Kapittel 12: GUI programmering

Del 2 – utvalgte widges, canvas, hendelser, simuleringer

Mer om hendelseshåndtering (1)

- 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)

Mer om hendelseshåndtering (2)

- 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.

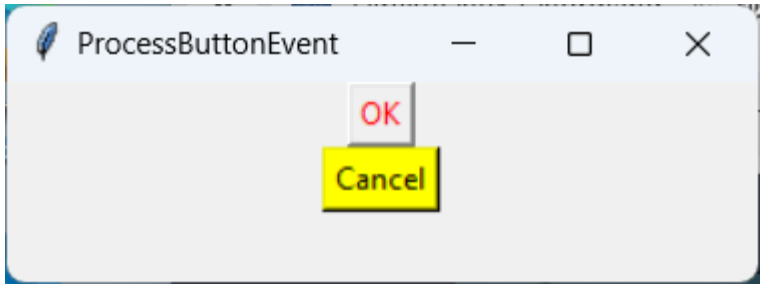
`command` uten parametre

- Følgende widgets har `command` opsjonen:
  - Button - vanlige trykknapper
  - Checkbutton - avkryssingsbokser
  - Radiobutton - radioknapper
  - Scale - glidebrytere
  - Scrollbar - rullefelt
  - Menu/Menubutton - menyer og menyknapper

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

def processOK():
    print("OK button is clicked")

def processCancel():
    print("Cancel button is clicked")

root = tk.Tk() # Create a root window
root.title("ProcessButtonEvent") # Set title
root.geometry("300x80") # Set window size to make title visible
bt_OK = tk.Button(root, text = "OK", fg = "red", command = processOK)
bt_Cancel = tk.Button(root, text = "Cancel", bg = "yellow",
                  command = processCancel)
bt_OK.pack() # Place the button in the window
bt_Cancel.pack() # Place the button in the window

root.mainloop() # Create an event loop
```

Callback metoder

Funksjonsnavn uten parenteser!

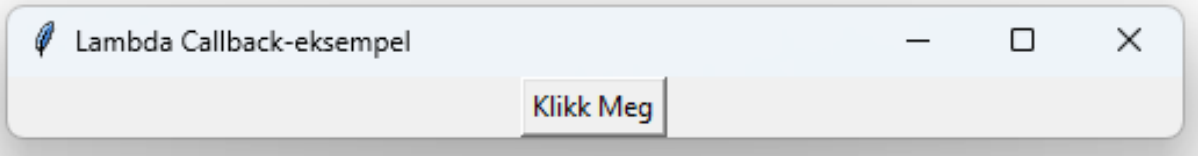



```
/OneDrive - UiT Office 365/Faglig/Python/Liang/Bool
eDrive - UiT Office 365/Faglig/Python/Liang/BookSo
OK button is clicked
Cancel button is clicked
```

command med parametre - lambda

```
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()
```

```
Tekst: Hei, Liste: [1]
Tekst: Hei, Liste: [1, 2]
Tekst: Hei, Liste: [1, 2, 3]
Tekst: Hei, Liste: [1, 2, 3, 4]
Tekst: Hei, Liste: [1, 2, 3, 4, 5]
```



`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)
```

`lambda` – hva skjer?

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

  `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

lambda – hva skjer?

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) # closure 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

closure = inner function

```
def outer(x):
    def inner(y):
        return x + y
    return inner


add5 = outer(5)
print(add5(3))  # Output: 8
```

`outer(x)` er en funksjon som returnerer en annen funksjon (`inner`).
`inner(y)` bruker verdien `x` fra det ytre scope.
Når vi kaller `outer(5)`, får vi tilbake en funksjon som husker at `x = 5`.
Når vi så kaller `add5(3)`, blir det som å gjøre `5 + 3`.

Dette er en closure: `inner` husker verdien av `x` selv etter at `outer` er ferdig.

```
# samme med lambda
def outer(x):
    return lambda y: x + y

add5 = outer(5)
print(add5(3))  # Output: 8
```

`lambda y: x + y` er en anonym funksjon som tar `y` og returnerer `x + y`.
Den fanger `x` fra det ytre scope – akkurat som `inner` gjorde.
Dette er en lambda-basert closure.

`bind()` for å kalle hendelseslytter

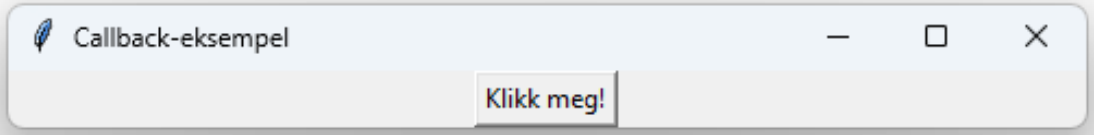
- Den generelle syntaksen for `bind()` i Tkinter er:
  `widget.bind(hendelse, callback)`
- **`widget`**
  - Referanse til widgeten du vil knytte hendelsen til (f.eks. en knapp).
- **`hendelse`**
  - En streng som beskriver hendelsen, for eksempel `"<Button-1>"` for venstre museklikk, `"<Key>"` for tastetrykk, 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)`

command vs bind(), her command

sc_12_01_button.py M

ch_12_GUI_with_Tkinter > sc_12_01_button.py > ...

```python
# file: sc_12_01_button.py
import tkinter as tk

def on_button_click():
    print("Knappen ble trykket!")

root = tk.Tk()
root.title("Callback-eksempel")

bt_simple = tk.Button(root, text="Klikk meg!", command=on_button_click)
bt_simple.pack()

root.mainloop()
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

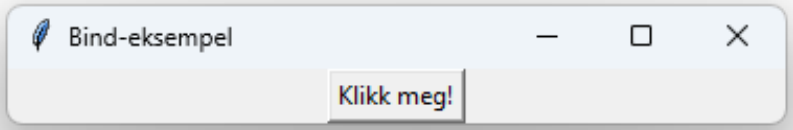
```
PS C:\Users\fna003\OneDrive - UiT Office 365\Faglig\Python\Book\Code\tb_sc> & "C:/Program Files/Python310/python
sers/fna003/OneDrive - UiT Office 365/Faglig/Python/Book/Code/tb_sc/ch_12_GUI_with_Tkinter/sc_12_01_button.py"
Knappen ble trykket!
```

command vs bind(), her bind()

sc_12_07_button_and_bind.py 5, U

ch_12_GUI_with_Tkinter > sc_12_07_button_and_bind.py > on_button_click

```python
# file: sc_12_07_button_and_bind.py
import tkinter as tk

def on_button_click(event):
    print(f"Museklikk koordinater: x={event.x}, y={event.y}")

root = tk.Tk()
root.title("Bind-eksempel")

bt_simple = tk.Button(root, text="Klikk meg!")
bt_simple.pack()

# Bruker bind i stedet for command
bt_simple.bind("<Button-1>", on_button_click)

root.mainloop()

```



PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS C:\Users\fna003\OneDrive - UiT Office 365\Faglig\Python\Book\Code\tb_sc> & "C:/Program Files/Python310/python.exe" "
sers/fna003/OneDrive - UiT Office 365/Faglig/Python/Book/Code/tb_sc/ch_12_GUI_with_Tkinter/sc_12_07_button_and_bind.py"
Museklikk koordinater: x=28, y=11
```

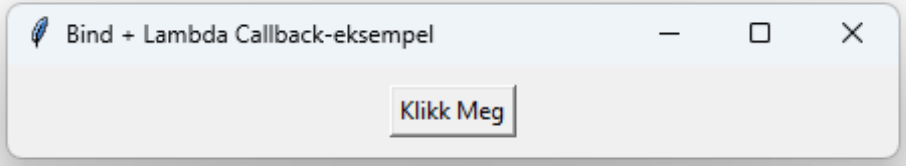
bind() og lambda

sc_12_08_button_and_bind_lambda.py 9+, U

ch_12_GUI_with_Tkinter > sc_12_08_button_and_bind_lambda.py > ...

```python
# file: sc_12_08_button_and_bind_lambda.py
import tkinter as tk

def on_button_click(text, a_list, event):
    print(f"Event: {event}")
    print(f"Tekst: {text}, Liste: {a_list}")
    print(f"Widget som ble klikket: {event.widget}")
    print(f"Museklikk koordinater: x={event.x}, y={event.y}")
    a_list.append(a_list[-1] + 1)
    print("---")

root = tk.Tk()
root.title("Bind + Lambda Callback-eksempel")

some_list = [1]

# Bruk bind i stedet for command
bt_simple = tk.Button(root, text="Klikk Meg")
bt_simple.bind("<Button-1>", lambda event: on_button_click("Hei fra bind", some_list, event))
bt_simple.pack(pady=10)

root.mainloop()
```

```
Event: <ButtonPress event num=1 x=18 y=5>
Tekst: Hei fra bind, Liste: [1]
Widget som ble klikket: .!button
Museklikk koordinater: x=18, y=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.

innhold i `event` objektet

- 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

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.).

Canvas: canvas *items*

- Canvas er en kraftig widget som lar oss tegne grafiske elementer (*canvas items*) 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.

Canvas og mus-hendelser

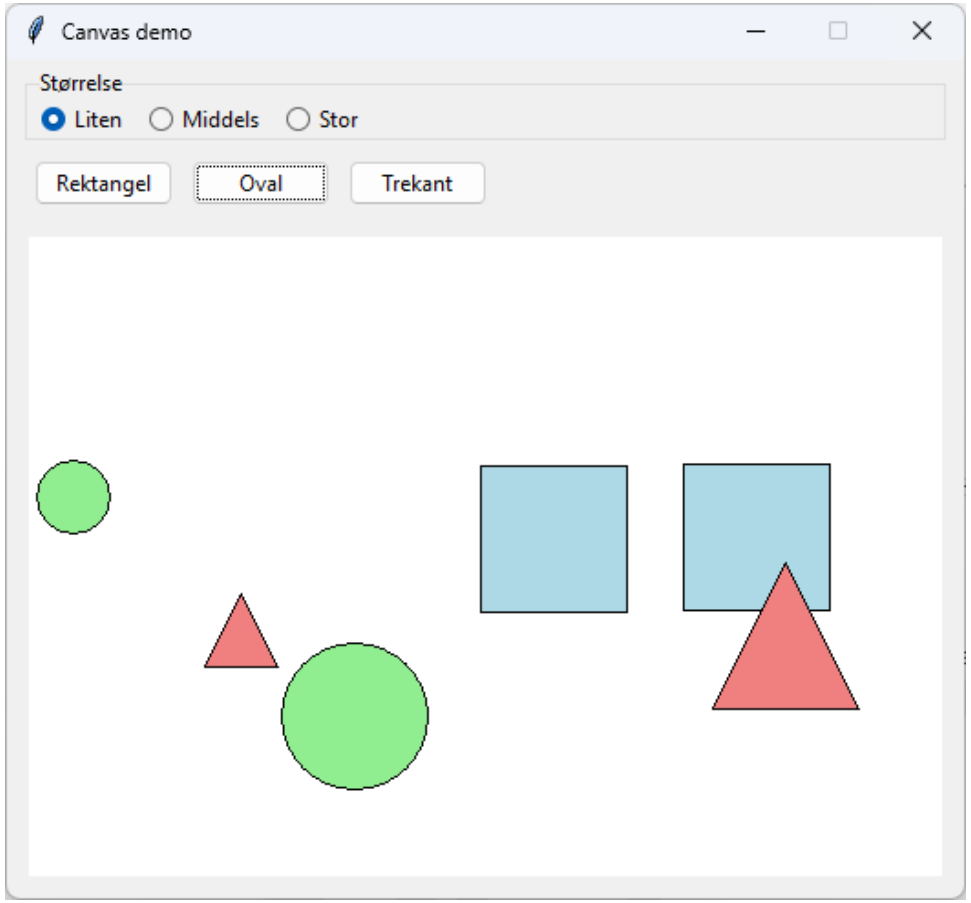
Mus-hendelser kan knyttes til alle typer widgets, og vi får ofte behov for å detektere slike i forbindelse med bruk av Canvas.

Programmet til høyre 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.

Kode: sc_12_09_canvas_demo.py



Oversikt:
Re-tegning og oppdatering av figurer i et Tkinter canvas

Metode	**Beskrivelse**
`move(item, dx, dy)`	`Flytter et objekt relativt dx, dy i forhold til sin nåværende posisjon.`

`coords(item, ...)`	*Endrer* koordinatene til et objekt (absolutt posisjon), eller *henter* koordinater.
	
	Hente koordinater:
	`coords = canvas.coords(shape_id)`
	
	Oppdatere koordinater:
	`canvas.coords(shape_id, new_x0, new_y0, new_x1, new_y1)`

Oversikt:
Re-tegning og oppdatering av figurer i et Tkinter canvas

`itemconfig(item, **kwargs)`	Endrer egenskaper som farge, fyll, strektykkelse, tekstinnhold osv.

`delete(item)`	Fjerner et objekt fra canvas. Kan bruke keyword argument tags for å fjerne shapes med bestemt tag.
	
	`# Slett alle objekter med tag "ball"`
	
	`canvas.delete("ball")`

`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).

itemconfig – endre canvas items

```
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()
```

Før kall til itemconfig()

Etter kall til itemconfig()

Forskjell `config` og `itemconfig`

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

Eksempel på 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).

Animasjoner (1)

Oppskrift lage animasjon i Tkinter

1) 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")
```

2) Bestem bevegelse per oppdatering

- Definer hvor mange piksler figuren skal flyttes i hver retning:

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

Animasjoner (2)

3) 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
```

4) Start animasjonen

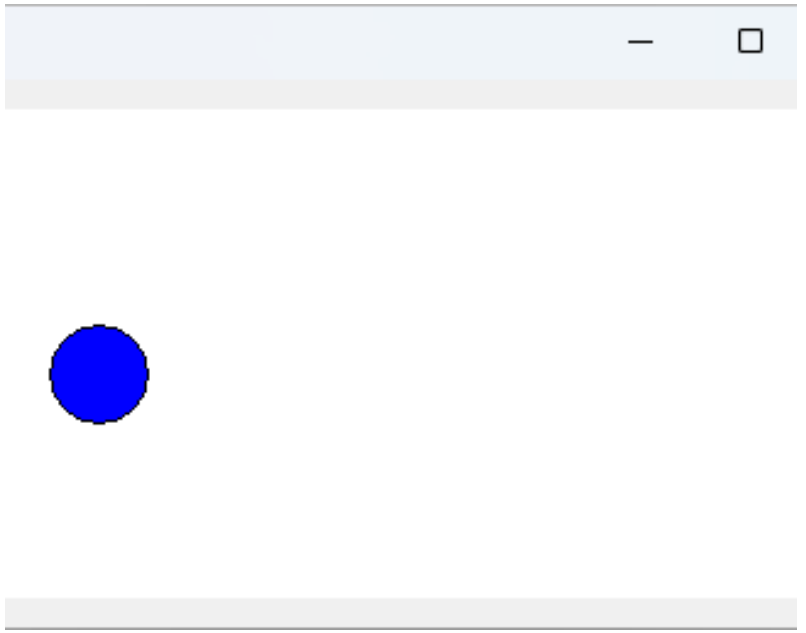
- Kall `animate()` én gang for å sette i gang løkken:

```
animate()
```

```
# 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()
```

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

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



En litt mer sofistikert ball animasjon

- Neste versjon er 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.

Fil:
sc_12_12_animation_ball_class.py

```
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
```