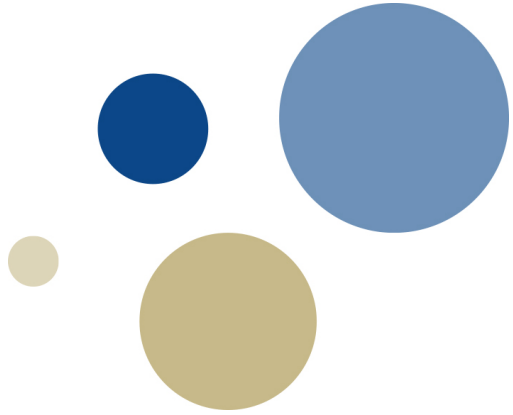
NTNU

Kunnskap for en bedre verden



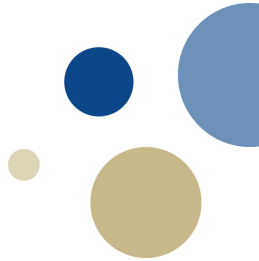
Kræsjkurs

Python

Martin Eek Gerhardsen
Institutt for Teknisk Kybernetikk, NTNU

2. desember 2019

Plan

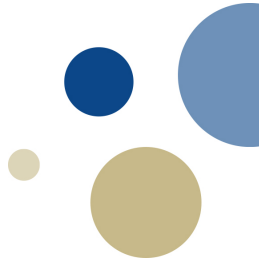


- Grunnleggende
- Visualisering
- Gjennomgang av gamle eksamensoppgaver

Plan

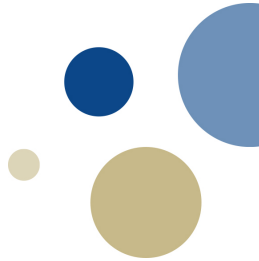
1. Tilordningsoperator
2. Aritmetiske operatorer
3. Sammenligningsoperatorer
4. Datatyper
5. Input og output fra bruker
6. Funksjoner
7. Lister og tupler
8. Løkker
9. Slicing
10. Dictionaries og mengder
11. Input og output fra filer
12. Unntaksbehandling
13. Eksamensoppgaver

Tilordningsoperatoren



Gitt to variabler *a* og *b*, bytt om innholdet.
Tilordningsoperatoren =
Original verdi blir overskrevet.

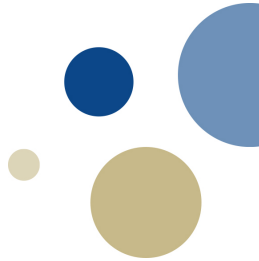
Tilordningsoperatoren



```
a = "hei"
b = 3.14
a = b
b = a
```

	1	2	3	4
a				
b				

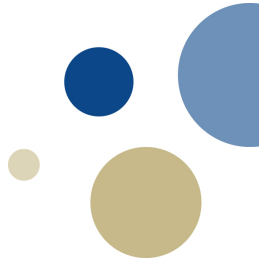
Tilordningsoperatoren



```
a = "hei"
b = 3.14
a = b
b = a
```

	1	2	3	4
a	"hei"			
b	?			

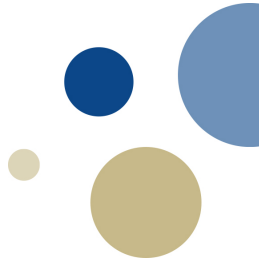
Tilordningsoperatoren



```
a = "hei"
b = 3.14
a = b
b = a
```

	1	2	3	4
a	"hei"	"hei"		
b	?	3.14		

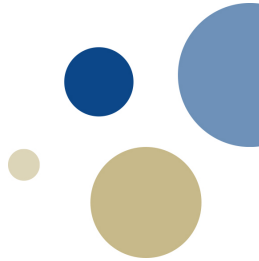
Tilordningsoperatoren



```
a = "hei"
b = 3.14
a = b
b = a
```

	1	2	3	4
a	"hei"	"hei"	3.14	
b	?	3.14	3.14	

Tilordningsoperatoren

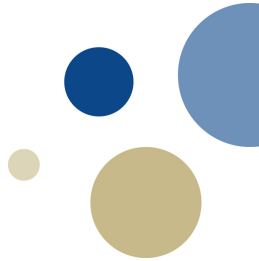


Problem: Gammel verdi blir overskrevet. Vi kan bare endre en ting av gangen.

```
a = "hei"
b = 3.14
a = b
b = a
```

	1	2	3	4
a	"hei"	"hei"	3.14	3.14
b	?	3.14	3.14	3.14

Tilordningsoperatoren

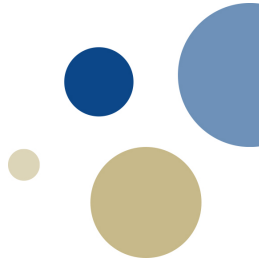


To vanlige løsninger

```
a = "hei"
b = 3.14
old_a = a    # ofte kalt temporary / temp
a = b
b = old_a
```

	1	2	3	4	5
a	"hei"	"hei"	"hei"	3.14	3.14
b	?	3.14	3.14	3.14	"hei"
a_old	?	?	"hei"	"hei"	"hei"

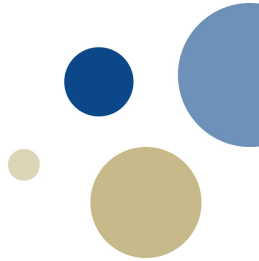
Tilordningsoperatoren



To vanlige løsninger

```
a = "hei"
b = 3.14
(b, a) = (a, b)
# bruker tupler
# kommer tilbake til dette senere
```

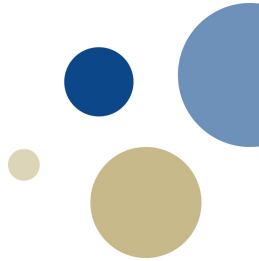
Tilordningsoperatoren



Flere tilordninger etter hverandre

```
a = b = c = 3.14
```

Tilordningsoperatoren



Flere tilordninger etter hverandre

Triks: Parenteser.

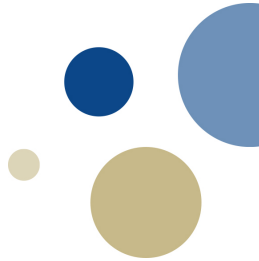
```
a = (b = (c = 3.14))
# ikke riktig syntaks, kun for illustrere
```

Flere tilordningsoperatorer

Finnes flere tilordningsoperatorer.
VIKTIG: Bortsett fra standard tilordning, må variabelen som tilordnes har en verdi fra før av.

Operator	Eksempel	Det samme som
=	a = 1	a = 1
-=	a -= 1	a = a - 1
/=	a /= 1	a = a / 1
//=	a //= 1	a = a // 1
+=	a += 1	a = a + 1
*=	a *= 1	a = a * 1
%=	a %= 1	a = a % 1
**=	a **= 1	a = a**1

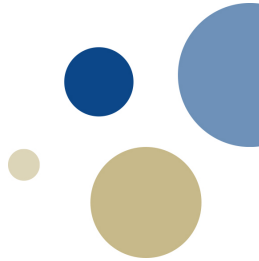
Aritmetiske operatorer



Viktig: Holde tunga rett i munnen med //, % og **

Operator	Eksempel
+	1 + 2
-	1 - 2
/	1 / 2
//	1 // 2
*	1 * 2
%	1 % 2
**	1**2

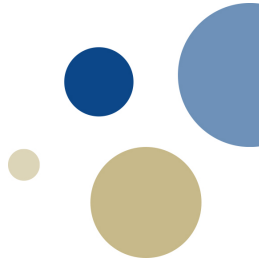
Aritmetiske operatorer



Hva skjer her?

```
from math import sqrt
a = sqrt(2)
b = 2**(1/2)

print(a == b)
```

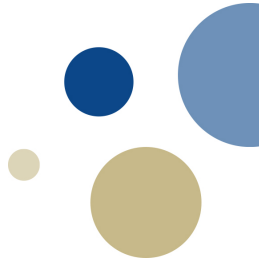
Aritmetiske operatorer



Hva skjer her?
True printes. Hvorfor trenger man da *sqrt* funksjonen?

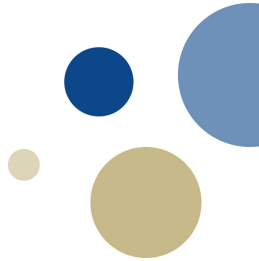
```
from math import sqrt
a = sqrt(2)
b = 2**(1/2)

print(a == b)
```

Sammenligningsoperatorer



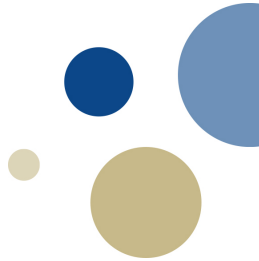
Sammenligningsoperatorer og betingede hendelser er kanskje det viktigste med datamaskiner. Det er nødvendig for å gjøre en datamaskin *Turing-complete*.

Sammenligningsoperatorer



Boolske variabler er essensielle for sammenligningsoperatorer. En boolsk variabel kan kun være to verdier, *True* eller *False*.

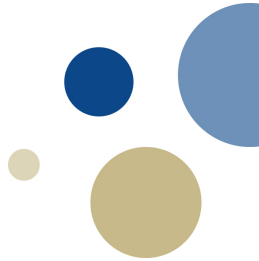
Sammenligningsoperatorer



Sammenligningsoperatorer og logiske operatorer.

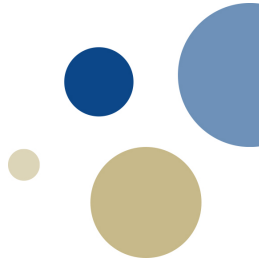
Operator	Eksempel	Resultat
<	2 <2	False
>	2 >2	False
<=	2 <= 2	True
>=	2 >= 2	True
==	2 == 2	True
!=	2 != 2	False
and	True and False	False
or	True or False	True
not	not False	True
in	1 in [3, "hei", 1]	True
not in	"hade" not in [3, "hei", 1]	True

Betingelser



if, *elif* og *else* brukes for å kontrollere programflyten. Man kan dermed aktivere en utvalgt del av koden ved å sjekke boolske verdier.

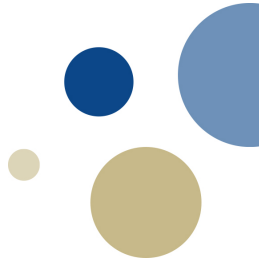
Betingelser



if, *elif* og *else* brukes for å kontrollere programflyten.
Man kan dermed aktivere en utvalgt del av koden ved å sjekke boolske verdier.

```
num = 1
if num == 0:
    print("num er 0")
elif num == 1:
    print("num er 1")
else:
    print("num er noe annet")
```

Betingelser



if, *elif* og *else* brukes for å kontrollere programflyten.
Man kan kjede så mange *elif*-er etter hverandre.

```
num = 1
if num == 0:
    print("num er 0")
elif num == 1:
    print("num er 1")
elif num == 2:
    print("num er 2")
elif num == 3:
    print("num er 3")
else:
    print("num er noe annet")
```

Betingelser

if, *elif* og *else* brukes for å kontrollere programflyten.
Man kan kjede så mange *elif*-er etter hverandre.
Viktig: Kun en av kode-snuttene vil kjøre mellom en *if* og *else*!

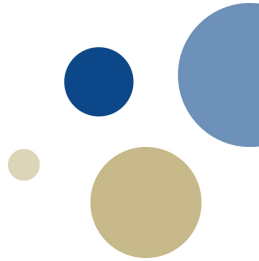
```
num = 1
if num == 0:
    print("num er 0")
elif num == 1:
    print("num er 1")
elif num == 2:
    print("num er 2")
elif num == 3:
    print("num er 3")
else:
    print("num er noe annet")
```

Betingelser

if, *elif* og *else* brukes for å kontrollere programflyten.
Man kan kjede så mange *elif*-er etter hverandre.
Viktig: Kun en av kode-snuttene vil kjøre mellom en *if* og *else*!
Her er det nye *if*-er, og dermed kan flere aktiveres samtidig.

```
num = 1
if num < 0:
    print("num mindre enn 0")
if num < 1:
    print("num mindre enn 1")
if num < 2:
    print("num mindre enn 2")
if num < 3:
    print("num mindre enn 3")
else:
    print("num er noe annet")
```

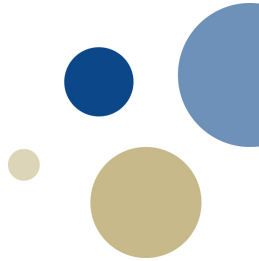
Datatyper



Variabler har bestemte typer, men kan byttes med tilordningsoperatoren.

```
a = "hei" # type er str
a = 1 # type er int
```

Datatyper



Noen aritmetiske operatorer fungerer for andre typer enn tall, men ikke alltid.

```
liste1 = [1, 2, 3]
liste2 = [4, 5]
liste3 = liste1 + liste2

liste4 = liste3 * 2
```

Datatyper

Eksempler på datatyper:

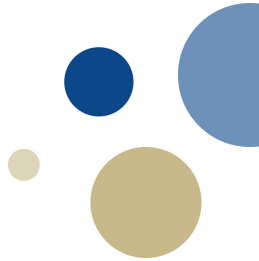
- int: heltall, 1023913213, 0, -1
- float: flyttall, 0.1, 1.2e-10, 3.1415e6
- str: streng, "hei", "sacasdcasdc"
- list: liste, [1, 2, 3], ["hei", 1.3e2]
- tuple: tupler, (1, 2, 3), ("hei", 1.3e2)

Datatyper

Funksjoner for å konvertere mellom datatyper. De gjør forskjellige ting for forskjellig input-typer, og fungerer ikke for alle.

Funksjon	Eksempel	Resultat
bin()	bin(92)	’0b1011100’
bool()	bool(12), bool(0)	True, False
chr()	chr(97)	’a’
ord()	ord(’a’)	97
float()	float(1)	1.0
int()	int(2.6), int("2")	2, 2
list()	list(range(2))	[0, 1]
str()	str(100)	’100’

Datatyper



Type gir hvilken datatype en variabel er, dermed kan vi sjekke med en if. Kan også bruke *isinstance*

```
a = 1.23
if type(a) == float:
    print("a er en float")
if isinstance(a, float):
    print("a er fortsatt en float")
```

Datatyper

Dette kan ofte være nyttig i funksjoner for å gjøre forskjellige ting for forskjellige parametere.

Input og output

Hva er input og output?

Input og output

Hva er input og output?

— Kommunikasjon mellom bruker og kode

Input og output

Hva er input og output?

- Kommunikasjon mellom bruker og kode
- Kommunikasjon mellom datamaskin og kode

Input og output

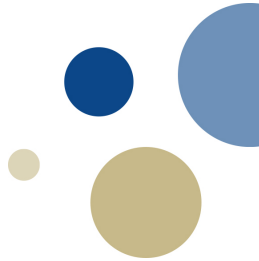
Hva er input og output?

- Kommunikasjon mellom bruker og kode: *input* and *print*
- Kommunikasjon mellom datamaskin og kode: *open*

Input og output - print

print-funksjonen.

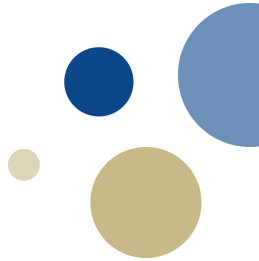
Input og output - print



print-funksjonen.

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

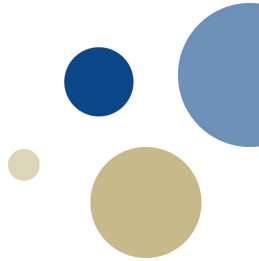
Input og output - print



print-funksjonen. Skriver disse ut det samme?

```
print("Hello World!")
print("Hello ", "World!")
```

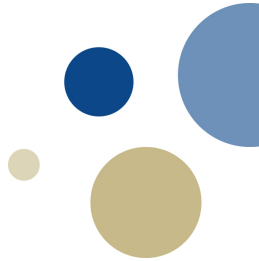
Input og output - print



print-funksjonen. Skriver disse ut det samme?
Default parameter: *sep*

```
print("Hello World!", sep=" ")
print("Hello ", "World!", sep=" ")
```

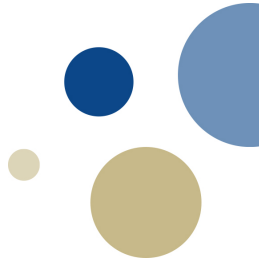
Input og output - print



print-funksjonen. Skriver disse ut det samme?
Default parameter: *sep*
Kan brukes til å endre hva som printes mellom argumentene i print-funksjonen.

```
print("Hello World!", sep=" ")
print("Hello ", "World!", sep="")
```

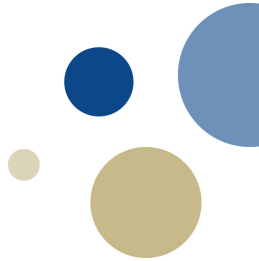
Input og output - print



print-funksjonen. Skriver disse ut det samme?

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

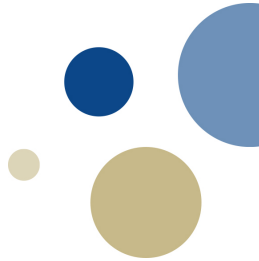
Input og output - print



print-funksjonen. Skriver disse ut det samme?
Default parameter: *end*

```
print("Hello World!", end="\n")
print("Hello ", end="\n")
print("World!", end="\n")
```

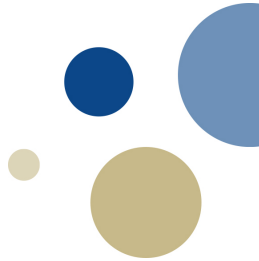
Input og output - print



print-funksjonen. Skriver disse ut det samme?
Default parameter: *end*

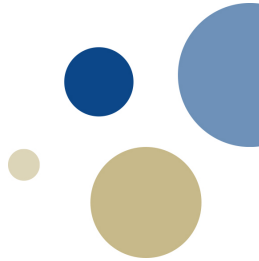
```
print("Hello World!", end="\n")
print("Hello ", end="")
print("World!", end="\n")
```

Input og output - print



sep og *end* er ikke alltid nødvendig å bruke, men kan gjøre koding lettere, og gjøre det enklere å formatere kode og plassere tekst.

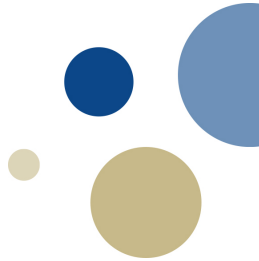
Input og output - input



input-funksjonen

```
x = input("Skriv inn x")
```

Input og output - input



input-funksjonen.
Viktig: Tar inn en (1) streng og returnerer en (1) streng! Argumenter kan ikke skrives inn som i print, vi må bruke streng-konkatenering (plusse sammen strenger).

```
antall_heltall = 1
x = input("Skriv inn " + str(antall_heltall) + " x: ")
x_heltall = int(x)
```

Funksjoner

Svært ofte ønsker man å gjenbruke deler av koden man skriver.

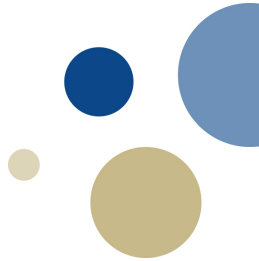
Funksjoner

Svært ofte ønsker man å gjenbruke deler av koden man skriver. Det å printe / skrive til skjermen er egentlig en veldig kompleks prosess, men den har blitt *abstrahert bort*.
Abstraksjoner er veldig viktige for å løse oppgaver, både til eksamen og senere.
Dette begrepet kommer vi tilbake til når vi begynner å se på eksamensoppgaver.

Funksjoner

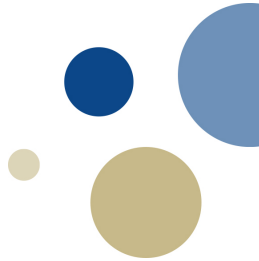
Et viktig prinsipp er det å *kalle* en funksjon. Når man *definerer* en funksjon, lager man bare en generell oppskrift for ubestemte parametere.
Når man kaller på en funksjon, så utfører man den oppskriften. Det vil si at vi gir det noen *fysiske* verdier å jobbe med.

Funksjoner - Funksjoner uten retur-verdi



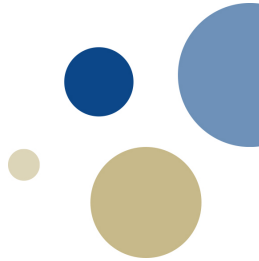
I mange tilfeller skal en funksjon gjøre noe enkelt som ikke krever at koden husker på endringen etter funksjonen har blitt kjørt.

Funksjoner - Funksjoner uten retur-verdi



I mange tilfeller skal en funksjon gjøre noe enkelt som ikke krever at koden husker på endringen etter funksjonen har blitt kjørt.
Dette er ofte relatert til *output*, som f.eks. printing og skrive til fil.

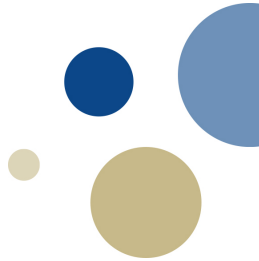
Funksjoner - Funksjoner uten retur-verdi



Funksjoner begynner med kodeordet *def*, funksjonsnavnet, en parentes med alle parameterene funksjonen tar inn og **VIKTIG** kolon :. Det må også være minst en kodesnutt i funksjonen.
Kanskje den enkleste funksjonen?

```
def func():
    pass
```

Funksjoner - Funksjoner uten retur-verdi



Eksempel med printing.

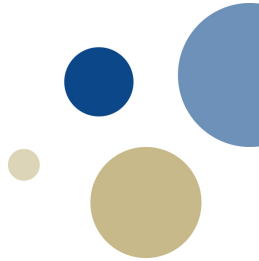
```
def print_resultater(res, tekst):
    print("Resultatet ble",res,sep=": ",end=" og ")
    print("dette er teksten: " + tekst)
```

Funksjoner - Funksjoner uten retur-verdi

Vi har allerede jobbet med default-parametere. *sep* og *end* er default-verdier i *print*-funksjonen, og med mindre de spesifiseres vil *sep* alltid være " " (mellomrom) og *end* alltid være "\n" (ny linje). Her bruker vi det til å bestemme om vi skal printe eller lagre til fil.

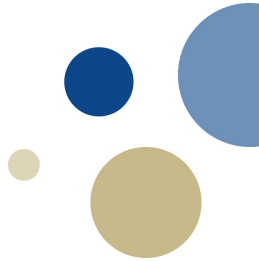
```
def output_resultater(res, file_path=""):
    if file_path != "":
        f = open(file_path, "w")
        f.write(res)
        f.close()
    else:
        print("Her er resultatet:", res)

output_resultater("hei") # printes
output_resultater("hei", "test.txt") # skrives til fil
output_resultater("hei", file_path="test.txt")
```

Funksjoner - Funksjoner med retur-verdi



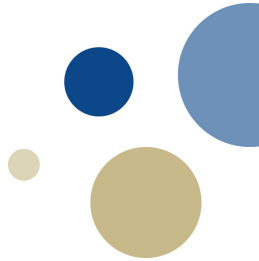
Dersom man ikke returnerer en verdi fra en funksjon, kan man ikke egentlig forvente at noe endres utenfor funksjonen vår.

Funksjoner - Funksjoner med retur-verdi



Dersom man ikke returnerer en verdi fra en funksjon, kan man ikke egentlig forvente at noe endres utenfor funksjonen vår.
Dette er ikke helt korrekt, men kommer tilbake til dette senere.

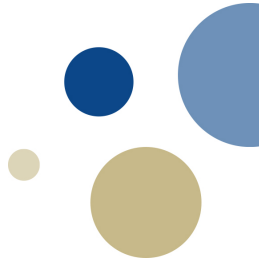
Funksjoner - Funksjoner med retur-verdi



Det vi vil skal komme ut av funksjonen vår, det som skal returneres, skriver vi etter return.

```
def pluss(tall_1, tall_2):
    tall_sum = tall_1 + tall_2
    return tall_sum
```

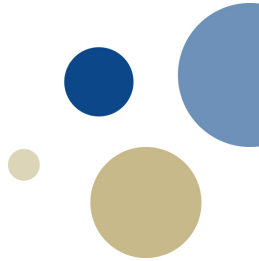
Funksjoner - Funksjoner med retur-verdi



Viktig: Når en verdi returneres fra en funksjon, så kan det tolkes som at vi *bytter* ut funksjonskallet med det som beregnes i og returneres av funksjonen.

```
def pluss(tall_1, tall_2):
    tall_sum = tall_1 + tall_2
    return tall_sum

resultat = pluss(1, 2)
print(resultat)
```

Funksjoner - Funksjoner med retur-verdi



Viktig: Når en verdi returneres, må den tilordnes, ellers glemmer programmet hva resultatet er.
I noen tilfeller trenger vi kanskje ikke resultatet til noe annet enn f.eks. printing, men ofte er det lurt å tilordne.

```
def pluss(tall_1, tall_2):
    tall_sum = tall_1 + tall_2
    return tall_sum

resultat = pluss(1, 2)
print(resultat)
```

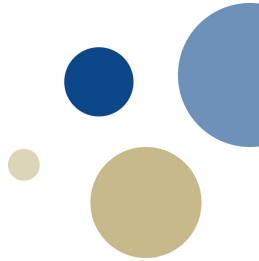
Funksjoner - Funksjoner med retur-verdi

Styrken med funksjoner er at de kan være ganske generelle, og spesifiseres kun når de kalles.
Sammen med løkker og lister er funksjoner noe av det viktigste med programmering.

Iterables

Iterables er generelt grupper med objekter man kan *iterere* seg gjennom.

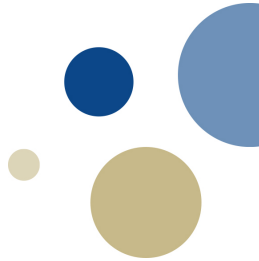
Lister



Lister er en gruppe med objekter som ligger etter hverandre i *minnet*.
De defineres med firkant-parenteser rundt objekter som er separert med komma.
De kan være av forskjellige typer, men det er ikke alltid lurt. Da kan man ikke behandle elementene likt.

```
liste = ["hei", 1, 0.1]
```

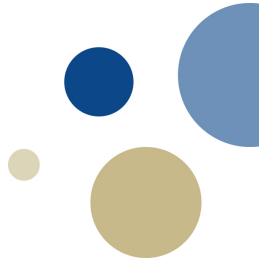
Lister



Lister kan endres.

```
liste = ["hei", 1, 0.1]
liste[1] = "alle"
```

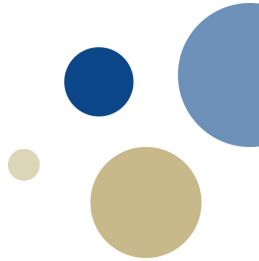
Lister



Lister kan utvides.

```
liste = ["hei", 1, 0.1]
liste = liste + ["alle", "sammen"]
```

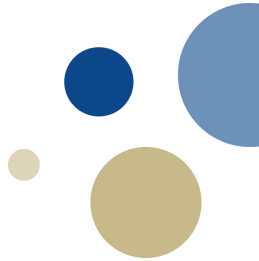
Lister



Alle gjør endringer på den originale listen, **bortsett fra** *copy()*, som returnerer en ny liste. Dette er viktig dersom man vil beholde originalen.

Metode	Eksempel		Metode	Eksempel
append	liste.append("hei")		reverse	liste.reverse()
insert	liste.insert(2, "hei")		sort	liste.sort()
remove	liste.remove("hei")		copy	kopi = liste.copy()
pop	liste.pop(1)		clear	liste.clear()
count	liste.count("hei")			

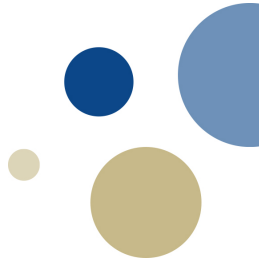
Lister



Når man bruker lister i funksjoner, så må man huske at hvis man endrer listen man tar inn, så endres den utenfor også!
Dette gjelder ikke for vanlige variabler/parametere.

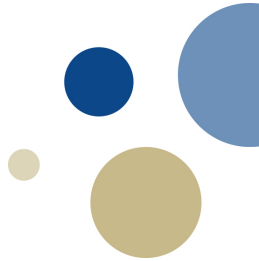
```
def sett_tall(tall):
    tall = 1
def legg_til(liste):
    liste.append(1)
a = "hei"
sett_tall(a) # a endres ikke
l = [1, 5.12]
legg_til(l) # l endres
```

Tupler



Tupler er basically det samme som lister, men de er ikke-muterbare, som strenger.

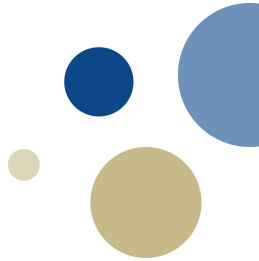
Tupler



Defineres på samme måte som lister, bare med parenteser istedenfor firkantparenteser.

```
liste = [1, 2, 3]
tuple = (1, 2, 3)
```

Tupler

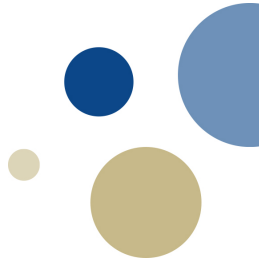


Metoder: Siden tupler er ikke-muterbare, er det svært få metoder, og de er generelt ikke nødvendige å kunne.

- count
- index

```
t = (1, 2, 3, 1)
t.count(1) # -> 2
t.index(1) # -> 0
```

Tupler



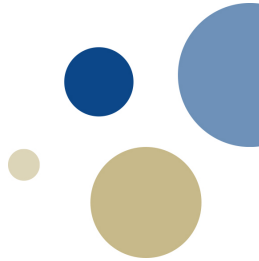
Hvorfor bruke tupler?

Tupler

Hvorfor bruke tupler?
Tupler kan brukes der du vet du har en mengde som er konstant

```
mynter = (20, 10, 5, 1)
penger = 19
liste  = [0] * len(mynter)
for i in range(len(mynter)):
    liste[i] = penger \\ mynter[i]
    penger  %= mynter[i]
```

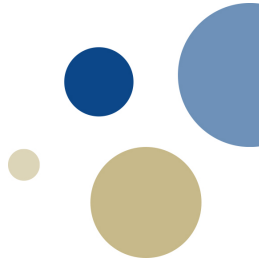
Tupler



Kan forenkle kode som trenger flere linjer. Tenk tilbake til tilordningsproblemet der vi ville bytte om på verdiene i to variabler.

```
a = "hei"
b = 2
temp = a
a = b
b = temp
```

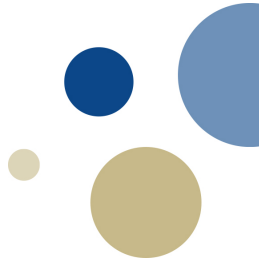
Tupler



Kan forenkle kode som trenger flere linjer. Tenk tilbake til tilordningsproblemet der vi ville bytte om på verdiene i to variabler. Dette kan nå forenkles.

```
a = "hei"
b = 2
a, b = b, a
```

Tupler



Mange nyttige funksjoner man kan få bruk for bruker også tupler.

```
for index, element in enumerate(liste)
for key, item in dictionary.items()
```

Tupler

Generelt: hvis du kan gjøre det med tupler, kan det gjøres med lister eller med annen kode. Tupler er hovedsakelig *ekstra*.

Løkker

Løkker er helt essensielle i koding.

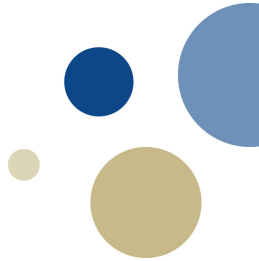
Løkker

Løkker er helt essensielle i koding.
Idéen bak er å repetere en viss kodesnutt om og om igjen.

Løkker - while-løkken

Strengt tatt er alle løkker en while løkke.

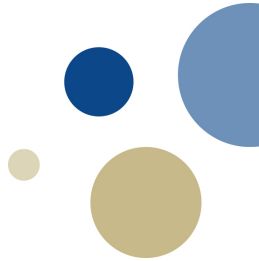
Løkker - while-løkken



While-løkker kjører kun hvis en boolsk verdi er True. Det betyr at vi kan sjekke en condition i løkka.

```
while True:
    print("to infinity!")
print("and beyond?")
```

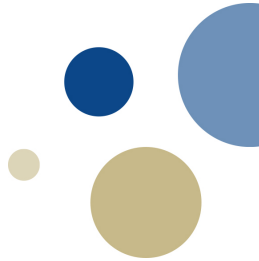
Løkker - while-løkken



Minner om en for-løkke?

```
i = 0
while i < 10:
    print(i)
    i += 1
```

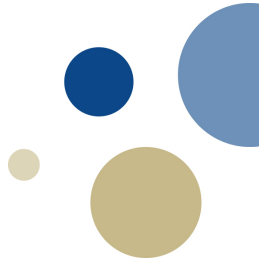
Løkker - while-løkken



Veldig bra å bruke dersom man ikke vet hvor mange ganger løkka skal kjøre, men *vet det når man ser det*.

```
# dette er en metode som sikrer at det tomme svaret
# ikke blir tatt med
answer = input("Skriv inn ditt svar: ")
answer_lst = []
while answer != "": # tomt svar, trykket enter med en gang
    answer_lst.append(answer)
    answer = input("Skriv inn ditt svar: ")
```

Løkker - for-løkken

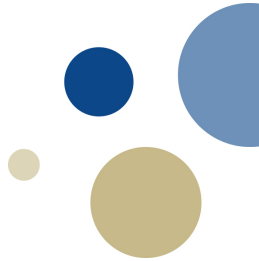


for-løkker brukes når man har en bestemt *iterable* man ønsker å iterere gjennom.

Løkker - for-løkken

for-løkker brukes når man har en bestemt *iterable* man ønsker å iterere gjennom. Dette gjelder spesielt de vi allerede har gått gjennom, i form av *lister* og *tupler*.

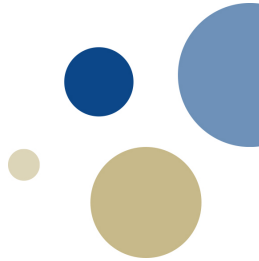
Løkker - for-løkken



for-løkker brukes når man har en bestemt *iterable* man ønsker å iterere gjennom. Dette gjelder spesielt de vi allerede har gått gjennom, i form av *lister* og *tupler*.
Disse kan defineres *inline*

```
for word in ("hei", "hei", "alle", "sammen"):
    print(word)
for word in ["hei", "hei", "alle", "sammen"]:
    print(word)
```

Løkker - for-løkken



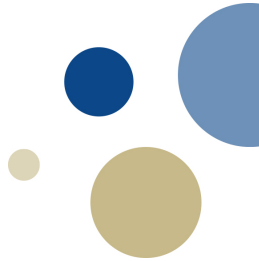
for-løkker brukes når man har en bestemt *iterable* man ønsker å iterere gjennom. Dette gjelder spesielt de vi allerede har gått gjennom, i form av *lister* og *tupler*.
Eller på forhånd. Da kan de gjenbrukes senere, eller redefineres før.

```
tpl = ("hei", "hei", "alle", "sammen")
lst = ["hei", "hei", "alle", "sammen"]
for word in tpl:
    print(word)
for word in lst:
    print(word)
```

Løkker - for-løkken

Det er hovedsakelig to måter å *iterere* gjennom en liste.

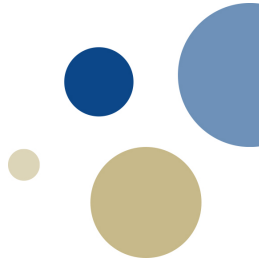
Løkker - for-løkken



Det er hovedsakelig to måter å *iterere* gjennom en liste.
Element-vis. Man henter ut hvert *element* i listen.

```
lst = [1, 2, 3, "hei", (1, 2, 3)]
for element in lst:
    print(element)
```

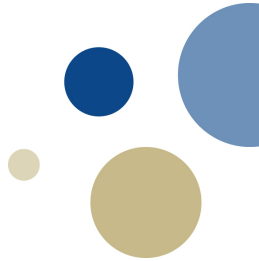
Løkker - for-løkken



Det er hovedsakelig to måter å *iterere* gjennom en liste.
Indeks-vis. Man henter ut *plasseringen* til hvert element i listen.

```
lst = [1, 2, 3, "hei", (1, 2, 3)]
for i in range(len(lst)):
    print(lst[i])
```

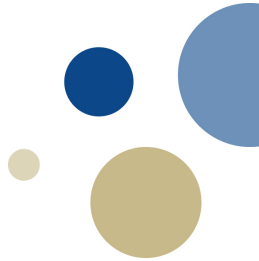
Løkker - for-løkken



Det er hovedsakelig to måter å *iterere* gjennom en liste.
Indeks-vis. Man henter ut *plasseringen* til hvert element i listen.
Viktig: hvis vi itererer element-vis gjennom en liste, vet vi egentlig ingenting om plasseringen, og hvis vi trenger indeksen, er det best å gå gjennom indeks-vis.

```
lst = [1, 2, 3, "hei", (1, 2, 3)]
for i in range(len(lst)):
    print(lst[i])
```

Løkker - for-løkken



Det er hovedsakelig to måter å *iterere* gjennom en liste.
Kan bruke *enumerate* for å få ut begge med en gang.

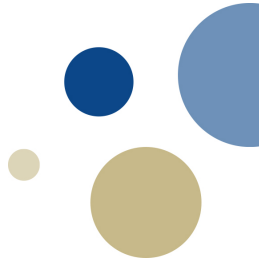
```
lst = [1, 2, 3, "hei", (1, 2, 3)]
for i, element in enumerate(lst):
    # lst[i] == element -> True
    print(i, lst[i], element)
```

Løkker - for-løkken

Det er hovedsakelig to måter å *iterere* gjennom en liste.
Kan bruke *enumerate* for å få ut begge med en gang. Sjeldent dette ikke kan løses med bare indekser, og tilordning.

```
lst = [1, 2, 3, "hei", (1, 2, 3)]

for i, element in enumerate(lst):
    # lst[i] == element -> True
    print(i, lst[i], element)

for i in range(len(lst)):
    element = lst[i]
    # lst[i] == element -> True
    print(i, lst[i], element)
```

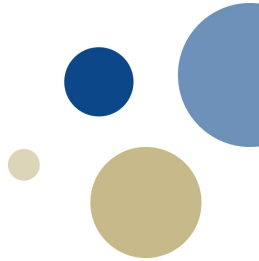
Løkker - for-løkken



Løkker kan enkelt skrives inne i hverandre.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken



Løkker kan enkelt skrives inne i hverandre. Går gjennom hvert element i *lst1*, sjekker hvert element i *lst2* og ser om jeg finner det.

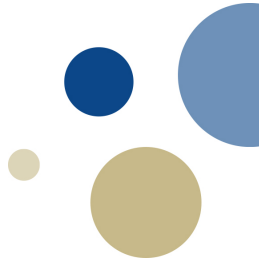
```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken

Løkker kan enkelt skrives inne i hverandre. Går gjennom hvert element i *lst1*, sjekker hvert element i *lst2* og ser om jeg finner det. Her trenger vi ikke gå gjennom med indekser (range), siden vi trenger ikke vite noe om posisjon.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

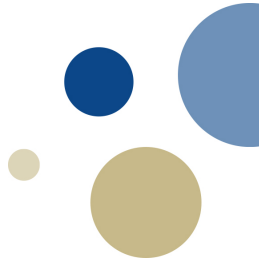
Løkker - for-løkken



Det kan se vanskelig ut å analysere når det er flere løkker.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

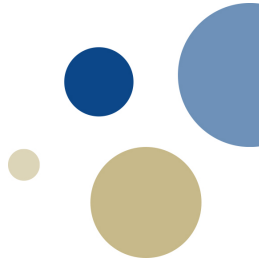
Løkker - for-løkken



Det kan se vanskelig ut å analysere når det er flere løkker. Det er enklest å begynne å analysere *innenfra og ut*.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

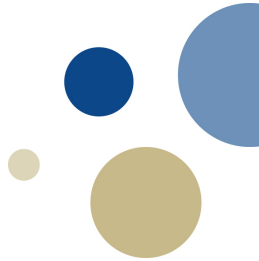
Løkker - for-løkken



Det kan se vanskelig ut å analysere når det er flere løkker. Det er enklest å begynne å analysere *innenfra og ut*. Dette gjelder også når vi skal designe kode som bruker kode inni kode.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

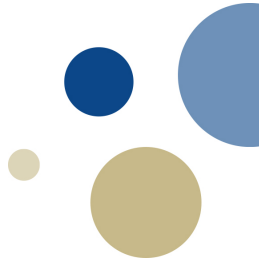
Løkker - for-løkken



Vi antar at *x* og *y* er bestemte verdier (f.eks. $x = 1$, $y = 2$). Hva er det vi vil gjøre med denne bestemte verdien?

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken



Vi antar at *x* og *y* er bestemte verdier (f.eks. $x = 1$, $y = 2$). Hva er det vi vil gjøre med denne bestemte verdien?
Hvis disse er like, øker vi *telleren* med 1.

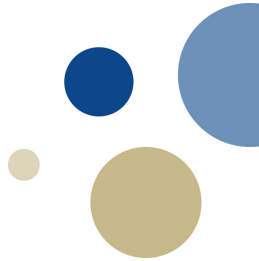
```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken

Vi antar at *x* og *y* er bestemte verdier (f.eks. $x = 1$, $y = 2$). Hva er det vi vil gjøre med denne bestemte verdien?
Hvis disse er like, øker vi *telleren* med 1.
Nå er den *innerste* delen av koden abstrahert bort. Hva er neste steg?

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken



Vi antar at *x* er en bestemt verdi (f.eks. $x = 1$).
Vi vet hva som skjer for en bestemt *y*, så det kan generaliseres. Vi vet hva *x* er, og vi vet hva vi skal gjøre hvis vi har en *x* og en *y*. Så vi går gjennom alle mulige *y*-er.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken

Nå trenger vi ikke lenger å se på en bestemt verdi for *x*, siden vi vet hva som skjer med en (den sammenlignes med hver mulige *y*), kan den også generaliseres.
For hver mulige *x*, sjekk hver mulige *y*. Dersom *x* == *y*, så øker vi telleren med en.

```
def antall_like(lst1, lst2):
    res = 0
    for x in lst1:
        for y in lst2:
            if x == y:
                res += 1

    return res
```

Løkker - for-løkken

Strenger og tupler kan behandles ganske likt, siden begge er ikke-muterbare. F.eks., dersom vi vil endre noe, så må vi bygge disse strukturene på nytt.
Dette kan vi gjøre litt enklere senere med slicing.

```
def insert_string(original, index, new_string):
    # antar at index er valid
    resultat = "" # starter med tom streng
    for i in range(len(original)):
        if i == index:
            resultat += new_string
        # om vi plusser foran eller bak new_string
        # bestemmer om vi inserter foran eller bak.
        resultat += original[i]
    return resultat
```

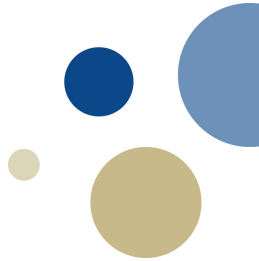
Iterables

Iterables er generelt grupper med objekter man kan *iterere* seg gjennom.

Slicing

Slicing er en måte å indeksere en bestemt del av en iterable, hovedsakelig lister, strenger og tupler.

Slicing



Slicing er en måte å indeksere en bestemt del av en iterable, hovedsakelig lister, strenger og tupler. Vi bruker firkantparenteser, siden vi aksesserer, og samme syntaks som en *range* for å bestemme hva vi vil aksessere.

```
iterable[fra_og_med_start:til_men_ikke_med_slutt:steg]
```

Slicing

Dersom vi vil starte på start, *trenger man ikke skrive noe før første kolon*.

```
iterable[:til_men_ikke_med_slutt:steg]
```

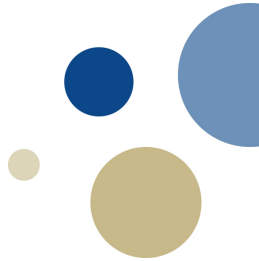
Dersom vi vil slutte på, inklusiv, slutten, *trenger man ikke skrive noe mellom første og andre kolon*.

```
iterable[fra_og_med_start::steg]
```

Dersom vi vil bruke 1 som *steg*, *trenger man ikke skrive noe etter siste kolon*, og man *trenger ikke ha med siste kolon heller*.

```
iterable[fra_og_med_start:til_men_ikke_med_slutt:]
iterable[fra_og_med_start:til_men_ikke_med_slutt]
```

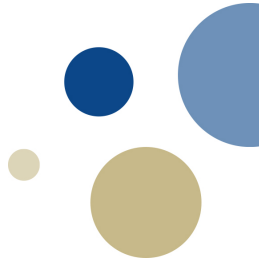
Slicing



Grunnen til at slicing går fra og med, til men ikke med, er at da kan man dele opp en iterable med samme indeks.

```
lst[:ind] + lst[ind:] == lst # -> True
```

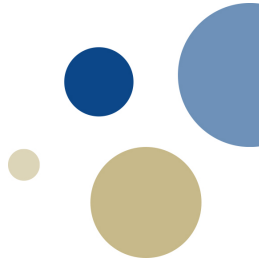
Slicing



Dette kan f.eks. brukes for kjapp kopiering eller reversering av lister.
Slicing returnerer **nye** objekter, i motsetning til f.eks. *lst.reverse()*.

```
kopi = lst[:]
rev  = lst[::-1]
```

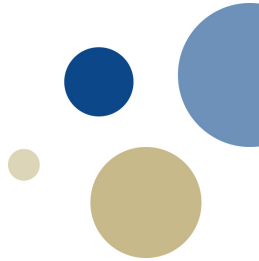
Slicing



Vi kan også bruke slicing til å korte ned denne koden fra tidligere.

```
def insert_string(original, index, new_string):
    # antar at index er valid
    resultat = "" # starter med tom streng
    for i in range(len(original)):
        if i == index:
            resultat += new_string
        # om vi plusser foran eller bak new_string
        # bestemmer om vi inserter foran eller bak.
        resultat += original[i]
    return resultat
```

Slicing



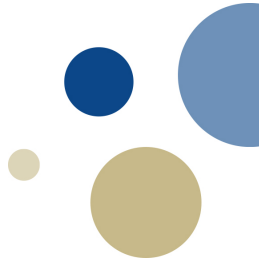
Vi kan også bruke slicing til å korte ned denne koden fra tidligere.

```
def insert_string(original, index, new_string):
    # antar at index er valid
    return original[:index] + new_string + original[index:]
```

Dictionaries

Dictionaries er kanskje den nyttigeste datastrukturen i Python, og en av de mest brukte.

Dictionaries

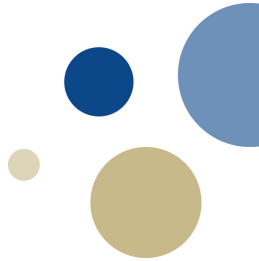


Dictionaries er kanskje den nyttigeste datastrukturen i Python, og en av de mest brukte. Det som er viktig å huske på er at en dictionary *kobler* sammen ulike verdier.

Dictionaries

Dictionaries er kanskje den nyttigeste datastrukturen i Python, og en av de mest brukte. Det som er viktig å huske på er at en dictionary *kobler* sammen ulike verdier.
Vi kobler fra en *nøkkel* til en *verdi*.

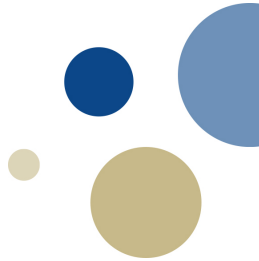
Dictionaries



Syntaksen fungerer i prinsipp på samme måte som en liste. Vi bruker firkantparenteser til å aksessere *verdien* som ligger på en *nøkkel*.

```
dictionary = {"katt": 1, "hund": 2}
print(dictionary["katt"]) # gir 1
```

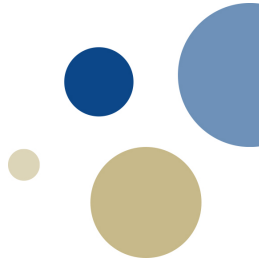
Dictionaries



Kan forenkle problemer som kan være vanskelige med lister.
Eksempel: Telle opp antall bokstaver i en streng.

```
streng = "heihei"
# alle bokstaver fra a til z
bokstaver = [chr(bokstav) for bokstav in range(ord("a"),
                                ord("z") + 1, 1)]
oppteller = [0] * len(bokstaver)
for i in range(len(bokstaver)):
    oppteller[i] = streng.count(bokstaver[i])
```

Dictionaries



Kan forenkle problemer som kan være vanskelige med lister.
Eksempel: Telle opp antall bokstaver i en streng.
Kan være vanskelig å håndtere dette, siden vi har to separate lister.
Finnes selvsagt andre løsninger, men for å få en direkte kobling mellom bokstav og antall er det enklest med dictionary.

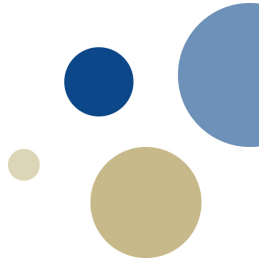
```
streng = "heihei"
bokstaver_til_antall = {}
for bokstav in streng:
    antall = bokstaver_til_antall.setdefault(bokstav, 0)
    bokstaver_til_antall[bokstav] = antall + 1
```

Dictionaries

Metoder

Metode	Eksempel
clear	dictionary.clear()
copy	kopi = dictionary.copy()
fromkeys	dict.fromkeys(keys, values)
get	value = dictionary.get(key)
items	t_list = dictionary.items()
values	verdier = dictionary.values()

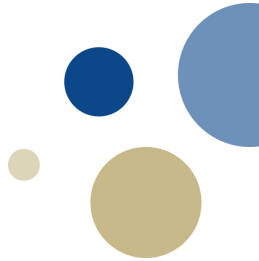
Dictionaries



Metoder (Cont.)

Metode	Eksempel
keys	nøkler = dictionary.keys()
pop	dictionary.pop(key)
popitem	dictionary.popitem()
setdefault	value = dictionary.setdefault(key, default)
update	dictionary.update(iterable)
update	dictionary.update(other_dict)

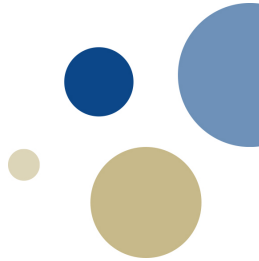
Dictionaries



Kombineres godt med for-løkker. Avhengig av hva du trenger, har vi flere metoder for å iterere oss gjennom en dictionary.

```
dictionary = {"hello": "world", 12: 9}
```

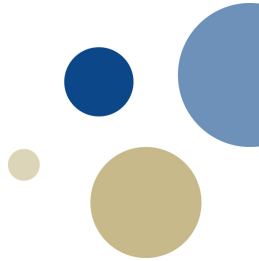
Dictionaries



Kombineres godt med for-løkker. Avhengig av hva du trenger, har vi flere metoder for å iterere oss gjennom en dictionary.
Dersom vi kun er interessert i verdiene, og ikke nøklene (ofte hvis noe kun skal printes ut):

```
dictionary = {"hello": "world", 12: 9}
for value in dictionary.values():
    print(value)
```

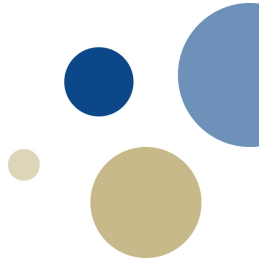
Dictionaries



Kombineres godt med for-løkker. Avhengig av hva du trenger, har vi flere metoder for å iterere oss gjennom en dictionary.
Dersom vi kun er interessert i nøklene, og ikke verdiene, eller ønsker å aksessere verdiene med firkantparenteser:

```
dictionary = {"hello": "world", 12: 9}
for key in dictionary.keys():
    print(key, dictionary[key])
```

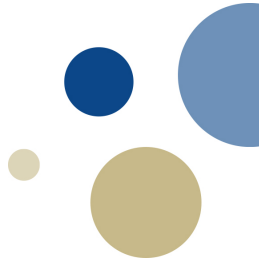
Dictionaries



Kombineres godt med for-løkker. Avhengig av hva du trenger, har vi flere metoder for å iterere oss gjennom en dictionary.
Dersom vi er interessert i både verdi og nøkler, som vi kanskje oftest er:

```
dictionary = {"hello": "world", 12: 9}
for key, value in dictionary.items():
    print(key, value)
```

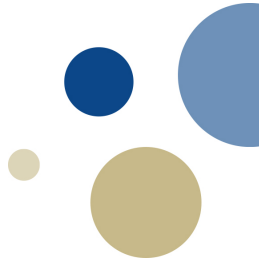
Dictionaries



Husk: det er lett å gå fra nøkkel til verdi, men vanskelig å gå fra verdi til nøkkel:

```
# value to key
dictionary = {"hello": "world", 12: 9}
key = "hello"
value = dictionary[key]
```

Dictionaries



Husk: det er lett å gå fra nøkkel til verdi, men vanskelig å gå fra verdi til nøkkel:

```
# value to key
dictionary = {"hello": "world", 12: 9}
key = None # ingen key enda, setter den til None
value = "hello"
for current_key, current_value in dictionary.items():
    if current_value == value:
        key = current_key
        break # bare en key pr. value
```

Sets

Det finnes også set, som ligner veldig på lister. Disse kan kun ha unike verdier, som er egentlig eneste grunn til å bruke disse. Problemer man kan løse med set kan også løses med lister, men man kan få kortere (men muligens mindre leslige) løsninger.

Input og output

Hva er input og output?

Input og output

Hva er input og output?

— Kommunikasjon mellom bruker og kode

Input og output

Hva er input og output?

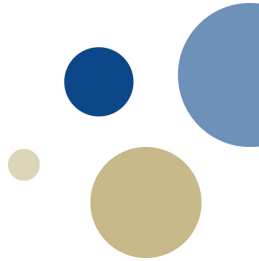
- Kommunikasjon mellom bruker og kode
- Kommunikasjon mellom datamaskin og kode

Input og output

Hva er input og output?

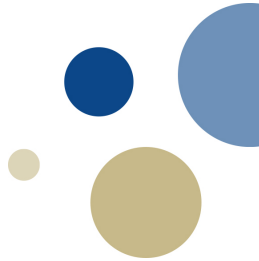
- Kommunikasjon mellom bruker og kode: *input* and *print*
- Kommunikasjon mellom datamaskin og kode: *open*

Input og output - Lese filer



open-funksjonen brukes til å åpne filer generelt. Dette gjelder både lesing av og skriving til filer.

Input og output - Lese filer

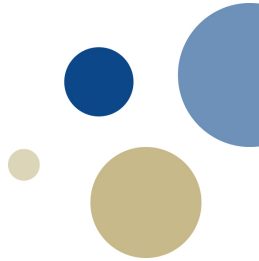


open-funksjonen brukes til å åpne filer generelt. Dette gjelder både lesing av og skriving til filer.
Vi begynner med å lese fra filer som allerede eksisterer.

Input og output - Lese filer

open-funksjonen tar inn to parametere. Begge er strenger, den første er hvilken fil du vil åpne, og den andre er hvilken *modus* du vil åpne filen i. For å lese fra filer bruker vi "r" for å lese.
Vi kan videre bestemme om vi vil åpne fila som en tekstfil eller binærfil.
Det vil si, "rt" og "rb", men "r" tolkes som "rt", siden det er vanligst, og med mindre det står eksplisitt på eksamen at man skal lese fila som en binærfil, er det "r" som mest sannsynlig skal brukes for å lese filer.

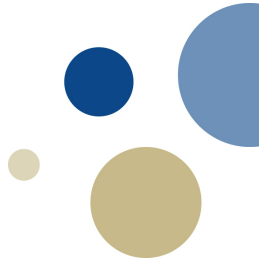
Input og output - Lese filer



Filen "test.txt" må eksistere, ellers får vi en feilmelding. I tillegg må den ligge i **samme mappe som du kjører Python-fila fra**.

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

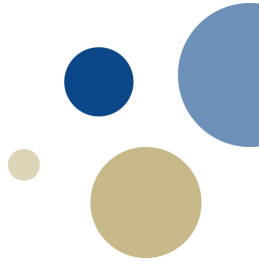
Input og output - Lese filer



Filen "test.txt" må eksistere, ellers får vi en feilmelding. I tillegg må den ligge i **samme mappe som du kjører Python-fila fra**.
Dette kalles relativ filplassering.

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

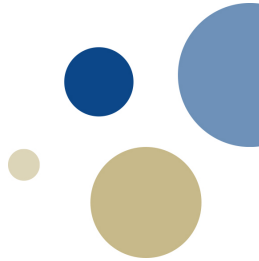
Input og output - Lese filer



Metoden *read()* henter ut **all** teksten fra fila.

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

Input og output - Lese filer



Metoden *read()* henter ut **all** teksten fra fila. Det vil si at vi ikke kan lese to ganger fra samme objekt, og eventuelt må åpne fila på nytt.

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

Input og output - Lese filer

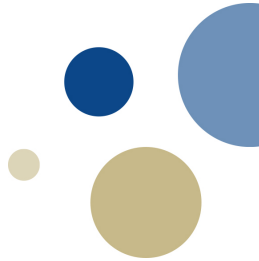
Metoden *read()* henter ut **all** teksten fra fila. Det vil si at vi ikke kan lese to ganger fra samme objekt, og eventuelt må åpne fila på nytt. Husk å lukke fila til slutt. Dette er ikke like viktig når man leser fra en fil som når man skriver til en fil, men er lurt siden man kan miste poeng på eksamen.

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

Input og output - Lese filer

Det finnes andre metoder også, for eksempel, dersom tekstfila er for stor til å åpnes i minnet.

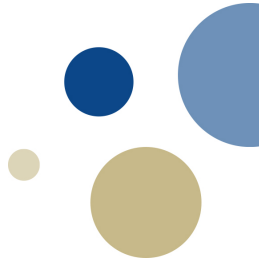
Input og output - Lese filer



Det finnes andre metoder også, for eksempel, dersom tekstfila er for stor til å åpnes i minnet.
Metoden *readline()* leser en og en linje, og når alt er lest ut returnerer den bare tomme strenger.

```
f = open("test.txt", "r")

f_lesing = f.readline()
print(f_lesing)
while f_lesing != "":
    f_lesing = f.readline()
    print(f_lesing)
```

Input og output - Lese filer



Det finnes andre metoder også, for eksempel, dersom tekstfila er for stor til å åpnes i minnet.
Metoden *readlines()* er veldig lik, men leser ut alle linjene i fila, og legger dem i en liste.
Begge disse kan være nyttige (spesielt *readlines()*) dersom man skal gjøre noe pr. linje, og kan gjøre det mer intuitivt å jobbe med filer.

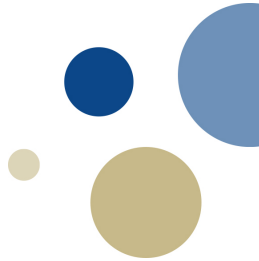
```
f = open("test.txt", "r")

f_linjer = f.readlines()
for linje in f_linjer:
    print(linje)
```

Input og output - Lese filer

Det finnes andre metoder også, for eksempel, dersom tekstfila er for stor til å åpnes i minnet.
Begge disse kan være nyttige (spesielt *readlines()*) dersom man skal gjøre noe pr. linje, og kan gjøre det mer intuitivt å jobbe med filer.

```
f = open("test1.txt", "r")
g = open("test2.txt", "r")

f_lesing = f.readline()
print(f_lesing)
while f_lesing != "":
    f_lesing = f.readline()
    print(f_lesing)

g_linjer = g.readlines()
for linje in g_linjer:
    print(linje)
```

Input og output - Skrive til filer



Igjen brukes *open*-funksjonen. Viktig å holde tunga rett i munnen, siden det er mye som kan gå galt her.

Input og output - Skrive til filer

Igjen brukes *open*-funksjonen. Viktig å holde tunga rett i munnen, siden det er mye som kan gå galt her.
Her er det tre nye moduser, ikke bare *"r"*, som tidligere.

- "w": Write, standard skriving. Dersom fila eksisterer fra før av, slettes den opprinnelige fila.
- "a": Append, legger til tekst på slutten av fila. Dersom den ikke eksisterer, opprettes den.
- "x": Exclusive write, vil feile dersom fila allerede eksisterer. Kan bare brukes til å opprette nye filer.

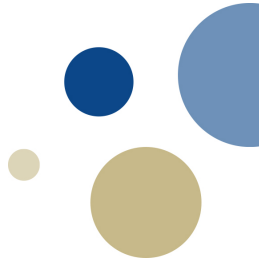
Input og output - Skrive til filer

Igjen brukes *open*-funksjonen. Viktig å holde tunga rett i munnen, siden det er mye som kan gå galt her.
Her er det tre nye moduser, ikke bare *"r"*, som tidligere.

- "w": Write, standard skriving. Dersom fila eksisterer fra før av, slettes den opprinnelige fila.
- "a": Append, legger til tekst på slutten av fila. Dersom den ikke eksisterer, opprettes den.
- "x": Exclusive write, vil feile dersom fila allerede eksisterer. Kan bare brukes til å opprette nye filer.

Det finnes også en modus "+", som tillater oppdatering (read og write samtidig), men er oftest bedre å gjøre dette i to steg.

Input og output - Skrive til filer

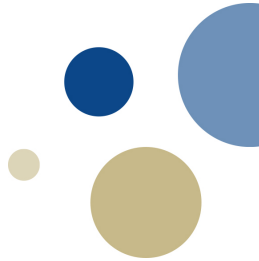


Igjen brukes *open*-funksjonen. Viktig å holde tunga rett i munnen, siden det er mye som kan gå galt her.
Tilsvarende som med å lese fra filer, kan filer også åpnes som binærfiler.

Input og output - Skrive til filer

Igjen brukes *open*-funksjonen. Viktig å holde tunga rett i munnen, siden det er mye som kan gå galt her.
Tilsvarende som med å lese fra filer, kan filer også åpnes som binærfiler.

Input og output - Skrive til filer



Skriving til fil er dermed ganske rett frem. Man skriver fra *der man er* i fila.

```
f = open("test.txt", "w")
f.write("Hello\n")
f.write("World!\n")
f.close()

g = open("test.txt", "a")
g.write("New\n")
g.write("lines!\n")
g.close()
```

Unntak

Kode kjører ikke alltid som vi forventer, spesielt hvis vi har bruker-input.

Unntak

Kode kjører ikke alltid som vi forventer, spesielt hvis vi har bruker-input.
Det som ofte gjøres da er at man skriver kode, som forventer at svarene lar koden kjøre, og har unntak dersom noe uventet skjer.

Unntak

Kode kjører ikke alltid som vi forventer, spesielt hvis vi har bruker-input.
Det som ofte gjøres da er at man skriver kode, som forventer at svarene lar koden kjøre, og har unntak dersom noe uventet skjer.
Dette er en svært vanlig form for feilhåndtering.

Unntak

Kode kjører ikke alltid som vi forventer, spesielt hvis vi har bruker-input.
Det som ofte gjøres da er at man skriver kode, som forventer at svarene lar koden kjøre, og har unntak dersom noe uventet skjer.
Dette er en svært vanlig form for feilhåndtering.

Unntak

Eksempel

```
try:
    number = int(input("Write a number: "))
except ValueError as error:
    print(error)
    print("Not a number")
except:
    print("Some other mistake")
else:
    print("The number is:", number)
finally:
    print("Done")
```

Unntak

Eksempel

```
try:
    number = int(input("Write a number: ")) # runs always,
                                            but may crash
except ValueError as error: # specifics assigned to
                                variable error
    # runs specifically if there is a ValueError
    print(error) # prints the specifics of this error
    print("Not a number")
except: # runs for any other exception
    print("Some other mistake")
else: # runs if no crash
    print("The number is:", number)
finally: # runs always
    print("Done")
```

Eksamensoppgaver

Går gjennom noen eksamensoppgaver, både kodeforståelse og koding.

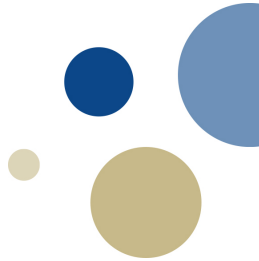
Eksamensoppgaver

Går gjennom noen eksamensoppgaver, både kodeforståelse og koding.
Fokus på forståelse, visualisering og "tankeprosessen".

Kodeforståelse

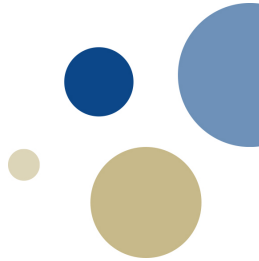
Viktig å lese oppgaveteksten.
Ofte er det spørsmål om både hva som returneres for en bestemt input-verdi, i tillegg til å beskrive funksjonen med en setning.

Kodeforståelse



Viktig å lese oppgaveteksten.
Ofte er det spørsmål om både hva som returneres for en bestemt input-verdi, i tillegg til å beskrive funksjonen med en setning.
Det vil *ikke* si at vi skriver hva som skjer, men heller hva som er poenget med funksjonen.

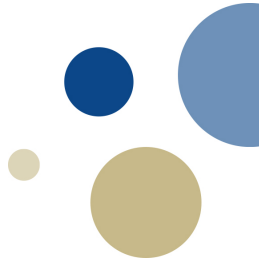
Kodeforståelse



Feil svar: Funksjonen setter $r = 1$, og så lenge x er større eller lik 1 ganges r med x, og x minker med 1.

```
def f(x):
    r = 1
    while x >= 1:
        r *= x
        x -= 1
    return r
```

Kodeforståelse



Riktig(ere) svar: Funksjonen beregner verdien $x!$, x fakultet.

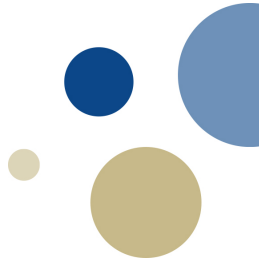
```
def f(x):
    r = 1
    while x >= 1:
        r *= x
        x -= 1
    return r
```

Kodeforståelse 2016 2d

Hva blir skrevet ut på skjermen hvis koden vist under blir kjørt?
Forklar med en setning hva funksjonen *f* gjør

```
def f(x):
    y = 0
    while x > 0:
        y = y + x % 10
        x = int( x / 10 )
    if y >= 10:
        y = f( y )
    return y

print( f(32145) )
```

Kodeforståelse 2016 kont 4b



6

f regner ut den rekursive tverrsummen tallet som den får inn som parameter, dvs. fortsetter å ta tverrsum helt til vi får et ensifret tall. Tverrsummen av 32145 blir 15, tverrsummen av 15 blir deretter 6.

Kodeforståelse 2016 kont 4b

Hva returneres hvis man kaller $f([[3,5],[2,4],[1,3]])$ med koden nedenfor?
Forklar med en setning hva funksjonen $f()$ gjør?

```
def f(b):
    c=len(b[0])
    d=len(b)
    g = [[0 for row in range(d)]
         for col in range(c)]
    for e in range(0, c):
        for f in range(0, d):
            g[e][f] = b[f][e]
    return g
```

Kodeforståelse 2016 kont 4b

[[3, 2, 1], [5, 4, 3]]
Evt. ingenting ettersom det er feil med innrykk i oppgaveteksten.
Begge svar gir full pott!
Funksjonen transponerer en matrise.

Kodeforståelse 2017 2a

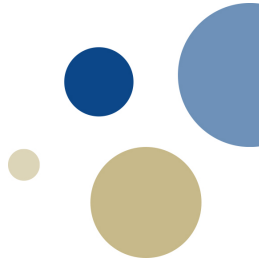
Funksjonen *bin$_s$earch* er ment til å skulle utføre binærsøk, men resulterer i feilmeldingen *"IndexError: list index out of range"*.
I hvilken linje er feilen?
Hva skulle egentlig stått på den linjen for at funksjonen skal virke etter sin hensikt?

```
def bin_search(liste, verdi, imin, imax):
    if (imax < imin):
        return False
    else:
        imid = (imin + imax)
        if verdi < liste[imid]:
            return bin_search(liste, verdi, imin, imid-1)
        elif verdi > liste[imid]:
            return bin_search(liste, verdi, imid+1, imax)
        else:
            return imid
```

Kodeforståelse 2017 2a

Linje 5
$imid = (imid + imax)/2$

Kodeforståelse 2018 sett 1 2a



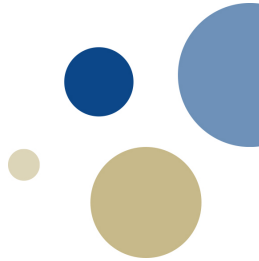
Hva returneres ved funksjonskallet under?
myst(((*TrueandFalse*)*or*(*FalseandTrue*)),((*FalseorTrue*)*and*(*not*(*notTrue*

```
def myst(val1, val2):
    if (val1 and val2):
        return 1
    elif (val1 and not val2):
        return 2
    elif (not val1 and val2):
        return 3
    else:
        return 4
```

Kodeforståelse 2017 2a

3

Kodeforståelse 2018 kont 2e og f



Hva blir skrevet ut til skjerm når du kjører programmet vist under?
$z = ((2, 0, 11, 8, 5), (14, 17, 13, 8, 0))$ *print*(*myst*3(z))
Beskriv med en setning hva koden i oppgave 2e gjør.

```
def myst3(a):
  s  = ''
  for r in a:
    for c in r:
      s += chr(ord('A') + c)
  return s
```

Kodeforståelse 2018 kont 2e og f

CALIFORNIA Store bokstaver (pga. $ord('A')$) og ingen fnutter (pga. printing)
Henter ut ett og ett tall fra et tuppel av tupler, og lager en sammenhengende tekst av store bokstaver hvor c angir hvor langt unna hver bokstav er fra A i alfabetet (ASCII- tabellen).

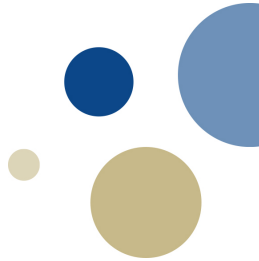
Programmering 2016 kont 2

Du kan anta at alle funksjonene mottar gyldige argumenter (inn-verdier). Du kan benytte deg av funksjoner fra deloppgaver selv om du ikke har løst deloppgaven.
I denne oppgaven skal man lage funksjoner for å lese en tekstfil med binærkode og gjøre om dette til tegn kodet i eget kodesett for tegn og bokstaver og lagre det til en tekstfil.

Programmering 2016 kont 2a

Lag funksjonen *load$_b$in* som har en inn-parameter *filename*, som er navnet på fila som skal lastes inn. Funksjonen skal lese inn alt innholdet i fila og returnere innholdet som en tekststreng uten linjeskift eller mellomrom. Fila som det leses fra er en tekstfil bestående av binære tall (0 og 1). Hvis fila ikke eksisterer eller ikke kan åpnes, skal funksjonen returnere en tom streng samt skrive ut følgende feilmelding til skjerm: "Error: Could not open file <filename>", der <filename> er navnet på fila.

Programmering 2016 kont 2a

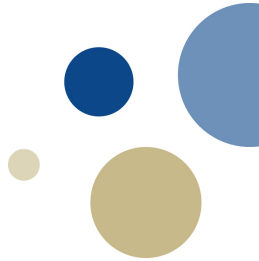


```
def load_bin(filename):
    binstring = ""
    try:
        f = open(filename, "r")

        for line in f:
            binstring += line.strip()

        f.close()
    except:
        print("Error: Could not open file", filename)
    return binstring
```

Programmering 2016 kont 2b

Lag funksjonen *bin$_t$o$_d$ec* som har en inn-parameter *binary*, som er en tekststreng av ukjent størrelse bestående av binære tall (tekststreng med nuller og enere). Funksjonen skal returnere et heltall (dvs. i titallssystemet) som tilsvarer det binære tallet angitt med tekststrengen *binary*. Oppgaven **skal ikke løses** ved hjelp av innebygde funksjoner for å oversette binærtall til heltall.

Programmering 2016 kont 2b



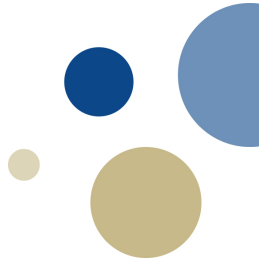
```
def bin_to_dec(binary):
    decimal = 0
    reversed_binary = binary[::-1]
    for i in range(len(reversed_binary)):
        decimal += int(reversed_binary[i]) * 2**i
    return decimal
```

Programmering 2016 kont 2c

Lag funksjonen *dec*$_t$*o*$_c$*har* som har en inn-parameter *dec*, som er et heltall med verdi mellom 0 og 31. Funksjonen skal returnere et tegn eller en bokstav avhengig av verdien av *dec*.

- Hvis *dec* har verdien 0 skal tegnet for " " (mellomrom) returneres
- Hvis *dec* har verdien 1 skal tegnet for "," (komma) returneres
- Hvis *dec* har verdien 2 skal tegnet for "." (punktum) returneres
- Hvis *dec* har en verdi mellom 3 og 31 skal en stor bokstav i det norske alfabetet returneres, der $dec = 3$ gir bokstaven "A", $dec = 4$ gir bokstaven "B", helt opp til $dec = 31$ som gir bokstaven "Å".
- For alle andre verdier av *dec* skal funksjonen returnere en tom streng.

Programmering 2016 kont 2c



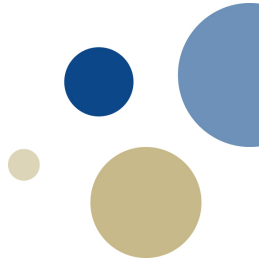
Antar denne konstanten eksisterer.
DTC = '', .ABCDEFGHIJKLMNOPQRSTUVWXYZ"

```
def dec_to_char(dec):
    if dec < len(DTC):
        return DTC[dec]
    else:
        return ""
```

Programmering 2016 kont 2d

Lag funksjonen *bin$_t$o$_t$xt* som har en inn-parameter *binstring*, som er en tekststreng av ukjent lengde bestående av binære tall (tekststreng ed nuller og enere). Funksjonen skal returnere en tekststreng bestående av bokstaver og tegn som er kodet i henhold til oppgave 2c der hvert tegn er representert med 5 bit. Inn-parameteren *binstring* vil alltid være et multiplum av fem siffer.

Programmering 2016 kont 2d



```
def bin_to_txt(binstring):
    txt = ""
    for i in range(0, len(binstring), 5):
        decimal = bin_to_dec(binstring[i:i+5])
        txt     += dec_to_char(decimal)
    return txt
```

Programmering 2016 kont 2e

Lag funksjonen *main*, uten parametere. Funksjonen skal gjøre følgende:

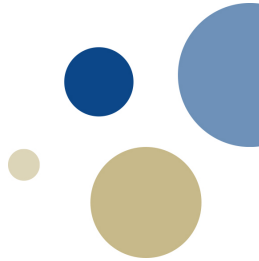
1. Skrive ut teksten "Binary-to-text converter" til skjerm
2. Spørre brukeren om navn på fil det skal lastes fra (tekstfil som inneholder binære tall) og ta vare på filnavnet i variabelen b_file.
3. Oversette tekststrengen av binære tall til tekst med bokstaver og tegn og lagre innholdet i variabelen *txt*.
4. Spørre brukeren om navnet på fil som resultatet skal lagres til og ta vare på filnavnet i variabelen t_file.
5. Skrive innholdet av variabelen *txt* til fila med filnavn angitt i variabelen t_file.
6. Skrive ut til brukeren at: "b_file has been converted and saved to t_file".

Hvis funksjonen får problemer med å skrive til fila, skal følgende feilmelding skrives: "Error: Could not write to file t_file".

Programmering 2016 kont 2e

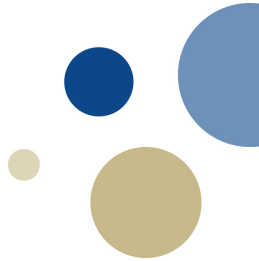
```
def main():
    print("Binary-to-text converter")
    b_file      = input("Name of binary file to load from:
                                    ")
    b_string    = load_bin(b_file)
    txt         = bin_to_txt(b_string)
    t_file      = input("Name of text file to save to: ")
    try:
        f = open(t_file, "w")
        f.write(txt)
        f.close()
        print(b_file, "has been converted and saved to",
                                          t_file)
    except:
        print("Error: Could not write to file", t_file)
```

Programmering 2013 2a



Lag funksjonen *yatzy*. Den skal ha 5 innparametere, kalt t1, t2, t3, t4 og t5. Innparamtrene representerer 5 tall mellom 1 og 6 (5 terninger). Funksjonen skal returnere ei liste som inneholder de 5 tallene i sortert rekkefølge, eller en feilmelding hvis en av tallene er større enn 6 eller mindre enn 1.

Programmering 2013 2a



```
def yatzy(t1, t2, t3, t4, t5):
    l = [t1, t2, t3, t4, t5]
    for i in range(len(l)):
        for j in range(len(l) - i - 1):
            if l[j] > 6 or l[j + 1] > 6:
                return "Ikke bruk input stoerre enn 6!"
            if l[j] < 1 or l[j + 1] < 1:
                return "Ikke bruk input mindre enn 1!"

            if l[j] > l[j + 1]:
                l[j], l[j + 1] = l[j + 1], l[j]
    return l
```

Programmering 2013 2b

Lag funksjonen *maxi$_y$atzy*. Den skal ta inn en liste med 5 eller 6 tall, og den skal returnere en skriftlig melding til brukeren som sier hvor mange terninger som ble kastet, hvilken verdi det var flest av, og hvor mange like det var av den verdien. Hvis det blir "uavgjort mellom to tall" brukes det høyeste tallet.

Programmering 2013 2b

```
def maxi_yatzy(lst):
n = len(lst)
most_val = None
most_num = 0

for base_dice in range(1, 6 + 1):
    num = 0
    for dice in lst:
        if dice == base_dice:
            num += 1
    if num >= most_num:
        most_val = base_dice
        most_num = num
return "Du kastet " + str(n) + " terninger og fikk
                                flest " + str(most_val)
                                + " (" + str(most_num) +
                                " like)."
```