne i en `if`-setning, for å avslutte funksjonen tidlig under gitte betingelser.

```python
def go_to_club(name, age):
    if age < 18:
        return f"Yo {name}, you're too young!"
    result = f"Hi there, {name}! It's time to party"
    result += "party"
    result += "party"
    result += "party"
    result += "party"
    return result + "!"

print(go_to_club("Ole", 14)) # Yo Ole, you're too young!
print(go_to_club("Kari", 18)) # Hi there, Kari! It's time to partypartypa...
```

↑ Innhold

Vanlig feil: forveksle returverdi og sideeffekt

En av de vanligste feilene ferske programmerere gjør, er å forveksle returverdi og sideeffekt. Dette er en feil som er lett å gjøre, fordi det er lett å tenke at en funksjon som skriver ut noe på skjermen, *returnerer* det den skriver ut. Dette er **ikke** tilfelle.

```
def cubed(x):
    print(x**3)   # Funksjon uten retur-verdi, kun side-effekt

cubed(2)          # ser ut til å virke
print(cubed(3))   # rart (skriver også ut `None`)
print(2*cubed(4)) # Krasj!
```

Gjør det heller slik:

```
def cubed(x):
    return x**3   # Funksjonen har retur-verdi, men ingen side-effekt

cubed(2)          # ser ikke ut til å virke (hvorfor?)
print(cubed(3))   # funker!
print(2*cubed(4)) # funker!
```

Mer enn én returverdi

Det er *teknisk sett* kun mulig å ha én returverdi i en funksjon. Men: det er mulig at returverdien er en «tuple» (en slags liste). Da kan vi for alle praktiske formål tenke på det som om funksjonen returnerer flere verdier.

```
def plus_minus(x, epsilon):
    x_lo = x - epsilon
    x_hi = x + epsilon
    return x_lo, x_hi # komma for å skille verdier lager en «tuple»

# Man får egentlig kun én returverdi, men den kan pakkes opp til flere
window = plus_minus(1000, 5)
a = window[0]
b = window[1]
print(a, b) # 995 1005

# Python har en snarvei for å pakke opp en tuple med én gang
a, b = plus_minus(50, 2)
print(a, b) # 48 52
```

Man kan ha så mange elementer man vil i en tuple (men det må matche antall variabler når vi pakker opp).

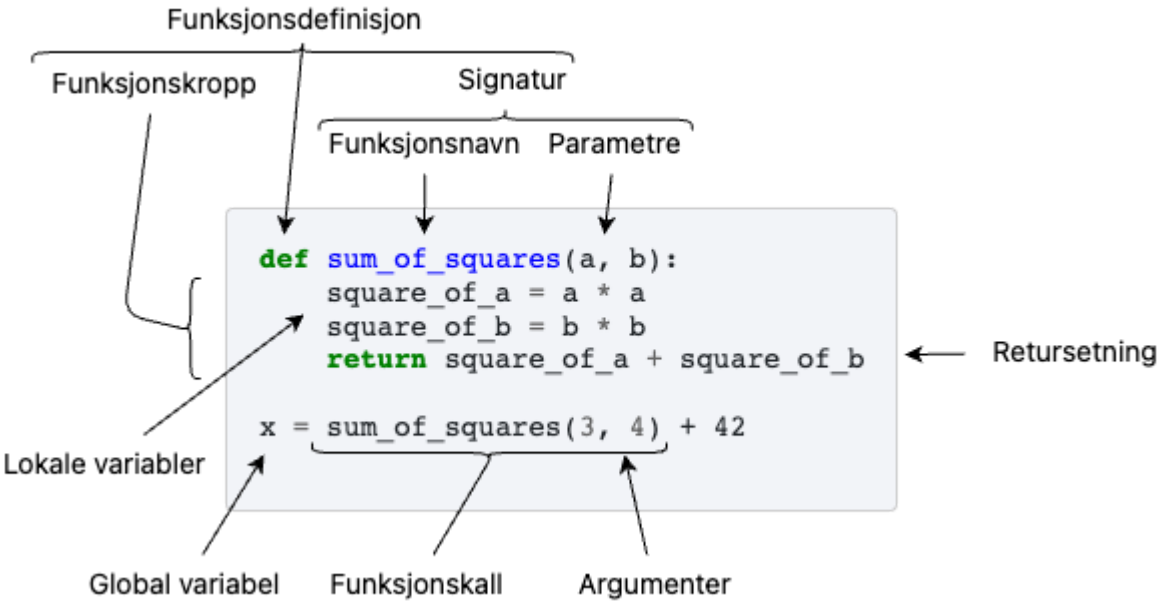
↑ Innhold

```python
def three_next_numbers(n):
    return n + 1, n + 2, n + 3

a, b, c = three_next_numbers(10)
print(a, b, c) # 11 12 13

a, b = three_next_numbers(10) # Krasjer: returnert tuple har 3 verdier, ikke bare 2
print(a, b)
```

Ordbok



Kjørbar og kopierbar versjon av koden i illustrasjonen

```python
def sum_of_squares(a, b):
    square_of_a = a * a
    square_of_b = b * b
    return square_of_a + square_of_b

x = sum_of_squares(3, 4) + 42


print(x)
```

Funksjonsdefinisjon. En funksjonsdefinisjon forteller Python at vi ønsker å definere en ny funksjon. En funksjonsdefinisjon består av følgende komponenter:

- det spesielle ordet `def`,
- et valgfritt navn vi ønsker å gi funksjonen (funksjonsnavn),

↑ Innhold

- en opplisting av valgfritt antall parametre omsluttet av paranteser (parentesene må være med selv om det ikke er noen parametre),
- et kolon (`:`), og
- en funksjonskropp med innrykk. Dette er en sekvens med setninger som skal utføres når funksjonen kalles.

I eksempelet vist over utgjør linje 1-4 en funksjonsdefinisjon.

> *At en funksjon defineres, betyr **ikke** at funksjonskroppen kjøres. Det er bare når funksjonenen kalles at funksjonskroppen kjøres.*

Funksjonskall. Et funksjonskall er en instruksjon om å kjøre en funksjon. Et funksjonskall består av følgende komponenter:

- funksjonsnavnet, og
- en opplisting av argumenter omsluttet av paranteser. Parentesene *må* være med selv om det ikke er noen argumenter.

I eksempelet vis over utgjør `sum_of_squares(3, 4)` på linje 6 et funksjonskall.

> *Alle funksjonskall er uttrykk (men ikke alle uttrykk er funksjonskall). Et funksjonskall evalueres til returverdien fra funksjonen som kalles. I eksempelet over evalueres funksjonskallet `sum_of_squares(3, 4)` på linje 6 til verdien `25`.*

Parameter. En parameter er en lokal variabel. Parametre får angitt hvilke verdier de refererer til når funksjonen kalles, og slutter å eksistere når funksjonskallet er ferdig/returnerer. I eksempelet over er `a` og `b` parametre til funksjonen `sum_of_squares`, og verdiene de blir angitt å referere til når funksjonen kalles er `3` og `4`.

Argument. Et argument er en verdi som gis en funksjon når den kalles. I eksempelet over er `3` og `4` argumenter ved funksjonskallet til `sum_of_squares` på linje 6. Når en funksjon kalles, vil parametrene (i utgangspunktet) referere til argumentene.

📖 **Argumenter vs parametre**

↑ Innhold

Det er fort gjort å blande sammen begrepene *argument* og *parameter*, da de på en måte beskriver to sider av samme sak; men tenk på det slik:

- Et argument er en en *verdi*, f. eks. `42` eller `"Ole"`.
- En parameter er en *variabel*, altså en navngitt referanse, f. eks. `age` eller `name`.

Et argument må eksistere *før* en funksjon kalles, mens en parameter er en variabel som opprettes *når* funksjonen kalles. Så vil parameteren referere til argumentet.

- En parameter kan potensielt endre hvilket objekt den referer til underveis i funksjonskallet; men hvis parameteren endres slik, så slutter den altså å referere til argumentet (det er da ikke argumentet som endrer seg, kun parameteren).

Lokal variabel. En lokal variabel er en variabel som er definert inne i en funksjonskropp. Lokale variabler eksisterer kun underveis i et funksjonskall, og slutter å eksistere når kallet er ferdig/returnerer. I eksempelet over er `square_of_a` og `square_of_b` i tillegg til parametrene `a` og `b` lokale variabler i funksjonen `sum_of_squares`.

Global variabel. En global variabel er en variabel som er definert *utenfor* en funksjonskropp. Slike variabler slettes ikke fra funksjonskall til funksjonskall, men eksisterer så lenge programmet kjører. I eksempelet over er `x` en global variabel.

Signatur. En funksjonssignatur består minimum av funksjonsnavnet og en opplisting av parametrene til funksjonen. Funksjonssignaturen i eksempelet over er altså «sum_of_squares(a, b)». En signatur er den informasjonen som er nødvendig å kjenne til for å kunne kalle funksjonen. I tillegg til funksjonsnavn og parametre, kan også informasjon om returverdien (f. eks. hvilke type den har) og docstring-kommentarer inngå i signaturen.

Skop

En variabel eksisterer i ett *skop* basert på hvor variabelen ble definert. Hver funksjon har sitt eget skop; variabler som er definert i dette skopet kan ikke nås utenfra.

```
def foo(x):
    print(x)

foo(2) # skriver ut 2
print(x) # Krasjer, siden variabelen x kun var definert i foo sitt skop
```

```
def bar():
    y = 42
    print(y)

bar() # skriver ut 42
print(y) # Krasjer, siden variabelen y kun var definert i bar sitt skop
```

↑ Innhold

Det samme variabelnavnet kan eksistere i ulike skop. Men selv om variablene heter det samme, er de helt uavhengig av hverandre.

```
def f(x):
    print("Vi er i f, x =", x)
    x += 5
    return x

def g(x):
    y = f(x*2)
    print("Vi er i g, x =", x)
    z = f(x*3)
    print("Vi er i g, x =", x)
    return y + z

print(g(2))
```

Det kan eksistere flere skop samtidig når koden kjører.

- Det *globale* skopet opprettes når Python begynner å kjøre programmet, og fjernes ikke før Python avslutter.
  - Alle variabler som blir definert utenfor en funksjon, befinner seg i det globale skopet.
- Hver gang du kaller en funksjon, opprettes et nytt skop som tilhører dette funksjonskallet. Dette kalles et *lokalt* skop. Det slettes fullstendig når funksjonen returnerer/er ferdig.
  - Alle parameterne er variabler som hører til det lokale skopet
  - Alle variabler som opprettes i funksjonen tilhører det lokale skopet
- Siden du kan kalle én funksjon fra en annen, kan det være mange slike skop «oppå hverandre.»
- Det er teknisk sett alltid mulig å se variabler fra det globale skopet, men det er ikke en anbefalt praksis. Dersom vi har en lokal variabel med samme navn, vil den maskere den globale variabelen:

```
x = "x i globalt skop"
y = "y i globalt skop"

def f():
    y = "y i lokalt skop"
    z = "z i lokalt skop"
    print(x)
    print(y)
    print(z)

f()
```

Hold tungen rett i munnen og regn ut hva svaret blir før du kjører koden under. Ta notater på papir for å holde styr på hva som foregår.

↑ Innhold

```
def f(x):
    print("Vi er i f, x =", x)
    x += 7
    return round(x / 3)

def g(x):
    x *= 10
    return 2 * f(x)

def h(x):
    x += 3
    return f(x+4) + g(x)

print(h(f(1)))
```

Minne og kodesporing

↑ Innhold

- Minne
- En variabel endrer seg
- En funksjon blir kalt
- Debuggeren
- Manuell kodesporing med tabell
- Manuell kodesporing av funksjoner med tabell

Å lese kode og forstå hva som vil skje om den kjøres er en helt essensiell ferdighet for en programmerer. I dette notatet skal vi forsøke å bygge en intuisjon for hva som egentlig skjer i datamaskinen når et program kjører. Å manuelt forutsi hvordan «minnet» vil endre seg steg for steg, kaller vi *kodesporing*.

Minne

Kildekoden til et program består av en sekvens av *setninger* (statements) som skal utføres én etter én i rekkefølge. Dataprogrammet som leser kildekoden og utfører setningene kalles en *fortolker*.

En fortolker innholder i hovedsak tre bevegelige deler som vil endre tilstand for hvert steg. Dette kaller vi for *minnet* til programmet:

- En samling av *verdier* (også kalt *objekter* eller *data*).
- En samling av *variabler*. Hver variabel har et navn og refererer til en verdi.
- En referanse til neste steg som skal utføres.

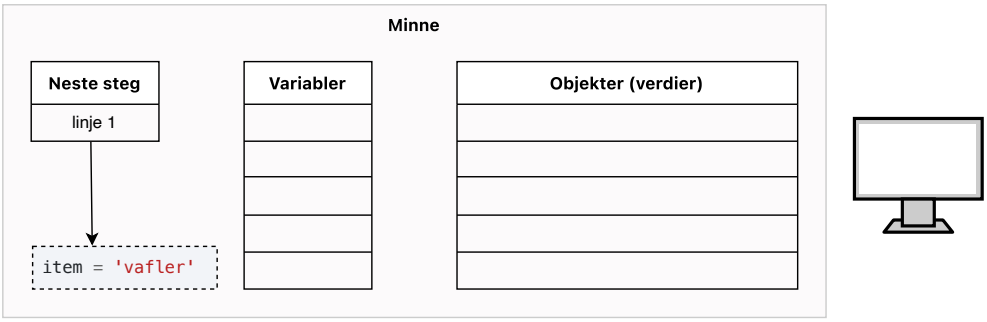
Dersom vi vet tilstanden til de tre komponentene over, er det en mekanisk prosess å beregne hvordan tilstanden vil endre seg når neste setning utføres.

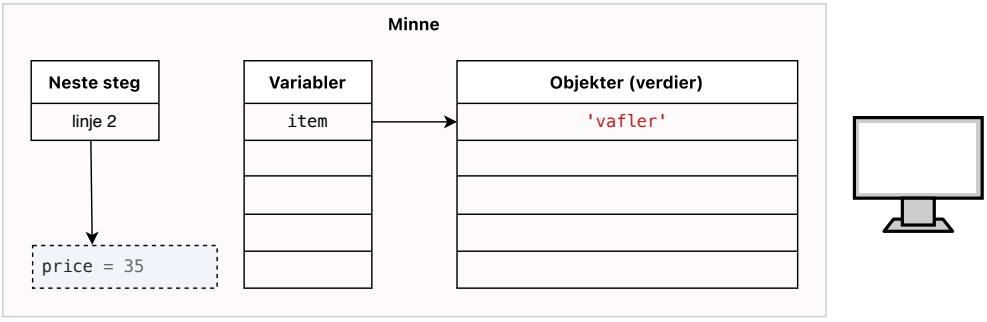
La oss se på et eksempel:

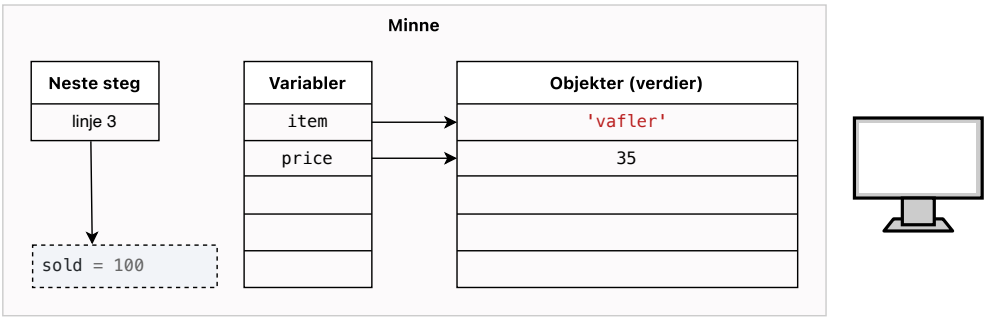
```
1  item = 'vafler'
2  price = 35
3  sold = 100
4  total = price * sold
5  result = f'Vi solgte {sold} {item} for til sammen {total} kroner'
6  print(result)
```

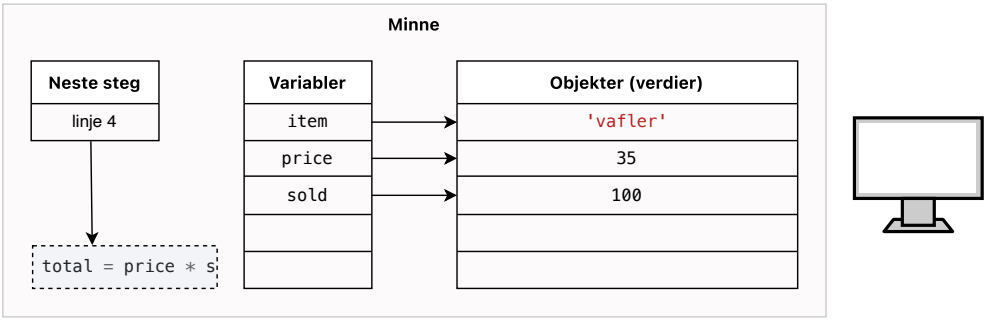
Programmet over består av 6 setninger, som utføres linje for linje. Vi kan følge med på hvordan tilstanden til minnet endrer seg for hver setning som utføres; klikk på neste-knappen under for å se hvordan minnet endrer seg for hvert steg.

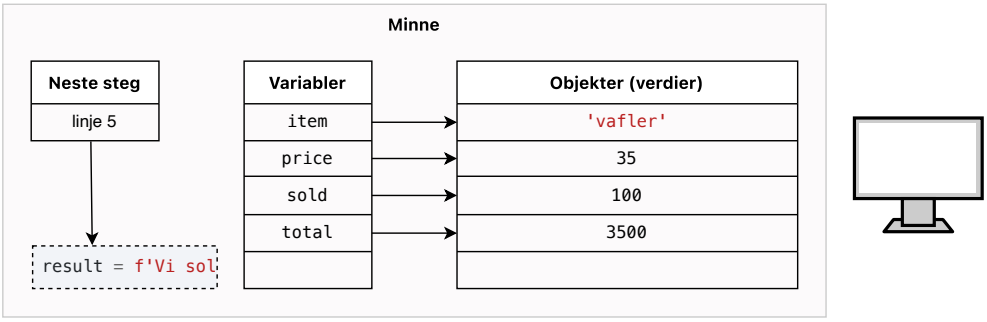
↑ Innhold

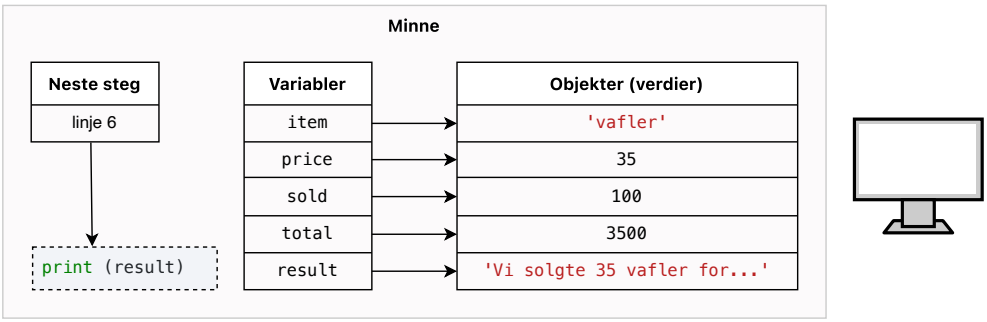

Minne
Neste steg
linje 1
item = 'vafler'
Variabler
Objekter (verdier)


Minne
Neste steg
linje 2
price = 35
Variabler
item
Objekter (verdier)
'vafler'

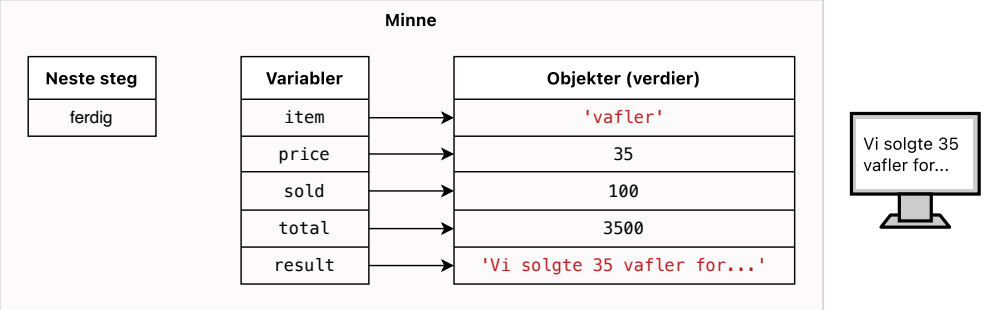

Minne
Neste steg
linje 3
sold = 100
Variabler
item
price
Objekter (verdier)
'vafler'
35


Minne
Neste steg
linje 4
total = price * s
Variabler
item
price
sold
Objekter (verdier)
'vafler'
35
100


Minne
Neste steg
linje 5
result = f'Vi sol
Variabler
item
price
sold
total
Objekter (verdier)
'vafler'
35
100
3500


Minne
Neste steg
linje 6
print (result)
Variabler
item
price
sold
total
result
Objekter (verdier)
'vafler'
35
100
3500
'Vi solgte 35 vafler for...'

↑ Innhold



Vi kan se en tilsvarende sekvens for alle kodeeksempler på denne nettsiden ved å klikke på «se steg» - knappene under kodeeksemplene. Der vises «neste steg» som en rød pil. Prøv det gjerne med en gang!

En variabel endrer seg

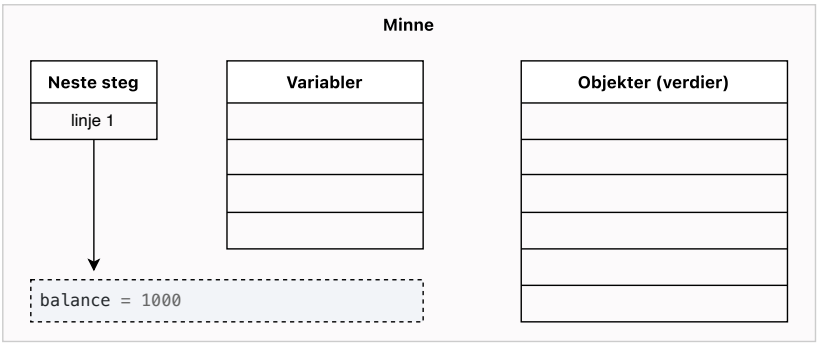
En variabel er en navngitt referanse til en verdi. Når vi tilordner en ny verdi til en variabel, endrer vi hvilket objekt variabelnavnet refererer til. Den gamle verdien kan faktisk bli liggende i minnet en stund selv uten at noen peker på den – men etter en stund vil objekter som ingen refererer til bli automatisk slettet for å frigjøre plassen til noe annet.

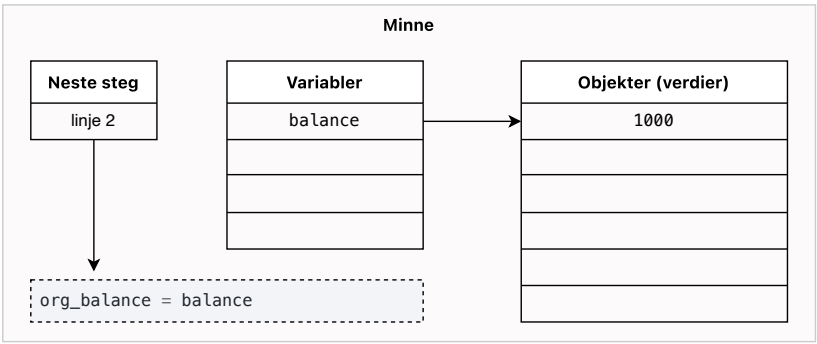
Vi skal benytte følgende eksempel for å illustrere hva som skjer i minnet når en variabel endrer seg. Les gjennom koden og tenk igjennom hva hensikten med koden er; kjør koden og se hva den skriver ut. Deretter kan du klikke deg gjennom steg for steg hvordan minnet endrer seg.

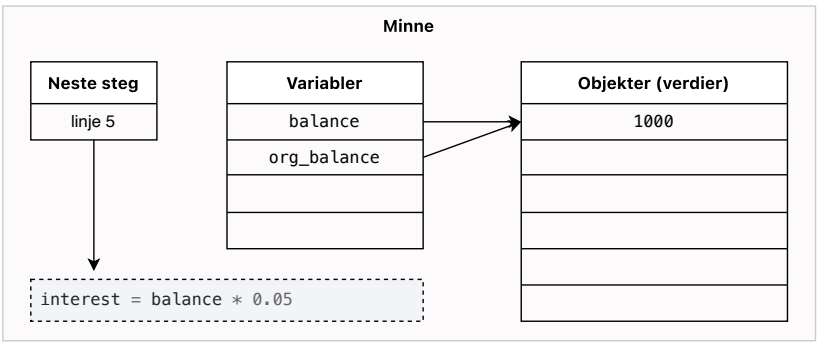
> *Tips: forsøk å forutsi hva som skjer i minnet før du klikker deg videre til neste steg.*

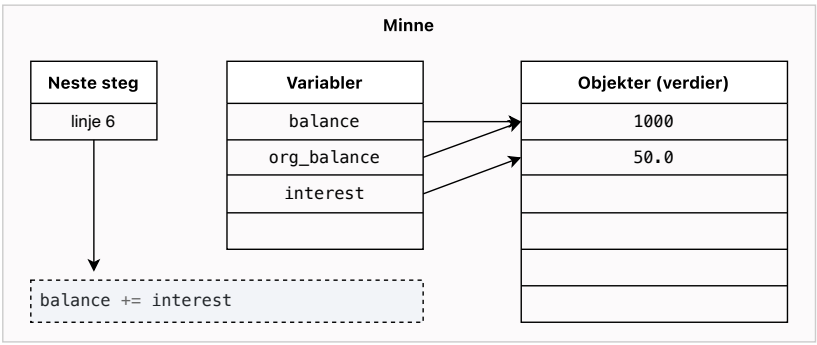
```
1  balance = 1000
2  org_balance = balance
3
4  # Regn ut renter etter ett år (vi antar 5% rente)
5  interest = balance * 0.05
6  balance += interest
7  print(f'Etter ett år har vi {balance} kroner på konto')
8
9  # Etter to år
10 interest = balance * 0.05
11 balance += interest
12 print(f'Etter to år har vi {balance} kroner på konto')
13
14 difference = balance - org_balance
15 print(f'Vi har fått {difference} kroner i renter etter to år')
```

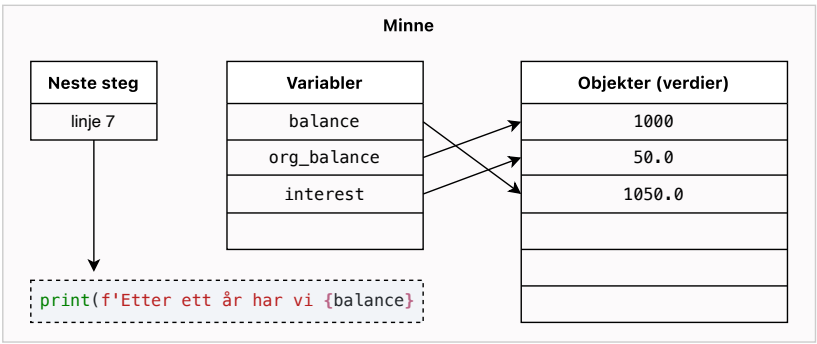
↑ Innhold


Minne
Neste steg
linje 1
Variabler
Objekter (verdier)
balance = 1000

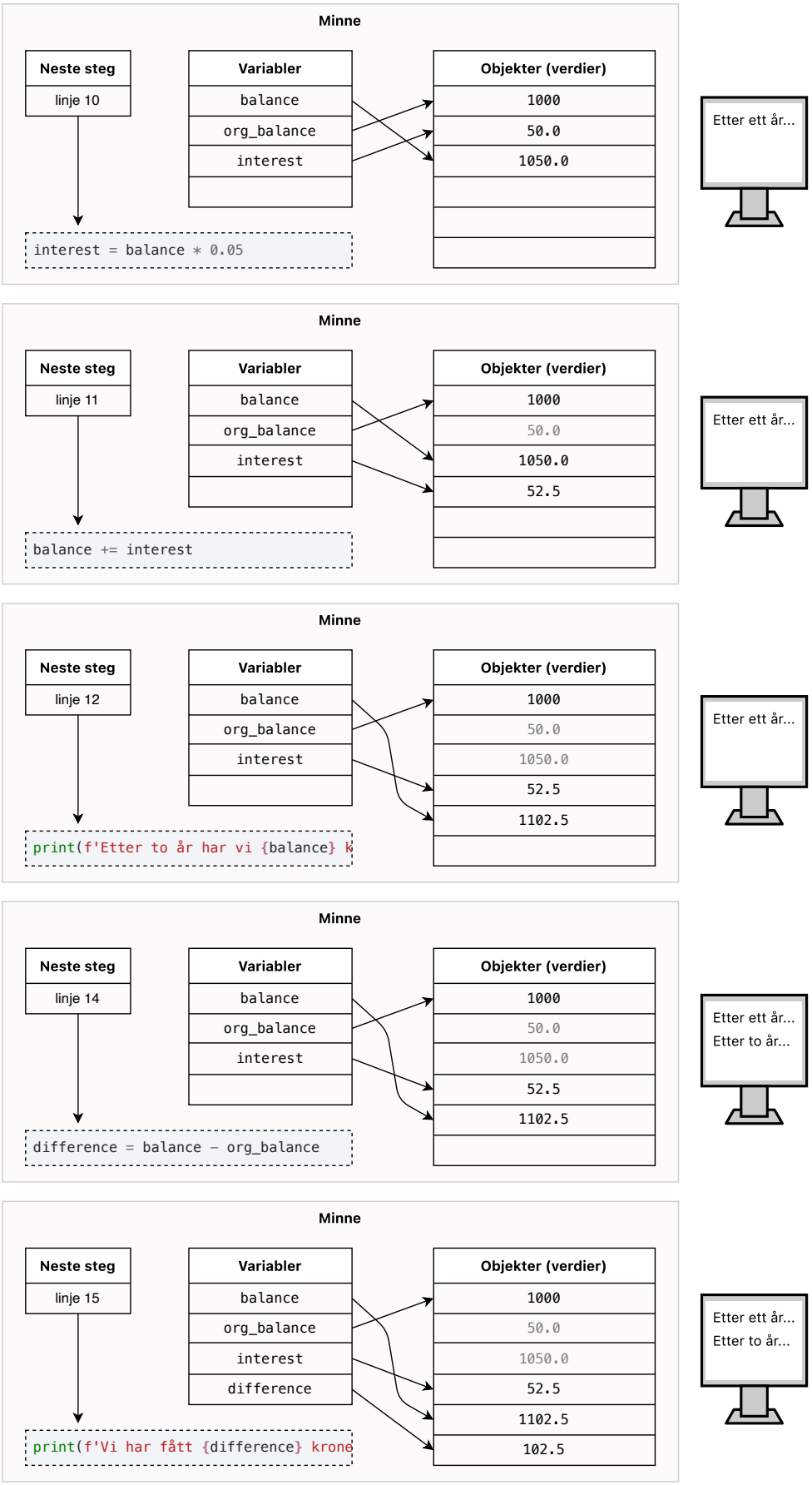

Minne
Neste steg
linje 2
Variabler
balance
Objekter (verdier)
1000
org_balance = balance


Minne
Neste steg
linje 5
Variabler
balance
org_balance
Objekter (verdier)
1000
interest = balance * 0.05

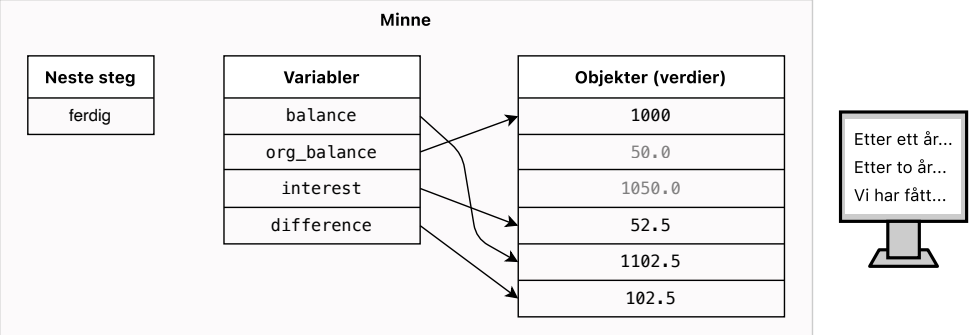

Minne
Neste steg
linje 6
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
balance += interest


Minne
Neste steg
linje 7
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
1050.0
print(f'Etter ett år har vi {balance}

↑ Innhold


Minne
Neste steg
linje 10
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
1050.0
interest = balance * 0.05
Etter ett år...
Minne
Neste steg
linje 11
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
1050.0
52.5
balance += interest
Etter ett år...
Minne
Neste steg
linje 12
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
1050.0
52.5
1102.5
print(f'Etter to år har vi {balance} k
Etter ett år...
Minne
Neste steg
linje 14
Variabler
balance
org_balance
interest
Objekter (verdier)
1000
50.0
1050.0
52.5
1102.5
difference = balance - org_balance
Etter ett år...
Etter to år...
Minne
Neste steg
linje 15
Variabler
balance
org_balance
interest
difference
Objekter (verdier)
1000
50.0
1050.0
52.5
1102.5
102.5
print(f'Vi har fått {difference} krone
Etter ett år...
Etter to år...

↑ Innhold



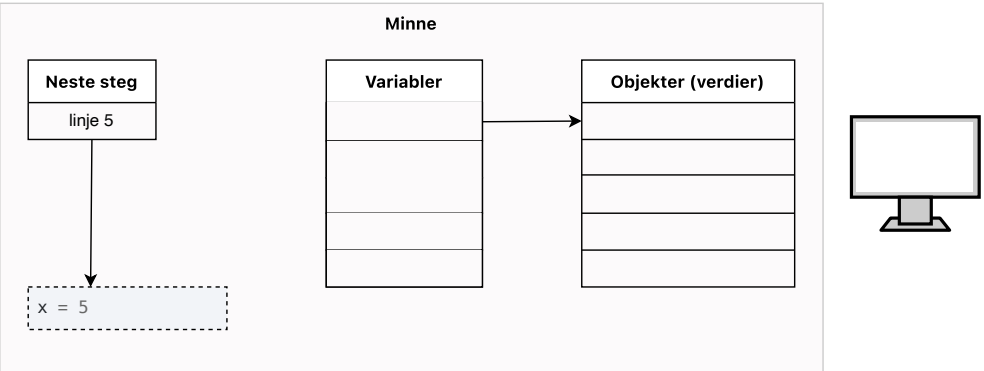
Legg spesielt merke til forskjellen på `balance`-variabelen før og etter at linje 6 blir utført: det opprettes en ny verdi i minnet som balance-variabelen nå peker på. Den gamle verdien blir værende i minnet, og `org_balance` -variabelen refererer fremdeles til den gamle verdien.

> *En variabel er en navngitt referanse; men hva er egentlig en referanse? I tegningene over tegner vi referanser som piler – men egentlig er det en tallverdi som angir på hvilken posisjon i listen over objekter verdien som det referes til ligger. Vi kan tenke på en referanse som en «adresse» til en posisjon i minnet hvor det befinner seg et objekt.*

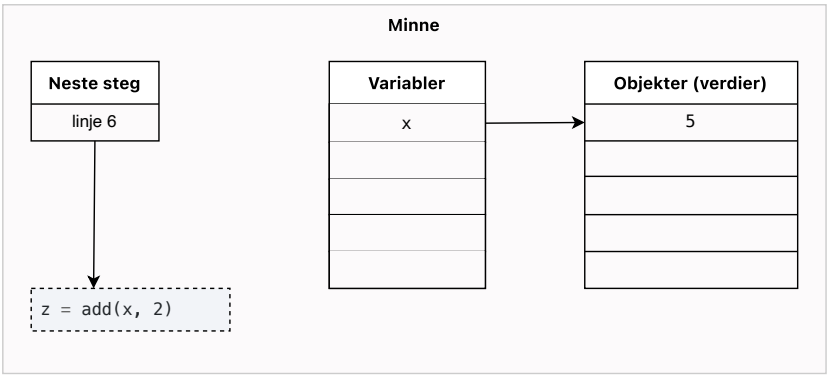
En funksjon blir kalt

Når en funksjon blir kalt, opprettes det en ny *funksjonsramme* i minnet. En funksjonsramme er egentlig bare en ny samling med variabler som legger seg «oppå» de gamle. Når funksjonen er ferdig, blir funksjonsrammen slettet fra minnet.

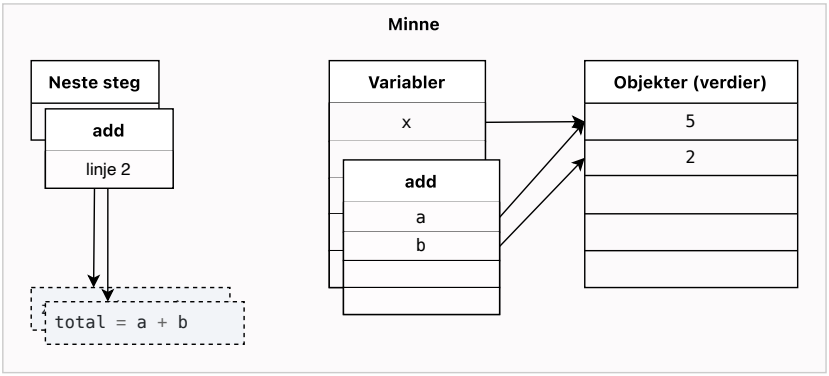
```
1  def add(a, b):
2      total = a + b
3      return total
4
5  x = 5
6  z = add(x, 2)
7  z = add(z, 1)
8  print(z) # 8
```



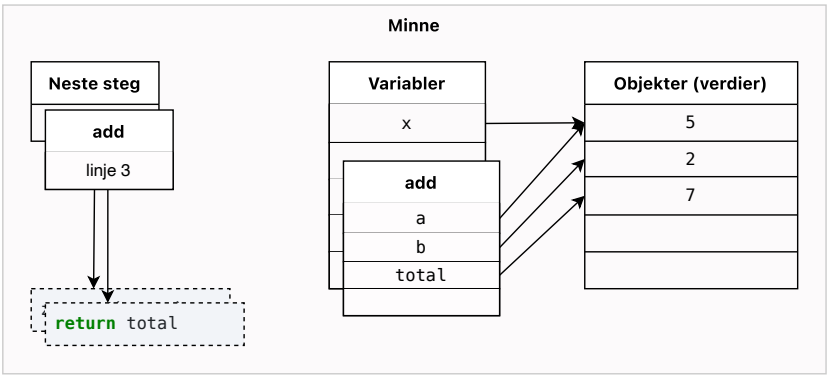
↑ Innhold


Minne
Neste steg
linje 6
z = add(x, 2)
Variabler
x
Objekter (verdier)
5

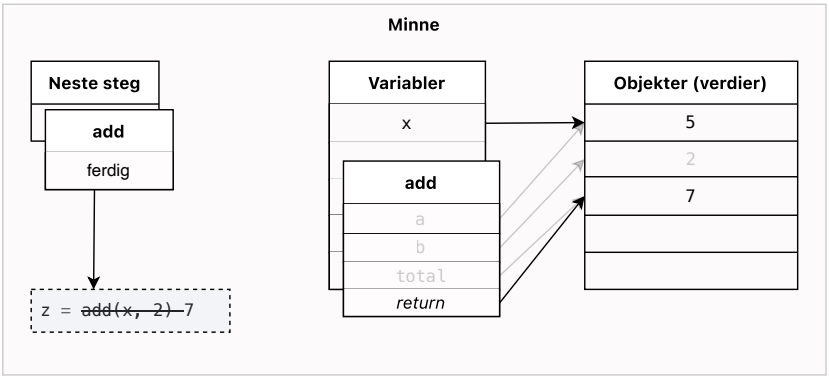



Minne
Neste steg
add
linje 2
total = a + b
Variabler
x
add
a
b
Objekter (verdier)
5
2

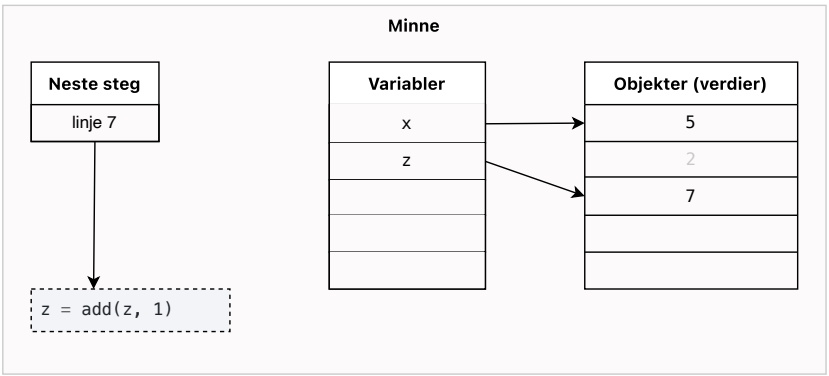



Minne
Neste steg
add
linje 3
return total
Variabler
x
add
a
b
total
Objekter (verdier)
5
2
7



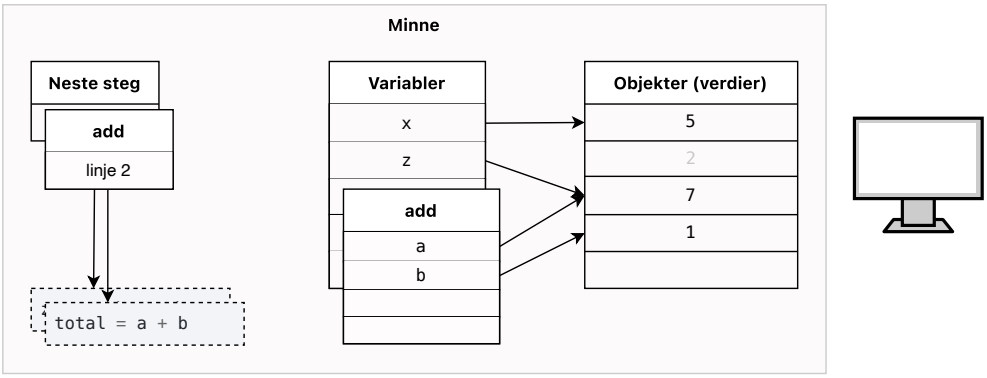

Minne
Neste steg
add
ferdig
z = add(x, 2) 7
Variabler
x
add
a
b
total
return
Objekter (verdier)
5
2
7

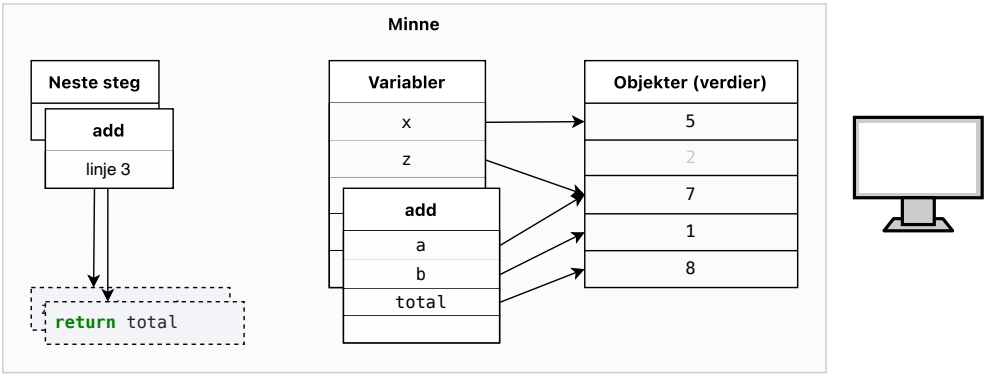


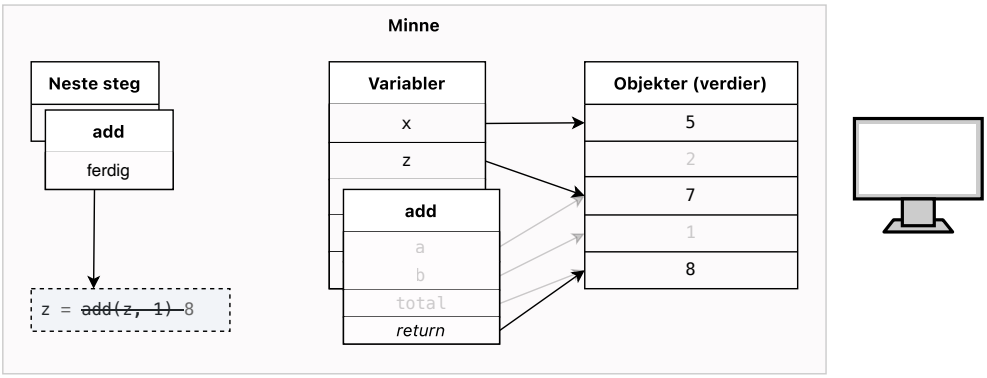

Minne
Neste steg
linje 7
z = add(z, 1)
Variabler
x
z
Objekter (verdier)
5
2
7

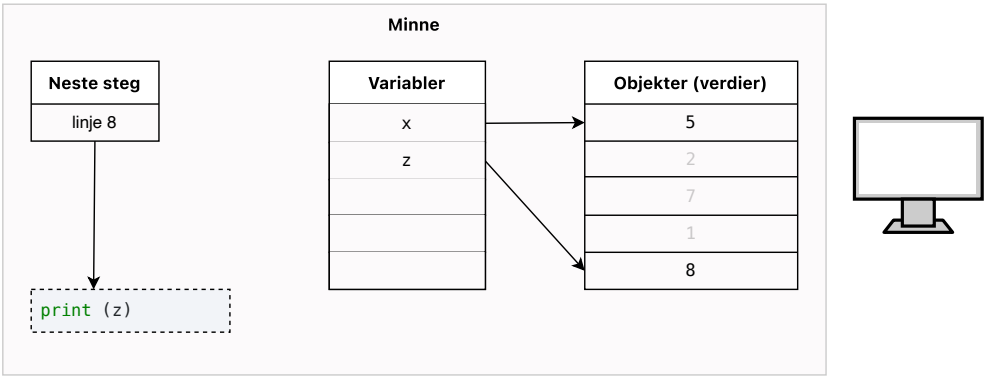


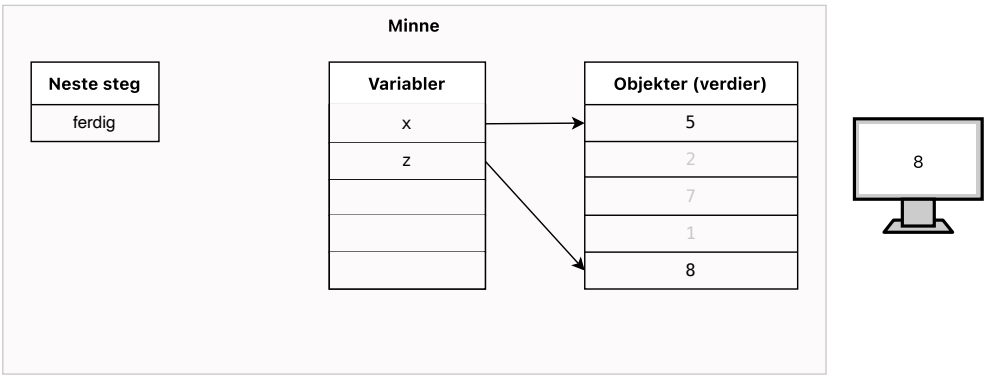
↑ Innhold


Minne
Neste steg
add
linje 2
total = a + b
Variabler
x
z
add
a
b
Objekter (verdier)
5
2
7
1


Minne
Neste steg
add
linje 3
return total
Variabler
x
z
add
a
b
total
Objekter (verdier)
5
2
7
1
8


Minne
Neste steg
add
ferdig
z = add(z, 1) 8
Variabler
x
z
add
a
b
total
return
Objekter (verdier)
5
2
7
1
8


Minne
Neste steg
linje 8
print (z)
Variabler
x
z
Objekter (verdier)
5
2
7
1
8


Minne
Neste steg
ferdig
Variabler
x
z
Objekter (verdier)
5
2
7
1
8
8

↑ Innhold

Debuggeren

I VSCode (og egentlig alle gode kodeeditorer for Python) finnes det en debug-modus som tillater oss å gå gjennom vår egen kode steg for steg på lignende måte som «se steg» -knappen gjør det for oss med kodeeksemplene på denne nettsiden.

En god gjennomgang laget av Boris Paskhaver:

Manuell kodesporing med tabell

Modellen av hva som skjer i minnet som er beskrevet i avsnittene over, gir oss en god forståelse av hva som egentlig skjer. Ulempen er at det er veldig mange tegninger å lage hvis du manuelt skal spore flere steg.

En *kodesporingstabell* er en forenklet modell av minnet som lar oss visualisere hva som skjer over flere steg på en mer kompakt måte. Denne metoden for å spore kode er som regel god nok i praksis; i hvert fall så lenge vi har den mer presise modellen i bakhodet.

Når vi sporer koden med en tabell, oppretter vi en tabell med én kolonne for hver variabel som finnes i den delen av koden vi skal spore. Den første kolonnen kan være en tidlinje som viser hvilket linjenummer i koden som skal utføres, og den siste kolonnen kan være utskrift. Deretter fyller vi ut rad for rad i tabellen nedover. Når vi kommer til et nytt steg i koden, fyller vi ut en ny rad i tabellen.

Samme eksempel som vi har sett tidligere:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
balance = 1000
org_balance = balance

# Regn ut renter etter ett år (vi antar 5% rente)
interest = balance * 0.05
balance += interest
print(f'Etter ett år har vi {balance} kroner på konto')

# Etter to år
interest = balance * 0.05
balance += interest
print(f'Etter to år har vi {balance} kroner på konto')

difference = balance - org_balance
print(f'Vi har fått {difference} kroner i renter etter to år')
```

↑ Innhold

linje	balance	org_balance	interest	difference	utskrift
1	~~1000~~				
2		1000			
5			~~50.0~~		
6	~~1050.0~~				
7					Etter ett år har vi 1050.0 kroner på konto
10			52.5		
11	1102.5				
12					Etter to år har vi 1102.5 kroner på konto
14				102.5	
15					Vi har fått 102.5 kroner i renter etter to år

- Vi hopper over blanke linjer i kildekoden. Vi kan også hoppe over linjer med utskrifter vi ikke er interessert i å spore.
- Det er ikke nødvendig å fylle ut alle cellene, bare de som endrer seg (vi vet jo at den reelle verdien til en ikke-utfylt celle vil være den nederste verdien som er fylt inn i samme kolonne).
- Hvis du har det travelt (og holder tungen rett i munnen) er det ikke nødvendig å inkludere kolonnen med linjenummer.

Manuell kodesporing av funksjoner med tabell

Et eksempel på kodesporing som inkluderer funksjonsskall:

```
1  def add(a, b):
2      total = a + b
3      return total
4
5  x = 15
6  z = add(x, 7)
7  z = add(z, 1)
8  print(z) # 8
```

↑ Innhold

linje	x	z	add a	add b	add total	add returverdi	utskrift
5	15						
6; 1			~~15~~	~~7~~			
6; 2					~~22~~		
6; 3			—	—	—	22	
6		~~22~~					
7; 1			~~22~~	~~1~~			
7; 2					~~23~~		
7; 3			—	—	—	23	
7		23					
8							23

- Når kode inne i et funksjonskall kjøres, er vi på en måte flere steder i koden samtidig: både ved den konkrete kodelinjen inne i funksjonen som utføres, men også på linjen hvor funksjonen ble kalt. For eksempel er vi både på linje 6 og på linje 2 når total-variabelen blir satt til 22.
- Funksjonen add blir kalt to ganger, både på linje 6 og på linje 7. Variablene a, b, total og returverdiene for disse to ulike kallene til add-funksjonen er *egentlig* helt forskjellige variabler, som ikke hører hjemme i samme kolonne; men fordi vi vet at variablene i det første kallet fjernes før det andre kallet begynner, jukser vi litt og bruker samme kolonne likevel. Vi markerer at variabelen er slettet med å skrive en strek i tabellen der variablene opphører å eksistere.

Feil og debugging

↑ Innhold

Man vil ofte oppleve å ha såkalte *bugs* i koden, som gjør at programmet vårt ikke virker. Hvis vi er heldig, krasjer programmet med en gang, slik at vi kan finne feilen ved å lese feilmeldingen. Er vi litt mindre heldig, oppdager vi at programmet ikke virker som det skal fordi vi skjønner at svarene eller oppførselen til programmet må være feil. I verste fall oppdager vi ikke feilen i det hele tatt, men lever med et program som gir oss feil data uten at vi er klar over det.

Ulike former for feil:

- Syntaks-feil
- Krasj (kjøretidsfeil)
- Logiske feil

Strategier for å undersøke logiske feil

- Assert
- Print
- Se steg med Python Tutor
- VSCode sin debugger

Strategier for å unngå feil

- Test-drevet utvikling

Syntaks-feil

Hvis programmet krasjer før det i det hele tatt har begynt, har du en *syntaks*-feil. I disse tilfellene gir ofte feilmeldingen en god visuell indikasjon på hvor feilen ligger. Om du bruker en teksteditor laget for Python, vil den som regel gi beskjed om syntaksfeil i form av røde streker.

```
print('foo') # kjøres ikke
x = 1 ? 2
print('bar') # kjøres ikke
```

Flere eksempler på syntaks-feil

```
x = 3
if x < 5:
    print('hurra)

#   File "/demo/demo.py", line 3
#     print('hurra)
#           ^
# SyntaxError: unterminated string literal (detected at line 3)
```

```
x = 3
if x < 5:
    print 'hurra')

#   File "/demo/demo.py", line 3
#     print 'hurra')
#                  ^
# SyntaxError: unmatched ')'
```

```
x = 3
if x < 5:
print('hurra')

#   File "/demo/demo.py", line 3
#     print('hurra')
#     ^
# IndentationError: expected an indented block after 'if' statement on line
2
```

```
x = 3
if x < 5
    print('hurra')

#   File "/demo/demo.py", line 2
#     if x < 5
#             ^
# SyntaxError: expected ':'
```

```
x = 3
if x < 5;
    print('hurra')

#   File "/demo/demo.py", line 2
#     if x < 5;
#             ^
# SyntaxError: invalid syntax
```

↑ Innhold

```
3 = x
if x < 5:
    print('hurra')

#   File "/demo/demo.py", line 1
#     3 = x
#     ^
# SyntaxError: cannot assign to literal here. Maybe you meant '==' instead
of '='?
```

Krasj (kjøretidsfeil)

En kjøretidsfeil (engelsk: runtime error) fører til at programmet krasjer underveis når det kjører. Vanlige kjøretidsfeil er blant annet NameError, AttributeError, TypeError, IndexError, ZeroDivisionError og FileNotFoundError

```
print('foo') # kjøres
x = '42' + 1
print('bar') # kjøres ikke
```

Flere eksempler på kjøretidsfeil

NameError • AttributeError • TypeError • IndexError • ZeroDivisionError • FileNotFoundError

NameError

```
color = 'green'
print(colour)

#   File "/demo/demo.py", line 2, in <module>
#     print(colour)
# NameError: name 'colour' is not defined. Did you mean: 'color'?
```

```
def foo(x):
    return x*x

print(bar(2))

#   File "/demo/demo.py", line 4, in <module>
#     print(bar(2))
# NameError: name 'bar' is not defined
```

```
x = 8
if x > 100:
    msg = 'Huzza'
print(msg)

#   File "/demo/demo.py", line 4, in <module>
#     print(msg)
# NameError: name 'msg' is not defined
```

UnboundLocalError er også en variant av NameErrror

```
def foo():
    y = y + 1

foo()

#   File "/demo/demo.py", line 3, in foo
#     y = y + 1
# UnboundLocalError: local variable 'y' referenced before assignment
```

AttributeError

```
s = 'FOO'
print(s.convert_to_lowercase())

#   File "/demo/demo.py", line 3, in <module>
#     print(s.convert_to_lowercase())
# AttributeError: 'str' object has no attribute 'convert_to_lowercase'
```

TypeError

```
print('foo' + 2)

#   File "/demo/demo.py", line 1, in <module>
#     print('foo' + 2)
# TypeError: can only concatenate str (not "int") to str
```

```
a = ['foo', 'bar']
a['zig'] = 'zag'

#   File "/demo/demo.py", line 3, in <module>
#     a['zig'] = 'zag'
# TypeError: list indices must be integers or slices, not str
```

IndexError

```
a = ['foo', 'bar']
a[2] = 'zag'

#   File "/demo/demo.py", line 2, in <module>
#     a[2] = 'zag'
# IndexError: list assignment index out of range
```

```
s = 'foo'
print(s[-4])

#   File "/demo/demo.py", line 2, in <module>
#     print(s[-4])
# IndexError: string index out of range
```

ZeroDivisionError

```
x = 5
y = 0
print(x/y)

#   File "/demo/demo.py", line 3, in <module>
#     print(x/y)
# ZeroDivisionError: division by zero
```

FileNotFoundError

```
filename = 'no/such/file.txt'
with open(filename, 'rt', encoding='utf-8') as f:
    content = f.read()
    print(content)

#   File "/demo/demo.py", line 2, in <module>
#     with open(filename, 'rt', encoding='utf-8') as f:
# FileNotFoundError: [Errno 2] No such file or directory: 'no/such/file.txt'
```

↑ Innhold

Logiske feil

Logiske feil oppstår når programmet kjører, men gir oss feil svar.

```
x = 2
y = 3
z = x + y
print(f'{x} + {z} = {y}') # Logisk feil, vi har byttet z og y

# Gir output:
#   2 + 5 = 3
```

Assert

Ordet *assert* har flere mulige oversettelser til norsk; men de oversettelsene som passer best i vår kontekst er «å påse» eller «å forsikre om» at noe er sant.

En assert er en måte for oss til å *krasje programmet på egen hånd* dersom vi oppdager at noe ikke er som det skal. Dette hjelper oss å finne logiske feil så tidlig som mulig.

Hvorfor krasje programmet med vilje?

- *Det er bedre å krasje enn å få feil svar.*
- *Hvis programmet krasjer, er det best om det krasjer så nærmt feilen som mulig.*

```
everything_is_ok = True
assert everything_is_ok # Her skjer det ingen ting
...
everything_is_ok = False
assert everything_is_ok # Krasjer!

#   File "/demo/demo.py", line 5, in <module>
#     assert everything_is_ok
# AssertionError
```

↑ Innhold

Vi kan bruke assert-setninger rundt om kring i koden for å sjekke at ting er slik vi forventer.

```
def calculate_rectangle_area(width, height):
    # Forsikrer oss om at bredden og høyden ikke er negative tall
    assert width >= 0
    assert height >= 0

    # Utfører funksjonen som ellers
    return width * height

print(calculate_rectangle_area(2, 16)) # Fungerer fint
print(calculate_rectangle_area(6, -3)) # Krasjer!
```

Eller vi kan bruke assert-setninger for å teste at funksjoner og hjelpefunksjoner gir de svarene vi forventer

```
def distance(x0, y0, x1, y1):
    return ((x0 - x1)**2 + (y0 - x1)**2)**0.5

assert 5 == distance(0, 3, 4, 0) # Ojsann, her oppdager vi at noe er feil!
```

Fordelen med assert-setninger (i forhold til print-setninger, se under) er at en assert er helt stille og plager ingen så lenge ting fungerer som de skal; men sier i fra med én gang noe er feil. Ulempen er at de ikke gir særlig detaljert informasjon.

Print

Assert-setninger kan fortelle oss at noe er feil. Men ofte er det slik at vi ikke helt vet hva som er feil, eller hvorfor det ble feil. Da ønsker vi å spore hva koden gjør; og da er det nyttig å vite hvilke verdier som faktisk befinner seg i programmet vårt. For å se dette kan vi bruke print-setninger.

Når vi bruker print-setninger for feilsøkings-formål, kan det være nyttig å skrive ut verdien til alle variablene vi har. Dette kan gjøres med `print(locals())`, som vil skrive ut verdien til alle lokale variabler (altså variabler som er definert inne i funksjonen vi befinner oss i). Dersom du kaller `print(locals())` utenfor en funksjon, vil du få en liste over alle variablene som er definert i det globale skopet – merk at dette også inkluderer en del variabler som er definert av Python selv, og som vi ikke trenger å bry oss om.

Tips: hvis du bruker mer enn én print-setning, er det lurt å inkludere inkludere informasjon om hvor i koden du befinner deg. For eksempel `print('debug: line 3', locals())` vil skrive ut en linje som begynner med «debug: line 3» og deretter skriver ut alle lokale variabler.

Eksempel på debugging med print-setning: innrammet tekst

↑ Innhold

Dette programmet rammer inn tre linjer med tekst og skriver det ut til skjermen.

```
# DENNE KODEN HAR EN BUG (MED VILJE)

def frame_border(length):
    result = '*' * length
    return result

def frame_line(text, length):
    extra_spaces = length - len(text) - 4
    center_string = text + ' ' * extra_spaces
    return '* ' + center_string + ' *'

def frame_text(line1, line2, line3):
    longest_word = max(line1, line2, line3)
    length = len(longest_word) + 4

    result = (
        frame_border(length) + '\n' +
        frame_line(line1, length) + '\n' +
        frame_line(line2, length) + '\n' +
        frame_line(line3, length) + '\n' +
        frame_border(length)
    )
    return result

result = frame_text('Foo', 'Hello', 'Chilibom')
print(result)
```

Vi kjører kode over og ser at vi får følgende output (som inneholder en logisk feil):

```
*********
* Foo   *
* Hello *
* Chilibom *
*********
```

Vi ser at rammen ikke er bred nok til å omslutte det lengste ordet, som er «Chilibom.» Kan det være at noe er feil med hvordan `length` -variabelen beregnes? Vi setter inn en velplassert print-setning like etter (eller før) lengden på linjen er beregent, slik at vi ser hvilke verdier som inngikk i å beregne den.

```
# DENNE KODEN HAR EN BUG (MED VILJE)
# Eksempel på bruk av print-setninger for å lete etter feilen

def frame_border(length):
    result = '*' * length
    return result

def frame_line(text, length):
    extra_spaces = length - len(text) - 4
    center_string = text + ' ' * extra_spaces
    return '* ' + center_string + ' *'

def frame_text(line1, line2, line3):
    longest_word = max(line1, line2, line3)
    length = len(longest_word) + 4
    print('debug: frame_text line 3', locals())  # <-- VELPLASSERT PRINT

    result = (
        frame_border(length) + '\n' +
        frame_line(line1, length) + '\n' +
        frame_line(line2, length) + '\n' +
        frame_line(line3, length) + '\n' +
        frame_border(length)
    )
    return result

result = frame_text('Foo', 'Hello', 'Chilibom')
print(result)
```

Vi får nå denne utskriften:

```
debug: frame_text line 3 {'line1': 'Foo', 'line2': 'Hello', 'line3':
'Chilibom', 'longest_word': 'Hello', 'length': 9}
*********
* Foo   *
* Hello *
* Chilibom *
*********
```

Vi inspiserer variablene, og finner noe urovekkende: `longest_word` er satt til `'Hello'`, selv om det lengste ordet er `'Chilibom'`. Det forklarer i det minste hvorfor lengden ikke ble riktig.

Det viser seg at max-funksjonen ikke gjorde som vi forventet: vi ønsket at den skulle gi oss det lengste ordet, men i stedet gav den oss det siste ordet i alfabetet. Da kan vi fikse koden ved å bruke `max`-funksjonen på en annen måte, for eksempel slik:

```
length = max(len(line1), len(line2), len(line3)) + 4
```

Ulempen med print-setninger er at man må fjerne dem når man er ferdig med å fikse bug'en; og hvis det blir veldig mange print-setninger, kan det være krevende å lese gjennom.

Se steg med Python Tutor

Et alternativ til å bruke print-setninger for å se verdiene i programmet, er å kjøre koden i Python Tutor sitt visualiseringsverktøy (https://pythontutor.com/visualize.html#mode=edit). Dette vil fungere så lenge koden din befinner seg kun i én fil og ikke benytter seg av eksotiske biblioteker, leser filer, bruker nettverk eller kjører parallelle prosesser; altså er det mest aktuelt for små og enkle programmer. Til gjengjeld er visualiseringen svært god, og man kan ta steg både fremover og bakover i tid.

I verktøyet kan man kopiere inn koden sin, gå gjennom koden steg for steg, og se hvordan variablene endrer seg. I kursnotatene kan man klikke på «se steg» -knappen for å laste eksempelet automatisk inn i dette verktøyet.

VSCode sin debugger

Debug-verktøyet i VSCode (og andre gode kodeeditorer) er den profesjonelle utvikleren sitt viktigste verktøy. Det fungerer nesten som Python Tutor sitt visualiseringsverktøy, men har en del flere funksjoner, og fungerer uten de begrensningene som ligger i Python Tutor. Det eneste Python Tutor kan gjøre som ikke kan gjøres med denne debuggeren, er å gå «baklengs» i tid gjennom stegene (som riktignok er en svært hendig funksjon, og en grunn til å bruke Python Tutor hvis det ellers er egnet).

En god gjennomgang laget av Boris Paskhaver:

Test-drevet utvikling

Test-drevet utvikling er en måte å skrive kode på hvor man kontinuerlig inkluderer tester for alle delene av koden man skriver. Typisk skrives testene som assert-setninger i en eller annen form. I større prosjekter kan man gjerne ha egne filer som kun inneholder tester for «produksjonskoden».

```
import other_file as M

assert 5 == M.distance(0, 3, 4, 0)
```

I dette kurset har noen av oppgavene i labene inkludert egne assert-setninger som tester funksjonen som skal skrives. Dette er et eksempel på test-drevet utvikling, hvor vi allerede har gjort litt av jobben for dere. Dersom det ikke er ferdigskrevne tester fra før, eller testene som finnes fra før er for dårlige, bør vi skrive våre egne tester.

Noen mener at testene alltid skal skrives *før* koden. Det kan ofte være en god idé; men det er også nyttig å skrive tester like etter man (tror man) er ferdig. Da kan man oppdage feil man har gjort. En av de viktigste funksjonen ved tester er dessuten å beskytte kode mot idioter fra fremtiden (gjerne oss fremtidige selv)

som ønsker å endre på koden. Hvis vi har vært flinke til å utstyre koden vår med tester, vil ødeleggende endringer som gjøres i fremtiden oppdages med én gang.

↑ Innhold

Grafikk

↑ Innhold

- Kom i gang
- Koordinatsystemet

Grunnleggende tegnefunksjoner

- create_rectangle
- create_oval
- create_line
- create_polygon
- create_arc
- create_text
- create_image

Flere muligheter

- Farger
- Tekst i boks
- Bilde i boks

Eksempler

- Buss

Kom i gang

I dette kurset benytter vi et bibliotek for å lage grafikk som heter *uib-inf100-graphics*. Dette biblioteket er en forenklet utgave av et standard grafikk-rammeverk i Python som heter *tkinter*.

- Selv om vi har gjort noen forenklinger for å komme rask i gang, gir rammeverket frihet til å være kreativ og lage et rikt utvalg av grafiske applikasjoner.
- Alt vi lærer vil være direkte anvendbart i tkinter hvis du bestemmer deg for å oppgradere til et større rammeverk senere.

Forkrav

I dette kurset benytter vi versjon **0.4.0** av uib-inf100-graphics.

Før du kan installere uib-inf100-graphics, må du

- Ha installert Python 3.10 eller nyere, og
- Ha installert tkinter.

Tkinter følger automatisk med dersom du installerte Python med hjelp av installeren fra python.org (https://python.org/), men fulgte ikke med hvis du installerte Python med hjelp av en pakkebehandler som f.eks. `apt` eller `brew`. Hvis du ikke har tkinter, må du installere det før du kan installere uib-inf100-graphics.

Installasjon med script

Den letteste måten å installere *uib-inf100-graphics* er å kopiere skripet under inn i en tom Python-fil og så kjøre den. Svar `yes` når du blir spurt om du vil installere pakken.

```
import sys
from subprocess import run

package_name = 'uib-inf100-graphics'

# Sjekk at Python-versjonen er 3.10 eller nyere
if ((sys.version_info[0] != 3) or (sys.version_info[1] < 10)):
    raise Exception(f'{package_name} requires Python 3.10 or later. '
            + 'Your current version is: '
            + f'{sys.version_info[0]}.{sys.version_info[1]}')

# Spør brukeren om å installere pakken
ans = input(f'\n\nType yes to install {package_name}: ')
if ((ans.lower() == 'yes') or (ans.lower() == 'y')):
    print()

    # Oppdater pip
    cmd_pip_update = f'{sys.executable} -m pip install --upgrade pip'
    print(f'Attempting to update pip with command: {cmd_pip_update}')
    run(cmd_pip_update.split())
    print()

    # Installer pakken
    cmd_install = f'{sys.executable} -m pip install {package_name}'
    print(f'Attempting to install {package_name} with command:
{cmd_install}')
    run(cmd_install.split())
else:
    print(f'Did not attempt to install {package_name} now')
print('\n')
```

Installasjon med terminal

Åpne terminalen (Windows: PowerShell). Skriv inn følgende kommando for å oppdatere pip:

```
python -m pip install --upgrade pip
```

Skriv deretter denne kommandoen for å installere *uib-inf100-graphics*:

```
python -m pip install uib-inf100-graphics
```

Dersom du får en feilmelding om at python-kommandoen ikke finnes, prøv å erstatte `python` med `python3`, `python3.11` eller `py` i stedet.

For å sjekke at installasjonene var vellykket, opprett en ny Python-fil og skriv inn følgende:

```
from uib_inf100_graphics.simple import canvas, display

canvas.create_rectangle(100, 50, 300, 150, outline='red')
canvas.create_text(200, 100, text='Hei, grafikk!', font='Arial 20 bold')

display(canvas)
```

Når du kjører filen skal du se et vindu som ser omtrent slik ut:

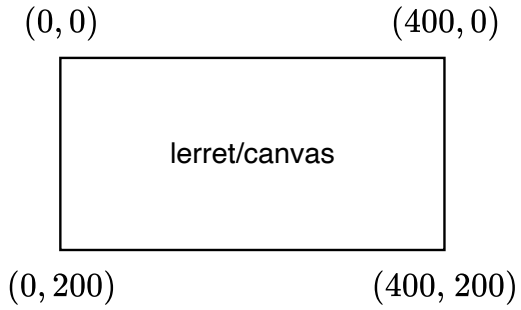


Koordinatsystemet

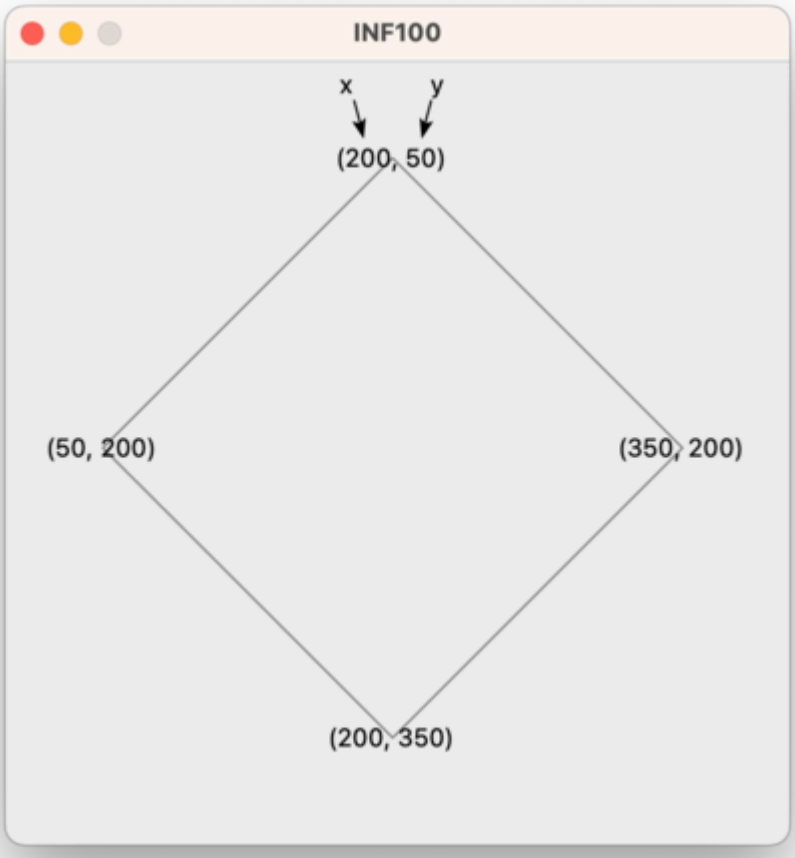
Ulikt det vi er vant til fra matematikken på skolen, vokser y-aksen *nedover* istedet for oppover. Dermed er $(0, 0)$ punktet til venstre øverst på lerretet, mens punktet $(\text{width}, \text{height})$ er punktet til høyre nederst.

↑ Innhold

For et lerret med bredde 400 og høyde på 200, får hjørnene koordinatene under:



I eksempelet under har vi tegnet noen punkter på lerretet for å illustrere koordinatsystemet. Vindusstørrelsen ved bruk av `uib_inf100_graphics.simple` er 400x400 som standard.



Kildekode for programmet vist over

↑ Innhold

```
from uib_inf100_graphics.simple import canvas, display

# polygon defined by four (x, y) coordinates (the corners)
canvas.create_polygon(200, 50,
                      350, 200,
                      200, 350,
                      50, 200,
                      outline='gray',
                      fill='')

# draw labels on the same points as corners of polygon
canvas.create_text(200, 50, text='(200, 50)')
canvas.create_text(350, 200, text='(350, 200)')
canvas.create_text(200, 350, text='(200, 350)')
canvas.create_text(50, 200, text='(50, 200)')

# x-label with arrow
canvas.create_text(180, 20, text='x', anchor='se')
canvas.create_line(180, 20, 185, 40, arrow='last')

# y-label with arrow
canvas.create_text(220, 20, text='y', anchor='sw')
canvas.create_line(220, 20, 215, 40, arrow='last')

display(canvas)
```

create_rectangle

Påkrevde parametre *(x1, y1, x2, y2)*.

- De første to parametrene x1 og y1 er koordinatene til ett av rektangelets hjørner, mens de neste to parametrene x2 og y2 er koordinatene til det motsatte hjørnet. Konvensjon tilsier at (x1, y1) er hjørnet til venstre øverst, mens (x2, y2) er hjørnet til høyre nederst.

Valgfrie parametre *(outline, fill, width, …)*.

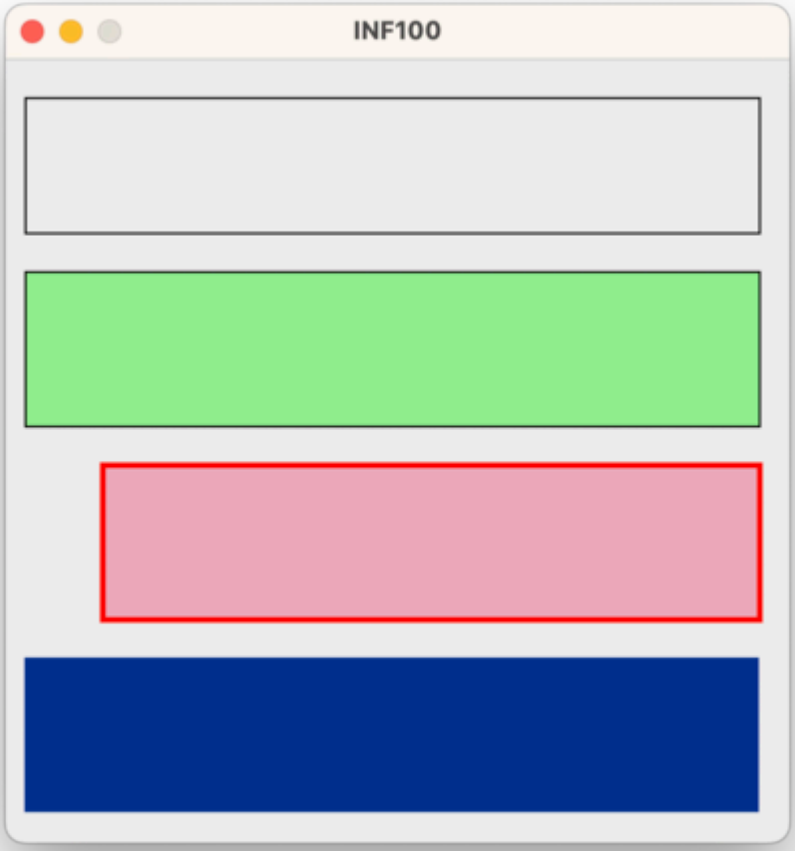
- Som standard tegnes rektangelet med en svart strek og uten farge i midten. Dette kan endres ved å angi farge-verdier til valgfrie parametrene `outline` og `fill`. Parameteren `width` kan benyttes for å angi tykkelsen (i antall piksler) til streken som tegnes.

↑ Innhold

```
from uib_inf100_graphics.simple import canvas, display

canvas.create_rectangle(10, 20, 390, 90)
canvas.create_rectangle(10, 110, 390, 190, fill='lightGreen')
canvas.create_rectangle(50, 210, 390, 290, fill='#eeaabb',
                        outline='red', width=3)
canvas.create_rectangle(10, 310, 390, 390, fill='#00308f', width=0)

display(canvas)
```



create_oval

Påkrevde parametre *(x1, y1, x2, y2)*.

- Plasseringen til en oval spesifiseres ved å angi koordinater som beskriver et tenkt rektangel som omslutter ovalen. De første fire parameterne skal være koordinatene til to motstående hjørner i dette rektangelet.

Valgfrie parametre *(outline, fill, width, …)*.

- Som standard tegnes ovalen med en svart strek og uten farge i midten. Dette kan endres ved å angi farge-verdier til valgfrie parametrene `outline` og `fill`. Parameteren `width` kan benyttes for å

angi tykkelsen (i antall piksler) til streken som tegnes.

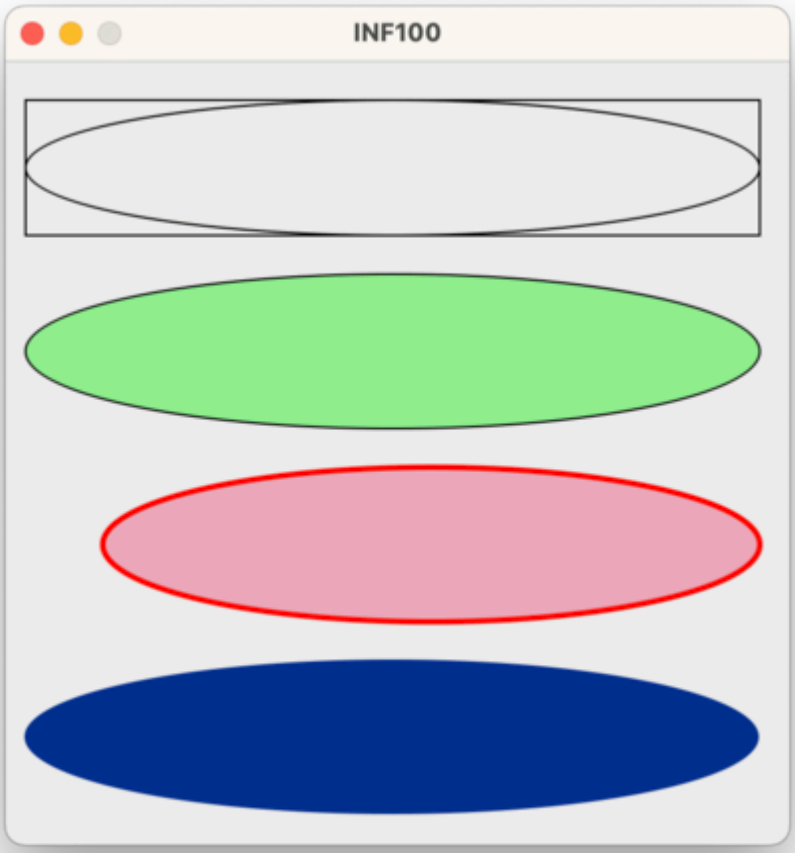
↑ Innhold

```
from uib_inf100_graphics.simple import canvas, display

canvas.create_oval(10, 20, 390, 90)
canvas.create_rectangle(10, 20, 390, 90)

canvas.create_oval(10, 110, 390, 190, fill='lightGreen')
canvas.create_oval(50, 210, 390, 290, fill='#eeaabb', outline='red', width=3)
canvas.create_oval(10, 310, 390, 390, fill='#00308f', width=0)

display(canvas)
```



create_line

Påkrevde parametre *(x1, y1, x2, y2, …)*.

- For å tegne en linje, må vi oppgi koordinatene til to (eller flere) punkter. De første to parameterne er koordinatene til det første punktet, mens de neste to parameterne er koordinatene til det andre punktet (og de to neste er koordinatene til det tredje punktet, og så videre).
- Det er mulig å angi punktene som en liste med koordinater i stedet for én og én koordinat.

↑ Innhold

Valgfrie parametre *(fill, width, arrow, smooth, …)*.

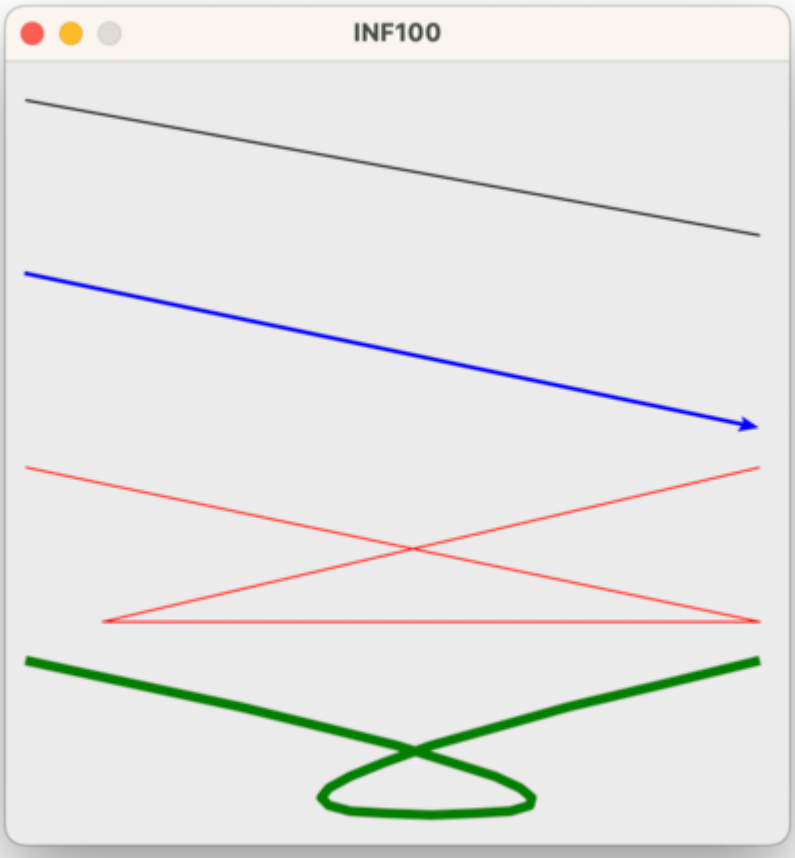
- Som standard tegnes linjen med en svart strek. Dette kan endres ved å angi farge-verdier til valgfrie parametrene `fill`. Parameteren `width` kan benyttes for å angi tykkelsen (i antall piksler) til streken som tegnes.
- Det er mulig å angi at streken skal starte eller slutte som en pil ved å angi `arrow="first"`, `arrow="last"` eller `arrow="both"` (standard er `arrow="none"`).
- Det er mulig å angi at streken skal tegnes med en glattere kurve ved å angi `smooth=True`.

```
from uib_inf100_graphics.simple import canvas, display

canvas.create_line(10, 20, 390, 90)
canvas.create_line(10, 110, 390, 190, fill='blue', width=2, arrow='last')
canvas.create_line(10, 210, 390, 290, 50, 290, 390, 210, fill='red')

points = [(10, 310), (390, 390), (50, 390), (390, 310)]
canvas.create_line(points, fill='green', width=5, smooth=True)

display(canvas)
```



↑ Innhold

create_polygon

Påkrevde parametre *(x1, y1, x2, y2, x3, y3, …)*.

- For å tegne en polygon, må vi oppgi koordinatene til tre eller flere punkter. De første to parameterne er koordinatene til det første punktet, mens de neste to parameterne er koordinatene til det andre punktet, og de to neste er koordinatene til det tredje punktet, og så videre. Dette ligner på å tegne en linje, men hvor den siste linjen for å lukke polygonen tegnes automatisk.
- Det er mulig å angi punktene som en liste med koordinater i stedet for én og én koordinat.

Valgfrie parametre *(fill, outline, width, smooth, …)*.

- Som standard tegnes polygonen uten at linjen tegnes, men med en svart fyllfarge. Dette kan endres ved å angi farge-verdier til valgfrie parametrene `fill` og `outline`. Parameteren `width` kan benyttes for å angi tykkelsen (i antall piksler) til streken som tegnes.
- Det er mulig å angi at streken skal tegnes med en glattere kurve ved å angi `smooth=True`.

```
from uib_inf100_graphics.simple import canvas, display

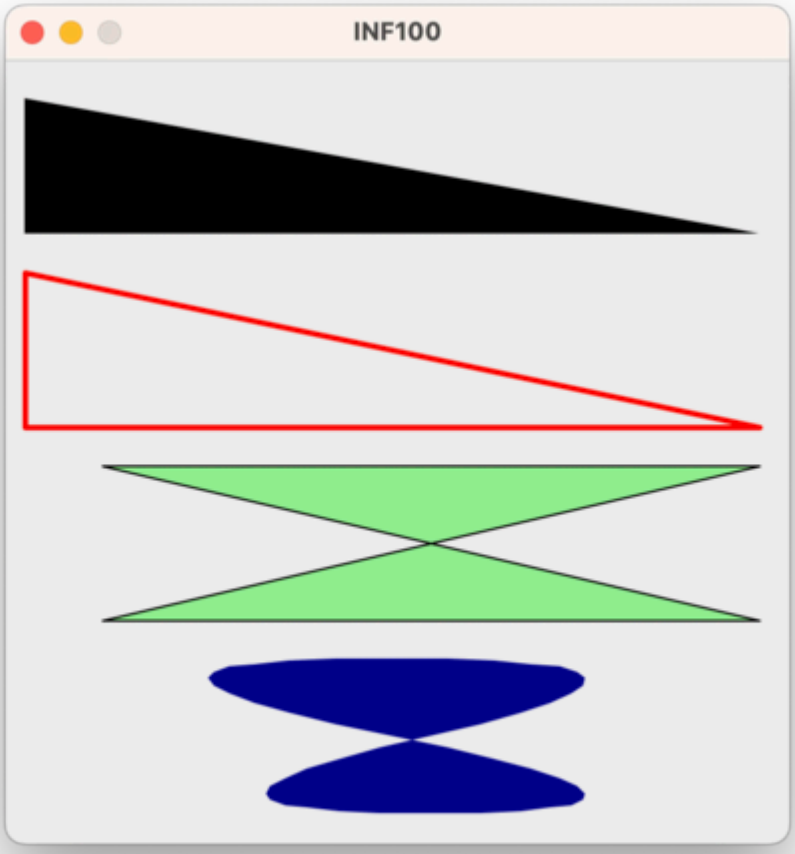
canvas.create_polygon(10, 20, 390, 90, 10, 90)
canvas.create_polygon(10, 110, 390, 190, 10, 190,
                      fill='', outline='red', width=3) # fill='' -> no fill

points = [50, 210, 390, 290, 50, 290, 390, 210]
canvas.create_polygon(points, fill='lightGreen', outline='black', width=1)

points = [(10, 310), (390, 390), (50, 390), (390, 310)]
canvas.create_polygon(points, fill='darkblue', smooth=True)

display(canvas)
```

↑ Innhold



create_arc

Påkrevde parametre *(x1, y1, x2, y2)*.

- For å tegne en bue, må vi oppgi koordinatene til to motstående hjørner i et rektangel som omslutter den ovalen buen er en del av. De første to parameterne er koordinatene til det første hjørnet, mens de neste to parameterne er koordinatene til det andre hjørnet. Konvensjon tilsier at (x1, y1) er hjørnet til venstre øverst, mens (x2, y2) er hjørnet til høyre nederst.

Valgfrie parametre *(start, extent, fill, outline, width, style, …)*.

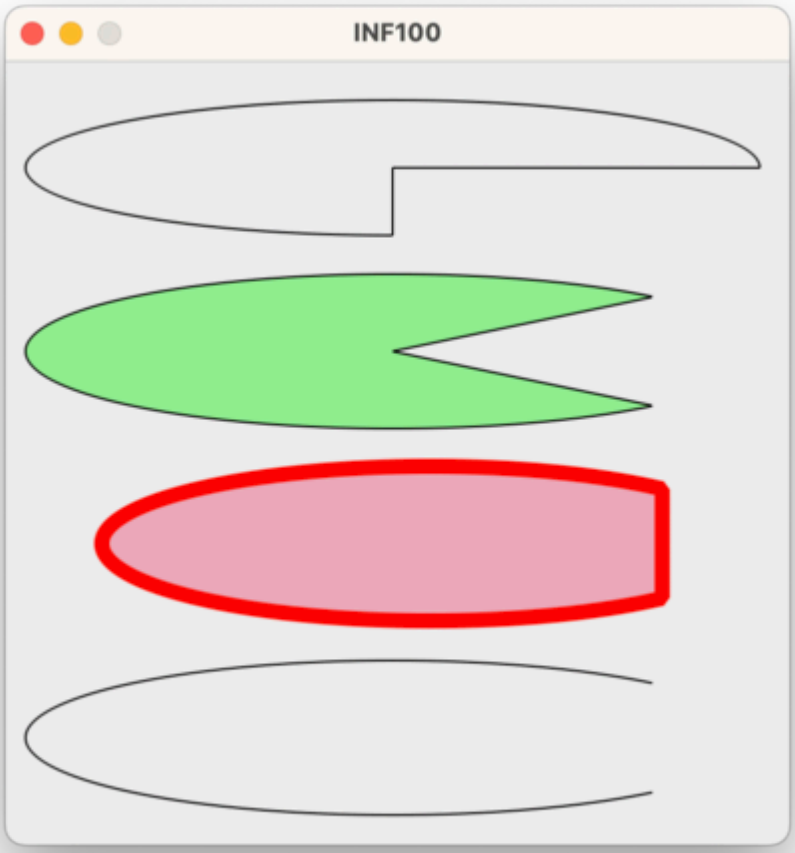
- Det er mulig å angi at buen skal starte på en annen vinkel enn 0 grader ved å angi `start` (standard er `start=0`). Verdien skal oppgis i grader.
- Det er mulig å angi hvor stor andel av ovalen buen skal dekke ved angi `extent` (standard er `extent=90`). Verdien skal oppgis i grader.
- Som standard tegnes buen med en svart strek og uten farge i midten. Dette kan endres ved å angi farge-verdier til valgfrie parametrene `fill` og `outline`. Parameteren `width` kan benyttes for å angi tykkelsen (i antall piksler) til streken som tegnes.
- Hvordan buen knyttes sammen i endepunktene kan endres ved å angi `style`. Mulige verdier er `'pieslice'` (standard), `'chord'` og `'arc'`.

↑ Innhold

```
from uib_inf100_graphics.simple import canvas, display

canvas.create_arc(10, 20, 390, 90, extent=270)
canvas.create_arc(10, 110, 390, 190, start=45, extent=270, fill='lightGreen')
canvas.create_arc(50, 210, 390, 290, start=45, extent=270, style='chord',
                  fill='#eeaabb', outline='red', width=8)
canvas.create_arc(10, 310, 390, 390, start=45, extent=270, style='arc')

display(canvas)
```



create_text

Påkrevde parametre *(x, y)*.

- For å skrive tekst, må vi oppgi koordinatene til hvor teksten skal være. Dette punktet kalles for *ankeret* til teksten.

Valgfrie parametre *(text, anchor, font, fill, angle, width, justify, …)*.

- Selve teksten som skal skrives oppgis ved å angi `text`.
- Som standard plasseres teksten slik at ankeret er i midten av teksten. Dette kan endres ved å angi `anchor`. Mulige verdier er `'n'`, `'ne'`, `'e'`, `'se'`, `'s'`, `'sw'`, `'w'`, `'nw'` og `'center'`

(standard er `'center'`). Hvis for eksempel ankeret er `'sw'`, vil teksten plasseres slik at ankerpunktet havner ved det sør-vestlige (nede til venstre) hjørnet av teksten.

- Det er mulig å angi hvilken font som skal brukes ved å angi `font`.

Mer om fonter

↑ Innhold

↑ Innhold

Det er flere gyldige formater å angi fonter på. Eksempler:

`'TkFixedFont'` er et eksempel på en *navngitt* font.

- Avhengig av hvilket operativsystem du er på, kan den samme navngitte fonten se forskjellig ut. På Windows er for eksempel 'TkFixedFont' en font som heter *Courier*, mens på Mac er den en font som heter *Monaco*.
- Disse navngitte fontene er garantert tilgjengelig: 'TkDefaultFont', 'TkTextFont', 'TkFixedFont', 'TkMenuFont', 'TkHeadingFont', 'TkCaptionFont', 'TkSmallCaptionFont', og 'TkIconFont'.
- Avhengig av operativsystem kan det finnes flere navngitte fonter. For å finne ut hvilke navngitte fonter som er tilgjengelige på ditt operativsystem, kan du kjøre følgende kode:

```
from tkinter import Tk, font

Tk().withdraw()
print(font.names())
```

- Standard font dersom ingenting er spesifisert eller den spesifiserte fonten ikke blir funnet er er 'TkDefaultFont'.

`('Times new roman', 12, 'italic bold')` er et eksempel på en *tupel* som beskriver en font.

- Denne tupelen består av tre elementer: navnet på fontfamilien, størrelsen på fonten, og en streng som beskriver hvilke *attributter* fonten skal ha.
- Mulige attributter: *italic*, *bold*, *underline* og *overstrike* (kan kombineres med mellomrom). Om du ikke ønsker noen av dem, angi en tom streng `''`.
- En font spesifisert på denne måten ser lik ut på alle operativsystemer, såfremt font-familien er installert på den aktuelle maskinen.
- Hvilke font-familier som er tilgjengelige på ulike datamaskiner varierer.
  - Disse familiene er nesten alltid tilgjengelig: 'Helvetica', 'Arial', 'Times', 'Times new roman', 'Courier' og 'Courier new'.
  - Disse familiene er ofte tilgjengelige (ikke alltid på Linux): 'Symbol', 'Verdana', 'Georgia', 'Comic Sans MS', 'Trebuchet MS', 'Arial Black', 'Impact'.
  - Andre fonter kan virke på din maskin, men ikke regn med at det virker alle andre steder.
- For å se en komplett liste av font-familier tilgjengelig på din maskin, kan du kjøre følgende kode:

```
from tkinter import Tk, font

Tk().withdraw()
print(font.families())
```

↑ Innhold

`'Arial 20'` er et eksempel på en *streng* som beskriver en font basert på fontfamilie og størrelse. Merk at fontfamilien ikke kan inneholde mellomrom, så om du ønsker å bruke en font-familie som består av flere ord, må du bruke en font-tupel som beskrevet over.

`'Arial 20 italic underline'` er et annet eksempel på en streng som beskriver en fontfamilie, størrelse og attributter. Merk at fontfamilien ikke kan inneholde mellomrom, så om du har behov for det må du bruke en font-tupel som beskrevet over.

- Farge angis med `fill`.
- Det er mulig å angi at teksten skal roteres ved å angi `angle`. Verdien skal oppgis i grader.
- For å angi maksimal bredde på et avsnitt og bryte teksten over flere linjer automatisk, kan du angi `width`. Tekst som går over flere linjer kan sidejusteres med `justify` (left/center/right).

```
from uib_inf100_graphics.simple import canvas, display

ax, ay = 200, 50
canvas.create_oval(ax - 5, ay - 5, ax + 5, ay + 5, fill='pink', outline='')
canvas.create_text(ax, ay, text='Hello, world!')

ax, ay = 200, 100
canvas.create_oval(ax - 5, ay - 5, ax + 5, ay + 5, fill='pink', outline='')
canvas.create_text(ax, ay, text='Carpe diem!', anchor='sw')

ax, ay = 200, 150
canvas.create_oval(ax - 5, ay - 5, ax + 5, ay + 5, fill='pink', outline='')
canvas.create_text(ax, ay, text='Ay caramba!', anchor='n', font='TkFixedFont')

ax, ay = 200, 200
canvas.create_text(ax, ay, text="Don't panic!", font=('Courier new', 20, ''))

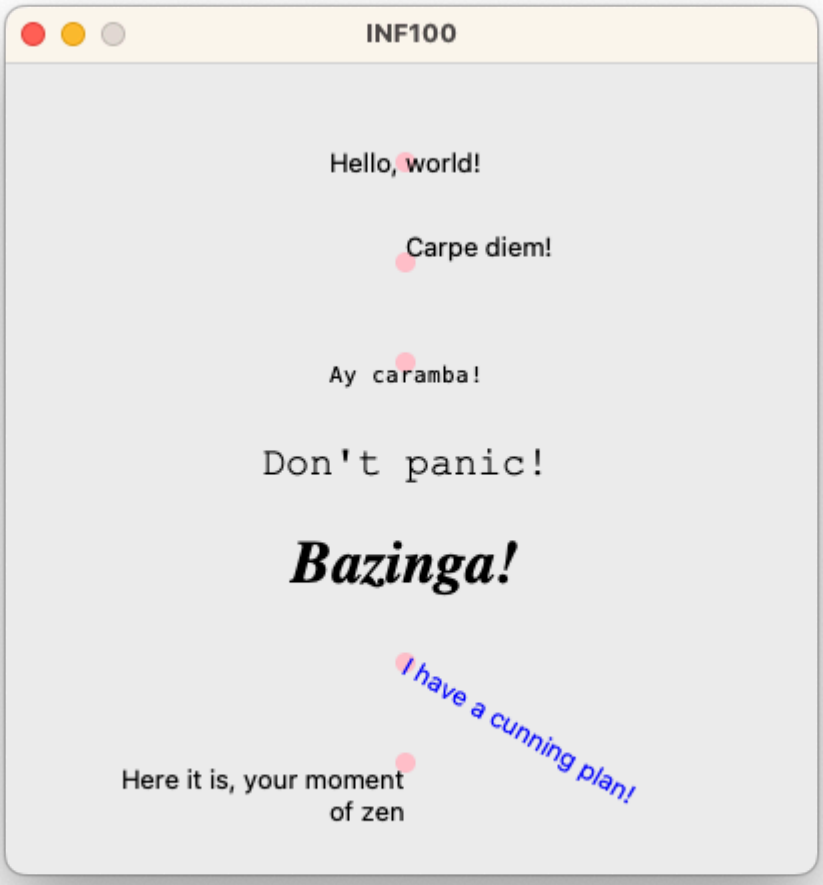
ax, ay = 200, 250
canvas.create_text(ax, ay, text='Bazinga!', font=('Times', 30, 'italic bold'))

ax, ay = 200, 300
canvas.create_oval(ax - 5, ay - 5, ax + 5, ay + 5, fill='pink', outline='')
canvas.create_text(ax, ay, text='I have a cunning plan!', fill='blue',
                   anchor='w', angle=-30)

ax, ay = 200, 350
canvas.create_oval(ax - 5, ay - 5, ax + 5, ay + 5, fill='pink', outline='')
canvas.create_text(ax, ay, text='Here it is, your moment of zen',
                   anchor='ne', justify='right', width=150)

display(canvas)
```

↑ Innhold



create_image

Påkrevde parametre *(x, y)*.

- For å tegne et bilde, må vi oppgi koordinatene til hvor bildet skal være. Dette punktet kalles for *ankeret* til bildet.

Valgfrie parametre *(pil_image, anchor, …)*.

- Bildet som skal tegnes oppgis ved å angi `pil_image`. Dette kan være et bilde som er lastet inn med hjelp av `load_image` eller `load_image_http` -funksjonen fra pakken *uib_inf100_graphics.helpers*.
- Som standard plasseres bildet slik at ankeret er i midten av bildet. Dette kan endres ved å angi `anchor`. Mulige verdier er `'n'`, `'ne'`, `'e'`, `'se'`, `'s'`, `'sw'`, `'w'`, `'nw'` og `'center'` (standard er 'center'). Hvis for eksempel ankeret er 'sw', vil bildet plasseres slik at ankerpunktet havner ved det sør-vestlige (nede til venstre) hjørnet av bildet.

↑ Innhold

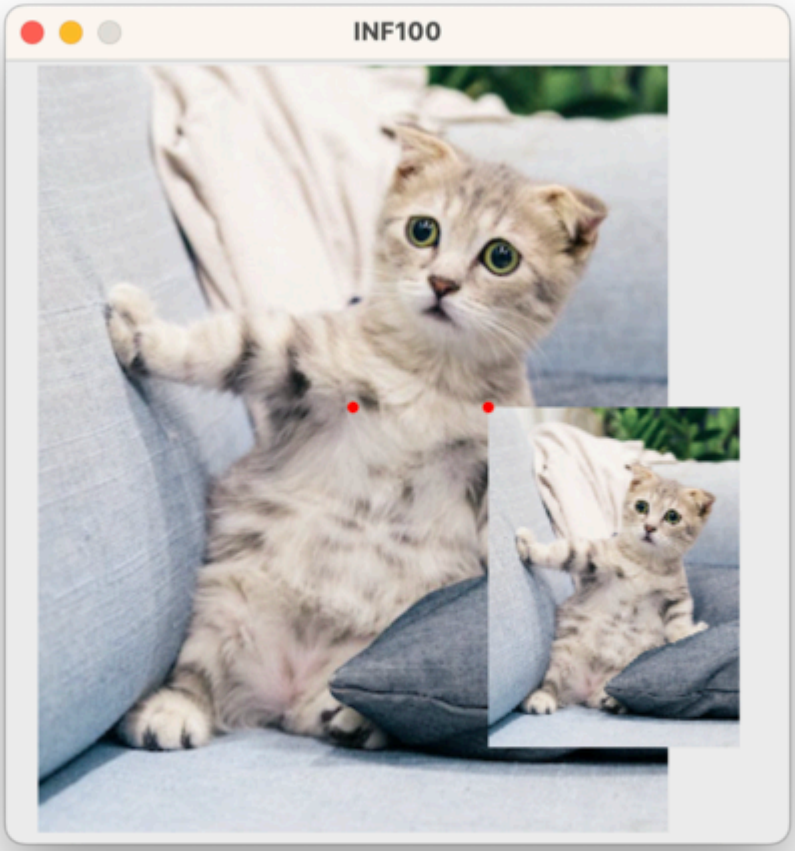
```
from uib_inf100_graphics.simple import canvas, display
from uib_inf100_graphics.helpers import load_image_http, scaled_image

# Image credits: unsplash.com/@tranmautritam
image = load_image_http('https://tinyurl.com/inf100kitten-png')

canvas.create_image(180, 180, pil_image=image)
canvas.create_oval(180 - 3, 180 - 3, 180 + 3, 180 + 3, fill='red', outline='')

smaller_image = scaled_image(image, 0.4)
canvas.create_image(250, 180, pil_image=smaller_image, anchor='nw')
canvas.create_oval(250 - 3, 180 - 3, 250 + 3, 180 + 3, fill='red', outline='')

display(canvas)
```



Farger

Et par farger er innebygget, som demonstrert i eksemplene over: `'black'` `'white'` `'gray'` `'red'` `'green'` `'blue'` `'lightGreen'` `'rosyBrown'`, samt en hel del andre spenstige farger som du finner i dokumentasjonen til tkinter (https://www.tcl.tk/man/tcl/TkCmd/colors.htm). Vi er imidlertid ikke begrenset til kun disse fargene.

🕮 Bakgrunn: piksler og farger som RGB

→ Innhold

↑ Innhold

Piksel

I en LED-skjerm (som er en vanlig dataskjerm) tegnes bildet på skjermen ved at hver enkelt *piksel* (liten prikk på skjermen) får en bestemt farge. Inne i selve skjermen sitter det tre lamper inne i hver piksel: en rød lampe, en grønn lampe og en blå lampe. Når alle tre lampene lyser med maksimal intensitet, ser vi hvitt lys komme ut av pikselen. Dersom ingen av lampene lyser, er pikselen svart. Alle fargene skjermen kan produsere, blir laget av en kombinasjon av lysintensiteter i de tre pikslene.

Hvis man zoomer inn svært tett på dataskjermen, kan man skimte at en hvit piksel ikke faktisk er helt hvit, men består egentlig av en rød, grønn og blå lampe ved siden av hverandre som lyser. Her er et bilde jeg har tatt av musenpekeren min på skjermen:



Om vi zoomer litt inn på bildet, kan vi skimte at hver piksel består av tre lamper: en rød, en grønn og en blå.



Fordi menneskets øye bare er i stand til å registrere lyssignaler på rød, grønn og blå frekvens, vil en blanding av røde, grønne og blå lyssignaler være tilstrekkelig for å simulere alle oppfattelser av farge et menneskeøye kan gi. Når menneskeøyet får utslag på alle tre fargekanalene, vil vi oppfatte det som hvitt lys; selv om det egentlig bare er en blanding av rødt, grønt og blått lys, og strengt tatt ikke er en blanding av alle mulige slags lysfrekvenser («ekte» hvitt lys).

↑ Innhold

Når man kjøper en LED-skjerm på butikken, finnes det ulike *fargedypder* eller man får oppgitt *antall farger* skjermen kan vise. Denne spesifikasjonen bestemmes av i hvor mange «trinn» man kan justere intensiteten til hver av de fargede lampene i en piksel. Det har lenge vært vanlig at man bruker 256 slike trinn. En farge i dette systemet kan derfor sees på som tre tall (r, g, b), der hver av r, g og b er et tall mellom 0 og 255.

Selv om nyere og dyre skjermer teknisk sett kan ha flere trinn, bruker som regel software som ikke er rettet spesielt mot high-end bildebehandling fremdeles dette systemet som standard.

Alle farger har en *RGB*-verdi. I tabellen ser vi at hver farge har en gitt styrke av rød (R), grønn (G) og blå (B), som er et tall mellom 0 og 255. Denne RGB-verdien kan også skrives i heksadesimalt format (se kolonnen *Hex*), hvor de to første tegnene etter hashtag reprsenterer styrken på rød, de to neste representerer grønn, og de to siste representerer blå sin styrke.

Farge	R	G	B	Hex	Kallenavn
	0	0	0	`#000000`	`black`
	255	0	0	`#ff0000`	`red`
	0	255	0	`#00ff00`	`green1`
	0	0	255	`#0000ff`	`blue`
	255	255	0	`#ffff00`	`yellow`
	0	255	255	`#00ffff`	`cyan`
	255	0	255	`#ff00ff`	`magenta`
	255	255	255	`#ffffff`	`white`
	128	128	128	`#808080`	`gray`
	128	0	0	`#800000`	`maroon`
	255	140	0	`#ff8c00`	`dark orange`
	224	227	206	`#e0e3ce`	-
	248	249	245	`#f8f9f5`	-

For flere farger, se for eksempel listen over farger (A-F) (https://en.wikipedia.org/wiki/List_of_colors:_A%E2%80%93F) på Wikipedia, eller prøv RGB-kalkulatoren til w3schools.com (https://www.w3schools.com/colors/colors_rgb.asp).

Vårt rammeverk for grafikk kan tolke alle RGB-verdier skrevet i hex-format.

↑ Innhold

Tekst i boks

For enkelhets skyld har vi laget en hjelpefunksjon som kan brukes til å tegne tekst midt i et rektangel, og som også gjør teksten så stor som mulig innenfor rektangelet. Denne funksjonen heter `text_in_box` og må importeres fra pakken *uib_inf100_graphics.helpers*.

Påkrevde parametre *(canvas, x1, y1, x2, y2, text)*.

- `canvas` er lerretet som skal tegnes på.
- `x1, y1` er koordinatene til det øverste venstre hjørnet i rektangelet.
- `x2, y2` er koordinatene til det nederste høyre hjørnet i rektangelet.
- `text` er strengen som skal skrives.

Valgfrie parametre *(font, fit_mode, padding, min_font_size, justify, align, fill, …)*.

- `font` er fonten som skal benyttes. Merk at størrelsen på fonten vil bli ignorert, men må likevel spesifiseres. Les mer om fonter i avsnittet om create_text over.
- `fit_mode` er en streng som angir hvordan teksten skal tilpasses rektangelet. Mulige verdier er `'contain'` (standard), `'fill'`, `'height'` og `'width'`.
- `padding` er minimum antall piksler som skal være mellom teksten og kanten av rektangelet (standard 0).
- `min_font_size` er minimum størrelse på fonten som skal brukes. Dersom teksten ikke får plass med denne fontstørrelsen, vil teksten gå utenfor området sitt (standard 1).
- `justify` er en streng som angir hvordan teksten skal justeres horisontalt innenfor rektangelet. Mulige verdier er `'left'`, `'center'` (standard) og `'right'`.
- `align` er en streng som angir hvordan teksten skal justeres vertikalt innenfor rektangelet. Mulige verdier er `'top'`, `'center'` (standard) og `'bottom'`.
- `fill` er fargen teksten skal ha.

↑ Innhold

```python
from uib_inf100_graphics.simple import canvas, display
from uib_inf100_graphics.helpers import text_in_box

text = 'Hello, world!'

# First example
canvas.create_rectangle(100, 20, 300, 70)
text_in_box(canvas, 100, 20, 300, 70, text)

# Named font and padding.
canvas.create_rectangle(100, 120, 300, 170)
text_in_box(canvas, 100, 120, 300, 170, text,
            font='TkFixedFont',
            justify='left',
            padding=15)

# System installed font family name, fill color and fit_mode.
canvas.create_rectangle(100, 200, 300, 270)
text_in_box(canvas, 100, 200, 300, 270, text,
            font=('Times new roman', 1, ''),
            fit_mode='height', # fit_mode='height' ignores width of rectangle
            fill='blue')

# Multiline text, font style, justification.
multiline_text = text+'\n'+text+' '+text+'\n'+text
canvas.create_rectangle(100, 320, 300, 370)
text_in_box(canvas, 100, 320, 300, 370, multiline_text,
            font='Arial 42 bold italic overstrike underline',
            justify='right', # justify is 'left', 'center' or 'right'
            padding=5)

display(canvas)
```

↑ Innhold



Bilde i boks

For enkelhets skyld har vi laget en hjelpefunksjon som kan brukes til å tegne et bilde midt i et rektangel, og som også skalerer bildet slik at det passer innenfor rektangelet. Denne funksjonen heter `image_in_box` og må importeres fra pakken *uib_inf100_graphics.helpers*.

Påkrevde parametre *(canvas, x1, y1, x2, y2, pil_image)*.

- `canvas` er lerretet som skal tegnes på.
- `x1, y1` er koordinatene til det øverste venstre hjørnet i rektangelet..
- `x2, y2` er koordinatene til det nederste høyre hjørnet i rektangelet
- `pil_image` er bildet som skal tegnes. Dette kan være et bilde som er lastet inn med hjelp av `load_image` eller `load_image_http` -funksjonen fra pakken *uib_inf100_graphics.helpers*.

Valgfrie parametre *(fit_mode, antialias)*.

- `fit_mode` er en streng som angir hvordan bildet skal tilpasses rektangelet. Mulige verdier er `'contain'` (standard), `'fill'`, `'crop'` og `'stretch'`.
- `antialias` er en boolsk verdi som angir om bildet skal antialiaseres (standard `True`). Antialiasering er en teknikk som gjør at bildet ser skarpere ut når det skaleres ned, men bruker mer tid/prosessorkraft.

```python
from uib_inf100_graphics.simple import canvas, display
from uib_inf100_graphics.helpers import load_image_http, image_in_box

# Image credits: unsplash.com/@tranmautritam
image = load_image_http('https://tinyurl.com/inf100kitten-png')

image_in_box(canvas, 20, 40, 180, 180, image)
canvas.create_rectangle(20, 40, 180, 180, outline='red', width=2)
canvas.create_text(100, 35, text="fit_mode='contain'", anchor='s')

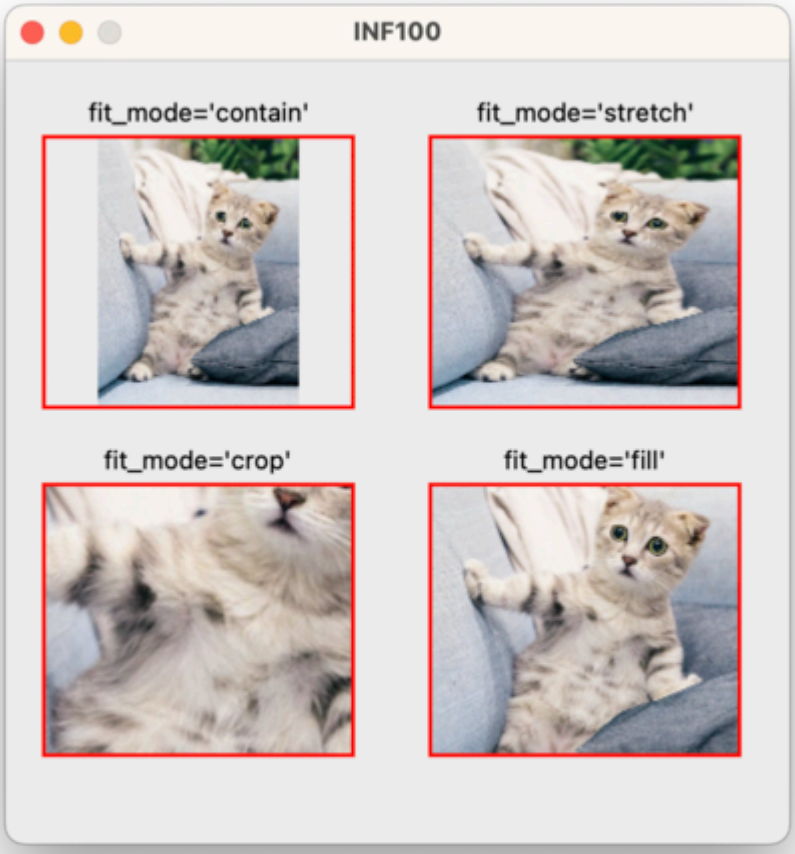
image_in_box(canvas, 20, 220, 180, 360, image, fit_mode='crop')
canvas.create_rectangle(20, 220, 180, 360, outline='red', width=2)
canvas.create_text(100, 215, text="fit_mode='crop'", anchor='s')

image_in_box(canvas, 220, 40, 380, 180, image, fit_mode='stretch')
canvas.create_rectangle(220, 40, 380, 180, outline='red', width=2)
canvas.create_text(300, 35, text="fit_mode='stretch'", anchor='s')

image_in_box(canvas, 220, 220, 380, 360, image, fit_mode='fill')
canvas.create_rectangle(220, 220, 380, 360, outline='red', width=2)
canvas.create_text(300, 215, text="fit_mode='fill'", anchor='s')

display(canvas)
```

↑ Innhold



Eksempel: buss

I videoen vises litt av tankeprosessen for å tegne en enkel buss. Her er koden som blir skrevet:

```python
from uib_inf100_graphics.simple import canvas, display

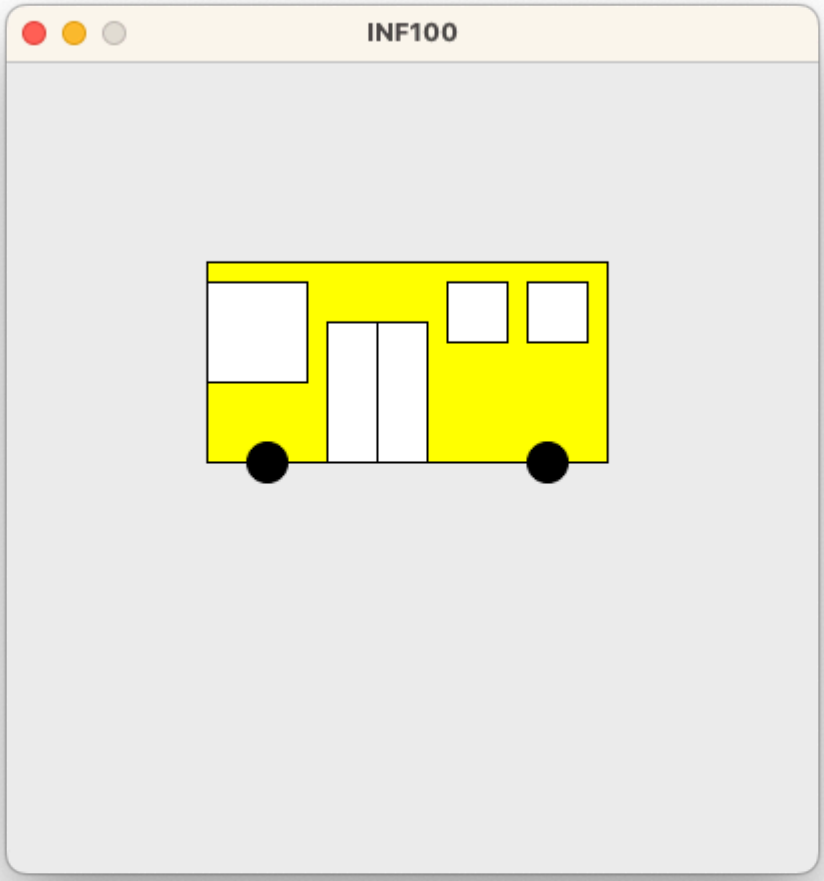
# Body
canvas.create_rectangle(100, 100, 300, 200, fill='yellow')

# Windows
canvas.create_rectangle(100, 110, 150, 160, fill='white')
canvas.create_rectangle(260, 110, 290, 140, fill='white')
canvas.create_rectangle(220, 110, 250, 140, fill='white')

# Door
canvas.create_rectangle(160, 130, 210, 200, fill='white')
canvas.create_line(185, 130, 185, 200)

# Wheels
canvas.create_oval(120, 190, 140, 210, fill='black')
canvas.create_oval(260, 190, 280, 210, fill='black')

display(canvas)
```



Løkker

↑ Innhold

- While-løkker
- Uendelig løkke
- Break
- Continue
- For-løkker over range
- Nøstede løkker
- Stil

> ***Hjelp, programmet mitt avsluttes ikke!*** *For å avbryte en evig løkke, trykk* `ctrl + c` *når fokuset er på terminalen hvor koden kjøres.*

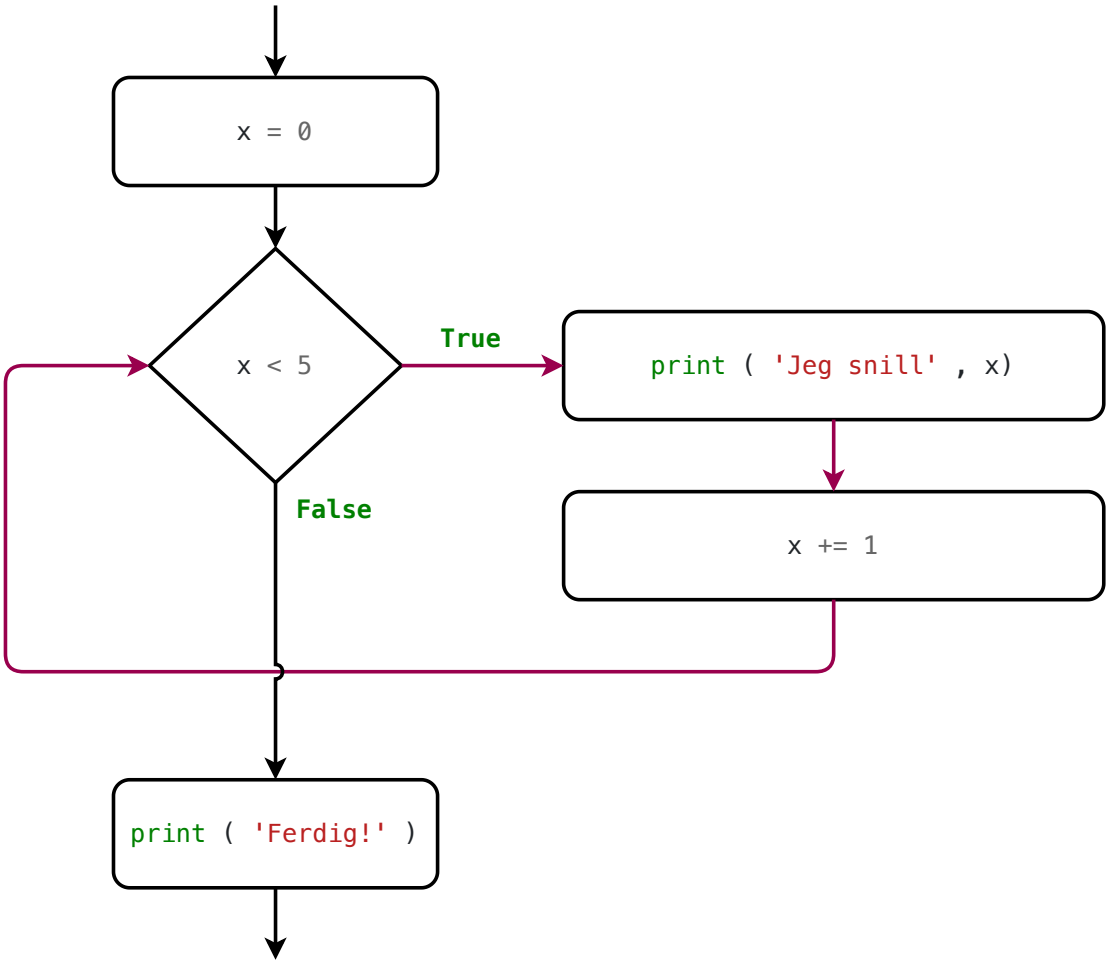
While-løkker

For å utføre en blokk med kode flere ganger, kan man benytte en while-løkke. Koden inne i løkken utføres så lenge betingelsen evaluerer til True.

```
x = 0
while x < 5:
    print('Jeg snill', x)
    x += 1

print('Ferdig!')
```

↑ Innhold



En *iterasjon* er en gjennomkjøring av kodeblokken inni en løkke. I eksempelet over utføres koden fem ganger, og vi sier at løkken har fem iterasjoner.

While-løkker er spesielt godt egnet for tilfeller der vi ønsker å gjenta en sekvens med operasjoner, men vi vet ikke på forhånd hvor mange ganger. For eksempel kan vi ønske å finne det minste heltallet n slik at n^2 er større enn 1000. Ved hjelp av en løkke kan vi prøve alle mulige positive heltall helt til vi finner det vi leter etter:

```
n = 1
while n * n <= 1000:
    n += 1
print(n, 'er laveste heltall slik at n^2 er større enn 1000')
```

Et irriterende program.[1]

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input()
print('Thank you!')
```

Et program for å telle antall siffer i et tall.

↑ Innhold

```
x = 222222

part_of_x = abs(x)
count = 0

while part_of_x > 0:
    count += 1
    part_of_x = part_of_x // 10

print(f'Det er {count} siffer i {x}.')
```

Uendelig løkke

Når man skriver en while-løkke, kan man risikere at løkken varer evig dersom betingelsen alltid blir tilfredsstilt.

```
x = 1
while x < 10:
    print(x)
    x + 1     # Glemt tilordning; variabelen (x) endres ikke
```

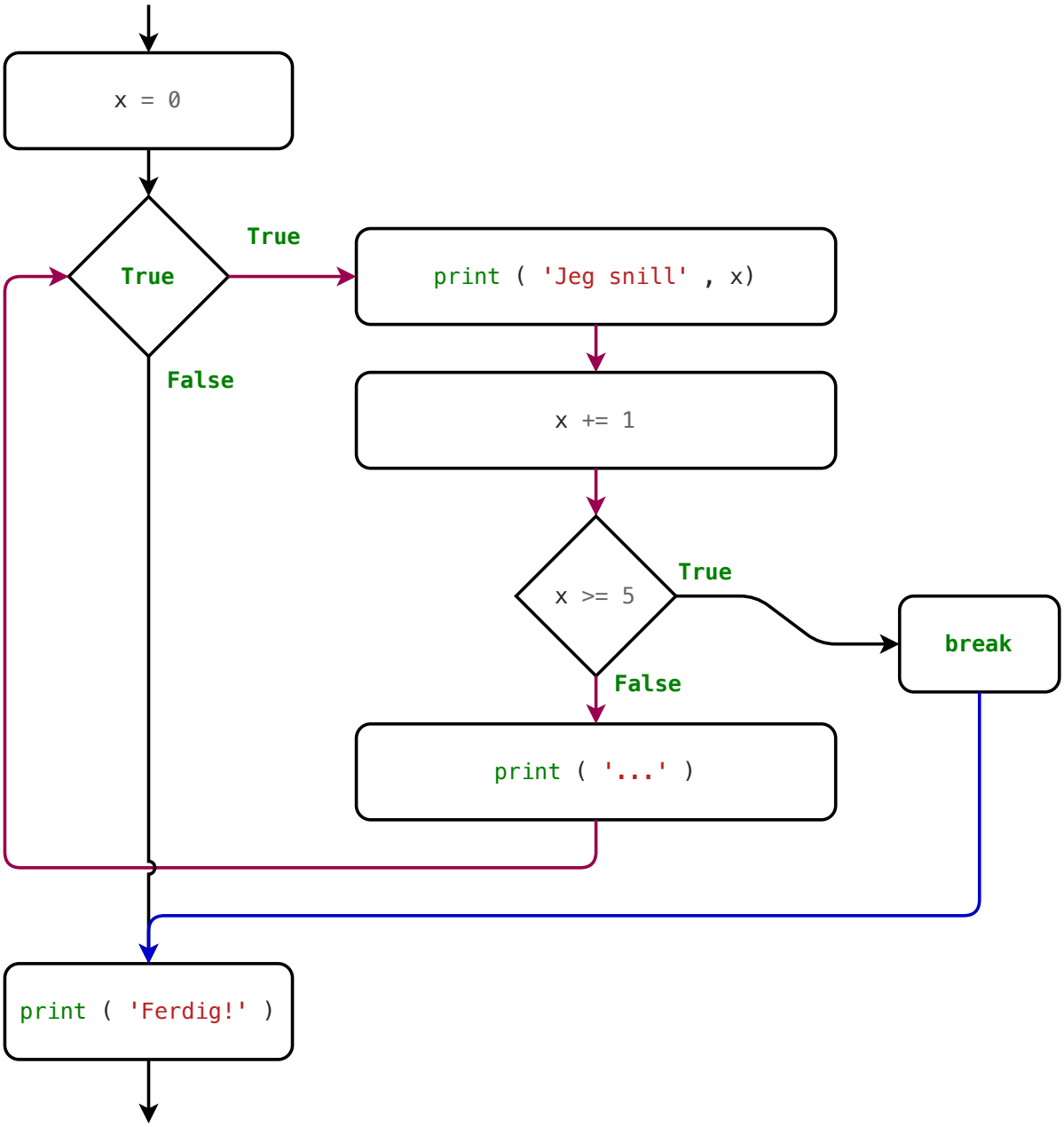
For å avbryte en evig løkke, trykk `ctrl + c` når fokuset er på terminalen hvor koden kjøres.

Break

Break kan benyttes for å bryte ut av en løkke umiddelbart.

```
x = 0
while True:
    print('Jeg snill', x)
    x += 1
    if x >= 5:
        break
    print('...') # kjøres ikke siste iterasjon
print('Ferdig!')
```

↑ Innhold



Et irriterende program (som tidligere), skrevet med `break` i stedet for en betingelse:

```
while True:
    print('Please type your name.')
    name = input()
    if name == 'your name':
        break
print('Thank you!')
```

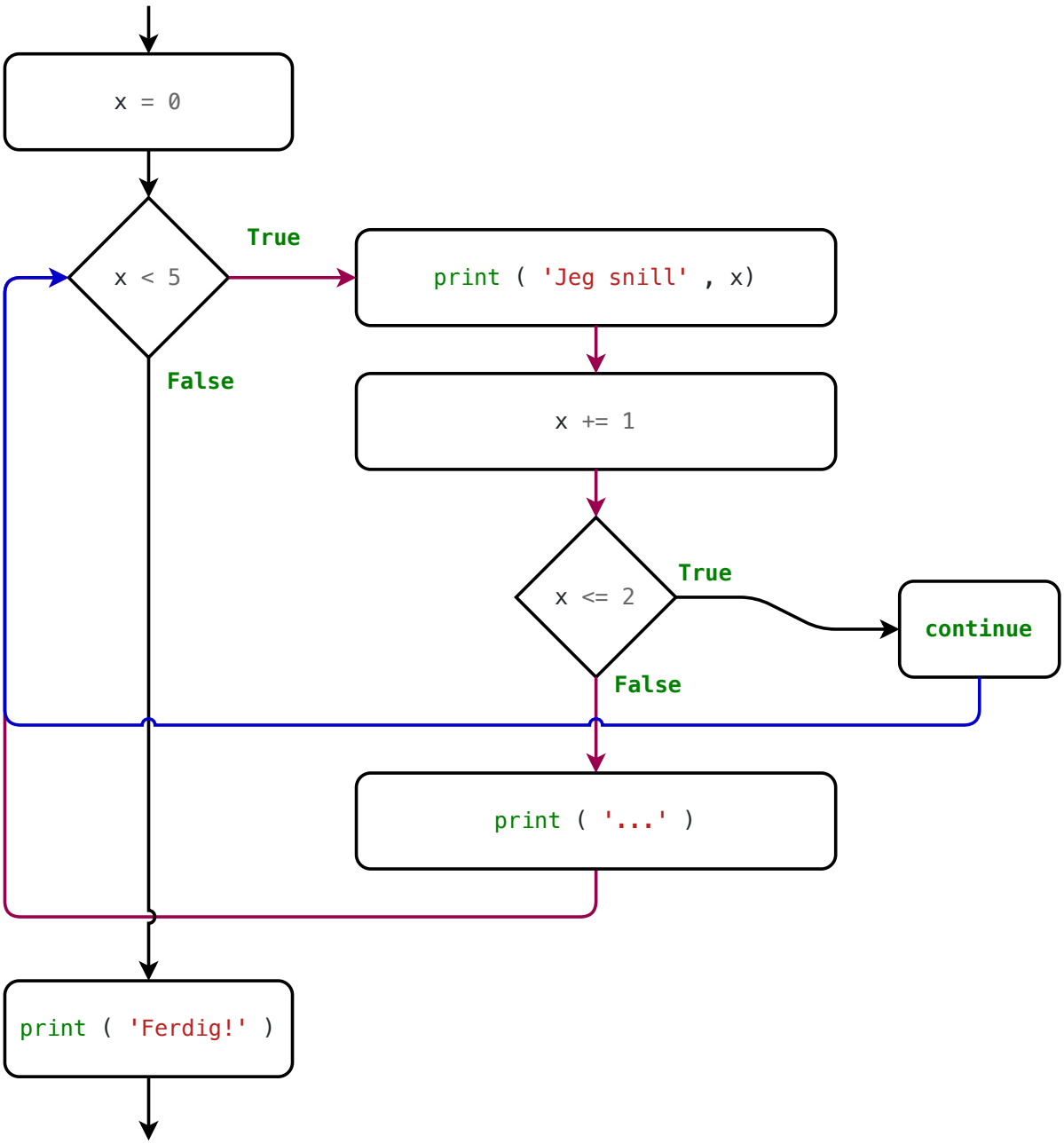
Mange nybegynnere syntes det er lettere å forstå kombinasjonen av `while True` og `break` (som vist over) enn andre typer løkker. Denne måten å skrive løkker på er svært fleksibel, og er en fin strategi for å bli vant til å skrive løkker. En while-løkke med en betingelse er imidlertid mer kompakt, og kommuniserer ofte tydeligere hvilke omstendigheter som gjør at løkken kjøres eller termineres.

↑ Innhold

Continue

Continue benyttes for å hoppe over resten av kodeblokken i løkken og gå direkte tilbake til begynnelsen (neste steg blir å sjekke betingelsen på nytt).

```
x = 0
while x < 5:
    print('Jeg snill', x)
    x += 1
    if x <= 2:
        continue
    print('...') # hoppes over når x <= 2
print('Ferdig!')
```



Eksempel med både `break` og `continue`:

↑ Innhold

```python
while True:
    print("Brukernavn: ", end="")
    username = input()

    if username == "":
        break # Avbryter løkken

    if username != "admin":
        print(f"Fant ikke bruker {username}")
        continue # Avbryter resten av iterasjonen

    print(f"Passord for {username}: ", end="")
    password = input()
    if password == "42":
        print("Du er nå logget inn")
        print("Bla bla bla")
        print("Du er nå logget ut igjen")

print("Slår av maskinen nå.")
```

For-løkker over range

Det er ofte at vi vet før løkken starter hvor mange ganger vi ønsker at løkken skal kjøres (f. eks. om vi har en variabel som inneholder dette). Da bør vi benytte en *for*-løkke over en *range* – også kalt en «vanlig» for-løkke. En for-løkke er kortere å skrive enn en while-løkke, og reduserer faren for feil og bugs.

For-løkke over range:

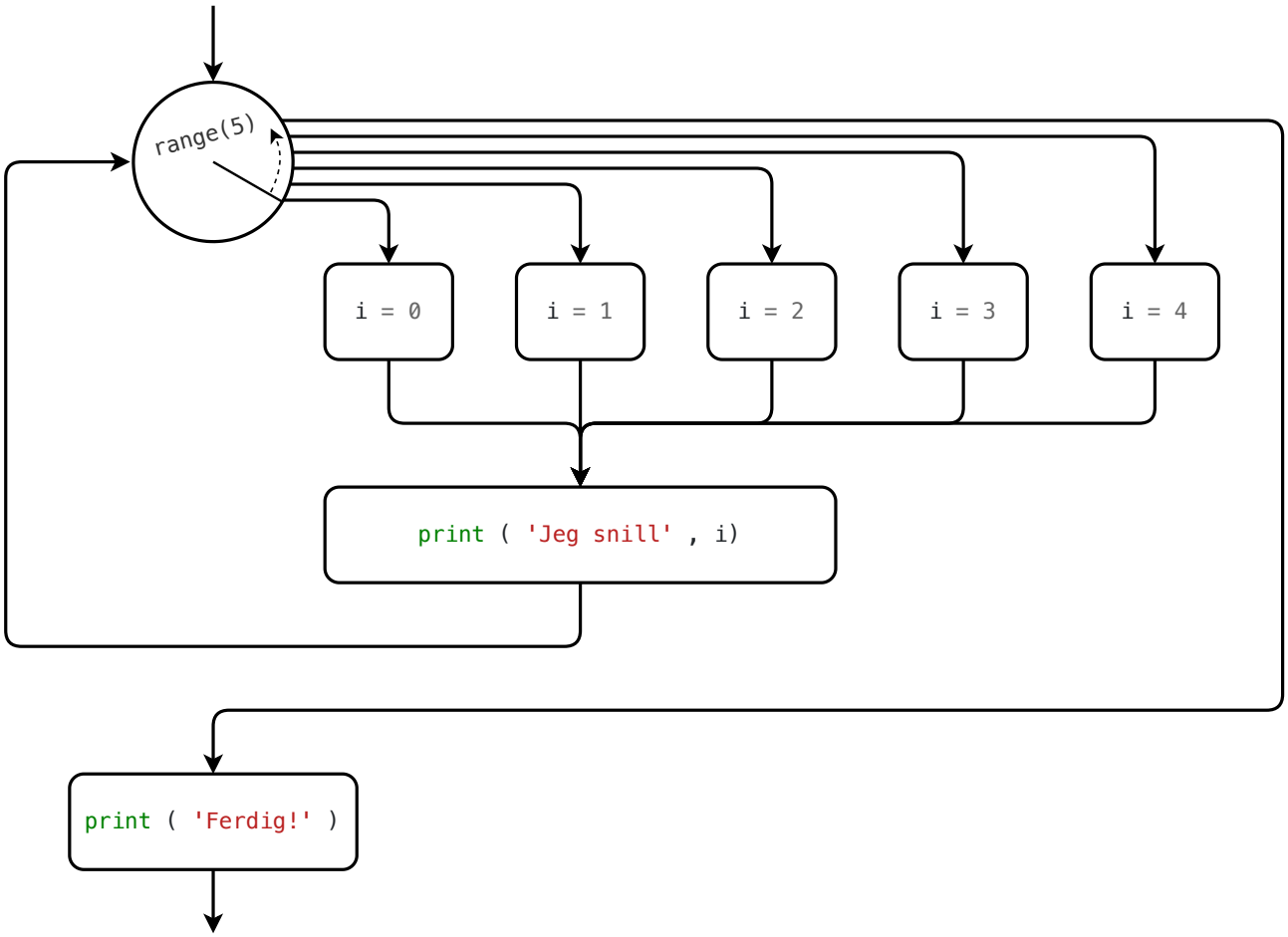
```python
for i in range(5):
    print('Jeg snill', i)


print('Ferdig!')
```

Til sammenligning: while-løkke.

```python
i = 0
while i < 5:
    print('Jeg snill', i)
    i += 1
print('Ferdig!')
```

↑ Innhold



*Legg merke til at vi i for-løkken ikke trenger å opprette en variabel `i` på en egen linje før løkken starter. Variabelen `i` blir i stedet automatisk opprettet av for-løkken. Denne variabelen kalles en **iterand** (eller iterasjonsvariabel). Vi kan gi iteranden hvilket navn vi selv ønsker (etter samme regler som andre variabelnavn).*

I hver iterasjon av løkken får iteranden en ny verdi hentet fra range-objektet. I eksempelet over:

- *første gang løkken kjøres, får iteranden `i` verdien 0,*
- *andre gang løkken kjøres, får iteranden `i` verdien 1,*
- *tredje gang løkken kjøres, får iteranden `i` verdien 2,*
- *og så videre.*

Den femte og siste gangen løkken kjøres i eksempelet over, får iteranden `i` verdien 4. Etter dette avsluttes løkken, fordi iteranden har nådd slutten av range-objektet.

En **range** i Python tilsvarer det vi i matematikken kaller en *endelig aritmetisk følge*: altså en endelig sekvens med tall som starter på et gitt tall, og øker med en fast verdi for hvert ledd. For eksempel er $2, 5, 8, 11, 14, 17$ en endelig aritmetisk følge med startverdi 2 og økning 3. I Python kan vi skrive denne følgen som `range(2, 20, 3)`.

- Det første argumentet (`2`) er startverdien (inklusiv),
- det andre argumentet (`20`) er sluttverdien (eksklusiv), og
- det tredje argumentet (`3`) er den faste økningen.

↑ Innhold

Med for-løkke over range:

```
for x in range(2, 20, 3):
    print(x)
```

Til sammenligning: while-løkke.

```
x = 2
while x < 20:
    print(x)
    x += 3
```

En `range` kan opprettes med både ett, to og tre argumenter.

- *Ett argument: startverdi er 0, sluttverdi er det gitte argumentet, og økningen er 1. For eksempel er `range(4)` det samme som `range(0, 4, 1)`, og gir følgen* $0, 1, 2, 3$.
- *To argumenter: startverdi er det første argumentet, sluttverdi er det andre argumentet, og økningen er 1. For eksempel er `range(2, 5)` det samme som `range(2, 5, 1)`, og gir følgen* $2, 3, 4$.
- *Tre argumenter: startverdi er det første argumentet, sluttverdi er det andre argumentet, og økningen er det tredje argumentet. For eksempel er `range(2, 20, 3)` en følge med startverdi 2, sluttverdi 20, og økning 3. Denne følgen er* $2, 5, 8, 11, 14, 17$.

Eksempel: skriv ut alle tallene mellom a og b (inklusiv både a og b).

Med for-løkke over range:

```
a = 3
b = 10

for i in range(a, b + 1):
    print(i)
```

Til sammenligning: while-løkke.

```
a = 3
b = 10

i = a
while i <= b:
    print(i)
    i += 1
```

Eksempel: negativ økning.

```
# Hopp kan også være negativ.
# Eksempel: 10 9 8 7 6
for i in range(10, 5, -1):
    print(i, end=" ")
print()
```

Eksempel: en range kan være tom

↑ Innhold

```
for i in range(0):
    print(i, end=" ")
print("---")

for i in range(-4):
    print(i, end=" ")
print("---")

for i in range(3, 3):
    print(i, end=" ")
print("---")

for i in range(19, 3):
    print(i, end=" ")
print("---")

for i in range(5, 10, -1):
    print(i, end=" ")
print("---")
```

En range er en *samling* med tall. En streng er en samling med bokstaver. En for-løkke kan brukes for å iterere over alle typer samlinger – altså både strenger, ranges, lister eller en annen type samling.

```
s = "foo"
for letter in s:
    print(letter)
print()

for x in ["en", "liste", "med", "strenger"]:
    print(x)
print()
```

Nøstede løkker

```
# Vi kan ha løkker inni løkker
rows = 3
cols = 5

for row in range(rows):
    for col in range(cols):
        print(f"({row}, {col})", end=" ")
    print()
```

↑ Innhold

```
# Hvilken figur tegner vi her?
height = 5

for row in range(height):
    for col in range(row):
        print("*", end="")
    print()
```

Stil

Benytt alltid en for-løkke hvis det er naturlig. Dette gjør det lettere å forstå koden, og er mindre utsatt for bugs som gjør at programmet blir sittende fast i en uendelig løkke.

```
# Dårlig
repetitions = 5
x = 0
while x < repetitions:
    print("Jeg skal være snill", x)
    x += 1
```

```
# Bra
repetitions = 5
for x in range(repetitions):
    print("Jeg skal være snill", x)
```

---

1. Fra Automate the Boring Stuff with Python (https://automatetheboringstuff.com/2e/chapter2/#calibre_link-1717). Al Sweigart, CC BY-NC-SA 3.0. ↩

Strenger

↑ Innhold

- Basics
- Fire måter å skrive strenger
- Linjeskift og escape-sekvenser
- Konvertering til strenger
- Konvertering og formatering med f-strenger
- Operasjoner og metoder
  - Grunleggende operasjoner
  - Metoder
  - Split og join
  - Søking i strenger
- Indeksering og beskjæring
- Løkker over strenger
- Palindromer
- Representasjon i minnet
- Lese og skrive til fil
  - Hjelp, filen blir ikke funnet

Basics

Se notatene fra kom i gang om strenger.

Fire måter å skrive strenger

```
# I kildekoden kan streng-verdier oppgis på fire ulike måter
print('apostrof')
print("hermetegn")
print('''trippel-apostrof''')
print("""trippel-hermetegn""")

# Hvilken variant som brukes har absolutt ingenting å si
print('foo' == "foo") # True

# Så hvorfor ha flere varianter?
# Svar 1: kompatibilitet
# Svar 2: for å enklere skrive hermetegn og apostrof
print("Her er 'apostrof'")
print('Her er "hermetegn"')
print("""Her er både "hermetegn" og 'apostrofer'""")
```

```
print("Hvis vi kun bruker "hermetegn" går det galt")
```

↑ Innhold

Linjeskift og escape-sekvenser

```
# Et tegn med en bakstrek foran seg, som \n, er en escape-sekvens.
# Selv om det ser ut som to tegn, er det bare ett tegn når Python er
# ferdig med å lese kildekoden. I tilfellet \n er dette et linjeskift.

# Merk at de to setningene under gjør det samme
print("abc\ndef")  # \n er ett enkelt linjeskift
print("""abc
def""") # Trippel-hermetegn/apostrof tillater linjeskift uten escape-sekvens

print('''\
Du kan bruke bakstrek på slutten av en linje for å ekskludere
et påfølgende linjeskiftet i kildekoden. Dette er svært sjeldent
brukt, men et anvendelsesområde er som i dette eksempelet, på
starten av en lengre streng over flere linjer. På den måten kan
hele strengen bli skrevet inn med samme innrykk (inkludert første
linje).
''')
```

Flere escape-sekvenser:

```
print("Hermetegn i hermetegn-streng: \"")
print("Bakstrek: \\")
print("Linjeskift: [\n]")
print("Tab: [\t]")
print()

print("Denne teksten er skilt av tab'er, 3 per linje:")
print("abc\tdef\tg\nhi\tj\\\tk\n---")
print()

# En escape-sekvens telles som ett tegn
s = "a\\b\"c\td"
print("s =", s)
print("len(s) =", len(s))
```

Konvertering til strenger

Verdier som ikke er strenger kan konverteres til en streng-representason med bruk av funksjonene `str` og `repr`. Kjør programmet under for å se forskjellen:

↑ Innhold

```
def print_string_conversion(x):
    print('klasse:', type(x))
    print('   str:', str(x))
    print('  repr:', repr(x))
    print()

print('Vanligvis konverteres verdier til streng med str-funksjonen')
print('Å bruke repr-funksjonen gir for mange vanlige klasser samme resultat')
print_string_conversion(10)
print_string_conversion(True)
print_string_conversion(2/11)

print('Men for strenger, viser repr oss whitespace og escape-sekvenser.')
print_string_conversion('Litt tekst')
print_string_conversion('Tekst med spesielle symobler, som \t og \\  ')

# Generelt vil `repr` vise mer detaljert informasjon enn `str`.
# Hensikten med `str` er at resultatet skal være leselig, hensikten
# med `repr` er å gi oss presis informasjon.

print('For andre typer kan forskjellen være stor')
import datetime
today = datetime.datetime.now()
print_string_conversion(today)
```

Generelt vil `repr` vise mer detaljert informasjon enn `str`. Hensikten med `str` er at resultatet skal være leselig på en pen måte, mens hensikten med `repr` er å være presis.

Konvertering og formatering med f-strenger

F-strenger lar oss konvertere til en streng i kontekst av en større streng. Man angir en f-streng ved å sette bokstaven `f` foran hermetegnet som angir at den større kontekst-strengen begynner; deretter kan vi angi hvilke variabler/uttrykk vi vil konvertere til streng ved å bruke `{}` inne i kontekst-strengen. Vises best med et eksempel:

```
name = 'Eva'
age = 23
print(f'{name} er {age} år gammel') # Eva er 23 år gammel
```

F-strenger kan også brukes til å formatere verdier – for eksempel kan man spesifisere hvor lang strengen skal være, eller hvor mange desimaler som skal vises for et flyttall. Formatering spesifiseres ved å legge til et kolon `:` og en formaterings-spesifikasjon etter selve variabelnavnet/uttrykket inne i krøllparentesene `{}`. Her er noen eksempler:

Minimum bredde

- Spesifiserer `5` for å konvertere til streng med minst fem tegn, `10` for å konvertere til streng med minst ti tegn, osv.

↑ Innhold

- Hvis strengen er kortere enn spesifikasjonen, vil den ledige plassen fylles med mellomrom.
- Tall er høyrejustert som standard, strenger er venstrejustert. Dette kan endres ved å legge til `>`, `<` eller `^` før bredden.

```
x = 10
s = 'abc'
pi = 3.141592653589793

print('Standard-justert')
print(f'** {x:10} **')   # **         10 **
print(f'** {s:10} **')   # ** abc        **
print(f'** {pi:10} **')  # ** 3.141592653589793 ** (pi er mer enn 10 tegn)
print()

print('Høyrejustert')
print(f'** {x:>10} **')  # **         10 **
print(f'** {s:>10} **')  # **        abc **
print()

print('Venstrejustert')
print(f'** {x:<10} **')  # ** 10         **
print(f'** {s:<10} **')  # ** abc        **
print()

print('Sentrert')
print(f'** {x:^10} **')  # **     10     **
print(f'** {s:^10} **')  # **    abc     **
print()

print('Fyll med nuller')
print(f'** {x:010} **')  # ** 0000000010 **
```

Antall desimaler i et flyttall

- Spesifiseres med `.2f` for å vise to desimaler, `.3f` for å vise tre desimaler, osv.
- Kan kombineres med minimum bredde, f. eks. `{pi:10.3f}` for å vise pi med tre desimaler og minimum bredde 10.

```
x = 10
pi = 3.141592653589793

print('Nøyaktig 3 desimaler (.3f)')
print(f'x er ca {x:.3f}')  # x er ca 10.000
print(f'pi er ca {pi:.3f}')  # pi er ca 3.142
print()

print('Minimum bredde 10 og 3 desimaler (10.3f)')
print(f'{"x":3} er ca {x:10.3f}')    # x   er ca     10.000
print(f'{"pi":3} er ca {pi:10.3f}')  # pi  er ca      3.142
```

Snarvei for å vise både uttrykk og evaluering

Der er relativt vanlig å ønske å se verdien av en variabel eller et uttrykk samtidig som man ønsker å skrive ut hvilket uttrykk/variabel det faktisk er snakk om. Til dette har f-strenger en snarvei ved å avslutte uttrykket med `=`.

```
# Noen variabler
x = 10
y = 42

# Uten bruk av snarvei! (ulempe: vi gjentar samme uttrykk flere ganger; da er
# det fort gjort å endre kun ett av dem senere, som kan føre til logiske feil)
print('x =', x)         # x =  10
print('x + y =', x + y) # x + y =  52

# BRA! (vi bruker f-strenger til å skrive ut både uttrykk og evalueringsverdi)
print(f'{x = }')         # x = 10
print(f'{x + y = }')     # x + y = 52
```

Konstanter

```
import string
print(string.ascii_letters)   #
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.ascii_lowercase) # abcdefghijklmnopqrstuvwxyz
print('-----------')
print(string.ascii_uppercase) # ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.digits)          # 0123456789
print('-----------')
print(string.punctuation)     # '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
print(string.printable)       # siffer + bokstaver + tegn + whitespace
print('-----------')
print(string.whitespace)      # mellomrom + tab + linjeskift etc....
print('-----------')
```

↑ Innhold

Operasjoner og metoder

Grunnleggende operasjoner

```
print('abc' + 'def') # Konkatenasjon
print('abc' * 3)     # Repetisjon
print(len('abc'))    # Lengde
print()

 # Medlemskap (sjekk om venstresiden finnes som substreng av høyresiden)
print('a' in 'abc')  # True
print('bc' in 'abc') # True
print('ac' in 'abc') # False, 'ac' er ikke sammenhengde i 'abc'
print('A' in 'abc')  # False, 'A' er ikke det samme som 'a'
print('' in 'abc')   # True, den tomme strenger er alltid en substreng
```

Metoder

En *metode* er en funksjon som kalles «på» et objekt. Kallet utføres ved hjelp av et punktum mellom objektet og metode-navnet (se eksempler under). Ulike klasser har ulike metoder tilgjengelig; strenger (`str`) har for eksempel en metode `lower` som returnerer en kopi av strengen der alle store bokstaver gjøres små.

```
# .upper og .lower lager en kopi av teksten med bare store eller små bokstaver
s = 'FooBar'
s_low = s.lower()
s_upp = s.upper()
print(s)
print(s_low)  # foobar
print(s_upp)  # FOOBAR
print('---')
```

Noen flere eksempler på metoder. Dette er bare et lite utvalg, se den offisielle dokumentasjonen (https://docs.python.org/3/library/stdtypes.html#string-methods) for fullstendig oversikt.

↑ Innhold

Beskrivelse	Eksempel	
	Kall	Returverdi
replace Opprett kopi av strengen hvor enkelte deler er erstattet med noe annet.	`'Foo'.replace('o', 'op')`	`'Fopop'`
	`'hahahaha'.replace('hah', 'l')`	`'lala'`
	`'hahah'.replace('a', '')`	`'hhh'`
strip Opprett kopi av strengen som ikke inkluderer whitespace (mellomrom, linjeskift etc.) i begynnelsen og slutten.	`'Foo\n'.strip()`	`'Foo'`
	`'  bar  ! '.strip()`	`'bar !'`

Split og join

Split kan brukes for å klippe opp en streng i biter, mens join kan brukes for å lime sammen biter til én streng.

```
# .split() deler opp en streng i biter, og legger bitene i en liste
names = 'Marshall,Rubble,Chase,Rocky,Zuma,Sky'
names_list = names.split(',')
print(names)
print(names_list) # ['Marshall', 'Rubble', 'Chase', 'Rocky', 'Zuma', 'Sky']
print('---')

# .join() limer sammen strenger med en limestreng
print(''.join(names_list)) # MarshallRubbleChaseRockyZumaSky
print(','.join(names_list)) # Marshall,Rubble,Chase,Rocky,Zuma,Sky
print('-+-'.join('ABC')) # A-+-B-+-C
print('---')
```

Søking i strenger

```
print('Dette er et ran'.count('et')) # 2
print('Dette er ETT ran'.count('et')) # 1
print('-------')
print('Hunder og katter'.startswith('Hun'))     # True
print('Hunder og katter'.startswith('Hun der')) # False
print('-------')
print('Hunder og katter'.endswith('er'))      # True
print('Hunder og katter'.endswith('mer'))     # False
print('-------')
print('Hunder og katter'.find('og'))          # 7
print('Hunder og katter'.find('eller'))       # -1
print('-------')
print('Hunder og katter'.index('og'))         # 7
print('Hunder og katter'.index('eller'))      # Krasj!
```

Indeksering og beskjæring

Indeksering

```
s = 'abcdefgh'
print(s)
print(s[0]) # a
print(s[1]) # b
print(s[2])
print()

length = len(s)
print(s[length - 1])
print(s[length]) # Krasjer (string index out of range)
```

Negative indekser

```
s = 'abcdefgh'
print(s)
print(s[-1]) # Snarvei for s[len(s) -  1]
print(s[-2])
```

Beskjæring (engelsk: slicing)

↑ Innhold

```
# Beskjæring er som å indeksere, men vi kan hente ut mer enn ett tegn
#
# For en streng s vil s[<start>:<slutt>] evaluere til en streng som
# begynner med tegnet på indeks <start> i s og går opp til men ikke
# inkludert tegnet på indeks <slutt>.
#
# Minner dette om range(a, b)?

s = 'abcdefgh'
print(s)       # abcdefgh
print(s[0:3])  # abc
print(s[1:3])  # bc
print()

print(s[2:3])  # c
print(s[3:3])  #         (ingenting -- dette er den tomme strengen (''))
print('---')
```

Beskjæring med default-verdier

```
s = 'abcdefgh'
print(s)       # abcdefgh
print(s[3:])   # defgh
print(s[:3])   # abc
print(s[:])    # abcdefgh
print('---')
```

Beskjæring med steg

```
# Dette er ikke vanlig, men illustrerer slektskapet med range()
#
# For en streng s vil s[<start>:<slutt>:<steg>] beskjære strengen
# ved å begynne med tegnet på indeks <start>, og gå opp til og ikke
# inkludert <slutt> med avstand på <steg>

s = 'abcdefgh'
print(s)             # abcdefgh
print(s[1:7:2])      # bdf
print(s[1:7:3])      # be
print('---')
print(s[0:len(s):2]) # aceg
print(s[::2])        # aceg
print(s[1::2])       # bdfh
print('---')
print(s[3:0:-1])     # dcb
print('---')
```

↑ Innhold

Å reversere en streng

```
s = 'abcdefgh'

print('Dette virker, men er forvirrende:')
print(s[::-1])

print('Dette virker også, men er fremdeles forvirrende:')
print(''.join(reversed(s)))

print('Beste løsning: skriv funksjon med selvforklarende navn.')
def reversed_string(s):
    return s[::-1]

print(reversed_string(s)) # klart og tydelig!
```

Løkker over strenger

Uten indeksering

```
s = 'abcd'
# For-løkke over s. Iteranden er et symbol i strengen.
for c in s:
    print(c)

# a
# b
# c
# d
```

Med indeksering

↑ Innhold

```python
s = 'abcd'
# For-løkke over lengden til s. Iteranden er en indeks til strengen.
for i in range(len(s)):
    c = s[i]
    print(i, c)

# 0 a
# 1 b
# 2 c
# 3 d


print('---')
# For-løkke med enumerate. Både indeks og symbolet er iterander.
for i, c in enumerate(s):
    print(i, c)

# 0 a
# 1 b
# 2 c
# 3 d
```

Oppdeling med `split`

```python
names = 'Marshall,Rubble,Chase,Rocky,Zuma,Sky'
for name in names.split(','):
    print(name)

# Marshall
# Rubble
# Chase
# Rocky
# Zuma
# Sky


# Med indeksering
for i, name in enumerate(names.split(',')):
    print(i, name)

# 0 Marshall
# 1 Rubble
# 2 Chase
# 3 Rocky
# 4 Zuma
# 5 Sky
```

Oppdeling med `splitlines`

↑ Innhold

```
quotes = '''\
Dijkstra: Simplicity is prerequisite for reliability.
Knuth: If you optimize everything, you will always be unhappy.
Dijkstra: Perfecting oneself is as much unlearning as it is learning.
Knuth: Beware of bugs in the above code; I have only proved it correct, \
not tried it.
Dijkstra: Computer science is no more about computers than astronomy is \
about telescopes.
'''

for line in quotes.splitlines():
    if line.startswith('Knuth'):
        print(line)

# Knuth: If you optimize everything, you will always be unhappy.
# Knuth: Beware of bugs in the above code; I have only proved it correct, not
tried it.
```

Oppdeling med `splitlines` hvor man også inkluderer selve linjeskift-symbolet

```
paragraph = '''\
Denne strengen
inneholder linjeskift.
'''

for line in paragraph.splitlines(keepends=True):
    print('Line:', repr(line))

# Line: 'Denne strengen\n'
# Line: 'inneholder linjeskift.\n'
```

Palindromer

Et palindrom er en streng som er lik fremlengs og baklengs.

↑ Innhold

```
# Det er mange måter å skrive en is_palindrome(s) -funksjon
# Her er flere. Hvilken er best?

def reversed_string(s):
    return s[::-1]

def is_palindrome1(s):
    return (s == reversed_string(s))

def is_palindrome2(s):
    for i in range(len(s)):
        if (s[i] != s[len(s)-1-i]):
            return False
    return True

def is_palindrome3(s):
    for i in range(len(s)):
        if (s[i] != s[-1-i]):
            return False
    return True

def is_palindrome4(s):
    while (len(s) > 1):
        if (s[0] != s[-1]):
            return False
        s = s[1:-1]
    return True

def is_palindrome5(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome5(s[1:-1])

print(is_palindrome1('abcba'), is_palindrome1('abca'))
print(is_palindrome2('abcba'), is_palindrome2('abca'))
print(is_palindrome3('abcba'), is_palindrome3('abca'))
print(is_palindrome4('abcba'), is_palindrome4('abca'))
print(is_palindrome5('abcba'), is_palindrome5('abca'))
```

Representasjon i minnet

En streng representeres fysisk i datamaskinen (som alt annet) med en rekke av høye og lave spenninger vi kan tenke på som en sekvens av 1'ere og 0'ere. Hvordan en slik sekvens med 1'ere og 0'ere oversettes til ulike meningsbærende tegn og symboler avgjøres først og fremst av hvilken *enkoding* som brukes. Under panseret benytter Python faktisk flere ulike enkodinger, men de har alle til felles at de er i overenkomst med *unicode* -standarden. *Unicode* er en standard som tilordner hvert enkelt tegn med en såkalt *ordinal*-verdi

som er et heltall mellom 0 og $1\ 111\ 998$. I skrivende stund er det $149\ 186$ av disse tallverdiene som faktisk har symboler knyttet til seg.

↑ Innhold

Vi kan se en oversikt over en del vanlige tegn og deres unicode-verdi på wikipedia (https://en.wikipedia.org/wiki/List_of_Unicode_characters#Basic_Latin). Vi kan for eksempel lese i tabellen at tegnet `A` har verdien 65 (i desimal), mens symbolet `a` har verdien 97.

```
# For å finne unicode-verdien (ordinal) til et tegn
c1 = 'A'
u1 = ord(c1)
print(c1, u1)

# For å konvertere en ordinal tilbake til et tegn (character)
u2 = 97
c2 = chr(u2)
print(c2, u2)

# Skriv ut alfabetet
for i in range(ord('A'), ord('Z') + 1):
    print(chr(i), end='')
print()
```

Når man sammenligner to strenger, sammenlignes egentlig ordinal-verdien til tegnene i de to strengene. På grunn av rekkefølgen de engelske bokstavene har i unicode-tabellen, vil denne sammenligningen være «alfabetisk» dersom det ikke blandes mellom store og små bokstaver

```
print("'A' < 'a':", 'A' < 'a') # True, siden 65 < 97 er True
print("'a' < 'A':", 'a' < 'A') # False
print()

def compare_lt(s1, s2):
    print(f'{repr(s1)} < {repr(s2)}: {s1 < s2}')

compare_lt('abc', 'abx') # True, siden c har lavere ordinal enn liten x
compare_lt('abc', 'abX') # False, siden c ikke har lavere ordinal enn stor X
compare_lt('abc', 'abc') # False, når verdiene er like vil ikke < gi True
compare_lt('ab', 'abc') # True, den første strengen er prefiks for den andre
compare_lt('ac', 'abc') # False, c har ikke lavere ordinal enn b
```

Eksempel på bruk av ordinaler: simpel kryptering.

↑ Innhold

```python
# Vi kan utnytte ordinalene for å kryptere en melding
def encode(message, shift):
    message = message.upper()
    result = ''
    for c in message:
        ordinal = ord(c) - ord('A')
        ordinal = (ordinal + shift) % (ord('Z') - ord('A') + 1)
        result += chr(ord('A') + ordinal)
    return result

def decode(message, shift):
    return encode(message, -shift)

# Eksempel på kryptering
print(encode('ABCDEFGHIJKLMNOPQRSTUVWXYZ', 3))
print(encode('HELLOCRYPTO', 5))

# Eksempel på dekryptering
print(decode('DEFGHIJKLMNOPQRSTUVWXYZABC', 3))
print(decode('MJQQTHWDUYT', 5))
```

Lese og skrive til fil

```python
# Vi kan benytte Path fra 'pathlib' for å lese og skrive filer
from pathlib import Path

# Skrive til fil
contents_to_write = 'Dette er en test!\nDet er bare en test!'
name_of_file = 'foo.txt'
Path(name_of_file).write_text(contents_to_write, encoding='utf-8')

# Lese fra fil
name_of_file = 'foo.txt'
content = Path(name_of_file).read_text(encoding='utf-8')
```

*Funksjonene for å lese og skrive filer vil tolke filnavn/fil-stier relativt til den mappen skriptet blir **startet** fra – merk at dette ikke nødvendigvis er samme mappe hvor skriptet **ligger**. Når du kjører koden gjennom VSCode er start-mappen den mappen hvor du har åpnet VSCode, og ikke nødvendigvis den mappen hvor filen ligger (f. eks. dersom filen ligger i en undermappe).*

Hjelp, filen blir ikke funnet

Når du kjører et Python-program, kjører programmet «i» en mappe som kalles *current working directory* (cwd). Du kan se hvilken mappe dette er med koden:

↑ Innhold

```python
from pathlib import Path
cwd = Path.cwd()
print(cwd)
```

Denne mappen blir bestemt av hvilket program som *starter* python. F. eks. hvis du bruker VSCode for å starte python, vil cwd være samme mappe som VSCode er åpnet i (som altså ikke har noen sammenheng med hvilken mappe filen som kjøres ligger i).

Når python får beskjed om å åpne en fil, vil den tolke filstien som blir oppgitt *relativt til* cwd. For eksempel, hvis filstien er kun et filnavn, antas det at filen ligger i cwd.

La oss si at du bruker funksjonskallet `Path('foo.txt').read_file(encoding='utf-8')` og ønsker å åpne filen *foo.txt*, som ligger i samme mappe som python-filen du kjører, la oss si mappen *labX*. La oss videre tenke oss at *labX* i sin tur ligger i mappen *inf100*, og det er i den sistnevnte mappen du har åpnet VSCode. Da vil programmet krasje med en *FileNotFoundError*.

For å klare å åpne filen *foo.txt* ved å kjøre Python fra VSCode, kan du gjøre ett av fire tiltak:

- flytte *foo.txt* til den mappen du har åpnet VSCode i (altså *inf100* -mappen i vårt eksempel), eller
- åpne VSCode i den mappen *foo.txt* ligger i (altså *labX* -mappen i vårt eksempel), eller
- endre funksjonskallet til `Path('labX/foo.txt').read_file(encoding='utf-8')`, eller
- (for terminal-brukere) bruk terminalen til å starte Python i stedet for å bruke run-knappen i VSCode: naviger først til mappen *labX* med `cd` -kommandoen, og start så programmet derfra (e.g. en av kommandoene `python <filnavn>`, `py <filnavn>` eller `python3 <filnavn>`).

Det er teknisk sett mulig å endre cwd til å bli samme mappe som filen som kjøres ligger i programmatisk:

```python
import os
directory_of_current_file = os.path.dirname(__file__)
os.chdir(directory_of_current_file) # endrer cwd
```

*Dette kan kanskje gjøre ting lettere i utviklingsfasen og for raske og enkle formål, men er sannsynligvis **ikke** noe en erfaren programmerer ville ønsket seg; siden man da må flytte selve kildekodefilene bare fordi man vil bruke programmet i en annen mappe.*

Lister

↑ Innhold

- Lister ABC
- Hva en liste er (egentlig)
- Alias og mutasjon
- Alias og funksjonsparametre (destruktive funksjoner)
- Opprette lister
- Funksjoner og operasjoner
- Indeksering og beskjæring
- Mutasjon og alias (reprise)
- Kopiering av lister
- Destruktive funksjoner
- Leting etter elementer
- Legge til elementer
- Fjerne elementer
- Løkker over lister
- Sortering og reversering
- Pakke ut en liste i variabler
- Tupler
- Listebygging / list comprehension (løkker inni lister)
- Konvertering mellom lister og strenger (split/join)

Lister ABC

En liste er en ordnet samlig med andre verdier. De viktigste operasjonene er:

↑ Innhold

```python
# Opprette en liste
a = ['foo', 'bar', 42]
print(a) # ['foo', 'bar', 42]
print(len(a)) # 3 (lengden på listen)

# Se på enkelt-element med indeksering (første element har index 0)
x = a[0]
y = a[2]
z = a[-1] # negativ indeks begynner bakfra
print(x, y, z) # foo 42 42

# Bruk en løkke for å se på alle elementene i listen
for element in a:
    print(element, end=' ') # foo bar 42
print()

# Bruk en løkke for å gå gjennom alle indeksene til listen
for i in range(len(a)):
    print(i, end=' ') # 0 1 2
print()

# Sjekk om et element er i listen med `in` -operatoren
if 'bar' in a:
    print('bar er i listen') # bar er i listen
else:
    print('bar er ikke i listen')

# Legge til et element på slutten av listen med .append (PS: muterende)
a.append(99)
print(a) # ['foo', 'bar', 42, 99]

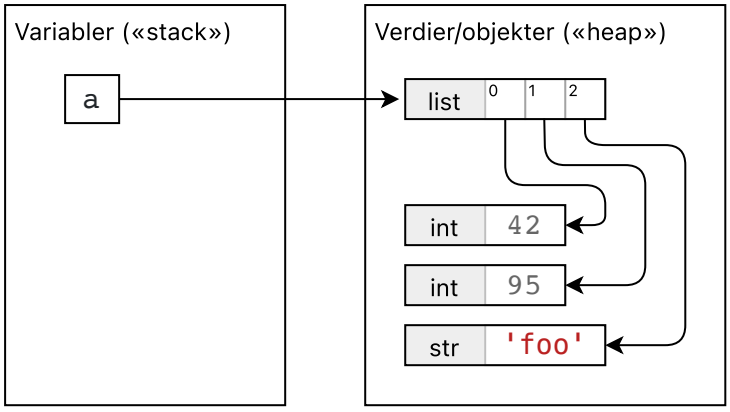
# Endre på en posisjon i listen ved indeksering (PS: muterende)
a[1] = 3.14
print(a) # ['foo', 3.14, 42, 99]
```

Hva en liste er (egentlig)

Til vanlig tenker vi på en liste som en samling av verdier; men egentlig bør vi tenke på det som en samling av *referanser* til verdier. Under viser vi en illustrasjon av minnet til datamaskinen etter at vi har opprettet en liste med tre elementer.

```python
# Oppretter en liste a med tre elementer
a = [42, 95, 'foo']
```

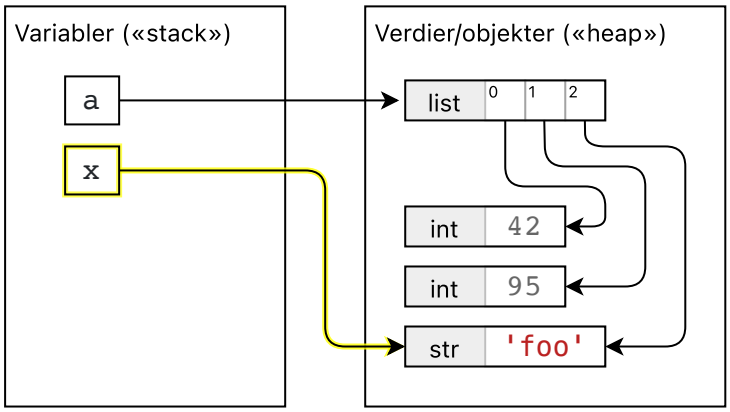
↑ Innhold



Vi kan hente ut enkelt-verdier fra en liste ved å slå opp i listen med indeksering. Indeksering starter på 0, så første element i listen har indeks 0, andre element har indeks 1, osv.

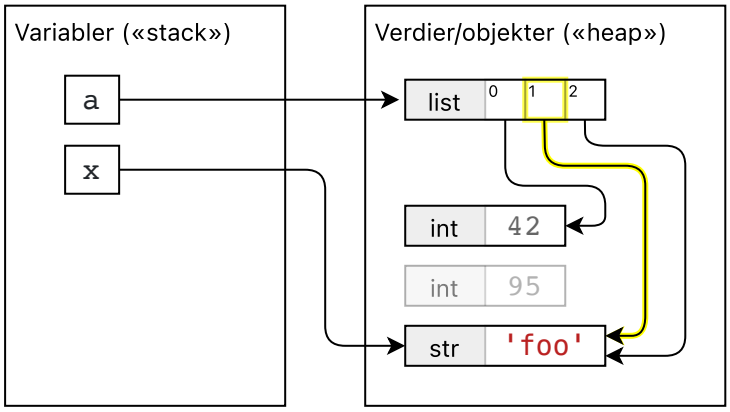
Hvis vi henter ut en enkelt-verdi fra listen *a* ved å benytte indeksering (f. eks. `a[2]`) vil uttrykket evaluere til verdien som pekes på. Eksempel: uttrykket `a[2]` vil i eksempelet over evaluere til verdien `'foo'`. Hvis vi angir at `a[2]` er verdien til en ny variabel *x*, vil minnet endres som vist under:

```
x = a[2]
```



I motsetning til de verdi-typene vi har sett tidligere i emnet, er det mulig å endre en liste *uten* å opprette et helt nytt objekt i minnet. Dette kalles å *mutere* en liste. Eksempel: hvis vi angir at verdien til *a[1]* skal settes til verdien av *x*, vil minnet endres som vist under:

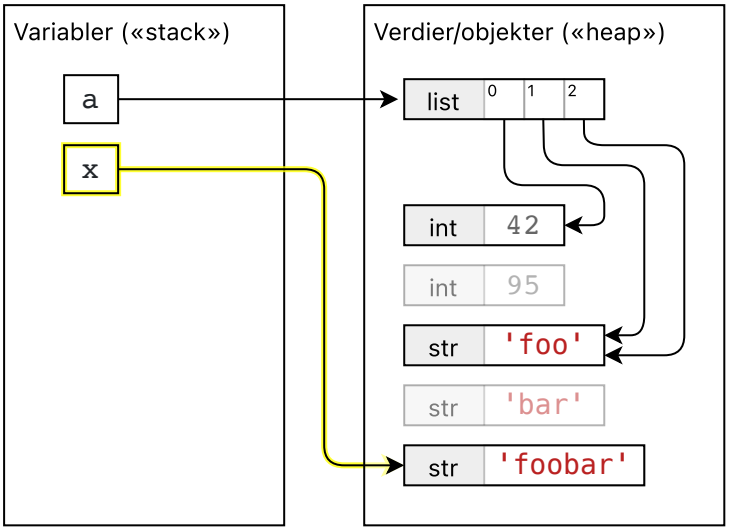
```
a[1] = x
```



↑ Innhold

Til sammenligning: om vi endrer verdien av *x* (en streng) ved å konkatinere mer tekst, vil det opprettes en helt ny verdi i minnet – strengen `'foo'` som *x* opprinnelig pekte på blir ikke endret:

```
x += 'bar'
```



Strenger kan nemlig *ikke* muteres.

Hele programmet over

```
a = [42, 95, 'foo']
x = a[2]
a[1] = x
x += 'bar'

print(a) # [42, 'foo', 'foo']
print(x) # foobar
```

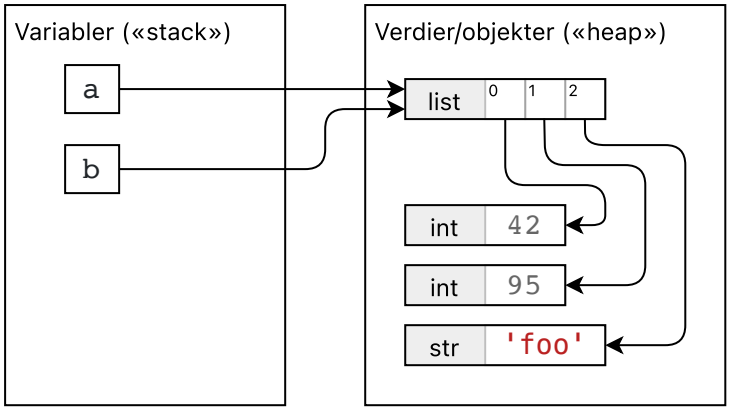
Alias og mutasjon

I motsetning til datatyper vi har sett tidligere, er det (som vist i avsnittet over) mulig å *endre* på en liste uten å opprette en ny verdi i minnet. Dette kaller vi å *mutere* listen. Dette gjør at vi må være forsiktige dersom to variabler peker på samme liste. Dersom vi muterer listen via den ene variabelen, vil endringen også gjelde for den andre variabelen.

Hvis to variabler peker på samme liste, kaller vi dem *aliaser* for hverandre.

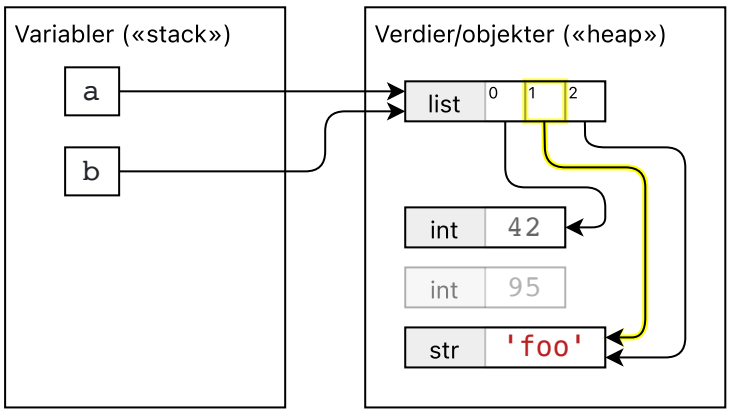
```
a = [42, 95, 'foo']
b = a                # b er nå et alias for a
```

↑ Innhold



Om vi muterer listen gjennom `a` vil altså `b` påvirkes (og vice versa).

```
a[1] = 'foo'          # vi muterer a
```



Komplett eksempel:

```
a = [42, 95, 'foo']
b = a                 # b er nå et alias for a

print(a) # [42, 95, 'foo']
print(b) # [42, 95, 'foo']

a[1] = 'foo'          # vi muterer a
b[0] = 95             # vi muterer b

# Begge mutasjoner reflekteres i begge aliasene
print(a) # [95, 'foo', 'foo']
print(b) # [95, 'foo', 'foo']
```

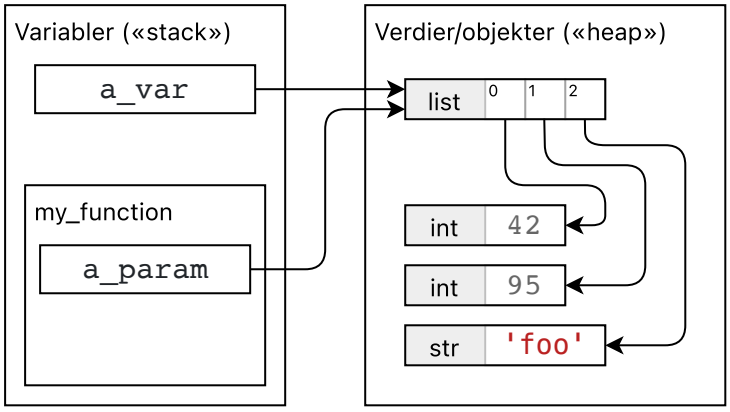
Alias og funksjonsparametre (destruktive funksjoner)

En av de vanligste typen alias vi har, opplever vi dersom vi kaller en funksjon med et enkelt variabeluttrykk som argument. Da vil parameteren og det opprinnelige variabelen være aliaser.

↑ Innhold

```python
def my_function(a_param):
    ... # a_var og a_param er aliaser når vi kommer hit

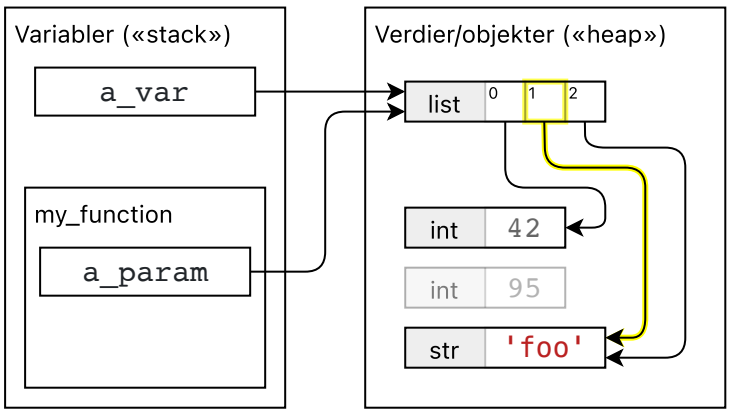
a_var = [42, 95, 'foo']
my_function(a_var)
```



Dersom funksjonen muterer parameteren, vil det også påvirke den opprinnelige variabelen (aliaset). Vi kaller slike funksjoner for *destruktive* funksjoner, siden de «ødelegger» (muterer) verdiene den blir gitt som argument. Dette noen ganger er ønskelig, og andre ganger ikke (avhengig av situasjonen). Det er viktig å være bevisst på om en funksjon er destruktiv eller ikke, ellers kan det oppstå feil i programmet som er krevende å debugge.

```python
def my_function(a_param):
    a_param[1] = 'foo' # muterer a_param

a_var = [42, 95, 'foo']
my_function(a_var)
print(a_var) # [42, 'foo', 'foo']
```



Opprette lister

Tomme lister

↑ Innhold

```
# To standard måter å opprette tomme lister på
a = []
b = list()

print(a) # []
print(b) # []
print(a == b) # True

# Lengden til en liste
print(len(a)) # 0

# Typen til en liste
print(type(a)) # <class 'list'>
```

Lister med ett element

```
a = ['foo']
b = [42]

print(a) # ['foo']
print(b) # [42]
print(len(a)) # 1
print(len(b)) # 1
print(a == b) # False
```

Lister med flere elementer

```
a = [2, 3, 5, 7, 11]
b = list(range(5))
c = ['foo', 42, True, None, '']

print(len(a), a) # 5 [2, 3, 5, 7, 11]
print(len(b), b) # 5 [0, 1, 2, 3, 4]
print(len(c), c) # 5 ['foo', 42, True, None, '']
```

Lister med et variabel antall elementer

```
n = 5
a = ['foo'] * n
b = [7, 99] * n
c = list(range(n))

print(len(a), a) # 5 ['foo', 'foo', 'foo', 'foo', 'foo']
print(len(b), b) # 10 [7, 99, 7, 99, 7, 99, 7, 99, 7, 99]
print(len(c), c) # 5 [0, 1, 2, 3, 4]
```

Funksjoner og operasjoner

↑ Innhold

```
a = [2, 3, 5, 3, 7]
print('a = ', a)
print('len =', len(a)) # 5
print('min =', min(a)) # 2
print('max =', max(a)) # 7
print('sum =', sum(a)) # 20

# Et par forskjellige lister
b = [2, 3, 5, 3, 7]   # lik til a
c = [2, 3, 5, 3, 8]   # forskjellig fra a
d = [2, 3, 5]         # prefix for a

print('a =', a)
print('b =', b)
print('c =', c)
print('d =', d)

print('------------------')
print('a == b', (a == b)) # True
print('a == c', (a == c)) # False
print('a == d', (a == d)) # False

print('------------------')
print('a < c', (a < c)) # True (sammenligning skjer basert på første ulike
element)
print('d < a', (d < a)) # True
print('d > a', (d > a)) # False
```

```
# Konkatenering
a = [3, 4]
b = [8, 9]
print('a     =', a)
print('b     =', b)
print('a + b =', a + b) # [3, 4, 8, 9]

# Repetisjon
print('a * 3 =', a * 3) # [3, 4, 3, 4, 3, 4]
```

↑ Innhold

Indeksering og beskjæring

```
# Indeksering og beskjæring fungerer på samme måte som for strenger
a = [2, 3, 5, 7, 11, 13]
print('a        =', a)

# Indeksering. Første indeks er 0.
print('a[0]     =', a[0]) # 2
print('a[2]     =', a[2]) # 5

# Negative indekser
print('a[-1]    =', a[-1]) # 13
print('a[-3]    =', a[-3]) # 7

# Beskjæring a[start:slutt] eller a[start:slutt:steg]
print('a[0:2]   =', a[0:2]) # [2, 3]
print('a[1:4]   =', a[1:4]) # [3, 5, 7]
print('a[1:6:2] =', a[1:6:2]) # [3, 7, 13]
```

Mutasjon og alias

I motsetning til datatyper vi har sett hittil, kan vi endre på en liste *uten* å opprette en ny verdi i minnet. Dette kaller vi å *mutere* listen.

```
# Opprett en liste
a = [2, 3, 4]

# La b være en variabel som referer til samme liste som a. Siden a og b
# er variabler som refererer til det samme muterbare objekt, kaller vi
# a og b for aliaser.
b = a

# Mutasjon (endring) av listen
a[0] = 99
b[1] = 42
print(a) # [99, 42, 4]
print(b) # [99, 42, 4]
```

Dersom to variabler refererer til samme muterbare objekt, kalles de for *aliaser*. Funksjonsparametre er eksempler på aliaser.

↑ Innhold

```python
# Når en funksjon muterer en liste via et alias har funksjonen en
# sideeffekt
def f(my_list_parameter):
    my_list_parameter[0] = 42

a = [2, 3, 5, 7]
print(a) # [2, 3, 5, 7]
f(a)
print(a) # [42, 3, 5, 7]
print("---")

# Alias kan bli brutt ved å endre variabelen
def foo(a):
     a[0] = 99
     a = [5, 2, 0] # aliaset blir brutt her
     a[0] = 42

a = [3, 2, 1]
print(a) # [3, 2, 1]
foo(a)
print(a) # [99, 2, 1]
```

Vi kan benytte `is` og `is not` -operatorene for å sjekke om to variabler er aliaser for samme objekt.

↑ Innhold

```python
# Opprett en liste
a = [2, 3, 5, 7]

# Opprett et alias for listen
b = a

# Opprett en ny liste med de samme elementene
c = [2, 3, 5, 7]

# a og b er referanser til (/aliaser for) DEN SAMME listen
# c er en referanse til en annen, men LIK liste

print("først:")
print("       a:", a)
print("       b:", b)
print("       c:", c)
print("== -operatoren forteller hvorvidt to verdier er LIKE")
print("  a == b:", a == b) # True
print("  a == c:", a == c) # True
print("is -operatoren forteller hvorvidt to verdier er DEN SAMME")
print("  a is b:", a is b) # True
print("  a is c:", a is c) # False
print("\n")

# Mutasjon av a endrer også b (DEN SAMME listen) men ikke c (en annen liste)
a[0] = 42
print("etter mutasjonen a[0] = 42")
print("       a:", a) # [42, 3, 5, 7]
print("       b:", b) # [42, 3, 5, 7]
print("       c:", c) # [2, 3, 5, 7]
print("  a == b:", a==b) # True
print("  a == c:", a==c) # False
print("  a is b:", a is b) # True
print("  a is c:", a is c) # False
```

↑ Innhold

Kopiering av lister

```python
# Vi må være forsiktig ved kopiering av lister, slik at vi ikke kommer i
# skade for å muterer en liste av vanvare gjenomm et alias.

import copy

a = [2, 3]

# To kopier
b = a               # Ikke en kopi, bare et alias
c = copy.copy(a)    # Ekte kopi

# I begynnelsen ser kopiene tilforlatelig like ut
print("Først...")
print("   a =", a) # [2, 3]
print("   b =", b) # [2, 3]
print("   c =", c) # [2, 3]

# Så muterer vi a[0]
a[0] = 42
print("Etter mutasjonen a[0] = 42")
print("   a =", a) # [42, 3]
print("   b =", b) # [42, 3]
print("   c =", c) # [2, 3]
```

Andre måter å kopiere

↑ Innhold

```python
import copy

a = [2, 3]

b = a
c = copy.copy(a)
d = a[:]
e = a + []
f = list(a)
g = a.copy()
*h, = a

li = []
for element in a:
    li.append(element)

print("Først...")
print(a, b, c, d, e, f, g, h, li)
a[0] = 42

print("Etter mutering a[0]") # Klarer du å gjette hvilke «kopier» som blir
mutert?
print(a, b, c, d, e, f, g, h, li)
```

Destruktive funksjoner

En funksjon er *destruktiv* dersom den har sideeffekter som muterer en parameter (eller hvis den muterer en global variabel).

```python
# En destruktiv funksjon er skrevet for å mutere en liste. Den trenger ikke
# returnere noe, siden den som kaller også har et alias til listen.
def fill(a, value):
    for i in range(len(a)):
        a[i] = value

a = [1, 2, 3, 4, 5]
print("Først, a =", a) # [1, 2, 3, 4, 5]
fill(a, 42)
print("Etter fill(a, 42), a =", a) # [42, 42, 42, 42, 42]
```

En *ikke-destruktiv* funksjon vil ikke ha sideeffekter, og vi benytter oss av returverdien i stedet.

↑ Innhold

```python
import copy

def destructive_remove_all(a, value):
    while value in a:
        a.remove(value)

def non_destructive_remove_all(a, value):
    # Vanligvis skriver vi ikke-destruktive funksjoner ved å opprette
    # en ny liste fra scratch, og så muterer vi den nye listen
    result = []
    for element in a:
        if element != value:
            result.append(element)
    return result # ikke-destruktive funksjoner MÅ returnere svaret!

def alternate_non_destructive_remove_all(a, value):
    # Vi kan også skrive en ikke-destruktiv funksjon ved å først bryte
    # aliaset, og deretter benytte en destruktiv tilnærming
    a = copy.copy(a)
    destructive_remove_all(a, value)
    return a # ikke-destruktive funksjoner må uansett returnere!

a = [1, 2, 3, 4, 3, 2, 1]
print("Først")
print("   a =", a) # [1, 2, 3, 4, 3, 2, 1]

destructive_remove_all(a, 2)
print("Etter destructive_remove_all(a, 2)")
print("   a =", a) # [1, 3, 4, 3, 1]

b = non_destructive_remove_all(a, 3)
print("Etter b = non_destructive_remove_all(a, 3)")
print("   a =", a) # [1, 3, 4, 3, 1]
print("   b =", b) # [1, 4, 1]

c = alternate_non_destructive_remove_all(a, 1)
print("Etter c = alternate_non_destructive_remove_all(a, 1)")
print("   a =", a) # [1, 3, 4, 3, 1]
print("   c =", c) # [3, 4, 3]
```

Leting etter elementer

↑ Innhold

```
# Inneholder listen min verdi?
a = [2, 3, 5, 2, 6, 2, 2, 7]
print("a      =", a)
print("2 in a =", (2 in a)) # True
print("4 in a =", (4 in a)) # False

# eller ikke?
print("2 not in a =", (2 not in a)) # False
print("4 not in a =", (4 not in a)) # True
```

```
# Hvor mange ganger opptrer min verdi?
a = [2, 3, 5, 2, 6, 2, 2, 7]
print("a          =", a)
print("a.count(1) =", a.count(1)) # 0
print("a.count(2) =", a.count(2)) # 4
print("a.count(3) =", a.count(3)) # 1
```

```
# Hvor i listen befinner verdien seg, da?
# a.index(element) eller a.index(element, start_after)
a = [2, 3, 5, 2, 6, 2, 2, 7]
print("a             =", a)
print("a.index(6)    =", a.index(6)) # 4
print("a.index(2)    =", a.index(2)) # 0
print("a.index(2, 1) =", a.index(2, 1)) # 3
print("a.index(2, 5) =", a.index(2, 5)) # 5
```

```
# Oj! Krasjer dersom elmentet ikke er der
a = [2, 3, 5, 2]
print("a          =", a)
print("a.index(9) =", a.index(9)) # krasjer!
print("Vi kom visst ikke så langt...")
```

```
# Løsning: benytt (element in liste) først.
a = [2, 3, 5, 2]
print("a =", a)
if (9 in a):
    print("a.index(9) =", a.index(9))
else:
    print("9 er ikke der", a)
print("Hurra, ingen krasj!")
```

↑ Innhold

Legge til elementer

Destruktive metoder for å legge til elementer:

```
# Vi oppretter en liste og gir den et alias. Alle endringer vi gjør her
# reflekteres i aliaset, hvilket betyr at endringene er destruktive
a = [2, 3]
alias = a

# Legg til på slutten med .append
a.append(7)
print(a) # [2, 3, 7]

# Legg til på en bestemt posisjon med .insert
a.insert(2, 42)
print(a) # [2, 3, 42, 7]

# Utvid listen med flere elementer på en gang med .extend eller '+='
b = [100, 200]
a.extend(b)
print(a) # [2, 3, 42, 7, 100, 200]
a += b
print(a) # [2, 3, 42, 7, 100, 200, 100, 200]
print()

print(alias) # [2, 3, 42, 7, 100, 200, 100, 200]
```

Ikke-destruktive operasjoner for å legge til elementer:

```
a = [2, 3]

# Legg til på slutten med +
b = a + [13, 17]
print(a)
print(b)

# Legg til midt inne i listen med beskjæring
c = a[:1] + [42] + a[1:]
print(a)
print(c)
```

Destruktiv vs. ikke-destruktiv utvidelse

↑ Innhold

```
print("Destruktiv:")
a = [2, 3]
b = a           # lager alias
a += [4]
print(a)
print(b)

print("Ikke-destruktiv:")
a = [2, 3]
b = a           # lager alias
a = a + [4]   # bryter aliaset med b, a er nå referanse til ny liste
print(a)
print(b)
```

Fjerne elementer

Destruktive metoder for å fjerne elementer

```
a = [2, 3, 5, 3, 7, 6, 5, 11, 13]
print("a =", a)

# Fjerne første opptreden av et bestemt element
a.remove(5)
print("Etter a.remove(5), a=", a)
a.remove(5)
print("Etter enda en a.remove(5), a=", a)

# Fjerne det siste elementet i listen
item = a.pop()
print("Etter item = a.pop()")
print("   item =", item)
print("   a =", a)

# Fjerne et element på en bestemt indeks
item = a.pop(3)
print("Etter item = a.pop(3)")
print("   item =", item)
print("   a =", a)
```

Ikke-destruktive operasjoner for å fjerne elementer

↑ Innhold

```
a = [2, 3, 5, 3, 7, 5, 11, 13]
print("a =", a)

# Ikke-destruktiv fjerning av elementene mellom indeks 2 og 3
b = a[:2] + a[3:]
print("Etter b = a[:2] + a[3:]")
print("   a =", a)
print("   b =", b)
```

Løkker over lister

```
# Iterasjon med indeks
a = [2, 3, 5, 7]
for index in range(len(a)):
    print(f"a[{index}] =", a[index])

print("---")

for index, item in enumerate(a):
    print(f"a[{index}] =", item)
```

```
# Iterasjon uten indeks, såkalt for-hver -løkke (engelsk: foreach)
# Lister og strenger er begge samlinger, såkalte «itererbare» typer.
# Det betyr at vi kan benytte en for-løkke på dem direkte
a = [2, 3, 5, 7]
for item in a:
    print(item)
```

```
# IKKE FJERN ELLER LEGG TIL ELEMENTER TIL SAMME LISTE DU GÅR GJENNOM
# MED EN FOR-LØKKE! INDEKSER KRØLLER SEG TIL!(dette er ikke et problem
# for strenger, siden de ikke kan muteres)
a = [2, 3, 5, 3, 7]
print("a =", a)

# Mislykket forsøk på å fjerne alle 3'erne
for index in range(len(a)):
    if (a[index] == 3):  # vi krasjer her etter en stund
        a.pop(index)

print("Hit kommer vi ikke")
```

↑ Innhold

```
# IKKE MUTER EN LISTE INNI EN FOR-HVER -LØKKE!
# Vil ikke krasje, men gjør heller ikke som vi forventer
a = [3, 3, 2, 3, 4]
print("Først, a =", a)

# Mislykket forsøk på å fjerne alle 3'erne
def should_be_removed(x):
    return x == 3

for item in a:
    if should_be_removed(item):
        a.remove(item)

print("Etter, a =", a) # [2, 3, 4]
```

```
# Bedre: mutering i en while-løkke.
# Her har vi full kontroll på hvordan indeks endrer seg.
a = [3, 3, 2, 3, 4]
print("Først, a =", a)

# Vellykket forsøk på å fjerne alle 3'erne
def should_be_removed(x):
    return x == 3

index = 0
while (index < len(a)):
    value = a[index]
    if (should_be_removed(value)):
        a.pop(index)
    else:
        index += 1

print("Huzza! a =", a) # [2, 4]
```

```
# Enda en annen variant som virker tilfeldigvis for akkurat å fjerne alle 3'ere
a = [3, 3, 2, 3, 4]
while 3 in a:
    a.remove(3)
print("a =", a)
```

Sortering og reversering

Destruktiv sortering og reversering

↑ Innhold

```
# Sortering
a = [7, 2, 5, 3, 5, 11, 7]
print("Først, a =", a)
a.sort()
print("Etter a.sort(), a =",a)
print("---")

# Reversering
a = [2, 3, 5, 7]
print("Først, a =", a)
a.reverse()
print("Etter a.reverse(), a =", a)
```

Ikke-destruktiv sortering og reversering

```
# Sortering
a = [7, 2, 5, 3, 5, 11, 7]
print("Først, a =", a)
b = sorted(a)
print("Etter b = sorted(a)")
print("    a =", a)
print("    b =", b)
print("---")

# Reversering
a = [2, 3, 5, 7]
print("Først, a =", a)
b = reversed(a)
c = list(reversed(a))
print("Etter b = reversed(a)  og  c = list(reversed(a))")
print("    a =", a)
print("    b =", b)
print("    c =", c)
print("Her er elementene i b:")
for x in b:
    print(x, end=" ")
print()

print("Her er elementene i b en gang til (men hæ???):")
for x in b:
    print(x, end=" ")
print()
print("---")
```

Sortering med nøkkel

Noen ganger ønsker man å sortere objekter basert på en egendefinert funksjon av hva det vil si å være tidlig eller sent. Dette er for eksempel aktuelt dersom man ønsker å sortere en liste av oppslagsverk med like

↑ Innhold

nøkler; da kan man velge hvilken nøkkel i oppslagsverkene man skal sorte på. I slike tilfeller bruker vi *key* -parameteren til sort (eller sorted).

```
people = [
    { 'name': 'Ada', 'age': 20},
    { 'name': 'Caroline', 'age': 19},
    { 'name': 'Brage', 'age': 22}
]

def get_name(person):
    return person['name']

def get_age(person):
    return person['age']

people_a = sorted(people, key=get_name)
for person in people_a:
    print(person['name'], end=' ')  # Ada Brage Caroline
print()

people_b = sorted(people, key=get_age)
for person in people_b:
    print(person['name'], end=' ')  # Caroline Ada Brage
```

Man kan benytte samme prinsipp for å sortere lister med alle slags objekter, så lenge man selv definerer en funksjon som (a) aksepterer ethvert objekt i listen som argument, og (b) returnerer sammenlignbare verdier.

Et annet eksempel, hvor vi sorterer en liste med tall basert på avstand fra 100.

```
nums = [101, 108, 103, 98, 95]

def distance_from_100(num):
    return abs(num - 100)

nums.sort(key=distance_from_100)
print(nums)  # [101, 98, 103, 95, 108]
```

Pakke ut en liste i variabler

Gitt en liste kan du «pakke ut» verdiene i variabler.

```
a = ["Florida", 15.4, "2022-09-16"]

place, temp, date = a
print(f"{place = }", f" {temp = }", f" {date = }", sep="\n")
```

↑ Innhold

Hvis listen er lang, kan du pakke opp kun de par første verdiene og la resten bli en ny liste. Operasjonen er ikke-destruktiv.

```
a = ["Florida", 15.2, 13.5, 17.2, 13.6, 14.2]

place, *temps = a
print(f"{a=}")
print(f"{place=}")
print(f"{temps=}")
```

eller de par første *og* de par siste. Variabelen med `*` foran plukker opp resten i en ny liste.

```
a = ["Florida", "not interested", 15.2, 13.5, 17.2, 13.6, 14.2, "OK"]

place, _, *temps, last_temp, status = a

print(f"{a         = }")
print(f"{place     = }")
print(f"{_         = }")
print(f"{temps     = }")
print(f"{last_temp = }")
print(f"{status    = }")
```

Ved å sette `*` foran en liste, kan vi pakke opp elementene i listen og bruke dem som om vi bare skilte verdiene med komma uten at de var i en liste. For eksempel kan vi bruke elementene som argumenter til et funksjonskall.

```
def add(x, y):
    return x + y

a = [2, 2]
result = add(*a)
print(result)
```

Eller vi kan kombinere to lister ikke-destruktivt:

```
a = [1, 2]
b = [3, 4]
c1 = [a, b] # En liste av lister -- en 2-dimensjonell liste
c2 = [*a, *b] # En liste med verdiene fra to lister -- en "flat" liste
print(a, b, c1, c2, sep="\n")
```

En funksjon kan akseptere et ukjent antall argumenter ved å pakke dem inn i en liste

↑ Innhold

```python
def multiply(*nums):
    # nums er en liste som inneholder alle argumentene
    result = 1
    for num in nums:
        result *= num
    return result

print(multiply(2, 2))
print(multiply(2, 2, 3))
```

Eller kreve et minimums antall argumenter ved å bare pakke inn bare de siste argumentene i en liste.

```python
def multiply(first_num, second_num, *rest):
    result = first_num * second_num
    for num in rest:
        result *= num
    return result

print(multiply(2, 2, 3))
print(multiply(2, 2))
print(multiply(2)) # krasjer, ikke nok argumenter
```

Tupler

En *tuple* er en slags liste som ikke kan muteres.

```python
t = (1, 2, 3)
print(type(t), len(t), t)

a = [1, 2, 3]
t = tuple(a)
print(type(t), len(t), t)
```

```python
# Tupler kan ikke muteres
t = (1, 2, 3)
print(t[0])

t[0] = 42    # Krasj!
print(t[0])
```

↑ Innhold

```
# Parallell tilording av verdier
(x, y) = (1, 2)
print(x)
print(y)

# Paralell tilordning er nyttig for bytting av verdier
(x, y) = (y, x)
print(x)
print(y)
```

```
# Tuple med kun ett element
t = (42)
print(type(t), t*5) # oj, det var visst ikke en tuple likevel

t = (42,) # bruk komma for å lage en tuple med ett element
print(type(t), t*5)
```

Tupler fungerer på samme måte som lister, men de kan ikke muteres. For å endre på tupler må man bruke ikke-destruktive funksjoner. Ikke-destruktive funksjoner for lister og for tupler er de samme og pleier å ha identisk syntaks.

En vanlig bruk av tupler er å returnere flere verdier fra samme funksjon:

```
# Bruk en tuple til å returnere flere verdier
def positive_and_negative_of(x):
    return (x, -x)

# Pakk ut resultatet til flere variabler (variablene skilles med ,)
hi, lo = positive_and_negative_of(5)
print(f"{hi} {lo}")

# Behold resultatet som en tuple (én variabel på venstresiden av =)
hilo = positive_and_negative_of(7)
print(f"{hilo}")
```

Listebygging / list comprehension (løkker inni lister)

I Python er det mulig å opprette lister med en løkke. Dette kalles for *list comprehension* på engelsk og *listebygging* på norsk (noen steder også kalt *listeinklusjon* eller *listekomprehensjon*).

↑ Innhold

```python
# Den lange måten
a = []
for i in range(10):
    a.append(i + 1)
print(a)

# Med listebygging
a = [i + 1 for i in range(10)]
print(a)

# For de ambisiøse: listebygging med betingelser
a = [i + 1 for i in range(20) if i % 2 == 0]
print(a)

# Listebygging for å ikke-destruktivt filtrere en liste
def divisible_by_3(x):
    return x % 3 == 0
b = [x for x in a if divisible_by_3(x)]
print(b)

# Listebygging for å ikke-destruktivt anvende en funksjon på
# hvert element i en liste
def square(x):
    return x * x
c = [square(x) for x in b]
print(c)
```

Konvertering mellom lister og strenger (split/join)

```python
# bruk list(s) for å konvertere en streng til liste med tegn
a = list("hurra!")
print(a)  # ['h', 'u', 'r', 'r', 'a', '!']

# bruk s1.split(s2) for å konvertere en streng s1 til en liste
# med strenger, klippet opp langs s2'er inne i s1
a = "Hva holder du på med?".split(" ")
print(a) # ['Hva', 'holder', 'du', 'på', 'med?']

# bruk "".join(a) for å lime sammen/konkatenere en liste med strenger
print("".join(a))  #  Hvaholderdupåmed?

# s.join(a) for å lime sammen med s som lime-streng
print(" ".join(a)) # Hva holder du på med?
print("--".join(a)) # Hva--holder--du--på--med?
```

Oppslagsverk

↑ Innhold

- Basics
- Enkle eksempler
- Opprette oppslagsverk
- Egenskaper ved oppslagsverk
- Operatorer, funksjoner og metoder
- Løkker over oppslagsverk

Se også offisiell dokumentasjon (https://docs.python.org/3/library/stdtypes.html#dict) for `dict`.

Basics

Et oppslagsverk (engelsk: *dictionary*) er en datastruktur hvor man kan slå opp på nøkkelverdier («keys») og hente ut en verdi som tidligere har blitt knyttet til denne nøkkelverdien. Tenk på nøkkelverdi som et slags «variabelnavn» og på et oppslagsverk som en samling med variabler.

```
# Opprett et tomt oppslagsverk
d = dict()              # kan også skrives som:   d = {}

# Legg til nøkler og verdier
d['my key'] = 'my value'
d['name'] = 'Arnoldus'
d['age'] = 42

# Hent ut verdiene
print(d['my key'])      # my value
print(d['name'])        # Arnoldus
print(d['age'])         # 42
print(d['age'] + 53)    # 95

# Hente ut verdiene, med default-verdi dersom nøkkel ikke eksisterer
print(d.get('age', 9)) # 42
print(d.get('foo', 9)) # 9   ('foo' er ikke en nøkkel i d)

# Endre på en verdi
d['age'] += 1           # kan også skrives som:   d['age'] = d['age'] + 1

# Verdien er endret
print(d['age'])         # 43
```

↑ Innhold

```python
# Statisk opprettelse av oppslagsverk
d = { 'key': 'value', 'foo': 42, 95: 'McQueen', 99: 200 }

print(d['key'])                # value
print(d['foo'])                # 42
print(d[95])                   # McQueen
print(d[99])                   # 200
print(d['key_does_not_exist']) # Krasjer
```

Enkle eksempler

```python
# Oppretter oppslagsverket
country_map = {
    'Oslo': 'Østlandet',
    'Bergen': 'Vestlandet',
    'Drammen': 'Østlandet',
    'Stavanger': 'Vestlandet',
    'Kristiansand': 'Sørlandet',
}

# Ber bruker om navnet på en by og skriver ut hvor byen ligger
city = input('Skriv inn navnet på en by: ')
if city in country_map:
    print(f'{city} er på {country_map[city]}')
else:
    print(f'Unnskyld, jeg aner ikke hvor {city} er')
```

```python
# Oppretter et oppslagsverk
counts = dict()

# Ber brukeren om å skrive tall, og forteller så brukeren hvor mange ganger
# tallet er sett før.
while True:
    user_input = input('Skriv inn et tall (eller ingenting for å avslutte): ')
    if (user_input == ''):
        break
    n = int(user_input)
    if n in counts:
        counts[n] += 1
    else:
        counts[n] = 1
    print('Jeg har nå sett tallet', n, 'totalt', counts[n], 'ganger.')
print('Ferdig, counts:', counts)
```

↑ Innhold

Opprette oppslagsverk

```
# Opprett et tomt oppslagsverk
d1 = dict()
print(d1) # {}

d2 = {}
print(d2) # {}

# Opprett oppslagsverk statisk
d3 = {'foo':'bar', 42:99}
print(d3) # {'foo': 'bar', 42: 99}

d4 = dict(foo='bar', baz=[1, 2, 3])
print(d4) # {'foo': 'bar', 'baz': [1, 2, 3]}

# Opprett oppslagsverk fra en liste med tupler på størrelse 2
a = [('ku', 5), ('hund', 98), ('katt', 1)]
d5 = dict(a)
print(d5) # {'ku': 5, 'hund': 98, 'katt': 1}
```

Egenskaper ved oppslagsverk

Oppslagsverk knytter nøkler til verdier.

```
ages = dict()
key = 'fred'
value = 38
ages[key] = value  # 'fred' er nøkkelen, 38 er verdien
print(ages[key])
```

En nøkkel er unik, og er tilknyttet kun én verdi.

```
d = dict()
d[2] = 100 # ny nøkkel 2 peker på verdien 100
d[2] = 200 # endrer nøkkel 2 til å peke på verdien 200
d[2] = 400 # endrer nøkkel 2 til å peke på verdien 400
print(d)  # { 2: 400 }
```

Rekkefølge betyr egentlig ingenting; men ved iterasjon er det den rekkefølgen nøklene ble først opprettet som teller.

↑ Innhold

```
d1 = dict()
d1['a'] = 'foo' # Oppretter nøkkelen 'a' først i d1
d1['b'] = 'B'
d1['a'] = 'A'   # Selv om vi endrer på 'a' igjen, er den fremdeles først
print(d1)       # {'a':'A', 'b':'B'}

d2 = {'b':'B', 'a':'A'} # Nøkkelen 'b' kommer først i d2
print(d2)       # {'b':'B', 'a':'A'}

print(d1 == d2) # True, rekkefølge betyr ingenting for likhet
```

Et oppslagsverk kan *muteres*.

```
d1 = { 'foo': 42 }
d2 = d1

d2['foo'] = 95
print(d1['foo']) # 95
```

Nøklene i et oppslagsverk kan være mange forskjellige slags typer; men de kan *ikke* være av en muterbar type. *Verdiene* i et oppslagsverk kan være hva som helst, inkludert muterbare verdier.

```
d = dict()

d['this key is a string'] = 42
d[95] = 'this value has an int as key'
d[('this key', 'is', 'a tuple')] = ['this', 'value', 'is', 'a', 'list']
d[False] = 'booleans are also fine as keys'
d[None] = 'even None is OK'

# Hent ut noen verdier
print(d['this key is a string'])  # 42
print(d[False])                   # booleans are also fine as keys

# Prøver å bruke en liste (altså en muterbar verdi) som nøkkel
a = ['trying', 'to', 'use', 'list', 'as', 'key']
d[a] = 'foo'  # Krasjer
```

Oppslagsverk er *svært* effektive.

↑ Innhold

```python
# Vi kan bruke en liste av tupler som om det var et oppslagsverk
# Prøv å endre på n (hvor mye data vi har) og se effekten på kjøretiden
n = 200
trails = 100
a = [(i, i) for i in range(n)] + [("foo", 42), ("bar", 95)]

# La oss sammenligne hvor effektivt det er i forhold til et oppslagsverk
d = dict(a)

# Operasjonen vi skal sammenligne:
# Sjekk om vi et gitt et (key, value) -par eksisterer i samlingen
def contains_key_value_pair_list(a, key, value):
    return (key, value) in a

def contains_key_value_pair_dict(d, key, value):
    return key in d and value == d[key]


# Ta tiden på listen først
import time
time_before_start = time.time()
result = []
for _ in range(trails):
    val = contains_key_value_pair_list(a, "foo", 42)
time_when_done = time.time()
time_taken_list = (time_when_done - time_before_start) * 1000
print(f"Tid for oppslag på ('foo', 42) med liste: {time_taken_list:.0f}ms")

# Så det samme men med et oppslagsverk
time_before_start = time.time()
for _ in range(trails):
    val = contains_key_value_pair_dict(d, "foo", 42)
time_when_done = time.time()
time_taken_dict = (time_when_done - time_before_start) * 1000
print(f"Tid for oppslag på ('foo', 42) med oppslagsverk:
{time_taken_dict:.0f}ms")
ratio = time_taken_list / time_taken_dict
print(f"For {n=} er oppslagsverk {ratio:.1f} ganger raskere enn lister")
print(f"Prøv med høyere verdi av n for å se større forskjeller")
```

Operatorer, funksjoner og metoder

Funksjoner og operatorer

↑ Innhold

```python
d = { 'a' : 1, 'b' : 2, 'c' : 3 }

print(len(d))       # Antall nøkler i oppslagsverket

print('a' in d)     # True
print(2 in d)       # False (in -operatoren sjekker kun nøkler)
print(2 not in d)   # True
print('a' not in d) # False
```

Metoder for å mutere oppslagsverk

```python
d = { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4}

# Fjern en nøkkel (og tilhørende verdi)
d.pop('d')            # Fjerner nøkkelen uten å bry seg om verdien
value = d.pop('b')    # Fjerner nøkkelen og tar vare på verdien
print(value)            # 2
print(d)                # {'a': 1, 'c': 3}

# Fjern en nøkkel og returner default-verdi dersom nøkkelen ikke finnes
print(d.pop('x', 42))   # 42
print(d.pop('c', 42))   # 3
print(d)                # {'a': 1}

# Legg til en nøkkel eller endre dens verdi
d['e'] = 5
d.update({'f': 6})
print(d) # {'a': 1, 'e': 5, 'f': 6}

# Slå sammen to oppslagsverk
d2 = {'f': 106, 'g': 107}
d.update(d2)
print(f'{d=}')  # d bli mutert. Verdier fra d2 overskriver verdier fra d.
print(f'{d2=}') # d2 er urørt
```

Løkker over oppslagsverk

↑ Innhold

```python
d = {'foo': 42, 'bar': 25, 'baz': 95}

# Løkke over nøklene (foo bar baz)
for key in d:
    print(key, end=' ')
print()

# Alternativ løkker over nøklene (foo bar baz)
for key in d.keys():
    print(key, end=' ')
print()

# Løkke over kun verdiene (42, 25, 95)
for value in d.values():
    print(value, end=' ')
print()

# Løkke over både nøkler og verdier (foo:42 bar:25 baz:95)
for key in d:
    value = d[key]
    print(f'{key}:{value}', end=' ')
print()

# Alternativ løkke over både nøkler og verdier (foo:42 bar:25 baz:95)
for key, value in d.items():
    print(f'{key}:{value}', end=' ')
print()
```

Tilfeldighet

↑ Innhold

- Tilfeldige verdier
- Frø
- Flere muligheter

Tilfeldige verdier

Python kan brukes for å generere «pseudo-tilfeldige» verdier. Med dette mener vi verdier som fremstår som tilfeldige, men som egentlig er generert basert på tidspunktet programmet starter. Om du startet det samme programmet på to ulike maskiner *akkurat* samtidig, ville begge programmene altså generert den samme verdien.

```
import random

x = random.random() # et flyttall slik at 0.0 <= x < 1.0
print(x)
```

```
import random

x = random.randrange(10) # et heltall slik at 0 <= x < 10
print(x)
```

```
import random

dice_roll = random.randrange(1, 7) # et heltall slik at 1 <= x < 7
print(dice_roll)
```

```
import random

character = random.choice('abcdef') # en bokstav fra strengen
print(character)
```

```
import random

options = ['heads', 'tails']
coin_flip = random.choice(options) # et element i listen
print(coin_flip)
```

↑ Innhold

Frø

Når man skal feilsøke noe, er det en fordel om programmet oppfører seg helt likt hver gang man kjører programmet. Introduksjonen av tilfeldige verdier gjør dette i utgangspunktet vanskelig; men siden verdiene er *pseudo*-tilfeldige, finnes det en løsning. For å late som vi starter programmet på akkurat samme tidspunkt hver gang det kjøres (slik at det alltid blir de samme «tilfeldige» verdiene som blir generert), kan vi manuelt sette «frøet» (engelsk: seed) til tilfeldighetsgeneratoren. Det er denne verdien som ellers ville blitt bestemt av starttidspunktet.

```
import random
random.seed(42) # vi setter frøet til 42 (kan være hva som helst)

x = random.random()
print(x)

x = random.random()
print(x)
```

Selv om vi får ulike verdien innad i én kjøring av programmet, vil det være de samme verdiene som genereres hver gang man kjører programmet på nytt. Prøv å endre frøet i programmet over; observer at du får de samme verdiene fra generatoren så lenge frøet er likt.

Flere muligheter

Random-modulen har massevis av mer avanserte muligheter også, for eksempel å velge tilfeldige tall fra en rekke ulike fordelinger (uniform, normal, gammavariat, etc.). Det finnes også funksjoner for å velge flere tilfeldige elementer fra en samling med og uten tilbakelegging (henholdsvis random.choices og random.sample), samt for å blande en samling (med random.shuffle). Se offisiell dokumentasjon (https://docs.python.org/3/library/random.html) for mer informasjon.

Flerdimensjonale lister

↑ Innhold

- Opprette 2D-lister og indeksering
- Opprette 2D-lister med dynamisk størrelse
- Dimensjonene til en 2D-liste
- Løkker over 2D-lister
- Hente ut rader og koloner
- Grunne og dype kopier
- 3D-lister

Opprette 2D-lister og indeksering

```
# Opprett en 2D-liste med gitte verdier (statisk opprettelse)
a = [[2, 3, 4], [5, 6, 7]]
print(a)
print(a[0])     # Elment nr 0 i a er en vanlig 1D-liste    --> [2, 3, 4]
print(a[0][1])  # Subliste nr 0, element nr 1 i sublisten  -->  3
```

```
# Opprett en 2D-liste
my_table = [
    ['a', 'b', 'c', 'd', 'e'],
    ['f', 'g', 'h', 'i', 'j'],
    ['k', 'l', 'm', 'n', 'o'],
    ['p', 'q', 'r', 's', 't'],
]

# Indeksering med to indekser; første indeks -> rad, andre indeks -> kolonne
print(my_table[0][0]) # a
print(my_table[0][1]) # b
print(my_table[0][2]) # c
print()
print(my_table[1][0]) # f
print(my_table[2][0]) # k
print(my_table[3][0]) # p
print()
print(my_table[1][4]) # gjett selv før du sjekker
print(my_table[3][2]) # gjett selv før du sjekker
```

Opprette 2D-lister med dynamisk størrelse

Vær forsiktig når du oppretter 2D-lister med dynamisk størrelse. Det er lett å gjøre feil. Se eksempelet under. For å få en god forståelse for *hvorfor* det blir feil her, er det lurt å benytte «se steg» -knappen.

↑ Innhold

```
# MISLYKKET forsøk på å opprette 2D-liste med dynamisk størrelse
rows = 3
cols = 2

a = [[0] * cols] * rows # Oppretter en «grunn» (overfladisk) kopi
                        # Oppretter én unik rad, resten er aliaser

print("Dette VIRKER SOM det er ok.  I begynnelsen:")
print("   a =", a) # [[0, 0], [0, 0], [0, 0]]

a[0][0] = 42
print("MEN, hva skjer etter at vi muterer a[0][0]=42?")
print("   a =", a) # [[42, 0], [42, 0], [42, 0]]
```

```
# RIKTIG måte å opprette en 2D-liste med dynamisk størrelse
rows = 3
cols = 2

a = []
for _ in range(rows):
    a.append([0] * cols)

print("Dette ER ok.  I begynnelsen:")
print("   a =", a) # [[0, 0], [0, 0], [0, 0]]

a[0][0] = 42
print("Etter at vi muterer a[0][0]=42")
print("   a =", a) # [[42, 0], [0, 0], [0, 0]]
```

```
# Alternativ RIKTIG måte å opprette en 2D-liste med dynamisk størrelse
# Her benytter vi listebygging (engelsk: list comprehension)
rows = 3
cols = 2

a = [[0] * cols for _ in range(rows)]

print("Dette ER ok.  I begynnelsen:")
print("   a =", a) # [[0, 0], [0, 0], [0, 0]]

a[0][0] = 42
print("Etter at vi muterer a[0][0]=42")
print("   a =", a) # [[42, 0], [0, 0], [0, 0]]
```

Dimensjonene til en 2D-liste

En 2D-liste er en liste av lister. Selv om det ikke er påkrevd fra Python sin side, er det vanlig at alle de «innerste» listene er like lange; da kaller vi listen for et *rutenett*. Hver av de innerste listene representerer én

rad, mens den ytterste listen inneholder alle radene.

↑ Innhold

```
# a er 2D-liste som representerer et rutenett (alle rader er like lange)
a = [
    [2, 3, 5],
    [1, 4, 7],
]
print("a = ", a)

# La oss finne dimensjonene til rutenettet
rows = len(a)
cols = len(a[0])
print("rows =", rows) # 2
print("cols =", cols) # 3
```

Løkker over 2D-lister

```
# a er 2D-liste som representerer et rutenett (alle rader er like lange)
a = [
    [2, 3, 5],
    [1, 4, 7],
]
print("Først: a =", a)

# Vi finner dimensjonene til rutenettet
rows = len(a) # 2
cols = len(a[0]) # 3

# En løkke over hvert element i listen
# I eksempelet under øker vi verdien til hver celle i rutenettet med 1
for row in range(rows):
    for col in range(cols):
        # Koden her inne kjøres rows*cols ganger, én gang for hver
        # kombinasjon av verdier for row og col (altså én gang for hver «celle»)
        a[row][col] += 1

# Til slutt, utskrift av resultatet
print("Etter:  a =", a)
```

Hente ut rader og kolonner

Få tilgang til en hel rad.

↑ Innhold

```
# Et alias, ikke en kopi! Ingen ny liste opprettes!
a = [[2, 3, 5], [1, 4, 7]]
row = 1
row_list = a[row]
print(row_list)
```

Få tilgang til en hel kolonne.

```
# IKKE et alias, men en kopi! Ny liste opprettes!
a = [[2, 3, 5], [1, 4, 7]]
col = 1
col_list = []
for row in range(len(a)):
    col_list.append(a[row][col])
print(col_list)
```

Få tilgang til en hel kolonne med listebygging.

```
# IKKE et alias, men en kopi! Ny liste opprettes!
a = [[2, 3, 5], [1, 4, 7]]
col = 1
col_list = [a[row][col] for row in range(len(a))]
print(col_list)
```

Grunne og dype kopier

Først, et mislykket forsøk på å kopiere en 2D-liste. Du vil (dessverre) få samme oppførsel uansett hvilken variant av kopiering fra kursnotatene om 1D-lister du bruker.

Problemet med å lage en kopi av flerdimensjonell liste, er at innholdet i lister egentlig er *referanser* til verdier, og ikke er verdier i seg selv. Så om du kopierer en referanse - da får vi et alias.

```
# MISLYKKET forsøk på å kopiere en 2D-liste
import copy
a = [[1, 2, 3], [4, 5, 6]]

b = copy.copy(a) # Fra kursnotatene om 1D-lister

print("Dette VIRKER SOM det er ok.  I begynnelsen:")
print("   a =", a)
print("   b =", b)

a[0][0] = 42
print("MEN, hva skjer etter at vi muterer a[0][0]=42?")
print("   a =", a)
print("   b =", b)
```

↑ Innhold

```python
# RIKTIG måte å kopiere en 2D-liste
import copy
a = [[1, 2, 3], [4, 5, 6]]

b = copy.deepcopy(a)

print("Dette ER er ok.  I begynnelsen:")
print("   a =", a)
print("   b =", b)

a[0][0] = 42
print("Etter at vi muterer a[0][0]=42")
print("   a =", a)
print("   b =", b)
```

3D-lister

En 2D-liste er bare en liste av lister. Det er selvsagt mulig å ha en liste av 2D-lister. Dette blir da en 3D-liste. En liste med 3D-lister blir en 4D-liste, og så videre.

```python
a = [
        [
            [1, 2],
            [3, 4],
        ],
        [
            [5, 6, 7],
            [8, 9],
        ],
        [
            [10],
        ],
    ]

for i in range(len(a)):
    for j in range(len(a[i])):
        for k in range(len(a[i][j])):
            print(f'a[{i}][{j}][{k}] = {a[i][j][k]}')
```

Grafiske brukergrensesnitt

↑ Innhold

- Første eksempel: tell antall tastetrykk
- Model-View-Controller
- Identifisering av tastetrykk
- Flytte en prikk med piltastene
- Flytte en prikk med museklikk
- Flytte en prikk med timer
- Endre hastighet for timer
- Sette timer på pause
- Brudd med MVC
- Bilder

- Eksempel: legg til og fjern prikker
- Eksempel: sprettende figur
- Eksempel: museklikk i rutenett
- Eksempel: knapper

- Oversikt over kontroller-funksjoner
- Appendiks: enda flere muligheter med uib_inf100_graphics

Et *grafisk brukergrensesnitt* er et dataprogram som lar brukeren interagere med programmet ved å klikke på knapper, benytte tastaturet, flytte på musen eller andre handlinger uten å primært forholde seg til terminalen.

> *Det finnes flere ulike rammeverk som tilbyr funksjonalitet for å lage grafiske brukergrensesnitt. I dette emnet skal vi benytte oss av* `uib_inf100_graphics`*, som er en forenklet versjon av rammeverket* `tkinter` *som er en del av standardbiblioteket i Python. Om du har fulgt emnet har du allerede installert rammeverket på din datamaskin (se kursnotatene om grafikk for instruksjoner om installasjon).*
>
> *Vi beveger oss nå videre fra subpakken «simple» som vi lærte om tidligere, og skal i stedet benytte subpakken «event_app» som gir oss mulighet til å lage interaktive grafiske applikasjoner. Selve funksjonene for å tegne på canvas vil fungere på samme måte som før; forskjellen er at vi nå også kan skrive kode som reagerer på brukerens handlinger.*

↑ Innhold

Første eksempel: tell antall tastetrykk

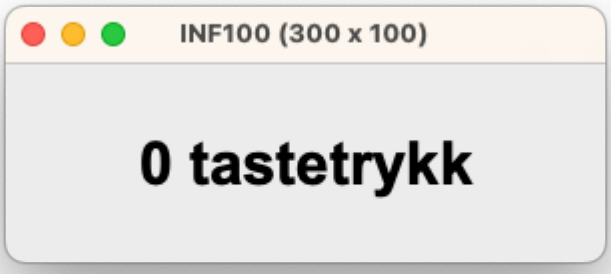
```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    # app_started: kjøres én gang når programmet starter.
    # Her oppretter vi variabler i `app`` og gir dem initiell verdi.
    app.counter = 0

def key_pressed(app, event):
    # key_pressed: kjøres hver gang en tast trykkes.
    # Vi kan endre variabler i `app` her.
    app.counter += 1

def redraw_all(app, canvas):
    # redraw_all: kode for å tegne noe på skjermen. Kjøres vanligvis
    # flere ganger i sekundet.
    # Vi kan benytte (se på) variablene i `app` her, men ikke endre dem.
    canvas.create_text(
        app.width/2, app.height/2,
        text=f'{app.counter} tastetrykk',
        font='Arial 30 bold'
    )

run_app(width=300, height=100)
```



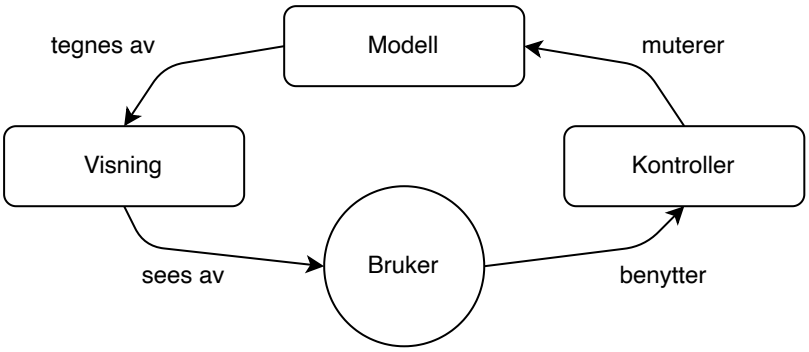
Model-View-Controller

Når man skriver programmer med grafiske brukergrensesnitt, kan koden fort bli rotete og uoversiktelig. For å hjelpe oss å skrive oversiktelig kode det er mulig å feilsøke, benytter vi oss av et prinssipp som kalles *model-view-controller* (MVC). I dette paradigmet er det tre sentrale begreper:

- **Modell.** En modell er en samling med variabler og data som representerer tilstanden til programmet. I eksempelet over er objektet `app` modellen, og funksjonen `app_started` er ansvarlig for å opprette variablene i den.
- **Visning.** Funksjoner for å tegne noe på skjermen, fortrinnsvis basert på variablene og dataen i modellen. I eksempelet over er det funksjonen `redraw_all` som er visningen.

↑ Innhold

- **Kontroller.** Funksjoner som responderer på tastetrykk, museklikk, klokkeslag/timer eller andre hendelser og oppdaterer modellen på bakgrunn av dette. I eksempelet over er funksjonen `key_pressed` en kontroller, men det finnes også mange andre (for eksempel *mouse_pressed* og *timer_fired* som introduseres litt senere).



I vårt MVC-baserte rammeverk `uib_inf100_graphics` må koden vi skriver forholde seg til følgende kjøreregler:

- Aldri gjør et kall til en kontroller-funksjon (f. eks. *key_pressed, mouse_pressed, timer_fired*) eller til *redraw_all* på egen hånd. Rammeverket gjør dette for deg automatisk. I eksempelet over, legg merke til at det eneste funksjonskallet vi gjør selv er til `run_app`.
- Kontroller-funksjonene skal kun oppdatere modellen (*app*), de skal *ikke* oppdatere visningen.
- Visningen skal kun tegne ting på skjermen, den skal *ikke* endre på noe i modellen (*app*).
- Variabler i modellen (*app*) opprettes første gang i funksjonen *app_started*.

Dersom du bryter noen av disse reglene, kalles det *brudd med MVC* (engelsk: MVC violation). Hvis rammeverket vårt oppdager et slikt brudd, vil det umiddelbart stoppe programmet og vise en feilmelding.

> *PS: Bruk av rammeverket vårt hjelper deg å følge MVC, men gir ingen garantier. Med andre ord, det er fullt mulig å bryte MVC med vårt rammeverk uten å få en feilmelding. Å bryte MVC er dårlig stil og gjør koden din uoversiktelig og vanskelig å feilsøke; så ikke gjør det selv om det teknisk sett er mulig.*

Identifisering av tastetrykk

Tastetrykk kan være forskjellige. Vi kan se hvilken tast som ble trykket ved å se på verdien `event.key` som er en streng som beskriver tasten. Under er et program som lar oss se hvilken streng dette er når vi trykker på en tast.

```python
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.message = 'Press any key'

def key_pressed(app, event):
    app.message = f"event.key == '{event.key}'"

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 40, text=app.message,
                       font='Arial 20 bold')

    key_names_text = '''\
        Here are the legal event.key names:
        * Keyboard key labels (letters, digits, punctuation)
        * Arrow directions ('Up', 'Down', 'Left', 'Right')
        * Whitespace ('Space', 'Enter', 'Tab', 'BackSpace')
        * Other commands ('Delete', 'Escape')'''

    canvas.create_text(
        app.width/2, 80,
        text=key_names_text,
        anchor="n",
        font='Arial 16'
    )

run_app(width=500, height=250)
```



Flytte en prikk med piltastene

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def key_pressed(app, event):
    if (event.key == 'Left'):
        app.cx -= 10
    elif (event.key == 'Right'):
        app.cx += 10

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Flytt med piltaster høyre/venstre')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```



↑ Innhold

```python
# I denne versjonen kan ikke ballen flytte seg ut av lerretet

from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def key_pressed(app, event):
    if (event.key == 'Left'):
        app.cx -= 10
        if (app.cx - app.r < 0):
          app.cx = app.r
    elif (event.key == 'Right'):
        app.cx += 10
        if (app.cx + app.r > app.width):
          app.cx = app.width - app.r

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Flytt med piltaster høyre/venstre')
    canvas.create_text(app.width/2, 40,
                       text='Ballen kan ikke kan flytte seg ut av vinduet')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

↑ Innhold

```python
# I denne versjonen kommer ballen tilbake på motsatt side

from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def key_pressed(app, event):
    if (event.key == 'Left'):
        app.cx -= 10
        if (app.cx + app.r <= 0):
          app.cx = app.width + app.r
    elif (event.key == 'Right'):
        app.cx += 10
        if (app.cx - app.r >= app.width):
          app.cx = 0 - app.r

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Flytt med piltaster høyre/venstre')
    canvas.create_text(app.width/2, 40,
                       text='Ballen kommer rundt på motsatt side')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

↑ Innhold

```
# I denne versjonen kan ballen bevege seg i to dimensjoner

from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def key_pressed(app, event):
    if (event.key == 'Left'):    app.cx -= 10
    elif (event.key == 'Right'): app.cx += 10
    elif (event.key == 'Up'):    app.cy -= 10
    elif (event.key == 'Down'):  app.cy += 10

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Flytt med piltaster høyre/venstre/opp/ned')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

Flytte en prikk med museklikk

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def mouse_pressed(app, event):
    app.cx = event.x
    app.cy = event.y

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Flytt ved å klikke med musen')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

↑ Innhold

Flytte en prikk med timer

Funksjonen `timer_fired` demonstrert her regnes som en kontroller, selv om det ikke strengt tatt er brukererens handling som gjør at metoden kalles; i stedet er det rammeverket *uib_inf100_graphics* selv som «opptrer som en bruker» ved å periodisk kalle denne funksjonen med et fast intervall.

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2
    app.r = 40

def timer_fired(app):
    app.cx -= 10
    if (app.cx + app.r <= 0):
        app.cx = app.width + app.r

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Prikken flytter seg automatisk')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

Endre hastighet for timer

Som standard kalles funksjonen `timer_fired` med et intervall på 100 millisekunder (dvs. 10 ganger i sekundet). Vi kan endre dette ved å endre på variabelen `app.timer_delay`. Vi kan endre variabelens verdi i `app_started` eller (for å dynamisk endre hastigheten) i en kontroller-funksjon.

↑ Innhold

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.timer_delay = 128 # milliseconds
    app.cx = app.width/2
    app.cy = app.height/2 + 15
    app.r = 40

def key_pressed(app, event):
    if event.key == "Up":
        app.timer_delay *= 2
        app.timer_delay = max(app.timer_delay, 1)
    elif event.key == "Down":
        app.timer_delay //= 2

def timer_fired(app):
    app.cx -= 10
    if (app.cx + app.r <= 0):
        app.cx = app.width + app.r

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text=f"{app.timer_delay=}")
    canvas.create_text(app.width/2, 40,
                       text=f"Trykk pil opp/ned for å doble/halvere delay")
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

Legg merke til at hastigheten på animasjonen ikke endres vesentlig når vi kommer ned til delay-verdier i nærheten av 0. Det er fordi det på en vanlig datamaskin med moderne spesifikasjoner fremdeles tar et par millisekunder å faktisk tegne skjermbildet, og da betyr ventetiden vi har mellom hvert kall mindre og mindre.

↑ Innhold

Timeren i `uib_inf100_graphics` fungerer omtrent slik: først kalles `timer_fired`, og umiddelbart etter kallet er ferdig, kalles `redraw_all`. Når kallet til redraw_all er ferdig, venter timeren i `app.timer_delay` millisekunder, og begynner deretter på nytt. Tiden det tar mellom hvert nye kall til timer_fired kan derfor grovt sett regnes ut som

- *tiden det tar å kalle timer_fired, pluss*
- *tiden det tar å kalle redraw_all, pluss*
- *antall millisekunder definert i app.timer_delay.*

Ventetiden kan også påvirkes av andre forhold, slik som prosessorbelastningen din datamaskin er utsatt for av andre kontroller-funksjoner eller til og med av andre programmer som kjøres samtidig på datamaskinen.

Sette timer på pause

Nyttig for feilsøking av animasjoner!

↑ Innhold

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.cx = app.width/2
    app.cy = app.height/2 + 15
    app.r = 40
    app.paused = False

def timer_fired(app):
    if not app.paused:
        do_step(app)

def do_step(app):
    app.cx -= 10
    if (app.cx + app.r <= 0):
        app.cx = app.width + app.r

def key_pressed(app, event):
    if event.key == 'p':
        app.paused = not app.paused
    elif event.key == 'Space' and app.paused:
        do_step(app)

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Prikken flytter seg automatisk')
    canvas.create_text(app.width/2, 40,
                       text='Trykk p for å sette på pause')
    canvas.create_text(app.width/2, 60,
                       text='Trykk mellomrom for å ta steg i pausen')
    canvas.create_oval(app.cx-app.r, app.cy-app.r,
                       app.cx+app.r, app.cy+app.r,
                       fill='lightblue')

run_app(width=300, height=200)
```

Brudd med MVC

Vi kan ikke endre modellen i `redraw_all`.

↑ Innhold

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.x = 42

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Et brudd med MVC')

    app.x = 10 # Her er bruddet! Ikke lov å endre modellen i visningen

run_app(width=300, height=200)
```

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.x = [42, 43]

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Også et brudd med MVC')

    app.x[0] = 99 # Her er bruddet! Ikke lov å mutere noe i modellen her

run_app(width=300, height=200)
```

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.x = [42, 43]

def key_pressed(app, event):
    mutany(app) # Det er ikke MVC-brudd når mutany kalles fra en kontroller

def mutany(app):
    app.x.append(42) # Her skjer selve MVC-bruddet

def redraw_all(app, canvas):
    canvas.create_text(app.width/2, 20,
                       text='Enda et brudd med MVC!')
    canvas.create_text(app.width/2, 40,
                       text='Trykk på en tast et par ganger')
    canvas.create_text(app.width/2, 60,
                       text=f'{app.x=}')

    if len(app.x) == 5:
        mutany(app) # Under dette kallet skjer det et MVC-brudd

run_app(width=300, height=200)
```

Legg merke til at feilmeldingen (se under) ikke spesifiserer i hvilken funksjon bruddet skjer (nemlig i *mutany* -funksjonen), bare at det skjedde under utførelsen av *redraw_all*. Når du får en slik feilmelding, må du altså også undersøke at ingen av hjelpefunksjonene som benyttes muterer modellen.

```
Traceback (most recent call last):
    No traceback available. Error occurred in redraw_all.
Exception: MVC Violation: you may not change the app state (the model) in
redraw_all (the view)
```

Det er noen brudd med MVC rammeverket vårt ikke vil avsløre, men som likevel er brudd med MVC som *må* unngås for å beholde god kodestil. For eksempel:

- Skrive til en global variabel i visningen (redraw_all), eller
- Skrive til en ekstern ressurs i visningen (redraw_all): for eksempel bruk av filer eller kommunikasjon med andre prosesseser eller nettverkskommunikasjon. Alt slikt skal skje i kontroller-funksjoner for å unngå brudd med MVC.

Bilder

Du kan benytte alle de samme metodene og hjelpefunksjonene for å laste og vise bilder som vi kjenner fra kursnotatene om grafikk, *men*:

- **Ikke last inn bildet i *redraw_all*.** Dette vil føre til at bildet lastes inn på nytt hver gang skjermen tegnes på nytt; fordi lasting av bilder (både fra fil og fra internett) er svært tidkrevende, vil dette føre til at

programmet ditt blir tregt og lite responsivt.

↑ Innhold

I stedet bør du laste inn alle bildene du trenger i *app_started*. Da vil bildene lastes inn én gang, og deretter kan du bruke dem som du vil i redraw_all.

```
from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http

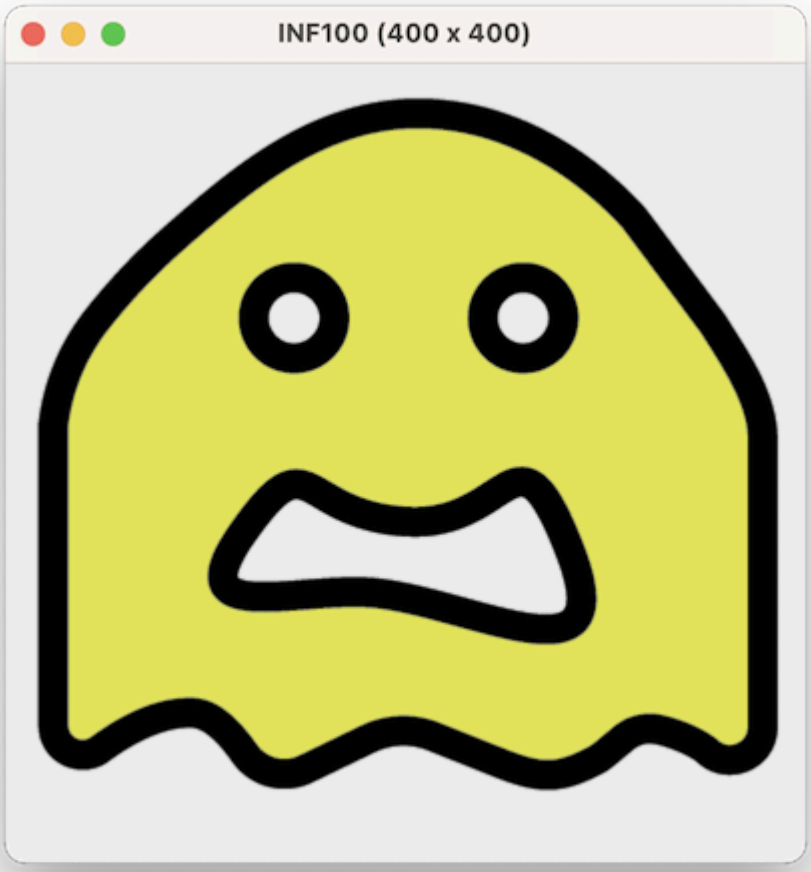
def app_started(app):
    # Laster alle bilder vi kan tenke oss å bruke
    app.image_yellow = load_image_http('https://tinyurl.com/inf100yellowghost')
    app.image_black = load_image_http('https://tinyurl.com/inf100blackghost')

    # Selve modellen
    app.use_yellow_image = True

def key_pressed(app, event):
    app.use_yellow_image = not app.use_yellow_image

def redraw_all(app, canvas):
    image = app.image_yellow if app.use_yellow_image else app.image_black
    canvas.create_image(
        app.width / 2,
        app.height / 2,
        pil_image=image,
    )

run_app(width=400, height=400)
```


INF100 (400 x 400)

↑ Innhold

Eksempel: legg til og fjern prikker

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.circle_centers = [ ]

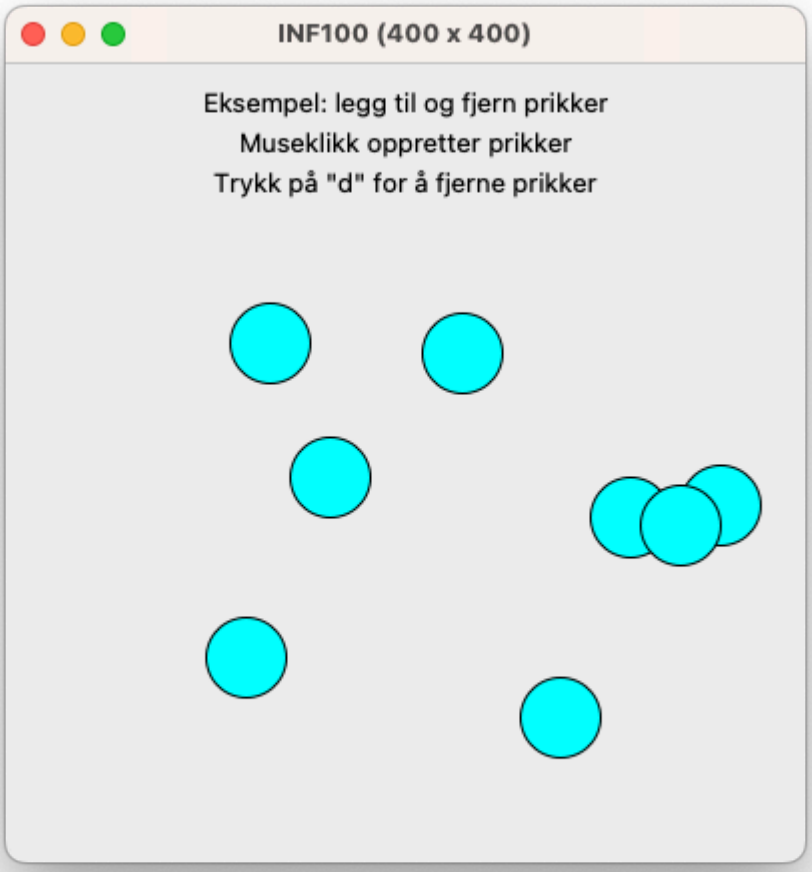
def mouse_pressed(app, event):
    new_circle_center = (event.x, event.y)
    app.circle_centers.append(new_circle_center)

def key_pressed(app, event):
    if (event.key == 'd'):
        if (len(app.circle_centers) > 0):
            app.circle_centers.pop(0)
        else:
            print('Ingen flere prikker å fjerne!')

def redraw_all(app, canvas):
    # tegn prikkene
    for circle_center in app.circle_centers:
        (cx, cy) = circle_center
        r = 20
        canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill='cyan')
    # tegn teksten
    canvas.create_text(app.width/2, 20,
                       text='Eksempel: legg til og fjern prikker')
    canvas.create_text(app.width/2, 40,
                       text='Museklikk oppretter prikker')
    canvas.create_text(app.width/2, 60,
                       text='Trykk på "d" for å fjerne prikker')

run_app(width=400, height=400)
```

↑ Innhold


INF100 (400 x 400)
Eksempel: legg til og fjern prikker
Museklikk oppretter prikker
Trykk på "d" for å fjerne prikker

Eksempel: sprettende figur

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.square_left = app.width//2
    app.square_top = app.height//2
    app.square_size = 25
    app.dx = -4
    app.dy = 5
    app.is_paused = False
    app.timer_delay = 25 # millisekunder

def key_pressed(app, event):
    if event.key == "p":
        app.is_paused = not app.is_paused
    elif event.key == "s":
        do_step(app)

def timer_fired(app):
    if not app.is_paused:
        do_step(app)

def do_step(app):
    # Flytt horisontalt
    app.square_left += app.dx

    # Sjekk om firkanten har gått utenfor lerretet, og hvis ja, snu
    # retning; men flytt også firkanten til kanten (i stedet for å gå
    # forbi). Merk: det finnes andre, mer sofistikerte måter å håndtere
    # at rektangelet går forbi kanten...
    if app.square_left < 0:
        # snu retningen!
        app.square_left = 0
        app.dx = -app.dx
    elif app.square_left > app.width - app.square_size:
        app.square_left = app.width - app.square_size
        app.dx = -app.dx

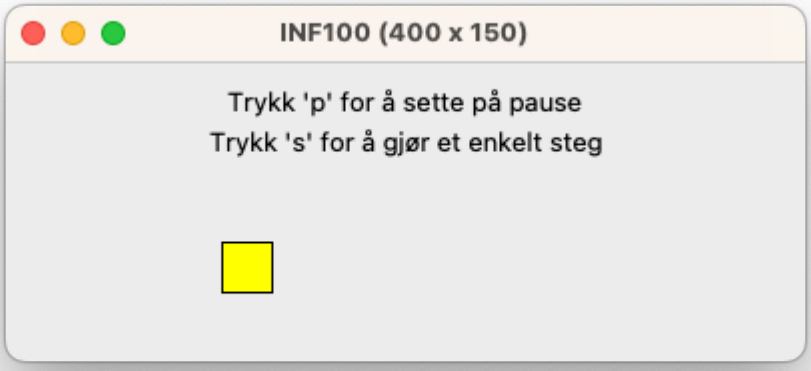
    # Flytt vertikalt på samme måte
    app.square_top += app.dy
    if app.square_top < 0:
        # snu retningen!
        app.square_top = 0
        app.dy = -app.dy
    elif app.square_top > app.height - app.square_size:
        app.square_top = app.height - app.square_size
        app.dy = -app.dy

def redraw_all(app, canvas):
    # tegn firkanten
    canvas.create_rectangle(
```

↑ Innhold

```
        app.square_left,
        app.square_top,
        app.square_left + app.square_size,
        app.square_top + app.square_size,
        fill="yellow",
    )
    # tegn teksten
    canvas.create_text(
        app.width/2, 20,
        text="Trykk 'p' for å sette på pause",
    )
    canvas.create_text(
        app.width/2, 40,
        text="Trykk 's' for å gjør et enkelt steg",
    )

run_app(width=400, height=150)
```



↑ Innhold

Eksempel: museklikk i rutenett

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.rows = 5
    app.cols = 8
    app.margin = 50 # margin rundt rutenettet
    app.selection = (-1, -1) # (row, col) for valgt rute, (-1,-1) for ingen

def point_in_grid(app, x, y):
    # returner True hvis piksel-koordinatet (x, y) er på innsiden av
    # rutenettet slik det blir tegnet i visningen.
    return ((app.margin <= x <= app.width-app.margin) and
            (app.margin <= y <= app.height-app.margin))

def get_cell(app, x, y):
    # "visning-til-modell"
    # returnerer (row, col) for ruten hvor piksel-koordnatet (x, y) hører
    # hjemme, eller (-1, -1) hvis koodinatet er utenfor rutenettet
    if (not point_in_grid(app, x, y)):
        return (-1, -1)
    grid_width  = app.width - 2*app.margin
    grid_height = app.height - 2*app.margin
    cell_width  = grid_width / app.cols
    cell_height = grid_height / app.rows

    # Merk: vi trenger å konvertere til int her; det er ikke
    # tilstrekkelig å benytte //, siden x, y, eller app.margin kan
    # være flyttall, og da vil også // returnere flyttall
    row = int((y - app.margin) / cell_height)
    col = int((x - app.margin) / cell_width)

    return (row, col)

def get_cell_bounds(app, row, col):
    # "modell-til-visning"
    # returnerer (x0, y0, x1, y1), piksel-koordinater for hjørnene til
    # den gitte ruten
    grid_width  = app.width - 2*app.margin
    grid_height = app.height - 2*app.margin
    column_width = grid_width / app.cols
    row_height = grid_height / app.rows
    x0 = app.margin + col * column_width
    x1 = app.margin + (col+1) * column_width
    y0 = app.margin + row * row_height
    y1 = app.margin + (row+1) * row_height
    return (x0, y0, x1, y1)

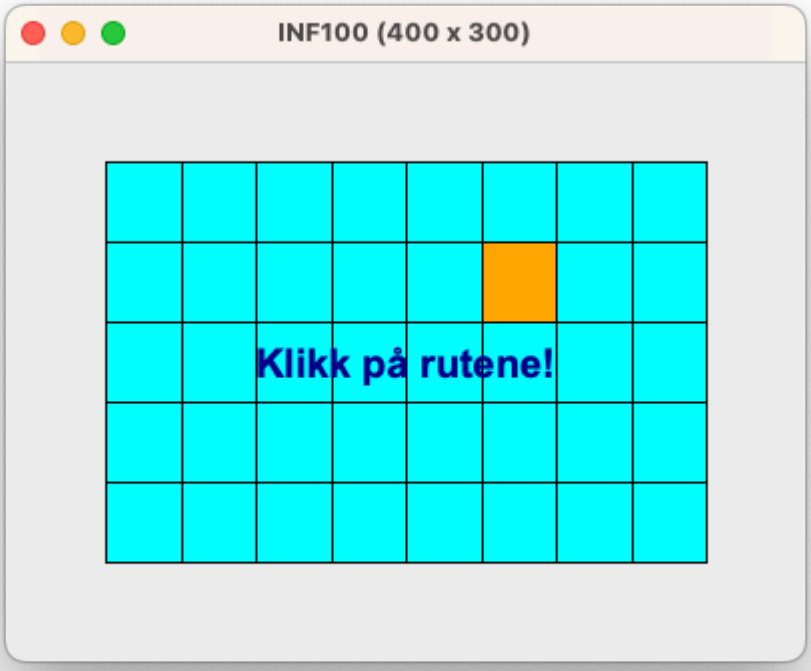
def mouse_pressed(app, event):
    (row, col) = get_cell(app, event.x, event.y)
    # velg denne ruten med mindre den allerede er valgt
```

↑ Innhold

```
    if (app.selection == (row, col)):
        app.selection = (-1, -1)
    else:
        app.selection = (row, col)

def redraw_all(app, canvas):
    # tegn alle rutene
    for row in range(app.rows):
        for col in range(app.cols):
            (x0, y0, x1, y1) = get_cell_bounds(app, row, col)
            fill = "orange" if (app.selection == (row, col)) else "cyan"
            canvas.create_rectangle(x0, y0, x1, y1, fill=fill)
    canvas.create_text(app.width/2, app.height/2, text="Klikk på rutene!",
                       font="Arial 20 bold", fill="darkBlue")

run_app(width=400, height=300)
```



↑ Innhold

Eksempel: knapper

```
from uib_inf100_graphics.event_app import run_app

##############
## Modellen ##
##############

def app_started(app):
    app.count = 0
    app.buttons = [
        # [x1, y1, x2, y2, "Navn på knapp", funksjon]
        [30, 30, 130, 60, "Opp", increase],
        [150, 30, 250, 60, "Ned", decrease]
    ]

#################
## Kontrollere ##
#################

def increase(app):
    app.count += 1

def decrease(app):
    app.count -= 1

def point_in_rectangle(x1, y1, x2, y2, x, y):
    return (min(x1, x2) <= x <= max(x1, x2)
        and min(y1, y2) <= y <= max(y1, y2))

def execute_button_action_if_clicked(app, button, mouse_x, mouse_y):
    x1, y1, x2, y2, label, func = button
    if point_in_rectangle(x1, y1, x2, y2, mouse_x, mouse_y):
        func(app)

def mouse_pressed(app, event):
    for button in app.buttons:
        execute_button_action_if_clicked(app, button, event.x, event.y)

#############
## Visning ##
#############

def redraw_all(app, canvas):
    # tegn knappene
    for button in app.buttons:
        draw_button(canvas, button)
    # tegn telleren
    canvas.create_text(app.width/2, app.height*2/3, text=f"{app.count}",
                                                    font="Arial 20")

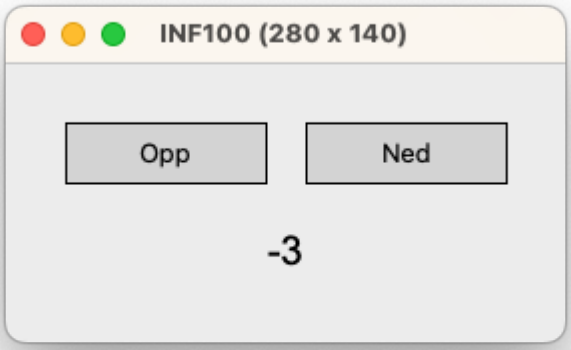
def draw_button(canvas, button):
```

↑ Innhold

```
    x1, y1, x2, y2, label, func = button
    canvas.create_rectangle(x1, y1, x2, y2, fill="lightgray")
    mid_x = (x1 + x2) / 2
    mid_y = (y1 + y2) / 2
    canvas.create_text(mid_x, mid_y, text=label)

####################
## Kjør programmet ##
####################

run_app(width=280, height=140)
```



Oversikt over kontroller-funksjoner

Rammeverket *uib_inf100_graphics.event_app* gir spesiell betydning til følgende funksjoner hvis de er definert i samme fil som *run_app* kalles i:

- `redraw_all(app, canvas)` en spesiell funksjon som kalles når skjermen skal tegnes på nytt. Denne funksjonen er ikke en kontroller-funksjon, men representerer visningen. Det vil være et brudd med MVC å endre på modellen i denne funksjonen.

I tillegg til redraw_all, gis det spesiell betydning til følgende funksjoner, som er knyttet til «kontroller»-funksjonalitet i programmet. I disse funksjonene kan vi endre på modellen (variabler i `app`) basert på brukerens handlinger:

↑ Innhold

Kontroller-funksjon	Beskrivelse
`app_started(app)`	Kjøres én gang når programmet starter. Det er her man bør opprette variablene i modellen første gang og gi dem initiell verdi.
`app_stopped(app)`	Kjøres én gang når programmet avsluttes.
`key_pressed(app, event)`	Kjøres hver gang en tast trykkes ned. Hvilken tast som ble trykket finnes i `event.key`.
`key_released(app, event)`	Kjøres hver gang en tast slippes opp. Hvilken tast som ble sluppet finnes i `event.key`.
`mouse_pressed(app, event)`	Kjøres hver gang musen trykkes *ned*. Musen posisjon på lerretet finnes i `event.x` og `event.y`.
`mouse_released(app, event)`	Kjøres hver gang musen slippes *opp*. Musen posisjon på lerretet finnes i `event.x` og `event.y`.
`mouse_dragged(app, event)`	Kjøres hver gang musen flyttes mens museknappen er trykket ned. Musen posisjon på lerretet finnes i `event.x` og `event.y`.
`mouse_moved(app, event)`	Kjøres hver gang musen flyttes uten at noen museknapp er trykket ned. Musen posisjon på lerretet finnes i `event.x` og `event.y`.
`timer_fired(app)`	Kjøres med jevne mellomrom. Intervallet funksjonen kalles med er definert av `app.timer_delay` (oppgitt i antall millisekunder).

Appendiks

Fant du ikke det du letet etter? Du kan også se appendiks til grafiske brukergrensesnitt for flere eksempler og mer informasjon om hvordan du kan lage grafiske brukergrensesnitt i Python ved hjelp av `uib_inf100_graphics`.

Appendiks til grafiske brukergrensesnitt

↑ Innhold

> *Innholdet på denne siden vil ikke komme på eksamen. Vi publiserer det her slik at dem som ønsker får se flere muligheter som finnes i `uib_inf100_graphics.event_app` -rammeverket.*

Denne siden er appendiks til notatene om grafiske brukergrensesnitt.

- Innebygde tastaturbindinger
- Brukerhendelser (kontroller-funksjoner)
- Dialogbokser og popup
- Bilder
- Animerte bilder
- Modus (ulike skjermer)
- Scrolling
- Spille av lyder

Innebygde tastaturbindinger

```
from uib_inf100_graphics.event_app import run_app

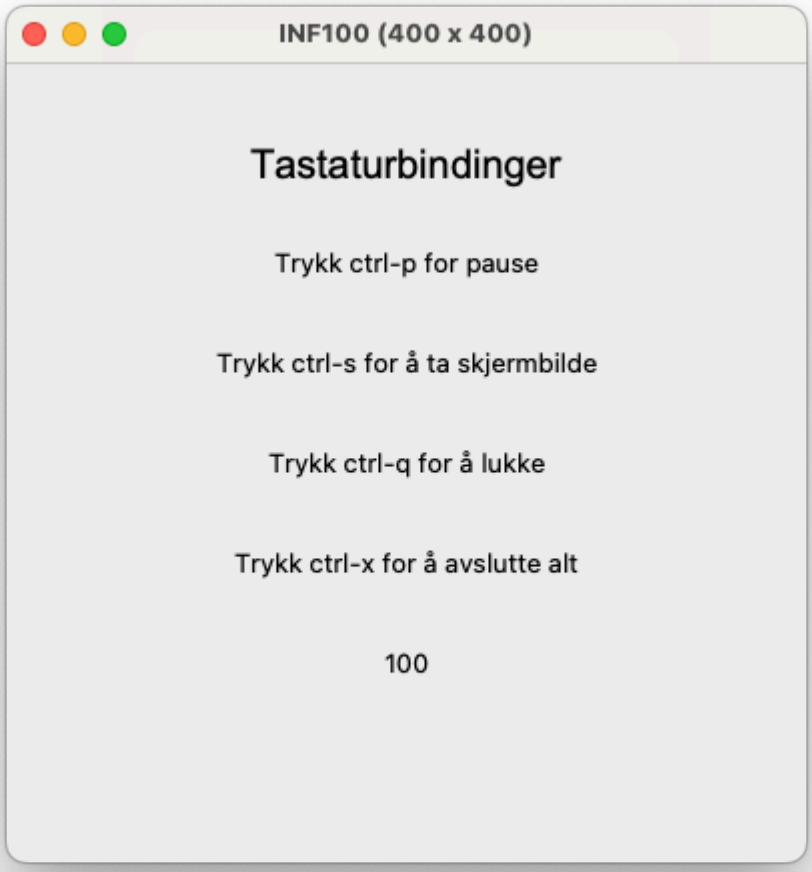
def app_started(app):
    app.counter = 0

def timer_fired(app):
    app.counter += 1

def redraw_all(app, canvas):
    canvas.create_text(200,  40, text='Tastaturbindinger', font='Arial 20')
    canvas.create_text(200,  80, text='Trykk ctrl-p for pause')
    canvas.create_text(200, 120, text='Trykk ctrl-s for å ta skjermbilde')
    canvas.create_text(200, 260, text='Trykk ctrl-q for å lukke')
    canvas.create_text(200, 200, text='Trykk ctrl-x for å avslutte alt')
    canvas.create_text(200, 240, text=f'{app.counter}')

run_app(width=400, height=300) # ctrl-q fortsetter videre, ctrl-x gjør ikke
run_app(width=600, height=400)
```

↑ Innhold


INF100 (400 x 400)
Tastaturbindinger
Trykk ctrl-p for pause
Trykk ctrl-s for å ta skjermbilde
Trykk ctrl-q for å lukke
Trykk ctrl-x for å avslutte alt
100

Brukerhendelser (kontroller-funksjoner)

```
from uib_inf100_graphics.event_app import run_app

def app_started(app):
    app.messages = ['app_started']

def app_stopped(app):
    app.messages.append('app_stopped')
    print('app_stopped!')

def key_pressed(app, event):
    app.messages.append('key_pressed: ' + event.key)

def key_released(app, event):
    app.messages.append('key_released: ' + event.key)

def mouse_pressed(app, event):
    app.messages.append(f'mouse_pressed {(event.x, event.y)}')

def mouse_released(app, event):
    app.messages.append(f'mouse_released {(event.x, event.y)}')

def mouse_moved(app, event):
    app.messages.append(f'mouse_moved {(event.x, event.y)}')

def mouse_dragged(app, event):
    app.messages.append(f'mouse_dragged {(event.x, event.y)}')

def size_changed(app):
    app.messages.append(f'size_changed {(app.width, app.height)}')

def redraw_all(app, canvas):
    font = 'Arial 20 bold'
    canvas.create_text(app.width/2, 30, text='Brukerhendelser', font=font)
    n = min(10, len(app.messages))
    i0 = len(app.messages) - n
    for i in range(i0, len(app.messages)):
        canvas.create_text(
            app.width / 2,
            100 + 50 * (i - i0),
            text=f'#{i}: {app.messages[i]}',
            font=font
        )

run_app(width=600, height=600)
```

Dialogbokser og popup

Rammeverket *uib_inf100_graphics* bygger på og er en forenklet versjon av *tkinter*. Dette gjør det er mulig å integrere komponenter som for eksempel dialogbokser (https://docs.python.org/3/library/dialog.html) og

meldingsbokser (https://docs.python.org/3/library/tkinter.messagebox.html) fra *tkinter*.

↑ Innhold

```
from uib_inf100_graphics.event_app import run_app
from tkinter import simpledialog, messagebox

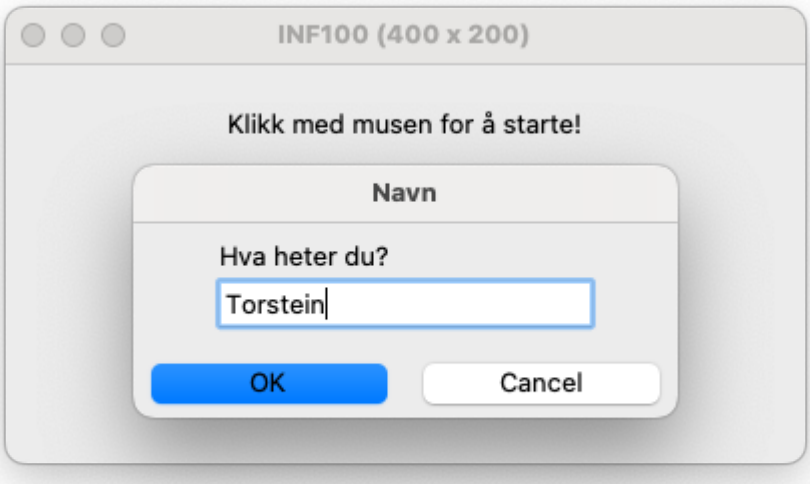
def app_started(app):
    app.message = 'Klikk med musen for å starte!'
    app.count = 0

def timer_fired(app):
    app.count += 1

def mouse_pressed(app, event):
    name = simpledialog.askstring('Navn', 'Hva heter du?', parent=app._root)
    if name is None:
        app.message = 'Du avbrøt!'
    else:
        messagebox.showinfo('Info her', 'Du skrev: ' + name, parent=app._root)
        app.message = f'Hei, {name}!'

def redraw_all(app, canvas):
    canvas.create_text(200, 30, text=app.message)
    canvas.create_text(200, 70, text=f'{app.count}')

run_app(width=400, height=200)
```



↑ Innhold

Bilder

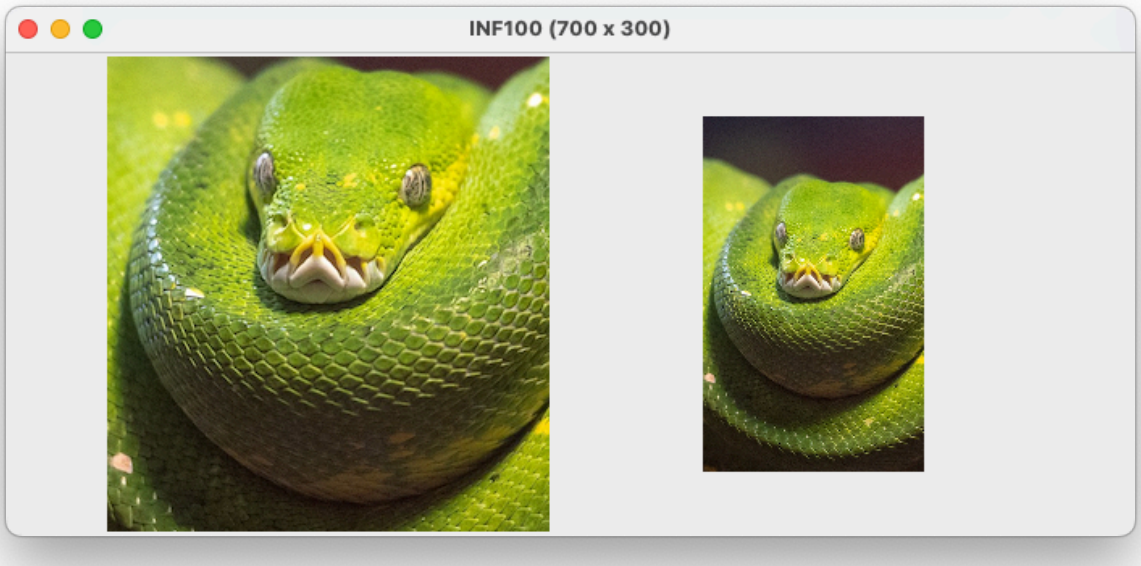
```
# Vis et bilde hentet fra internett

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http, scaled_image

def app_started(app):
    url = 'https://tinyurl.com/inf100logo-png'
    app.image1 = load_image_http(url)
    app.image2 = scaled_image(app.image1, 0.5)

def redraw_all(app, canvas):
    canvas.create_image(200, 150, pil_image=app.image1)
    canvas.create_image(500, 150, pil_image=app.image2)

run_app(width=700, height=300)
```



For å vise et bilde lagret lokalt på datamaskinen, husk at filstien til bildet er relativ til mappen programmet *kjøres* fra, en mappe som ikke nødvendigvis er den samme mappen hvor programmet *ligger*. For å kjøre eksempelkoden under, last ned greensnake.png. Resultatet blir akkurat som over, men koden vil også fungere uten internett-tilgang.

↑ Innhold

```python
# Vis et bilde lagret lokalt på datamaskinen

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image, scaled_image

def app_started(app):
    app.image1 = load_image('greensnake.png')
    app.image2 = scaled_image(app.image1, 0.5)

def redraw_all(app, canvas):
    canvas.create_image(200, 150, pil_image=app.image1)
    canvas.create_image(500, 150, pil_image=app.image2)

run_app(width=700, height=300)
```

```python
# Ankeret bestemmer hvor bildet tegnes i forhold til koordinatet (x, y)

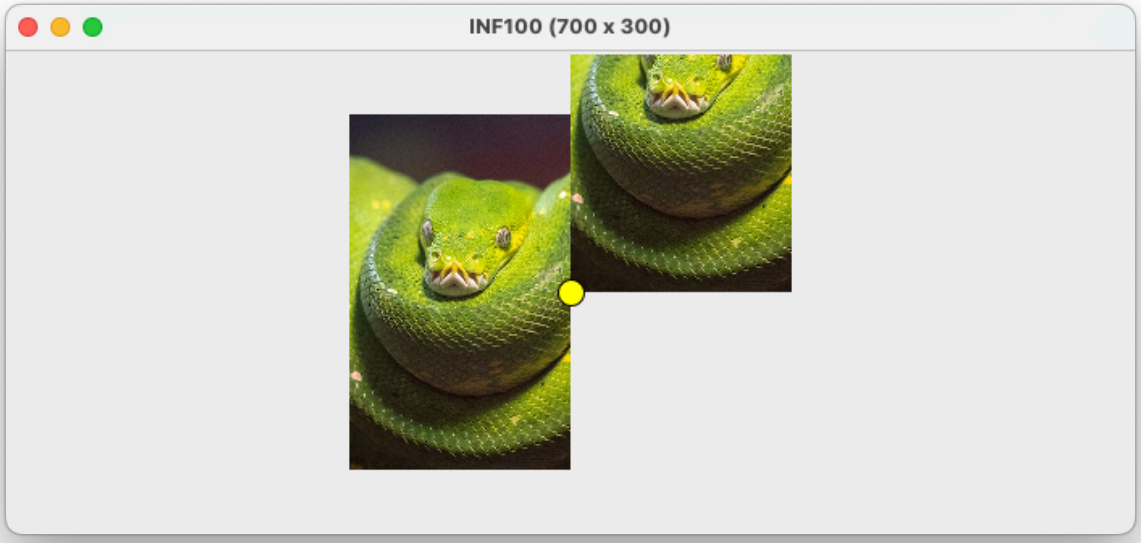
from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http, scaled_image

def app_started(app):
    url = 'https://tinyurl.com/inf100logo-png'
    app.image1 = load_image_http(url)
    app.image2 = scaled_image(app.image1, 0.5)

def redraw_all(app, canvas):
    x, y = app.width/2, app.height/2
    # Prøv ulike verdier for anchor:
    # 'n', 's', 'e', 'w', 'ne', 'nw', 'se', 'sw', 'center'
    canvas.create_image(x, y, pil_image=app.image2, anchor='e')
    canvas.create_image(x, y, pil_image=app.image2, anchor='sw')
    canvas.create_oval(x-8, y-8, x+8, y+8, fill='yellow')

run_app(width=700, height=300)
```

↑ Innhold




```python
# Finn bildet sin størrelse

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http, scaled_image

def app_started(app):
    url = 'https://tinyurl.com/inf100logo-png'
    app.image1 = load_image_http(url)
    app.image2 = scaled_image(app.image1, 0.5)
    app.image3 = scaled_image(app.image1, 0.3)

def draw_image_with_size_below_it(app, canvas, image, cx, cy):
    canvas.create_image(cx, cy, pil_image=image)
    image_width, image_height = image.size
    msg = f'Bildestørrelse: {image_width} x {image_height}'
    canvas.create_text(cx, cy + image_height/2 + 20, text=msg, font='Arial 16')

def redraw_all(app, canvas):
    draw_image_with_size_below_it(app, canvas, app.image2, 200, 150)
    draw_image_with_size_below_it(app, canvas, app.image3, 500, 150)

run_app(width=700, height=300)
```

↑ Innhold

INF100 (700 x 300)



Bildestørrelse: 137 x 220



Bildestørrelse: 82 x 132

```python
# Bruk transpose-metoden for å snu eller flippe et bilde

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http, scaled_image
from PIL import Image

def app_started(app):
    url = 'https://tinyurl.com/inf100logo-png'
    org_image = load_image_http(url)
    org_image = scaled_image(org_image, 1/5)

    # Rotere et bilde 90 grader
    rotated_image = org_image.transpose(Image.ROTATE_90)
    app.images = [org_image, rotated_image]

    # Flere måter å rotere/flippe et bilde på
    app.images += [org_image.transpose(tp_method) for tp_method in (
        Image.ROTATE_180,
        Image.ROTATE_270,
        Image.FLIP_LEFT_RIGHT,
        Image.TRANSPOSE,
        Image.FLIP_TOP_BOTTOM,
        Image.TRANSVERSE,
    )]

def redraw_all(app, canvas):
    allocated_width_per_image = app.width / len(app.images)
    for i, img in enumerate(app.images):
        x = (i + 0.5) * allocated_width_per_image
        y = app.height / 2
        canvas.create_image(x, y, pil_image=img)

run_app(width=700, height=200)
```



↑ Innhold

```python
# Hent farger med getpixel, og manipulér eller tegn nye bilder med putpixel

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http
from PIL import Image

def app_started(app):
    url = 'https://tinyurl.com/inf100logo-png'
    app.image1 = load_image_http(url)

    # la oss lage en kopi som bytter om blå og grønn -verdier
    app.image1 = app.image1.convert('RGB')
    app.image2 = Image.new(mode='RGB', size=app.image1.size)
    for x in range(app.image2.width):
        for y in range(app.image2.height):
            r, g, b = app.image1.getpixel((x, y))
            app.image2.putpixel((x, y), (r, b, g)) # merk: b og g byttet plass

def redraw_all(app, canvas):
    canvas.create_image(200, 300, pil_image=app.image1)
    canvas.create_image(500, 300, pil_image=app.image2)

run_app(width=700, height=600)
```

INF100 (700 x 600)



↑ Innhold

```python
# Bruk ImageDraw for å lage nye bilder og tegne på dem

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import scaled_image
from PIL import Image, ImageDraw

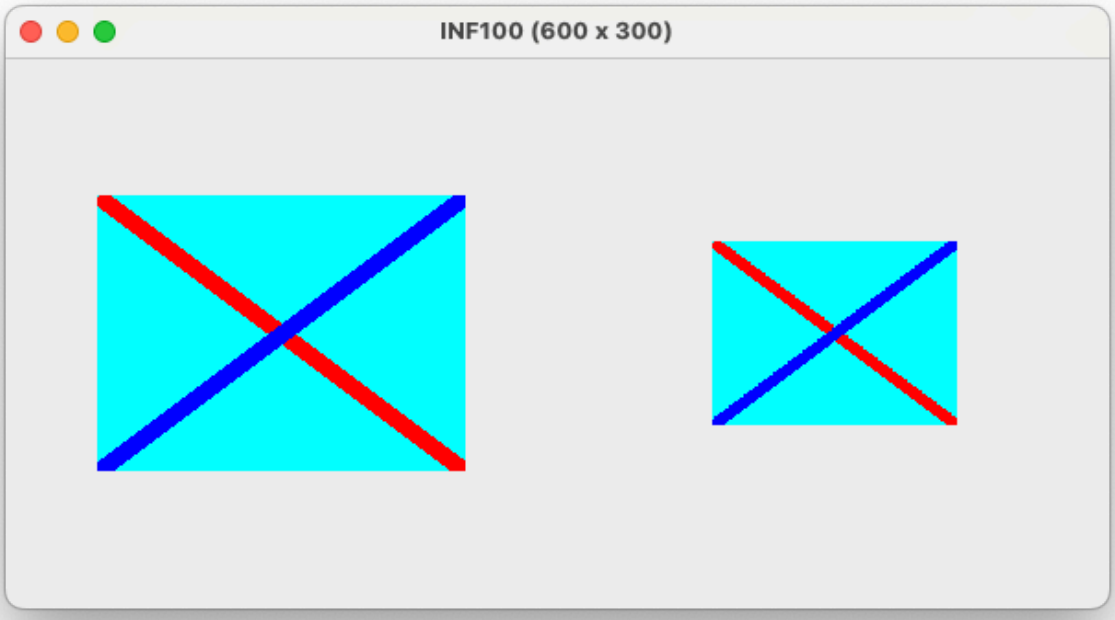
def app_started(app):
    image_width, image_height = app.width//3, app.height//2
    bg_color = (0, 255, 255) # cyan
    app.image1 = Image.new('RGB', (image_width, image_height), bg_color)

    # Nå som vi har laget et nytt bilde, bruk ImageDraw for å tegne i det
    # Se https://pillow.readthedocs.io/en/stable/reference/ImageDraw.html
    draw = ImageDraw.Draw(app.image1)
    draw.line((0, 0, image_width, image_height), width=10, fill=(255, 0, 0))
    draw.line((0, image_height, image_width, 0), width=10, fill=(0, 0, 255))

    # Så lager vi en skalert kopi for å vise at dette er et helt vanlig bilde
    app.image2 = scaled_image(app.image1, 2/3)

def redraw_all(app, canvas):
    canvas.create_image(app.width/4, app.height/2, pil_image=app.image1)
    canvas.create_image(app.width*3/4, app.height/2, pil_image=app.image2)

run_app(width=600, height=300)
```



↑ Innhold

```python
# Bruk get_snapshot og save_snapshot for å ta bilde av programmet
# Merk: dette er litt ustabilt på noen operativsystemer, og kan
# kreve at du gir programmet spesiell tilgang for å ta screenshot

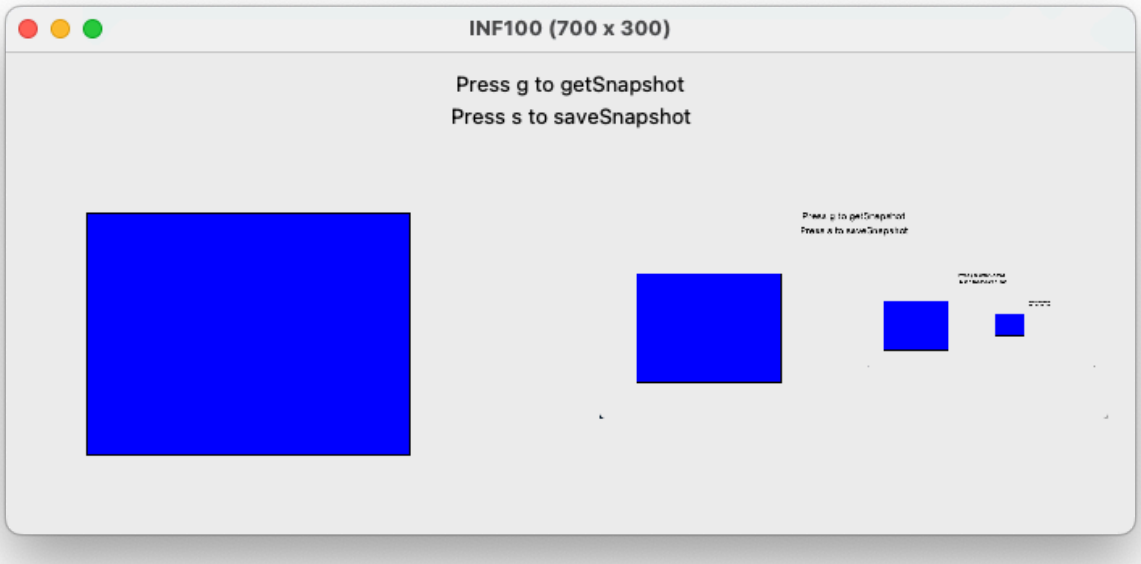
from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import scaled_image

def app_started(app):
    app.image = None

def key_pressed(app, event):
    if (event.key == 'g'):
        snapshotImage = app.get_snapshot()
        app.image = scaled_image(snapshotImage, 0.45)
    elif (event.key == 's'):
        app.save_snapshot()

def redraw_all(app, canvas):
    canvas.create_text(350, 20, text='Press g to get_snapshot', fill='black')
    canvas.create_text(350, 40, text='Press s to save_snapshot', fill='black')
    canvas.create_rectangle(50, 100, 250, 250, fill='blue')
    if app.image is not None:
        canvas.create_image(525, 160, pil_image=app.image)

run_app(width=700, height=300)
```

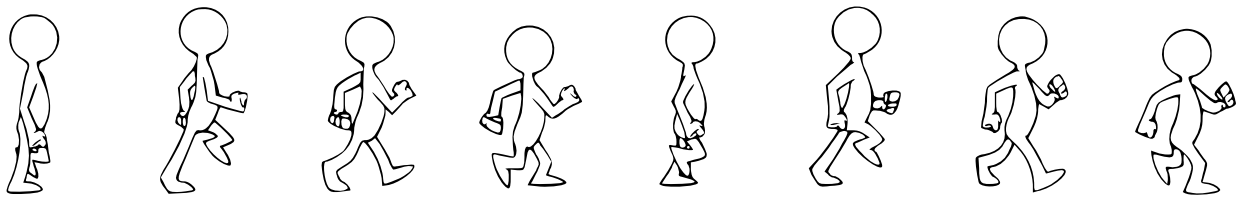


Animerte bilder

En *sprite* er et bilde/en tegning som representerer et objekt som utgjør en (liten) del av skjermbildet. I sammenheng med dataspill vil en sprite typisk være knyttet til et logisk element i spillet, slik som en figur, et hus, en fiende, en energidrikk, et stykke med gress eller lignende. Det er vanlig at sprites er animert; for

eksempel en animasjon av at figuren går. En slik animasjon består av en sekvens med bilder som spilles av i ring. Under er et eksempel på en bildefil som representerer en animert figur.

↑ Innhold



Bilde: CC0, Walkin man figure av JayNick, via freesvg.org (https://freesvg.org/walking-man-figure).

For å animere denne spriten, klipper vi opp bildet i 8 like store biter, og viser dem frem etter hverandre i et sirkulært mønster.

↑ Innhold

```python
# En demo for å animere sprites med bruk av Pillow/PIL
# crop-metoden klipper ut en del av bildet
# Se her for flere detaljer:
# https://pillow.readthedocs.io/en/stable/reference/Image.html

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http

def app_started(app):
    # Bilde av JayNick, CC0, https://freesvg.org/walking-man-figure
    url = 'https://tinyurl.com/inf100-strekmann-png'
    spritestrip = load_image_http(url)

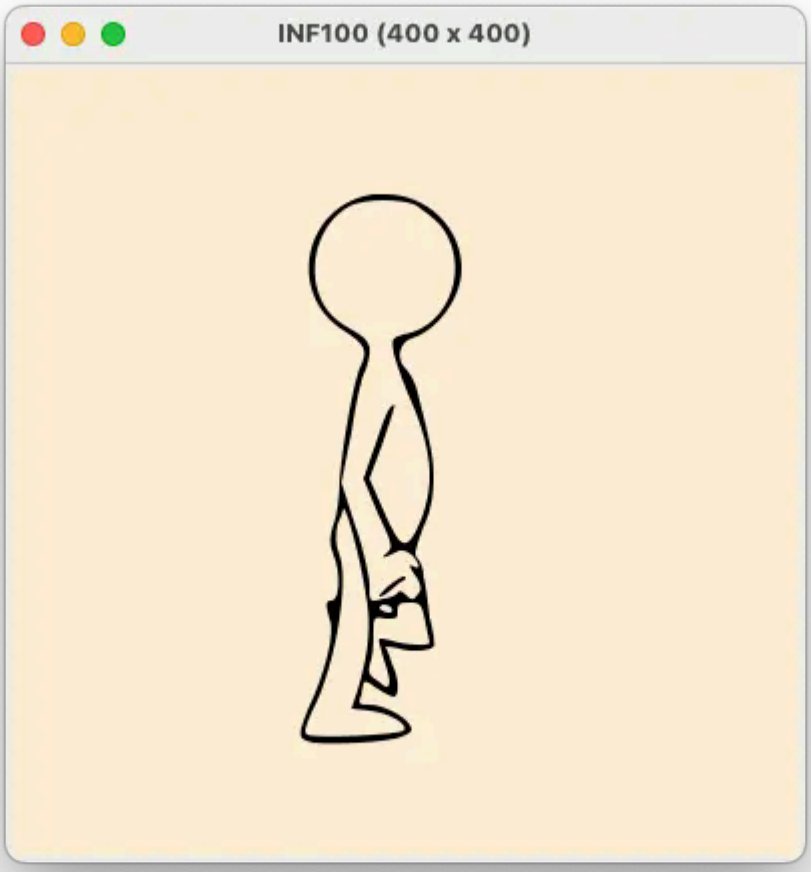
    # Vi lager en liste som inneholder hvert av de 8 bildene separat
    app.sprites = []
    image_width, image_height = spritestrip.size
    sprite_width = image_width / 8
    for i in range(8):
        left_x = i * sprite_width
        right_x = left_x + sprite_width
        sprite = spritestrip.crop((left_x, 0, right_x, image_height))
        app.sprites.append(sprite)
    app.sprite_counter = 0

def timer_fired(app):
    # Gå til neste bilde i bilde-sekvensen
    app.sprite_counter = (1 + app.sprite_counter) % len(app.sprites)

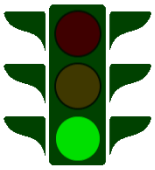
def redraw_all(app, canvas):
    # Bakgrunnsfarge
    bg = 'papaya whip'
    canvas.create_rectangle(0, 0, app.width, app.height, fill=bg, outline='')
    # Tegn strekmann
    sprite = app.sprites[app.sprite_counter]
    canvas.create_image(200, 200, pil_image=sprite)

run_app(width=400, height=400)
```

↑ Innhold



Gif er en type bilder som er animert i utgangspunktet, slik som bildet under. Vi kan bruke animerte gif'er for å lage en animert sprite. Det er mulig å hente ut timing-informasjon for hvert stillbilde i gif'en, men det krever en del omtanke å få programmet ditt til å vise gif'en med riktig timing (det skjer ikke automatisk).



Bilde: CC0, «Traffic light» fra EditableGIFs.com

```python
# En demo for å vise animert gif/apng sprites

from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http
from datetime import datetime, timedelta

def app_started(app):
    animation = load_image_http('https://files.stromme.me/inf100/traffic.gif')
    app.frames = extract_frames(animation)
    app.frame_index = 0
    app.last_frame_switch = datetime.now()

def extract_frames(multiframe_image):
    '''Given an image with multiple frames (such as a gif), return a list of
    individual frames. The frames will be tagged with their original duration
    in the 'duration' property of the 'info' dictionary.'''
    frames = []
    i = 0
    while True:
        try:
            multiframe_image.seek(i)
        except EOFError:
            break
        duration = multiframe_image.info['duration']
        frames.append(multiframe_image.copy())
        frames[-1].info['duration'] = duration
        i += 1
    return frames

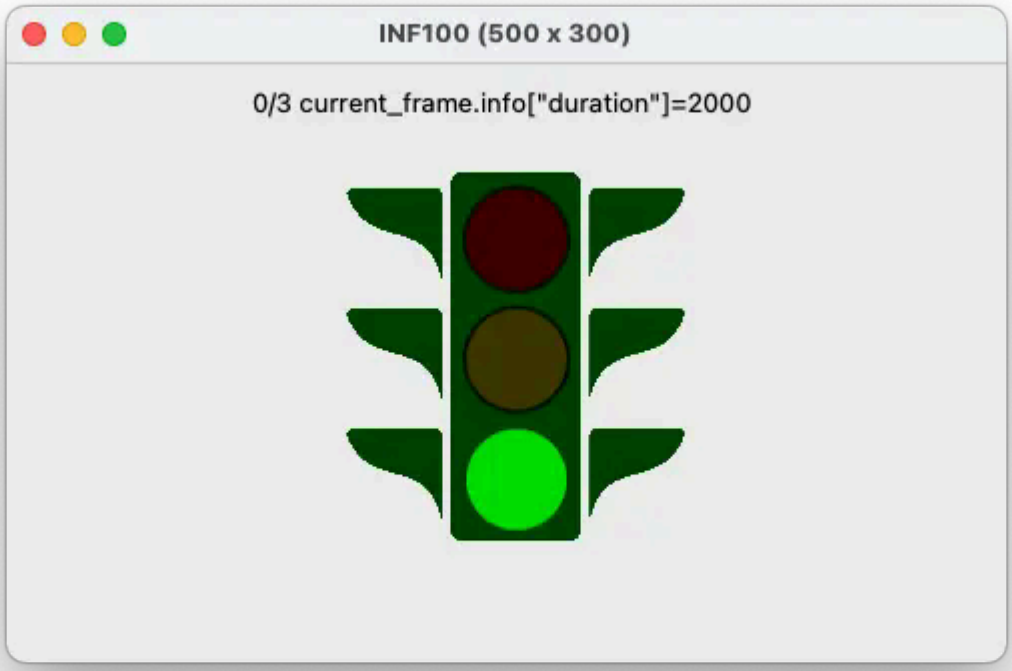
def timer_fired(app):
    maintain_animation(app)

def maintain_animation(app):
    '''Ensure that the animation is displayed at the intended speed.
    This method interprets the duration as a minimum display time for
    the frame, and will hence drift over time.'''
    current_frame = app.frames[app.frame_index]
    current_time = datetime.now()
    min_display_time = timedelta(milliseconds=current_frame.info['duration'])
    time_since_last_frame_switch = current_time - app.last_frame_switch
    if min_display_time < time_since_last_frame_switch:
        # Go to next frame
        app.frame_index = (1 + app.frame_index) % len(app.frames)
        app.last_frame_switch = current_time

def redraw_all(app, canvas):
    current_frame = app.frames[app.frame_index]
    canvas.create_image(app.width/2, app.height/2, pil_image=current_frame)
    canvas.create_text(app.width/2, 20, text=(
        f'{app.frame_index}/{len(app.frames)} '
        f'{current_frame.info["duration"]=} '
```

```
    ))

run_app(width=500, height=300)
```



Modus (ulike skjermer)

```
from uib_inf100_graphics.event_app import run_app
from uib_inf100_graphics.helpers import load_image_http, scaled_image
import random

########################################
# Splash-skjerm -modus
########################################

def splash_screen_mode_redraw_all(app, canvas):
    canvas.create_image(
        app.width / 2,
        app.height / 2,
        pil_image=app.splash_image,
    )
    canvas.create_text(
        app.width / 2,
        app.height / 2,
        text=(
            'Demo av en modal applikasjon!\n'
            'Dette er en modal skjerm!\n'
            'Trykk en tast for å starte!'
        ),
        font=app.default_font,
        justify='center',
    )

def splash_screen_mode_key_pressed(app, event):
    app.mode = 'game_mode'

########################################
# Spill -modus
########################################

def game_mode_redraw_all(app, canvas):
    canvas.create_text(
        app.width / 2,
        10,
        text=(
            f'Poeng: {app.score}\n'
            'Klikk på rundingen!\n'
            'Trykk h for hjelpe-skjerm\n'
            'Trykk v for å bryte MVC'
        ),
        font=app.default_font,
        justify='center',
        anchor='n',
    )
    draw_dot(canvas, app.x, app.y, app.r, app.color)
    if app.make_MVC_violation:
```

↑ Innhold

```
        app.ohNo = 'Gjør et brudd med MVC skjer her!'

def game_mode_timer_fired(app):
    move_dot(app)

def game_mode_mouse_pressed(app, event):
    d = ((app.x - event.x)**2 + (app.y - event.y)**2)**0.5
    if d <= app.r:
        app.score += 1
        randomize_dot(app)
    elif app.score > 0:
        app.score -= 1

def game_mode_key_pressed(app, event):
    if event.key == 'h':
        app.mode = 'help_mode'
    elif event.key == 'v':
        app.make_MVC_violation = True

####################################
# Hjelpeskjerm -modus
####################################

def help_mode_redraw_all(app, canvas):
    canvas.create_text(
        app.width / 2,
        app.height / 2,
        text=(
            'Her er hjelpeskjermen!\n'
            '(hjelpsom melding her)\n'
            'Trykk en tast for å snu'
        ),
        font=app.default_font,
        justify='center',
    )

def help_mode_key_pressed(app, event):
    app.mode = 'game_mode'

####################################
# Felles hjelpefunksjoner
####################################

def randomize_dot(app):
    app.x = random.randrange(20, app.width - 20)
    app.y = random.randrange(20, app.height - 20)
    app.r = random.randrange(10, 20)
    app.color = random.choice(['red', 'orange', 'yellow', 'green', 'blue'])
    app.dx = random.choice([+1, -1]) * random.randrange(3, 7)
    app.dy = random.choice([+1, -1]) * random.randrange(3, 7)

def move_dot(app):
```

↑ Innhold

```
    app.x += app.dx
    if (app.x < 0) or (app.x > app.width):
        app.dx = -app.dx
    app.y += app.dy
    if (app.y < 0) or (app.y > app.height):
        app.dy = -app.dy

def draw_dot(canvas, x, y, r, color):
    canvas.create_oval(x - r, y - r, x + r, y + r, fill=color)

####################################
# Oppstart av applikasjonen
####################################

def app_started(app):
    splash_img_url = 'https://files.stromme.me/inf100/splashbox-alpha.png'
    app.splash_image = load_image_http(splash_img_url)
    app.splash_image = scaled_image(app.splash_image, 0.85)
    app.default_font = 'Arial 20'
    app.mode = 'splash_screen_mode'
    app.score = 0
    app.timer_delay = 50
    app.make_MVC_violation = False
    randomize_dot(app)

run_app(width=500, height=250)
```



↑ Innhold

Scrolling

```python
# Sidelengs scrolling (hele verden beveger seg):

from uib_inf100_graphics.event_app import run_app
import random

def app_started(app):
    app.scroll_x = 0
    app.dots = []
    for _ in range(50):
        x = random.randrange(app.width)
        y = random.randrange(60, app.height)
        app.dots.append((x, y))

def key_pressed(app, event):
    if event.key == 'Left':
        app.scroll_x -= 5
    elif event.key == 'Right':
        app.scroll_x += 5

def redraw_all(app, canvas):
    # Tegn spilleren (som alltid er midt på skjermen)
    draw_dot(canvas, app.width/2, app.height/2, 10, 'cyan')

    # Tegn prikkene, sideforskjøvet med et offset scroll_x
    for (cx, cy) in app.dots:
        cx -= app.scroll_x  # <-- Her sideforskyver vi prikkene på lerretet
        draw_dot(canvas, cx, cy, 10, 'lightGreen')

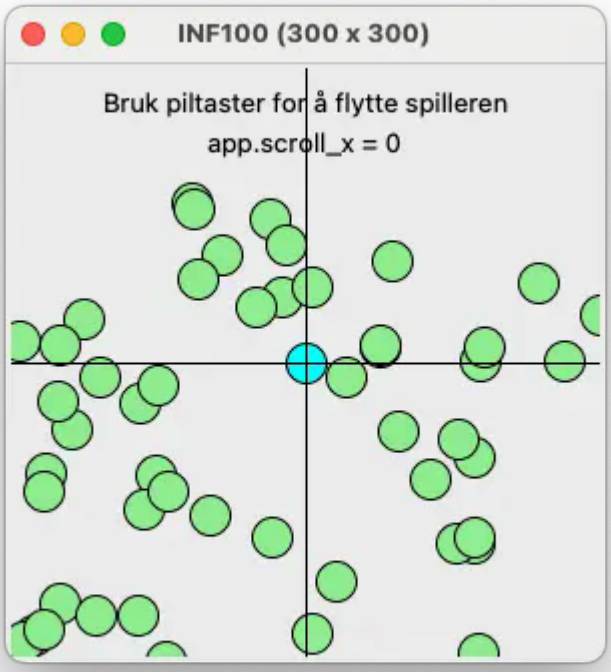
    # Tegn x- og y-aksen
    x = app.width/2 - app.scroll_x # <-- Her sideforskyver vi
    y = app.height/2
    canvas.create_line(x, 0, x, app.height)
    canvas.create_line(0, y, app.width, y)

    # Tegn instruksjoner og debug-informasjon
    x = app.width/2
    canvas.create_text(x, 20, text='Bruk piltaster for å flytte spilleren')
    canvas.create_text(x, 40, text=f'{app.scroll_x = }')

def draw_dot(canvas, x, y, r, color):
    canvas.create_oval(x - r, y - r, x + r, y + r, fill=color)

run_app(width=300, height=300)
```

↑ Innhold


INF100 (300 x 300)
Bruk piltaster for å flytte spilleren
app.scroll_x = 0

↑ Innhold

```python
# Sidelengs scrolling når figuren er på vei ut av skjermen

from uib_inf100_graphics.event_app import run_app
import random

def app_started(app):
    app.scroll_x = -app.width/2 # Initiell scroll slik at x=0 er sentrert
    app.scroll_margin = 50
    app.player_x = 0 # spillerens initielle posisjon
    app.dots = [(random.choice(range(-app.width//2, app.width//2)),
                 random.choice(range(60, app.height))) for _ in range(50)]

def make_player_visible(app):
    # scroll skjermen så mye som nødvendig for at spilleren vises
    if (app.player_x < app.scroll_x + app.scroll_margin):
        app.scroll_x = app.player_x - app.scroll_margin
    if (app.player_x > app.scroll_x + app.width - app.scroll_margin):
        app.scroll_x = app.player_x - app.width + app.scroll_margin

def move_player(app, dx, dy):
    app.player_x += dx
    make_player_visible(app)

def key_pressed(app, event):
    if (event.key == "Left"):    move_player(app, -5, 0)
    elif (event.key == "Right"): move_player(app, +5, 0)

def redraw_all(app, canvas):
    # Tegn spilleren, sideforskjøvet med et offset scroll_x
    cx, cy, r = app.player_x, app.height/2, 10
    cx -= app.scroll_x # <-- Her sideforskyver vi spilleren
    canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill='cyan')

    # Tegn prikkene, sideforskjøvet med et offset scroll_x
    for (cx, cy) in app.dots:
        cx -= app.scroll_x  # <-- Her sideforskyver vi prikkene
        canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill='lightGreen')

    # Tegn x- og y-aksen
    x = -app.scroll_x # <-- Her sideforskyver vi y-aksen
    y = app.height/2
    canvas.create_line(x, 0, x, app.height)
    canvas.create_line(0, y, app.width, y)

    # Tegn instruksjoner og debug-informasjon
    x = app.width/2
    canvas.create_text(x, 20, text='Use arrows to move left or right',
                       fill='black')
    canvas.create_text(x, 40, text=f'app.scroll_x = {app.scroll_x}',
                       fill='black')
```

```
run_app(width=300, height=300)
```

→ Innhold

↑ Innhold

```python
# Sidelengs scrolling når figuren er på vei ut av skjermen
# Kollisjonsdeteksjon for spilleren

from uib_inf100_graphics.event_app import run_app

def app_started(app):
    # Scrolling
    app.scroll_margin = 50
    app.scroll_x = -app.scroll_margin
    app.scroll_y = -app.height / 2

    # Spiller - koordinate for hjørnet til venstre nederst
    app.player_x = 0
    app.player_y = 0
    app.player_width = 10
    app.player_height = 20

    # Murer
    app.walls = 5
    app.wall_points = [0]*app.walls
    app.wall_width = 20
    app.wall_height = 40
    app.wall_x_positions = [90 * (i + 1) for i in range(app.walls)]
    app.current_wall_hit = -1 # -1 når ingen mur er truffet

def get_player_bounds(app):
    # returnerer absolutt-posisjon til spiller på et uendelig lerret
    # tar ikke hensyn til scrolling
    (x0, y1) = (app.player_x, app.player_y)
    (x1, y0) = (x0 + app.player_width, y1 - app.player_height)
    return (x0, y0, x1, y1)

def get_wall_bounds(app, wall):
    # returnerer absolutt-posisjon til en mur på et uendelig lerret
    # tar ikke hensyn til scrolling
    (x0, y1) = (app.wall_x_positions[wall], 0)
    (x1, y0) = (x0 + app.wall_width, y1 - app.wall_height)
    return (x0, y0, x1, y1)

def get_wall_hit(app):
    # returnerer hvilken mur spilleren er over for øyeblikket
    # merk: i større spill bør denne funksjonen optimeres til å kun
    # sjekke de objektene som er synlige.
    player_bounds = get_player_bounds(app)
    for wall in range(app.walls):
        wall_bounds = get_wall_bounds(app, wall)
        if bounds_intersect(app, player_bounds, wall_bounds):
            return wall
    return -1

def bounds_intersect(app, bounds_a, bounds_b):
```

```
    # return l2<=r1 and t2<=b1 and l1<=r2 and t1<=b2
    (ax0, ay0, ax1, ay1) = bounds_a
    (bx0, by0, bx1, by1) = bounds_b
    return ((ax1 >= bx0) and (bx1 >= ax0) and
            (ay1 >= by0) and (by1 >= ay0))

def check_for_new_wall_hit(app):
    # sjekk om vi treffer en ny mur for første gang
    wall = get_wall_hit(app)
    if (wall != app.current_wall_hit):
        app.current_wall_hit = wall
        if (wall >= 0):
            app.wall_points[wall] += 1

def make_player_visible(app):
    # scroll for å gjøre spilleren synlig
    m = app.scroll_margin
    if (app.player_x < app.scroll_x + m):
        app.scroll_x = app.player_x - m
    if (app.player_x > app.scroll_x + app.width - m - app.player_width):
        app.scroll_x = app.player_x - app.width + m + app.player_width
    if (app.player_y < app.scroll_y + m + app.player_height):
        app.scroll_y = app.player_y - m - app.player_height
    if (app.player_y > app.scroll_y + app.height - m):
        app.scroll_y = app.player_y - app.height + m

def move_player(app, dx, dy):
    app.player_x += dx
    app.player_y += dy
    make_player_visible(app)
    check_for_new_wall_hit(app)

def size_changed(app):
    make_player_visible(app)

def mouse_pressed(app, event):
    app.player_x = event.x + app.scroll_x - app.player_width/2
    app.player_y = event.y + app.scroll_y + app.player_height/2
    make_player_visible(app)
    check_for_new_wall_hit(app)

def key_pressed(app, event):
    if (event.key == "Left"):    move_player(app, -5, 0)
    elif (event.key == "Right"): move_player(app, +5, 0)
    elif (event.key == "Up"):    move_player(app, 0, -5)
    elif (event.key == "Down"):  move_player(app, 0, +5)

def redraw_all(app, canvas):
    sx = app.scroll_x
    sy = app.scroll_y

    # Tegn x-aksen
```

↑ Innhold

```
    line_y = -sy
    line_height = 5
    canvas.create_rectangle(0, line_y, app.width, line_y+line_height,
                            fill="black")
    # Tegn murene
    # (Merk: kan optimiseres til å kun tegne synlige murer)
    for wall in range(app.walls):
        (x0, y0, x1, y1) = get_wall_bounds(app, wall)
        fill = "orange" if (wall == app.current_wall_hit) else "pink"
        canvas.create_rectangle(x0-sx, y0-sy, x1-sx, y1-sy, fill=fill)
        (cx, cy) = ((x0+x1)/2 - sx, (y0 + y1)/2 - sy)
        canvas.create_text(cx, cy, text=str(app.wall_points[wall]),
                           fill='black')
        cy = line_y + 5
        canvas.create_text(cx, cy, text=str(wall), anchor=N, fill='black')

    # Tegn spilleren
    (x0, y0, x1, y1) = get_player_bounds(app)
    canvas.create_oval(x0 - sx, y0 - sy, x1 - sx, y1 - sy, fill="cyan")

    # Tegn instruksjonene
    msg = "Bruk pilene eller musen for å flytte"
    canvas.create_text(app.width/2, 20, text=msg, fill='black')

run_app(width=300, height=300)
```

Spille av lyder

For å spille av lyder må du først installere *pygame*, og så laste ned uib_inf100_music.py og legge den i samme mappe som python-filen du kjører. For å installere pygame:

- bruk kommandoen `pip install pygame` i terminalen (eksakt kommando kan variere litt mellom ulike operativsystem og hvordan python er installert på maskinen. Du kan også benytte samme skript du brukte for installasjon i grafikk, men bytt ut *uib-inf100-graphics* med *pygame*)

For at eksempelet under skal virke, må du også laste ned button.mp3 og music.mp3 i samme mappe programmet *kjøres fra* (som ikke nødvendigvis er samme mappe hvor programmet *ligger*).

↑ Innhold

```python
from uib_inf100_graphics.event_app import run_app
from uib_inf100_music import load_sound_effect, load_looping_sound,
stop_all_sounds

def app_started(app):
    app.sound_effect = load_sound_effect('button.mp3')
    app.background_music = load_looping_sound('music.mp3')

def app_stopped(app):
    stop_all_sounds()

def key_pressed(app, event):
    match event.key:
        case 'b':
            # toggle background music
            if app.background_music.is_playing():
                app.background_music.stop()
            else:
                app.background_music.play()
        case 'e':
            # play sound effect
            app.sound_effect.play()
        case 's':
            # stop all sounds
            stop_all_sounds()
        case _ if event.key.isdigit():
            # set volume
            new_volume = float(event.key)/9
            app.background_music.set_volume(new_volume)

def redraw_all(app, canvas):
    text = (
        f'{app.background_music.is_playing()=}\n'
        f'{app.background_music.get_volume()=}\n\n'
        'Trykk b for å starte/stoppe bakgrunnslyd\n'
        'Trykk e for å spille en lydeffekt\n'
        'Trykk s for å stoppe alle lyder\n'
        'Trykk 0–9 for å sette volum for bakgrunnslyd'
    )
    canvas.create_text(app.width/2, app.height/2, text=text, font='Arial 20')

run_app(width=600, height=300)
```

JSON

↑ Innhold

JSON er et tekst-basert filformat. Teksten i en JSON-fil følger en bestemt struktur som gjør det ekstra lett for et Python-program å jobbe med innholdet.

Ren JSON

- Eksempel
- Gyldig JSON

JSON og Python

- Eksempel
- Gyldig JSON
- Objekter i Python
- Oppslag
- Indeksering
- Dype oppslag og indekseringer
- Endre verdier
- Legge til verdier
- Fjerne verdier
- Lese en JSON-fil med Python
- Skrive en JSON-fil med Python

> *PS: På denne siden benytter vi Python-samfunnet sin «ordbok» for å beskrive datastrukturene; vokabularet kan derfor være litt forskjellig fra hva som brukes om JSON i andre kontekster.*

Eksempel

Den letteste strategien for å bli kjent med JSON er å se på et eksempel. Last ned: kontakter.json eller bare kikk på innholdet ved å se under:

↑ Innhold

```
{
  "contacts": [
    {
      "name": "Martin",
      "number": "+47 12345678",
      "address": {
        "street": "Gateveien",
        "number": 24,
        "country": "Norway"
      }
    },
    {
      "name": "Ada",
      "number": "+47 87654321",
      "address": {
        "street": "Veitraséen",
        "number": 42,
        "country": "Norway"
      }
    }
  ],
  "last missed call": {
    "timestamp": {
      "year": 2025,
      "month": 8,
      "day": 19,
      "hour": 14,
      "minute": 15,
      "second": 0
    },
    "number": "+47 12345678"
  }
}
```

💬 Diskusjonsoppgave

1. Endre filen slik at forrige tapte anrop kom fra Ada akkurat samme tidspunkt som forelesning i INF100 denne uken.
2. Endre filen slik at vi også får Erling i kontaktlisten. Erling bor i Allégaten 66 og har telefonnummer +47 23456789.

Gyldig JSON

Strukturen i en JSON-fil er basert på at innholdet i filen beskriver ett *objekt*. Det finnes flere ulike *typer* objekter, som vi beskriver her

Atomiske typer

↑ Innhold

Vi beskriver først noen atomiske typer. At typen er «atomisk» betyr bare at den ikke er bygget opp av andre objekter.

Type	*Forklaring*	*Eksempler*

int	Heltall.	`0`
		`42`
		`-3`

float	Flyttall. Tilsvarer desimaltall i matematikk. Selv om en float slutter på `.0`, er det likevel en float og ikke en int.	`1.5`
		`42.0`
		`-0.4`

str	Strenger. Dette er «tekst» -objekter; en streng er en sekvens av skrifttegn (bokstaver, tall, mellomrom linjeskift, symboler, emojii etc). I en JSON-fil angir hermetegnet `"` at en streng begynner eller slutter. Det er mulig å ha en sekvens med 0 skrifttegn, dette kalles den «tomme» strengen.	`"hei"`
		`"42"`
		`""`

bool og null/none

I tillegg til datatypene nevnt over, støtter JSON også den atomiske datatypen *bool* og det spesielle objektet *null* som er sin egen type. Akkurat disse to typene er bittelitt ulike i JSON og i Python:

bool	Boolske verdier er de to verdiene `true` og `false`. I Python finnes også disse to verdiene, men de kalles der for `True` og `False` (med stor forbokstav).
null/None	I JSON finnes det en spesiell verdi `null` som indikerer «ingen verdi». I Python kalles denne verdien for `None`.

Når du leser JSON-data med Python sin `json`-modul, vil verdiene konverteres automatisk til «pythonsk». Tilsvarende, hvis du skriver JSON-data med Python sin `json`-modul vil verdiene konverteres automatisk til «jsonsk».

De atomiske typene er de minste byggeklossene i en JSON-fil. For å representere større mengder data, trenger vi også minst én av de to sammensatte objekttypene: lister (list) og oppslagsverk (dict).

Lister

En liste er et objekt som inneholder 0, 1, 2 eller flere objekter. En liste er omsluttet av firkantklammer (`[` og `]`) og objektene i listen skilles fra hverandre med komma (`,`). Det er valgfritt med mellomrom og linjeskift for å skille objektene i tillegg. Lister følger mønsteret

```
[ ▲ , ▲ , … , ▲ ]
```

hvor `▲` er andre gyldige JSON -objekter. For eksempel:

```
[42, 45, 40, 88, 45]
```

Flere eksempler

```
["hello",42]
```

```
[
  "Martin",
  "Ada",
  "Erling"
]
```

Det er også mulig at et liste-objekt har 0 objekter inni seg. Da sier vi at listen er «tom».

```
[]
```

Oppslagsverk

Oppslagsverk (dict) er objekter bygget opp av *nøkkel-verdi –par*. En nøkkel er en streng, mens en verdi kan være hvilket som helst objekt. Nøkkel-verdi –par skilles med komma (`,`), og oppslagsverket er omgitt av krøllparenteser (`{` og `}`). Oppslagsverk følger mønsteret

```
{ ▮: ▲, ▮: ▲, …, ▮: ▲ }
```

Hvor ▮ er unike strenger, og ▲ er gyldige JSON-objekter. For eksempel:

```
{ "my_key": "my_value", "other_key": 42, "third_one": 0.0 }
```

```
{
  "id": 42,
  "values": [24, 25, 26]
}
```

↑ Innhold

↑ Innhold

Objekter i Python

Datastrukturene i JSON er i mange tilfeller identiske med datastrukturene vi benytter i et Python-program; men i et Python-program vil vi ofte ha *variabler* som peker på et objekt slik at vi kan referere til det flere ganger. For eksempel:

```
1   data = {
2       "name": "Martin",
3       "number": "+47 12345678"
4   }
5
6   print(data)
7   print(data["name"])
8   print(data["number"])
9
10  # Skriver ut:
11  # {'name': 'Martin', 'number': '+47 12345678'}
12  # Martin
13  # +47 12345678
```

I eksempelet over er *data* en variabel som refererer til et objekt `{ "name": "Martin", "number": "+47 12345678"}`.

På linje 6, 7 og 8 gir programmet instrukser om å skrive ut informasjon fra objektet til skjermen. Legg merke til at firkantklammene som følger umiddelbart etter variabelnavnet på linje 7 og 8 indikerer at vi henter ut en verdi fra oppslagsverket knyttet til den oppgitte nøkkelen. For eksempel vil `data["name"]` evaluere til `"Martin"`

Oppslag

Hvis man i Python har en variabel som peker på et oppslagsverk, er det vanlig at man ønsker å hente ut et av de «indre» verdiene i objektet og at man allerede vet hvilken nøkkel verdien er knyttet til. Dette gjøres ved oppslag. Eksempel:

```
1   my_object = {
2       "name": "Ada",
3       "number": "+47 87654321"
4   }
5
6   my_name = my_object["name"]
7   my_number = my_object["number"]
8
9   print(my_name)    # Ada
10  print(my_number)  # +47 87654321
```

Oppslaget er altså det som foregår på linje 6-7 i eksempelet. Vi ser det benyttes firkantklammer som plasseres like bak objektet man skal slå opp i. Mellom firkantklammene oppgis nøkkelen. Det hele evaluerer

til den tilhørende verdien.

↑ Innhold

Indeksering

Hvis man i Python har en variabel som peker på en liste, er det vanlig at man ønsker å hente ut et av de «indre» verdiene fra listen. Dette gjøres ved *indeksering*. Eksempel:

```
1   my_list = [42, 43, 94, 95]
2
3   x = my_list[0]  # Indeks 0 viser til det første posisjonen i listen (42)
4   y = my_list[1]  # Indeks 1 viser til den andre posisjonen i listen (43)
5   z = my_list[3]  # Indeks 3 viser til den fjerde posisjonen i listen (95)
6
7   total = x + y + z
8
9   print(x)      # 42
10  print(y)      # 43
11  print(z)      # 95
12  print(total)  # 180
```

Indekseringen er altså det som foregår på linje 3-5. Legg merke til at i Python er lister *null*-indeksert. Det vil si at det første elementet er på posisjon 0 i listen, det andre elementet er på posisjon 1, det tredje elementet er på posisjon 2 og så videre.

Ved både indeksering og oppslag brukes det firkantklammer som plasseres like bak objektet man skal slå opp i. I eksempelet over er *my_list* en variabel som viser til listen vi skal slå opp i.

Dype oppslag og indekseringer

Noen ganger befinner objektet seg nokså «dypt» omsluttet av sammensatte objekter. For å få tak i slike objekter, må vi gjøre flere oppslag og indekseringer for å kommer frem. Dette kan gjøres på samme linje ved å legge oppslagene etter hverandre. For eksempel, se linje 30 her:

↑ Innhold

```
data = {
  "contacts": [
    {
      "name": "Martin",
      "number": "+47 12345678",
      "address": {
        "street": "Gateveien",
        "number": 24
      }
    },
    {
      "name": "Ada",
      "number": "+47 87654321",
      "address": {
        "street": "Veitraséen",
        "number": 42
      }
    }
  ],
  "last missed call": {
    "timestamp": {
      "year": 2025,
      "month": 8,
      "day": 19
    },
    "number": "+47 12345678"
  }
}

martins_street = data["contacts"][0]["address"]["street"]
print(f"Martin's gate er {martins_street}")  # Martin's gate er
Gateveien
```

I det selve det dype oppslaget `data["contacts"][0]["address"]["street"]` legger vi merke til at oppslagene bli mer og mer spesifikk jo lengre mot høyre vi kommer.

Endre verdier

I både oppslagsverk og lister er det mulig å endre på verdier.

```
# Endring av verdi i liste
my_list = ["one", "two", "three"]
my_list[0] = 42

print(my_list)  # [42, 'two', 'three']
```

↑ Innhold

```
# Endring av verdi i oppslagsverk
my_dict = {
  "key1": "value1",
  "key2": "value2"
}

my_dict["key1"] = "new value"

print(my_dict)  # {'key1': 'new value', 'key2': 'value2'}
```

Legge til verdier

I både oppslagsverk og lister er det mulig å legge til nye verdier.

```
# Legge til ny verdi i liste
my_list = ["one", "two", "three"]
x = "four"
my_list.append(x)

print(my_list)  # ['one', 'two', 'three', 'four']
```

```
# Legge til nytt nøkkel-verdi par i oppslagsverk
my_dict = {
  "key1": "value1",
  "key2": "value2"
}
x = 'new value'
my_dict['new key'] = x

print(my_dict)  # {'key1': 'value1', 'key2': 'value2', 'new key': 'new value'}
```

Fjerne verdier

I både oppslagsverk og lister er det mulig å fjerne verdier.

```
# Fjerne verdi på en gitt indeks i listen
my_list = ["one", "two", "three"]
my_list.pop(1)

print(my_list)  # ['one', 'three']
```

↑ Innhold

```
# Fjerne nøkkel-verdi par med gitt nøkkel
my_dict = {
  "key1": "value1",
  "key2": "value2"
}
my_dict.pop("key1")

print(my_dict)  # {'key2': 'value2'}
```

Lese en JSON-fil med Python

Når et Python-program skal lese en JSON-fil, er innebærer det to steg:

1. Man leser filen til en streng som om det var en vanlig tekst-fil (på linje 4 under).
2. Man benytter `json` -modulen for å oversette fra strenger til riktig objekttype (på linje 5 under).

```
1  from pathlib import Path
2  import json
3
4  content = Path("name_of_file.json").read_text(encoding="utf-8")
5  data = json.loads(content)
```

Test på din egen maskin

1. Last ned kontakter.json til arbeidsmappen din. *Hvis du bruker VSCode vil arbeidsmappen din være det øverste området du har åpnet i VSCode sin explorer, altså ikke inne i noen undermapper.*
2. Opprett et program *my_program.py* og skriv inn

```
1  from pathlib import Path
2  import json
3
4  content = Path("kontakter.json").read_text(encoding="utf-8")
5  data = json.loads(content)
6
7  print(data["contacts"])
```

Som resultat skal du få utskriften:

```
[{'name': 'Martin', 'number': '+47 12345678', 'address': {'street':
'Gateveien', 'number': 24, 'country': 'Norway'}}, {'name': 'Ada', 'number':
'+47 87654321', 'address': {'street': 'Allégaten', 'number': 42, 'country':
'Norway'}}]
```

↑ Innhold

Skrive en JSON-fil med Python

Dersom du har et objekt i Python som kun er bygget opp av de vanlig objekttypene støttet av JSON, kan vi med hjelp av dumps-funksjonen i `json` -modulen lagre objektet som en JSON-fil.

Det hele foregår i to steg:

1. Man konverterer objektet til en streng med dumps-funksjonen fra `json` -modulen (linje 9).
2. Man skriver strengen til en fil som en vanlig tekst (linje 10).

```
1  from pathlib import Path
2  import json
3
4  example_data = {
5    "user_id": "foo8247",
6    "password": "HinabeaRDOnt"
7  }
8
9  content = json.dumps(example_data, indent=2)
10 Path("sample.json").write_text(content, encoding="utf-8")
```

Filer og CSV

↑ Innhold

Tekstfiler generelt

- Skrive til og lese fra fil
- Hjelp, filen blir ikke funnet

Komma-separerte verdier (CSV)

- Enkel CSV -håndtering
- CSV som 2D-liste eller liste av oppslagsverk
- CSV-modulen

Skrive til og lese fra fil

Den enkleste måten å lese og skrive filer er ved å bruke den innbygde modulen *pathlib*. Dette virker i Python 3.4 og nyere, og er den metoden vi anbefaler for de fleste tilfeller.

```
from pathlib import Path

# Skrive til en fil
content_string_a = 'Hei, verden!'
Path('minfil.txt').write_text(content_string_a, encoding='utf-8')

# Lese fra en fil
content_string_b = Path('minfil.txt').read_text(encoding='utf-8')
print(content_string_b) # Skriver ut 'Hei, verden!'
```

Den navngitte parameteren `encoding=` bør alltid spesifiseres, ellers kan du få problemer når programmet kjøres på et annet operativsystem. Dersom du skriver til en fil, bør du alltid spesifisere `encoding='utf-8'`. Dersom du leser fra en fil, må du benytte samme koding som ble brukt da filen ble skrevet. Les mer i kursnotater om unicode.

Alternativ: with open('myfile.txt', mode) as f

↑ Innhold

Det er fremdeles mange som benytter `with ... as ...` -strukturen sammen med funksjonen `open` i stedet for å benytte *pathlib* -modulen for å lese og skrive til filer. Fordeler med denne metoden er at den er mer fleksibel for ekspert-bruk, og du trenger ikke å importere noe for å bruke den. Ulempen er at den er litt mer kompleks og tar større plass å skrive.

Open-funksjonen tar inn et filnavn/sti til en fil samt en *modus* og returnerer et «filobjekt». Filobjektet kan vi bruke for å lese fra eller skrive til filen. For å skrive til filen bruker vi modus 'wt' og metoden `write` på filobjektet, mens for å lese fra filen bruker vi modus 'rt' og metoden `read` på filobjektet.

```
# Skrive til en fil
with open('minfil.txt', 'wt', encoding='utf-8') as fileobj:
    fileobj.write('Hei, verden!')

# Lese fra en fil
with open('minfil.txt', 'rt', encoding='utf-8') as fileobj:
    content = fileobj.read()
print(content) # Skriver ut 'Hei, verden!'
```

Noen vanlige moduser for filobjektet er:

- `'r'` : åpner filen for lesing (read)
- `'w'` : åpner filen for skriving (write). Hvis filen ikke eksisterer, opprettes den. Hvis filen eksisterer, overskrives den.
- `'a'` : åpner filen for skriving (append). Hvis filen ikke eksisterer, opprettes den. Hvis filen eksisterer, legges det nye innholdet til på slutten av filen.
- `'x'` : åpner filen for skriving (exclusive). Hvis filen ikke eksisterer, opprettes den. Hvis filen eksisterer fra før, krasjer programmet.

I tillegg kan vi spesifisere om filen skal åpnes i tekstmodus eller binærmodus ved å legge til `'t'` eller `'b'` i modus-strengen. For eksempel `'wt'` eller `'wb'`.

- *`'t'` : åpner filen i tekstmodus (text). Dette er standard, og trenger derfor egentlig ikke å spesifiseres. Benytt denne modusen hvis du skal lese eller skrive tekst, som f. eks. .txt, .csv, .json, .html, .xml, .py etc.*
- *`'b'` : åpner filen i binærmodus (binary). Dette er nødvendig hvis du skal lese eller skrive binære filer (f. eks. bilder, lyd, video, etc.).*

Hjelp, filen blir ikke funnet

Når du kjører et Python-program, kjører programmet «i» en mappe som kalles *current working directory* (cwd). Du kan se hvilken mappe dette er med koden:

↑ Innhold

```python
from pathlib import Path
cwd = Path.cwd()
print(cwd)
```

Denne mappen blir bestemt av programmet som *starter* Python. For eksempel hvis du bruker VSCode for å starte Python, vil terminalen være i den samme mappen som VSCode er åpnet i (den som er nevnt med STORBOKSTAVER i filutforskeren til venstre). Cwd har altså ikke noen sammenheng med hvilken mappe filen som kjøres ligger i.

Når Python får beskjed om å åpne en fil, vil den tolke filstien som blir oppgitt *relativt til* cwd. For eksempel, hvis filstien er kun et filnavn, antas det at filen ligger i cwd.

Eksempel: filen lagres uventet sted

↑ Innhold

La oss si at vi har følgende filstruktur:

```
topfolder/
    foo.txt
    subfolder/
        myscript.py
        qux.txt
```

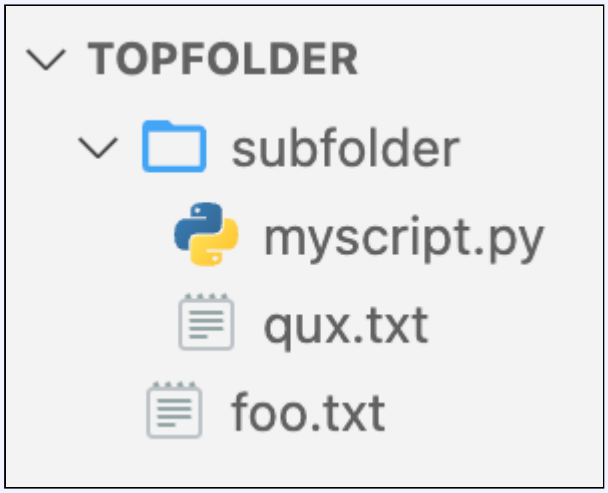
I skriptet *myscript.py* har vi følgende kode:

```
Path('bar.txt').write_text('Hello from bar.txt!', encoding='utf-8')
```

Hvor vil da filen *bar.txt* bli opprettet? Svaret er: **det kommer an på**.

- Hvis du kjører *myscript.py* fra *topfolder* (altså hvis cwd er *topfolder*), vil filen *bar.txt* bli opprettet i *topfolder*.
- Hvis du kjører *myscript.py* fra *subfolder* (altså hvis cwd er *subfolder*), vil filen *bar.txt* bli opprettet i *subfolder*.

Dersom du kjører programmet fra VSCode, vil cwd være den mappen du har åpnet VSCode i. Hvis vi har åpnet VSCode i *topfolder*, ser filutforskeren i VSCode slik ut:



Da vil filen *bar.txt* bli opprettet i mappen topfolder – til tross for at kildekoden befinner seg i mappen subfolder.

Eksempel: finner ikke filen som skal leses

↑ Innhold

La oss si at vi har følgende filstruktur:

```
inf100/
    lab5/
    lab6/
    lab7/
        check_valid_word.py
        wordlist.txt
```

I skriptet *check_valid_word.py* har vi følgende kodelinje:

```
content = Path('wordlist.txt').read_text(encoding='utf-8')
```

Programmet krasjer med følgende feilmelding:

```
Traceback (most recent call last):
  File "/path/to/inf100/lab7/check_valid_word.py", line 2, in <module>
    content = Path('wordlist.txt').read_text(encoding='utf-8')
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/torsteins/.pyenv/versions/3.11.4/lib/python3.11/pathlib.py",
line 1058, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/torsteins/.pyenv/versions/3.11.4/lib/python3.11/pathlib.py",
line 1044, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'wordlist.txt'
```

Hva er feilen? Svaret er sannsynligvis: du kjører programmet fra feil mappe (cwd er altså ikke */path/to/inf100/lab7*). Kan det for eksempel være at du kjører programmet fra *inf100* -mappen?

Feilsøkingssteg:

- Sjekk hva cwd er ved å legge til følgende linje øverst i programmet: `print(Path.cwd())`
- Hvis cwd ikke er */path/to/inf100/lab7*: åpne VSCode i *lab7* -mappen (File -> Open Folder -> velg lab7 -mappen) og kjør programmet nå.
- Alternativt kan du navigere til *lab7* -mappen med kommandoer i terminalen og kjøre programmet derfra.

Det er *mulig* å programmatisk endre cwd til å bli samme mappe som filen som kjøres ligger i:

↑ Innhold

```
import os
directory_of_current_file = os.path.dirname(__file__)
os.chdir(directory_of_current_file) # endrer cwd
```

Dette kan kanskje gjøre ting lettere i utviklingsfasen og for raske og enkle formål, men er sannsynligvis *ikke* noe en erfaren programmerer ville ønsket seg, siden man da må flytte hele programmet hvis man vil bruke det i en annen mappe.

Enkel CSV -håndtering

En CSV-fil er en tekstfil som inneholder tabell-data. CSV står for «comma separated values», og det er nettopp det det er: en tekstfil hvor hver linje inneholder en rekke verdier som er separert med komma (eller et annet symbol). Hver linje i filen representerer en rad i tabellen, og skille-symbolet indikerer hvor skillet mellom de ulike kolonnene i tabllene er.

Regneark i Microsoft Excel eller Google Sheets kan lagres som CSV-filer. Dette er et vanlig format for å utveksle data mellom ulike programmer.

Innholdet i en CSV-fil (people.csv) kan se slik ut:

```
Navn,Alder,Høyde
Ola,20,1.80
Kari,19,1.65
Per,21,1.73
Oda,20,1.74
```

Det finnes en modul i standardbiblioteket som er spesielt laget for å lese CSV-filer (se avsnitt lengre nede), men i *dette* avsnittet skal vi vise hvordan vi kan lese dem helt selv. En CSV-fil er nemlig bare en tekstfil, og vi kan lese den på akkurat samme måte som vi leser andre tekstfiler.

↑ Innhold

```python
from pathlib import Path

##########################################
### LESE INPUT OG OPPRETTE DATASTRUKTUR ###
##########################################

# Les inn innholdet i filen 'people.csv' som en streng
content_string = Path('people.csv').read_text(encoding='utf-8')

# .splitlines klipper opp strengen ved linjeskift, og gir oss en
# liste med bitene som er igjen; altså radene i tabellen
content_lines = content_string.splitlines()

# Vi oppretter en 2D-liste (en liste av lister) som skal inneholde
# tabellen vår
table = []
for line in content_lines:
    # .split(',') klipper opp strengen ved komma, og gir oss en
    # liste med bitene som er igjen
    row = line.split(',')
    table.append(row)


# Vi kan nå aksessere enkeltverdier i tabellen vår ved å bruke
# indeksering på samme måte som vi gjør med andre 2D-lister
print(table[0][1]) # Alder
print(table[1][0]) # Ola
print(table[3][2]) # 1.73

# Ofte gir det mening å ha overskriftene og selve dataene i separate
# variabler.
headers = table[0] # første rad
data = table[1:]  # alle rader unntatt den første

##############################################
### UTFØR SELVE DATABEHANDLINGEN VI ØNSKER ###
##############################################

# Et år har passert! Øk alle aldre med 1 i datasettet.
for row in data:
    row[1] = 1 + int(row[1]) # PS: dette endrer typen fra string til int


############################
### PRESENTER RESULTATET ###
############################

result_headers = headers
result_headers_string = ','.join(result_headers)
result_lines = [result_headers_string]
for row in data:
```

```
    string_row = [str(x) for x in row]
    line_string = ','.join(string_row)
    result_lines.append(line_string)
result_string = '\n'.join(result_lines) + '\n'

Path('people_one_year_later.csv').write_text(result_string, encoding='utf-8')
```

CSV som 2D -liste eller liste av oppslagsverk

Når vi skal representere tabell-data i våre Python-programmer, finnes det i hovedsak to muligheter som ofte brukes. Vi illustrerer ved eksempel. La oss anta at vi har følgende tabell:

Navn	**Alder**	**Høyde**
Ola	20	1.80
Per	21	1.73
Eva	20	1.74

Vi kan representere tabellen som en CSV-fil:

```
'Navn','Alder','Høyde'
'Ola',20,1.80
'Per',21,1.73
'Eva',20,1.74
```

I Python -programmet vårt kan vi representere samme data som en 2D-liste:

```
table = [
    ['Navn', 'Alder', 'Høyde'],
    ['Ola', 20, 1.80],
    ['Per', 21, 1.73],
    ['Eva', 20, 1.74],
]
```

eller (mer hensiktsmessig) en 2D-liste med data pluss en vanlig liste med overskriftene:

```
headers = ['Navn', 'Alder', 'Høyde']
data = [
    ['Ola', 20, 1.80],
    ['Per', 21, 1.73],
    ['Eva', 20, 1.74],
]
```

Den siste representasjonen vi vil vise, er å representere innholdet som en liste av oppslagsverk:

↑ Innhold

```
headers = ['Navn', 'Alder', 'Høyde']
data = [
    {'Navn': 'Ola', 'Alder': 20, 'Høyde': 1.80},
    {'Navn': 'Per', 'Alder': 21, 'Høyde': 1.73},
    {'Navn': 'Eva', 'Alder': 20, 'Høyde': 1.74},
]
```

I mange tilfeller er det denne sistnevnte representasjonen som er å foretrekke. Denne representasjonen er lettere å lese og mer robust for endringer. Det er litt knotete (men slett ikke umulig) å konvertere en string til dette formatet på egen hånd; men vi kan også bruke csv-modulen fra Python sitt standardbibliotek.

CSV -modulen

*Grunnleggende håndtering av enkle CSV-filene kan vi gjøre med `.split` og `.join` som vi viser i notatene om enkel CSV-håndtering over. Men i noen tilfeller kan det bli mer komplisert: for eksempel når innholdet i en celle inkluderer selve **skillesymbolet** (komma). For å løse dette, blir innholdet i en celle som inneholder skillesymbolet omsluttet av et **sitattegn** for å angi hvor en celle begynner og slutter. Og hva om innholdet i cellen også inneholder selve sitattegnet? Dette løses ved å la to sitattegn på rad regnes som selve sitattegn-symbolet. Uansett: koden for å tolke slike kompliserte CSV-filer blir nokså komplisert. Heldigvis finnes `csv` -modulen som en del standardbiblioteket i Python, slik at vi slipper å forholde oss til detaljene*

Når vi tolker en CSV-fil må vi ta hensyn til at følgende parametre er konfigurert riktig:

- `delimiter` : Skilletegnet mellom cellene. Standard er komma (`,`). Semikolon, mellomrom og tabulator er også vanlige.
- `quotechar` : Sitattegn som brukes for å omslutte celler som inneholder skilletegnet. Standard er anførselstegn (`"`). Noen ganger brukes apostrof.
- `lineterminator` : Skilletegnet mellom radene. Standard er `\r\n` for skriving, og enten `\r\n` eller `\n` for lesing.

Merk at parametrene *delimiter*, *quotechar* og *lineterminator* skal angis når «leseobjektet» (eller «skriveobjektet») opprettes. Dersom de ikke angis, benyttes standardverdiene som nevnt over.

CSV <—> Liste med oppslagsverk

↑ Innhold

```python
# Fra CSV-formattert streng til liste med oppslagsverk
import csv
import io

# Eksempelstreng
my_string = '''\
'Navn','Alder','Høyde'
'Ola',20,1.80
'Per',21,1.73
'Eva',20,1.74
'''

# Konvertering til liste av oppslagsverk
reader = csv.DictReader(io.StringIO(my_string), delimiter=',', quotechar="'")
headers = reader.fieldnames
data = list(reader)

# headers = ['Navn', 'Alder', 'Høyde']
# data = [
#   {'Navn': 'Ola', 'Alder': '20', 'Høyde': '1.80'},
#   {'Navn': 'Per', 'Alder': '21', 'Høyde': '1.73'},
#   {'Navn': 'Eva', 'Alder': '20', 'Høyde': '1.74'},
# ]

# PS: merk at alle nøkler og verdier alle er strenger
```

```python
# Fra liste med oppslagsverk til CSV-formattert streng
import csv
import io

# Eksempeldata
headers = ['Navn', 'Alder', 'Høyde']
data = [
    {'Navn': 'Ola', 'Alder': 20, 'Høyde': 1.80},
    {'Navn': 'Per', 'Alder': 21, 'Høyde': 1.73},
    {'Navn': 'Eva', 'Alder': 20, 'Høyde': 1.74},
]

# Konvertering til CSV-streng
output = io.StringIO()
writer = csv.DictWriter(output, fieldnames=headers, lineterminator='\n')
writer.writeheader()
writer.writerows(data)
my_string = output.getvalue()

# my_string = 'Navn,Alder,Høyde\nOla,20,1.8\nPer,21,1.73\nEva,20,1.74\n'
```

CSV <—> 2D-liste

↑ Innhold

```python
# Fra CSV-formattert streng til 2D-liste
import csv
import io

# Eksempelstreng
my_string = '''\
'Navn','Alder','Høyde'
'Ola',20,1.80
'Per',21,1.73
'Eva',20,1.74
'''

# Konvertering til 2D-liste
reader = csv.reader(io.StringIO(my_string), delimiter=',', quotechar="'")
table = list(reader)
headers = table[0]
data = table[1:]

# headers = ['Navn', 'Alder', 'Høyde']
# data = [
#   ['Ola', '20', '1.80'],
#   ['Per', '21', '1.73'],
#   ['Eva', '20', '1.74'],
# ]

# PS: merk at alle verdier alle er strenger
```

```python
# Fra 2D-liste til CSV-formattert streng
import csv
import io

# Eksempeldata
headers = ['Navn', 'Alder', 'Høyde']
data = [
    ['Ola', 20, 1.80],
    ['Per', 21, 1.73],
    ['Eva', 20, 1.74],
]

# Konvertering til CSV-streng
output = io.StringIO()
writer = csv.writer(output, delimiter='&', lineterminator='\n')
writer.writerow(headers)
writer.writerows(data)
my_string = output.getvalue()

# my_string = 'Navn&Alder&Høyde\nOla&20&1.8\nPer&21&1.73\nEva&20&1.74\n'
```

Unicode og tekstkoding

↑ Innhold

- Unicode og ordinaler
- Tekstkoding
- Sammenblanding av tekstkodinger

Unicode og ordinaler

Som alle andre datatyper i Python er også strenger *egentlig* representert under panseret som en sekvens av **0** og **1**. For eksempel representeres bokstaven `'A'` som `1000001` og bokstaven `'B'` som `1000010`. Hvis vi i stedet for å tolke sekvensen av 0 og 1 som en *bokstav* later som sekvensen er et *tall* i totall-systemet, får vi henholdsvis 65 for `'A'` og 66 for `'B'`.

På denne måten er hvert eneste symbol (bokstav og tegn, emoji og andre symboler som kan opptre i en streng) knyttet til et tall. Dette tallet kalles en **ordinal** for symbolet. Hvordan symboler matcher ordinaler gjøres i Python på en standardisert måte som kalles *Unicode*. Unicode er med andre ord en matching mellom ordinal og symbol. Under viser vi et lite utdrag.

Ordinal	**Symbol**
65	`'A'`
66	`'B'`
67	`'C'`
…	…
90	`'Z'`
198	`'Æ'`
216	`'Ø'`
197	`'Å'`

Ordinal	**Symbol**
97	`'a'`
98	`'b'`
99	`'c'`
…	…
122	`'z'`
230	`'æ'`
248	`'ø'`
229	`'å'`

Ordinal	**Symbol**
9	`'\t'` (tab)
10	`'\n'` (linjeskift)
32	`' '` (mellomrom)
33	`'!'`
48	`'0'`
49	`'1'`
50	`'2'`
128013	`'🐍'`

For å finne ordinalen til et symbol kan vi bruke funksjonen `ord`:

```
symbol = 'A'
ordinal = ord(symbol)
print('Ordinal til', symbol, 'er' , ordinal) # Ordinal til A er 65
```

For å konvertere fra ordinal til symbol («character») kan vi bruke funksjonen `chr`:

↑ Innhold

```
ordinal = 97
symbol = chr(ordinal)
print('Ordinal', ordinal, 'har symbol', symbol) # Ordinalen 97 har symbol a
```

Tekstkoding

Ett symbol kan representeres som ett tall, som vist i forrige avsnitt. Men hva om det er flere symboler etter hverandre, som i en streng eller en fil med tekst? Fordi det ikke eksisterer noe naturlig «mellomrom» i noe som representeres som en sekvens av **0** og **1**, må vi bestemme oss for noen regler for å skille hvor ett symbol slutter fra hvor det neste starter.

Det finnes flere ulike strategier for dette, som vi kaller *koding* av en streng (engelsk: *encoding*). Kodinger som støtter alle Unicode-symboler begynner med «UTF» (Unicode Transformation Format).

Eksempler på kodinger:

🕮 ASCII

ASCII er en gammel standard som Unicode-ordinalene er bakoverkompatibel med. Den deler opp sekvensen av 0'er og 1'ere i blokker på akkurat 8 biter, og så tolker den hver blokk som en ordinal oppgitt i totallsystemet. Hver blokk begynner alltid med 0. Eksempler noen ulike symboler og hvordan de kodes i ASCII:

Symbol	Unicode	Koding i ASCII
\n	10	`00001010`
A	65	`01000001`
B	66	`01000010`
Æ	198	ikke støttet
🐍	128013	ikke støttet

Den delen av kodingen som er markert i gult over er selve ordinalen (i totallsystemet). Nullene som kommer foran dette bare fyller opp plassen slik at blokken får størrelse på 8 bit.

- Fordeler med *ASCII*: lett å forstå og implementere. Bruker lite lagringsplass. Støttes også av svært gamle systemer.
- Ulemper med *ASCII*: Støtter kun symboler med unicode-ordinal under 128. Altså ingen støtte for norske bokstaver æ, ø og å.

🕮 UTF-32

↑ Innhold

UTF-32 er en koding som er enkel å forstå (men som i praksis er lite brukt). Den deler opp sekvensen av 0'er og 1'ere i blokker på akkurat 32 biter, og så tolker den hver blokk som en ordinal oppgitt i totallsystemet (akkurat som ASCII, altså, men tar større plass). Eksempler noen ulike symboler og hvordan de kodes i UTF-32:

Symbol	Unicode	Koding i UTF-32
\n	10	`00000000000000000000000000001010`
A	65	`00000000000000000000000001000001`
B	66	`00000000000000000000000001000010`
Æ	198	`00000000000000000000000011000110`
🐍	128013	`00000000000000011111010000001101`

Den delen av kodingen som er markert i gult over er selve ordinalen (i totallsystemet). Nullene som kommer foran dette bare fyller opp plassen slik at blokken får størrelse på 32 bit.

- Fordeler med *UTF-32*: lett å forstå og implementere. Man kan raskt finne ut hvilken bokstav som er i en gitt posisjon. Støtter alle Unicode-symboler.
- Ulemper med *UTF-32*: bruker mye unødvendig lagringsplass. Ikke bakoverkompatibel med ASCII.

📖 UTF-8

↑ Innhold

UTF-8 er den vanligste unicode-kodingen. Den deler opp sekvensen av 0'er og 1'ere i blokker på 8 biter; de vanligste symbolene bruker bare én slik blokk, mens de mer sjeldne bruker flere blokker.
Eksempler på hvordan noen ulike symboler blir kodet i UTF-8:

Symbol	Unicode	Koding i UTF-8
\n	10	`00001010`
A	65	`01000001`
B	66	`01000010`
Æ	198	`11000011` `10000110`
🐍	128013	`11110000` `10011111` `10010000` `10001101`

Den delen av kodingen som er markert i gult over er selve ordinalen (i totallsystemet). Den delen av kodingen som er markert i rødt inneholder informasjon som UTF-8 bruker for å avgjøre hvor mange blokker symbolet består av. De øvrige nullene bare fyller opp plassen slik at hver blokk får en størrelse på 8 bit.

Fordeler med *UTF-8*

- Bakoverkompatibel med den eldre standarden ASCII.
- Bruker i praksis mye mindre lagringsplass enn UTF-32.
- Støtter alle Unicode-symboler.
- Er de facto standard på internett (HTTP).

📖 cp1252 (og latin-1/iso-8859-1)

↑ Innhold

Tekstkodingen **cp1252** (også kalt *Windows-1252* og noen ganger upresist referert til som *ISO 8859-1* eller *Latin 1*) er en tekstkoding som (dessverre) er standard i noen Microsoft-produkter på Windows fremdeles. Enkodingen er i likhet med UTF-8 bakoverkompatibel med ASCII, men er likevel *ikke* basert på unicode 😡 . Selv om den støtter noen flere symboler enn ASCII (som f. eks. de norske bokstavene æ, ø og å), har den likevel et svært begrenset utvalg av mulige symboler i forhold til unicode-baserte tekstkodinger.

Et symbol i cp1252 er kodet som en sekvens av 0'er og 1'ere i blokker på akkurat 8 biter. Eksempler på hvordan noen ulike symboler blir kodet i cp1252:

Symbol	Ordinal	Koding i cp1252
\n	10	`00001010`
A	65	`01000001`
B	66	`01000010`
Æ	198	`11000110`
🐍	128013	ikke støttet

Fordeler med *cp1252*

- Bakoverkompatibel med ASCII.
- Bruker lite lagringsplass.
- Støtter de norske bokstavene æ, ø og å.

Ulemper med *cp1252*

- Er ikke basert på unicode; for eksempel er ordinalen for `€` (euro-tegnet) 128 i cp1252, mens det i unicode er 8364.
- Det er kun støtte for 256 ulike symboler (dobbelt så mange som ASCII, men likevel veldig lite i forhold til unicode-baserte tekstkodinger).

Forskjellen på cp1252 og latin-1/iso-8859-1 er minimal, men cp1252 er bakoverkompatibel med latin-1 og støtter noen få ekstra symboler, for eksempel skråstilte anførselstegn.

Når man leser eller skriver en tekstfil, må man velge hvilken koding man skal benytte.

- For å **skrive** til fil er valget enkelt: du bør alltid velge **UTF-8** med mindre du har helt spesielle grunner til å gjøre noe annet (f. eks. kompabilititet med et gammelt system).
- For å **lese** fra en fil må du vite hvilken koding som ble brukt da filen ble lagret. Med 80% sannsynlighet er dette UTF-8, men av og til kan det være noe annet – for eksempel er cp1252 ikke helt uvanlig hvis filen ble opprettet på en Windows-maskin av noen som ikke visste helt hva de gjorde. Hvis du ikke vet hva tekstkodingen er, må du dessverre gjette deg frem eller spørre den som har laget filen. Hvis du får rare tegn i stedet for norske bokstaver, er det sannsynligvis fordi du har gjettet feil. Da kan du prøve å gjette på en annen koding.

↑ Innhold

Hvis du ikke vet hvilken koding som er brukt, prøv disse tekstkodingene først:

- *UTF-8* (anbefalt)
- *latin-1* (også kalt *ISO-8859-1*)
- *cp1252* (også kalt *Windows-1252*)
- *UTF-16*
- *UTF-32*

Hvis teksten er på et spesielt språk og ingen av unicode-kodingene (utf-XX) fungerer, kan du også søke på internett etter kodinger som er vanlige for språket. Det er en stor jungel av tekstkodinger tilpasset symboler fra forskjellige språk: for eksempel er *cp1251* (også kalt *Windows-1251*) et alternativ til cp1252 som er tilpasset kyrillisk tekst (russisk etc.), mens *cp865* (også kalt *IBM865*) er et sjeldent benyttet alternativ som er spesielt tilpasset norsk og dansk. Liste over alle tekstkodinger som støttes av Python finner du her (https://docs.python.org/3/library/codecs.html#standard-encodings).

```
from pathlib import Path

# Eksempel på å skrive til en fil
text_w = 'Dette er en tekst. Den har æøå og 🐍 i seg.'
Path('myfile.txt').write_text(text_w, encoding='utf-8')

# Eksempel på å lese fra en fil
text_r = Path('myfile.txt').read_text(encoding='utf-8')
print(text_r) # Dette er en tekst. Den har æøå og 🐍 i seg.
```

Dersom du ikke angir noe for `encoding=` vil Python bruke standarden for ditt operativsystem. Dette er vanligvis `utf-8` på Mac og Linux, og `cp1252` på Windows, men kan også variere basert på «locale» - konfigurasjonen av operativsystemet (språk, etc.). Det er derfor lurt å alltid spesifisere `encoding=` når du skriver til/leser fra filer, slik at du ikke får problemer når du bytter datamaskin.

Sammenblanding av tekstkodinger

Dersom man åpner en fil med feil tekstkoding, vil innholdet tolkes feil, eller det kan oppstå feilmeldinger.

Anta at vi har tekstfiler sample_utf8.txt og sample_cp1252.txt som begge inneholder samme innhold, men er lagret med ulike tekstkodinger (høyreklikk på link og velg «lagre link som» eller lignende for å laste ned). Innholdet i filene er identisk:

```
blåbærsyltetøy
```

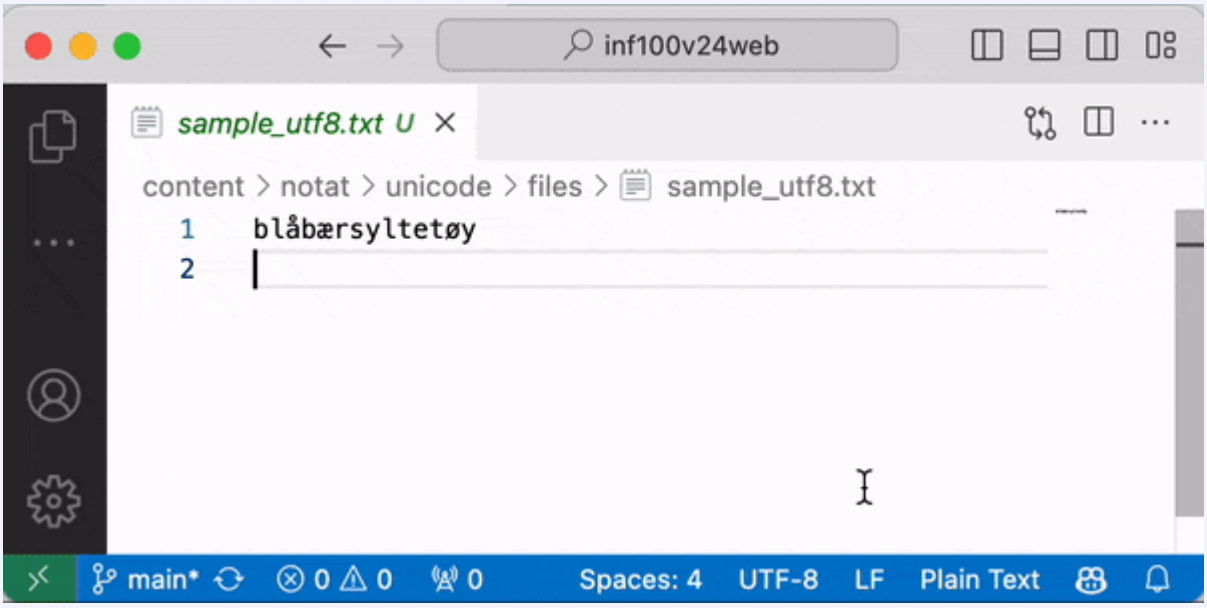
Avhengig av hvilken tekstkoding vi bruker når vi *leser* filene, vil vi få ulike resultater (kan avvike noe avhengig av hvilket program du benytter for å lese filene):

↑ Innhold

	Lest med UTF-8	**Lest med cp1252**
Fil lagret med UTF-8	`blåbærsyltetøy`	`blÃ¥bÃ¦rsyltetÃ¸y`
Fil lagret med cp1252	`bl�b�rsyltet�y`	`blåbærsyltetøy`

Prøv å lese begge filene over i nettleseren (klikk på linkene over på vanlig måte uten å laste dem ned). Hvilken av filene ser riktig ut? Hvilken ser feil ut? Hvorfor? Hvilken tekstkoding benytter nettleseren når den leser filene?

Dersom du åpner en tekstfil i VSCode vil programmet vise på linjen nederst til høyre hvilken tekstkoding som benyttes for å lese filen som vises (f. eks. UTF-8 eller Windows-1252). Om du klikker der, kan du velge å åpne eller lagre filen med en annen tekstkoding.



Tips: for best mulig kompatibilitet, bør du alltid velge UTF-8 når du lager nye filer. Hvis du ser at filen din ser riktig ut i en annen tekstkoding, anbefaler jeg å velge «Save with encoding» og lagre filen på nytt med UTF-8. Dette gjelder også kildekoden til Python-filer du skriver.

Når vi i våre Python-program leser filer og spesifiserer feil tekstkoding, får vi akkurat samme problemer som vi ser over. Merk:

- å lese med cp1252 en fil som egentlig er kodet i utf-8 krasjer ikke, men vil feile i stillhet;
- å lese med utf-8 en fil som egentlig er kodet i cp1252 vil ofte (men ikke alltid) resultere i at programmet krasjer.

↑ Innhold

```python
from pathlib import Path

print(Path('sample_utf8.txt').read_text(encoding='utf-8')) # blåbærsyltetøy
print(Path('sample_utf8.txt').read_text(encoding='cp1252')) # blÃ¥bÃ¦rsyltetÃ¸y
print(Path('sample_cp1252.txt').read_text(encoding='utf-8')) # --> krasjer <--
print(Path('sample_cp1252.txt').read_text(encoding='cp1252')) # blåbærsyltetøy
```

Fordi både UTF-8, cp1252 og latin-1 er bakoverkompatible med ASCII, vil filer som kun inneholder ASCII-symboler være helt identiske enten de er lagret med UTF-8 eller cp1252 eller latin-1. Dette er grunnen til at man av og til ønsker å begrense seg til å kun bruke ASCII-symboler såfremt det ikke medfører noen andre ulemper.

Håndtere krasj

↑ Innhold

Krasj av programmet en *bra* ting, fordi vi ønsker å vite at noe er feil så fort som mulig. Men i noen tilfeller ønsker vi at programmet skal håndtere krasjen selv; dette gjelder egentlig bare når vi vet på forhånd hva slag krasj som kan oppstå, og er *ikke* et lurt triks å bruke dette for å skyve problemer under teppet.

Generelt vil jeg anbefale å være sparsom med bruken av try og except; hvis det kan håndteres uten på en enkel og grei måte, er det som oftest å foretrekke. Kode som er basert på mye try og except i kontrollflyten er litt mer utfordrende å feilsøke. Samtidig, dersom å bruke try/except sparer mye omstendelig kode kan det være å foretrekke likevel.

- Krasjhåndtering med try/except
- Stilguide for krasjhåndtering
- Krasje på egen hånd

Krasjhåndtering med try/except

```
# Håndtere en krasj
#  -- Prøv å gi programmet noe som ikke er et tall
#  -- Prøv å gi programmet et tall som er for stort
# Se: programmet krasjer ikke selv om brukeren gir dårlig input svar!

animals = ["katt", "hund", "kanin", "hamster", "krokodille"]
user_input = input(f"Velg ett tall [0-{len(animals) - 1}]:")

try:
    i = int(user_input)
    animal = animals[i]
except:
    # Kjøres dersom try-blokken krasjet
    print("Ugyldig valg!")
else:
    # Kjøres dersom try-blokken gikk bra
    print("Gratulerer, du fikk en ny", animal)

print("Nå er programmet ferdig")
```

Bruk krasjhåndtering (try/except) med varsomhet. Å bruke mye krasjhåndetring kan gjøre koden din litt vanskeligere å feilsøke.

```
# FARE!! bruk av except: håndterer for mange krasjer --> vanskelig å feilsøke!
#  -- Prøv nå å gi programmet en GYLDIG tall som input
# Se: programmet krasjer ikke, men gjør i stedet en logisk feil! FYFYFY!

animals = ["katt", "hund", "kanin", "hamster", "krokodille"]
user_input = input(f"Velg ett tall [0-{len(animals) - 1}]:")

try:
    i = int(user_input)
    animal = animal[i] # Skrivefeil (s mangler)! Vi VIL krasje her med
NameError!
except:
    print("Ugyldig valg!") # Oops! Kommer hit selv om input er gyldig
else:
    print("Gratulerer, du fikk en ny", animal)

print("Nå er programmet ferdig")
```

Man bør alltid spesifisere *hvilken type krasj* man håndterer.

```
# Spesifiser hvilken type krasj du håndterer
#
#  -- Prøv å gi programmet en GYLDIG tall som input (se: krasjer)
#  -- Fiks kodefeilen (skrivefeilen) ved å rette koden der det krasjer
#  -- Prøv nå å gi programmet et tall som er for stort
#  -- Prøv nå å gi programmet noe som ikke er et tall

animals = ["katt", "hund", "kanin", "hamster", "krokodille"]
user_input = input(f"Velg ett tall [0-{len(animals) - 1}]:")

try:
    i = int(user_input)
    animal = animal[i] # Skrivefeil (s mangler) -- men nå krasjer vi! YAY!
except ValueError:
    print("Ugyldig valg, du må oppgi et tall!")
except IndexError:
    print("Tallet du oppgav er ugyldig!")
else:
    print("Gratulerer, du fikk en ny", animal)

print("Nå er programmet ferdig")

# NameError blir ikke fanget av except nå,
# så vi oppdager det nå hvis variabeler har feil navn.
```

↑ Innhold

Stilguide for krasjhåndtering

- *Bruk krasjhåndtering for å håndtere **andre** sine feil; ikke for å skjule **egne** feil i koden.*
  - *Eksempler på andre sine feil: en bruker skriver inn ugyldig input, filen du prøver å lese fra finnes ikke eller har feil innhold, nettverket er nede, etc.*
  - *Eksempler på egne feil: du har skrevet feil variabelnavn, du har gitt feil argumenter til en funksjon, du ufører en beregning på gal måte etc.*
- *Hvis du enkelt kan løse problemet uten try/except, er det ofte en bedre løsning.*
  - *For eksempel: benytt if-setninger for å håndtere hjørnetilfeller som er enkle å sjekke.*
- *Ha **minst mulig** kode i try-blokken.*
  - *Flytt så mye kode som mulig utenfor try-blokken: enten før try-blokken begynner eller inn i else-blokken.*
- *Alltid angi **hvilken type** feil du håndterer.*

Krasje på egen hånd

Kodeordet `raise` brukes for å krasje på egen hånd. Dette kan brukes for å krasje med en feilmelding som gir mer detaljert informasjon enn ellers, eller for å krasje programmet så tidlig som mulig dersom noe er galt.

*Å krasje med `raise` gir noenlunde samme funksjonalitet som å skrive `assert False` (se kursnotater om feil og debugging). Forskjellen ligger primært i at du kan lage flere ulike **typer** feil med `raise`. En krasj forårsaket av `assert False` vil alltid krasje med typen `AssertionError`.*

```
# Krasj programmet dersom input ikke er gyldig
food = input()
if food not in ["salat", "tomat", "agurk", "paprika"]:
    raise ValueError(f"Maten '{food}' er ikke akseptabel.")

print("Takk for maten!")
```

```python
# Krasj programmet dersom argumenter ikke er gyldige

def process_payment(amount, card_number):
    if amount <= 0:
        raise ValueError("Cannot process payment for amount <= 0")
    if len(card_number) != 16:
        raise ValueError("Card number must be 16 digits long")

    ... # do the actual payment processing here
    print("Payment processed successfully")

process_payment(-40, "1234567890123456")
```

Kan brukes for å gi ekstra informasjon ved krasj

```python
def foo(i, j):
    y = j
    for x in range(i, j):
        try:
            y += abs(y/x)
        except ZeroDivisionError as err: # err variabel som 'husker' krasjen
            # Skriver ut debug-informasjon
            print("Divisjon med 0")
            print("Lokale variabler: ", locals())
            raise err # Kaster samme krasj på nytt
    return x

print(foo(-5, 10))
```

Mengder

↑ Innhold

- Enkelt eksempel
- Opprette mengder
- Egenskaper ved mengder
- Operasjoner på mengder
- Frozenset

Se også offisiell dokumentasjon (https://docs.python.org/3/library/stdtypes.html#set) for `set`.

Enkelt eksempel

En *mengde* (engelsk: set) er en datastruktur som kan holde på mange verdier (i likhet med lister, tupler og oppslagsverk). Det er noen forskjeller:

- verdiene i en mengde har ikke en posisjon, og er derfor ikke tilknyttet en indeks eller nøkkel, og
- det er ikke mulig å ha flere like verdier i en mengde.

Med en mengde kan man i hovedsak gjøre fire ting:

- legge en verdi inn i mengden
- fjerne en verdi fra mengden
- spørre om en verdi er i mengden (veldig effektivt!), og
- se gjennom verdiene i mengden.

```
# Opprett en mengde
s = {2, 3, 5}

# Legg til en verdi i mengden (muterer mengden)
s.add(6)

# Antall elementer i mengden
print(len(s))  # 4

# Spør om en verdi er i mengden
print(3 in s)  # True
print(4 in s)  # False

# Se gjennom verdiene i mengden
for x in s:
    print(x, end=' ')  # 2 3 5 6
print()

# Fjern en verdi fra mengden (muterer mengden)
s.discard(3)
print(s)  # {2, 5, 6}
```

↑ Innhold

Opprette mengder

```
# Opprett en tom mengde
s = set()
print(s)
```

```
# PS: MISLYKKET forsøk på å opprette en tom mengde:
s = {}  # dette oppretter et oppslagsverk, ikke en mengde!
print(type(s))
```

```
# Opprett en mengde statisk (med verdier angitt direkte i kildekoden)
s = {2, 3, 5}
print(s)
```

```
# Opprett en mengde fra en liste (eller annen samling med elementer)
# Legg merke til at mengden kun inneholder hvert element én gang
a = [2, 3, 3, 5]
s = set(a)
print(s)  # {2, 3, 5}

greeting = 'hello'
required_letters = set(greeting)
print(required_letters)  # {'h', 'e', 'l', 'o'}
```

```
# Mengdebygging.
# Vi tar utgangspunkt i en annen samling (her a), og bruker deretter
# benytte en mengdebygger-for-løkke mellom krølleparentesene
a = ['foo', 'bar', 'baz', 'qatchita']
myset = {s[0] for s in a}
print(myset)  # {'f', 'b', 'q'}

# Vi kan også legge til en betingelse for inklusjon etter 'if'
myset = {s[0] for s in a if len(s) <= 3}
print(myset)  # {'f', 'b'}
```

Egenskaper ved mengder

Elementene i en mengde har ingen rekkefølge. Når du går igjennom elementene i en mengde med en for-løkke eller du printer ut mengden vil det selvfølgelig være en eller annen rekkefølge, *men*: du kan ikke gjøre noen antakelser om hvilken rekkefølge dette er. Den kan variere fra maskin til maskin og Python-versjon til Python-versjon og fra kjøring til kjøring.

To mengder er ansett for å være like dersom de inneholder de samme elementene, uavhengig av hvilken rekkefølge elementene ble lagt til i mengdene.

↑ Innhold

```
s = set()
s.add(2)
s.add(44)
s.add(11)
s.add(5)
s.add(33)

for e in s:
    print(e, end=' ')  # Rekkefølgen kan være ulik fra maskin til maskin
print()

print({2, 3, 5} == {5, 3, 2}) # True
```

Elementer er unike (én verdi finnes bare én gang i mengden).

```
# Duplikater blir borte
s = set([2, 2, 2])
print(s)          # {2}
print(len(s))     # 1
```

Mengder kan muteres.

```
s = {2, 3, 5}
alias = s

s.add(9)
print(s) # {2, 3, 5, 9}
print(alias) # {2, 3, 5, 9}
```

Elementene i en mengde må *ikke* være mulig å mutere.[1]

```
s = set()
s.add(42)       # int OK
s.add('foo')    # str er OK
s.add(False)    # bool er OK
s.add(1.4)      # float er OK
s.add((2, 3))   # tupler er OK
print(s)

s.add([2, 3])   # Krasj! lister er IKKE OK (lister kan muteres)
s.add({2, 3})   # Ville også krasjet! (mengder kan også muteres)
```

Mengder er *svært* effektive.

↑ Innhold

```
# En liste kan brukes for samme formål som en mengde. La oss sammenligne
# hvor effektive de er til oppgaven «spør om en verdi er tilstede»
n = 2000
trails = 1000 # Flere forsøk utjevner forskjeller som skyldes forstyrrelser
a = list(range(n))
s = set(range(n))

import time
time_before = time.time()
for _ in range(trails):
    does_contain_minus_one = -1 in a
time_after = time.time()
elapsed_a = (time_after - time_before) * 1000
print(f'Det tok {elapsed_a:.0f}ms å sjekke listen {trails} ganger')

time_before = time.time()
for _ in range(trails):
    does_contain_minus_one = -1 in s
time_after = time.time()
elapsed_s = (time_after - time_before) * 1000
print(f'Det tok {elapsed_s:.0f}ms å sjekke mengden {trails} ganger')
ratio = elapsed_a/elapsed_s
print(f'Mengder var {ratio:.1f} ganger raskere enn lister for {n=}')
print('Prøv større verdi for `n` for å se større forskjeller')
```

Operasjoner på mengder

Det finnes flere måter å manipulere mengder på. For hver av operasjonene her finnes det destruktive metoder som muterer mengden vår, i tillegg finnes også ikke-destruktive alternativer som oppretter en helt nytt objekt i minnet. Det destruktive alternativet vil ofte være mer effektivt med tanke på minnebruk og kjøretid – samtidig er det av og til nødvendig for korrektheten i programmet ditt for øvrig at du benytter en ikke-destruktiv variant.

Se mer detaljer om operasjoner på mengder i den offisielle dokumentasjonen (https://docs.python.org/3/library/stdtypes.html#set).

↑ Innhold

Legg til elementer (add/update/union/ `|`)

```
# Legg til elementer
# destruktivt (ved mutasjon)
myset = {1, 2}
alias = myset

myset.add(6)         # ett
myset.update([2, 8]) # flere
myset |= {1, 2, 9}   # flere

print(myset) # {1, 2, 6, 8, 9}
print(alias) # {1, 2, 6, 8, 9}
```

```
# Legg til elementer
# ikke-destruktivt (nytt objekt)
myset = {1, 2}
alias = myset


myset = myset.union([2, 8])
myset = myset | {1, 2, 9}

print(myset) # {1, 2, 8, 9}
print(alias) # {1, 2}
```

Fjerne elementer (remove/discard/difference/ `–` /mengdebygging)

```
# Fjern elementer
# destruktivt (ved mutasjon)
myset = {1, 2, 3, 4, 5}
alias = myset

# 'remove' fjerner ett element
# (ikke funnet -> krasjer)
myset.remove(1)

# 'discard' fjerner ett element
# (ikke funnet -> ignorer)
myset.discard(2)
myset.discard(42)

# difference_update/-= fjerner
# flere elementer hvis de finnes
myset.difference_update([4, 6])
myset -= {5, 7, 9}

print(myset) # {3}
print(alias) # {3}
```

```
# Fjern elementer
# ikke-destruktivt (nytt objekt)
myset = {1, 2, 3, 4, 5}
alias = myset




# Mengdebygging med betingelse
myset = {x for x in myset if x != 2}
myset = {x for x in myset if x !=
42}

# difference/- fjerner elementer
# hvis de finnes
myset = myset.difference([4, 6])
myset = myset - {5, 7, 9}

print(myset) # {1, 3}
print(alias) # {1, 2, 3, 4, 5}
```

↑ Innhold

Beholde elementer (intersection/ `&`)

```
# Behold kun elementer som er
# i begge mengdene.
# destruktivt (ved mutasjon)
myset = {1, 2, 3, 4, 5}
alias = myset

a = [3, 4, 5, 6, 7]
myset.intersection_update(a)
# myset er nå {3, 4, 5}

myset &= {1, 3, 5, 7, 9}

print(myset) # {3, 5}
print(alias) # {3, 5}
```

```
# Behold kun elementer som er
# i begge mengdene.
# ikke-destruktivt (nytt objekt)
myset = {1, 2, 3, 4, 5}
alias = myset

a = [3, 4, 5, 6, 7]
myset = myset.intersection(a)
# myset er nå {3, 4, 5}

myset = myset & {1, 3, 5, 7, 9}

print(myset) # {3, 5}
print(alias) # {1, 2, 3, 4, 5}
```

Symetrisk forskjell (symmetric_difference/ `^`)

```
# Behold kun elementer som er
# i akkurat én av mengdene
# destruktivt (ved mutasjon)
myset = {1, 2, 3, 4, 5}
alias = myset

a = [3, 4, 5, 6, 7]
myset.symmetric_difference_update(a)
# myset er nå {1, 2, 6, 7}

myset ^= {1, 3, 5, 7, 9}

print(myset) # {2, 3, 5, 6, 9}
print(alias) # {2, 3, 5, 6, 9}
```

```
# Behold kun elementer som er
# i begge mengdene.
# ikke-destruktivt (nytt objekt)
myset = {1, 2, 3, 4, 5}
alias = myset

a = [3, 4, 5, 6, 7]
myset = myset.symmetric_difference(a)
# myset er nå {1, 2, 6, 7}

myset = myset ^ {1, 3, 5, 7, 9}

print(myset) # {2, 3, 5, 6, 9}
print(alias) # {1, 2, 3, 4, 5}
```

Frozenset

Det finnes en type mengder som ikke kan muteres, kalt `frozenset` . De fungerer nøyaktig som `set` , men operasjonene som ville mutert set vil nå enten opprette et nytt objekt eller det vil krasje.

Fordelen med *frozenset* er at de kan brukes som nøkler i oppslagsverk, eller de kan legges som elementer i andre mengder (siden de ikke kan muteres, tilfredstiller de kravet).

↑ Innhold

```
# Opprett et frozenset
myset = frozenset({2, 3, 5})
alias = myset

myset |= {42, 43} # muterer ikke, men oppretter nytt objekt

print(myset) # {2, 3, 5, 42, 43}
print(alias) # {2, 3, 5}

myset.discard(2) # Krasjer
```

---

1. Det er ikke *heelt* sant at elementene i en mengde ikke kan muteres, men det er en hvit løgn vi lever godt med i INF100. ↩

Moduler

↑ Innhold

- Ordbok: modul, pakke, bibliotek og rammeverk
- Importere moduler
- Egne moduler
- Hovedfil og modul

Ordbok: modul, pakke, bibliotek og rammeverk

- En *modul* er en samling med relaterte funksjoner og variabler man kan importere. For eksempel `random` og `math` som vi har vært borti tidligere i kurset. Du kan tenke på en modul som én «fil» med kode.
- En *pakke* er en samling av moduler (og av og til andre, mindre pakker) som er relatert til hverandre. Du kan tenke på en pakke som en slags «mappe» med relatert kode.
- Et *bibliotek* er egentlig bare en kjempestor pakke. Det er teknisk sett ingen forskjell på et bibliotek og en pakke, men dersom pakken blir veldig stor og generell, kalles det ofte et bibliotek. Hvor grensen går mellom pakke og bibliotek er litt opp til øyet som ser.
- Et relatert begrep er et *rammeverk*. Et rammeverk kan ta form av alt fra en modul til et bibliotek, men har den egenskapen at det legger «rammene» for hvordan koden skal skrives; for eksempel er `uib_inf100_graphics` ikke bare en pakke men også et rammeverk, siden vi er nødt til å strukturere koden vår på en spesiell måte for å bruke det. Et rammeverk kan kjennes igjen ved at andre sin kode kaller på funksjonene vi skriver, og ikke bare omvendt.

Importere moduler

Det finners flere måter å importere en modul på.

```
# Standard import
# For å bruke ting fra modulen, skriver vi modulnavn.navnpåting
import math
print(math.ceil(1.1))
print(math.pi)
```

```
# Import med nytt navn
# Kjekt hvis man vil bruke modulnavnet til et annet formål, eller
# hvis man har lyst på et kortere kallenavn på modulen
import math as ma
math = 42
print(ma.ceil(1.1))
print(ma.pi)
print(math)
```

↑ Innhold

```
# Import av kun spesifikke ting
# Man slipper bruke modulnavn først
# Man kan gi kallenavn til hver enkelt ting med `as`
from math import ceil as round_up, pi
print(round_up(1.1))
print(pi)
print(math.floor(1.9)) # Krasjer
print(floor(1.9)) # Ville også krasjet
```

```
# Import av alle tingene i math.
# FARE! DETTE BØR IKKE GJØRES! MAN HAR INGEN KONTROLL PÅ HVA MAN IMPORTERER!
# PLUTSELIG KAN FUNKSJONER HA FÅTT EN HELT NY BETYDNING VI IKKE FORVENTER!
from math import *
print(ceil(1.1))
print(pi)
print(floor(1.9))
```

Egne moduler

Å lage sin egen modul er så enkelt som å lagre en python-fil. For eksempel, vi kan ha to python-filer *foo.py* og *bar.py*, hvor sistnevnte importerer førstnevnte som modul:

```
# foo.py
def hello():
    return "Hello there"
```

```
# bar.py
import foo
print(foo.hello())
```

For at dette skal virke, er det viktig at foo.py og bar.py ligger i samme mappe. Det er også mulig å importere moduler som er lagret i undermapper. Dersom vi flytter *foo.py* inn i en mappe *lib* slik at bar ligger i samme mappe som lib, kan vi importere foo slik:

```
# bar.py
import lib.foo
print(lib.foo.hello())
```

eller

↑ Innhold

```python
# bar.py
from lib import foo
print(foo.hello())
```

Hovedfil og modul

Når en modul importeres, kjøres all koden som er i filen. For eksempel,

```python
# foo.py
def hello():
    return "Hello there"

print("Tester hello i foo:", hello())
```

```python
# bar.py
from foo import hello # Oops, skriver ut 'Tester hello i foo: Hello there'
print(hello())
```

Det kan være praktisk at en modul kan benyttes som en selvstendig enhet, men også kan importeres uten at det da kommer utskrift til terminalen eller andre merkelige sideeffekter. Til dette kan vi sjekke om en spesiell variabel som heter `__name__` har den spesielle verdien `"__main__"`:

```python
# foo.py
def hello():
    return "Hello there"

if __name__ == "__main__":
    # Denne koden kjøres når foo.py blir kjørt som hovedfil, men ikke
    # når foo.py blir importert som modul
    print("Tester hello i foo:", hello())
```

Standardbiblioteket

↑ Innhold

Python har mange moduler som er innebygget i selve språket, men som likevel ikke er umiddelbart tilgjengelig uten av vi importer dem først. Slike moduler er en del av Python sitt *standardbibliotek*, og inkluderer moduler som `math`, `random`, `copy`, `time`, `datetime`, `csv`, `decimal`, `sys`, `os` og mange andre. Se docs.python.org/3/library (https://docs.python.org/3/library/) for en fullstendig oversikt.

For å bruke en modul fra python sitt standardbibliotek, holder det å skrive `import <modulnavn>`. Konvensjon tilsier at dette gjøres øverst i filen.

Under viser vi frem et par eksempler fra noen utvalgte moduler fra Python sitt standardbibliotek.

Matematikk

- math
- random

Dato og tid

- time
- datetime

Håndtere vanlige filformater

- csv
  - Konvertere mellom CSV-fil og 2D-liste
  - DictReader og DictWriter: data som liste av oppslagsverk
  - DictReader med CSV-formatterte strenger
- json

Interaksjon med operativsystemet og filstrukturen

- math
- random
- time
- datetime
- csv
- json
- sys
- os og shutil
- pathlib

---

math

https://docs.python.org/3/library/math.html

↑ Innhold

```python
import math

# Noen utvalgte konstanter
print(math.pi)                 # 3.141592653589793
print(math.e)                  # 2.718281828459045
print(math.inf)                # uendelig
print(-math.inf)               # minus uendelig
print()

# Noen utvalgte funksjoner
print(math.ceil(3.22))         # 4, runder alltid av oppover
print(math.floor(3.9))         # 3, runder alltid av nedover
print(math.radians(180))       # 3.14..., konverter grader til radianer
print(math.degrees(math.pi/2)) # 90.0, konvertere radianer til grader
print(math.cos(math.pi))       # -1.0, cosinus-funksjonen
print(math.factorial(4))       # 24, faktorial-funksjonen (24 = 1*2*3*4)
```

random

https://docs.python.org/3/library/random.html

```python
import random

# Et tilfeldig flyttall mellom 0 og 1
x = random.random()
print(x)

# Et tilfeldig element fra en liste/samling
a = ['foo', 'bar', 'baz']
s = random.choice(a)
print(s)

# Et tilfeldig tall mellom 0 og 9
y = random.randrange(10) # ca det samme som random.choice(range(10))
print(y)
```

Ved å sette et «frø» kan vi få samme sekvens av «tilfeldige» tall hver gang vi kjører programmet. Dette kan være nyttig for testing og debugging, eller hvis du ønsker å kunne gjenskape et eksperiment. Som standard er frøet satt til systemtiden (https://en.wikipedia.org/wiki/System_time): antall nanosekunder siden 1. januar 1970 delt på 100.

↑ Innhold

```python
import random
random.seed(42) # Vi setter frøet til 42 (kan være hva som helst)

# Kjør programmet mange ganger og observer: de samme valgene gjentar
# seg hver gang man kjører programmet på nytt med samme frø.
a = ['foo', 'bar', 'quz']
print(random.choice(a))
print(random.choice(a))
print(random.choice(a))
print(random.choice(a))
```

Random-modulen har massevis av mer avanserte muligheter også, for eksempel å velge tilfeldige tall fra en rekke ulike fordelinger (uniform, normal, gammavariat, etc.). Det finnes også funksjoner for å velge flere tilfeldige elementer fra en samling med og uten tilbakelegging (henholdsvis `random.choices` og `random.sample`), samt for å blande en samling (med `random.shuffle`). Se offisiell dokumentasjon (https://docs.python.org/3/library/random.html) for mer informasjon.

time

https://docs.python.org/3/library/time.html

```python
import time

print(time.time()) # Antall sekunder siden 1. januar 1970 som flyttall
```

datetime

https://docs.python.org/3/library/datetime.html

↑ Innhold

```python
from datetime import datetime, timedelta

# Et datetime-objekt representerer et bestemt tidspunkt
lecture_starts = datetime(2024, 3, 22, 12, 15, 00)
print(f"{lecture_starts = }")
print(f"{lecture_starts.year = }, {lecture_starts.month = }")
print(f"Ukedag: {lecture_starts.weekday()} (0=Mandag, 6=Søndag)")
print()

# Et timedelta-objekt representerer en gitt varighet
lecture_duration = timedelta(hours=1, minutes=45)
print(f"{lecture_duration = }")

# Forholdstall mellom varigheter kan brukes for å telle hvor mange
# dager/timer/sekunder/millisekunder det er i en varighet.
minute = timedelta(minutes=1)
print(f"{lecture_duration / minute = }") # Antall minutter totalt
hour = timedelta(hours=1)
print(f"{lecture_duration / hour = }")  # Antall timer
print()

# Tidspunkt + varigheter gir et nytt tidspunkt
lecture_ends = lecture_starts + lecture_duration
print(f"{lecture_ends = }")
print()

# Tidspunktet akkurat nå
now = datetime.now()
print(f"{now = }")

# Tidspunkt minus tidspunkt gir varighet
time_since_lecture_started = now - lecture_starts
print(f"{time_since_lecture_started = }")
```

Et datetime-objekt kan være bevisst eller ubevisst på hvilken tidssone tidspunktet tilhører. I eksempel over var var vi *ubevisst*. I eksempelet under er vi *bevisst* på hvilken tidssone tidspunktene vi opererer med tilhører; vi må derfor angi hvilken tidssone tidspunktet tilhører når vi oppretter datetime-objekter.

↑ Innhold

```python
from datetime import datetime, timedelta, timezone

# Tidspunkt i UTC (Universal Coordinated Time)
# UTC er den eneste tidssonen som er innbygget i Python fra før.
now_utc = datetime.now(timezone.utc)

# CEST (Central European Summer Time) er UTC+2
timezone_cest = timezone(timedelta(hours=2), name='CEST')
now_cest = datetime.now(timezone_cest)

print(f'{now_utc = }')
print(f'{now_cest = }')

# Forskjellen mellom to samtidige tidspunkt er 0.
difference = now_cest - now_utc
print(f'{difference = }')

# For å opprette bevisste datetime-objekter, må vi angi tzinfo-parameteren
lecture_time = datetime(2024, 4, 5, 12, 15, 00, tzinfo=timezone_cest)
print(f'{lecture_time = }')
```

Konvertering mellom tidssoner

↑ Innhold

```python
from datetime import datetime, timedelta, timezone

# Ubevisst tidspunkt
lecture_time_unaware = datetime(2024, 3, 22, 12, 15, 00)
print(f'{str(lecture_time_unaware) = }')

# Konvertere fra ubevisst datetime til bevisst datetime, med bruk av
# den lokale tidssonen som var gjeldende den aktuelle datoen.
lecture_time_local = lecture_time_unaware.astimezone()
print(f'{str(lecture_time_local) = }')

# Noen tidssoner
tz_utc = timezone.utc # Universal Coordinated Time
tz_cet = timezone(timedelta(hours=1), name='CET') # Central European Time
tz_est = timezone(timedelta(hours=-5), name='EST') # Eastern Standard Time
tz_edt = timezone(timedelta(hours=-4), name='EDT') # Eastern Daylight Time
tz_pst = timezone(timedelta(hours=-8), name='PST') # Pacific Standard Time

# Konvertere til en annen tidssone kan gjøres på to måter:
#
#  new_datetime = old_datetime.replace(tzinfo=new_timezone)
#       endrer tidssonen uten å endre klokkeslettet/dato. For eksempel hvis
#       klokkeslettet var 12:15 i CET, vil det nye tidspunktet være 12:15 i
#       UTC.
#
#  new_datetime = old_datetime.astimezone(new_timezone)
#       endrer tidssonen og klokkeslett/dato slik at det nye klokkeslettet
#      tilsvarer samme punkt i tid, bare oppgitt i en annen tidssone. For
#      eksempel hvis klokkeslettet var 12:15 i CET, vil det nye tidspunktet
#      være 10:15 i UTC.

lecture_time_replace_est = lecture_time_local.replace(tzinfo=tz_est)
print(f'{str(lecture_time_replace_est) = }')

lecture_time_astimezone_est = lecture_time_local.astimezone(tz_est)
print(f'{str(lecture_time_astimezone_est) = }')

# Dersom old_datetime er ubevvisst, vil både replace og astimezone fungere
# som replace, og vil da endre tidssonen uten å røre klokkeslett og dato.
```

CSV

https://docs.python.org/3/library/csv.html

De enkleste CSV-filene er det lett å håndtere med bruk av `.split` og `.join`, slik vi viser i notatene om filer. Men for noen CSV-filer kan det bli komplisert: for eksempel når innholdet i en celle selv inneholder komma. Da er det bedre å bruke `csv`-modulen, som løser slike problemer for oss.

Når man tolker en CSV-fil er det noen parametre som er viktige å kjenne til:

↑ Innhold

- `delimiter` : Skilletegnet mellom cellene. Standard er komma (`,`), men semikolon (`;`) og tabulator (`\t`) er også vanlige.
- `quotechar` : Tegnet som brukes for å omslutte celler som inneholder skilletegnet. Standard er anførselstegn (`"`), men noen ganger brukes apostrof (`'`) eller andre tegn.
- `quoting` : Hvordan omsluttningstegnet (quotechar) skal brukes.
  - `csv.QUOTE_MINIMAL` (standard - omslutt celler kun hvis nødvendig),
  - `csv.QUOTE_ALL` (alle celler omsluttes), eller
  - `csv.QUOTE_NONNUMERIC` (alle celler som ikke er tall omsluttes).

CSV-modulen tillater at vi angir våre egne verdier for disse parametrene dersom de avviker fra standardverdiene.

Konvertere mellom CSV-fil og 2D-liste

↑ Innhold

↑ Innhold

```python
import csv
# Du kan kopiere funksjonene for å lese/skrive csv-filer og bruke dem
# som du ønsker uten å sitere.

def read_csv_file(path, encoding="utf-8", **kwargs):
    r''' Reads a csv file from the provided path, and returns its
    content as a 2D list. The default encoding is utf-8, the default
    column delimitier is comma and the default quote character is the
    double quote character ("), though this can be overridden with
    named parameters "delimiter" and "quotechar".'''
    with open(path, "rt", encoding=encoding, newline='') as f:
        return list(csv.reader(f, **kwargs))

def write_csv_file(path, table_content, encoding='utf-8', **kwargs):
    r""" Given a file path and a 2D list representing the content, this
    method will create a csv file with the contents formatted as csv.
    By defualt the delimiter is a comma and the quote character is
    the double quote, but this can be overridden with named parameters
     "delimiter" and "quotechar". """
    with open(path, "wt", encoding=encoding, newline='') as f:
        writer = csv.writer(f, **kwargs)
        for row in table_content:
            writer.writerow(row)

# Eksempeler på bruk. Først, en 2D-liste med innholdet i tabellen.
org_content = [
    ["Name", "Age"],
    ["Ola", 74],
    ['"Kari"', "73"],
]
print("Original 2D-liste:", org_content)

# Eksempel 1: standard parametre (se resultat i foo.csv)
write_csv_file("foo.csv", org_content)
with open("foo.csv", encoding='utf-8') as f:
    print("write_csv_file, standard parametre:", repr(f.read()))
readback_content = read_csv_file("foo.csv")
print("read_csv_file, standard parametre:", readback_content)

# Eksempel 2: eksempel på bruk av navngitte parametre (se resultat i
bar.csv)
# delimiter="|" endrer skillesymbolet til vertikal strek
# quoting=csv.QUOTE_NONNUMERIC gjør at alt unntatt tall-verdier omsluttes
# av hermetegn; og ved lesing, at tall uten hermetegn konverteres til float.
write_csv_file("bar.csv", org_content, delimiter="|",
                                       quoting=csv.QUOTE_NONNUMERIC)
with open("bar.csv", encoding='utf-8') as f:
    print("write_csv_file, med egne parametre:", repr(f.read()))
readback_content = read_csv_file("bar.csv", delimiter="|",
                                            quoting=csv.QUOTE_NONNUMERIC)
print("read_csv_file, med egne parametre:", readback_content)
```

↑ Innhold

DictReader og DictWriter: data som liste av oppslagsverk

CSV-bibliotektet har innbygd funksjonalitet for å konvertere mellom CSV-filer og lister av oppslagsverk (dict). Da benyttes `csv.DictReader` og `csv.DictWriter` i stedet for `csv.reader` og `csv.writer`.

↑ Innhold

```python
import csv
from pathlib import Path

# Kopier gjerne funksjonene csv_dict_reader og csv_dict_writer herfra

def csv_dict_reader(path, encoding='utf-8', delimiter=',',
                    quotechar='"', quoting=csv.QUOTE_MINIMAL, **kwargs):
    '''Read a CSV file and return the headers as a list and the data as
    a list of dictionaries. Typical usage example:

        >>> headers, data = csv_dict_reader('foo.csv', delimiter=';')
        >>> headers
        ['Name', 'Age']
        >>> data
        [{'Name': 'Ola', 'Age': '74'}, {'Name': 'Kari', 'Age': '73'}]

    Args:
        path (str): The path to the CSV file.
        encoding (str, optional): The encoding of the CSV file. Default
            is 'utf-8'.
        delimiter (str, optional): The delimiter between cells used in
            the CSV file. Default is ','.
        quotechar (str, optional): The quote character used in the CSV
            file. Default is '"'.
        quoting (int, optional): The quoting style used in the CSV
            file. Default is csv.QUOTE_MINIMAL. Some other useful
            options are csv.QUOTE_ALL and csv.QUOTE_NONNUMERIC. See
            https://docs.python.org/3/library/csv.html#csv.QUOTE_ALL
            for more information.
        **kwargs: Additional keyword args to pass to csv.DictReader.

    Returns:
        headers (list): The headers of the CSV file.
        data (list): The data of the CSV file as a list of dictionaries.
    '''
    with Path(path).open('rt', encoding=encoding, newline='') as f:
        reader = csv.DictReader(f, delimiter=delimiter, quotechar=quotechar,
                                quoting=quoting, **kwargs)
        headers = reader.fieldnames
        data = list(reader)
    return headers, data

def csv_dict_writer(path, headers, data, encoding='utf-8', delimiter=',',
                    quotechar='"', quoting=csv.QUOTE_MINIMAL, **kwargs):
    '''Write a CSV file with the specified headers and data. Typical
    usage example:

        >>> headers = ['Name', 'Age']
        >>> data = [
        ...     {'Name': 'Ola', 'Age': 74},
        ...     {'Name': 'Kari', 'Age': 73},
```

```
        ... ]
        >>> csv_dict_writer('foo.csv', headers, data, delimiter=';')

    Args:
        path (str): The path to the CSV file.
        headers (list): The headers of the CSV file.
        data (list): The data of the CSV file as a list of dictionaries.
        encoding (str, optional): The encoding of the CSV file. Default
            is 'utf-8'.
        delimiter (str, optional): The delimiter between cells used in
            the CSV file. Default is ','.
        quotechar (str, optional): The quote character used in the CSV
            file. Default is '"'.
        quoting (int, optional): The quoting style used in the CSV
            file. Default is csv.QUOTE_MINIMAL. Some other useful
            options are csv.QUOTE_ALL and csv.QUOTE_NONNUMERIC. See
            https://docs.python.org/3/library/csv.html#csv.QUOTE_ALL
            for more information.
        **kwargs: Additional keyword args to pass to csv.DictWriter.

    Returns:
        None
    '''
    with Path(path).open('wt', encoding=encoding, newline='') as f:
        writer = csv.DictWriter(f, fieldnames=headers, delimiter=delimiter,
                                quotechar=quotechar, quoting=quoting)
        writer.writeheader()
        writer.writerows(data)

# Eksempel på bruk:
org_headers = ['Name', 'Age']
org_data = [
    {'Name': 'Ola', 'Age': 74},
    {'Name': 'Kari', 'Age': 73},
]

# Skriv til fil med ulike innstillinger med csv_dict_writer
csv_dict_writer('default.csv', org_headers, org_data)
csv_dict_writer('semicolon.csv', org_headers, org_data, delimiter=';')
csv_dict_writer('quotenonnumeric.csv', org_headers, org_data,
                quotechar="'", quoting=csv.QUOTE_NONNUMERIC)

# Les de produserte filene som rå tekst (plain text)
raw_csv_default = Path('default.csv').read_text(encoding='utf-8')
raw_csv_semicolon = Path('semicolon.csv').read_text(encoding='utf-8')
raw_csv_quotenumeric = Path('quotenonnumeric.csv').read_text(encoding='utf-
8')

# Les de produserte filene tilbake med csv_dict_reader
rb_default_headers, rb_default_data = csv_dict_reader('default.csv')
rb_semicolon_headers, rb_semicolon_data = csv_dict_reader(
    'semicolon.csv', delimiter=';'
```

↑ Innhold

```
)
rb_quotenumeric_headers, rb_quotenumeric_data = csv_dict_reader(
    'quotenonnumeric.csv', quotechar="'", quoting=csv.QUOTE_NONNUMERIC
)

print('Raw file content')
print('default:     ', repr(raw_csv_default))
print('semicolon:   ', repr(raw_csv_semicolon))
print('quotenumeric:', repr(raw_csv_quotenumeric))
# default:      'Name,Age\nOla,74\nKari,73\n'
# semicolon:    'Name;Age\nOla;74\nKari;73\n'
# quotenumeric: "'Name','Age'\n'Ola',74\n'Kari',73\n"

print()
print('Readback headers')
print('default:     ', rb_default_headers)
print('semicolon:   ', rb_semicolon_headers)
print('quotenumeric:', rb_quotenumeric_headers)
# rb_default:      ['Name', 'Age']
# rb_semicolon:    ['Name', 'Age']
# rb_quotenumeric: ['Name', 'Age']

print()
print('Readback data')
print('default:     ', rb_default_data)
print('semicolon:   ', rb_semicolon_data)
print('quotenumeric:', rb_quotenumeric_data)
# default:      [{'Name': 'Ola', 'Age': '74'}, {'Name': 'Kari', 'Age':
'73'}]
# semicolon:    [{'Name': 'Ola', 'Age': '74'}, {'Name': 'Kari', 'Age':
'73'}]
# quotenumeric: [{'Name': 'Ola', 'Age': 74.0}, {'Name': 'Kari', 'Age':
73.0}]
```

DictReader med CSV-formatterte strenger

↑ Innhold

Å benytte `DictReader` med strenger i stedet for filer er dessverre litt knotete. Vi må en liten omvei via `io.StringIO` fra `io`-modulen i standardbiblioteket.

```
import csv
import io

csv_string = '''\
Name;Age
Ola;74
Kari;73
'''

# Leser fra en streng med DictReader
reader = csv.DictReader(io.StringIO(csv_string), delimiter=';')
headers = reader.fieldnames
data = list(reader)

print(headers) # ['Name', 'Age']
print(data)  # [{'Name': 'Ola', 'Age': '74'}, {'Name': 'Kari', 'Age': '73'}]
```

json

https://docs.python.org/3/library/json.html

JSON er et filformat basert på ren tekst som i sin struktur er nesten nøyaktig som et oppslagsverk hvor alle nøklene er strenger. På samme måte som for CSV er det enkelte detaljer som gjør at import og eksport av JSON likevel gjøres best med en egnet modul.

↑ Innhold

```python
import json

# Eksempel på innholdet i en JSON-fil som en streng (såkalt JSON-streng)
sample_json_string = """\
{
  "name": "Kari",
  "is_alive": true,
  "age": 27,
  "address": {
    "street": "Gateveien 1234",
    "city": "En 'by'",
    "postal_code": "5000"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "12345678"
    },
    {
      "type": "office",
      "number": "23456789"
    }
  ]
}
"""

# For å konvertere fra JSON-streng til oppslagsverk
d = json.loads(sample_json_string)
print(type(d)) # dict
print(d["name"], "har telefonnummer", d["phone_numbers"][0]["number"])
print()

# For å konvertere fra oppslagsverk til JSON-streng
d["age"] += 42 # Liten endring først
s1 = json.dumps(d) # Kompakt JSON-streng, bra for nedlasting/datamaskiner
s2 = json.dumps(d, indent=2) # Pen og leselig JSON-streng, bra for mennesker
print(s1)
print(str(d)) # Legg merke til at s1 ligner på str(d). Ser du forskjellene?
print(s2)
```

sys

https://docs.python.org/3/library/sys.html

```python
import sys

# Avslutt python umiddelbart
sys.exit()
print("Vi kommer aldri hit")
```

↑ Innhold

os og shutil

https://docs.python.org/3/library/os.html

https://docs.python.org/3/library/os.path.html

https://docs.python.org/3/library/shutil.html

`os` er en modul med mange avanserte funksjoner, men også et par funksjoner som er greie å ha for å navigere filsystemet lokalt på datamaskinen. `shutil` er en modul som også jobber med filer, gjerne mer enn én om gangen.

For å testekoden under, lim den inn i en fil og kjør filen på din lokale maskin.

↑ Innhold

```python
import os
import shutil
# os er en modul med mange avanserte funksjoner, men også et par funksjoner
# som er greie å ha for å navigere filsystemet lokalt på datamaskinen.
# shutil er en modul som også jobber med filer, gjerne mer enn én om gangen.

# Vis `current working directory` (cwd). Dette er den mappen python
# vil bruke som utgangspunkt for å se etter filstier.
full_folder_path = os.getcwd() # en streng
print("Current working directory er:\n", full_folder_path)

print("Oppretter nå en fil foo.txt med innhold: 'woof\nbark\n'")
with open("foo.txt", "wt", encoding='utf-8') as f:
    f.write("woof\nbark\n")
print("Foo.txt ble opprettet i mappen:\n", os.getcwd())

# Komplett filsti for filen foo.txt vi nettopp opprettet. Dette er bedre enn
# `os.getcwd() + "/foo.txt"` fordi det virker både for Windows og Mac.
full_file_path = os.path.join(os.getcwd(), "foo.txt")
print("Filsti (path) for foo.txt:", full_file_path)
print("Sjekk at filen foo.txt ble opprettet i denne mappen")
input("Trykk enter for å fortsette")
print()

print("Oppretter nå mappen bar inne i en mappen egg, inne i en mappen temp")
os.makedirs(os.path.join("temp", "egg", "bar"))
print("Sjekk at mappene temp/egg/bar ble opprettet i ", os.getcwd())
input("Trykk enter for å fortsette")
print()

def print_contents_of_folder(path_to_folder):
    print("Ser på alle ting i mappen", path_to_folder)
    # os.listdir returnerer en liste med strenger
    for subpath in sorted(os.listdir(path_to_folder)):
        # subpath er navnet til fil/mappe som er i mappen folder_path
        full_subpath = os.path.join(path_to_folder, subpath)
        subpath_is_file = os.path.isfile(full_subpath)
        print("file    " if subpath_is_file else "folder  ", end="")
        print(subpath)

print_contents_of_folder(full_folder_path)
input("Trykk enter for å fortsette")
print()

print("Endrer cwd (current working directory) til mappen temp/egg/bar")
os.chdir(os.path.join(full_folder_path, "temp", "egg", "bar"))
print("Nå er os.getcwd() =", os.getcwd())
print("Oppretter filer xi.txt, tau.txt, alpha.txt")
for filename in ["xi.txt", "tau.txt", "alpha.txt"]:
    with open(filename, "wt", encoding='utf-8') as f:
        f.write("hiha")
```

↑ Innhold

```
print_contents_of_folder(os.getcwd())
input("Trykk enter for å fortsette")
print()

print("Endrer cwd ved å gå til mappen på nivået over tre ganger")
for _ in range(3):
    os.chdir(os.path.dirname(os.getcwd()))
    print("cwd er nå", os.getcwd())
input("Trykk enter for å fortsette")
print()

temp_folder_path = os.path.join(os.getcwd(), "temp")
print("Sjekker om stien eksisterer:", temp_folder_path)
print(os.path.exists(temp_folder_path))
print("Går igjennom alt innhold uansett dypbe:", temp_folder_path)
for dirpath, dirnames, filnames in os.walk(temp_folder_path):
    print(dirpath, dirnames, filnames)
input("Trykk enter for å fortsette")
print()

os.remove(full_file_path)
print("Fjernet foo.txt")
input("Trykk enter for å fortsette")
print()

import shutil
shutil.rmtree(temp_folder_path)
print("Fjernet temp-mappen inkludert alt innhold")
print("Ferdig!")
```

pathlib

Path fra pathlib er en moderne måte å navigere stier (til filer og mapper) i filsystemet. Vi presenterer her en walk-through av noen av mulighetene.

↑ Innhold

```
from pathlib import Path

# Hint: kopier og kjør programmet mens du følger med i filsystemet ditt

input('''
>>> # For å navigere filsystemet, kan vi benytte oss av Path
>>> # fra pathlib.
>>>
>>> from pathlib import Path

>>> # Trykk enter for å gå videre.............................
''')


print('''
>>> ######################################
>>> ### CWD (current working directory) ###
>>> ######################################
>>>
>>> # CWD (current working directory) er en mappe på datamaskinen som
>>> # Python-programmet anser som sin «arbeidsmappe». For å få vite
>>> # hvilken mappe dette er, kjører vi
>>>
>>> cwd = Path.cwd()
>>> print(cwd)
''')

cwd = Path.cwd()
print(cwd)

input('''\n>>> # Trykk enter for å gå videre.............................\n''')
print('''
>>> ##################################
>>> ### HOME (user home directory) ###
>>> ##################################
>>>
>>> # Brukeren sin hjemmemappe kalles «home» og er den mappen på
>>> # som er knyttet til brukeren som nå er innlogget på datamaskinen.
>>> # For å få vite hvilken mappe dette er, kjører vi
>>>
>>> home = Path.home()
>>> print(home)
''')

home = Path.home()
print(home)

input('''\n>>> # Trykk enter for å gå videre.............................\n''')
print(r'''
>>> #############
>>> ### Stier ###
```

↑ Innhold

```
>>> #############
>>>
>>> # En sti (engelsk: path) er en angivelse av et filnavn eller
>>> # et mappenavn og hvor den befinner seg. Filen/mappen trenger
>>> # ikke nødvendigvis eksistere på filsystemet for at det er en
>>> # sti. Man kan for eksempel bruke en sti for å opprette en ny
>>> # fil eller en ny mappe der.
>>>
>>> # En sti kan være enten fullt spesifisert, eller den kan være
>>> # oppgitt som en _relativ_ sti. En relativ sti tolkes relativt
>>> # til CWD. I eksempelet under oppretter vi først en relativ sti
>>> # til en fil foo.txt. Deretter oppretter vi filen i filsystemet
>>> # når vi skriver til den.
>>>
>>> foo = Path('foo.txt')
>>> foo.write_text('woof\nbark\n', encoding='utf-8')
''')

foo = Path('foo.txt')
foo.write_text('woof\nbark\n', encoding='utf-8')

input('''\
>>> # Verifiser at filen foo.txt ble opprettet i CWD.

>>> # Trykk enter for å gå videre.............................
''')



print('''\
>>> ##############################################################
>>> ### Se om en sti eksisterer, og om det er fil eller mappe ###
>>> ##############################################################
>>> #
>>> # Gitt en sti, kan det være interessant å vite om den eksisterer
>>> # i filsystemet; og om det i så fall er en sti til en fil eller
>>> # til en mappe.
>>>
>>> print(foo) \
''')
print(foo)

print('>>> print(path.exists())')
print(foo.exists())

print('>>> print(path.is_file())')
print(foo.is_file())

print('>>> print(path.is_dir())')
print(foo.is_dir())

input('''\n>>> # Trykk enter for å gå videre.............................\n''')
```

↑ Innhold

```
print('''
>>> ###################################
>>> ### Løkke gjennom alt i en mappe ###
>>> ###################################
>>> #
>>> # Dersom stien er til en mappe som eksisterer, kan vi gå igjennom
>>> # innholdet i mappen med en løkke og .iterdir(). Merk at
>>> # iterandene (løkkevariabelen) også vil være stier.
>>>
>>> my_dir = Path.cwd()
>>> for subpath in my_dir.iterdir():
...     print(subpath)
''')

my_dir = Path.cwd()
for subpath in my_dir.iterdir():
    print(subpath)

input('''\n>>> # Trykk enter for å gå videre.............................\n''')
print('''
>>> ###################
>>> ### Undermapper ###
>>> ###################
>>> #
>>> # Man kan lime sammen stier med / for å lage sti til undermappe.
>>>
>>> my_file = Path.home() / 'mysub' / 'nested' / 'file.txt'
>>> print(my_file)
''')

my_file = Path.home() / 'mysub' / 'nested' / 'file.txt'
print(my_file)


input('''\n>>> # Trykk enter for å gå videre.............................\n''')
print('''
>>> ###############################
>>> ### Finn siste ledd i stien ###
>>> ###############################
>>> #
>>> # Det siste leddet i stien er selve filnavnet, eller navnet på
>>> # selve mappen som stien viser til.
>>>
>>> filename_only = my_file.name
>>> print(my_file)
''')

filename_only = my_file.name
print(filename_only)

input('''\n>>> # Trykk enter for å gå videre.............................\n''')
```

↑ Innhold

```
print('''
>>> ########################
>>> ### Gå «opp» en mappe ###
>>> ########################
>>> #
>>> # Hvis man har en sti, kan vi få vite i hvilken mappe stien
>>> # ligger i. Dette er som å fjerne det siste leddet i stien.
>>>
>>> my_parent = my_file.parent
>>> print(my_parent)
''')

my_parent = my_file.parent
print(my_parent)


input('''\n>>> # Trykk enter for å gå videre.............................\n''')
print('''
>>> ########################
>>> ### Opprett en mappe ###
>>> ########################
>>> #
>>> # Vi oppretter en mappe for en gitt sti.
>>>
>>> tmp_folder = Path('tmp')
>>> tmp_folder.mkdir()
''')

tmp_folder = Path('tmp')
tmp_folder.mkdir()


input('''\
>>> # Verifiser at det er opprettet en mappe 'tmp' i CWD

>>> # Trykk enter for å gå videre.............................
''')
print('''
>>> ####################################
>>> ### Flytt/gi nytt navn til en fil ###
>>> ####################################
>>> #
>>> # Vi flytter filen foo.txt inn i mappen tmp og beholder samme
>>> # filnavn. For å endre filnavn, kunne vi brukt noe annet enn
>>> # foo.name i den tredje linjen under.
>>>
>>> foo = Path('foo.txt')
>>> move_to_folder = Path('tmp')
>>> new_foo_path = move_to_folder / foo.name
>>> foo.rename(new_foo_path)
''')
```

↑ Innhold

```
foo = Path('foo.txt')
move_to_folder = Path('tmp')
new_foo_path = move_to_folder / foo.name
foo.rename(new_foo_path)


input('''\
>>> # Verifiser at filen foo.txt ble flyttet til mappen tmp

>>> # Trykk enter for å gå videre.............................
''')
print('''
>>> ##############################
>>> ### Slette filer og mapper ###
>>> ##############################
>>> #
>>> # For å slette en fil, bruker vi unlink. For å sletten en
>>> # mappe, bruker vi rmdir (mappen må være tom først)
>>>
>>> new_foo_path.unlink()
>>>
>>> folder = Path('tmp')
>>> folder.rmdir()
''')

new_foo_path.unlink()

folder = Path('tmp')
folder.rmdir()

input('''>>> # Verifiser at mappen tmp og filen foo.txt er borte.''')
```

Eksterne pakker

↑ Innhold

- Installasjon med skript
- Installasjon med pip

Eksempler på eksterne pakker

- Installasjon med skript
- Installasjon med pip
- Requests: last ned ting fra internett
- Matplotlib: visualisering av data
  - Enkle plot
- Scatterplot
- Andre vanlige pakker
  - Numpy
  - Pandas

- Andre vanlige pakker

Installasjon med skript

Installasjon av eksterne moduler trenger man bare gjøre én gang for hver Python-installasjon. Under finner du en kodesnutt du kan forsøke å kjøre som prøver å installere noen moduler.

```
import sys
from subprocess import run

package_name = 'requests'  # <--- navnet på pakken du vil installere

run(f'{sys.executable} -m ensurepip'.split())
run(f'{sys.executable} -m pip install --upgrade pip'.split())
run(f'{sys.executable} -m pip install {package_name}'.split())
```

Installasjon med pip

For å installere eksterne pakker og moduler, bruker vi et program som heter pip (https://pip.pypa.io/en/stable/getting-started/). Dette er et program som ble installert sammen med python. Dersom du har flere versjoner av Python installert på din datamaskin (for eksempel fordi en gammel versjon av python var installert fra før), har du også flere versjoner av pip installert på maskinen din. Når du installerer programmer med pip er det viktig at du bruker den versjonen av pip som matcher den versjonen av python du bruker.

For å installere noe med pip, bruker vi Terminalen og skriver kommandoen:

```
<python-sti> -m pip install <pakkenavn>
```

↑ Innhold

(det er også mulig å skrive `<pip-sti> install <pakkenavn>`, men da har vi mindre kontroll på at biblioteket installeres i riktig Python-installasjon). Et par punkter å passe på:

- `<pakkenavn>` skal erstattes med navnet på pakken som skal installeres.
- `<python-sti>` skal erstattes med en sti til den Python-versjonen du skal installere pakken for. Dette kan være så enkelt som å skrive `python`, `py` eller `python3`, eller det kan være du må skrive ned fullstendig sti til den Python-fortolkeren du bruker. For å sjekke hvilken sti dette er, kan du kopiere dette programmet inn i en Python-fil og kjøre filen:

```python
import sys
print(f"{sys.executable}")
```

- Dersom filstien inneholder mellomrom, kan det være du må skrive den inn med hermetegn rundt.
- Med Windows PowerShell må ta med `&` helt i begynnelsen av kommandoen, altså `& <python-sti> -m pip install <pakkenavn>`.
- I noen tilfeller kan det kreves administrator-rettigheter for å installere. På Mac/Linux kan man da legge til `sudo` helt i begynnelsen av kommandoen og så skrive inn passordet for datamaskinen. På Windows kan PowerShell åpnes som administrator ved å høyreklikke på PowerShell i startmenyen og velge «kjør som administrator».

Nå viser vi noen eksempler på vanlige eksterne biblioteker og hvordan de brukes.

Requests: last ned ting fra internett

https://requests.readthedocs.io/

Requests-pakken kan benyttes dersom man ønsker å laste ned informasjon fra internett (via http/https) som skal brukes i et python-program.

```python
import requests
# Pakke for å laste ned data fra internett

url = "https://tinyurl.com/foo-txt" # Nettsiden som skal lastes ned
headers = {
    # Mange nettsider krever at «User-Agent» har en verdi for å gi respons.
    'User-Agent': 'no.uib.ii.inf100 abcd1234', # Noe som sier «hva du er»
}

webpage = requests.get(url, headers=headers)
print(webpage.text) # Innholdet på nettsiden som en streng
```

Matplotlib: visualisering av data

https://matplotlib.org/

↑ Innhold

Matplotlib er et stort bibliotek med utallige muligheter for å visualisere data på ulike måter. Under viser vi bare et helt enkelt eksempel; se igjennom dokumentasjonen på matplotlib sin hjemmeside for å lære om flere muligheter.

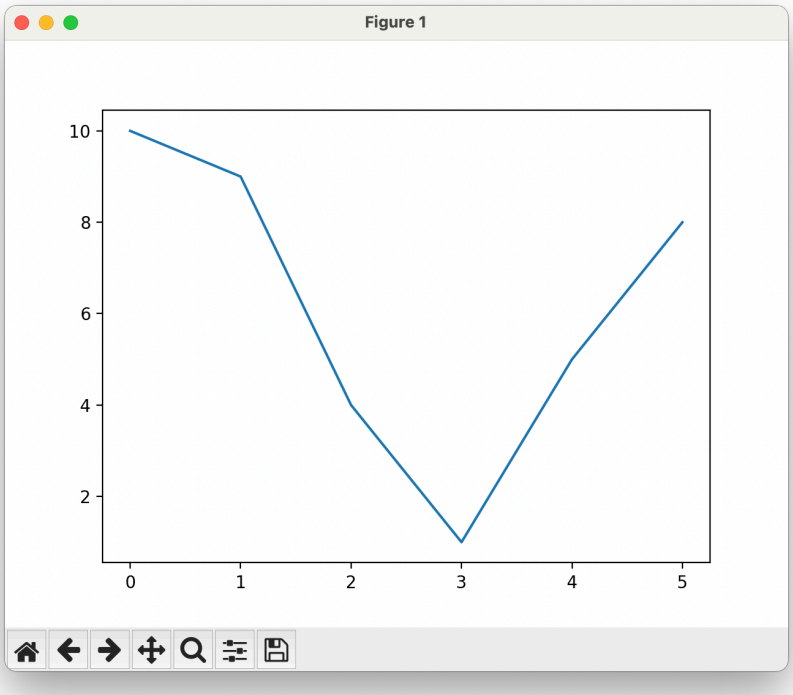
- Enkle plot
- Scatterplot

Enkle plot

```
from matplotlib import pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
y_values = [10, 9, 4, 1, 5, 8]

plt.plot(x_values, y_values)
plt.show()
```



↑ Innhold

```python
from matplotlib import pyplot as plt

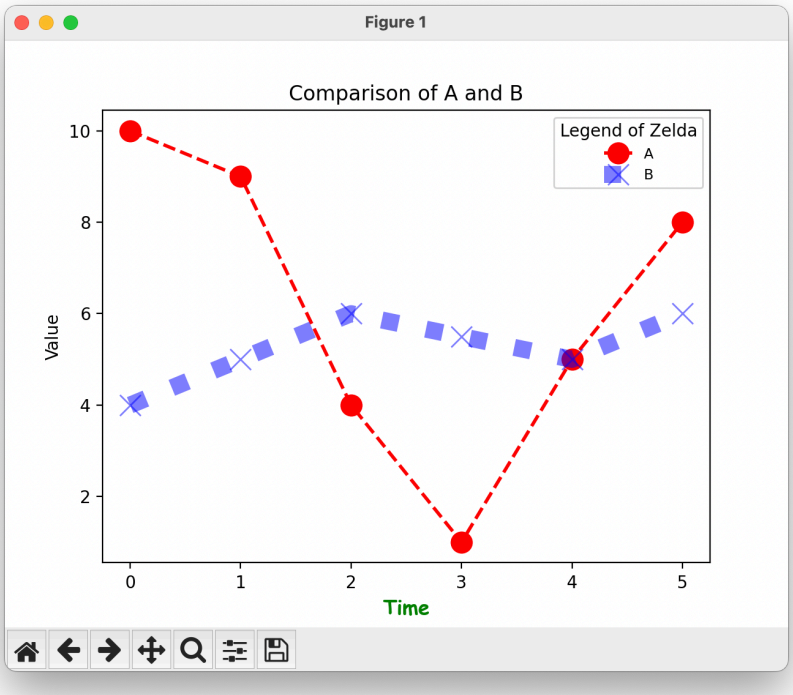
x_values_a = [0, 1, 2, 3, 4, 5]
y_values_a = [10, 9, 4, 1, 5, 8]
plt.plot(
    x_values_a, y_values_a,
    color='red',
    marker='o',
    linestyle='dashed',
    linewidth=2,
    markersize=12,
    label='A',
)

x_values_b = [0, 1, 2, 3, 4, 5]
y_values_b = [4, 5, 6, 5.5, 5, 6]
plt.plot(
    x_values_b, y_values_b,
    color='blue',
    marker='x',
    linestyle='dotted',
    linewidth=10,
    markersize=12,
    alpha=0.5,
    label='B',
)

plt.xlabel(
    'Time',
    fontsize='large',
    fontweight='bold',
    fontname='Comic Sans MS',
    color='green',
)
plt.ylabel('Value')
plt.title('Comparison of A and B')
plt.legend(
    title='Legend of Zelda',
    loc='upper right',
    fontsize='small',
)

plt.show()
```

↑ Innhold



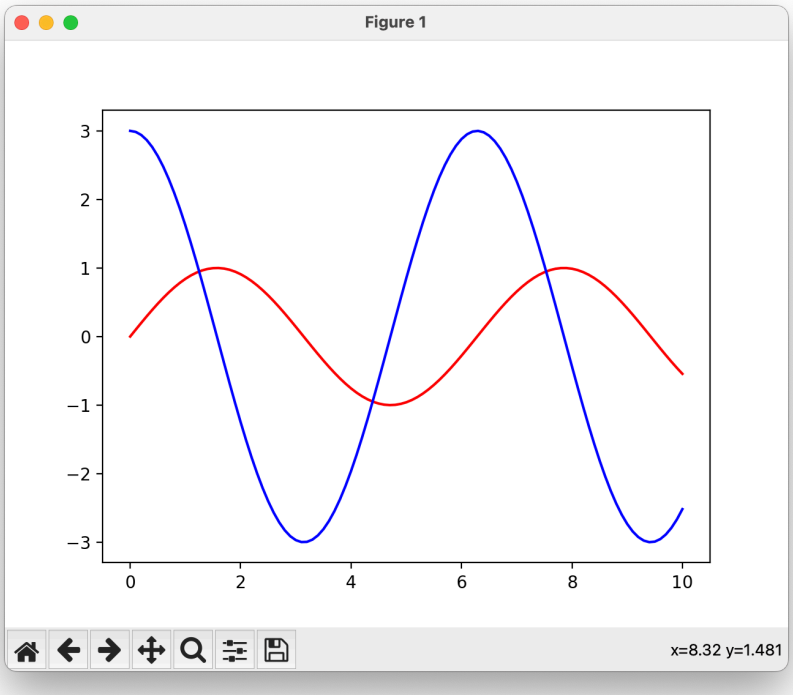

```python
from matplotlib import pyplot as plt
from math import sin, cos

# Eksempeldata som skal visualiseres
# liste med x-verdier
xs = [n / 10 for n in range(101)]
# 2 ulike lister med y-verdier
ys_1 = [sin(x) for x in xs]
ys_2 = [3 * cos(x) for x in xs]

# Opprette et plot
plt.plot(xs, ys_1, "r")
plt.plot(xs, ys_2, "b")

# savefig lagrer filene
plt.savefig("my_plot.svg")  # SVG vektorgrafikk (bra format for figurer!)
plt.savefig("my_plot.pdf")  # PDF
plt.savefig("my_plot.png")  # PNG er egentlig bedre egnet for foto enn for
                            # figurer, men har høy kryss-kompatibilitet

# interaktivt vindu
plt.show()
```



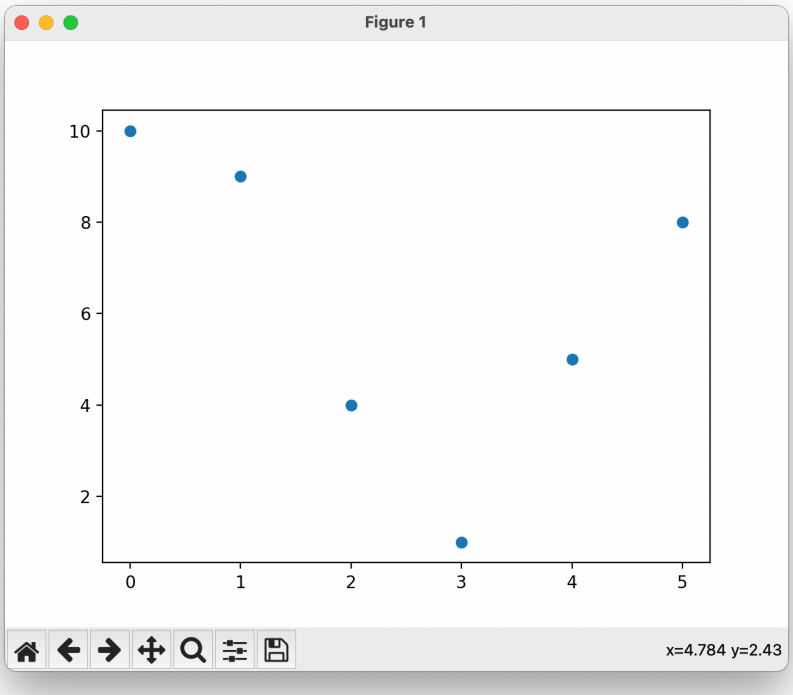
Scatterplot

```
from matplotlib import pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
y_values = [10, 9, 4, 1, 5, 8]

plt.scatter(x_values, y_values)
plt.show()
```

↑ Innhold



Kombinerte scatterplott og linjediagram, med farge og størrelse og diverse andre innstillinger:

```
from matplotlib import pyplot as plt

x_values_a = [0, 1, 2, 3, 4, 5]
y_values_a = [10, 9, 4, 1, 5, 8]
sizes_a = [50, 2000, 3000, 4000, 5000, 50000]
colors_a = ['red', 'blue', 'green', 'yellow', 'purple', 'orange']
alpha_a = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]

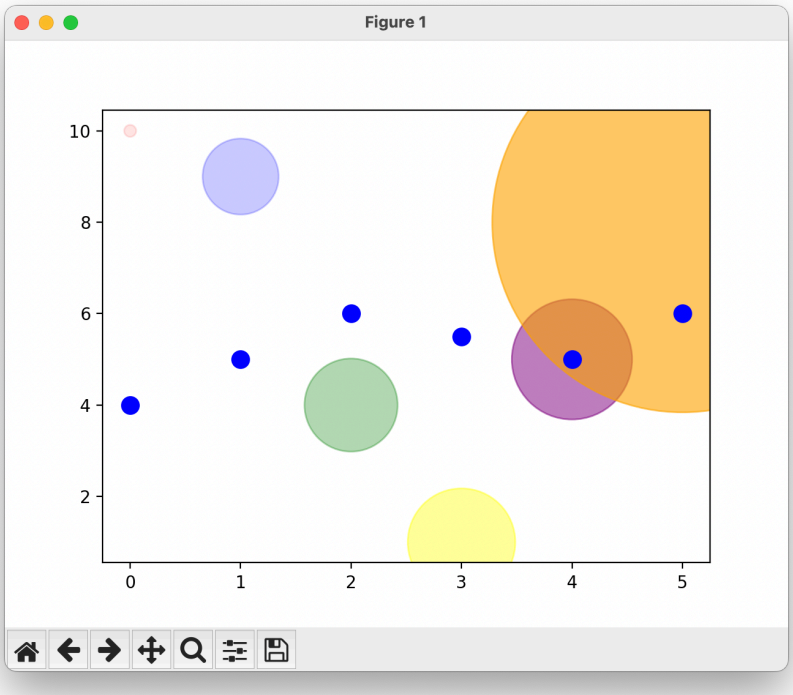
plt.scatter(
    x_values_a, y_values_a,
    s=sizes_a,
    c=colors_a,
    alpha=alpha_a,
)

x_values_b = [0, 1, 2, 3, 4, 5]
y_values_b = [4, 5, 6, 5.5, 5, 6]

plt.plot(
    x_values_b, y_values_b,
    color='blue',
    marker='o', # ligner også på et scatterplot
    linestyle='none', # ligner også på et scatterplotx
    markersize=10,
)

plt.show()
```

↑ Innhold



Andre vanlige pakker

Numpy

https://numpy.org/

Numpy er sammen med matplotlib et av de mest bruke eksterne bibliotekene til databehandling. En av grunnene til numpy sin store popluaritet, er at den kan behandle store datamengder svært raskt. Dette gjøres i bunn og grunn ved å *omgå* noen av Python sine mekanismer for å beskytte utviklere mot seg selv; men samtidig er numpy designet for å gjøre det enkelt å bedrive lineær algebra, hvor datatypene ofte er vektorer og matriser.

Pandas

https://pandas.pydata.org/

Pandas er et slags avansert «excel» for Python, som bygger på numpy og som er integrert med matplotlib. Med dette biblioteket kan man håndtere regneark og andre former for databaser (alt fra csv til json, sql, xlm osv), og utføre en rekke analyser av dissen dataene.