Repetisjon INF100

Av Tyra Fosheim Eide og Hilde Jordal

Temaliste

- Repetisjon av basiskonsepter
- Betingelser
- Løkker
- Funksjoner
- Strenger
- Lister, flerdimensjonale lister
- Grafisk brukergrensesnitt
- Filer og CSV
- Håndtere krasj
- Oppslagsverk, mengder
- Moduler, standardbiblioteket og eksterne pakker
- Plotting av grafer

Repetisjon av basiskonsepter

Algoritme

- Instruksjoner for hvordan noe skal gjøres

- Bryte ned oppgaven i enkelte steg
- Hvor mye må vi bryte ned??

- Datamaskinen din vet nesten ingenting!
  - Vær tydelig

Variabler

- Referanse til en verdi

```
min_var = 5
```


```
min_var = min_var + 1 # 5 + 1 = 6
```

- Viktig når vi lager variabler:
  - Beskrivende navn
  - Liten bokstav (som regel)
  - _ i stedet for mellomrom (snake_case)
    - lowerCamelCase er lov, men ikke offisiell Python syntax. Vær konsekvent!

```
shirt_size = "L"
```

  - Konstanter i caps lock

```
MINIMUM_AGE = 12
```

  - Ikke bruk innebygde nøkkelord eller funksjonsnavn!!!
    - False, None, True, and, as, assert …

Uttrykk

- Består av én eller flere verdier, variabler operasjoner og funksjoner
- Evalueres til en enkelt verdi

```
5
```

```
min(3, min_var)
```

  - **Setning:** et steg i et python-program, representerer en handling

```
print("Hallo")
```

- Variabler husker bare verdien, ikke uttrykket!!!

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

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

>> y = 7. Programmet husker ikke at det var x + 2

Print/input

- **print()** skriver ut verdier til terminalen med linjeskift (\n) på slutten
  - Kan skrive ut flere ting på en gang
  - Bruk end=«» for å avslutte med noe annet enn linjeskift

```
print("Foo")  # streng/tekst
print(42)     # nummer
print(True)   # boolske verdier
```

```
print("Foo", "Bar")
```

```
print("Foo", end="---*---")
print("Bar")
```

- **input()** leser inn verdier fra terminalen
  - Verdien som leses inn har alltid typen streng!!

```
a = input("Si et tall: ")
b = input("Si et tall til: ")
print("Summen er ", a + b)
```

>> 3

>> 4

>> «Summen er 34»

Hva skjer her?

Løsning:

```
a = int(a)
b = int(b)

print("Summen er ", a + b)
```

>> «Summen er 7»

f-streng

- En enklere måte å skrive ut variabler!
- Sett `f` foran strengen, og plasser variablene inni `{}`

```
melding = “Hvor er kaken?”
f_streng = f”Her er meldingen: {melding}”

streng = ”Her er meldingen: ” + melding
```

f-streng

- En enklere måte å skrive ut variabler!

```
1 navn = "Hege"
2 alder = 30
3 yrke = "Programmerer"
```

```
5 # Uten f-strings
6 print("Jeg heter " + navn + ", jeg er " + str(alder) + " år gammel, og jeg jobber som " + yrke + ".")
```

```
8 # Med f-strings
9 print(f"Jeg heter {navn}, jeg er {alder} år gammel, og jeg jobber som {yrke}.")
```

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 ...`

- Operatorer er listet etter presedens
- **Presedens:** vi bruker standard regneregler

```
print(2 + 3 * 4)   # 14
print((2 + 3) * 4) # 20
```

- **Assosiativitet:** operasjoner med samme presedens utføres som regel fra venstre til høyre
  - Unntak!! Høyre- og ventre-assossiativ
  - Relasjoner assossierer IKKE!

```
0 < 3 == 3    # (0 < 3) and (3 == 3)    True
(0 < 3) == 3  # True == 3               False
```

= vs. ==

- Hvordan vet jeg hvilken jeg skal bruke?
  - Spørsmål?
  - Eller fakta?

- = brukes til å tilordne verdier til variabler.
- == brukes for å sammenligne to verdier og returnerer en boolsk verdi (True/False).

Operatorer - oppgave

```
x = 1066
print(x % 2  == 0 and not x % 4 == 0)
```

```
print(2 ** 3 ** 4) # Hva evalueres først?
```

```
print(1 < 2 == True)
```

```
print((1 < 2) == True)
```

```
print (1 < 2 < 3)
```

Typer

- Alle verdier har en type!
  - Kan finne typen med type()-funksjonen
- Viktige typer i Python
  - str (streng/tekst)
  - int (heltall)
  - float (flyttall/desimaltall)
  - bool (boolsk verdi; True eller False)
  - NoneType («ingenting» – dette er også en type!)
  - list
  - tuple
  - set
  - dict
- Vi kan konvertere mellom typer
- Typen bestemmer hva en operasjon betyr

```
a = "2"
b = "3"
print(a + b) # 23

c = 2
d = 3
print(c + d) # 5
```

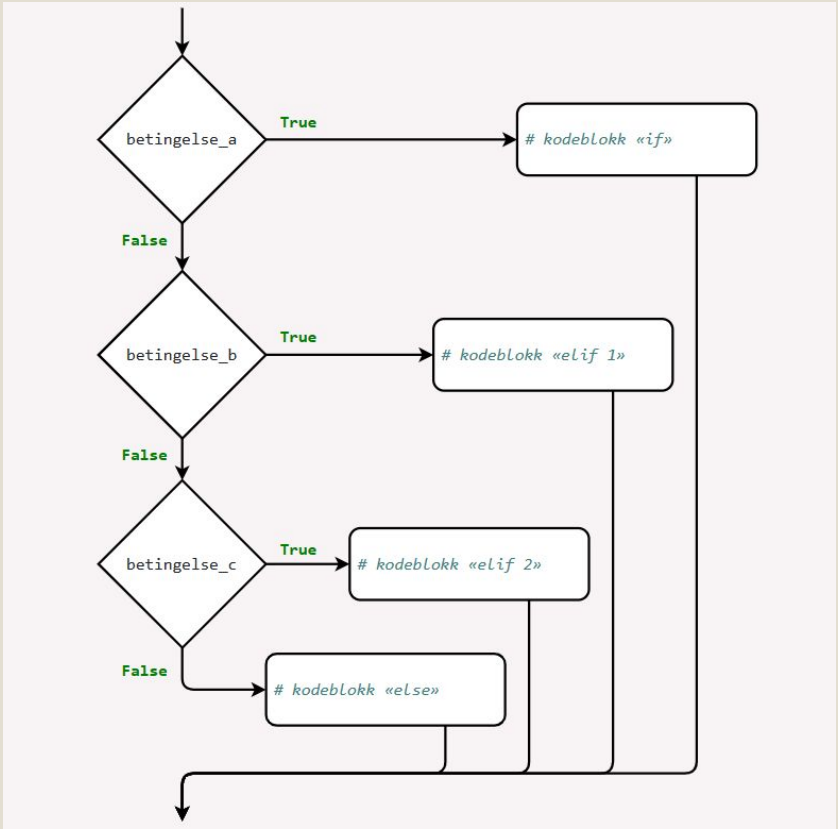
- Vi kan konvertere mellom typer

```
string = "this is a string"
stringnum = "3"
num = 0
flo = 23.5

#print(int(string))   # Krasj!
print(int(stringnum)) # 3
print(int(flo))       # 23
print(float(num))     # 0.0
print(bool(string))   # True
print(bool(num))      # False
```

Betingelser

Betingelser

- **if-setninger** kan brukes til å utføre en kodesnutt kun hvis en betingelse er oppfylt
- Relasjonsoperatorer:
  - ==, !=, <, >, <=, >=
- Logiske operatorer:
  - And, or, not
- **elif** evaluerer en ny betingelser hvis den første evaluerte til False. Første if/elif som er True kjøres.
- **else** utføres dersom alle betingelsene evalueres til False



Betingelser

- **If-else-uttrykk:**

```
print("Foo" if True else "Bar")
```

- **Truthy or falsy verdier:** betingelser som ikke evalueres til True eller False gis likevel en verdi True eller False
  - Verdier som tolkes som True kalles *truthy*, og verdier som tolkes som false heter *falsy*

```
print("Foo" if "Bar" else "Bar")
```

  - Kan bruke bool() til å sjekke om en verdi er truthy eller falsy

```
print(bool("Foobar"))
```

  - Som regel «tom» versjon av verdien som regnes som falsy
    - 0, [], (), {}, «», None

Betingelser - tips

- Husk!
  - Bruk else i stedet for en ekstra if som sjekker det motsatte
  - Ikke lag tomme if-setninger
  - Alle verdier evalueres til True/False! Du trenger ikke sammenligne med True/False

  - Hvis du har betingelser som utelukker hverandre, bruk elif og ikke flere if-setninger etter hverandre
  - Hold det enkelt og oversiktlig

Løkker

Løkker

- En måte å utføre en blokk med kode flere ganger
- While-løkke:
  - Utfører kodeblokken så lenge betingelsen er true
- For-løkke:
  - Itererer over alle elementer i en samling
    - Range, streng, lister, osv.
- Break bryter løkken og går videre i koden
- Continue avbryter kodeblokken og går til neste iterasjon

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

```
for i in range(10):
    print(i) # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
searching_for = 'w'
for letter in "This is a long string where I'm searching for the letter w":
    if letter == searching_for:
        print("Found w!")
        break
```

```
while True:
    answer = int(input())
    if (answer % 2 != 0):
        print("Please only say an even number.")
        continue
    print("Thanks!")
    break
```

Løkker - range()

- Returnerer en sekvens av integers
- Begynner på 0 som default
- Må spesifisere slutt, kan spesifisere start og step
- Step - hopper over elementer
  (default step 1, altså hopper ikke over noen)

```
for i in range(5):
    print(i)
# 0 1 2 3 4

for i in range(1, 6):
    print(i)
# 1 2 3 4 5

for i in range(0, 50, 5):
    print(i)
# 0 5 10 15 20 25 30 35 40 45

a = 2
b = 5
for i in range(a, b):
    print(i)
# 2 3 4
```

Løkker – når bruker jeg hva?

1: Jeg vet hvor mange ganger jeg skal kjøre koden (i hvert fall maks)
- For-løkke!

2: Jeg vet IKKE hvor mange ganger jeg skal kjøre koden, men jeg vet når den skal stoppe
- While-løkke!

- Bruk alltid en for-løkke hvis det er naturlig.

```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)
```

Løkker – nøstede løkker

- Vi kan ha løkker inni løkker
- For hver iterasjon av den ytterste løkken, kjøres alle iterasjonene av den innerste løkken

```
for a in range(1, 11):
    print(f"{a}-gangen: ", end="")
    for b in range(1, 11):
        print(b*a, end=" ")
    print()
```

- Skal du ha tak i hver b i hver a? Da kan det tenkes at du skal bruke en nøstet løkke.
  - Eks.: hver kolonne i hver rad, hver bokstav i hvert ord, hvert element i hver liste

Oppgave - nøstede løkker

- La oss si at vi i et år har 4 sesonger (“Vår”,”Sommer”,”Høst”,”Vinter”) og hver sesong har dager fra 1 til og med 28.
  - Print ut alle datoene i dette formatet: {sesong} {dag}
  - For eksempel:
    - Vinter 1
    - Vår 22
    - Høst 28
  - n = 28
  - sesonger = [“Vår”,”Sommer”,”Høst”,”Vinter”]

Funksjoner

Funksjoner

- En funksjon er en kodesnutt vi kan kjøre ved å *kalle* på den ved navn.
- Parameter vs. Argument
  - Parameter er bare en boks som venter på å bli fylt – en variabel

```
def sample_function(par_1, par_2):
    ...
```

  - Argument er en verdi som finnes her og nå

```
arg_1 = 1
arg_2 = 2

sample_function(arg_1, arg_2)
sample_function(1, 2)
```

  - Husk at variabelnavn bare er *referanser*, og at en verdi kan refereres til av flere variabler

- Funksjonen returnerer alltid noe
  - Hvis retur ikke er spesifisert, returnerer funksjonen None ☺
- Når funksjonen returnerer, *slettes alle lokale variabler*!

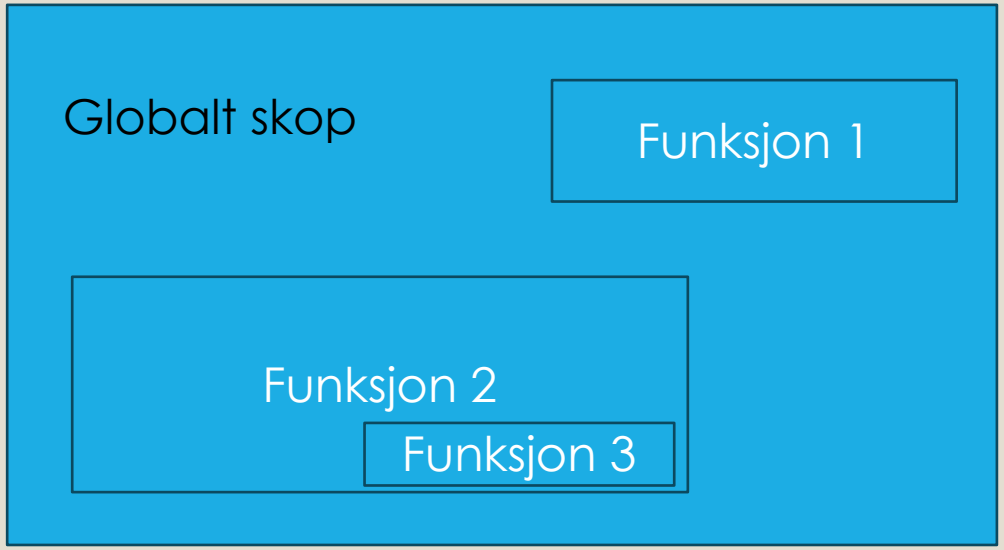
Funksjoner – vanlig feil

```python
def funskjon():
    tall_1 = 1
    tall_2 = 2
    hjelp = hjelpefunksjon(a, b) # ??? Hvorfor blir det feil? hjelpefunksjon() tar jo inn a og b...
    print(hjelp)

def hjelpefunksjon(a, b):
    return a + b
```

```python
def funskjon():
    tall_1 = 1
    tall_2 = 2
    hjelp = hjelpefunksjon(tall_1, tall_2) # ??? aha! a og b er variabler som kun eksisterer i hjelpefunksjon()!
    print(hjelp)

def hjelpefunksjon(a, b):
    return a + b
```

Funksjoner - skop

- Hver funksjon har sitt eget, lokale skop
- Alle variabler definert her slutter å eksistere når funksjonen er ferdig



Funksjoner

- Returverdi eller sideeffekt?
  - Returverdi er det funksjonen *evalueres* til – når vi «regner ut» funksjonen ender vi opp med retur.
  - Sideeffekt er en endring i «verdens tilstand for øvrig» - det dukker opp noe på skjermen
  - Hvordan vet jeg om jeg skal printe eller returnere?
    - Les oppgaveteksten!!

- Poenget med funksjoner er:
  - Å lage abstrakt kode – ta vekk unødvendige detaljer
    - Print funker uansett hva som skal printes!
  - Gjenbruk av kode – vi kan kalle funksjonen mange ganger
  - Selvdokumenterende kode – navnet sier noe om hva funksjonen gjør

Funksjoner - innebygde

- Python har en del innebygde funksjoner som kan være nyttige for oss
- Flere av disse skal vi innom senere, men her er noen av dem som ikke nevnes:
  - abs(x) - returnerer absoluttverdi av tallet x
  - round(x, n) - runder av tallet x med en presisjon på n
    - (om n ikke spesifiseres settes den til 0)
  - …og mange flere!!
  - Built-in Functions — Python 3.12.0 documentation

Programflyt

Programflyt

- Handler om rekkefølgen av stegene koden tar
  - Hvilken gren av en if-setning ender vi opp i?
  - Hvilken kode kjører først?
  - Hvilken verdi har en variabel på et visst steg?

Programflyt - eksempel

```
number = -5

if number > 0:
    print("The number is positive")
else:
    print("The number is negative")
```

Programflyt - oppgave

Finn verdien til x på de ulike kontrollpunktene A,B,C,D

Her får du også testet hva du husker om skop!

```
1  def plus_one(x):
2      x = x + 1
3
4  def add(x, y):
5      return x + y
6
7  x = 9
8
9  # A: x = ?
10
11 plus_one(x)
12
13 # B: x = ?
14
15 x = add(x,x)
16
17 # C: x = ?
18
19 if x < 15:
20     x = 0
21 elif x >= 18:
22     x = 19
23
24 # D: x = ?
```

Lister

Lister - oprettelse

- Opprette liste
  - List() tar argumenter som kan itereres over f.eks range.

```
# To standard måter å opprette tomme lister på
a = []
b = list()
```

- Lage liste med flere elementer

```
a = [2, 3, 5, 7, 11]
b = list(range(5))
c = ['foo', 42, True, None, '']
```

- Lage liste med variabelt antall elementer

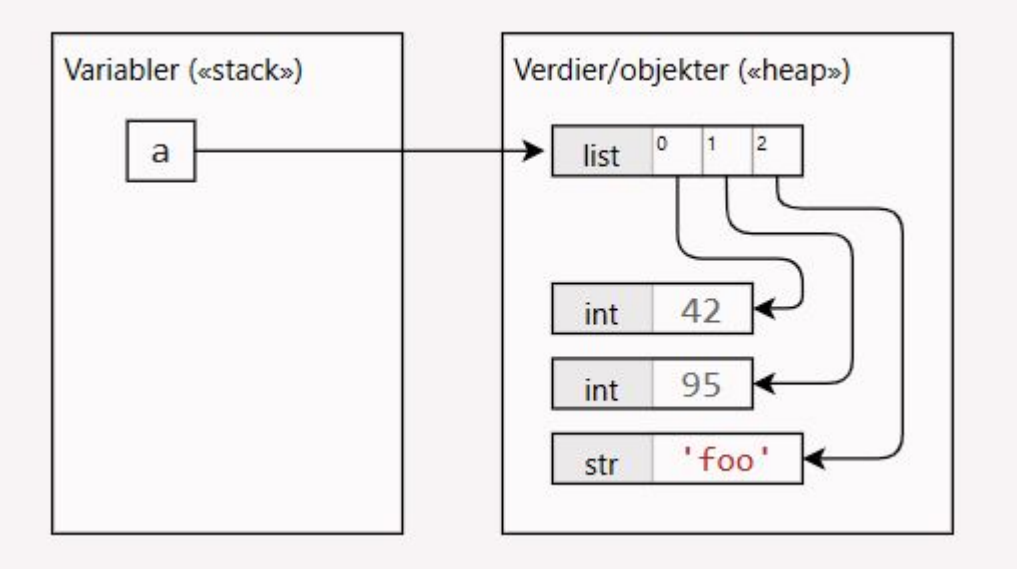
```
n = 5
a = ['foo'] * n
b = [7, 99] * n
c = list(range(n))
```

Lister - hva er det egentlig?

- Lister i Python kan inneholde verdier av mange slags typer

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

- FEIL! Eller, ikke feil i praksis men feil i teorien:
  - Lister i Python inneholder referanser til verdier. Referanse er en bestemt type i Python, men verdien som refereres kan ha hvilken som helst type.



I programmering begynner vi alltid på 0, ikke 1!

Lister - indeksering og beskjæring

- Fungerer på samme måte som for strenger
- Indeksering - henter elementet på en bestemt indeks ut av listen
  - list[0], indeks inni klammeparenteser
- Negative indekser teller baklengs fra slutten av listen (begynner på -1!)
  - list[-1] siste element i listen
  - list[-2] nest siste element i listen
- Beskjæring - henter en del av en liste
  - list[start:slutt] - lager en liste av elementene fra og med start til slutt (slutt er ikke med!)
    - Trenger ikke spesifisere start = 0 eller stop = length
  - list[start:slutt:step] - hopper step antall elementer fra start til slutt
    - Må ha med alle : !

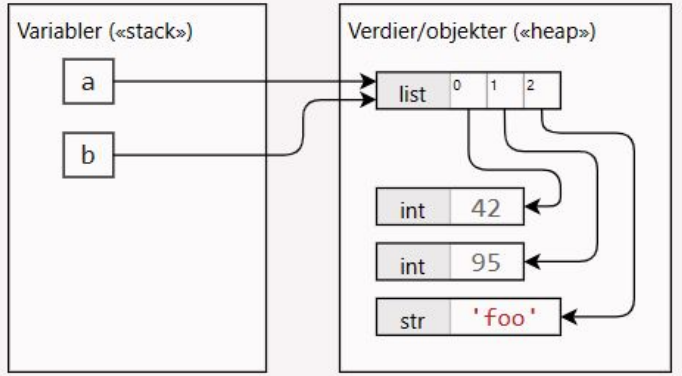
```
a = [2, 3, 5, 7, 11, 13]
print(a[1:6:2]) # [3, 7, 13]
print(a[1:6:3]) # [3, 11]
```

```
a = [2, 3, 5, 7, 11, 13]
print(a[::2]) # [2, 5, 11]
```

Lister - alias og mutasjon

- Si at x = 1 og y = x. Hva er y hvis vi sier x += 1?
  - Fortsatt 1, for den lagret verdien 1, ikke uttrykket x
- Samme skjer hvis vi sier x = a[2], og så sier a[2] += 1.
  - x lagrer verdien som a[2] refererte til.
- MEN! hvis a er en liste:
- Hva om vi sier b = a?
- Nå referer b til samme liste som a referer til.
- Den er et *alias*. Endrer vi på a, så endrer vi på b.

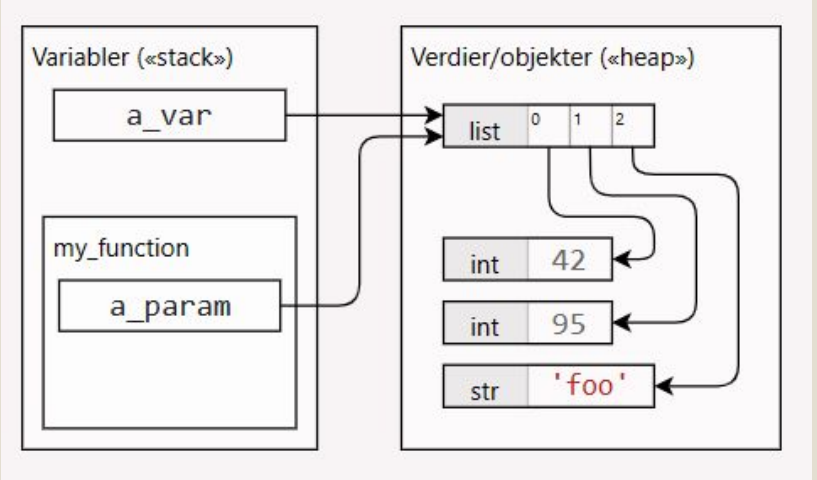
```
a = [42, 95, 'foo']
b = a                # b er nå et alias for a
```



- Når vi endrer en liste blir det *mutert*.
- Det vil si at vi peker til samme sted, men verdiene der er endret på. Samme liste, med annet innhold.

Lister - destruktive funksjoner

- Kalle en funksjon med liste som argument
  - Parameteret til funksjonen blir et alias med argumentet vi ga inn
  - Dette kalles en *destruktiv funksjon*, fordi det endrer på vår originale liste fra utenfor funksjonen.



- Hvis vi ikke vil ha en destruktiv funksjon, må vi lage en ny liste og returnere denne.
- Nå referer a og b til forskjellige steder.
  - a is not b, men a == b
  - Is/is not sier om to verdier er DEN SAMME
  - == sier om to verdier er LIKE

```
def my_func(a_param):
    b = []
    for i in a_param:
        b.append(i)
    return b

a = [1, 2, 3]
b = my_func(a)
```

```
assert(a == b)
assert(a[2] == b[2])
assert(a is not b)

a[2] += 2

assert(not a == b)
assert(not a[2] == b[2])
```

Lister - kopiering

- Vi må være nøye når vi skal kopiere lister så vi ikke lager et alias ved et uhell
- Noen måter å kopiere:
  - Copy-bibliotek:
    - import copy
    - b = copy.copy(a)
    - c = a.copy()
  - Indeksering
    - d = a[:]
  - Konkatenering
    - e = a + []
  - List()
    - f = list(a)
  - Pakke ut
    - *h, = a
  - Løkke
    - g = []
      for element in a:
        g.append(element)
- Poenget: Kopierer du en referanse, så får du et alias. Kopier verdier!

Lister - utpakking og tupler

- Tuple er en datastruktur i Python som fungerer likt som lister, men som ikke kan muteres
  - Man må bruke ikke-destruktive funksjoner
  - Tupler er praktisk for å returnere flere verdier fra en funksjon

```
a = [1, 2] # liste
b = (1, 2) # tuple
```

- Elementer kan hentes fra lister og tupler ved bruk av indeksering:

```
print(a[0]) # 1
print(b[1]) # 2
```

- Eller ved å pakke ut verdiene:
  - vi må ha like mange variabler som det er verdier!

```
x, y = a
n, m = b

print(f"{x} {y}") # 1 2
print(f"{n} {m}") # 1 2
```

Lister - utpakking og tupler

- Vi kan også delvis pakke ut lister
- * foran variabelen forteller at den skal samle resten av listen i en ny liste

```
x, *y = a

print(f"{x} {y}") # 1 [2]
```

- Vi kan pakke ut fra starten og fra slutten
  - x blir første element, z blir siste element
  - Alt annet samles opp av y

```
b = [1, 2, 3, 4, 5, 6]

x, *y, z = b

print(x) # 1
print(y) # [2, 3, 4, 5]
print(z) # 6
```

- Har vi en variabel vi ikke har bruk for, kan vi kalle den _

Lister - løkker over lister

- Vi kan *iterere* over lister.
  - (iterasjon - å utføre noe flere ganger)
- Vi kan enten iterere med indeks…
  - enumerate(a) holder styr på hvor mange ganger vi har iterert

```
# 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)
```

- Eller uten indeks (for-each)
  - item refererer til et element fra a

```
a = [2, 3, 5, 7]
for item in a:
    print(item)
```

Lister - løkker over lister

- Ikke muter en liste du holder på å gå gjennom med for-løkke!

  Python Tutor - mutering av liste inni for-løkke (indeksering)

  Python Tutor - mutering av liste inni for-løkke (for-each)

- Da er det bedre å bruke while løkke, hvor vi selv kan styre hvordan indeks skal endres

  Python Tutor - mutering av liste inni while-løkke

Lister - funksjoner og operasjoner

Informasjon om listen:

- len(a) - antall elementer i listen a
- min(a) - minste element i listen a
- max(a) - største element i listen a
- sum(a) - sum av alle elementer i listen a

Legge til elementer (destruktive):

- a.append(x) - legger til x på slutten av listen a
- a.insert(idx, x) - legger til x på indeks idx i listen a
- a.extend(b) - legger til en liste b på slutten av listen a
- a += b - legger til en liste b på slutten av listen a

Legge til elementer (ikke-destruktive):

- b = a + [1, 2] - tar verdien a og legger til [1, 2] til slutt
- c = a[:1] + [42] + a[1:] - legger til element på indeks 1

Fjerne elementer (destruktive):

- a.remove(x) - fjerner første opptreden av x fra listen a
- a.pop() - fjerner siste element fra listen a
- a.pop(idx) - fjerner elementet på indeks idx i listen a
- pop() returnerer elementet som ble fjernet

Fjerne elementer (ikke-destruktive):

- b = a[:2] + a[3:] - fjerner element på indeks 2

Lister - funksjoner og operasjoner

Operatorer:

- a == b - er alle elementer i a og b like?
- a < b - sammenligner første ulike element
- a > b - sammenligner første ulike element
- a + b - slår sammen listene (konkatenering)
- a * n - repeterer listen n ganger
- a is b - referer a og b til SAMME liste?
- a is not b - motsatt av den over

Sortering av elementer (destruktiv):

- a.sort() - sorterer elementene i stigende rekkefølge
- a.reverse() - reverserer rekkefølgen på elementene

Sortering av elementer (ikke-destruktiv)

- sorted(a) - returnerer en NY sortert liste
- reversed(a)- returnerer et <reversed object> (IKKE EN LISTE!!)
- list(reverse(a)) gjør <reversed object> om til en faktisk liste

Leting etter elementer:

- x in a - inneholder listen a verdien x?
- x not in a - inneholder listen a ikke verdien x?
- a.count(x) - hvor mange ganger forekommer x i listen a?
- a.index(x) - hvilken indeks har x i listen a?
- a.index(x, start) - hvilken indeks fra og med indeks start i listen a har x?

```
a = [2, 3, 4, 5, 3]
print(a.index(3, 2)) # 4
```

Lister - listeinklusjon

- Oppretter en liste ved bruk av en løkke

```
# 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)
```

- Kan også inkludere betingelser

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

Lister - listeinklusjon

- Listeinklusjon kan også brukes for å utføre ikke-destruktive operasjoner

```
# 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)
```

Strenger

Strenger

- Fire måter å skrive en streng i Python:
  - ‘apostrof’
  - “hermetegn”
  - ‘’’trippel apostrof’’’
  - “””trippel hermetegn”””
- Kan bruke hermetegn inni en apostrof-streng, og vice versa
  - “Dette ‘her’ går fint”
  - ‘Og detter “her” går også fint’
  - “Men dette “her” går galt”

Strenger - escape-sekvenser

- Tegn med bakstrek \ foran seg leses av Python som ett tegn
- Noen tegn får da en ny funksjon, mens andre tegn "mister" funksjon

- \n - linjeskift
- \t - tab

- \" - gjør at vi kan ha " inni en "" streng
- \\ - printer \ i stedet for å se på det som en escape-sekvens
- '''Trippel apostrof''' og """Trippel hermetegn""" gjør at vi slipper å bruke escape-sekvens for å få linjeskift.
  Der kan \ brukes for å ekskludere påfølgende linjeskift.

```
print("""\
Hei!
""") # Hei!\n
```

Strenger - konstanter

- Det finnes en del nyttige strenger i string-biblioteket

```
import string
print(string.ascii_letters)   # abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.ascii_lowercase) # abcdefghijklmnopqrstuvwxyz
print("-----------")
print(string.ascii_uppercase) # ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.digits)          # 0123456789
print("-----------")
print(string.punctuation)     # '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
print(string.printable)       # siffer + bokstaver + tegn + whitespace
print("-----------")
print(string.whitespace)      # mellomrom + tab + linjeskift etc....
print("-----------")
```

Strenger - likheter med lister

- En streng oppfører seg på mange måter likt som en liste
- Vi kan tenke på det som en liste av bokstaver, hvor én bokstav er en streng med lengde 1

- Vi kan bruke indeksering og beskjæring
- Vi kan lage løkker over elementene i en streng med og uten indeksering
- Vi kan bruke mange av de samme operatorene og funksjonene

- Strenger er ikke-muterbare
  - Det vil si at hver gang vi forandrer på en streng, så lager vi egentlig en ny streng

```
a = "Streng"
b = a
a += "er"

print(a) # Strenger
print(b) # Streng


a = [1, 2, 3]
b = a
a += [4]

print(a) # [1, 2, 3, 4]
print(b) # [1, 2, 3, 4]
```

Strenger - sammenligning

- Strenger representeres som en rekke av 1'ere og 0'ere i minnet
- Disse oversettes til meningsbærende tegn ved bruk av en *enkoding*
- Python benytter som standard UTF-8, som matcher med en unicode (ordinal-verdi)

- 'A' har unicode-verdien 65, 'a' har unicode-verdien 97
- Vi kan kovertere fra unicode til streng og tilbake
  - ord('a') - gir unicode for 'a'
  - chr(97) - gir tegnet med unicode 97

- Når vi sammenligner strenger, sammenligner vi unicoden!
- Unicodene er gitt slik at sammenligning blir i alfabetisk rekkefølge stort sett
- Pass på!!! Store bokstaver har lavere unicode enn små bokstaver

Strenger - funksjoner og operasjoner

Informasjon om listen:

- len(a) - antall tegn i strengen a
- min(a) - minste unicode-verdi i strengen a
- max(a) - største unicode-verdi i strengen a
- a.isupper() - er alle tegnene i a upper case bokstaver?
- a.islower() - er alle tegnene i a lower case bokstaver?
- a.isnumeric() - er alle tegnene tall?
- …og mye mer!

Flere metoder:

- a.lower() - konverterer alle bokstaver i a til lower case
- a.upper() - konverterer alle bokstaver i a til upper case
- a.replace(x, y) - erstatter substreng x med substreng y i strengen a
- a.strip() - fjerner whitespaces foran og bak i strengen a

Operatorer:

- a == b - er alle bokstaver i a og b like?
- a < b - sammenligner første ulike element
- a > b - sammenligner første ulike element
- a + b - slår sammen strengene (konkatenering)
- a * n - repeterer strengen n ganger

Søking i strenger:

- x in a - inneholder strengen a substrengen x?
- x not in a - inneholder listen a ikke substrengen x?
- a.count(x) - hvor mange ganger forekommer x i strengen a?
- a.startswith(x) - starter strengen a med substrengen x?
- a.endswith(x) - slutter strengen a med substrengen x?
- a.index(x) - hvilken indeks har x i strengen a?
- a.index(x, start) - hvilken indeks fra og med indeks start i strengen a har x?
- a.find(x) - samme som a.index(x), men returnerer -1 om ikke funnet i stedet for å krasje

Strenger - konvertering til liste

- Vi kan bruke metoder til å konvertere mellom streng og liste
- a.split() - deler opp en streng i biter og legger de i en liste
  - mellomrom som default, men vi kan spesifisere splitter
  - a.split(“,”) for eksempel
  - a.split(b)
- ““.join(list) - limer sammen en liste med strenger
  - Kan skrive hva vi vil foran .join() så lenge det er en streng
  - “ “.join(list)
  - “--”.join(list)
  - s.join(list)

Strenger - formatering med f-streng

- Vi har alt sett hvordan vi skriver ut variabler:

```
8  # Med f-strings
9  print(f"Jeg heter {navn}, jeg er {alder} år gammel, og jeg jobber som {yrke}.")
```

- Vi kan også formatere disse variablene:
  - sette minimum bredde
  - høyre-, og venstrejustere samt sentrere tekst
    - Strenger er høyresentrert som default, tall er venstrejuster som default
  - spesifisere desimaler i flyttall
  - vise både uttrykk og evaluering uten å gjenta uttrykket

```
x = 10
s = "abc"
y = 1.12345

print(f"**{x:5}**")   # **   10**
print(f"**{s:10}**")  # **abc       **
print(f"**{y:5}**")   # **1.12345** (y er mer enn 5 tegn)
print(f"**{x:05}**")  # **00010**

print(f"**{x:<5}**")  # **10   **
print(f"**{s:>5}**")  # **  abc**
print(f"**{s:^5}**")  # ** abc **

print(f"**{x:.3f}**")   # **10.000**
print(f"**{y:10.3f}**") # **     1.123**

print(f"{x = }")       # x = 10
print(f"{x + y = }")   # x + y = 11.12345
```

Flerdimensjonale lister

(2D-lister)

2D-lister - oppretting

- En flerdimensjonal liste er en liste hvor hvert element er en liste - en liste av lister!
- Statisk opprettelse: a = [[1, 2, 3], [4, 5, 6]]
- a[0] er nå listen [1, 2, 3]
- a[0][1] er element på indeks 1 i listen a[0] - altså 2

- Pass på! Reglene for alias og mutasjon gjelder fortsatt
- a = [[0] * 2] * 3 vil gi oss én unik liste og to aliaser
- Hver rad må opprettes så det blir et unikt element:

```
rows = 3
cols = 2

a = []
for _ in range(rows):
    a.append([0] * cols)

b = [[0] * cols for _ in range(rows)]
```

2D-lister - dimensjoner

- Det er ikke påkrevd, men det er vanlig å lage 2D-lister hvor alle innerste lister er like lang
  - *Rutenett*
- Ofte kan vi se på listen som rader og kolonner:
  - Hver av de innerste listene representerer én rad, og den ytterste listen holder alle rader

- Finne antall rader og kolonner i et rutenett:

```
a = [
    [1, 2, 3],
    [4, 5, 6]
]

rows = len(a)    # 2
cols = len(a[0]) # 3
```

2D-lister - løkker over 2D-lister

```python
# 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)
```

```
a = [
    [2, 3, 5],
    [1, 4, 7],
]
```

```
a = [
    [3, 4, 6],
    [2, 5, 8],
]
```

2D-lister - kopiering

- Alle måtene å kopiere som vi så på for lister vil gi **alias**
- **Kopierer vi en referanse, lager vi et alias**
- Her må vi gjøre noe som kalles *deepcopy*, hvor hvert enkelt element i hver enkelt liste i den todimensjonale listen kopieres

```
import copy

a = [
    [1, 2, 3],
    [4, 5, 6]
]

b = copy.deepcopy(a)
a[1] = [44, 55, 66]

print(a) # [[1, 2, 3], [44, 55, 66]]
print(b) # [[1, 2, 3], [4, 5, 6]]
```

2D-lister - oppgave

- Lag en funksjon column_sum(grid) som, Gitt en 2d liste (grid) med tall, returnerer en liste med summene av kolonnene.
- Hint:
  - Antall rader er len(grid)
  - Antall kolonner er len(grid[0])
  - Se på hva du gjorde i lab6 for å gå gjennom alle verdiene i et grid 🐍
  - *For hver kolonne*
    kolonnesum = ? #hva begynner vi på?
    For hver rad i kolonnen
    kolonnesum += 1

Eksempel input
-> output:

```
grid = [
    [5,2,8],
    [5,4,2],
    [1,2,3]
]

col_sum = [11,8,13]
```

Oppslagsverk (dict)

Dict - sette inn verdier

- Dictionary (ordbok)
- Et oppslagsverk!
  - Oversetter nøkler til verdier
  - Eksempel med pris:
    - “banan” -> 5
    - “servietter” -> 40

```
oppslagsverk[“nøkkel”] = “verdi”
```

Destruktiv!

```
print(oppslagsverk[“nøkkel”])
```

Ikke destruktiv.

```python
# Sette opp en dictionary
priser = dict() # eller priser = {}
priser["banan"] = 12
priser["eple"] = 2
priser["pære"] = 3

# Hente og skrive ut en verdi
print(priser["eple"])  # Output: 2

# Legge til en verdi eller endre hvis den finnes
priser["pære"] = 3
```

```python
priser = {"banan": 12,
          "eple": 2,
          "pære": 3
}
```

Dict - get

- dict.get(key, default_value)
  - default_value = returnert om key ikke finnes

- Hvorfor ikke bare gjøre sånn?:

```
priser = dict() # eller priser = {}
priser["banan"] = 12
priser["eple"] = 2
priser["pære"] = 3
```

- Om nøkkelen ikke finnes, kan vi få en bedre tilbakemelding, eller en standardverdi!

```
# Bruke dict.get() for å hente en verdi i stedet for priser["eple"]
print(priser.get("eple"))  # Output: 2
print(priser.get("appelsin"))  # Output: None
print(priser.get("appelsin", "Appelsin har vi ikke."))  # Output: Appelsin har vi ikke.
print(priser.get("appelsin", 100000))  # Output: 100000
```

Dict - iterere over

for key in dict.keys():

Itererer over nøklene i oppslagsverket

for key, item in dict.items():

Itererer over både nøkkel og verdi

for item in dict.values():

Itererer over verdier

```
18  # Iterere over en dictionary
19  for vare in priser:
20      print(vare, "koster", priser[vare])
21
22  # Iterere over en dictionary med items()
23  for vare, pris in priser.items():
24      print(vare, "koster", pris)
```

```
26  # Iterere over en dictionary med values()
27  for pris in priser.values():
28      print(vare)
```

Dict - oppgave

Skriv et program hvor:

- Du spør brukeren om noe de vil ha ved input
- Du bruker get(key, default_value) til å hente mengden av frukten i frukt
  - du skal få "ingen" hvis frukten ikke finnes. (default_value="ingen")
-

Eksempelkjøringer:

```
Hva vil du ha?: epler
Vi har 2 epler.
```

```
Hva vil du ha?: sjokolade
Vi har ingen sjokolade.
```

```
frukt = {
    "epler": 2,
    "bananer": 5,
    "appelsiner": 8
}
```

Mengder (Set)

Mengder

En mengde er en samling av verdier hvor man ikke holder styr på rekkefølgen til verdiene.

En mengde har ingen duplikate verdier.

I hovedsak kan man i en mengde:

- legge inn en verdi
- fjerne en verdi
- spørre om en verdi finnes
- Se gjennom verdiene i mengden
    - Rekkefølgen er ikke bestemt, så du kan ikke vite hvilke elementer som kommer først og sist!

Opprette en mengde

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

```python
# Opprett en mengde fra en liste (eller annen samling med elementer)
a = [2, 3, 3, 5]
s = set(a)
print(s) # {2, 3, 5}
```

```python
# Opprett med .add()
s = set()
s.add(2)
s.add(3)
s.add(5)
```

Destruktiv!

Grafiske brukergrensesnitt

GUI (Graphical User Interface)

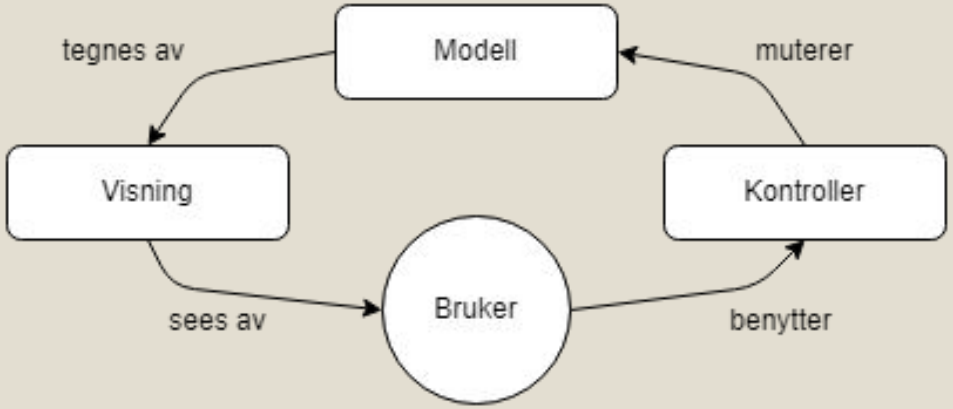
GUI

- Dataprogram som man kan interagere med uten fokus på terminalen.
- Kontrolleres oftest ved tastetrykk og mus.

Model-View-Controller

- Struktur for programflyt
- Deler:
  - Modell: Variabler for det som skal modelleres
  - Visning: Tegner basert på tilstanden til modellen
  - Kontroller: Oppdaterer modellen ved interaksjon fra bruker

Tilstand: De verdiene som variablene har i ett øyeblikk



Eksempel: Tastetrykk

app_started "modellerer" tastetrykkene med app.counter

I key_pressed muterer/endrer vi på app.counter

I redraw_all tegner vi det vi har modellert

Modell

```
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
```

Kontroller

```
def key_pressed(app, event):
    # key_pressed: kjøres hver gang en tast trykkes.
    # Vi kan endre variabler i `app` her.
    app.counter += 1
```

Visning

```
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'
    )
```

Filer og CSV

Åpne filer med 'with open()'

Åpner en fil som et filobjekt som vi gir et navn og kan manipulere

- 'r': "Read", åpner filen for lesing.
- 'w': "Write", åpner filen for skriving. Oppretter ny fil hvis den ikke eksisterer, ellers overskriver
- 'a': "Append", setter innholdet på slutten av filen, eller i en ny fil hvis den ikke eksisterer
- 'x': "Exclusive", Fungerer som "w" men krasjer om filen allerede finnes

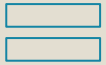
```
# Lese fra en fil
with open('minfil.txt', 'r', encoding='utf-8') as filobjekt:
    innhold = filobjekt.read()
```

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

Lese CSV

CSV = “Comma Separated Values”, en fil med tabell-data

people.csv



```
Navn,Alder,Høyde
Ola,20,1.80
Kari,19,1.65
Per,21,1.73
Oda,20,1.74
```

```
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()
```

```
# .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)
```

```
content_string =
“Navn,Alder,Høyde\nOla,20,1.80\n...”
```

```
content_lines =
[“Navn,Alder,Høyde”,”Ola,20,1.80”,...]
```

```
table =
[
    [“Navn”, “Alder”, “Høyde”],
    [“Ola”, “20”, “1.80”],
    ...
]
```

Filer - encoding

?

```
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()
```

Tekst i en fil er egentlig binærkoder!

- Men hva bestemmer hvilken binær-streng som skal representere hver bokstav?

  Ulike formater!

  - Unicode = Unicode Text Format = UTF
  - ASCII - "A" er 0100001
  - UTF-8 - "A" er 0100001
  - UTF-32 - "A" er 00000000000000001111101000000001101

- Skrive fil: Du kan velge det du trenger, men bruk som basis utf-8
- Lese fil: Du må bruke det formatet som filen du leser har

Krasj!

Programmet krasjer?

```
1     a = 0
2
3     print(b)
```

- Typiske feil er at noe er udefinert, index-out-of-range, feil argument til funksjoner
- Se på feilmeldingen!
  - filnavnet hvor feilen oppstod
  - linjen hvor feilen oppstod
  - hva slags feil det er

```
Traceback (most recent call last):
  File "c:\Users\Navn Navnesen\Dokumenter\INF100\\krasj.py", line 3, in <module>
    print(b)
          ^
NameError: name 'b' is not defined
```

Try-except

- Når vi vet hva slags feil som kan skje:
  - nyttig å fange den selv og håndtere på en hensynsfull måte, for eksempel en print() melding

Eksempel: Dele to tall på hverandre

krasj.py > ...

```
1  try:
2      num1 = int(input("Skriv inn et tall: "))
3      num2 = int(input("Skriv inn et annet tall: "))
4      result = num1 / num2
5      print("Resultatet er:", result)
6  except ZeroDivisionError:
7      print("Feil: Kan ikke dele på null.")
8  except:
9      print("Ukjent feil.")
```

```
Skriv inn et tall: 10
Skriv inn et annet tall: 5
Resultatet er 2.0
```

```
Skriv inn et tall: 8
Skriv inn et annet tall: 0
Feil: Kan ikke dele på null.
```

```
Skriv inn et tall: hei
Feil: Ukjent feil.
```

Moduler og pakker

Terminologi

- Modul: samling med relaterte funksjoner man kan importere og bruke i sin egen kode. Tenk på som en fil med kode.

- Pakke: En samling av moduler. Tenk på som en mappe med moduler.

- Bibliotek: En **stoooor** pakke. Er ikke noe teknisk forskjell fra pakke, men kalles et bibliotek når den er blitt "veldig stor".

- Rammeverk: Kan være modul/pakke/bibliotek. Hovedpoenget er at det setter "rammer" for hvordan koden skal skrives, for eksempel vår uib_inf100_graphics

Hvordan importere?

- import a
  - importer en hel modul
  - for å bruke modulen må du skrive a.foo()

```
import math
print(math.pi)
```

- from a import b
  - importer funksjon/verdi b fra modul a

```
from math import pi
print(pi)
```

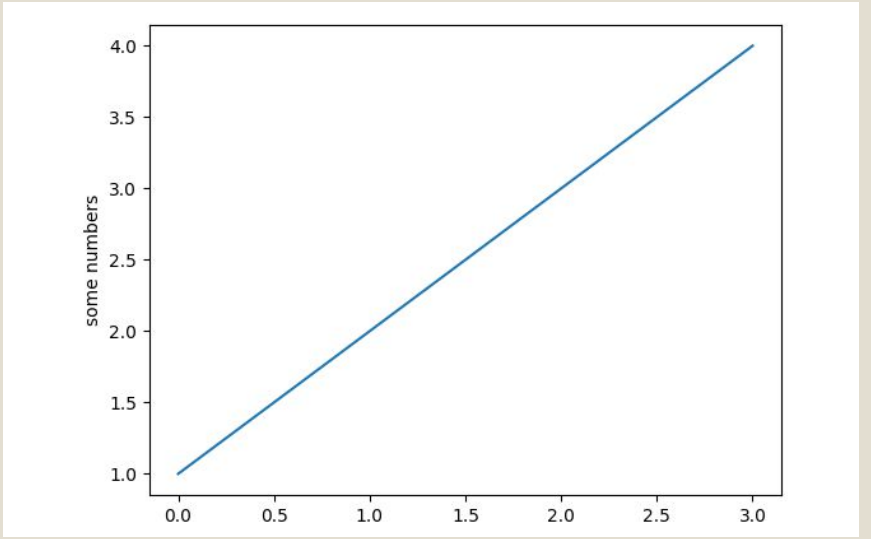
- from a import *
  - importer ALT fra a
    - **vær forsiktig!** Du har tilgang til alle funksjoner, men da kan dette også skape forvirring mellom dine egne funksjoner og importerte

Matplotlib

Matplotlib

- Et bibliotek for å tegne grafer!
  - plt.plot() for å sette verdiene
  - plt.ylabel()/plt.xlabel() setter navn på aksene
  - plt.show() viser den ferdige grafen!

```
import matplotlib.pyplot as plt

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

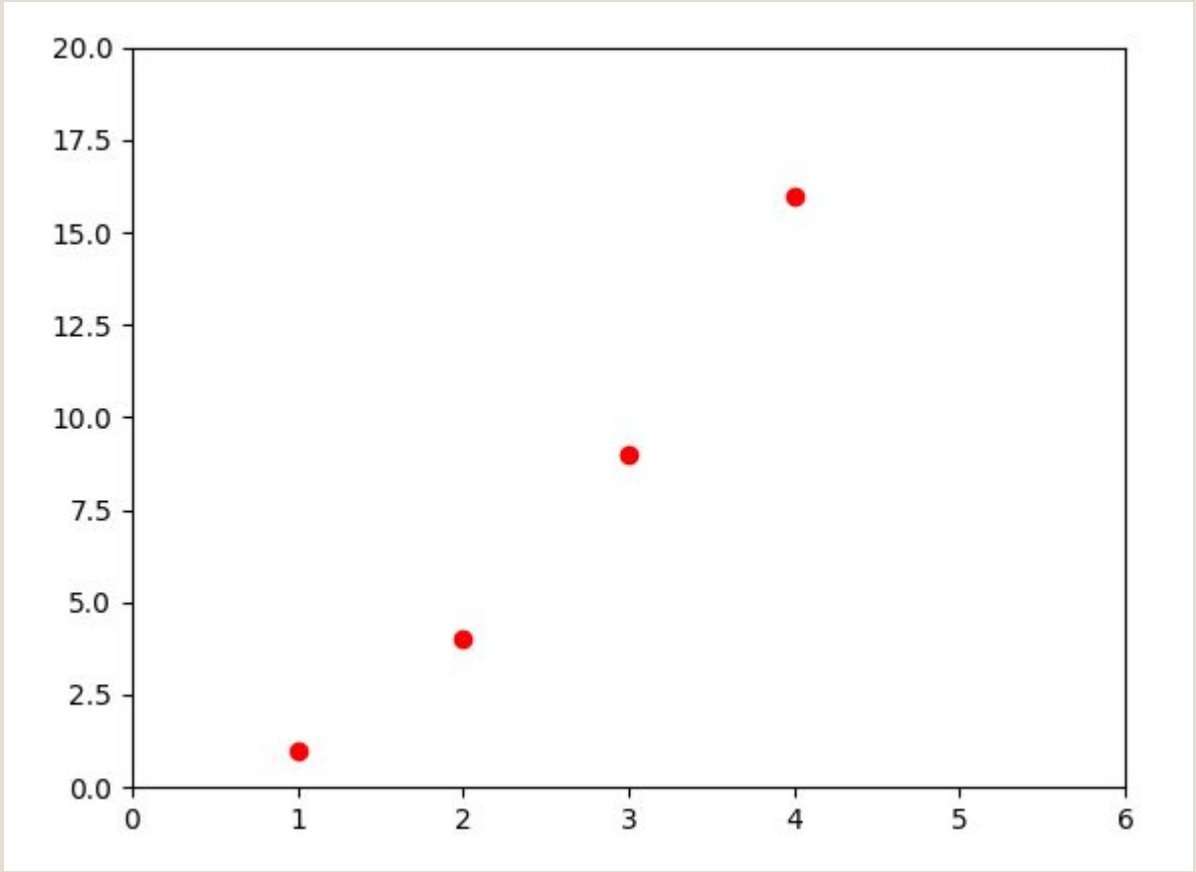


Matplotlib

Eksempel med:

- verdier på begge akser
- spesifisert farge og form på punkter

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



Spørsmål?

"Hva er en variabel?" er ok!

Lykke til!