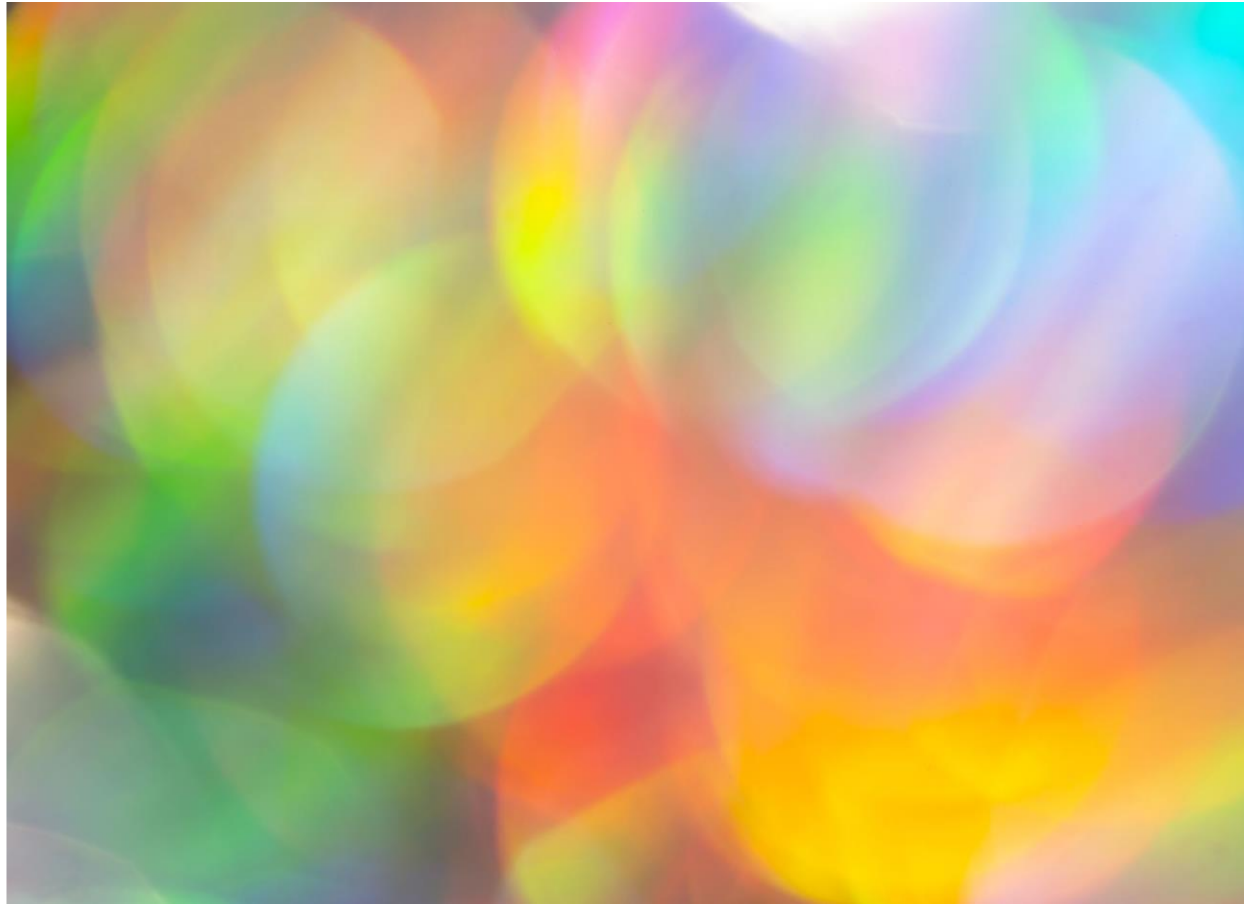


REPETISJONS-FORELESNING

IN1900, høst 2023

OM REPETISJONSFORELESNINGEN

- En **rask** oversikt over pensum fra et annet perspektiv
- (Se gjerne over lysarkene senere i ro og mak)
- **Ikke** en fullstendig oversikt – det er garantert detaljer som ikke kommer med
- Men skal dekke det mest sentrale!

PROGRAMMER BESTÅR AV KODELINJER (STATEMENTS)

- En og en linje kjøres av gangen, ovenfra og ned
- Unntak:
  - Kan hoppe over linjer (if/else)
  - Kan hoppe tilbake og repetere linjer (while/for)
  - Kan midlertidig gå til et helt annet sted i programmet (def) for så å returnere senere
- Å kunne vite hvilke kodelinjer som kjøres når er nyttig for å forstå og finne feil i programkode

TILORDNING: [VARIABEL] = [VERDI]

- For å kunne gjenbruke verdier (og enkelt endre dem senere), bruker vi variabler
- Variabelnavn: Ikke viktig for maskinen, men viktig for mennesker som leser koden
  - unntak 1: reserverte navn (*while, for, if…*)
  - unntak 2: utilsiktet overskriving av variabel (ikke bruk *c* til både lyshastighet og abc-formelen hvis du trenger begge verdiene senere…)
- Etter en tilordning vet Python hvor i minnet til maskinen verdien (objektet) til variablelen ligger, og henter frem verdien når variabelen brukes

EVALUERING AV UTTRYKK

- Verdien til en variabel kan også være resultatet av en utregning:
  *x = -b + sqrt(b**2 – 4*a*c) / 2*a*
- Regnerekkefølge som forventet fra matematikken (* / før + -)
- Bruk parenteser der det trengs eller du er usikker
  (husk at en brøkstrek på papir impliserer parenteser rundt teller og nevner)
  *x = (-b + sqrt(b**2 – 4*a*c)) / (2*a)*

VARIABLER HAR TYPER (SOM OPPFØRER SEG FORSKJELLIG)

- int: 4 + 4 = 8
- float: 4.0 + 4.0 = 8.0000000000000001
- str: “4” + “4” = “44”
- list: [4] + [4] = [4,4]
- De to nederste er eksempler på *samlinger* med flere *elementer*
- *Operatorer* (+ - * / ** == > >= <= < !=) vil ofte gjøre forskjellige ting med forskjellige typer (som vi ser eksempler på med + over)
- Vi kan sjekke hvilken type (dvs. klasse) en verdi (dvs. et objekt) har: *print(type(x))*

KONVERTERING MELLOM TYPER

- int(4.88) → 4 float → int
- round(4.88) → 5 float → int
  - round(4.88, 1) → 4.9 float → float
- str(4.88) → "4.88" float → str

Implisitt konvertering (som skjer uten at vi eksplisitt ber om det):

- 4/2 → 2.0 int/int → float/float → float
- True + True → 2 bool + bool → int + int → int

UTSKRIFT

1	svar = 42.001
2	print(“Svaret er”, svar)
3	print(“Svaret er “ + svar)
4	print(f“Svaret er {svar}”)
5	print(f“Svaret er {svar:.2f}”)

Linje 2, 3 og 4 gir samme resultat.

Linje 5 lar oss kontrollere antall desimaler uten å endre verdien med *round*

IMPORTERE FUNKSJONER/KLASSER FRA PAKKER:

- **import math** → nå kan du bruke *math.sqrt(4)*, men ikke *sqrt(4)*
- **from math import sqrt** → nå kan du bruke *sqrt(4)*, men ikke *math.sqrt(4)*
- **from math import ***
  er en dårlig ide
- Flere pakker kan ha klasser/funksjoner som er ulike, men med samme navn – disse bør ikke forveksles!
  (eksempel: *math.sqrt* og *numpy.sqrt*)

TESTER (WHILE / IF / ELIF)

- En test vil alltid lykkes (True) eller feile (False)
- Operatorer som sammenligner verdier (objekter): == != < <= >= >
- Tester kan kombineres fra del-tester: *and, or*
- Motsatt verdi: *not*

IF / ELSE / ELIF

- *if <test 1>:*
  - dette skjer hvis test 1 evaluerte til True
- *elif <test 2>:* # kjøres bare hvis test 1 feilet
  - dette skjer hvis test 2 kjørte og evaluerte til True
- *else:*
  - dette skjer bare hvis *alle* testene i denne if-elif-else-blokken (test 1 og 2) feilet

LØKKER ER DINE VENNER

- Med repetisjon fra løkker får vi kortere, mer oversiktlige programmer
- Lettere å endre ting *ett* sted enn 100 steder
- Kan repetere identiske kodelinjer, eller med noe variasjon hver gang
- While-løkker kan brukes i alle tilfeller med repetisjon
- For-løkker er enklere, men kan ikke alltid brukes
- *Innholdet* i en løkke (det som repeteres) kan være alt mulig, også nye løkker (det vi kaller en *nøstet* løkke / nested loop)

WHILE-LØKKER

- Vi kan tenke på while-løkker som en if-test som kjører om og om igjen til den endelig feiler
- (Litt som en 4-åring: "Er vi framme nå? Enn nå da? Hva med *nå* da?")
- Vi trenger ikke vite på forhånd hvor mange repetisjoner det blir – en while-løkke prøver bare til det ikke går lenger
- Ulempe: Lett å lage en *uendelig* while-løkke hvis testen *aldri* feiler
- Hvis du bruker en variabel i testen (while-setningen), husk å oppdatere denne variabelen – hvis den aldri oppdateres vil løkken enten aldri kjøre (testen feiler hver gang) eller kjøre uendelig (testen lykkes hver gang)

FOR-LØKKER

- For-løkker går gjennom en *samling* elementer fra start til slutt
- Samlinger: list, dict, str, range, …
- Stopper automatisk når vi har gått gjennom hele samlingen
- For-løkken vet i utgangspunktet hvor mange repetisjoner det blir (= samlingens antall elementer) med mindre vi endrer samlingen underveis (skummelt….)
- Ulempe: Kan ikke håndtere alle tilfeller
  (repetere til brukeren skriver gyldig input, repetere til klokka er 4, …)

LISTER

- En samling elementer (av hvilken som helst type – til og med nye lister!)
- Alltid i en bestemt rekkefølge (posisjon i listen angis med en indeks):
- Første element: *minliste[0]*
- Andre element: *minliste[1]*
- Eventuelt telle bakover fra slutten:
- Siste element: *minliste[-1]*
- Nest siste element: *minliste[-2]*
- *minliste.append(element)* legger til verdien av variabelen *element* helt sist i lista!

LØKKER OG LISTER

- *for element in liste* gjør at løkkevariabelen *element* får verdier etter tur:
  - første gang: *element = liste[0]*
  - andre gang: *element = liste[1]*
  - osv.
- *for i in range(len(liste))* gjør at løkkevariabelen *i* får verdiene til indeksene:
  - første gang: *i = 0*
  - andre gang: *i = 1*
- I det siste tilfellet kan vi hente elementer slik: *element = liste[i]*
- OBS: *element = element + 2* endrer <u>ikke</u> verdien som ligger i *liste*

HÆ, HVORFOR IKKE? (REFERANSER TIL VERDIER/OBJEKTER)

- *liste = [1, 2, 3, 4]*
- *element = liste[2]*
  element peker på adressen *001234* i minnet (der verdien 3 ligger)
  liste[2] peker på adressen *001234* i minnet (der verdien 3 ligger)
- *element = element + 2*
  element peker på adressen *001236* i minnet (der verdien 5 ligger)
  liste[2] peker fortsatt på adressen *001234* i minnet (der verdien 3 ligger)
- Selv om vi ga *element* en ny verdi, gjorde vi ingen endringer på variabelen *liste*!
- *liste[2] = liste[2]* + 2 ville derimot endret *liste* (men ikke *element*)!

REFERANSE VS. KOPI

- *a = [1, 2, 3]* *liste på adressen 05678*
  b = a *liste på adressen 05678*

  To variabler som begge peker på det samme objektet (den samme listen)

- *a = [1, 2, 3]* *liste på adressen 05678*
  b = a[:] *liste på adressen 05679*

  To variabler som peker på hvert sitt objekt (hver sin liste), med samme elementer

- *a[0] = 4* vil endre *b* i det øverste tilfellet, men ikke i det nederste!

RANGE OG SLICE

- *range(start, stop, step)* gir en liste med indekser (heltall)
- *liste[start:stop:step]* gir en bit (slice) av en liste
- **start:** fra og med
- **stop:** til, men ikke med
- **step:** steglengde til neste indeks
- spesialtilfelle: *range(lengde)* gir indeksene til en liste av denne lengden (dvs. *lengde* = *stop*)

LIST OG STR

- Likheter
  - Begge kan indekseres: *"Hei"[2]* → *"i"*
  - Slicing fungerer på begge: *"Hei"[:2]* → *"He"*
  - En **for**-løkke går gjennom alle elementene: **for** *bokstav* **in** *ord*
    - elementene i en streng er tegn (mellomrom er også tegn)
- Ulikheter
  - Du kan endre en liste, men ikke en streng: *"Hei"[2] = "y" # gir feilmelding*
  - En streng kan være nøkkel i en ordbok (mer om dette senere)

LIST COMPREHENSIONS (SNARVEI)

- Ofte brukt spesialtilfelle: For hvert element i en liste, gjør noe med dette elementet og putt svaret i en ny liste
- *ny_liste* = *[uttrykk* **for** *element* **in** *gammel_liste]*
- Eksempel: *kvadrattall* = *[tall***2 **for** *tall* **in** *naturlige_tall]*

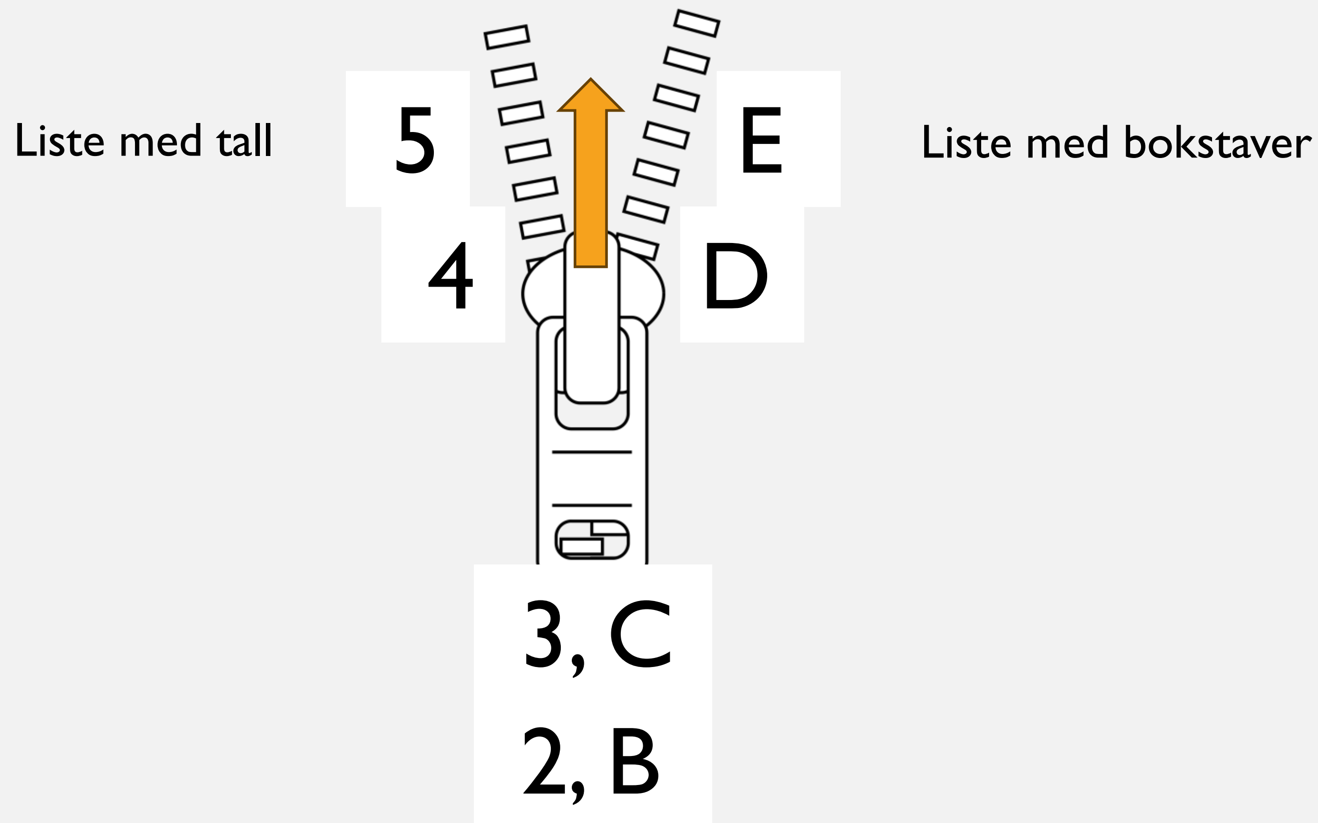
NØSTEDE LISTER (LISTER INNI LISTER)

- *ytre_liste = [[[1], [2, 3]], [[4], [5, 6]], [[7], [8, 9]]]*
- går fint å ta ett nivå av gangen:
  - *indre_liste = ytre_liste[2]* *[[7], [8, 9]]*
  - *innerste_liste = indre_liste[1]* *[8, 9]*
  - *tall = innerste_liste[0]* *8*
- eller alt på en gang (ytterst → innerst):
  - *tall = ytre_liste[2][1][0]* *8*

TUPLER

- Oppfører seg ganske likt lister, men garanterer at verdier ikke endres senere i programmet (kan ikke forlenges eller forkortes heller)
- *min_tupel = 1, 2, 3*
- Splitte opp en tupel i flere variabler: *a, b, c = 1, 2, 3*

ZIP: SNARVEI FOR Å SLÅ SAMMEN TO (ELLER FLERE) LISTER



zip lager en liste med *tupler* (her med tall og bokstav)

ARRAYS (NUMPY): HVA ER ULIKT LISTER?

- Arrays kan bare ha en bestemt type verdier (lister kan blande mange typer)
- Vi kan regne med en hel array av gangen: *array_c* = *array_a* + *array_b* gjør at operasjonen (+) gjøres på **hvert enkelt element**
  - Lister *må* vi bruke løkker for å gjøre dette med
- Funksjoner laget for tall fungerer vanligvis for arrays også!
  - (spesielt *numpy*-funksjoner, *math*-funksjoner er det litt verre med)
  - dette gjelder definitivt ikke for lister!
- Arrays kan indekseres [2, 1, 0] der lister i lister må bruke [2][1][0]
- Arrays er raskere (spesialisert for vektoriserte utregninger)

LINSPACE VS. RANGE

- range kun heltall:
  - *range(0, 3, 1)* gir tallene 0 1 2 — til, men ikke med, 3
- linspace også desimaltall:
  - *linspace(0, 2, 5)* gir tallene 0.0 0.5 1.0 1.5 2.0 — til og med 2
- Hvor mange tall blir det?
  - *range(start, stop, step)* → (stop – start) / step — (her: 3-0/1 = 3)
  - *linspace(first, last, n)* → n
- Spesialtilfelle: *range(n)* → *n*

ORDBØKER (DICT)

- Lister har alltid heltalls-indekser som starter på 0:
- Ordbøker har isteden *nøkler* som kan være (nesten) hva som helst:
  - ikke lister
  - ikke ordbøker
  - ikke egendefinerte klasser (i utgangspunktet)
- men alle disse kan være *verdier* i en ordbok
- *element = liste[indeks]*
- *verdi = ordbok[nøkkel]*

list

0	1	2
"a"	"b"	"c"

dict

1	"a"
42	"b"
1337	"c"

dict

"a"	1
"b"	42
"c"	1337

“FINNES DETTE I SAMLINGEN?”

- *“a”* **in** *liste* sjekker om *elementet “a”* finnes i listen
- *“a”* **in** *ordbok* sjekker om *nøkkelen “a”* finnes i ordboken
- hva om man vil lete etter en verdi isteden?
  - *“a”* **in** *ordbok.values()*

list

0	1	2
"a"	"b"	"c"

dict

1	"a"
42	"b"
1337	"c"

dict

"a"	1
"b"	42
"c"	1337

TESTING

- “Tradisjonell”: print alle mulige verdier til skjermen (litt vel enkelt)
- *assert <test>*
- Lager en feilmelding hvis testen feiler (False)
- Ikke så eksamensrelevant, men veldig lurt i den virkelige verden:
  Bruke en debugger (Python Tutor, VS Code) til å gå gjennom programmet linje for linje og ha oversikt over variabler og verdier hele tiden

HARDKODING OG INPUT

- Hardkodete verdier til variabler i programmet gjør at brukeren må kunne Python og endre selve programmet hver gang noe skal endres
- Bedre om programmet kan be brukeren skrive inn verdier selv i slike tilfeller
- Husk at *input* returnerer tekst (str):
  summen = input("Gi meg et tall: ") + input("Gi meg enda et tall:")
- Skriver jeg inn 4 og 2 får *summen* verdien *"42"*
- *summen = float(input("Gi meg et tall: ")) + float(input("Gi meg enda et tall:"))*
  (gir summen 8)

DATA FRA TEKSTFIL

- Åpne filen som et fil-objekt (*open*)
- **for** *linje* **in** *fil:*
  - *linje* er nå en streng med tekst (logisk nok siden det er en tekstfil) som du kan gjøre det du ønsker med
  - for eksempel: *ordene = linje.split()* *# standard skilletegn er " ", .split(",") for komma*
  - *for ord in ordene:*
    - …
- Lukk filen til slutt (*close*)

NYTTIGE STRENG-OPERASJONER

- .split() # lang streng → liste med korte strenger
- .join() # liste med korte strenger → lang streng
- .replace(…) # bytt ut en substreng med en annen
- .isdigit() # strengen inneholder bare tall
- .isalpha() # strengen inneholder bare bokstaver
- .upper() # sørger for bare store bokstaver
- .lower() # sørger for bare små bokstaver

PLOTTING

- Greit å kunne gjøre det grunnleggende nærmest i søvne
- Husk at du kan kalle *plot* flere ganger før du kaller *show* (får dermed flere kurver i samme vindu)
- Ikke glem *title* og *legend*
  - Bruk *label*="…" i *plot* slik at *legend* har noe å vise

FUNKSJONER

- Forenkler programmet ved at en kompleks operasjon kan skrives på en linje (der vi kaller funksjonen) uten at alle detaljene er midt i programmet
- Funksjoner kan også gjenbrukes – i samme program eller andre programmer
- Funksjoner er et lite "program i programmet" med sine helt egne lokale variabler som moder-programmet ikke har tilgang til
- For å sende informasjon inn til funksjonen bruker vi **parametre**
- Hver gang vi kaller funksjonen kan vi gi parametre forskjellige verdier (**argumenter**) som inndata – parametre er lokale variabler som får verdier når funksjonen kalles
- Parametre kan ha standardverdier (default) hvis vi ikke gir dem noen verdi
- For å sende informasjon tilbake til programmet etterpå bruker vi **returverdier**:
  return en_verdi
  return verdi1, verdi2, verdi3

Eksempel: parametre

```
def f(x,y):
  return x+y

a = 0
b = 1
verdi = f(a,b)
```

argumenter

FUNKSJONER ER OBJEKTER DE OGSÅ

- funksjonsnavn er egentlig bare variabelnavn – verdien er en funksjon
- praktisk bruksområde: en funksjon kan derfor være argument til en annen funksjon – gir den ene funksjonen beskjed om hvilken annen funksjon den skal bruke
- dette betyr også at vi kan overskrive (og miste tilgang til) funksjoner:
  print = 2   *# oops!*
  print("Hei")   *# gir nå feilmelding*
- *sum* og *list* er heller ikke anbefalte variabelnavn… lett å gå i fella her!

OBJEKTERS FUNKSJONER (METODER)

- alenestående funksjon: *print("Hei")*
  - ett argument (*"Hei"*)
- funksjon som tilhører et objekt: *liste.append("Hei")*
  - to agumenter (en liste og *"Hei"*)
- objektet bruker seg selv som argument!
  - dette blir verdien til parameteren *self*
- Dette for at vi mennesker skal kunne tenke at objektet er litt intelligent og selv "vet" hvordan det skal gjøre enkelte ting – objektorientert måte å tenke på
  - "lister vet hvordan de skal legge til nye elementer i seg selv"

KLASSE VS. OBJEKT

- Klassen er oppskriften
- Der ligger alle metodene (objektenes funksjoner) definert
- Alle objekter av samme type har dette felles

- Et objekt er et resultat av å bruke oppskriften
- Objekter kan være forskjellige fordi *instansvariablene* til objektene kan ha forskjellige verdier

- Eksempel: Alle lister kan bruke *.append()*, men ulike lister har ulike lengder

INSTANSVARIABLER

- En variabel som har forskjellig verdi for forskjellige objekter
- Altså *ikke* en felles verdi for alle objekter av denne typen
- Eksempel:
  jorden.tyngdeakselerasjon kan ha verdien 9.81
  mars.tyngdeakselerasjon kan ha verdien 3.72
- Men hvordan skal metodene (som er felles) forholde seg til disse variablene? Alle objektene har jo forskjellige navn. Hvordan skal oppskriften referere til "dette spesifikke objektet som metoden ble kalt på"?
- spesialparameteren **self** (som har objektet selv som verdi)
- **def** *en_metode(self, en_annen_parameter):*
- *et_objekt.en_metode(42)*

LOKAL VARIABEL VS. INSTANSVARIABEL

- Husk at funksjoner er små "programmer i programmet"
- En variabel definert inne i en funksjon forsvinner når funksjonen har kjørt ferdig:

```
def en_funksjon():
    snart_borte = True
```

- En instansvariabel (i en klasse) forsvinner ikke så lenge objektet eksisterer:

```
def en_metode(self):
    self.lever_videre = True
```

- Hvis vi vil ta vare på en verdi som oppsto inne i en vanlig funksjon (som ikke tilhører en klasse), må vi sende verdien tilbake med **return**, eller er den borte
- (Hvis en funksjon kaller en funksjon kaller enda en funksjon, kan vi måtte bruke den samme returverdien i alle disse funksjonene helt til den er framme)

KONSTRUKTØREN

- "Magisk" metode som vi aldri kaller direkte
- Den kalles automatisk med en gang objektet er laget, og klargjør det for bruk
- Kan for eksempel sette verdier på instansvariabler:

```
def __init__(self, a, b):
        self.a = a
        self.b = b
```

- Her er *a* en parameter og *self.a* en instansvariabel (to forskjellige variabler selv om de her får samme verdi)
- Når vi lager objektet kan vi gi parametrene verdier:

```
so = SuperdupertObjekt(x, y)
```

- *so.a* vil nå ha samme verdi som *x*, og *so.b* vil ha samme verdi som *y*

FLERE MAGISKE METODER

__call__

- "Magisk" metode som vi aldri kaller direkte
- Kalles automatisk når vi kaller *et_objekt(…)* som en funksjon
- Hvis ikke *__call__* er definert, gir dette feilmelding

__str__

- "Magisk" metode som vi aldri kaller direkte (generelt: *__metodenavn__*)
- Kalles automatisk når vi skriver *str(et_objekt)* eller *print(et_objekt)*
- Hvis ikke *__str__* er definert, får vi typen og adressen til objektet: *<__main__.MinKlasse object at 0x000002143793A4C0>*

ENDA FLERE MAGISKE METODER

__add__	+
__sub__	-
__eq__	==
__gt__	>

Derfor oppfører + seg forskjellig for *str* og *int* – man har skrevet *__add__*-metoden forskjellig for de to klassene

ARV

- La oss si at vi vil skrive en klasse som gjør nesten det samme som en annen klasse vi har fra før
  - Eksempel: Vi vil ha en spesiell type liste som kan summere hvert enkelt element med + slik arrays gjør, men ellers er helt lik en vanlig liste
- Å finne opp kruttet på nytt kan være gøy, men ikke nødvendig
-
```
class SpecialList(list):
    def __add__(self, other):
```
- Nå vil + mellom to *SpecialList*-objekter oppføre seg som mellom to arrays (vi har *overskrevet* metoden *__add__* med vår egen kode)
- Alt annet i *list* trenger vi ikke tenke på, det arves automatisk fra *list*
- (Vi kan også legge til helt nye metoder, instansvariabler…)

NÅR MANGE KLASSER VI LAGER HAR MYE TIL FELLES…

- …er det lurt å samle det som er felles i en klasse…
- …og lage sub-klasser som arver fra denne klassen for å håndtere hvert enkelt tilfelle (dette er klassene vi kommer til å bruke i praksis)

- Klasser (i likhet med funksjoner) er et verktøy for å håndtere kompleksitet og lar oss samle kode på ett sted slik at det kan gjenbrukes mange steder
- Eksempel: ODE-algoritmer har mye felles (men en av metodene blir forskjellig for hver algoritme)

NUMERISK LØSNING AV ODE

- Difflikning: $u'(t) = f(t, u(t))$
- → Differenslikning: $u'(t) \rightarrow (u(t+h) - u(t))/h$
- Dette er en måte (av flere) å dele opp en kontinuerlig funksjon i små steg (h) som gjør en numerisk løsning på datamaskin mulig
- Trenger også en startverdi, for eksempel $u(0)$
- Resultatet blir arrays med verdier som tilnærmer funksjonen og kan plottes
  - En array for t
  - En array for $u(t)$

FORWARD EULER

- *u[0]* og *t[0]* gitt
- *t[n+1] = t[n] + h*   *# steglengden h kan også kalles dt*
- *u[n+1] = u[n] + h*f(t[n], u[n])*

- Ikke den raskeste algoritmen for ODE'er (målt i antall steg)
- Finner heller ikke *alltid* svaret (numerisk ustabil)
- Det finnes mer effektive og pålitelige algoritmer
- Men når formålet er å bygge opp forståelse er denne ganske intuitiv

HVILKEN STEGLENGDE?

- Hvis h er for stor, blir svaret unøyaktig
  - Eller mer og mer feil for hvert steg hvis algoritmen er numerisk ustabil
- Hvis h er for liten, blir *avrundingsfeilen* i maskinen mye større enn tallene vi jobber med
- Finne en steglengde som balanserer disse ytterpunktene:
  - matematisk unøyaktighet (i tilnærmingen) så liten som mulig
  - men ikke så liten at vi får avrundingsproblemer
- Moral: Mindre steg er ikke alltid bedre
  (se for eksempel MAT-INF 1100 for detaljer)

VIKTIG Å KJENNE TIL

- Hvordan en enkel ODE-løser (laget med en klasse) fungerer
- Hvordan man bruker en slik ODE-løser i et program