Kursnotater INF100 høst 2023

- Installasjon og Hello World
- Kom i gang
- Variabler og uttrykk
- Grafikk
- Typer
- Operatorer
- Betingelser
- Løkker
- Strenger
- Funksjoner
- Feil og debugging
- Lister
- Flerdimensjonelle lister
- Grafiske brukergrensesnitt
- Filer og CSV
- Unicode
- Håndtere krasj
- Oppslagsverk
- Mengder
- Moduler
- Standardbiblioteket
- Eksterne pakker
- Matplotlib

Installasjon og Hello World

I dette kurset bruker vi Python 3.11 eller nyere som programmeringsspråk og Visual Studio Code som editor.

Installasjon av Python

Video (Windows) Video (Mac)

- Last ned og installer Python 3.11 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

Video

- 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.

Kom i gang

Hvis du ikke har gjort det enda, må du installere Python 3.11 og Visual Studio Code før du fortsetter.

- Hello World
- Tre måter å kjøre Python
- Kommentarer
- Utskrift til terminalen
- Python som enkel kalkulator
- 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 skjermen. I Python ser programmet slik ut:

```
print("Hello World")
```

Kopier Se steg Kjør

Tre måter å kjøre Python

Video

- **Fil/script:** Python vil gå igjennom en fil med python-kode og utføre én og én linje. Skriv kommandoen `python [filnavn]` (eller `python3 [filnavn]`) i terminalen eller klikk på *Run* -knappen i Visual Studio Code.

- **REPL:** En interaktiv modus, *Read-Evaluate-Print-Loop*, er praktisk om man ønsker å utføre en kort kommando eller utforske hvordan noe fungerer. Skriv kommandoen `python` (eller `python3`) i et terminal-vindu, eller gå til *View -> Command Pallette* i Visual Studio Code og skriv *Python: Start REPL*. For å avslutte, trykk `ctrl-d` .

- **Brython:** På denne nettsiden vil mange av kode-eksemplene være mulig å kjøre direkte i nettleseren, slik som *Hello World* -programmet over. Da benytter vi en variant av Python som heter Brython. Denne metoden har noen begrensninger: for eksempel har vi gitt den en timeout på noen få sekunder, og vi kan ikke lese filer som ligger på datamaskinen.

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.

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

print("Her er # i en streng.") # Hashtag mellom "hermetegn" telles ikke
```

Kopier | Se steg | Kjør

Utskrift til terminalen

Print-funksjonen skriver ut verdier til terminalen.

```
# Print-funksjonen kan skrive ut forskellige typer verdier
print("Foo")  # streng/tekst
print(42)     # nummer
print(True)   # boolske verdier

# Print-funksjonen kan skrive ut flere ting på én gang
print("Foo", "Bar")

# Bruk end="" for å avslutte med noe annet enn et linjeskift
print("Foo", end="---*---")
print("Bar")
```

Kopier | Se steg | Kjør

Python som enkel kalkulator

Man kan bruke Python som en enkel kalkulator.

```
print(2 + 2)       # Addisjon                               --> 4
print(5 - 3)       # Subtraksjon                            --> 2
print(2 * 3)       # Multiplikasjon                         --> 6
print(5 / 2)       # Divisjon                               --> 2.5
print(5 // 2)      # Heltallsdivisjon (runder alltid ned)   --> 2
```

```
print(5 % 2)        # Modulo (rest etter heltallsdivisjon)    --> 1
print(3 ** 2)       # Potens                                  --> 9
print(max(2, 3))    # Største verdi                           --> 3
print(min(2, 3))    # Minste verdi                            --> 2
print(abs(-3))      # Absoluttverdi                           --> 3

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

# Man kan bruke parenteser for å styre rekkefølgen på operasjoner
print((2 + 3) * 4)            # 20
```

Kopier Se steg Kjør

Strenger

Video

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

```
print("Strenger skrives mellom hermetegn.")
print('Man kan også bruke apostrof.')
print(Hvis du ikke har hermetegn er det ikke en streng, og det krasjer)
```

Kopier Se steg Kjør

```
# En streng kan lagres i en variabel.
foo = "bar"
print("foo") # Skriver ut "foo"
print(foo) # Skriver ut "bar"
```

Kopier Se steg Kjør

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

# En streng kan inneholde hermetegn (")
print("Her \"er\" et hermetegn")
print('Her "er" et hermetegn') # bedre: bruk apostrofer

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

Kopier Se steg Kjør

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)
```

Kopier Se steg Kjør

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

# Hvis du vil skrive ut noe uten å avslutte med linjeskift, bruk end=""
print("super", end="")
print("duper")
```

Kopier Se steg Kjør

Operasjoner på strenger

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

a = "Hei"
b = "der"
print(a + " " + b) # Hei der
```

Kopier Se steg Kjør

```
# Strenger repeteres flere ganger med gangesymbol (*) og et tall
print("*" * 3) # ***

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

Kopier Se steg Kjør

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

```
# Med en f-streng kan vi inkludere variabler i strengen
number_of_students = 800
course_id = "INF100"
print(f"Vi ønsker {number_of_students} studenter velkommen til {course_id}")

# Uten f'en får vi en logisk feil
print("Vi ønsker {number_of_students} studenter velkommen til {course_id}")
print()

# f-strenger kan også brukes for å lage verdier som lagres som en variabel
message = "Jeg lengter hjem"
log_message = f"Mottatt melding: {message}"
print(log_message) # "Mottatt melding: Jeg lengter hjem"

# f-strenger er ikke magiske
message = "Jeg er fornøyd"
print(log_message) # fremdeles "Mottatt melding: Jeg lengter hjem"
```

Kopier Se steg Kjør

Input fra terminalen

Video

Bruk av `input` -funksjonen for å spørre brukeren om verdier gir mest mening når man kjører en Python-fil i en terminal.

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

Kopier Se steg

```
# Når input er et tall
print("Skriv et tall")
a = input()

print("Skriv et tall til")
b = input()

# Hva tror du skrives ut her?
print("Summen er ", a + b)

a = int(a)
```

```
b = int(b)

# og hva tror du skrives ut her?
print("Summen er ", a + b)
```

Kopier Se steg

Syntaksfeil, krasj og logiske feil

Video

```
print("oj') # Syntaksfeil! (apostrof matcher ikke hermetegn rundt strengen)

# Gir output:
#   SyntaxError: EOL while scanning string literal
```

Kopier Se steg Kjør

```
print("42" + 3) # Krasj! (å legge sammen en streng og et tall er umulig)

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

Kopier Se steg Kjør

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

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

Kopier Se steg Kjør

```
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
```

Kopier Se steg Kjør

Universitetet i Bergen Om siden.

Variabler og uttrykk

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

Variabler

Video

En *variabel* er en referanse til en verdi.

```python
# Symbolet '=' betyr her at vi tilordner verdien 5 til variabelen x
x = 5
print(x)      # 5
print(2 * x)  # 10
print(x)      # fremdeles 5
```

Kopier Se steg Kjør

Til forskjell fra matematikk, kan en variabel i Python endre verdi.

```python
y = 10      # Variabelen y refererer nå til verdien 10
print(y)    # 10

y = True    # Variabelen y refererer nå til verdien True
y = "Hei"   # Variabelen y refererer nå til verdien "Hei"
y = 11      # Variabelen y refererer nå til verdien 11
y = y + 1   # Variabelen y refererer nå til verdien 12
print(y)    # 12
print(y)    # fremdeles 12
```

Kopier Se steg Kjør

Legg merke til at uttrykket `y = y + 1` er en fullstendig meningsfull ting å skrive i programmering, mens det ville vært en selvmotsigelse å skrive det samme i matematikk. I

programmering skal utsagnet tolkes slik: «la y fra nå av referere til en verdi som er én høyere enn hva den refererte til tidligere».

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

```
antall_studenter = 800
antall_gruppeledere = 42
kurskode = "INF100"
99problemer = True # Krasjer, variabler kan ikke begynne med tall
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

Uttrykk og setninger

Video

Et *uttrykk* består av én eller flere verdier, variabler, operasjoner og funksjoner som alltid vil evaluere til en enkelt verdi. Her er noen eksempler på uttrykk (vi antar her at `x` og `y` er variabler som allerede er definert/er tilordnet en verdi).

- `42`
- `"Hei verden"` (alle enkeltstående verdier er uttrykk)
- `x`
- `y` (alle enkeltstående variabler er også uttrykk)
- `2 + 2` (uttrykk kan inneholde operasjoner)
- `True or False`
- `2 < y and x > 0`
- `2 + abs(x)`
- `min(max(0, x), 100)`
- `x**(y - 1) + 2 > min(10, x) or y == 0`

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

- `x = 2 + 4` er en setning, mens `2 + 4` er et uttrykk.
- `print("Hallo")` er en setning, mens `"Hallo"` er et uttrykk.

Variabler referer til verdier, ikke til uttrykk

Video

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
```

Kopier Se steg Kjør

Gode variabelnavn

Følg disse prinsippene når du velger variabelnavn:

1. Variabelnavnet skal være selvforklarende og beskrive sin egen rolle.

```
# Dårlig
s = "L"
```

```
# Bra
shirt_size = "L"
```

2. Benytt kun små bokstaver. Selv om det teknisk sett er mulig å bruke store bokstaver, unngår vi det. Store bokstaver holder vi unna og bruker bare til spesielle formål, f. eks. konstanter (se under).

```
# Dårlig
Name = "Siri"
```

```
# Bra
name = "Siri"
```

3. Benytt `_` for å representere mellomrom, såkalt `snake_case`. Merk at du noen steder vil se `lowerCamelCase` som er ansett for å være god stil i en del andre programmeringsspråk. Vi godtar begge deler, men vær konsekvent.

```
# Tja
numberOfPeople = 42
```

```
# Offisiell python-stil
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
```

Du kan også se støte på `UpperCamelCase` noen steder. Det er god stil å benytte denne formen når man skal navngi såkalte «klasser»; men dette er et begrepet som er utenfor pensum i INF100, og du bør derfor ikke benytte denne formen når du navngir noe i dette kurset.

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

Grafikk

- 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.

Installasjon med script

Installasjon med terminal

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

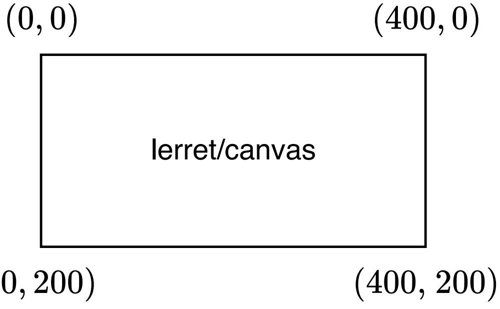
```
from uib_inf100_graphics.simple import canvas, display

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

display(canvas)
```

Kopier

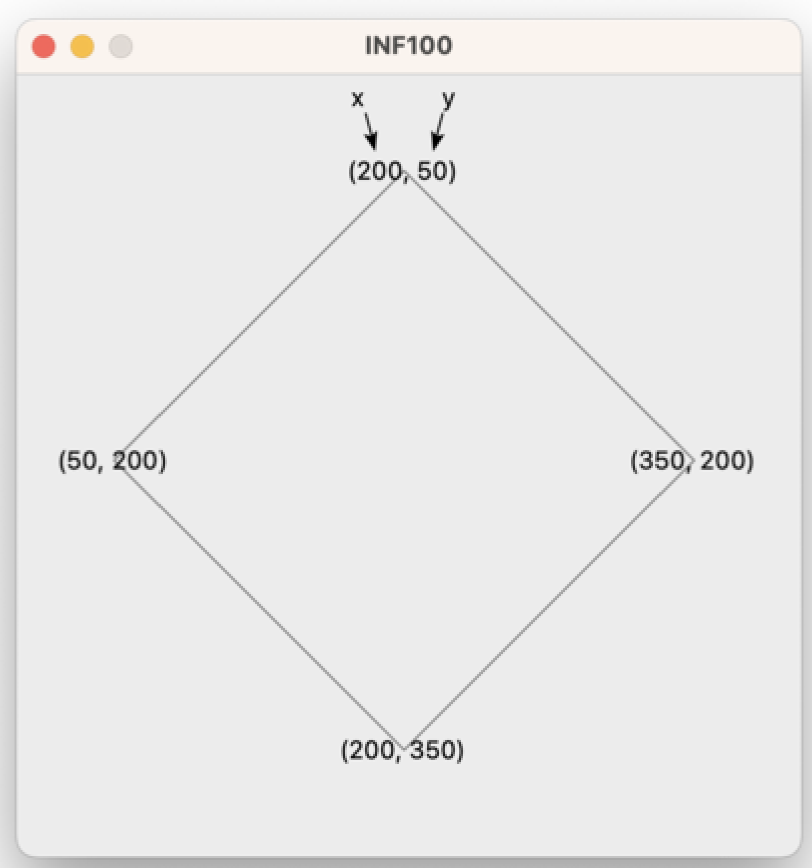
Når du kjører filen skal du se et vindu som ser omtrent slik ut:

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. 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

---

create_rectangle

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

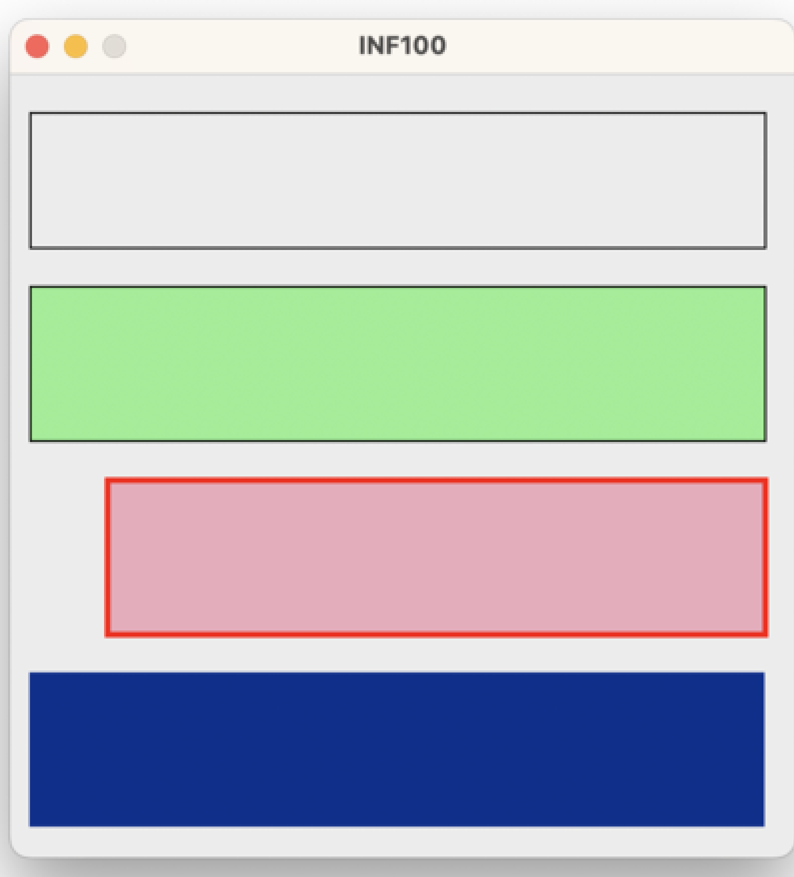
- De første to parametrene x1 og y1 er koordinatene til ett av rektangelets hjørner, mens de neste to parametrene x2 og y2 er koordinatene til det motsatte hjørnet. Konvensjon tilsier at (x1, y1) er hjørnet til venstre øverst, mens (x2, y2) er hjørnet til høyre nederst.

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.

```
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

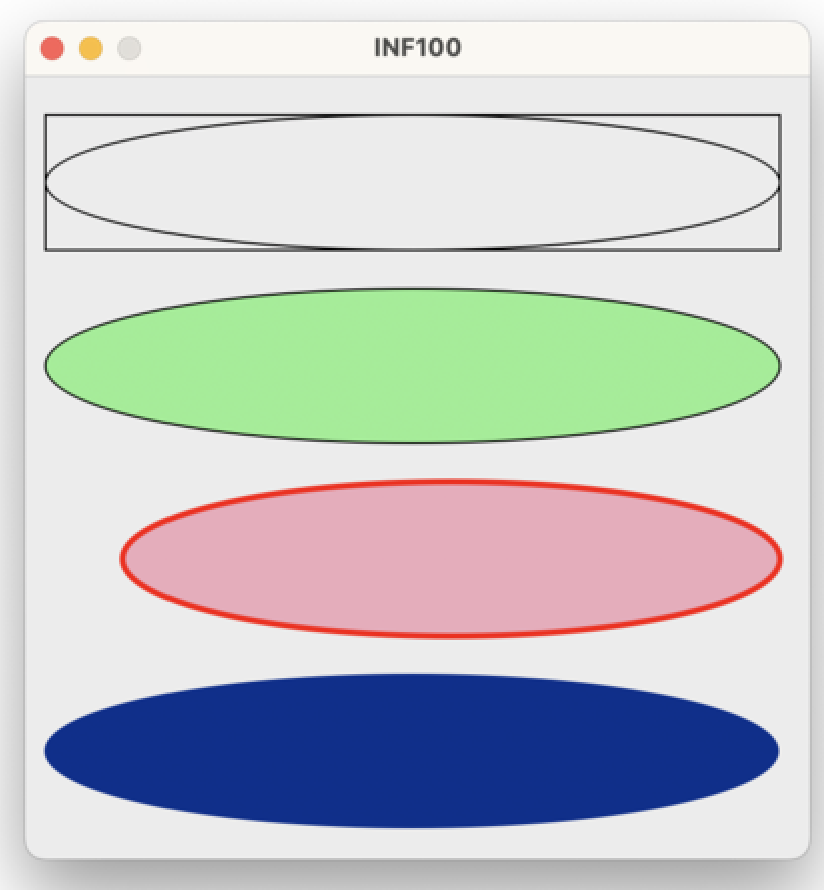
Påkrevde parametre *(x1, y1, x2, y2)*.

- 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.

```
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

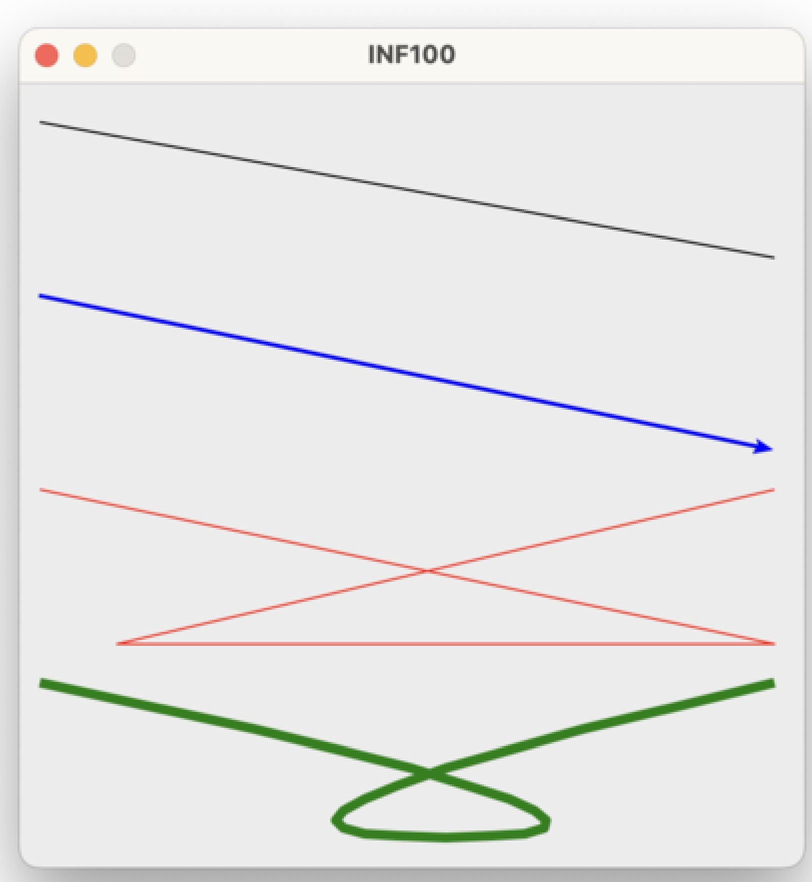
Påkrevde parametre *(x1, y1, x2, y2, ...)*.

- 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.

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)
```



create_polygon

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

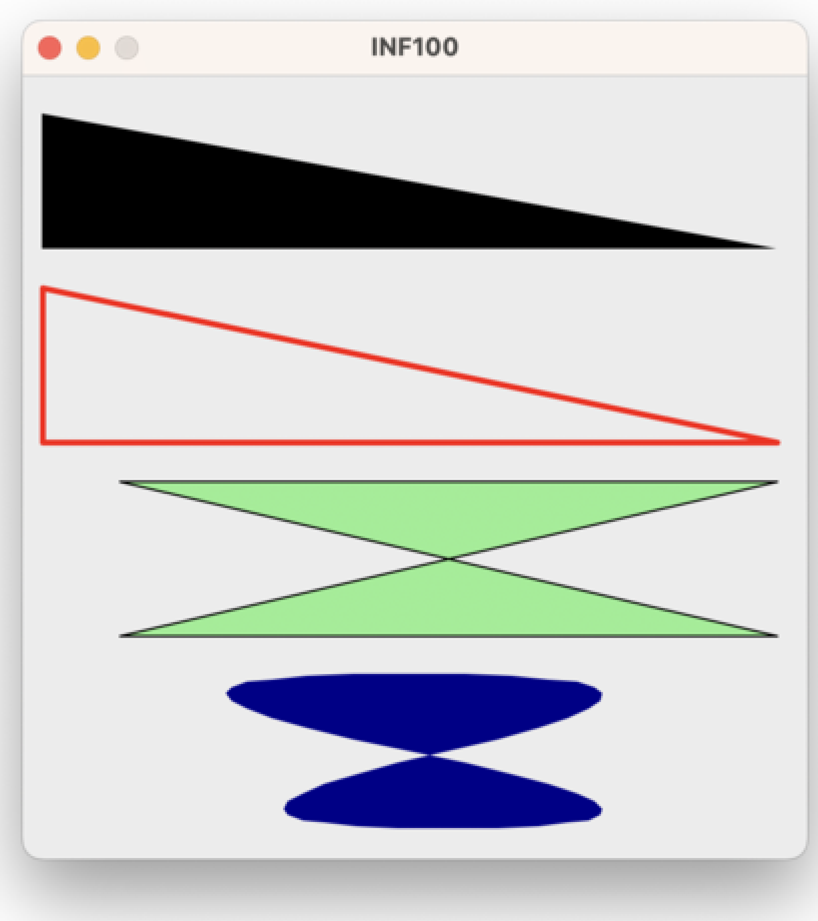
- For å tegne en polygon, må vi oppgi koordinatene til tre eller flere punkter. De første to parameterne er koordinatene til det første punktet, mens de neste to parameterne er koordinatene til det andre punktet, og de to neste er koordinatene til det tredje punktet, og så videre. Dette ligner på å tegne 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)
```

Kopier



create_arc

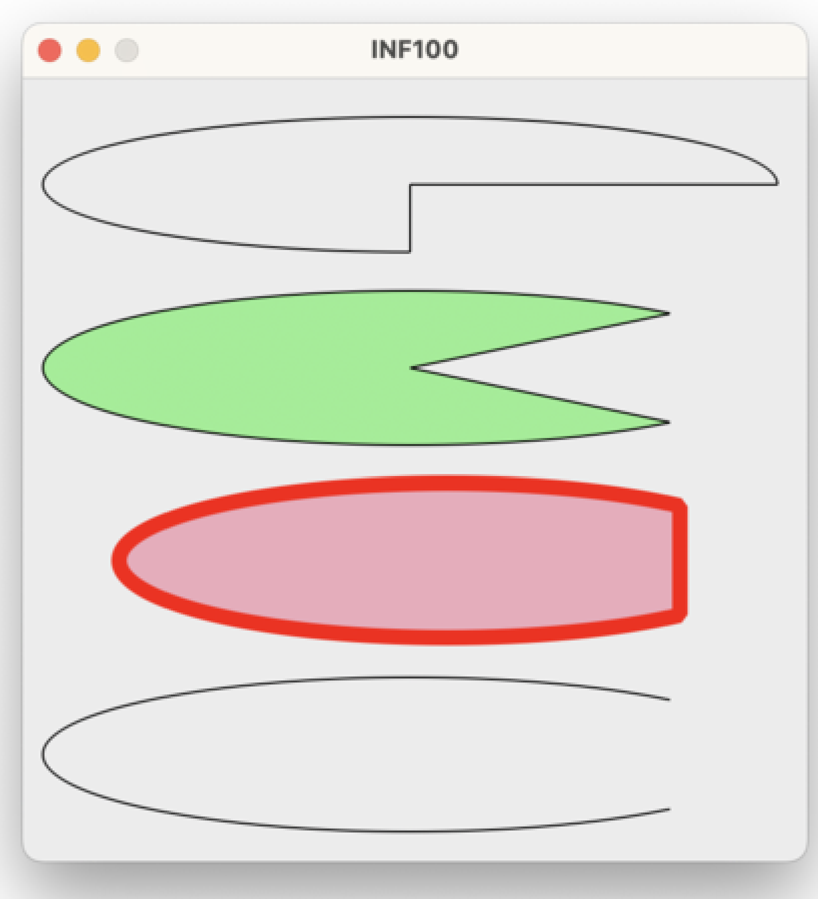
Påkrevde parametre *(x1, y1, x2, y2)*.

- 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'` .

```python
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, ...)*.

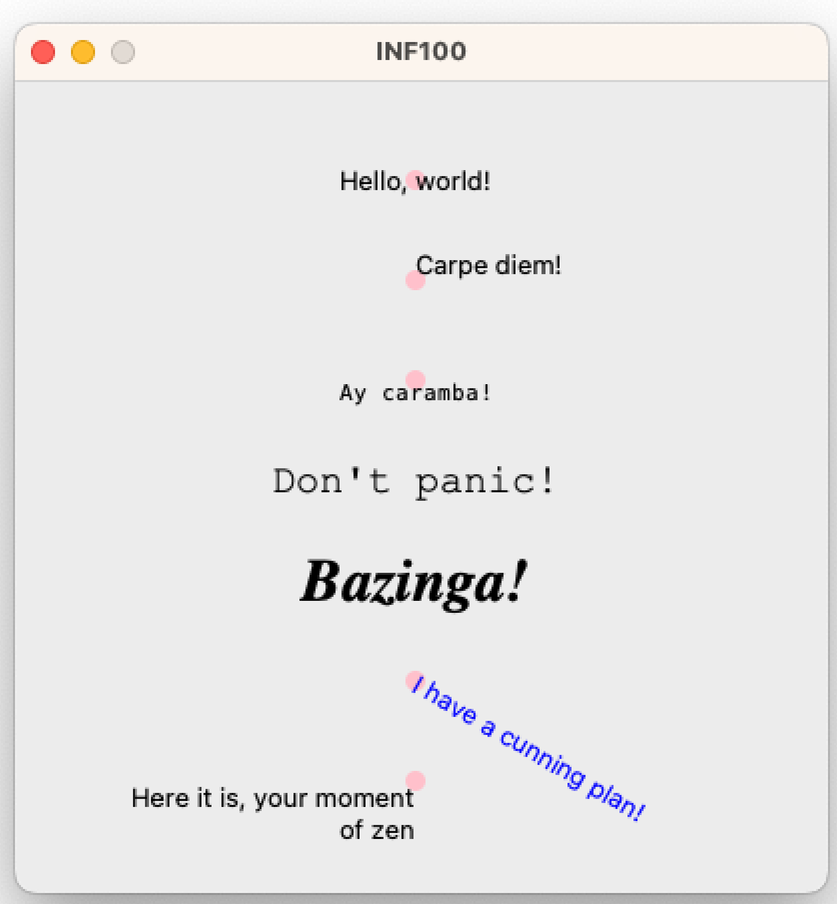
- Selve teksten som skal skrives oppgis ved å angi `text`.

- 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

- 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)
```



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.

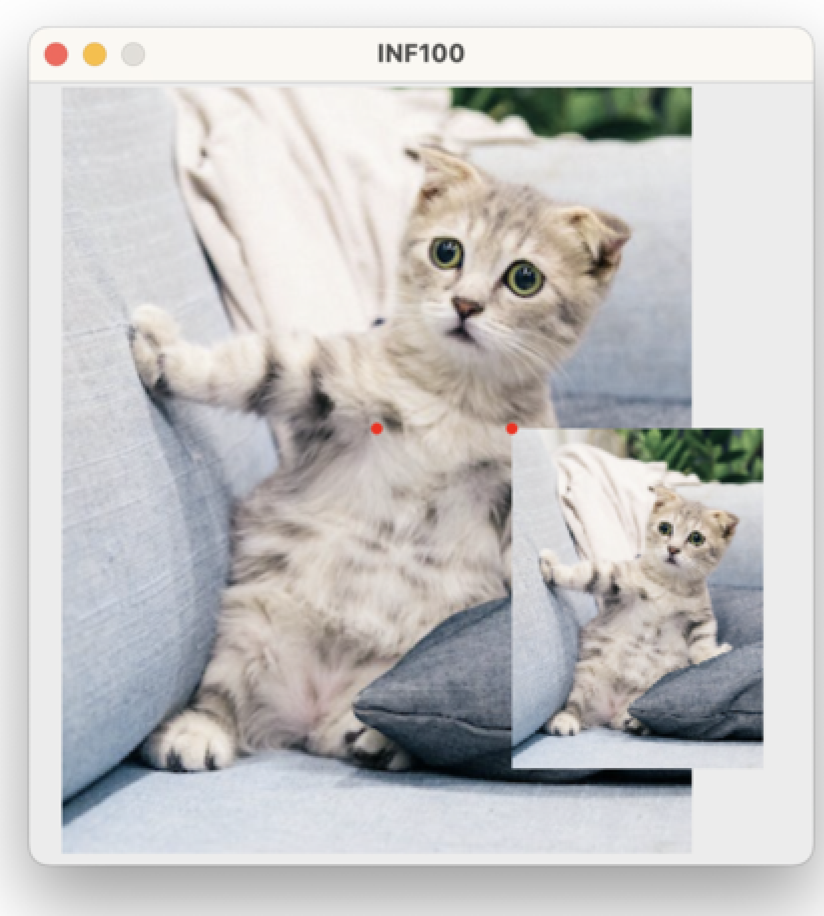
Valgfrie parametre *(pil_image, anchor, ...)*.

- 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.

```
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)
```

Kopier



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. Vi er imidlertid ikke begrenset til kun disse fargene.

Bakgrunn: piksler og farger som RGB

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) på Wikipedia, eller prøv RGB-kalkulatoren til w3schools.com.

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

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.

```
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)
```

Kopier

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*.

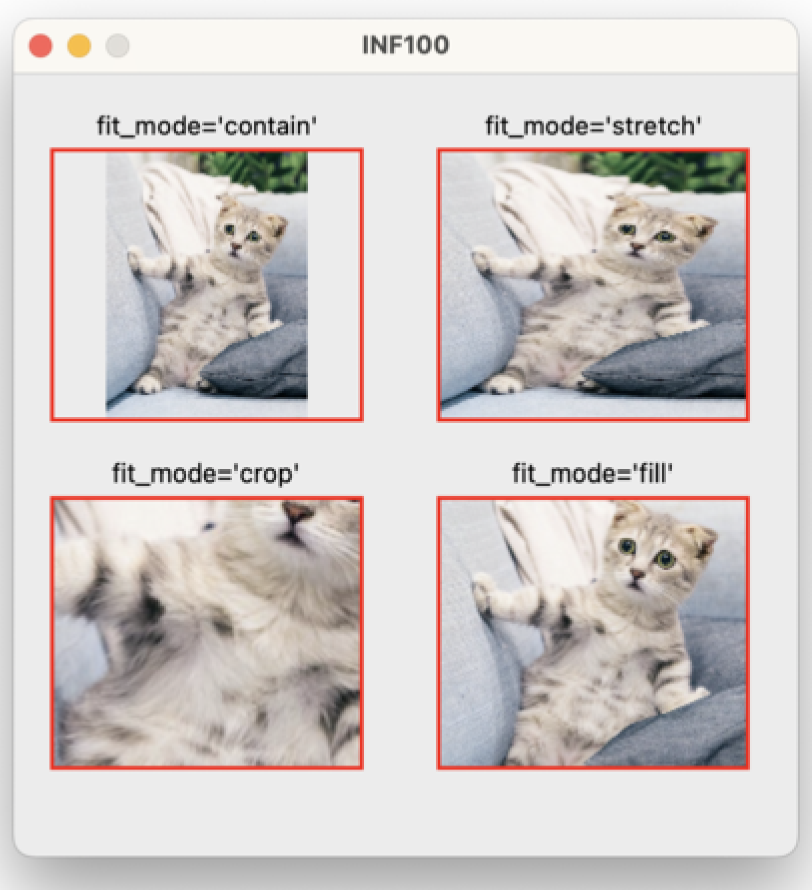
Påkrevde parametre *(canvas, x1, y1, x2, y2, pil_image)*.

- `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.

```
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)
```



Eksempel: buss

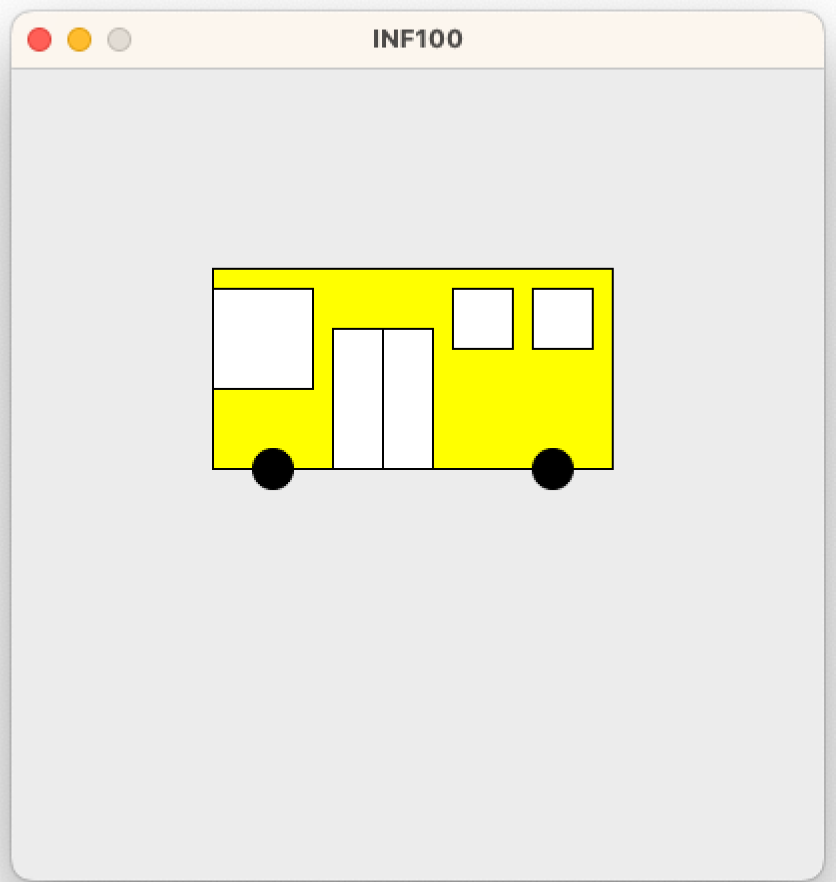
Video

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

```
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)
```

Kopier



Typer

- Vanlige typer
- Typen avgjør hva en operasjon betyr

Vanlige typer

Enhver verdi har en *type*. 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()

print("Flere viktige typer vi skal lære om senere")
print(type([1,2,3]))       # list
print(type((1,2,3)))       # tuple
print(type({1,2}))         # set
print(type({1:42}))        # dict
```

Kopier

Typen avgjør hva en operasjon betyr

```
# Asterisk (*) betyr forskjellige ting
print(3 * 2)               # 6
print(3 * "abc")           # "abcabcabc"

# Plusstegn (+) betyr også forskjellige ting
print(3 + 2)                # 5
print("3" + "2")            # "32"
```

Kopier

```
# Ikke alle operasjoner er definert for alle (kombinasjoner av) typer
```

```
print(3 + "def")          # Krasjer med TypeError
print("abc" * "def")      # Krasjer med TypeError
```

Kopier Se steg Kjør

Operatorer

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
```

```python
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)
```

Kopier Se steg Kjør

Aritmetikk med strenger

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

Kopier Se steg Kjør

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

Kopier Se steg Kjør

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:

```python
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.")
```

Kopier Se steg Kjør

Heltallsdivisjon er som vanlig divisjon, men runder alltid *nedover*.

```python
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)
```

Kopier Se steg Kjør

Modulo-operatoren (`%`) returnerer «resten» etter heltallsdivisjon. Vi kan bruke dette for å avgjøre om et tall er delelig med et annet (hvis resten er 0, er det delelig).

```
print(8 % 3) # 2
print(7 % 3) # 1
print(6 % 3) # 0    6 er delelig med 3
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
```

Kopier Se steg Kjør

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

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

Kopier Se steg Kjør

En siste vanlig bruk av modulo, er for å få en verdi til å «gå i ring», eller holde seg innenfor et visst intervall. For eksempel, la oss si at vi ønsker at en viss variabel alltid skal befinne seg innenfor intervallet 0-400. 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
```

Kopier Se steg Kjør

Sammenligning av verdier

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

```
print("== sammenligner 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("!= sammenligner 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
```

Kopier Se steg Kjør

```
print("< sammenligner 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("<= sammenligner 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
```

Kopier Se steg Kjør

Operatorene `>` og `>=` fungerer likt som `<` og `<=`, men med motsatt fortegn.

Flyttall og avrundingsfeil

Video

Se også videoen Floating point numbers 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 5.551115123125783e-17 (lite, men ikke 0!)
```

Kopier Se steg Kjør

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)  # Feil
print(almost_equals(0.1 + 0.1 + 0.1, 0.3))  # Riktig
```

Kopier Se steg Kjør

Medlemskap

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

```
# 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
print("ac" in "abc") # False (selv om både "a" og "c" er i "abc")
```

Kopier Se steg Kjør

```
# 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
```

Kopier Se steg Kjør

```
# 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
```

Kopier Se steg Kjør

```
# 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
```

Kopier Se steg Kjør

Logiske operatorer

Logiske operatorer bruker vi for å kombinere boolske verdier. De tre logiske operatorer er `and`, `or` og `not`.

```
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
```

Kopier Se steg Kjør

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("I find it quite hard to believe you're", age, "years old!")
```

Kopier Se steg Kjør

Uttrykk med betingelse (if else)

Video

```
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)
```

Kopier Se steg Kjør

Presedens og assosiativitet

Video

En av de aller vanligste feilene som gjøres (f. eks. på eksamen) er at man gjør feil antakelse om hvilken rekkefølge operatorer i et uttrykk utføres i; hvordan de «usynlige parentesene» i uttrykket er plassert.

Presedens

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 * )
```

Kopier Se steg Kjør

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)
```

Kopier Se steg Kjør

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`.

Parenteser

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 presedensrekkefølgen for operatorene gitt i tabellen øverst på siden 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) # (Brython/nettleser krasjer, men prøv i python)
print("a" and "b" in "box")
print("a" or "z" in "box")
```

Kopier Se steg Kjør

*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.*

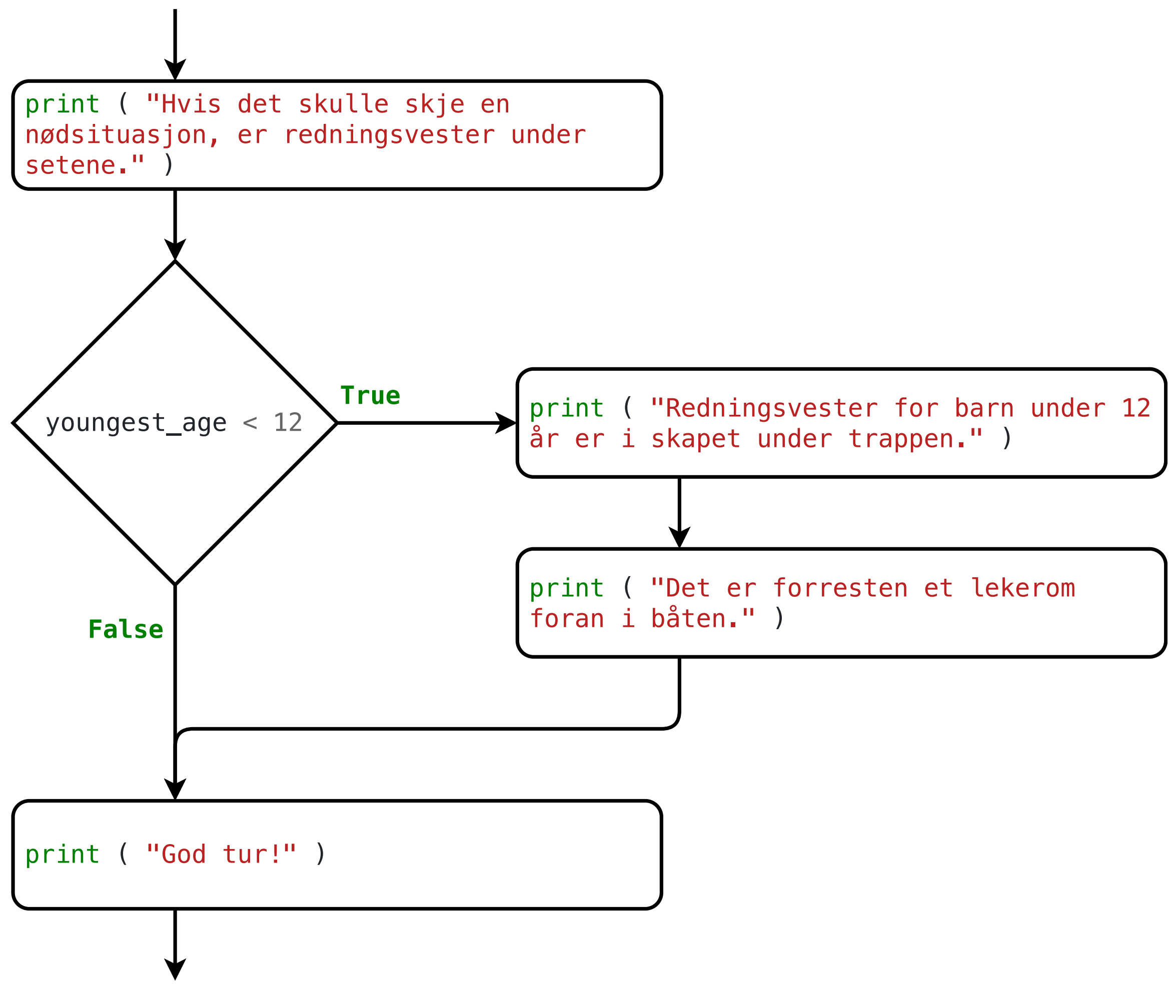
Betingelser

- If -setninger
- Betingelser
- If-else
- If-elif-else
- If-else -uttrykk
- God stil

If -setninger

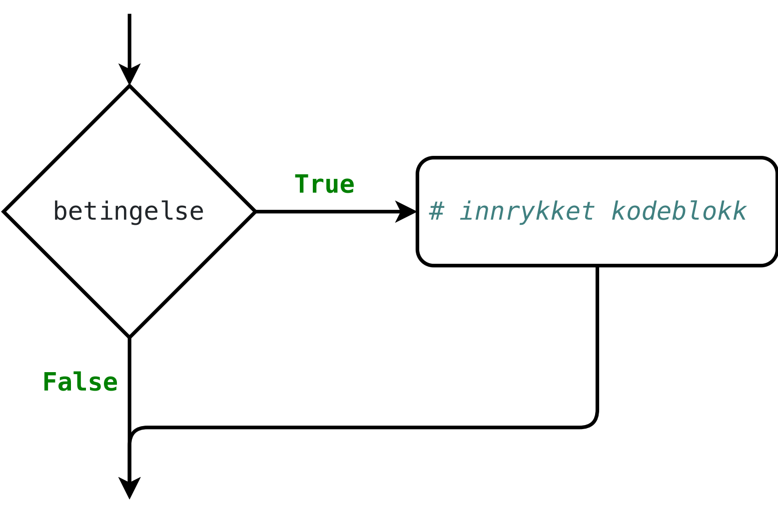
Man kan benytte en if-setning for å utøre en kodeblokk kun i gitte tilfeller.

```
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!")
```



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
```



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")
```

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

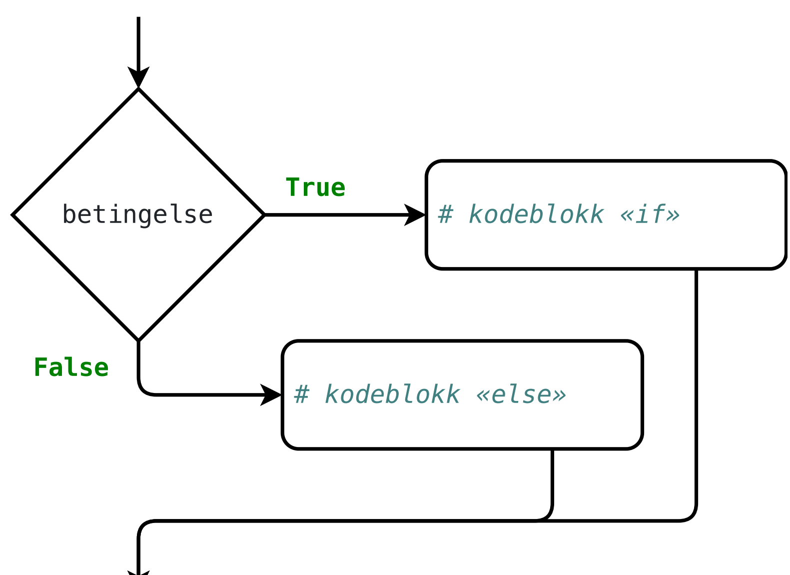
```
age = 18
residency = "Norway"
if (age >= 18) and (residency == "Norway"):
    print("Du kan stemme ved fylkes- og kommunevalget.")

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.")
```

Kopier Se steg Kjør

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:

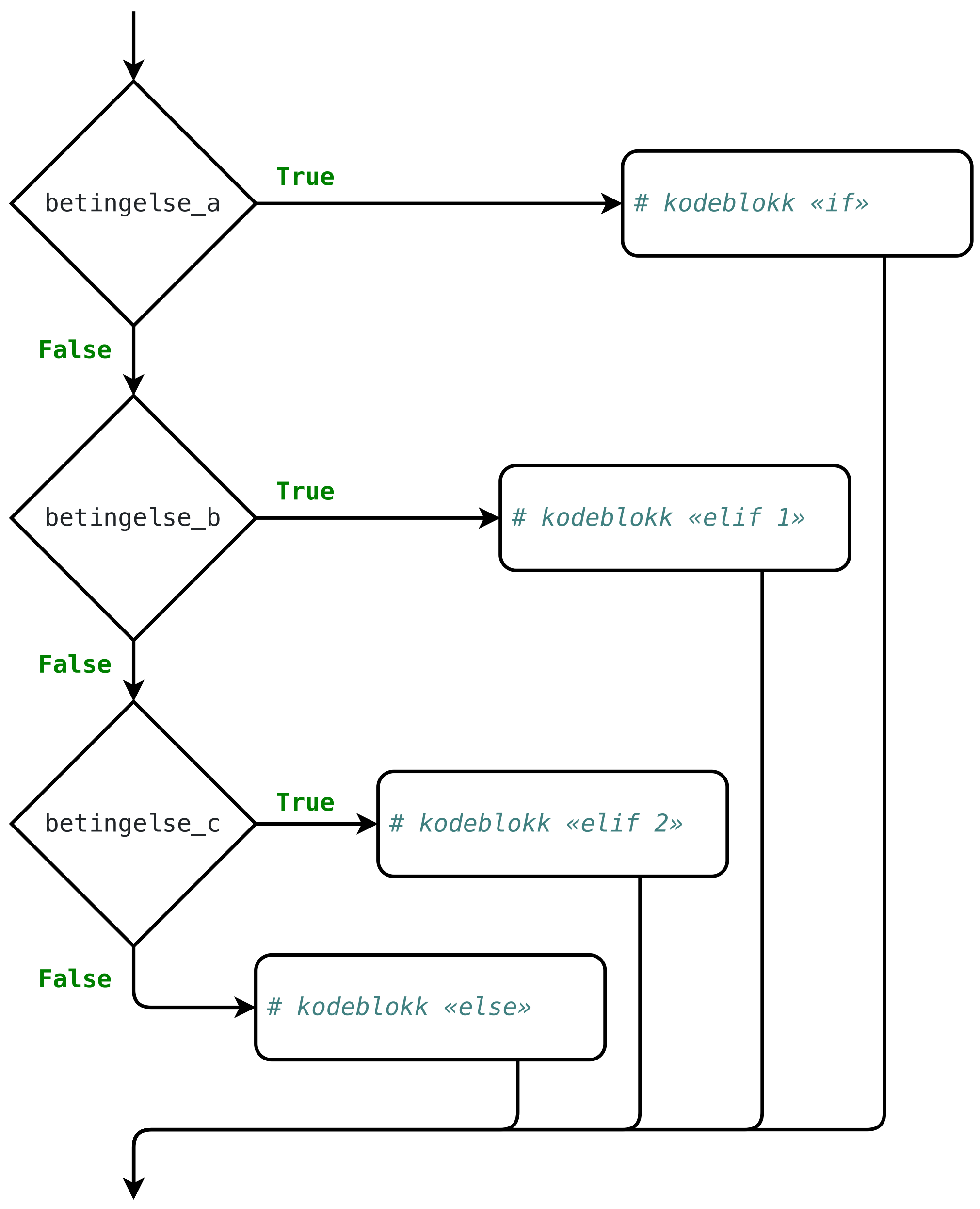
```
print("I hvilken kommune var du bostedsregistrert 30. juni i år?")
your_municipality = input()

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!")
```

Kopier | Se steg | Kjør

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»
```



Eksempel:

```
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}.")
```

Kopier Se steg Kjør

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)
```

Kopier Se steg Kjør

Se også avsnittet om uttrykk med betingelse i operatorer.

Truthy og falsy verdier

Video

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
```

Kopier Se steg Kjør

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

```
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
```

Kopier Se steg Kjør

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.

God stil

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

Negert betingelse:

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

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

Tom `if` -setning:

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

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

Unødvendig sammenligning med `True` / `False` :

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

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

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

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

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

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

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

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

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

```
# Dårlig
x = 10
if x < 5:
    print('small')
if (x >= 5) and (x < 10):
```

```
# Bra
x = 10
if x < 5:
    print('small')
elif x < 10:
```

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

```python
    print('medium')
elif x < 15:
    print('large')
else:
    print('extra large')
```

Fancy bruk av Python sin «aritmetikk»:

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

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

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

Løkker

- While-løkker
- Uendelig løkke
- Break og continue
- For-løkker og range
- Nøstede løkker
- Primtall
- Stil

While-løkker

Video

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 skal være snill", x)
    x += 1

print("Ferdig!")
```

Kopier Se steg Kjør

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 den samme operasjonen flere ganger, 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")
```

Kopier Se steg Kjør

Et irriterende program.[1]

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

Kopier

Et program for å telle antall siffer i et tall.

```
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}.")
```

Kopier Se steg Kjør

Uendelig løkke

Video

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 a endres ikke
```

Kopier Se steg Kjør

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

Break og continue

Video

Break benyttes for å bryte ut av en løkke.

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

Kopier

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

```
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.")
```

Kopier

For-løkker og range

Video

Det er ofte at vi vet på forhånd hvor mange ganger vi ønsker at løkken skal kjøres. Da bør vi benytte en *for*-løkke.

En for-løkke er kortere å skrive enn en while-løkke, og reduserer faren for feil og bugs.

```python
print("------------For-løkke--------------")

for i in range(5):
    print("Jeg skal være snill", i)
print("Ferdig!")


print("------------While-løkke------------")

i = 0
while i < 5:
    print("Jeg skal være snill", i)
    i += 1
print("Ferdig!")
```

Kopier | Se steg | Kjør

Man kan angi hvor en for-løkke begynner og slutter hvor store hopp som skal gjøres mellom hver iterasjon.

```python
print("---------For-loop--------")
for x in range(2, 19, 3):
    print(x)

print("---------While-loop---------")
x = 2
while x < 19:
    print(x)
    x += 3
```

Kopier | Se steg | Kjør

```python
# Skriv ut tallene mellom a og b
a = 3
b = 10

print("---------For-loop---------")
for i in range(a, b + 1):
    print(i, end=" ")
print()

print("---------While-loop---------")
i = a
while i <= b:
    print(i, end=" ")
    i += 1
```

```
print()
```

Kopier Se steg Kjør

`range()` -funksjonen gir oss en ordnet samling med tall. Funksjonen kan ta ett, to eller tre heltalls-argumenter.

```
# Ett argument: begynner med 0 og går opp til (og ikke
# inkludert) det gitte argumentet. Eksempel: 0 1 2 3
for i in range(4):
    print(i, end=" ")
print()

# To argument: tallene begynner på det første tallet,
# og går opp til (og ikke inkludert) det andre tallet.
# Eksempel: 6, 7, 8, 9, 10
for i in range(6, 11):
    print(i, end=" ")
print()

# Tre argument: de to første argumentene betyr det
# samme som for to argumenter. Det tredje argumentet
# beskriver hvor store hopp som gjøres mellom hver
# iterasjon. Eksempel:  3, 5, 7, 9 (hopper med 2)
for i in range(3, 11, 2):
    print(i, end=" ")
print()
```

Kopier Se steg Kjør

```
# Hopp kan også være negativ. Eksempel: 10 9 8 7 6
for i in range(10, 5, -1):
    print(i, end=" ")
print()
```

Kopier Se steg Kjør

```
# En range kan være tom
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()
print("---")
```

Kopier | Se steg | Kjør

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()
```

Kopier | Se steg | Kjør

Nøstede løkker

Video

```
# 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()
```

Kopier | Se steg | Kjør

```
# Hvilken figur tegner vi her?
height = 5

for row in range(height):
    for col in range(row):
        print("*", end="")
    print()
```

Kopier | Se steg | Kjør

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.

```python
# Dårlig
repetitions = 5
x = 0
while x < repetitions:
    print("Jeg skal være snill", x)
    x += 1
```

```python
# Bra
repetitions = 5
for x in range(repetitions):
    print("Jeg skal være snill", x)
```

---

1. Fra Automate the Boring Stuff with Python. Al Sweigart, CC BY-NC-SA 3.0. ↩

Strenger

- Basics
- Fire måter å skrive strenger
- Linjeskift og escape-sekvenser
- Konvertering til strenger
- Konvertering og formatering med f-strenger
- Operasjoner og metoder
- 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: kompabilitet
# 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")
```

Kopier Se steg Kjør

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 indentering (altså ingen
indentering).
""")
```

Kopier Se steg Kjør

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))
```

Kopier Se steg Kjør

Konvertering til strenger

Verdier som ikke er strenger kan konverteres til en streng-representason med bruk av funksjonene `str` og `repr`.

```
def print_string_conversion(x):
    print("type:", 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 typer 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("   Mellomrom\ttab   ")
print_string_conversion("Linje\nskift")

# 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)
```

Kopier | Se steg | Kjør

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
```

Kopier | Se steg | Kjør

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.
- 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.

```python
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("Venstrejustert")
print(f"** {x:>10} **") # **         10 **
print(f"** {s:>10} **") # **        abc **
print()

print("Høyrejustert")
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 **
```

Kopier | Se steg | Kjør

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
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

Konstanter

```
import string
print(string.ascii_letters)   # abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
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("-----------")
```

Kopier | Se steg | Kjør

Operasjoner og metoder

Noen 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
```

Kopier | Se steg | Kjør

En *metode* er en funksjon som kalles «på» et objekt/en verdi. Kallet utføres ved hjelp at et punktum mellom objektet og metode-navnet (se eksempler under). Ulike typer har ulike metoder tilgjengelig. Her er noen metoder på typen `str`:

```
# .upper og .lower endre teksten til bare store eller små bokstaver
s = "FooBar"
print(s)
print(s.lower())
print(s.upper())
print("---")

# .replace bytter ut substrenger
print(s.replace("o", "ahr"))
print("hahahaha".replace("hah", "l"))
print("hahahaha".replace("hah", "h"))
print("---")

# .split() deler opp en streng i biter, og legger bitene i en liste
names = "Marshall,Rubble,Chase,Rocky,Zuma,Sky"
```

```
print(names)
print(names.split(","))
print("---")

# .join() limer sammen strenger med en limestreng
print("+".join(names.split(",")))
print(s.join("ABC"))
print("---")

# .strip() fjerner whitespace foran og bak
s = "   FooBar  \n "
print(s, len(s))
print(s.strip(), len(s.strip()))
print("---")
```

Kopier | Se steg | Kjør

Flere metoder

```
# Kjør koden for å se en tabell av hva funksjonene returnerer
def print_cell(test):
    print(f"{str(test):9}", end="")

def print_row(s):
    print(f" {s:4}  ", end="")
    print_cell(s.isalnum())
    print_cell(s.isalpha())
    print_cell(s.isdigit())
    print_cell(s.islower())
    print_cell(s.isspace())
    print_cell(s.isupper())
    print()

def print_table():
    print("  s   isalnum  isalpha  isdigit  islower  isspace  isupper")
    for s in "ABCD,ABcd,abcd,ab12,1234,-123,1.0,    ,AB?!".split(","):
        print_row(s)

print_table()
```

Kopier | Se steg | Kjør

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!
```

Kopier | Se steg | Kjør

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)
```

Kopier | Se steg | Kjør

Negative indekser

```
s = "abcdefgh"
print(s)
print(s[-1]) # Snarvei for s[len(s) -  1]
print(s[-2])
```

Kopier | Se steg | Kjør

Beskjæring (engelsk: slicing)

```
# 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("---")
```

Kopier | Se steg | Kjør

Beskjæring med default-verdier

```
s = "abcdefgh"
print(s)       # abcdefgh
print(s[3:])   # defgh
print(s[:3])   # abc
print(s[:])    # abcdefgh
print("---")
```

Kopier | Se steg | Kjør

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("---")
```

Kopier | Se steg | Kjør

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!
```

Kopier | Se steg | Kjør

Løkker over strenger

Med indeksering

```
s = "abcd"
# Vanlig for-løkke over lengden til s
for i in range(len(s)):
    print(i, s[i])

# 0 a
# 1 b
# 2 c
# 3 d


print("---")
# Med enumerate blir det to iterander, både indeks og selve tegnet
for i, c in enumerate(s):
    print(i, c)

# 0 a
# 1 b
# 2 c
# 3 d
```

Kopier | Se steg | Kjør

Uten indeksering

```
s = "abcd"
for c in s:
    print(c)

# a
# b
# c
# d
```

Kopier | Se steg | Kjør

Oppdeling med `split`

```
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
```

Kopier | Se steg | Kjør

Oppdeling med `splitlines`

```
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 tr
```

Kopier Se steg Kjør

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'
```

Kopier Se steg Kjør

Palindromer

Et palindrom er en streng som er lik fremlengs og baklengs.

```
# 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
```

```python
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"))
```

Kopier | Se steg | Kjør

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. Python benytter som standard en enkoding som heter UTF-8. I denne enkodingen matches hvert enkelt tegn med en såkalt *unicode*-verdi (også kalt *ordinal*) 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.

Vi kan se en oversikt over en del vanlige tegn og deres unicode-verdi på wikipedia. Vi kan for eksempel lese i tabellen at tegnet `A` har verdien 65 (i desimal), mens symbolet `a` har verdien 97.

```python
# For å finne unicode-verdien (ordinal) til et tegn
c1 = "A"
u1 = ord(c1)
print(c1, u1)
```

```python
# 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()
```

Kopier Se steg Kjør

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

```python
print('"A" < "a":', "A" < "a") # True, siden 65 < 97 er True
print('"a" < "A":', "a" < "A") # False

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
print()
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
```

Kopier Se steg Kjør

Eksempel på bruk av ordinaler: simpel kryptering.

```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))
```

Kopier | Se steg | Kjør

Lese og skrive til fil

```
# Du kan kopiere read_file og write_file -funksjonene og bruke dem
# i din egen kode

def read_file(path):
    """ Given the file path (file name) of a plain text file, returns
    the content of the file as a string. """
    with open(path, "rt", encoding='utf-8') as f:
        return f.read()

def write_file(path, contents):
    """ Writes the contents to the file with the given file path. If
    the file does not exist, it will be created. If the file does
    exist, its old content will be overwritten. """
    with open(path, "wt", encoding='utf-8') as f:
        f.write(contents)

# Eksempler på bruk.
# Vi oppretter en fil foo.txt med et gitt innhold
contents_to_write = "Dette er en test!\nDet er bare en test!"
write_file("foo.txt", contents_to_write)

# V leser en fil foo.txt og lagrer innholdet som en streng
contents_read = read_file("foo.txt")
assert(contents_read == contents_to_write)

print("Sjekk at filen foo.txt ble opprettet (legg merke til i hvilken mappe),"
    + " og kikk på innholdet.")
```

Kopier

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:

```python
import os
cwd = os.getcwd()
print(cwd)
```

Kopier

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 `read_file("foo.txt")` 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 eksemepel), eller
- endre funksjonskallet til `read_file("labX/foo.txt")` , 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 *mulig* å programmatisk endre cwd til å bli samme mappe som filen som kjøres ligger i:

```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 hele programmet hvis man vil bruke det i en annen mappe.

Funksjoner

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

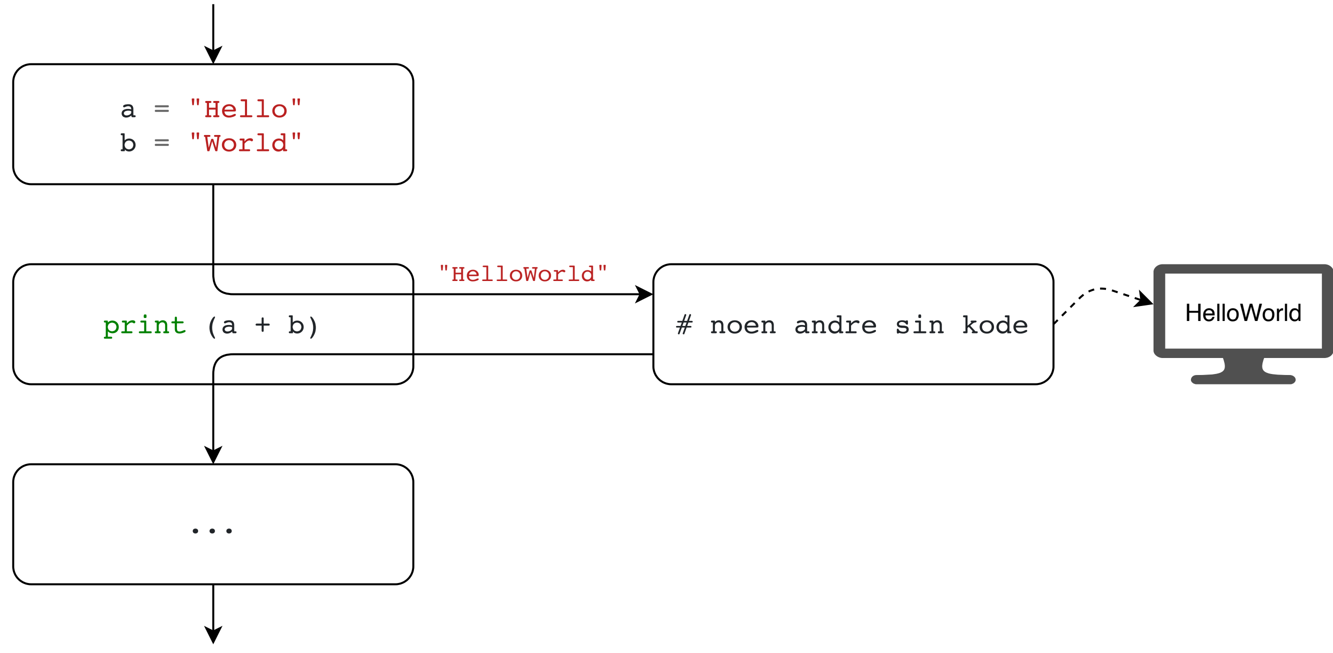
Funksjonskall

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)
...
```

Kopier Se steg Kjør

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).



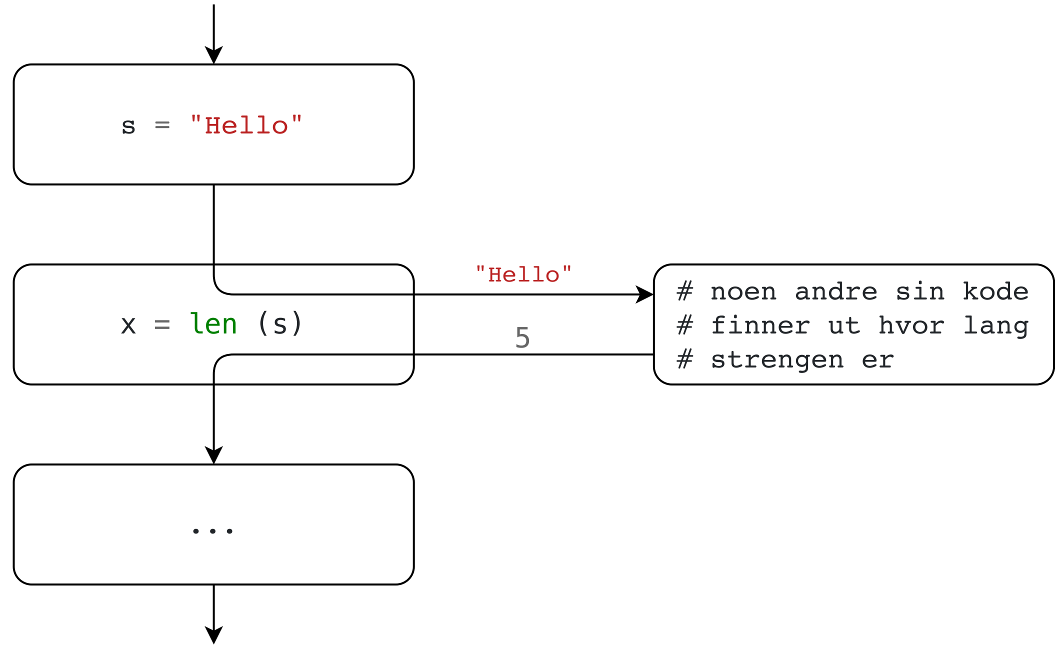
Funksjonskall med returverdi

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)
...
```

Kopier Se steg Kjør

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.



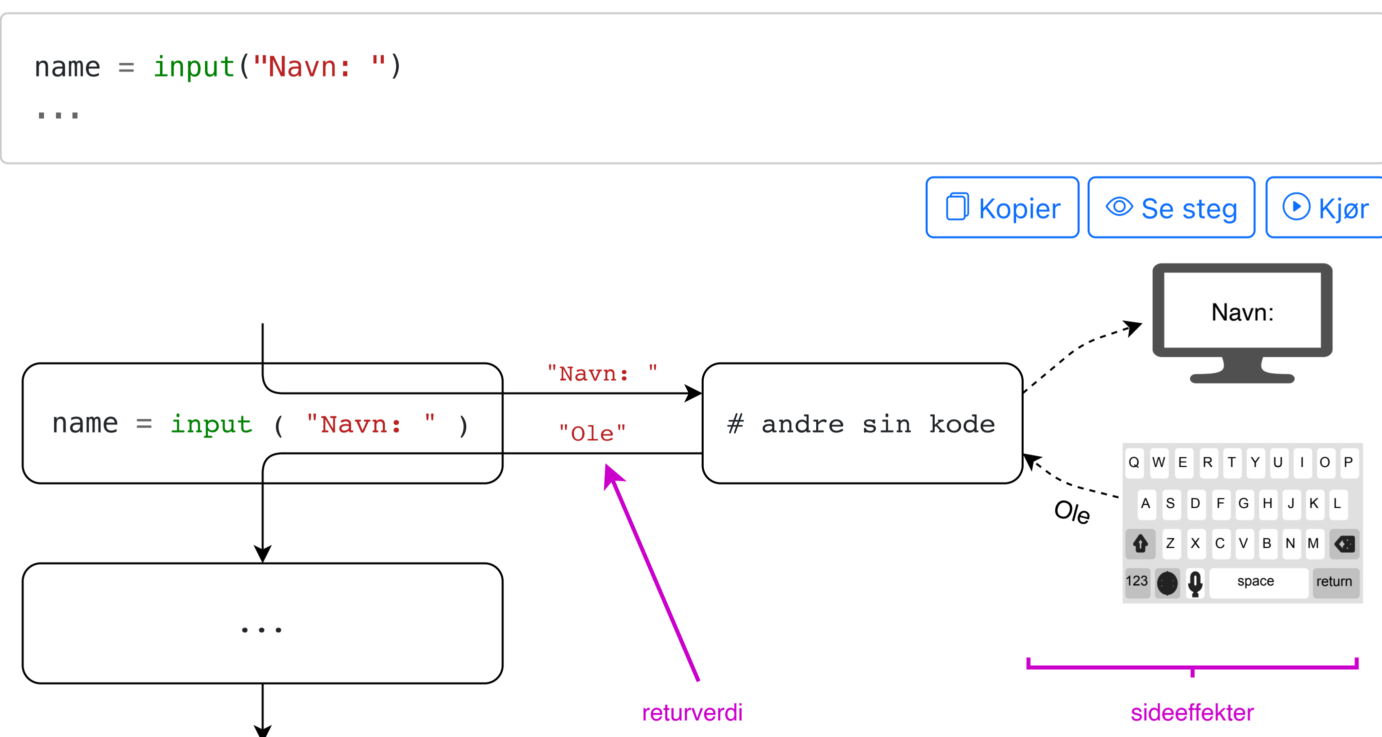
Mer om returverdier

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: ")
...
```

Kopier Se steg Kjør



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

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.

Hensikten med funksjoner

Abstraksjon

Vår første funksjon

Video

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.

```
# 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()
```

Kopier Se steg Kjør

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()
```

Kopier Se steg Kjør

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.

```
# Vi definerer en funksjon som heter «my_sample_function»
def my_sample_function(my_crazy_parameter, my_insane_parameter):
```

```
    print("*********")
    print("*", my_crazy_parameter)
    print("*", my_insane_parameter)
    print("*********")

# Vi kaller my_sample_function
my_sample_function("foo", "bar")
my_sample_function("baz", "qux")
```

Kopier | Se steg | Kjør

Parameter. En funksjon har som regel noen ukjente variabler den trenger for å gjøre jobben sin. De ukjente variablene kalles *parametre* til funksjonen. Den som definerer en funksjon bestemmer selv hva parameterne heter. Når funksjonen kalles må parametrene til funksjonen fylles med argumenter.

Argumenter vs parametre

En funksjon må være definert *før* den kalles

```
find_age() # Krasjer, funksjonen find_age er ikke definert enda

def find_age():
    print("Hvilket år ble du født?")
    birth_year = int(input())
    age = 2023 - birth_year
    print(f"Du blir {age} år i år")
```

Kopier | Se steg | Kjør

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()
```

Kopier Se steg Kjør

*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, 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
```

Kopier Se steg Kjør

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

```
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) # None

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

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

Kopier | Se steg | Kjør

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
```

Kopier | Se steg | Kjør

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...
```

Kopier | Se steg | Kjør

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.

```python
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!
```

Kopier | Se steg | Kjør

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!
```

Kopier | Se steg | Kjør

Skop

Video

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
```

Kopier | Se steg | Kjør

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

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

Kopier | Se steg | Kjør

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)
```

```python
    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))
```

Kopier | Se steg | Kjør

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:

```python
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()
```

Kopier | Se steg | Kjør

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.

```
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)))
```

Feil og debugging

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 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.

Eksempler på syntaks-feil

Krasj (kjøretidsfeil)

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

Eksempler på kjøretidsfeil

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
```

Kopier Se steg Kjør

Assert

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.

```
something_is_as_it_is_supposed_to_be = True
if_this_is_false_then_something_is_wrong = False # Noe er feil!
assert(something_is_as_it_is_supposed_to_be) # Her skjer det ingen ting
assert(if_this_is_false_then_something_is_wrong) # Krasjer!

#   File "/demo/demo.py", line 4, in <module>
#     assert(if_this_is_false_then_something_is_wrong) # Krasjer!
# AssertionError
```

Kopier Se steg Kjør

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

```
def sum_of_nums_between(nums, lo, hi):
    assert(lo <= hi) # Sjekker at lo faktisk er mindre enn hi
    return sum(nums[lo:hi])

print(sum_of_nums_between([0, 1, 2, 3], 0, 2)) # Fungerer fint
print(sum_of_nums_between([0, 1, 2, 3], 2, 0)) # Krasjer
```

Kopier Se steg Kjør

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!
```

Kopier | Se steg | Kjør

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. Vi kan imøtekomme dette noe ved å lage vår egne, forbedrede assert-funksjoner.

```
def assert_almost_equals(expected, actual, threshold=0.0000001):
    if not abs(expected - actual) <= threshold:
        raise AssertionError(f"Expected {expected} but actual was {actual}")

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

assert_almost_equals(5, distance(0, 3, 4, 0)) # Bedre feilmelding
```

Kopier | Se steg | Kjør

Print

Assert-setninger kan fortelle oss at noe er feil hvis vi klarer å tenke ut på forhånd hva slags feil som kan oppstå. 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.

Eksempel på debugging med print-setning: nth_prime

Ulempen med print-setninger er at man må fjerne dem når man er ferdig med å fikse bug'en.

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. 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:

Del 1: Introduksjon | Del 2: Step over | Del 3: Step into | Del 4: Eksempel

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 av en eller annen form. I større prosjekter kan man gjerne ha egne filer som kun inneholder tester for "den ekte" koden.

```
import other_file as M

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

Kopier | Se steg | Kjør

I dette kurset har de fleste 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.

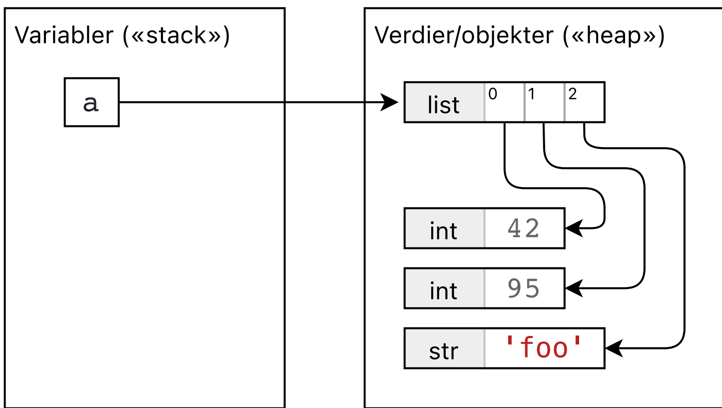
Lister

- 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
- Listeforståelse (løkker inni lister)
- Konvertering mellom lister og strenger (split/join)

Hva en liste er (egentlig)

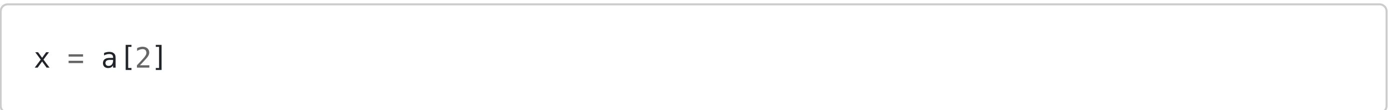
Til vanlig tenker vi på en liste som en samling av verdier; men egentlig bør vi tenke på det som en samling av *referanser* til verdier. Under viser vi en illustrasjon av minnet til datamaskinen etter at vi har opprettet en liste med tre elementer.

```
# Oppretter en liste a med tre elementer
a = [42, 95, 'foo']
```

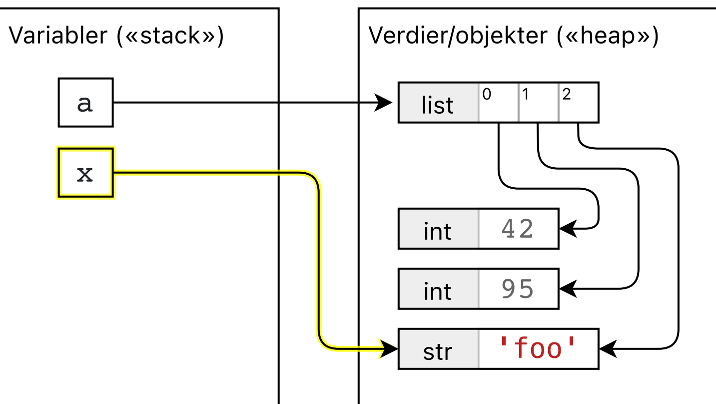


Vi kan hente ut enkelt-verdier fra en liste ved å slå opp i listen med indeksering. Indeksering starter på 0, så første element i listen har indeks 0, andre element har indeks 1, osv.

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:

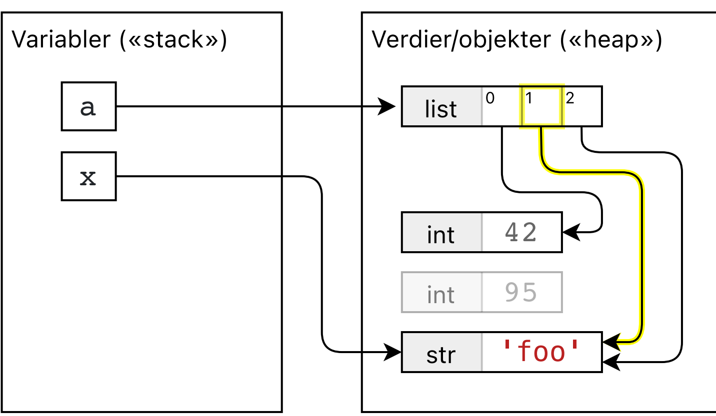


```
x = a[2]
```

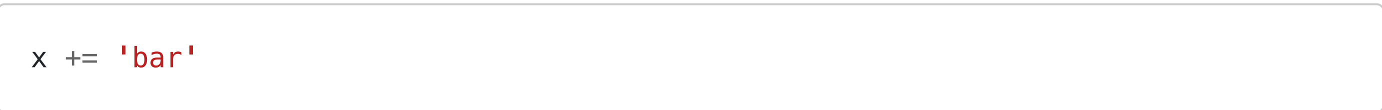


I motsetning til de verdi-typene vi har sett tidligere i emnet, er det mulig å endre en liste *uten* å opprette et helt nytt objekt i minnet. Dette kalles å *mutere* en liste. Eksempel: hvis vi angir at verdien til *a[1]* skal settes til verdien av *x*, vil minnet endres som vist under:

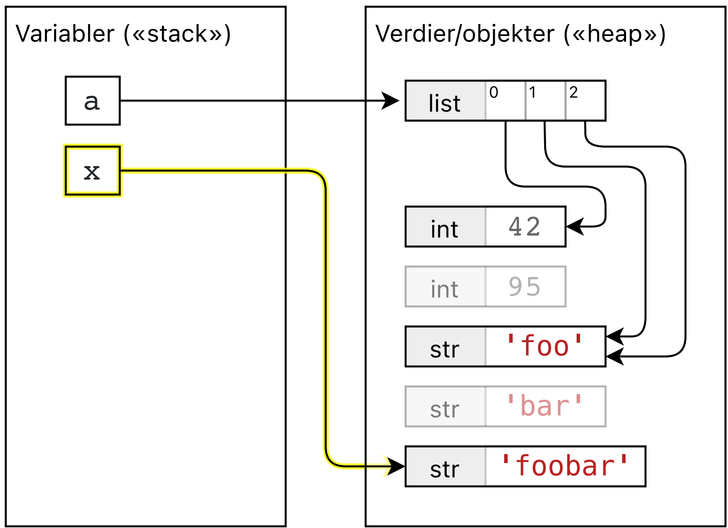
```
a[1] = x
```



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.

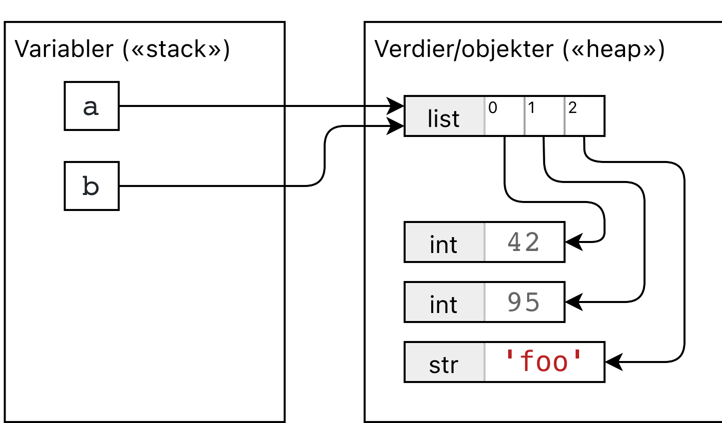
Hele programmet over

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.

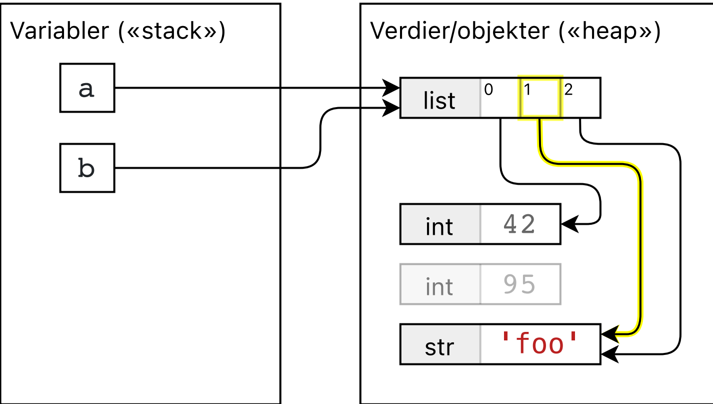
Hvis to variabler peker på samme liste, kaller vi dem *aliaser* for hverandre.

```
a = [42, 95, 'foo']
b = a                   # b er nå et alias for a
```



Om vi muterer listen gjennom `a` vil altså `b` påvirkes (og vice versa).

```
a[1] = 'foo'          # vi muterer a
```



Komplett eksempel:

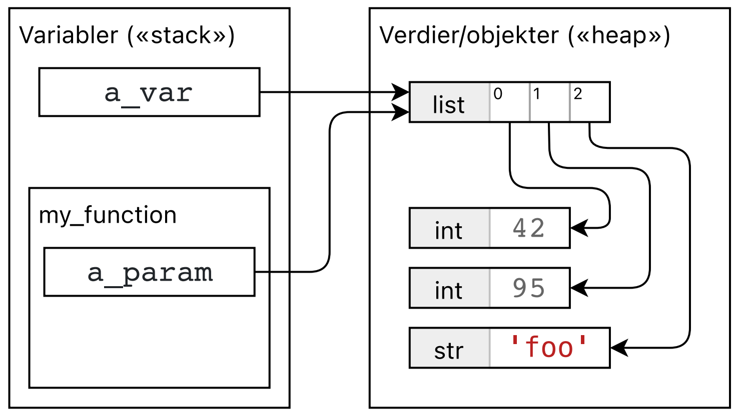
```
a = [42, 95, 'foo']
b = a                 # b er nå et alias for a

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']
```

Kopier | Se steg | Kjør

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.

```
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']
```

Kopier Se steg Kjør

Variabler («stack»)
a_var
my_function
a_param
Verdier/objekter («heap»)
list 0 1 2
int 42
int 95
str 'foo'

Opprette lister

Tomme lister

```python
# 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'>
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

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, '']
```

Kopier Se steg Kjør

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]
```

Kopier Se steg Kjør

Funksjoner og operasjoner

```
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 eleme
print('d < a', (d < a)) # True
print('d > a', (d > a)) # False
```

Kopier Se steg Kjør

```
# 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]
```

Kopier Se steg Kjør

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]
```

Kopier Se steg Kjør

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]
```

Kopier Se steg Kjør

Dersom to variabler refererer til samme muterbare objekt, kalles de for *aliaser*. Funksjonsparametre er eksempler på aliaser.

```
# 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]
```

Kopier Se steg Kjør

Vi kan benytte `is` og `is not` -operatorene for å sjekke om to variabler er aliaser for samme objekt.

```
# 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
```

Kopier Se steg Kjør

Kopiering av lister

```
# 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]
```

Kopier Se steg

Andre måter å kopiere

```
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 muter
print(a, b, c, d, e, f, g, h, li)
```

Kopier | Se steg | Kjør

Destruktive funksjoner

En funksjon er *destruktiv* dersom den har sideeffekter som muterer en parameter (eller hvis den muterer en global variabel).

```
# 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]
```

Kopier | Se steg | Kjør

En *ikke-destruktiv* funksjon vil ikke ha sideeffekter, og vi benytter oss av returverdien i stedet.

```
import copy

def destructive_remove_all(a, value):
    while value in a:
        a.remove(value)

def non_destructive_remove_all(a, value):
```

```python
    # 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]
```

Kopier Se steg Kjør

Leting etter elementer

```python
# 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
```

Kopier Se steg Kjør

```python
# 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
```

Kopier Se steg Kjør

```python
# Hvor i listen befinner verdien seg, da?
# a.index(element) eller a.index(element, start)
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(2,4) =", a.index(2,4)) # 6
```

Kopier Se steg Kjør

```python
# 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...")
```

Kopier Se steg Kjør

```python
# 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!")
```

Kopier Se steg Kjør

Legge til elementer

Destruktive metoder for å legge til elementer:

```python
# 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]
```

Kopier Se steg Kjør

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)
```

Kopier Se steg Kjør

Destruktiv vs. ikke-destruktiv utvidelse

```
print("Destruktiv:")
a = [2, 3]
b = a           # lager alias
a += [4]
print(a)
print(b)
```

```python
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)
```

Kopier | Se steg | Kjør

Fjerne elementer

Destruktive metoder for å fjerne elementer

```python
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)
```

Kopier | Se steg | Kjør

Ikke-destruktive operasjoner for å fjerne elementer

```python
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)
```

Kopier Se steg Kjør

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)
```

Kopier Se steg Kjør

```
# 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)
```

Kopier Se steg Kjør

```
# 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")
```

Kopier Se steg Kjør

```
# 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]
```

Kopier Se steg Kjør

```
# 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]
```

Kopier Se steg Kjør

```
# 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)
```

Kopier Se steg Kjør

Sortering og reversering

Destruktiv sortering og reversering

```
# 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)
```

Kopier Se steg Kjør

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("---")
```

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")
```

Kopier | Se steg | Kjør

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=}")
```

Kopier | Se steg | Kjør

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    = }")
```

Kopier | Se steg | Kjør

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)
```

Kopier Se steg Kjør

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")
```

Kopier Se steg Kjør

En funksjon kan akseptere et ukjent antall argumenter ved å pakke dem inn i en liste

```
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))
```

Kopier Se steg Kjør

Eller kreve et minimums antall argumenter ved å bare pakke inn bare de siste argumentene i en liste.

```
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
```

Kopier Se steg Kjør

Tupler

En *tuple* er en slags liste som ikke kan muteres.

```
t = (1, 2, 3)
print(type(t), len(t), t)

a = [1, 2, 3]
t = tuple(a)
print(type(t), len(t), t)
```

Kopier Se steg Kjør

```
# Tupler kan ikke muteres
t = (1, 2, 3)
print(t[0])

t[0] = 42    # Krasj!
print(t[0])
```

Kopier Se steg Kjør

```
# 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)
```

Kopier Se steg Kjør

```
# 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)
```

Kopier Se steg Kjør

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}")
```

Kopier | Se steg | Kjør

Listeinklusjon / list comprehension (løkker inni lister)

I Python er det mulig å opprette lister med en løkke. Dette kalles for *list comprehension* på engelsk (*listeinklusjon*).

```
# Den lange måten
a = []
for i in range(10):
    a.append(i + 1)
print(a)

# Med listeinklusjon
a = [i + 1 for i in range(10)]
print(a)

# For de ambisiøse: listeinklusjon med betingelser
a = [i + 1 for i in range(20) if i % 2 == 0]
print(a)

# Listeinklusjon 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)

# Listeinklusjon 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)
```

Kopier Se steg Kjør

Konvertering mellom lister og strenger (split/join)

```
# 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?
```

Kopier Se steg Kjør

Flerdimensjonelle lister

- 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
```

Kopier Se steg Kjør

```
# 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
```

Kopier Se steg Kjør

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.

```
# 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]]
```

Kopier Se steg Kjør

```
# 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]]
```

Kopier Se steg Kjør

```
# Alternativ RIKTIG måte å opprette en 2D-liste med dynamisk størrelse
# Her benytter vi listeinklusjon (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]]
```

Kopier Se steg Kjør

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.

```
# 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
```

Kopier Se steg Kjør

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)
```

Kopier | Se steg | Kjør

Hente ut rader og kolonner

Få tilgang til en hel rad.

```
# 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)
```

Kopier | Se steg | Kjør

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)
```

Kopier | Se steg | Kjør

Få tilgang til en hel kolonne med listeinklusjon.

```
# 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)
```

Kopier | Se steg | Kjør

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)
```

Kopier | Se steg | Kjør

```
# 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)
```

Kopier | Se steg | Kjør

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.

```
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]}')
```

Kopier Se steg Kjør

Grafiske brukergrensesnitt

- 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

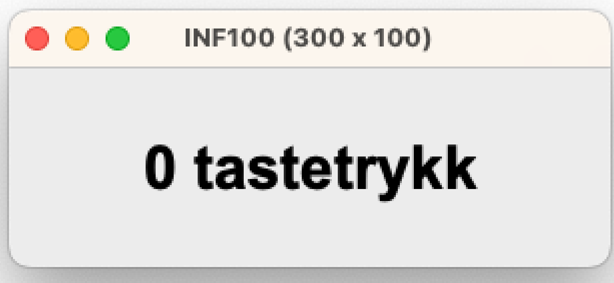
Et *grafisk brukergrensesnitt* er et dataprogram som lar brukeren interagere med programmet ved å klikke på knapper, benytte tastaturet, flytte på musen eller andre handlinger uten å primært forholde seg til terminalen.

> *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.*

Første eksempel: tell antall tastetrykk

```
from uib_inf100_graphics.event_app import run_app
```

```python
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)
```

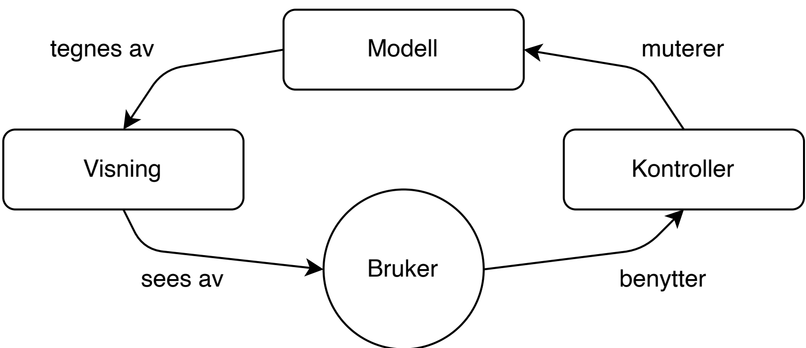


Model-View-Controller

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.

- **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.

```
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)
```

Kopier

```
# 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)
```

Kopier

```
# 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)
```

Kopier

```
# 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)
```

Kopier

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)
```

Kopier

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)
```

Kopier

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.

```
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)
```

Kopier

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.

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!

```
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)
```

Kopier

Brudd med MVC

Vi kan ikke endre modellen i `redraw_all`.

```
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)
```

Kopier

```
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)
```

Kopier

```
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)
```

Kopier

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_
```

Bilder

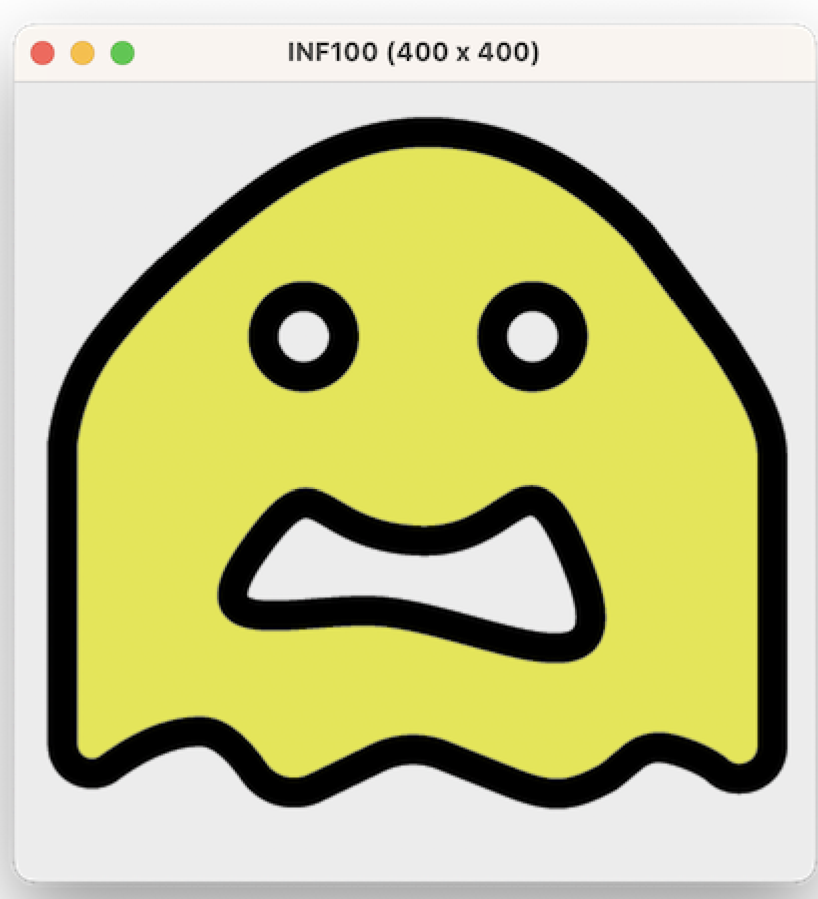
Du kan benytte alle de samme metodene og hjelpefunksjonene for å laste og vise bilder som vi kjenner fra kursnotatene om grafikk, *men*:

- **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.

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
```

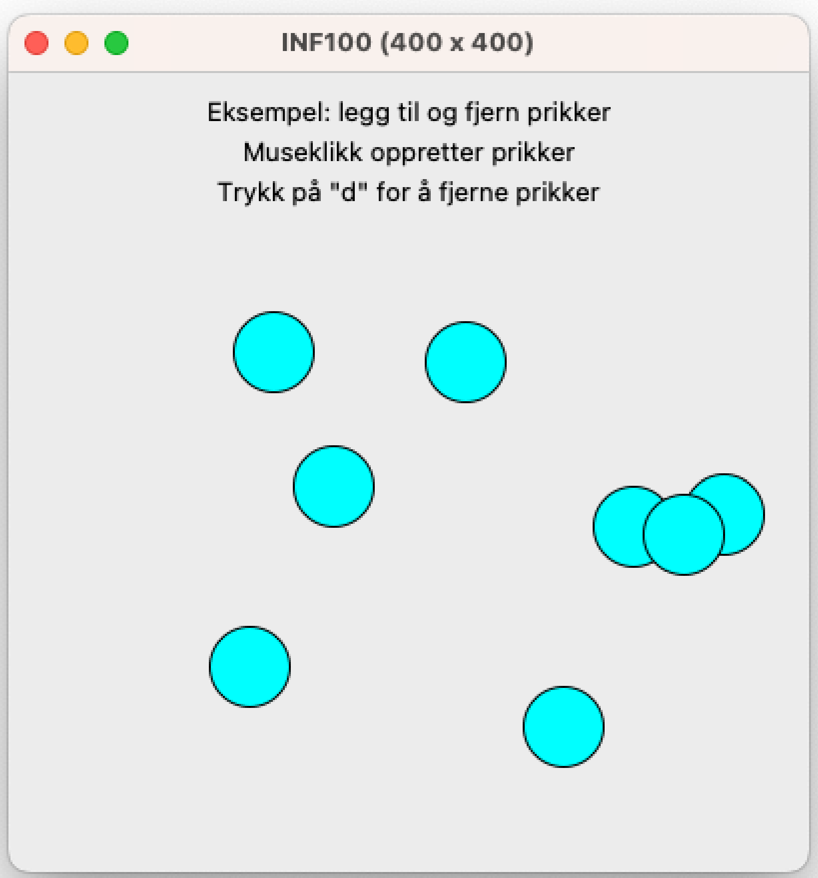
```python
def redraw_all(app, canvas):
    image = app.image_yellow if app.use_yellow_image else app.image_black
    canvas.create_image(
        app.width / 2,
        app.height / 2,
        pil_image=image,
    )

run_app(width=400, height=400)
```



Eksempel: legg til og fjern prikker

```python
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)
```

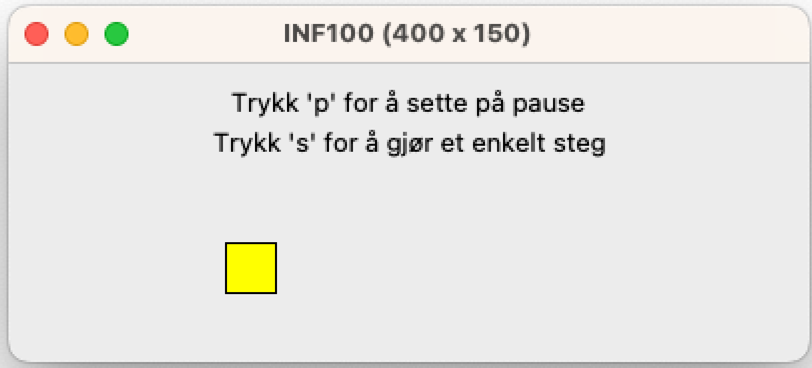


Eksempel: sprettende figur

```
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)
```

```python
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(
        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)
```

Kopier

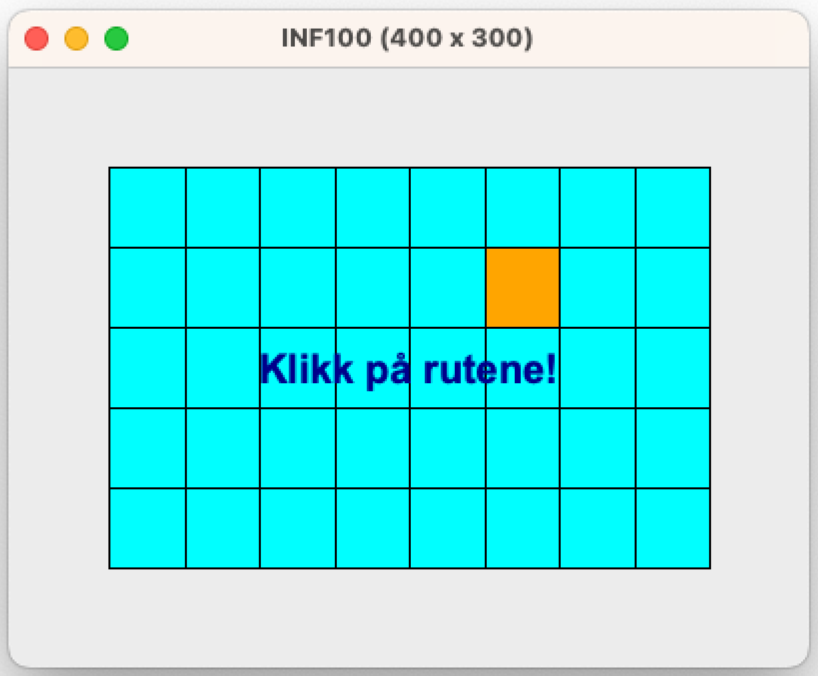


Eksempel: museklikk i rutenett

```
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)
```

```python
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
    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)
```

Kopier

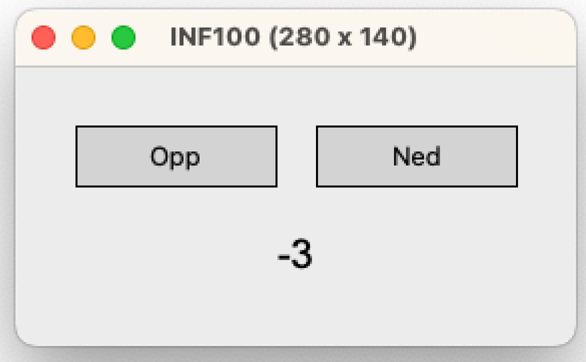


Eksempel: knapper

```
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):
    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)
```

Kopier


INF100 (280 x 140)
Opp
Ned
-3

Filer og CSV

- Skrive til og lese fra fil
- Hjelp, filen blir ikke funnet
- Enkel CSV -håndtering

Skrive til og lese fra fil

Vi kan åpne en fil ved å bruke syntaksen `with ... as ...` sammen med funksjonen `open`. Denne 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 metoden `write` på filobjektet, mens for å lese fra filen bruker vi metoden `read` på filobjektet.

```
# Skrive til en fil
with open('minfil.txt', 'w', encoding='utf-8') as filobjekt:
    filobjekt.write('Hei, verden!')

# Lese fra en fil
with open('minfil.txt', 'r', encoding='utf-8') as filobjekt:
    innhold = filobjekt.read()
print(innhold) # Skriver ut 'Hei, verden!'
```

Kopier

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.). Vi vil ikke bruke denne modusen i dette emnet.*

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.

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:

```
import os
cwd = os.getcwd()
print(cwd)
```

Kopier

Denne mappen blir bestemt av programmet som *starter* python. F. eks. 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

Eksempel: finner ikke filen som skal leses

Det er *mulig* å programmatisk endre cwd til å bli samme mappe som filen som kjøres ligger i:

```
import os
directory_of_current_file = os.path.dirname(__file__)
os.chdir(directory_of_current_file) # endrer cwd
```

Kopier

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 hver verdi representerer en kolonne i tabellen.

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 biblioteker i Python som er spesielt laget for å lese CSV-filer, 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.

```python
############################################
### LESE INPUT OG OPPRETTE DATASTRUKTUR ###
############################################

with open('people.csv', 'r', encoding='utf-8') as file_object:
    # content_string er streng som inneholder hele innholdet i filen
    content_string = file_object.read()

# .strip fjerner whitespace på begynnelsen og slutten av strengen
content_string = content_string.strip()

# .split('\n') klipper opp strengen ved linjeskift, og gir oss en
# liste med bitene som er igjen
content_lines = content_string.split('\n')

# 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
    values = line.split(',')
    table.append(values)
```

```python
# Vi kan nå aksessere enkeltverdier i tabellen vår ved å bruke
# indeksering på samme måte som vi gjør med andre 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 til int


############################
### PRESENTER RESULTATET ###
############################

# Skriv den endrede tabellen til en ny fil
with open('people_a_year_later.csv', 'w', encoding='utf-8') as file_object:
    for row in table:
        # Konverterer alle cellene i listen til strenger
        string_row = [str(x) for x in row]

        # .join() limer sammen strenger i en liste til én stor streng
        # med den gitte limestrengen mellom hver av dem
        line = ','.join(string_row)
        file_object.write(line + '\n')
```

Kopier

Unicode

- Unicode og ordinaler
- Tekstkoding

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 (nå) 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)
```

```python
print('Ordinal til', symbol, 'er' , ordinal) # Ordinal til A er 65
```

Kopier Se steg Kjør

For å konvertere fra ordinal til symbol («character») kan vi bruke funksjonen `chr`:

```python
ordinal = 97
symbol = chr(ordinal)
print('Ordinal', ordinal, 'har symbol', symbol) # Ordinalen 97 har symbol a
```

Kopier Se steg Kjør

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

UTF-32

UTF-8

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 95% sannsynlighet er dette UTF-8, men av og til kan det være noe annet. Hvis du ikke vet, kan du prøve å gjette deg frem. Hvis du får rare tegn i stedet for norske bokstaver, er det sannsynligvis fordi du har gjettet feil. Da må du prøve å gjette på en annen koding.

Hvis du ikke vet hvilken koding som er brukt, prøv disse ekodingene først: *UTF-8* (anbefalt), *Windows-1252* (også kalt *cp1252*), *ISO 8859-1* (også kalt *Latin-1*), *Windows-1251* (også kalt *cp1251*), *UTF-16* og *UTF-32*. Hvis teksten er på et spesielt språk kan du også søke på internett etter kodinger som er vanlige for det språket.

```python
# Eksempel på å skrive til en fil
writing_text = "Dette er en tekst. Den har æøå og 🐍 i seg."
with open("myfile.txt", "w", encoding="utf-8") as f:
    f.write(writing_text)

# Eksempel på å lese fra en fil
with open("myfile.txt", "r", encoding="utf-8") as f:
    reading_text = f.read()
print(reading_text) # Dette er en tekst. Den har æøå og 🐍 i seg.
```

Kopier

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 operativsystem.

Håndtere krasj

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")
```

Kopier | Se steg | Kjør

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.
```

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 uførerer 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")
```

Kopier Se steg Kjør

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))
```

Kopier Se steg Kjør

Oppslagsverk

- Basics
- Enkle eksempler
- Opprette oppslagsverk
- Egenskaper ved oppslagsverk
- Operatorer, funksjoner og metoder
- Løkker over oppslagsverk

Se også offisiell dokumentasjon 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
```

Kopier Se steg Kjør

```
# 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
```

Kopier Se steg Kjør

Enkle eksempler

```
# 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')
```

Kopier Se steg Kjør

```
# 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)
```

Kopier

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}
```

Kopier Se steg Kjør

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])
```

Kopier Se steg Kjør

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 }
```

Kopier Se steg Kjør

Rekkefølge betyr egentlig ingenting; men ved iterasjon er det den rekkefølgen nøklene ble først opprettet som teller.

```
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
```

Kopier Se steg Kjør

Et oppslagsverk kan *muteres*.

```
d1 = { 'foo': 42 }
d2 = d1

d2['foo'] = 95
print(d1['foo']) # 95
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

Oppslagsverk er *svært* effektive.

```
# 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")
```

Kopier Se steg Kjør

Operatorer, funksjoner og metoder

Funksjoner og operatorer

```
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
```

Kopier Se steg Kjør

Metoder for å mutere oppslagsverk

```
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
```

Kopier Se steg Kjør

Løkker over oppslagsverk

```
d = {'foo': 42, 'bar': 25, 'baz': 95}
```

```python
# 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()
```

Kopier Se steg Kjør

Mengder

- Enkelt eksempel
- Opprette mengder
- Egenskaper ved mengder
- Operasjoner på mengder
- Frozenset

Se også offisiell dokumentasjon for `set`.

Enkelt eksempel

En *mengde* (engelsk: set) er en datastruktur som kan holde mange verdier, men uten at det finnes noen rekkefølge på verdiene. Verdiene har ingen indeks/posisjon, slik de har i en liste. 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
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
s.discard(3)
```

```
print(s)                    # {2, 5, 6}
```

Kopier Se steg Kjør

Opprette mengder

```
# Opprett en tom mengde
s = set()
print(s)
```

Kopier Se steg Kjør

```
# PS: MISLYKKET forsøk på å opprette en tom mengde:
s = {} # dette oppretter et oppslagsverk, ikke en mengde!
print(type(s))
```

Kopier Se steg Kjør

```
# Opprett en mengde statisk (med verdier angitt direkte i kildekoden)
s = {2, 3, 5}
print(s)
```

Kopier Se steg Kjør

```
# Opprett en mengde fra en liste (eller annen samling med elementer)
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'}
```

Kopier Se steg Kjør

```
# Mengdeinklusjon.
# Vi tar utgangspunkt i en annen samling (her a), og bruker deretter
# benytte en inklusjons-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'}
```

Kopier Se steg Kjør

Egenskaper ved mengder

Elementene i en mengde har ingen (meningsfull/forutsigbar) rekkefølge.

```
s = set()
s.add(2)
s.add(44)
s.add(11)
s.add(5)
s.add(33)

for e in s:
    print(e, end=' ') # Rekkefølge kan være ulik fra maskin til maskin
print()

print({2, 3, 5} == {5, 3, 2}) # True
```

Kopier Se steg Kjør

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
```

Kopier Se steg Kjør

Mengder kan muteres.

```
s = {2, 3, 5}
alias = s

s.add(9)
print(s) # {2, 3, 5, 9}
print(alias) # {2, 3, 5, 9}
```

Kopier Se steg Kjør

Elementene i en mengde må *ikke* være mulig å mutere.[1]

```python
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)
```

Kopier | Se steg | Kjør

Mengder er *svært* effektive.

```python
# 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')
```

Kopier | Se steg | Kjør

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.

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}
```

Kopier | Se steg | Kjør

```
# 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}
```

Kopier | Se steg | Kjør

Fjerne elementer (remove/discard/difference/ – /mengdeinklusjon)

```
# 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







# Mengdeinklusjon 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}
```

Kopier Se steg Kjør Kopier Se steg Kjør

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}
```

Kopier Se steg Kjør Kopier Se steg Kjør

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}
```

Kopier

```
# 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)
```

```python
# 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}
```

Kopier Se steg Kjør

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).

```python
# 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
```

Kopier

---

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

- 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)
```

Kopier Se steg Kjør

```
# Import med nytt navn
```

```python
# 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)
```

Kopier Se steg Kjør

```python
# 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
```

Kopier Se steg Kjør

```python
# 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))
```

Kopier Se steg Kjør

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:

```python
# foo.py
def hello():
    return "Hello there"
```

Kopier

```python
# bar.py
import foo
print(foo.hello())
```

Kopier

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())
```

Kopier

eller

```
# bar.py
from lib import foo
print(foo.hello())
```

Kopier

Hovedfil og modul

Når en modul importeres, kjøres all koden som er i filen. For eksempel,

```
# foo.py
def hello():
    return "Hello there"

print("Tester hello i foo:", hello())
```

Kopier

```
# bar.py
from foo import hello # Oops, skriver ut 'Tester hello i foo: Hello there'
print(hello())
```

Kopier

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__"`:

```
# foo.py
```

```python
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())
```

Kopier

Standardbiblioteket

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 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
- json

Interaksjon med operativsystemet og filstrukturen

- sys
- os og shutil

---

math

https://docs.python.org/3/library/math.html

```
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)
```

Kopier Se steg Kjør

random

https://docs.python.org/3/library/random.html

```
import random

def demonstrate_random():
    y = random.random()                # Et tilfeldig flyttall mellom 0 og 1
    x1 = random.choice(range(10))      # En tilfeldig int mellom 0 og 9
    x2 = random.choice(range(10))      # En tilfeldig int mellom 0 og 9
    x3 = random.choice(range(10))      # En tilfeldig int mellom 0 og 9
    s = random.choice(["a", "b", "c"]) # Tilfeldig element fra en liste
    print(s, x1, x2, x3, y)

print("Før kall til seed:")
demonstrate_random()

# MERK: RESTEN AV DETTE EKSEMPELET FUNGERER DÅRLIG FOR PYTHON I NETTLESEREN,
# MÅ DERFOR TESTES LOKALT PÅ DIN MASKIN

# Tilfeldige tall er ikke egentlig tilfeldige, men blir generert med
# utgangspunkt i klokkeslettet når funksjonen blir kalt. Vi kan velge
# å i stedet ta utgangspunkt i en verdi vi selv bestemmer, med 'seed'.
random.seed(42) # Vi velger selv verdien av 42

# Forsøk å kjøre koden gjentatte ganger lokalt på din maskin og se hva
# forskjellen blir før og etter kallet til seed-metoden.
print("Etter kall til seed(42):")
demonstrate_random()
```

Kopier Se steg Kjør

time

https://docs.python.org/3/library/time.html

```
import time

print(time.time()) # Antall sekunder siden 1. januar 1970 som flyttall
```

Kopier | Se steg | Kjør

datetime

https://docs.python.org/3/library/datetime.html

```
from datetime import datetime, timedelta

# Et datetime -objekt representerer et bestemt tidspunkt
lecture_starts = datetime(2022, 10, 24, 14, 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 = }")
```

Kopier | Se steg | Kjør

CSV

https://docs.python.org/3/library/csv.html

```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)
```

```python
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)
```

Kopier

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.

```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()
```

```python
# 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)
```

Kopier Se steg

sys

https://docs.python.org/3/library/sys.html

```python
import sys

# Avslutt python umiddelbart
sys.exit()
print("Vi kommer aldri hit")
```

Kopier

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.

```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)
```

```python
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")
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!")
```

Kopier

Eksterne pakker

- Enkel installasjon
- Installasjon med pip

Eksempler på eksterne pakker

- Requests: last ned ting fra internett
- Matplotlib: visualisering av av data
- Andre vanlige pakker

Enkel installasjon (fungerer av og til)

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

libs = ['matplotlib', 'numpy', 'pandas', 'requests']
cmd = f"{sys.executable} -m pip install --user {' '.join(libs)}"

ans = input(f"\n\nType 'yes' to try direct install of {libs}: ")
if ans == "yes":
    from subprocess import run
    run(cmd.split())
else:
    print("copy this line into the terminal:\n")
    print(cmd)
    print()
```

Kopier

Installasjon med pip

For å installere eksterne pakker og moduler, bruker vi et program som heter pip. 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>
```

(det er også mulig å skrive `<pip-sti> install <pakkenavn>`, men den metoden dekker vi ikke her). 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».

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 = {
    # Noen nettsider krever at 'User-Agent' har fått en verdi
    # for at man skal få respons.
    'User-Agent': 'inf100.ii.uib.no abc123', # Noe som forteller hvem du er
}

webpage = requests.get(url, headers=headers)
```

```
print(webpage.content) # Innholdet på nettsiden før dekoding: 'byte-streng'

webpage_content = webpage.content.decode('utf-8')
print(webpage_content) # Etter dekoding: en vanlig streng
```

Kopier

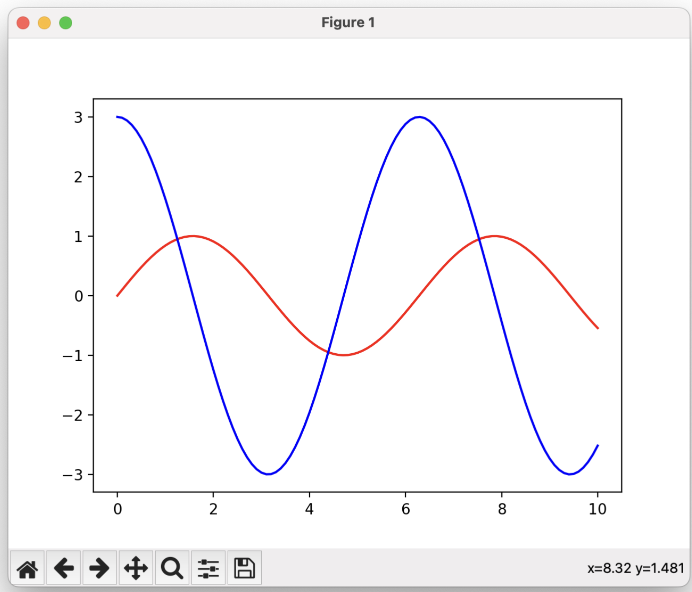
Matplotlib: visualisering av data

https://matplotlib.org/

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.

```
import matplotlib.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()
```

Kopier



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.

Pyplot tutorial

An introduction to the pyplot interface. Please also see Quick start guide for an overview of how Matplotlib works and Matplotlib Application Interfaces (APIs) for an explanation of the trade-offs between the supported user APIs.

Introduction to pyplot

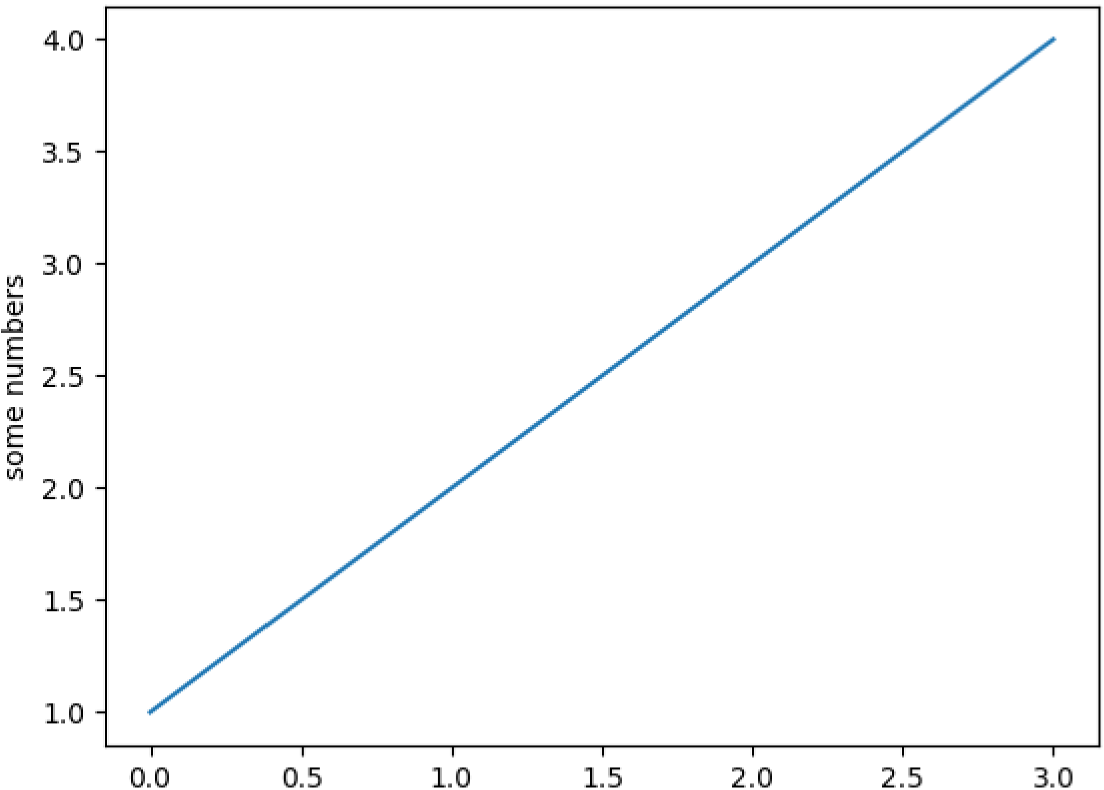
`matplotlib.pyplot` is a collection of functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the documentation refers to the *axes* part of a figure and not the strict mathematical term for more than one axis).

The implicit pyplot API is generally less verbose but also not as flexible as the explicit API. Most of the function calls you see here can also be called as methods from an `Axes` object. We recommend browsing the tutorials and examples to see how this works. See Matplotlib Application Interfaces (APIs) for an explanation of the trade-off of the supported user APIs.

Generating visualizations with pyplot is very quick:

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

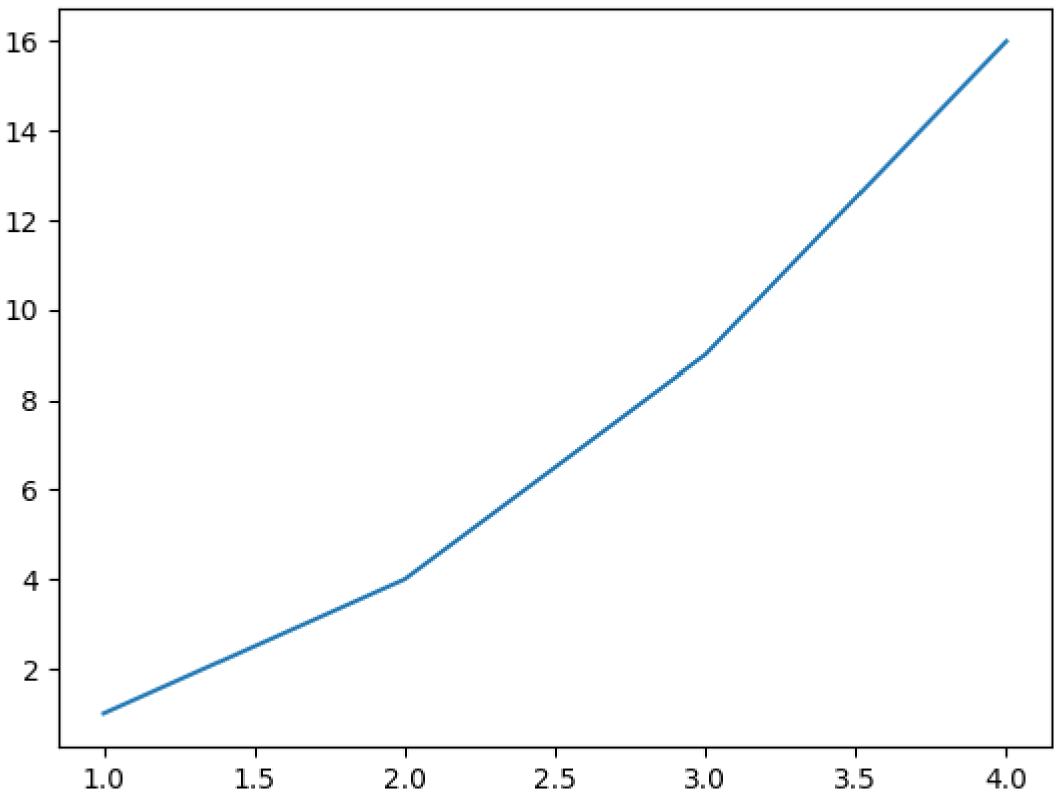


You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to `plot`, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0; therefore, the x data are `[0, 1, 2, 3]`.

`plot` is a versatile function, and will take an arbitrary number of arguments. For example, to plot x versus y, you can write:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

Skip to main content

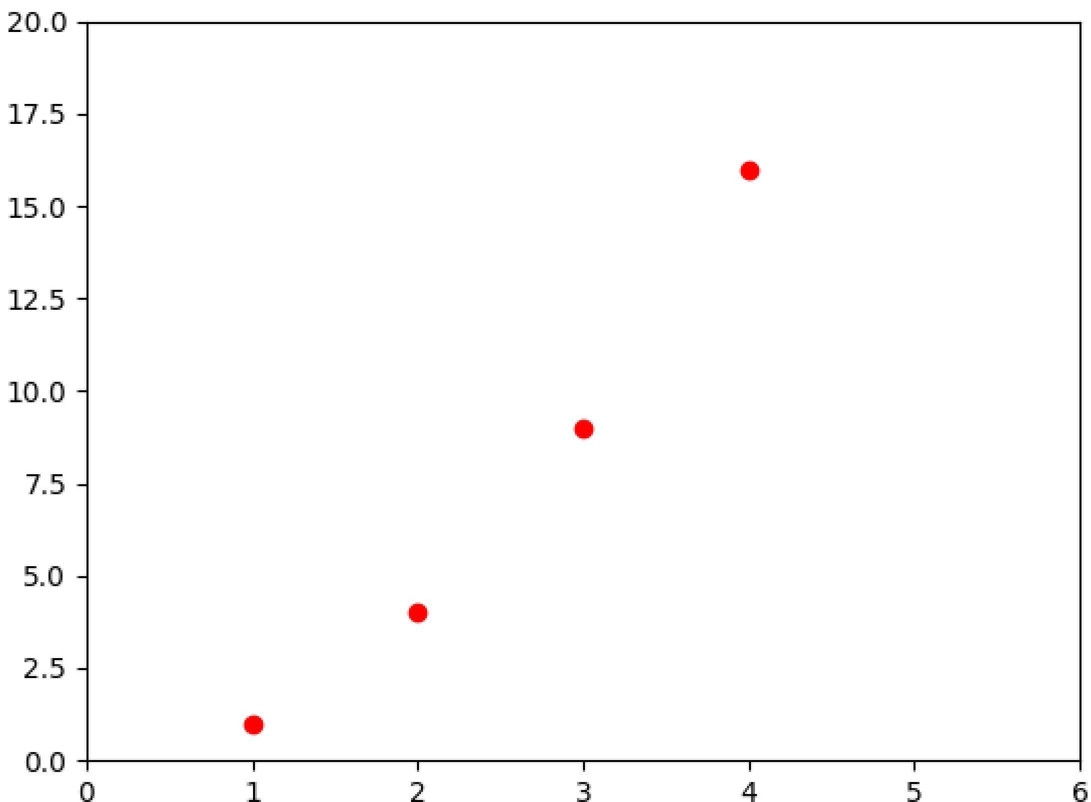


Formatting the style of your plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis((0, 6, 0, 20))
plt.show()
```

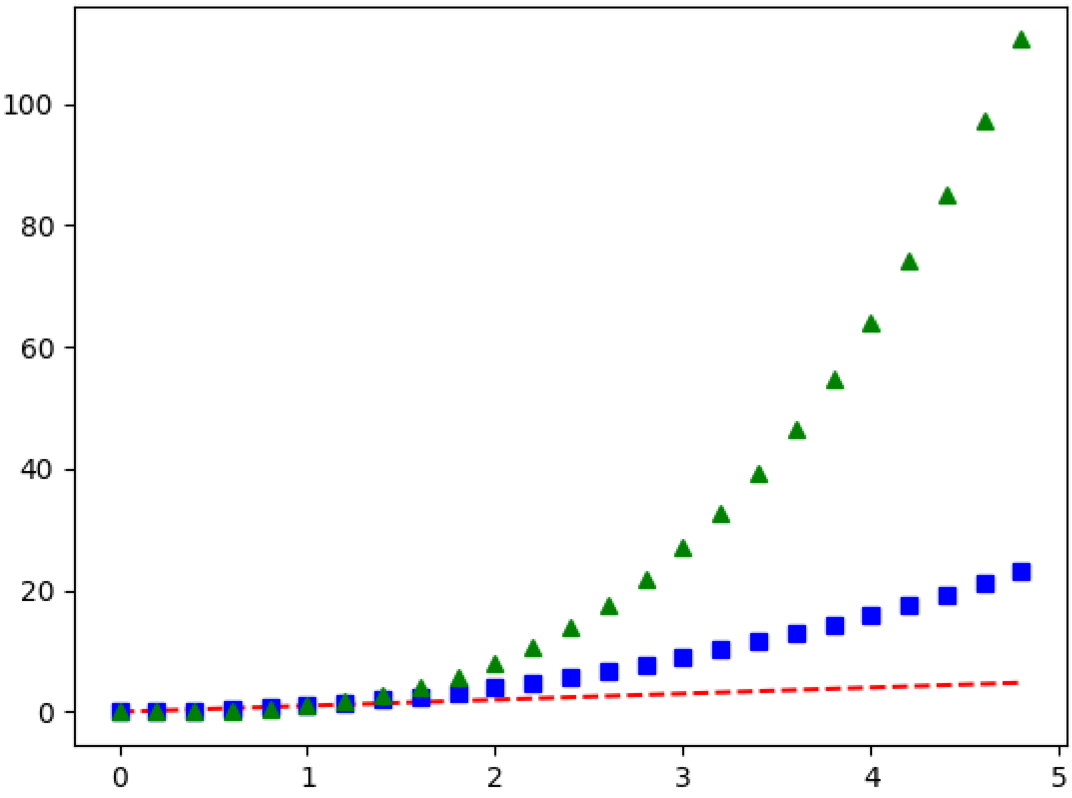
Skip to main content



See the `plot` documentation for a complete list of line styles and format strings. The `axis` function in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one function call using arrays.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



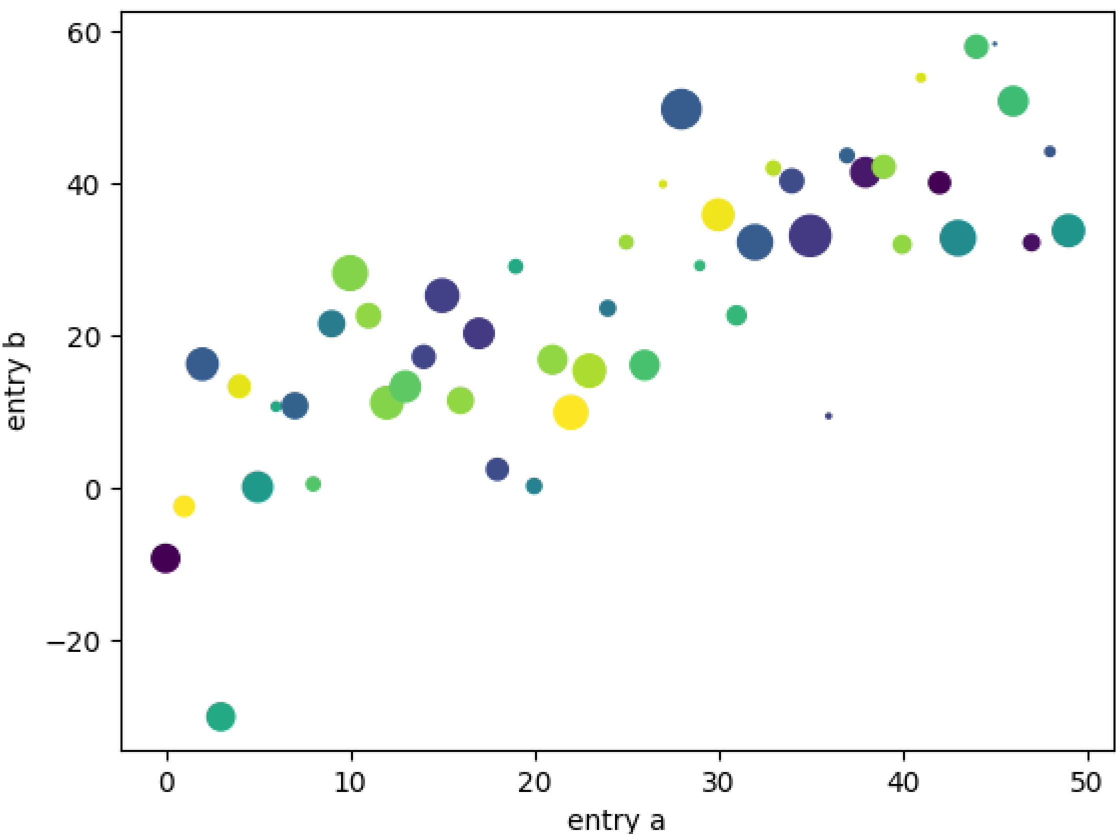
Plotting with keyword strings

There are some instances where you have data in a format that lets you access particular variables with strings. For example, with structured arrays or `pandas.DataFrame`.

Matplotlib allows you to provide such an object with the `data` keyword argument. If provided, then you may generate plots with the strings corresponding to these variables.

```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```

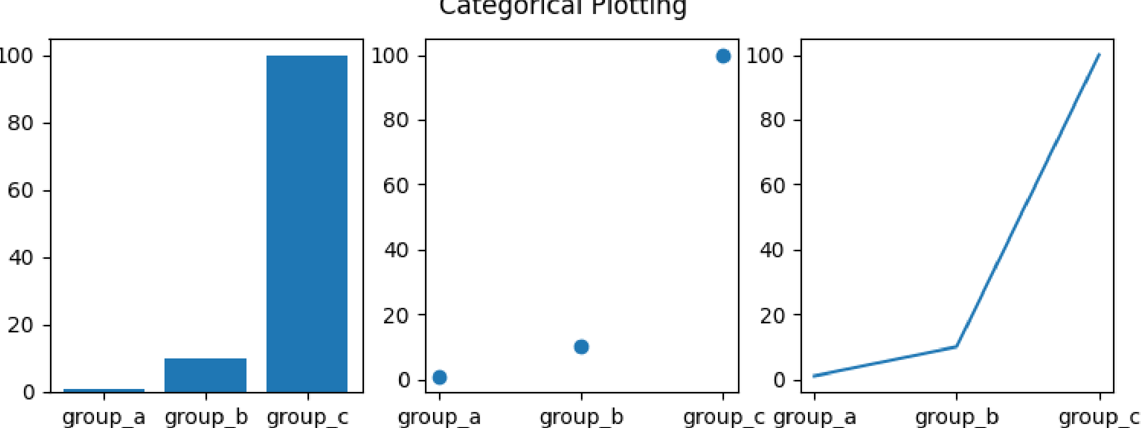
Skip to main content



Plotting with categorical variables

It is also possible to create a plot using categorical variables. Matplotlib allows you to pass categorical variables directly to many plotting functions. For example:

```
names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()
```



Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see `matplotlib.lines.Line2D`. There are several ways to set line properties

- Use keyword arguments:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a `Line2D` instance. `plot` returns a list of `Line2D` objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line,` to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialiasing
```

- Use `setp`. The example below uses a MATLAB-style function to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword arguments
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available `Line2D` properties.

Skip to main content

Property	Value Type
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	[`'butt'` | `'round'` | `'projecting'`]
dash_joinstyle	[`'miter'` | `'round'` | `'bevel'`]
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	[`'-'` | `'--'` | `'-.'` | `':'` | `'steps'` | ...]
linewidth or lw	float value in points
marker	[`'+'` | `','` | `'.'` | `'1'` | `'2'` | `'3'` | `'4'`]
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None | integer | (startind, stride)]

Property	Value Type
pickradius	the line pick selection radius
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True | False]
xdata	np.array
ydata	np.array
zorder	any number

To get a list of settable line properties, call the `setp` function with a line or lines as argument

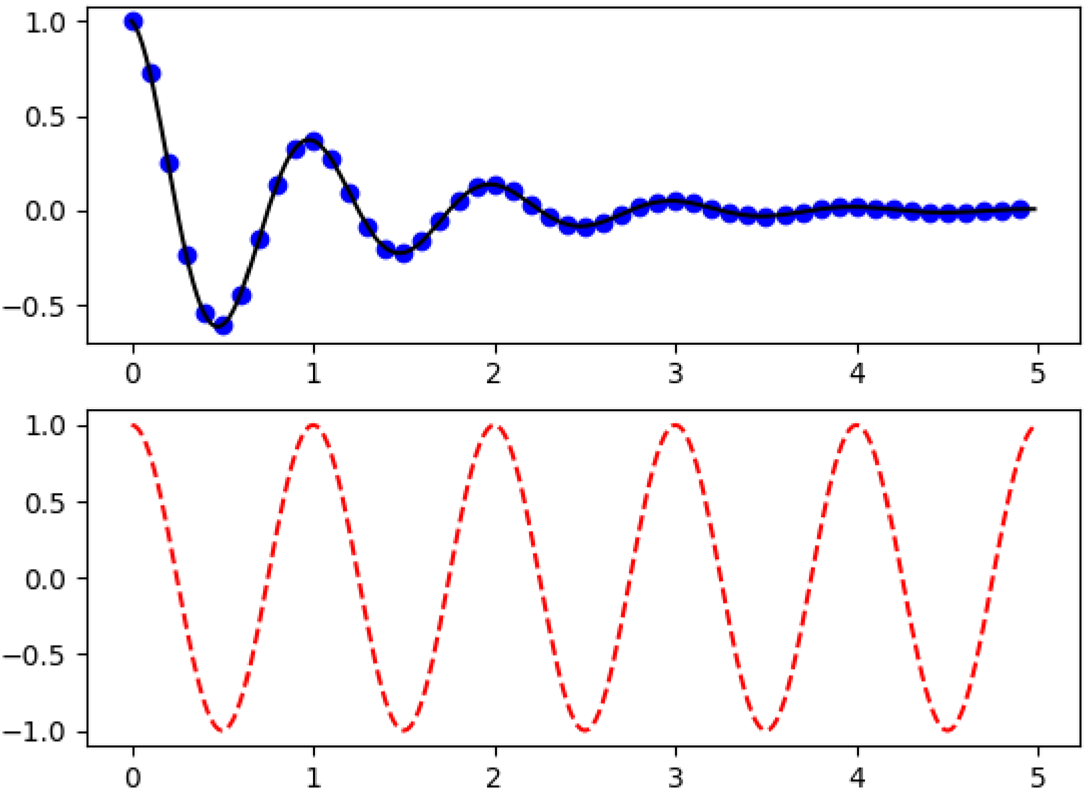
```
In [69]: lines = plt.plot([1, 2, 3])

In [70]: plt.setp(lines)
  alpha: float
  animated: [True | False]
  antialiased or aa: [True | False]
  ...snip
```

Working with multiple figures and axes

MATLAB, and `pyplot`, have the concept of the current figure and the current axes. All plotting functions apply to the current axes. The function `gca` returns the current axes (a `matplotlib.axes.Axes` instance), and `gcf` returns the current figure (a `matplotlib.figure.Figure` instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
```

```
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



The `figure` call here is optional because a figure will be created if none exists, just as an Axes will be created (equivalent to an explicit `subplot()` call) if none exists. The `subplot` call specifies `numrows, numcols, plot_number` where `plot_number` ranges from 1 to `numrows*numcols`. The commas in the `subplot` call are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`.

You can create an arbitrary number of subplots and axes. If you want to place an Axes manually, i.e., not on a rectangular grid, use `axes`, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See Axes Demo for an example of placing axes manually and Multiple subplots for an example with lots of subplots.

You can create multiple figures by using multiple `figure` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
```

Skip to main content

```
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot() by default

plt.figure(1)                # first figure current;
                             # subplot(212) still current
plt.subplot(211)             # make subplot(211) in the first figure
                             # current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

You can clear the current figure with `clf` and the current axes with `cla`. If you find it annoying that states (specifically the current image, figure and axes) are being maintained for you behind the scenes, don't despair: this is just a thin stateful wrapper around an object-oriented API, which you can use instead (see Artist tutorial)

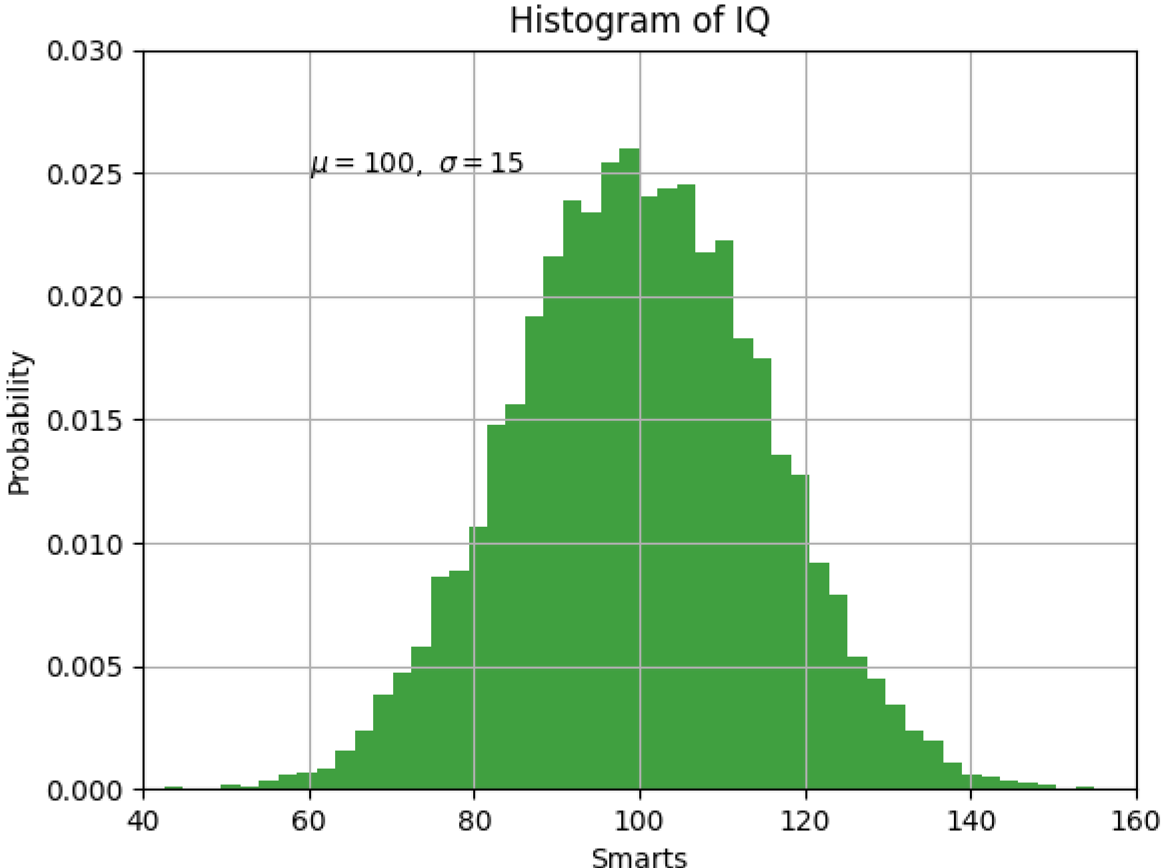
If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with `close`. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until `close` is called.

Working with text

`text` can be used to add text in an arbitrary location, and `xlabel`, `ylabel` and `title` are used to add text in the indicated locations (see Text in Matplotlib Plots for a more detailed example)

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=True, facecolor='g', alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Skip to main content



All of the `text` functions return a `matplotlib.text.Text` instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp`:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in Text properties and layout.

Using mathematical expressions in text

Matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

```
plt.title(r'$\sigma_i=15$')
```

The `r` preceding the title string is important -- it signifies that the string is a *raw* string and not to treat backslashes as python escapes. matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts -- for details see Writing mathematical expressions. Thus, you can use mathematical text across platforms without requiring a TeX

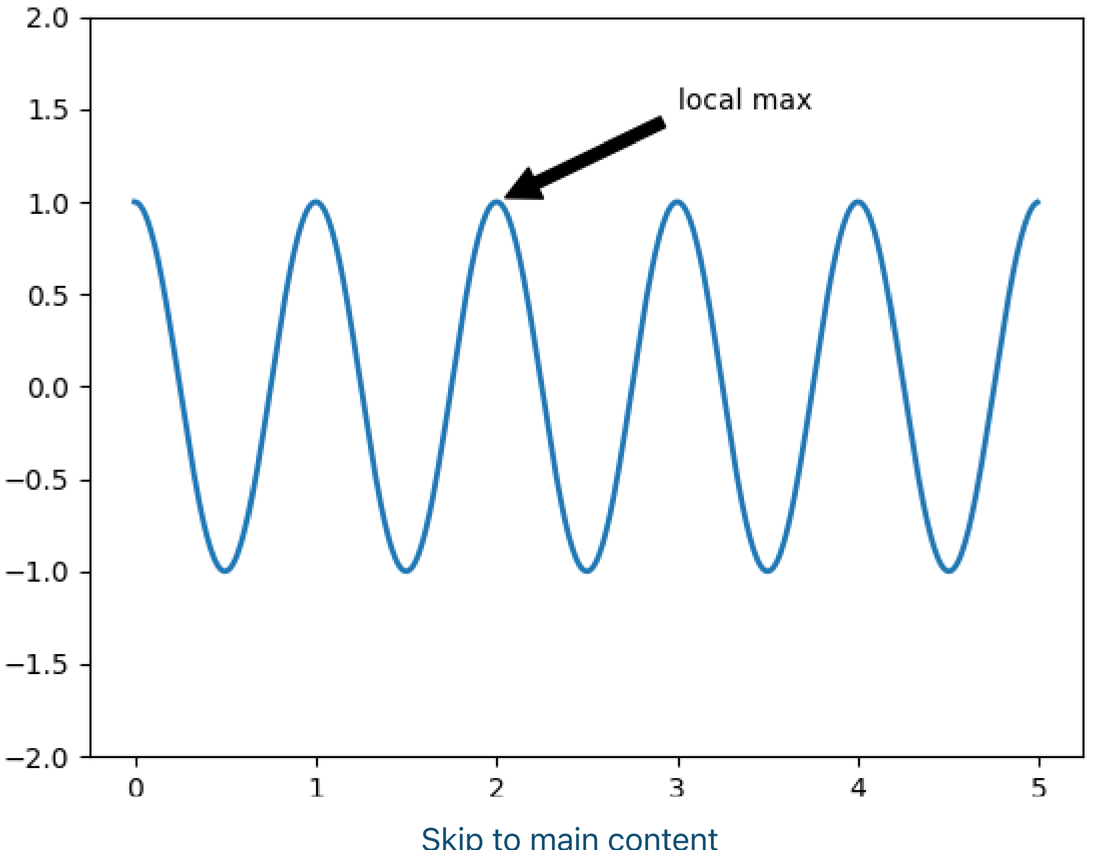
Skip to main content

your text and incorporate the output directly into your display figures or saved postscript -- see Text rendering with LaTeX.

Annotating text

The uses of the basic `text` function above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the `annotate` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x, y)` tuples.

```
ax = plt.subplot()

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )

plt.ylim(-2, 2)
plt.show()
```



Skip to main content

In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose -- see Basic annotation and Advanced annotation for details. More examples can be found in Annotating Plots.

Logarithmic and other nonlinear axes

`matplotlib.pyplot` supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

```
plt.xscale('log')
```

An example of four plots with the same data and different scales for the y-axis is shown below.

```
# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the open interval (0, 1)
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure()

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
```

© Copyright 2002–2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012–2023 The Matplotlib development team.

Created using Sphinx 7.2.6.

Built from v3.8.1-4-ge9df1f1e2d.

Built with the PyData Sphinx Theme 0.13.3.