12 GUI programmering med Tkinter

Læringsmål dette kapittel:

<< kommer >>

Man kan spørre seg om GUI programmering som tema har noe å gjøre i en «generell» lærebok om programmering.

Jeg vil si absolutt ja!

GUI-programmering gir oss mulighet til å lage programmer som er mer intuitive, interaktive og visuelt tiltalende. Det gir en umiddelbar følelse av mestring når man kan klikke på knapper, skrive inn tekst i felt og se hvordan programmet reagerer. Dette er ikke bare motiverende – det er også en naturlig vei inn i mer avanserte konsepter som **hendelsesorientert programmering** og **callback-funksjoner**.

Samtidig skal en ikke bruke for mye tid på å grave seg ned i detaljer som går på fininnstilling av farger, tykkelse på rammer og andre visuelle aspekter.

Vi vil heller bruke tid på de viktige mekanismene, som hendelseshåndtering, callback funksjoner, flytting av data mellom GUI komponenter, generell design med layout managere samt enkle simuleringer.

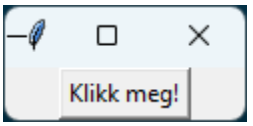
Hva er hendelsesorientert programmering?

I GUI-programmering er det ikke programmet som styrer flyten, men ***brukeren***. Programmet "venter" på at brukeren skal gjøre noe – som å klikke på en knapp eller skrive inn tekst – og reagerer deretter. Dette kalles **hendelsesorientert programmering**, og det krever en annen måte å tenke på enn tradisjonell sekvensiell programmering

12.1 Det første GUI-programmet med Tkinter

La oss starte med et enkelt program – et vindu med en knapp som får en tekst til å poppe opp et sted. Dette programmet kan vi senere bygge videre på med flere elementer og funksjonalitet.

Programmet vil se slik ut:




```
 # file: sc_12_01_button.py
 1: import tkinter as tk
 2:
 3: def on_button_click():
 4:     print("Knappen ble trykket!")
 5:
 6: root = tk.Tk()
 7: root.title("Callback-eksempel")
 8:
 9: bt_simple= tk.Button(root, text="Klikk meg!", command=
on_button_click)
10: bt_simple.pack()
11:
12: root.mainloop()
```

Forklaring til koden, og legg spesielt merke til linje 9:
Linje 1
Her importeres Tkinter-biblioteket, som gir tilgang til GUI-funksjonalitet.

Linje 3–4
Definerer en callback-funksjon *som skal kalles når knappen trykkes*. Merk at vi bruker en vanlig `print()` kommando for å skrive teksten `"Knappen ble trykket!"`. Dette betyr at denne teksten blir skrevet ut i terminalvinduet i VSCode, *ikke* i et GUI grensesnitt.

Linje 6
Oppretter hovedvinduet.

Linje 7
Setter tittelen på vinduet.

Linje 9
Lager en knapp med teksten "Klikk meg!":

- Første parameter `root` forteller hvilket vindu som eier denne button
- Andre *keyword parameter* `text` er teksten som framkommer på button
- Tredje parameter, også en keyword parameter `command` kobler button til *callback-funksjonen* `on_button_click`

Argumentet `command= on_button_click` sier til Tkinter:
"Når denne knappen trykkes, skal funksjonen `on_button_click` kalles."

Men merk:

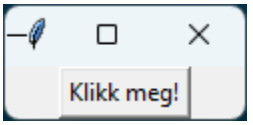
- Vi skriver `on_button_click` uten parenteser.
- Hadde vi skrevet `command=on_button_click()`, ville funksjonen blitt **kalt med én gang** når programmet starter – og ikke når knappen trykkes.

Ved å skrive bare *navnet* på funksjonen, uten (), gir vi Tkinter ***adressen* til funksjonen** – altså en referanse – slik at den kan kalles **senere**, når hendelsen faktisk skjer.
Dette er et klassisk eksempel på **callback-mekanismen** i hendelsesorientert programmering.

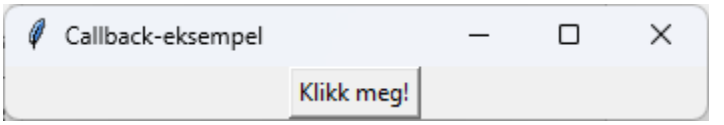
Linje 10
Plasserer knappen i vinduet ved hjelp av `pack()`, som fører til at knappen organiserer seg i foreldrevinduet fra toppen og i midten.
Mer om pack() og layout managere om litt.

Linje 12
Starter Tkinter sin hendelsesløkke (mainloop()), som overvåker hendelseskøen og kaller riktige callbacks.

Resultatet blir dette:



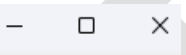
Eller, hvis vi forstørrer vinduet:



Tkinter og hendelseskøen

Tkinter bygger på et underliggende system kalt **event loop** (hendelsesløkke), som kontinuerlig overvåker en **hendelseskø**. Denne køen inneholder informasjon om brukerens interaksjoner med GUI-en – som museklikk, tastetrykk, vindusendringer, og andre hendelser.

Når en hendelse oppstår, legges den i køen. Tkinter sjekker så denne køen i en løkke (som startes med mainloop()), og når den finner en hendelse, ser den etter om det finnes en **callback-funksjon** som er registrert for den typen hendelse. Hvis det gjør det, kalles denne funksjonen automatisk. Noen hendelser håndteres uten at vi behøver å programmere hva som skal skje; - eksempler på dette er at vi har «dratt» i vinduet sånn at det har blitt større, eller dersom vi bruker minimaliserin, gjennopprett eller Avslutt valgene oppe i vinduets ramme:



I koden over, på linje 9 har vi koblet TRYKK hendelsen på knappen «Klikk meg» til funksjonen `knapp_trykket()`, ***så følgende blir skrevet i terminalvinduet i VSCode*** når vi trykker knappen:

Knappen ble trykket!

12.2Grunnleggende om GUI-elementer i Tkinter

12.2.1Vinduet

Når vi skriver `root = tk.Tk()` (linje 6), oppretter vi et hovedvindu – et tomt rammeverk der vi kan plassere ulike grafiske elementer.

Dette vinduet er en instans av klassen `Tk`, og fungerer som "roten" i GUI-programmet.

12.2.2 Widgets

En *widget* er et grafisk element i et vindu – som en knapp, tekstfelt, etikett (label), rullefelt osv. I linje 9 oppretter vi en Button-widget, som er en interaktiv komponent brukeren kan klikke på.

Her er en oversikt over de enkleste widgets. Merk at det kan være nyttig å prefikse variabelnavnene med to bokstaver som gjør at du lett kan se hvilken type widgeten er.

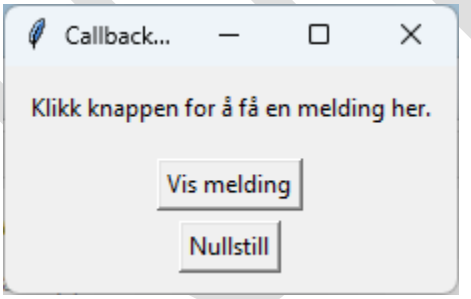
Widget	**Beskrivelse**	**Typisk bruk**	**Prefiks**
Label	Viser tekst i vinduet. Ikke interaktiv.	Meldinger, status, instruksjoner	`lb_`
Button	En trykkbar knapp som kan kobles til en funksjon via command.	Utføre handlinger, trigge callbacks	`bt_`
Entry	Et enkelt tekstfelt der brukeren kan skrive inn én linje tekst.	Input av navn, tall, e-post osv.	`en_`
Text	Et større tekstområde for flere linjer.	Skrive lengre tekst, vise logg	`tx_`
Frame	En beholder for å gruppere widgets.	Strukturere layout, dele opp GUI	`fr_`
Checkbutton	En avkrysningsboks som kan være av eller på.	Valg som kan aktiveres/deaktiveres	`cb_`

Radiobutton	En knapp i en gruppe der bare én kan være aktiv.	Velge én av flere alternativer	`rb_`
Listbox	Viser en liste med elementer som brukeren kan velge fra.	Valg fra liste, visning av data	`lbx_`
Scale	En skyveknapp for å velge en numerisk verdi.	Justering av volum, størrelse osv.	`sc_`
Canvas	Et tegneområde for grafikk, figurer og bilder.	Visualisering, tegning, animasjon	`cv_`

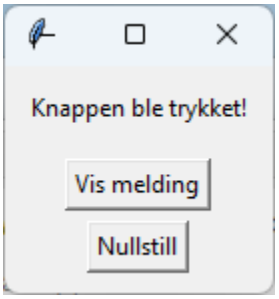
La oss utvikle programmet vårt slik at vi får skrevet ut teksten i en label i hovedvinduet. En label er et område hvor vi kan ha tekst, men bruker kan ikke skrive noe inn i den. Foreløpig har vi ikke behov for annet enn å få våre widgets, dvs knapper og labels under hverandre, så vi trenger hverken frames eller avanserte geometry managere.

Neste program har to knapper og en label.

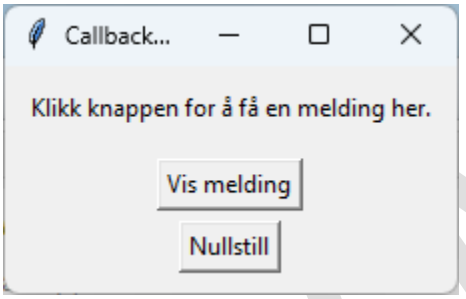
Det ser slik ut ved oppstart:



..og etter at vi har trykket «Vis melding»:



Et trykk på «Nullstill» knappen får oss tilbake til utgangspunktet:



Merk at det er flere måter å skrive en tekst inn i en label (og andre lignende widgets), i koden under er en av metodene demonstrert:

```
# file: sc_12_02_button_and_label.py
01: import tkinter as tk

03: # Hovedvindu
04: root = tk.Tk()
05: root.title("Callback + Nullstill")

07: # Standardtekst for label
08: INSTRUKSJON = "Klikk knappen for å få en melding her."

10: # En Label hvor vi viser meldinger til brukeren.
11: lb_demo = tk.Label(root, text=INSTRUKSJON)
12: lb_demo.pack(padx=10, pady=10)

14: def show_message():
15:     lb_demo.config(text="Knappen ble trykket!")

17: def reset_message():
18:     lb_demo.config(text=INSTRUKSJON)

21: # Knapp for å vise meldingen
```

```
22: bt_show = tk.Button(root, text="Vis melding",
command=show_message)
23: bt_show.pack(padx=5, pady=5)

25: # Knapp for å nullstille meldingen
26: bt_reset = tk.Button(root, text="Nullstill",
command=reset_message)
27: bt_reset.pack(padx=5, pady=(0, 10))

29: root.mainloop()
```

Forklaring til koden:

Linje 1 – 3:
Importerer `tkinter` som `tk`, og oppretter hovedvinduet med `tk.Tk()`.
`root.title(...)` setter tittelen i vindusrammen.

Linje 7 – 8:
Definerer en konstant tekst som skal vises i etiketten (Label) ved oppstart.

Linje 10 – 12:
Oppretter en Label-widget (`lb_demo`) som viser instruksjonsteksten.
`pack(padx=10, pady=10)` gir **luft rundt widgeten**:

- `padx`: horisontal padding (venstre og høyre)
- `pady`: vertikal padding (over og under)

Linje 14 – 15:
Funksjonen `show_message()` endrer teksten i `lb_demo` når knappen trykkes.
`config(...)` brukes til å **endre egenskaper** ved en eksisterende widget – her endres `text`.

Linje 17 – 18:
Funksjonen `reset_message()` setter teksten tilbake til den opprinnelige instruksjonen.

Linje 21 – 23:
Oppretter en knapp (`bt_show`) som kaller `show_message()` når den trykkes.
`pack(padx=5, pady=5)` gir litt luft rundt knappen.

Linje 25 – 27:
Oppretter en knapp (`bt_reset`) som kaller `reset_message()` når den trykkes.
`pady=(0, 10)` gir **ingen plass over**, men **10 piksler under** knappen – nyttig for å justere avstand mot bunnen av vinduet.

Linje 29:
Starter GUI-programmets hovedløkke. Programmet venter nå på brukerinteraksjon.

12.2.3 Nytten av keyword arguments

Mange av funksjonene vi bruker i Tkinter har veldig mange argumenter. Slik kan det se ut hvis vi ikke utnytter keyword arguments:

```
# Uten keyword arguments (må vite rekkefølgen, 8 argumenter):
label = tk.Label(root, "Hei", None, None, "blue", "white", None,
("Arial", 12))
```

Dette er:

- Vanskelig å lese
- Lett å gjøre feil
- Krever at du kjenner hele parameterlisten

Samme kall hvor vi bruker keyword arguments:

```
# med keyword, kun 5 argumenter
label = tk.Label(root, text="Hei", fg="blue", bg="white",
font=("Arial", 12))
```

Dette er:

- Lett å lese
- Lett å vedlikeholde
- Fleksibelt – du kan endre én ting uten å røre resten

12.2.4 config() metoden

`config()`-metoden i Tkinter er en svært nyttig måte å **endre egenskaper** ved en widget etter at den er opprettet. Den gir deg

fleksibilitet til å oppdatere utseende eller funksjonalitet dynamisk, uten å måtte lage en ny widget.

Når vi skriver

```
lb_demo.config(text="Knappen ble trykket!")
```

...så endrer vi **innholdet** (teksten) som vises i Label-widgeten `lb_demo`.

12.2.4.1 Generelt om config()

`config()` er en metode som finnes på alle Tkinter-widgets.
Den kan brukes til å **lese eller endre** widgetens egenskaper (også kalt *options*).

Du kan endre flere egenskaper samtidig, for eksempel:

```
lb_demo.config(text="Ny tekst", fg="blue", bg="yellow")
```

Typiske egenskaper du kan endre med config():

Egenskap	**Forklaring**
`text`	Teksten som vises i widgeten
`fg`	Tekstfarge (foreground)
`bg`	Bakgrunnsfarge
`font`	Skrifttype og størrelse, f.eks. ("Arial", 12)
`width`	Bredde på widgeten (i tegn eller piksler, avhengig av type)
`height`	Høyde på widgeten
`padx`	Horisontal padding rundt widgeten
`pady`	Vertikal padding rundt widgeten
`command`	Funksjon som skal kalles ved interaksjon (f.eks. knappetrykk)

`state`	Aktiv/inaktiv ("normal" eller "disabled")
`anchor`	Plassering av innholdet i widgeten (f.eks. "w" for venstre)
`justify`	Justering av tekst (venstre, høyre, senter) i f.eks. Label
`relief`	Kantstil: "flat", "raised", "sunken", "groove", "ridge"

Alternativet til å bruke `config()` er å bruke widgetens konstruktør, dette er selvsagt den foretrukne måten dersom du vet hvordan widgeten skal se ut og den ikke skal endres.

12.2.4.2 Et alternativ til config; - indeksere ved hjelp av keyword

I Tkinter er hver widget bygget opp med en intern **option dictionary** – en slags oppslagstabell der hver egenskap (som `text`, `fg`, `bg` osv.) har en tilhørende verdi. Denne kan du både lese fra og skrive til ved hjelp av **indeksering**, akkurat som med en vanlig Python-dictionary.

Her et kodeeksempel som viser begge metoder i bruk:


```
01: import tkinter as tk

03: root = tk.Tk()
04: root.title("To måter å endre widgets")

06: # Label 1 – endres med config()
07: label_config = tk.Label(root, text="Label med config",
fg="black")
08: label_config.pack(pady=5)
```

```
10: # Label 2 – endres med indeks
11: label_index = tk.Label(root, text="Label med indeks",
fg="black")
12: label_index.pack(pady=5)

14: # Endre Label 1 med config()
15: label_config.config(text="Endret med config()", fg="blue")

17: # Endre Label 2 med option dictionary (indeks)
18: label_index["text"] = "Endret med indeks"
19: label_index["fg"] = "green"

21: root.mainloop()
```

Forklaring til kode:

Linje 14–15:
Endrer `label_config` ved hjelp av `config()` – metoden oppdaterer flere egenskaper samtidig.

Linje 17–19:
Endrer `label_index` ved å bruke **option dictionary** direkte, altså ved å indeksere med nøkkelordene `"text"` og `"fg"`.

12.2.5 Hvordan finner vi ut hvilke egenskaper en widget har?

Det er nyttig å vite hvilke egenskaper (options) en bestemt widget har – både for å kunne sette dem ved opprettelse og for å kunne endre dem senere med `config()` eller indeksbasert tilgang.

Tkinter-widgeter har som nevnt en intern **option dictionary** som inneholder alle tilgjengelige egenskaper, med tilhørende verdier. Dette gir oss mulighet til å hente ut informasjon om widgetens konfigurasjon på en systematisk måte.

12.2.5.1 keys()

Metoden `keys()` viser alle opsjoner for en bestemt widget.

I REPL kan vi kjøre den slik:

```
>>> tk.Label().keys()
```

```
['activebackground', 'activeforeground', 'anchor',
'background', 'bd', 'bg', 'bitmap', 'borderwidth',
'compound', 'cursor', 'disabledforeground', 'fg',
'font', 'foreground', 'height', 'highlightbackground',
'highlightcolor', 'highlightthickness', 'image',
'justify', 'padx', 'pady', 'relief', 'state',
'takefocus', 'text', 'textvariable', 'underline',
'width', 'wraplength']
>>>
```

12.2.5.2 configure()

Hvis du ønsker å se både **standardverdier og nåværende verdier**, kan du bruke configure():

```
tk.Label().configure()
```

Dette gir oss en dictionary der hver nøkkel er en egenskap, og verdien er en tuple med informasjon om:

- option-navn
- option-type
- option-dokumentasjon
- standardverdi
- nåværende verdi

12.3 Strukturering av GUI med Frame og geometry-managere

Når vi lager grafiske brukergrensesnitt med Tkinter, blir det raskt behov for å **organisere widgets** på en ryddig og oversiktlig måte. Her kommer `Frame`-widgeten inn som et viktig verktøy.

En `Frame` fungerer som en **beholder** for andre widgets. Den lar deg gruppere elementer som hører sammen, og gir deg bedre kontroll over layout og struktur. Dette er spesielt nyttig når du har flere seksjoner i et vindu – for eksempel en toppmeny, et innholdsområde og en bunnlinje.

12.3.1 Hvorfor bruke Frame?

- **Modularisering:** Du kan utvikle og teste deler av GUI-en separat.
- **Layout-kontroll:** Hver `Frame` kan ha sin egen geometry-manager (`pack`, `grid`, `place`).
- **Lesbarhet:** Koden blir lettere å forstå når widgets er gruppert logisk.
- **Gjenbruk:** Du kan lage egne klasser basert på `Frame` for mer avanserte komponenter.

12.3.2 Geometry-managere – en naturlig del av Frame-bruk

Når du bruker `Frame`, må du også ta stilling til **hvordan widgets skal plasseres** inni den. Dette gjøres med en **geometry-manager**.

Tkinter har tre hovedtyper:

Geometry-manager	Beskrivelse
`pack()`	Enkel og intuitiv. Plasserer widgets i rekkefølge (vertikalt eller horisontalt).
`grid()`	Plasserer widgets i rader og kolonner – egnet for skjemaer og tabeller.
`place()`	Absolutt plassering med koordinater – brukes sjelden, men gir full kontroll.

Du kan bruke forskjellige layout-managere i ulike `Frame`-seksjoner, noe som gir stor fleksibilitet.

<< eksempel på bruk av frame og grid >>

12.4 Mer om hendelseshåndtering

Hittil har vi sett på enkel hendelseshåndtering; - et trykk på en button som får tekst skrevet ut et sted.

Vi må også kunne håndtere andre typer hendelser, som mus-klikk, tasteklikk, valg av element i listboks, comboboks etc.

Dette gjøres som vi har sett ved å koble funksjoner (callback-funksjoner) til hendelser. I Tkinter er det to hovedmåter å gjøre dette på:

- Ved å bruke `command`-parameteren (gjelder enkelte widgets)
- Ved å bruke `bind()`-metoden (kan brukes på alle widgets)

I `Tkinter` er det vanligst å bruke `command` for å koble en callback funksjon til en *knapp og andre enkle widgets*.

`bind`-metoden brukes gjerne når du vil håndtere mer spesifikke hendelser, eller når du trenger informasjon om selve hendelsen (for eksempel musekoordinater).

`bind` gir deg større fleksibilitet, men for enkle knappetrykk er `command` ofte enklest.

12.4.1command uten parametre

Følgende widgets har `command` opsjonen:

`Button` - vanlige trykknapper
`Checkbutton` - avkryssingsbokser
`Radiobutton` - radioknapper
`Scale` - glidebrytere
`Scrollbar` - rullefelt
`Menu/Menubutton` - menyer og menyknapper

Som vi allerede har sett, `command` opsjonen kan brukes til å kalle en callback på en enkel måte, se eksempelet `sc_12_01_button.py` beskrevet tidligere.

12.4.2command ***med*** parametre;- lambda

Vi kan *sende parametre* til funksjonen ved hjelp av et såkalt *lambda uttrykk*:

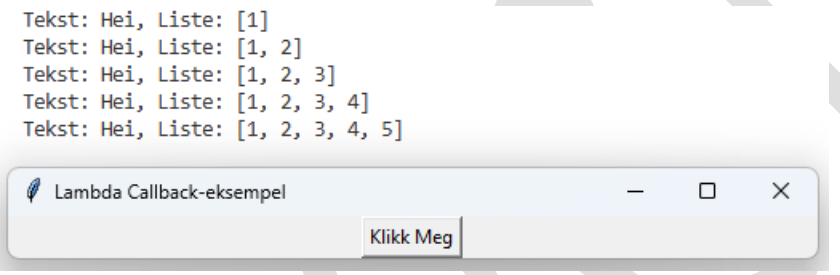
```
from tkinter import *

def on_button_click(text, a_list):
    print(f"Tekst: {text}, Liste: {a_list}")
```

```
    a_list.append(a_list[-1] + 1)

window = Tk()
some_list = [1]
button = Button(window, text="Klikk Meg", command=lambda:
on_button_click("Hei", some_list))
button.pack()
window.mainloop()
```

Resultatet blir: (linjene med tekst skrives ut i terminalvinduet, vi bruker kjedelig nok `print()` setningen for å få kodeeksempelet så kort som mulig...):



12.4.2.1 Foreløpig forklaring av lambda:

Lambda er en kortform for å lage små, anonyme funksjoner på én linje. I stedet for å skrive en hel funksjon med `def`, kan du lage en "mini-funksjon" der og da. Det skal *ikke* være en `return` setning i et lambda uttrykk; - det som står etter kolon returneres automatisk, f eks adressen til en funksjon.

Grunnleggende struktur av lambda:
`lambda parametere: hva_som_skal_skje`

I vårt eksempel:
`command=lambda: on_button_click("Hei", some_list)`

Hva skjer?

1. `lambda`: - lager en funksjon uten parametere (tom før kolonet)

2. `on_button_click("Hei", some_list)` - det som skjer når funksjonen kalles, adressen til denne funksjonen er resultatet av uttrykket

Parameteroverføring med *closure*:

- Lambda bruker noe som heter **closure** - den "fanger" og husker verdiene `"Hei"` og `some_list` fra det ytre scopet
- Selv når lambda-funksjonen kalles senere (når knappen trykkes), husker den fortsatt disse verdiene
- Dette kalles en *closure* fordi funksjonen "lukker seg rundt" variablene den trenger

Sammenligning:

```
# Dette fungerer IKKE:
command=on_button_click("Hei", some_list)  # Kaller
funksjonen med en gang!

# Dette fungerer:
command=lambda: on_button_click("Hei", some_list)  # Lager
en closure som husker verdiene
```

Analogi:

Lambda lager en "tidskapsel" som husker alt den trenger for å kalle funksjonen senere. Når knappen trykkes, åpnes tidskapselen og verdiene brukes (dette er selvsagt en for lettvint forklaring, vi kommer tilbake til lambda og closures).

12.4.3 `bind()` for å kalle hendelseslytter

Den generelle syntaksen for `bind()` i Tkinter er:

```
widget.bind(hendelse, callback)
```

widget: Referanse til widgeten du vil knytte hendelsen til (f.eks. en knapp).

hendelse: En streng som beskriver hendelsen, for eksempel `"<Button-1>"` for venstre museklikk, `"<Key>"` for tastetrykk, osv.

callback: Funksjonen som skal kalles når hendelsen skjer. Denne funksjonen må ta minst én parameter (event-objektet). Funksjonsnavnet kan erstattes av et lambda uttrykk.

Eksempel:

```
bt_simple.bind("<Button-1>", on_button_click)
```

Vanlige hendelser i Tkinter er:

`"<Button-1>"`: Venstre museknapp trykket
`"<Button-2>"`: Midtre museknapp trykket
`"<Button-3>"`: Høyre museknapp trykket
`"<Double-Button-1>"`: Dobbelklikk med venstre museknapp
`"<Enter>"`: Musen går inn i widgeten
`"<Leave>"`: Musen går ut av widgeten
`"<Key>"`: En tast trykkes ned
`"<KeyPress-a>"`: Bokstaven 'a' trykkes ned

Det finnes også ***virtuelle*** hendelser som skrives med doble vinkelparanteser:

`"<<Copy>>"`: Kopier-kommando (f.eks. Ctrl+C)

`"<<Paste>>"`: Lim inn-kommando (f.eks. Ctrl+V)

`"<<Cut>>"`: Klipp ut-kommando

Virtuelle hendelser brukes ofte for å lage egne hendelser eller for å reagere på standard handlinger i Tkinter.

Eksempel:

```
widget.bind("<<Copy>>", on_copy)
```

Da vil funksjonen `on_copy` kalles når brukeren kopierer tekst i widgeten.

Her er kode som bruker `bind()` (i stedet for `command`) for å koble til callback metoden , tilsvarer bruk av command i `sc_12_01_button.py`:

```
# file: sc_12_07_button_and_bind.py
01: import tkinter as tk
02:
03: def on_button_click(event):
04:     print(f"Museklikk koordinater: x={event.x}, y={event.y}")

05:
06: root = tk.Tk()
07: root.title("Bind-eksempel")
08:
09: bt_simple = tk.Button(root, text="Klikk meg!")
10: bt_simple.pack()
11:
12: # Bruker bind i stedet for command
13: bt_simple.bind("<Button-1>", on_button_click)
```

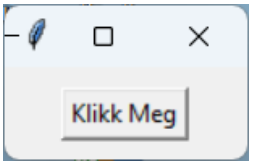
Forklaring til koden:

Linje 3:
Definerer en callback-funksjon `on_button_click`. Den tar et imot et `event`-argument, som sendes til funksjonen når vi bruker `bind()`

Linje 13:
`bind()` får adressen til funksjonen. Når brukeren klikker med venstre museknapp (<Button-1>), vil Tkinter automatisk kalle `on_button_click(event)` ***og sende med et `event`-objekt***.
Vi bruker vi event objektet til å skrive ut mus- koordinatene der knappen ble trykket.

12.4.4 Bind + parameteroverføring med lambda

Eksempelet under viser hvordan vi bruker et lambda uttrykk til både å sende egendefinerte argumenter, samt `event` objektet til callback funksjonen. Utskriften går til terminalvinduet, så vi slipper å lage GUI'en for komplisert.




```
 1: # file: sc_12_08_button_and_bind_lambda.py
 2: import tkinter as tk
 3:
 4: def on_button_click(text, a_list, event):
 5:     print(f"Event: {event}")
 6:     print(f"Tekst: {text}, Liste: {a_list}")
 7:     print(f"Widget som ble klikket: {event.widget}")
 8:     print(f"Museklikk koordinater: x={event.x}, y={event.y}")
 9:     a_list.append(a_list[-1] + 1)
10:     print("---")
11:
12: root = tk.Tk()
13: root.title("Bind + Lambda Callback-eksempel")
14:
15: some_list = [1]
16:
17: # Bruk bind i stedet for command
18: bt_simple = tk.Button(root, text="Klikk Meg")
19: bt_simple.bind("<Button-1>", lambda event:
on_button_click("Hei fra bind", some_list, event))
20: bt_simple.pack(pady=10)
21:
22: root.mainloop()
```

Linje 4 – 10:
Her lages en callback som tar imot tre parametre: en tekst, en liste, og et `event` objekt. Fra `event` objektet skriver vi ut diverse informasjon, se utskriften nedenfor. Callback metoden er kalt ved hjelp av et lambda uttrykk, som har sendt med en tekst og en liste i tillegg til event objektet.

Linje 19
Her bruker vi `bind` metoden til å knytte hendelsen `"<Button-1>"` (venstre mus-tast trykket) til trykk-knappen `bt_simple`. Lambda uttrykket vil sørge for å sende *tre* parametre til callback metoden.

Utskrift i terminalvinduet (etter to trykk på «Klikk meg» knappen)**:**

```
Event: <ButtonPress event num=1 x=15 y=10>
Tekst: Hei fra bind, Liste: [1]
Widget som ble klikket: .!button
Museklikk koordinater: x=15, y=10
---
Event: <ButtonPress event num=1 x=17 y=11>
Tekst: Hei fra bind, Liste: [1, 2]
Widget som ble klikket: .!button
Museklikk koordinater: x=17, y=11
```

Widget navnet «`.!button`» er et autogenerert navn.

12.4.5 Event objektet

Når vi bruker `bind()` i Tkinter for å koble en hendelse til en callback funksjon, blir et **event-objekt** automatisk sendt som argument til funksjonen. Dette objektet inneholder detaljer om hendelsen som fant sted, og gir oss mulighet til å reagere mer presist.

Her er noen av de mest brukte attributtene, spesielt ved museklikk og tastetrykk:

Attributt	**Beskrivelse**
`event.widget`	Referanse til widgeten som utløste hendelsen.
`event.x, event.y`	Koordinater for museklikket relativt til widgeten.
`event.x_root, event.y_root`	Koordinater for museklikket relativt til skjermen.
`event.char`	Tegnet som ble tastet inn (ved tastetrykk).
`event.keysym`	Symbolsk navn på tast (f.eks. "Return", "Escape").
`event.keycode`	Numerisk kode for tastetrykket.
`event.type`	Type hendelse (f.eks. ButtonPress, KeyPress).

`event.num`	Hvilken museknapp som ble trykket (1 = venstre, 2 = midten, 3 = høyre).
`event.state`	Status for modifikatortaster (Shift, Ctrl, etc.).

Innholdet i `event`-objektet **avhenger av både hendelsestypen og widgeten**.

For eksempel:

- En Button-widget som mottar et <Button-1>-klikk gir deg musekoordinater og hvilken knapp som ble trykket.
- En Entry-widget som mottar <KeyPress> gir deg informasjon om hvilken tast som ble trykket.
- En Canvas-widget kan gi oss mer avansert informasjon, som hvilke grafiske elementer som ble truffet. Se kodeeeksempel under delkapittel om Canvas.

12.5 Canvas

Canvas er en kraftig widget som lar oss tegne grafiske elementer direkte i brukergrensesnittet. Den fungerer som et *tomt lerret* der vi kan plassere figurer, tekst og bilder, og gir oss full kontroll over layout og interaktivitet.

Metode	**Beskrivelse**
`create_line()`	Tegner en linje mellom to eller flere punkter.
`create_rectangle()`	Tegner et rektangel definert av to hjørnepunkter.
`create_oval()`	Tegner en sirkel eller ellipse innenfor et rektangel.

`create_polygon()`	Tegner en figur med flere hjørner (f.eks. trekant, femkant).
`create_arc()`	Tegner en buet linje eller sektor.
`create_text()`	Viser tekst på et bestemt punkt.
`create_image()`	Viser et bilde (f.eks. PNG eller GIF).

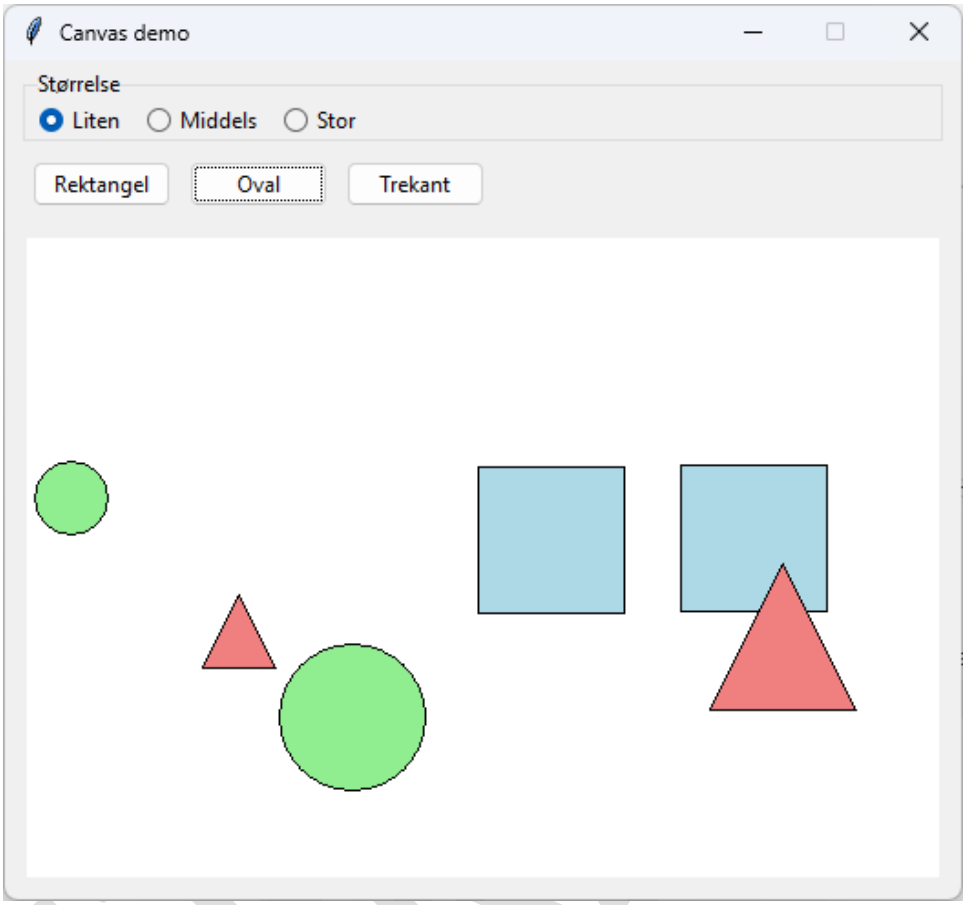
Hver figur får en unik ID som kan brukes til å endre, flytte eller slette den senere.

12.5.1 Canvas og mus-hendelser

Mus-hendelser kan knyttes til alle typer widgets, og vi får ofte behov for å detektere slike i forbindelse med bruk av Canvas.

Programmet nedenfor lar brukeren tegne rektangler, ovaler og trekanter i et canvas, med valg for størrelse (liten, middels, stor). Hver gang bruker trykker på en av figur-knappene, tegnes en ny figur på en tilfeldig plass innenfor canvaset.

Bruker kan slette figurer ved å bruke klikk–dra–slipp (drag) rundt figurene som skal slettes.



Se forklaring etter koden.

```
 1: # file: sc_12_09_canvas_demo.py
 2: import tkinter as tk
 3: from tkinter import ttk
 4: import random
 5:
 6: class CanvasDemo:
 7:     def __init__(self, root):
 8:         self.root = root
 9:         self.canvas_width = 500
10:         self.canvas_height = 350
11:         self.root.title("Canvas demo")
12:         self.root.resizable(False, False)
13:
14:         # Størrelsesvalg
15:         self.size_var = tk.StringVar(value="Middels")
16:         size_frame = ttk.LabelFrame(root, text="Størrelse")
```

```
 17:         size_frame.pack(padx=10, pady=5, fill="x")
 18:         rb_liten = ttk.Radiobutton(size_frame, text="Liten",
variable=self.size_var, value="Liten")
 19:         rb_middels = ttk.Radiobutton(size_frame,
text="Middels", variable=self.size_var, value="Middels")
 20:         rb_stor = ttk.Radiobutton(size_frame, text="Stor",
variable=self.size_var, value="Stor")
 21:         rb_liten.pack(side="left", padx=5)
 22:         rb_middels.pack(side="left", padx=5)
 23:         rb_stor.pack(side="left", padx=5)
 24:
 25:         # Knapperamme
 26:         button_frame = ttk.Frame(root)
 27:         button_frame.pack(padx=10, pady=5, fill="x")
 28:         bt_rektangel = ttk.Button(button_frame,
text="Rektangel", command=self.tegn_rektangel)
 29:         bt_oval = ttk.Button(button_frame, text="Oval",
command=self.tegn_oval)
 30:         bt_trekant = ttk.Button(button_frame,
text="Trekant", command=self.tegn_trekant)
 31:         bt_rektangel.pack(side="left", padx=5)
 32:         bt_oval.pack(side="left", padx=5)
 33:         bt_trekant.pack(side="left", padx=5)
 34:
 35:         # Canvas
 36:         self.canvas = tk.Canvas(root,
width=self.canvas_width, height=self.canvas_height, bg="white")
 37:         self.canvas.pack(padx=10, pady=10)
 38:
 39:         # For markering og sletting
 40:         self._select_rect = None
 41:         self._start_x = None
 42:         self._start_y = None
 43:
 44:         self.canvas.bind("<Button-1>",
self._on_canvas_click)
 45:         self.canvas.bind("<B1-Motion>",
self._on_canvas_drag)
 46:         self.canvas.bind("<ButtonRelease-1>",
self._on_canvas_release)
 47:
 48:     def _on_canvas_click(self, event):
 49:         # Start alltid markeringsrektangel ved klikk
 50:         self._start_x = event.x
 51:         self._start_y = event.y
 52:         self._select_rect =
self.canvas.create_rectangle(event.x, event.y, event.x, event.y,
outline="red", dash=(2,2))
```

```
 53:
 54:     def _on_canvas_drag(self, event):
 55:         # Oppdater markeringsrektangelet under drag
 56:         if self._select_rect is not None:
 57:             self.canvas.coords(self._select_rect,
self._start_x, self._start_y, event.x, event.y)
 58:
 59:     def _on_canvas_release(self, event):
 60:         # Slett alle figurer innenfor markeringsrektangelet
 61:         if self._select_rect is not None:
 62:             x0, y0, x1, y1 =
self.canvas.coords(self._select_rect)
 63:             x_min, x_max = min(x0, x1), max(x0, x1)
 64:             y_min, y_max = min(y0, y1), max(y0, y1)
 65:             items = self.canvas.find_enclosed(x_min, y_min,
x_max, y_max)
 66:             for item in items:
 67:                 self.canvas.delete(item)
 68:             self.canvas.delete(self._select_rect)
 69:             self._select_rect = None
 70:             self._start_x = None
 71:             self._start_y = None
 72:
 73:     def hent_storrelse(self):
 74:         size = self.size_var.get()
 75:         if size == "Liten":
 76:             return 40, 40
 77:         elif size == "Middels":
 78:             return 80, 80
 79:         else:
 80:             return 120, 120
 81:
 82:     def tegn_rektangel(self):
 83:         w, h = self.hent_storrelse()
 84:         x0 = random.randint(0, self.canvas_width - w)
 85:         y0 = random.randint(0, self.canvas_height - h)
 86:         x1, y1 = x0 + w, y0 + h
 87:         self.canvas.create_rectangle(x0, y0, x1, y1,
fill="lightblue", outline="black")
 88:
 89:     def tegn_oval(self):
 90:         w, h = self.hent_storrelse()
 91:         x0 = random.randint(0, self.canvas_width - w)
 92:         y0 = random.randint(0, self.canvas_height - h)
 93:         x1, y1 = x0 + w, y0 + h
 94:         self.canvas.create_oval(x0, y0, x1, y1,
fill="lightgreen", outline="black")
 95:
```

```
 96:     def tegn_trekant(self):
 97:         w, h = self.hent_storrelse()
 98:         # Sørg for at hele trekanten er innenfor canvas
 99:         x0 = random.randint(0, self.canvas_width - w)
100:         y0 = random.randint(h, self.canvas_height)  # y0 er
bunnen av trekanten
101:         points = [x0, y0, x0 + w, y0, x0 + w/2, y0 - h]
102:         self.canvas.create_polygon(points,
fill="lightcoral", outline="black")
103:
104: if __name__ == "__main__":
105:     root = tk.Tk()
106:     app = CanvasDemo(root)
107:     root.mainloop()
```

Forklaring (det viktigste):

Linje 16
Bruker en `ttk.LabelFrame` som gir en innramming av innholdet i frame.

Linje 15 – 33:
Koden er skrevet i en klasse. Legg merke til at kun de variablene som skal brukes i flere metoder (altså medlemsmetoder) trenger å være instansvariabler (med `self`). Variabler som kun brukes lokalt i `init`-metoden, kan være vanlige lokale variabler uten `self`.

Linje 40 – 42:
Hjelpevariabler for sletting med dragging.

Linje 44 – 46:
Knytter mus-hendelser til callbacks for å håndtere sletting med dragging.

Linje 48 – 52:
Lager starten til et markeringsrektangel der hvor mus klikkes.

Linje 54 – 57:

Oppdaterer markeringsrektangelet. `coords()` metoden oppdaterer objektets (`self.select_rect`) posisjon med de nye koordinatene fra `event` objektet.

Linje 59 – 71:
Finner ut hvilke objekter som ligger innenfor markeringsrektangelet, metoden `find_enclosed()` er nyttig!
Sletter figurene som er i lista, og sletter markeringsrektangelet.

Her er en oversikt over sentrale re-tegning og oppdatering av figurer i et Tkinter `canvas`:

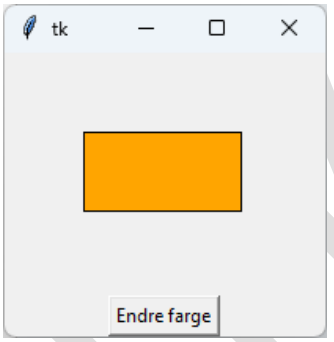
Metode	**Beskrivelse**
`move(item, dx, dy)`	Flytter et objekt relativt til sin nåværende posisjon.
`coords(item, ...)`	Endrer koordinatene til et objekt (absolutt posisjon).
`itemconfig(item, **kwargs)`	Endrer egenskaper som farge, fyll, strektykkelse, tekstinnhold osv.
`delete(item)`	Fjerner et objekt fra canvas.
`find_all()`	Returnerer ID-er til alle objekter på canvas.
`find_enclosed(x1, y1, x2, y2)`	Returnerer objekter innenfor et rektangel.
`find_overlapping(x1, y1, x2, y2)`	Returnerer objekter som overlapper et område.
`tag_raise(item)` / `tag_lower(item)`	Endrer rekkefølgen (z-indeksen) til objekter.
update()	Tvinger umiddelbar oppdatering av GUI (brukes sjelden).

12.5.2 `itemconfig()` metoden

Kort kodeeksempel `itemconfig()`:

Før kall til `itemconfig()`

Etter kall til `itemconfig()`



Koden:

```
# file: sc_12_12_itemconfig_demo.py
import tkinter as tk

def endre_farge():
    canvas.itemconfig(rektangel_id, fill="orange",
outline="black")

root = tk.Tk()
canvas = tk.Canvas(root, width=200, height=150)
canvas.pack()

rektangel_id = canvas.create_rectangle(50, 50, 150, 100,
fill="blue")
```

```
btn = tk.Button(root, text="Endre farge",
command=endre_farge)
btn.pack()

root.mainloop()
```

12.5.3 Forskjellen mellom config() og itemconfig()

I Tkinter er både `config()` og `itemconfig()` metoder som brukes til å endre egenskaper ved widgets eller canvas-objekter, men de brukes i forskjellige sammenhenger.

`Canvas` er en spesiell widget som kan inneholde mange uavhengige grafiske objekter.

`itemconfig()` er nødvendig for å *endre ett spesifikt objekt* på canvas, mens `config()` endrer hele widgeten.

Tabell over en del egenskaper som kan endres med `itemconfig()`:

Egenskap	**Beskrivelse**
`fill`	Fyllfarge på figuren (f.eks. "red", "#00FF00").
`outline`	Farge på kantlinjen.
`width`	Tykkelse på kantlinjen (i piksler).
`dash`	Stiplet linje (f.eks. (4, 2) for 4 piksler strek, 2 piksler mellomrom).
`state`	Synlighet: "normal" (synlig), "hidden" (skjult), "disabled".
`tags`	Legger til eller endrer tagger for objektet.
`text`	Tekstinnhold (kun for tekstobjekter).

`font`	Skrift og størrelse (f.eks. ("Arial", 12)).
`anchor`	Plassering av tekst i forhold til koordinatene (f.eks. "center", "nw").
`image`	Endrer bildet som vises (kun for bildeobjekter).

12.6 Animasjoner

Animasjoner med figurer i Tkinter lages prinsipielt på denne måten:

Tegn figuren i startposisjon
Bruk `canvas.create_oval()`, `create_rectangle()` eller lignende for å tegne figuren.
Lagre ID-en som returneres, slik at du kan flytte på figuren senere:

```
ball = canvas.create_oval(10, 10, 40, 40, fill="blue")
```

Bestem bevegelse per oppdatering
Definer hvor mange piksler figuren skal flyttes i hver retning:

```
dx = 3  # horisontal fart
dy = 2  # vertikal fart
```

Lag en animasjonsfunksjon
Denne funksjonen skal:
Flytte figuren med `canvas.move()`
Eventuelt sjekke for kollisjoner eller grenser; - bruk gjerne `coords()` til dette.
Kalle seg selv igjen etter en kort pause med `canvas.after()`:

```
def animate():
    canvas.move(ball, dx, dy)
    canvas.after(20, animate)  # vent 20ms, kall animate
igjen
```

Start animasjonen
Kall `animate()` én gang for å sette i gang løkken:

```
animate()
```

Hva styrer hastigheten på animasjonen?

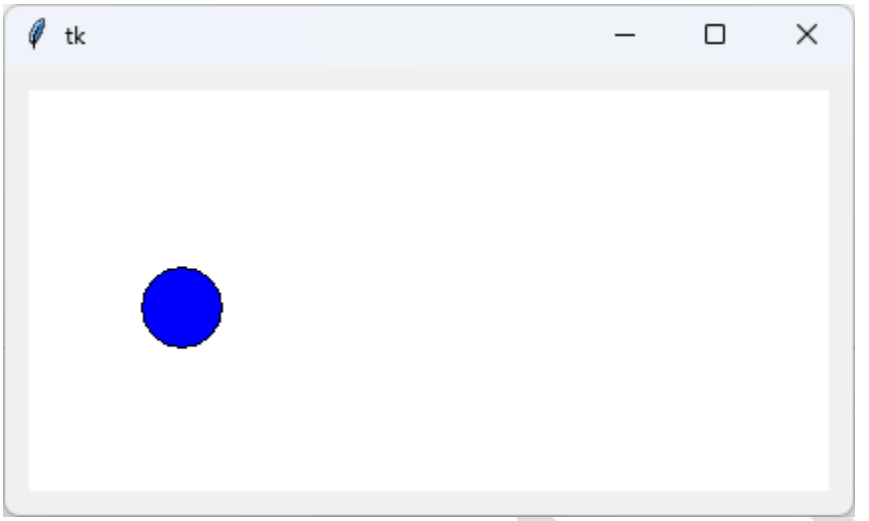
Faktor	Effekt
`dx, dy`	Hvor langt figuren flyttes per oppdatering (fart)
`after(ms)`	Hvor ofte oppdateringen skjer (frekvens)
Kombinasjonen	Bestemmer hvor jevn og rask animasjonen oppleves

«Vanlige» feil og hvorfor de bør unngås

Feil	Hvorfor det er et problem
`time.sleep()`	Fryser hele GUI-et – bruk `after()` i stedet
Tegne nye figurer i stedet for å flytte	Skaper "spor" og bruker mer minne
Glemme å kalle `animate()` på nytt	Animasjonen stopper etter én oppdatering
For høy `dx dy` uten høy frekvens	Bevegelsen blir hakkete og ujevn

12.6.1Enkel ball animasjon

Programmet under demonstrerer en enkel «ball bounce» animasjon som følger oppskriften over.




```
# file: sc_12_11_ball_simple.py
import tkinter as tk

root = tk.Tk()
canvas = tk.Canvas(root, width=400, height=200,
bg="white")
canvas.pack(padx=10, pady=10)

oval = canvas.create_oval(10, 90, 50, 130,
fill="blue")
dx = 4

def animate():
    global dx
    canvas.move(oval, dx, 0)
    x0, _, x1, _ = canvas.coords(oval)
    if x0 <= 0 or x1 >= 400:
        dx = -dx
    canvas.after(20, animate)

animate()
root.mainloop()
```

Kommentar til koden:

`dx` er deklarert `global` fordi den må «huske» verdien; - `dx` endres inne i `animate()`.

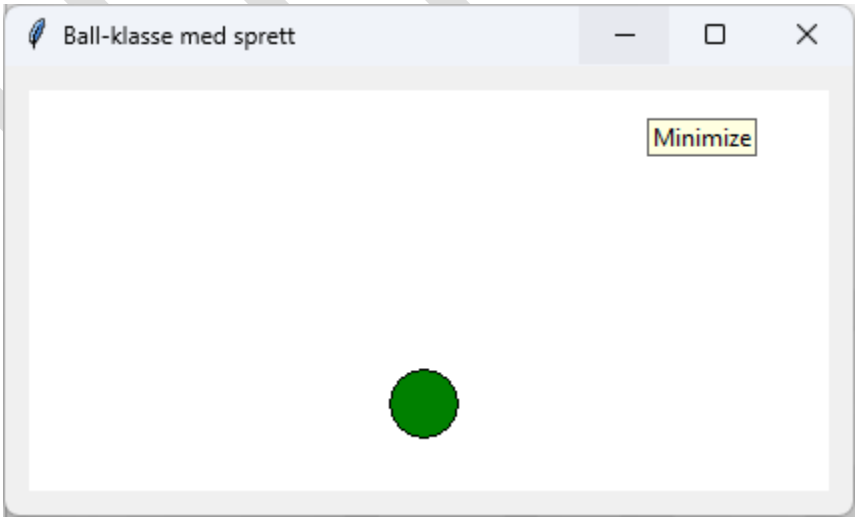
Variabelen `oval` må ikke deklareres som global fordi den ikke endres (reassignes) inne i funksjonen `animate()` – den bare brukes (leses). I Python trenger vi kun å bruke `global` hvis vi skal tilordne en ny verdi til en global variabel inne i en funksjon. Siden `oval` bare brukes som et argument til `canvas.move` og `canvas.coords`, holder det at den er definert utenfor funksjonen.

Funksjonen animate():
`move()` er metoden vi bruker for å flytte figuren

Vi bruker `coord()` for å få koordinatene til figuren, enkleste måte å sjekke om evt kollisjon.

12.6.1.1 En litt mer sofistikert ball animasjon

Koden under viser en mer sofistikert animasjon, med ballen implementert som en egen *klasse*. Det er ballen sjøl som bør finne ut hvordan den skal sprette, derfor er det laget en `move()` metode i klassen som finner ut dette.



```
# file: sc_12_12_animation_ball_class.py
import tkinter as tk
import random

class Ball:
    def __init__(self, canvas, color, size, dx, dy):
        self.canvas = canvas
        self.size = size
        self.dx = dx
        self.dy = dy
        self.id = canvas.create_oval(10, 90, 10+size,
90+size, fill=color)

    def move(self):
        self.canvas.move(self.id, self.dx, self.dy)
        x0, y0, x1, y1 = self.canvas.coords(self.id)
        # Sprett i x-retning
        if x0 <= 0 or x1 >= int(self.canvas['width']):
            self.dx = -self.dx
        # Sprett i y-retning
        if y0 <= 0 or y1 >= int(self.canvas['height']):
            self.dy = -self.dy

class BallAnimation:
    def __init__(self, root):
        self.canvas = tk.Canvas(root, width=400,
height=200, bg="white")
        self.canvas.pack(padx=10, pady=10)
        # Eksempel: tilfeldig farge, størrelse og retning
        color = random.choice(["blue", "red", "green",
"orange"])
        size = random.randint(20, 50)
        dx = random.choice([-4, 4])
        dy = random.choice([-3, 3])
        self.ball = Ball(self.canvas, color, size, dx, dy)
        self.animate()

    def animate(self):
        self.ball.move()
        self.canvas.after(20, self.animate)

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Ball-klasse med sprett")
```

```
BallAnimation(root)
root.mainloop()
```

12.7Dialogbokser