



# INF100 Kræsjkurs Tekna V23

*August Tidemann Klevberg*





```
innhold = ["Basics", "Betingelser", "Funksjoner", "Strenger",  
"Lister", "Tupler", "Løkker", "Oppslagsverk", "Mengder",  
"Filer", "Exceptions", "Moduler", "Problemløsning"]
```

# 0 - Basics



# Variabler

Vi bruker variabler i Python for å lagre informasjon for senere bruk.

```
name = "Jakob"
```

```
age = 23
```

```
height = 1.80
```

```
is_student = True
```

# Dat typer

I python finnes det mange datatyper. Vi skal begynne å se på fire av dem:

- Streng (string / str)
- Heltall (integer / int)
- Desimaltall (float)
- Boolsk verdi (boolean / bool)

```
name = "Jakob"    # string
age = 23           # integer
height = 1.80      # float
is_student = True  # boolean
```

Vi kan bruke type() funksjonen for å sjekke hvilken type en verdi er.

# Kommentarer

Vi kan legge til kommentarer til koden vår. Disse vil ikke gjøre endringer når koden kjører, men det vil gjøre koden mer forståelig.

Kode kan også kommenteres ut. Da vil den ikke kjøre.

```
# Dette er en kommentar
```

```
n = 5 # dette også
```

```
# print("hei")
```

```
"""
```

```
Slik kan man  
lage en flerlinjet  
kommentar
```

```
"""
```

# Matematiske uttrykk

Vi har tilgang til de vanligste matematiske operatorene, som pluss, minus, gange, dele, potens, osv.

Det finnes også flere nyttige operasjoner som heltallsdivisjon og modulo.

$$3 + 4 == 7$$

$$8 - 3 == 5$$

$$2 * 7 == 14$$

$$16 / 4 == 4$$

$$3 ** 2 == 9$$

$$5 // 2 == 2$$

$$5 \% 2 == 1$$

# print()

Vi bruker print for å få tekst skrevet ut i terminalen

```
name = "Mari"
```

```
number_of_cats = 5
```

```
print("The name is " + name)
```

```
print(name, "has", number_of_cats, "cats")
```



The name is Mari  
Mari has 5 cats

Print funksjonen har også et sep og end argument:

```
# print(value, ..., sep = "", end = "\n")
```



# input()

Input lar oss få inn informasjon fra brukeren som kjører programmet

```
name = input("What is your name? ")
```

```
print(name)
```



# len()

len() funksjonen brukes for å finne lengden på sekvenser som strenger og lister

```
name = "Nils"  
name_length = len(name)  
  
print(name_length) # 4
```



# Konvertering av data

Det er nyttig å kunne konvertere fra en datatype til en annen. Siden `input()` funksjonen alltid gir oss en streng, vil vi konvertere i de tilfellene hvor variabelen skal holde for eksempel en `int` eller `float`. Dette gjør vi slik:

```
name = input("Enter your name: ")
```

```
age = int(input("Enter your age: "))
```

```
height = float(input("Enter your height: "))
```

# Oppgave 1

Lag en enkel kalkulator som spør brukeren om to tall a og b. Programmet skal regne ut summen  $a + b$ .

Eksempelkjøring:

Enter a number: 4

Enter another number: 8

The sum is 12

# 1 - Betingelser

# Datatype: bool

Datatypes bool har alltid en av to verdier: True eller False

For eksempel:

```
python_is_fun = True
```

```
exam_complete = False
```

# Operasjoner for sammenligning

I python har vi følgende operatorer for sammenligning: == != > < >= <=

```
"hi" == 'h' + 'i'      # True
```

```
5 == 3                 # False
```

```
4 > 2                  # True
```

```
5 < 1                   # False
```

```
4 >= 4                  # True
```

```
2 <= 3.14               # True
```

```
"cat" != "bunny"       # True
```



# Betingelse (if, else)

```
passed_inf100 = True
```

```
if passed_inf100:
```

```
    print("The student has passed INF100")
```

```
else:
```

```
    print("The student has not passed INF100")
```

—————→ The student has passed INF100

Merk at vi ikke trenger å skrive: **if passed\_inf100 == True:**



# Betingelse (if, elif, else)

Programmet kan kun havne i *en* av kodeblokkene. Først sjekkes if, og hvis den gir False går programmet videre og sjekker elif. Else fungerer som en “catch all” ettersom den blir kjørt samme hva hvis ingen av betingelsene over er True.

```
number = -5
```

```
if number > 0:
```

```
    print(number, "is positive")
```

```
elif number < 0:
```

```
    print(number, "is negative")
```

```
else:
```

```
    print("The number is 0")
```

# Betinget verdi

```
number = 5
```

```
is_even = (number % 2 == 0)
```

```
print("The number is " + ("even" if is_even else "odd"))
```



The number is odd

# and, or, not operatorer

Disse operatorene  
lar oss kombinere  
betingelser.

True and True	# True
False and True	# False
False and False	# False

True or False	# True
True or True	# True
False or False	# False

not True	# False
not (not True)	# True
True and not False	# True
not True or False	# False

# Presedens

Dette forteller i hvilken rekkefølge operasjoner gjøres.

Operatorer	
<code>()</code>	Parentes. Det som er mellom parentesene evalueres før en operator utenfor parentesen.
<code>**</code>	Ekspontiering. Assosierer høyre-til-venstre.
<code>*</code> <code>/</code> <code>//</code> <code>%</code>	Multiplikasjon, divisjon og modulo.
<code>+</code> <code>-</code>	Addisjon og subtraksjon.
<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code> <code>==</code> <code>!=</code> <code>in</code> <code>not</code> <code>in</code>	Relasjoner. Disse vil <i>ikke</i> assosiere verken til høyre eller venstre; dersom man har flere slike etter hverandre vil de <i>komponeres</i> som en konjunksjon i stedet. For eksempel, <code>-1 &lt; 0 == False</code> gir det samme svaret som <code>-1 &lt; 0 and 0 == False</code> , og altså ikke det samme som <code>(-1 &lt; 0) == False</code> slik man ellers kunne trodd.
<code>not</code>	Logisk negasjon.
<code>and</code>	Logisk konjunksjon.
<code>or</code>	Logisk disjunksjon.
<code>if else</code>	Betinget verdi.

# Matematisk presedens

Python følger stort sett samme rekkefølge av operasjoner som vi gjør i matte

```
3 + 2*4      # 11
```

```
4 + 6%3      # 4
```

```
3**2*2       # 18
```

Alle tegn assosierer venstre til høyre med unntak av potens (\*\*)

```
3 - 2 - 1     # 0
```

```
3**3**2       # 19683
```

# Logisk presedens

Husk at av de logiske operatorene har <, >, (osv...) og 'in' høyest presedens. Etter det kommer 'not', så kommer 'and', etterfulgt av 'or'.

True or True and False  
True or (True and False)  
not False or True  
(not False) or True  
not (False or True)

# True  
# Use parentheses!  
# True  
# True  
# False

2 < 3 < 4  
not 3 < 2 < 1  
not 3 < 2 and 1 < 2

# True  
# True  
# True

"e" in "hello"  
"a" and "e" in "hello"

# True  
# True

# Truthy og falsy verdier

Alle verdier i Python har en gjemt egenskap. Enten er verdien *truthy*, noe som betyr at man får `True` dersom man konverterer den til en `bool`, ellers er den *falsy*, noe som gir `False`. Dette lar oss bruke verdien alene som en betingelse. Det er ganske lett å vite hva som er *truthy* og *falsy*. Regelen er at alle "tomme" verdier (som `int 0`, `float 0.0`, `str ""`, `list []`, `None`, osv) er *falsy*, og alt annet er *truthy*.

```
user_input = input("Write something: ")
```

```
if user_input:
```

```
    print(f"The user wrote \"{user_input}\"")
```

```
else:
```

```
    print("The user did not write anything")
```

Write something: test  
The user wrote "test"

Write something:  
The user did not write anything

# Oppgave 2

Utvid kalkulatoren slik at den først spør brukeren om to tall, så en matematisk operasjon. Programmet skal bruke operasjonen for å regne ut resultatet.

Eksempelkjøring:

Enter a number: 3

Enter another number: 8

Pick an operation (add, subtract, multiply, divide): multiply

The answer is 24



## 2 - Funksjoner

# Syntaks

For å definere en ny funksjon skriver vi def, etterfulgt av navnet på funksjonen og ett sett med parenteser.

Funksjonen har en kodeblokk som viser hvilken kode som er del av funksjonen. Dette er representert med innrykk.

Når vi kaller på funksjonen skriver vi navnet på funksjonen med et sett med parenteser.

```
def function_name():  
    print("Hello from within the function!")
```

```
print("Hello from outside the function!")
```

```
function_name()
```

```
print("Hello again from outside the function!")
```

# Argumenter og parametere

Disse brukes for å kommunisere med funksjoner. Argumenter er det som blir sendt inn til en funksjon når man kaller på den. Parametere er det som funksjonen tar inn av data.

```
def print_n_plus_two(n):  
    print(n + 2)
```

→ 7

```
number = 5  
print_n_plus_two(number)
```

# Return-verdi

Return-verdien er verdien som funksjonen sender tilbake til der hvor funksjonen ble kalt på.

```
def squared(x):  
    return x**2
```

```
clementines = 2
```

```
print(squared(clementines)) # 4
```

```
clementines = squared(clementines) # 4
```

```
clementines = squared(clementines) # 16
```

```
clementines = squared(clementines) # 256
```

```
clementines = squared(clementines) # 65536
```



# Innebygde funksjoner

Innebygde funksjoner er verktøy som vi kan bruke i koden vår

Noen eksempler:

- `print()`, `input()`, `len()`, `type()`
- `str()`, `int()`, `float()`, `bool()`, `list()`
- `abs()`, `max()`, `min()`, `round()`, `sum()`
- `range()`, `enumerate()`, `zip()`

Du kan lese om hver av disse her:

[https://www.w3schools.com/python/python\\_ref\\_functions.asp](https://www.w3schools.com/python/python_ref_functions.asp)

# Oppgave 3

Hva vil denne koden printe ut?



```
def add(a, b=4):
```

```
    a += b
```

```
    return a
```

```
def find_sum_of_abc(a):
```

```
    b = add(a*2)
```

```
    c = add(b, a)
```

```
    return a + b + c
```

```
print(find_sum_of_abc(2))
```

3 - Strenger

# Bygge strenger

```
message = name + " is " + str(age) + " years old and " + str(height) + " meters tall."
```

Merk at vi må konvertere **int** og **float** til **string**. Vi bruker **str()** funksjonen.

For at teksten skal bli fin må vi ta med mellomrom før og etter variablene.



# f-strenger

f-strenger lar oss enklere bygge en streng

```
message = f"{name} is {age} years old and {height} meters tall."
```

```
message += f" {name} has {'not ' if not has_passed else ''}passed INF100."
```

Merk:

- vi trenger ikke å konvertere variabler til strenger.
- vi bruker apostrof for betinget verdi.

# Jobbe med strenger

Escape tegn

```
print("start\n\t\\end")
```

→ start  
    \\end

Sette sammen strenger

```
"abc" + "def" # "abcdef"
```

```
"abc" * 3 # "abcabcabc"
```

Medlemskap

```
"a" in "abc" # True
```

```
"bc" in "abc" # True
```

```
"ac" in "abc" # False
```



# Hente ut tegn

For å hente ut et tegn fra en streng bruker vi hakeparenteser bak strengen, og skriver inn en indeks.

En indeks er posisjonen til et element i en sekvens. I python og de fleste programmeringsspråk begynner vi å telle fra 0.

```
s = "Hello, World!"
```

```
print(s[0])    # H
```

```
print(s[5])    # ,
```

```
print(s[-1])   # !
```

```
print(s[-4])   # r
```

# Slicing

Slicing fungerer veldig likt, men vi bruker kolon for å skille mellom start og slutt verdi, og evt steg.



```
s = "Hello, World!"
```

```
# s[start:stop]
```

```
# s[start:stop:step]
```

```
print(s[0:5])      # Hello
```

```
print(s[:5])       # Hello
```

```
print(s[7:])       # World!
```

```
print(s[0:5:2])    # Hlo
```

```
print(s[::2])      # Hlo ol!
```

# Reversere en streng

For å reversere en streng setter vi bare steg-delen til -1, slik:

```
s = "Hello, World!"
```

```
print(s[::-1]) # !dlroW ,olleH
```

# Streng-metoder

Metoder er (mye som innebygde funksjoner) verktøy som vi kan bruke i Python. Men de brukes på en litt annen måte. Forskjellige datatyper har forskjellige tilgjengelige metoder.

Det finnes mange nyttige metoder som vi kan bruke på strenger.

# Streng-metode .upper(), .lower(), .capitalize() og .title()

```
s = "hello, WORLD!"
```

<code>print(s.upper())</code>	→	HELLO, WORLD!
-------------------------------	---	---------------

<code>print(s.lower())</code>	→	hello, world!
-------------------------------	---	---------------

<code>print(s.capitalize())</code>	→	Hello, world!
------------------------------------	---	---------------

<code>print(s.title())</code>	→	Hello, World!
-------------------------------	---	---------------

# Streng-metoder .replace()

Med .replace() metoden kan vi bytte ut en substreng av en streng med en annen streng. Den fungerer slik:

```
food = "salmon"
```

```
print(food.replace("mon", "ad"))
```

salad







# Streng-metode .strip()

.strip() metoden fjerner whitespace på hver ende av en streng

Før:	<pre>s = ' hello \n' print(s) print(repr(s)) print("length:", len(s))</pre>	→	<pre>hello ' hello \n' length: 11</pre>
Etter:	<pre>s = s.strip() print(s) print(repr(s)) print("length:", len(s))</pre>	→	<pre>hello 'hello' length: 5</pre>

# Streng-metode .split()

.split() deler opp en streng inn i en liste. Den splitter på det tegnet som blir sendt inn som argument. Hvis ingen argument blir sendt inn splitter den på whitespace (mellomrom, newline, osv).

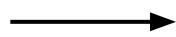
```
sentence = "Once upon a time"
```

```
print(sentence.split())
```



```
['Once', 'upon', 'a', 'time']
```

```
print(sentence.split("n"))
```



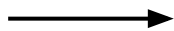
```
['O', 'ce upo', ' a time']
```

# Streng-metode .join()

.join() metoden tar en liste med strenger og setter den sammen til en streng. Strengen som kommer før .join() forteller hva som skal settes mellom hver streng. Siden vi har satt en mellomrom her og vi har en liste med ord, får vi en setning.

```
sentence = " ".join(["Once", "upon", "a", "time"])
```

```
print(sentence)
```



Once upon a time

# Streng som liste

Når vi programmerer i python kan det noen ganger hjelpe å tenke på en streng som en liste av tegn. Strenger og lister har, som vi kommer til å se, mange like egenskaper.

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



```
['H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

# 4 - Lister

# Lister

```
numbers = [1, 2, 3, 4, 5]
```

```
bools = [True, False, True]
```

```
misc = [1, "a", True, 3.14]
```

```
fruits = ["apple", "banana", "cherry"]
```

```
if "cherry" in fruits:
```

```
    print("Cherry is in the list")
```

```
else:
```

```
    print("Cherry is not in the list")
```



Cherry is in the list

# Hente ut elementer

Vi henter ut elementer fra lister på samme måte som vi henter ut tegn fra strenger

```
animals = ["elephant", "tiger", "giraffe", "lion"]
```

```
print(animals[3])      →      lion
```





# Slicing

Slicing fungerer  
også på lister  
som på strenger

```
fruits = ["apple", "banana", "cherry", "pear", "mango"]
```

```
# fruits[start:stop]
```

```
# fruits[start:stop:step]
```

```
print(fruits[1:3])    # ["banana", "cherry"]
```

```
print(fruits[:3])     # ["apple", "banana", "cherry"]
```

```
print(fruits[2:])     # ["cherry", "pear", "mango"]
```

```
print(fruits[::-2])   # ["apple", "cherry", "mango"]
```

```
print(fruits[::-1])   # ["mango", "pear", "cherry", "banana", "apple"]
```

# Unpacking

Å 'unpacke' en liste eller en tuppel lar oss lagre innholdet i egne variabler

```
city, country, population = ["Oslo", "Norway", 1_000_000]
```

```
print(f"{city} is in {country} and has a population of {population}")
```

Alternativet til 'unpacking' blir ofte å hente ut elementene med indeksering

```
details = ["Oslo", "Norway", 1_000_000]
```

```
city = details[0]
```

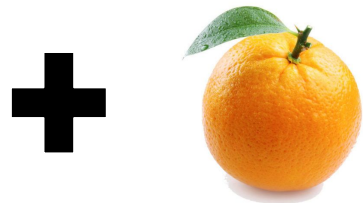
```
country = details[1]
```

```
population = details[2]
```

# Liste-metode .append() og .remove()

.append() legger til et element på slutten av listen

```
fruits = ["apple", "banana", "cherry"]
```



```
fruits.append("orange")
```

```
print(fruits)
```

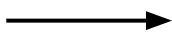


```
['apple', 'banana', 'cherry', 'orange']
```

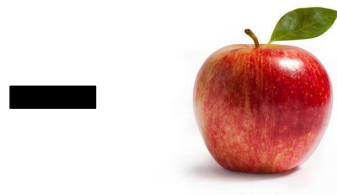
.remove() fjerner et spesifikt element fra listen

```
fruits.remove("apple")
```

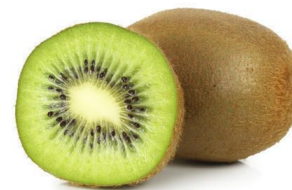
```
print(fruits)
```



```
['banana', 'cherry', 'orange']
```



# Liste-metode .insert() og .pop()



.insert() legger til et element på en spesifikk indeks i listen

```
fruits = ["apple", "banana", "cherry"]
```

```
fruits.insert(2, "kiwi")
```

```
print(fruits)
```



```
['apple', 'banana', 'kiwi', 'cherry']
```

.pop() fjerner et element fra en spesifikk indeks i listen. Hvis man ikke oppgir en indeks vil det siste elementet i listen bli fjernet

```
fruits.pop(1)
```

```
print(fruits)
```



```
['apple', 'kiwi', 'cherry']
```

# Liste-metode .extend()

.extend() metoden setter sammen to lister

```
fruits = ["apple", "banana", "cherry"]
```

```
fruits.extend(["mango", "papaya"])
```

```
print(fruits)
```



```
['apple', 'banana', 'cherry', 'mango', 'papaya']
```



Man kan også gjøre det samme ved å skrive:

```
fruits += ["mango", "papaya"]
```

# Liste-metode .sort()

.sort() metoden sorterer listen fra lavest til høyest verdi

```
numbers = [6, 2, 8, 3, 1]
```

```
numbers.sort()
```

```
print(numbers)
```



[1, 2, 3, 6, 8]

Om vi gir reverse=True som argument får vi listen sortert fra høyest til lavest verdi

```
numbers.sort(reverse=True)
```

```
print(numbers)
```



[8, 6, 3, 2, 1]

# Liste-metode .reverse()

.reverse() reverserer listen

```
numbers = [6, 2, 8, 3, 1]
```

```
numbers.reverse()
```

```
print(numbers)
```



```
[1, 3, 8, 2, 6]
```

# Liste-metode .index()

.index() metoden gir oss indeksen av et spesifikt element i listen med frukt

```
fruits = ["apple", "banana", "cherry", "pear"]
```

```
print(fruits.index("cherry"))
```



2



# Liste-metode .count()

.count() metoden gir oss antallet av et spesifikt element i listen



```
fruits = ["pineapple", "banana", "cherry", "pineapple", "pear"]
```

```
print(fruits.count("apple"))
```



0

```
print(fruits.count("pineapple"))
```



2



# Ikke-destruktive funksjoner

En ikke-destruktiv funksjon er en funksjon som tar inn en liste, men ikke gjør endringer på den samme listen. Funksjonen i eksempelet under er ikke destruktivt, siden den kun går gjennom elementene i listen `nums`, og gjør endringer i en annen liste.

```
def double(nums):  
  
    new_list = []  
  
    for num in nums:  
        new_list.append(num * 2)  
  
    return new_list
```

# Destruktive funksjoner

I motsetning til ikke-destruktive funksjoner tar en destruktiv funksjon inn en liste og gjør endringer på denne samme listen direkte.

```
def double(nums):  
    for i in range(len(nums)):  
        nums[i] *= 2
```



Her trenger vi ikke å returnere listen nums, siden lister er mutable. Listen blitt endret på også utenfor funksjonen.

# List comprehensions

'List comprehensions' kan brukes som en snarvei for å bygge lister. Følgende list comprehension gjør det samme som den ikke-destruktive funksjonen `double()`.

```
def double(nums):  
    return [num * 2 for num in nums]
```

Vi kan gjøre mye på en linje med bruk av list comprehensions.

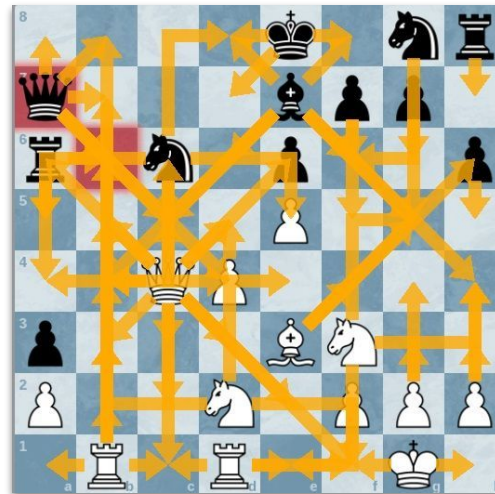
```
# The sum of all numbers up to 1000 that are divisible by 3 or 5  
print(sum([num for num in range(1000) if num % 3 == 0 or num % 5 == 0]))
```

Vi kan gjøre enda mer med list comprehensions, men det er ikke nødvendig å bruke dem på eksamen.

# 2D-lister

Lister kan også inneholde andre lister

```
board = [  
    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],  
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],  
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],  
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],  
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],  
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],  
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],  
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']  
]
```



5 - Tupler

# Tupler

En tuppel er som en liste, men den er immutable, som vil si at den ikke kan endres. For eksempel kan det gi mening å representere et sett med 2D koordinater som en tuppel, siden den alltid skal ha to elementer, hvor den første er x-verdi og den andre er y-verdi.

`coordinate = [3, 4]` ❌


`coordinate = (3, 4)` ✓

Å bruke tupler hvor det gir mening gjør koden sikrere, siden vi vet at tupplen ikke vil endre på seg.


# Tuppel-metoder

```
my_tuple = (1, 6, 4, 3, 1, 2, 4, 1)
```

```
print(my_tuple.count(1))
```

 3

```
print(my_tuple.index(4))
```

 2



# 6 - Løkker



# While-løkker

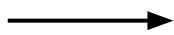
En while-løkke kjører så lenge en betingelse er sann (True)

```
n = 1
```

```
while n < 5:
```

```
    print('n =', n)
```

```
    n += 1
```



```
n = 1
```

```
n = 2
```

```
n = 3
```

```
n = 4
```

```
Done
```

```
print('Done')
```

# For-løkker

En for-løkke brukes når vi vet akkurat hvor mange ganger løkken skal kjøre.

```
animals = ["dog", "cat", "bird"]
```

```
for animal in animals:  
    print(animal)
```



```
dog  
cat  
bird
```

```
string = "Hi!"
```

```
for letter in string:  
    print(letter)
```



```
H  
i  
!
```

# range()

Range gir oss en "liste" med tall som teller på en spesifikk måte. For eksempel

`range(5)`  $\longrightarrow$  `[0, 1, 2, 3, 4]`

`range(4, 8)`  $\longrightarrow$  `[4, 5, 6, 7]`

range() funksjonen kan ta opp til tre argumenter, men den kan også ta to og en:

```
# range(start, stop, step)
```

```
# range(start, stop)      <- step = 1
```

```
# range(stop)           <- start = 0, step = 1
```

# For-løkker med range()

Vi bruker ofte range() i sammenheng med for-løkker for å telle gjennom indeksene i en sekvens.

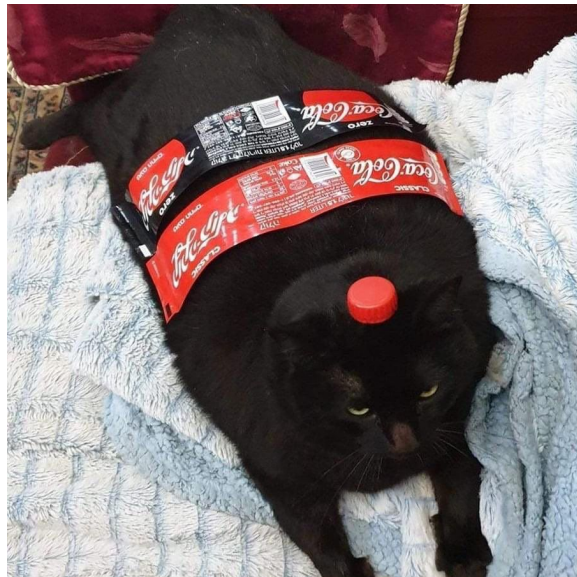
```
soda = ["Pepsi", "Coke", "Sprite"]
```

```
for i in range(len(soda)):
```

```
    print(i, soda[i])
```



0 Pepsi  
1 Coke  
2 Sprite



# enumerate()

Det er ofte vi vil jobbe med element fra en liste og samtidig vite hvilken indeks den har i listen. I stedet for å bruke range() kan vi bruke enumerate()

```
soda = ["Pepsi", "Coke", "Sprite"]
```

```
for i, drink in enumerate(soda):  
    print(i, drink)
```



```
0 Pepsi  
1 Coke  
2 Sprite
```

# continue & break

continue hopper rett til neste iterasjon av en løkke



```
for i in range(10):  
    if i == 5:  
        continue  
    print(i)
```

while True:

```
    user_input = input("Enter a number: ")
```

```
    if user_input == "quit":
```

```
        break
```

```
    total += int(user_input)
```



break hopper ut av en løkke

```
print("The total is", total)
```


# zip()

zip() funksjonen henter ut elementer i samme posisjon fra flere lister og plasserer dem i tupler i en ny liste

```
xs = [3, 2, 6]
```

```
ys = [2, 4, 1]
```

```
print( list( zip(xs, ys) ) )
```



```
[(3, 2), (2, 4), (6, 1)]
```

```
for x, y in zip(xs, ys):
```

```
    print(f"{x}, {y}")
```



```
3, 2
```

```
2, 4
```

```
6, 1
```



# Nøstede løkker

Nøstede løkker vil si at vi har en løkke inne i en annen løkke. Dette gjør vi ofte når vi skal gå gjennom alle elementer i en 2D-liste.

```
rows = 3
```

```
cols = 3
```

```
for row in range(rows):
```

```
    print("Outer loop. Row:", row)
```

```
    for col in range(cols):
```

```
        print("- Inner loop. Col:", col)
```



Outer loop. Row: 0

- Inner loop. Col: 0

- Inner loop. Col: 1

- Inner loop. Col: 2

Outer loop. Row: 1

- Inner loop. Col: 0

- Inner loop. Col: 1

- Inner loop. Col: 2

Outer loop. Row: 2

- Inner loop. Col: 0

- Inner loop. Col: 1

- Inner loop. Col: 2

# Oppgave 4

Ved å bruke en løkke, skriv et program som teller hvor mange unike tall som finnes i en liste.

Bruk gjerne koden til høyre for å komme i gang. Funksjonen skal være ikke-destruktiv.

```
numbers = [1, 6, 2, 9, 4, 2, 1]
```

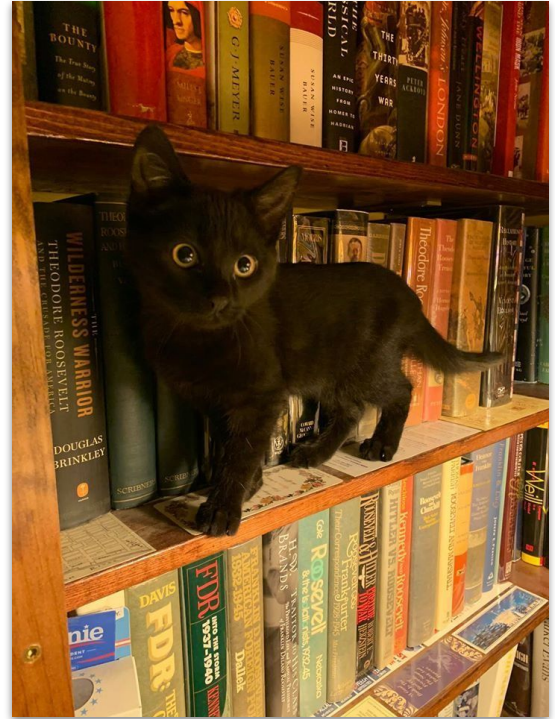
```
def unique_words(numbers):
```

```
    # Your code here
```

```
    ...
```

```
print(unique_words(numbers))
```

# 7 - Oppslagsverk



# Oppslagsverk (dictionaries)

Et oppslagsverk er en datastruktur som bruker nøkkel/verdi. Når vi henter ut verdier bruker vi nøkkelen, ikke posisjonen som en indeks.

```
pet_translator = {  
    'woof': 'Feed me, human!',  
    'meow': 'Pet me, but only exactly 3 times',  
    'chirp': 'Clean my cage, please',  
    'hiss': 'You are not my favorite human right now',  
}
```

Som navnet vil foreslå er det praktisk å bruke oppslagsverk når man trenger slå-opp type funksjonalitet.

# Hente ut verdier fra oppslagsverk

Man bruker en nøkkel for å hente ut en verdi

```
pet_phrase = 'woof'  
print(f"When your pet says '{pet_phrase}', it means '{pet_translator[pet_phrase]}'")
```



When your pet says 'woof', it means 'Feed me, human!'

Men man må være litt forsiktig. Hvis ikke nøkkel eksisterer i oppslagsverket får man `KeyError`. Derfor kan det noen ganger være lurt å bruke `.get()` metoden, som tillater en default verdi hvis ikke nøkkelen eksisterer.

# Oppslagsverk-metode .get()

Vi kan bruke .get() for å hente ut verdier på den trygge måten. Metoden tar en nøkkel og en valgfri verdi som argument. Det andre argumentet er hva metoden skal returnere dersom nøkkelen ikke eksisterer i oppslagsverket.

```
print(pet_translator.get("ribbit", "Translation not found"))
```



Translation not found

# Gjøre endringer til oppslagsverk

Vi kan legge til eller oppdatere et element ved å skrive følgende:

```
pet_translator['meow'] = "Pet me, but only if I'm in the mood"  
print(pet_translator)
```



```
{'woof': 'Feed me, human!', 'meow': "Pet me, but only  
if I'm in the mood", 'chirp': 'Clean my cage, please',  
'hiss': 'You are not my favorite human right now'}
```

# Oppslagsverk-metode `.keys()` og `.values()`


`.keys()` metoden gir oss en "liste" som inneholder alle nøklene i et oppslagsverk.

```
print( list( pet_translator.keys() ) )
```

 → `['woof', 'meow', 'chirp', 'hiss']`

`.values()` metoden gir oss en "liste" med alle verdier i et oppslagsverk.

```
print( list( pet_translator.values() ) )
```



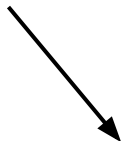
`['Feed me, human!', 'Pet me, but only if I'm in the mood', 'Clean my cage, please', 'You are not my favorite human right now']`



# Oppslagsverk-metode .items()

.items() metoden gir oss parvis nøkkel og verdi som en tuppel lagret i en "liste"

```
print( list( pet_translator.items() ) )
```



```
[('woof', 'Feed me, human!'), ('meow', "Pet me, but only if I'm in the mood"), ('chirp',  
'Clean my cage, please'), ('hiss', 'You are not my favorite human right now')]
```

# Fjerne ting fra oppslagsverk

For å fjerne nøkkel/verdi par fra et oppslagsverk kan vi bruke 'del' for delete, slik:

```
del pet_translator['chirp']  
print(pet_translator)
```



```
{'woof': 'Feed me, human!', 'meow': "Pet me, but only if I'm in  
the mood", 'hiss': 'You are not my favorite human right now'}
```

# 8 - Mengder



# Mengder (sets)

Mengder er som lister, men alle elementene er unike

```
numbers = {1, 4, 2, 1, 2}
print(numbers)
```

→ {1, 2, 4}

Derfor kan vi fjerne duplikate elementer fra en liste ved å konvertere den til en mengde, så tilbake til en liste

```
numbers = [1, 4, 2, 1, 2]
print( list( set(numbers) ) )
```

→ [1, 2, 4]

# Mengde-metode .add()

.add() metoden legger til et element i en mengde.

```
numbers = {1, 2, 3}
```

```
numbers.add(4)
```

```
print(numbers)
```



```
{1, 2, 3, 4}
```

# Mengde metode .remove()

Denne metoden fjerner et spesifikt element fra en mengde

```
numbers = {1, 2, 3}
```

```
numbers.remove(2)
```

```
print(numbers)
```



```
{1, 3}
```

# Mengde-operator: -

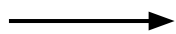
Som vi sikkert er kjent med fra set theory finnes det diverse operasjoner som vi kan bruke på mengder.

$x - y$  gir oss en mengde som inneholder kun elementer fra en mengde  $x$  som ikke er i den andre mengden  $y$ .

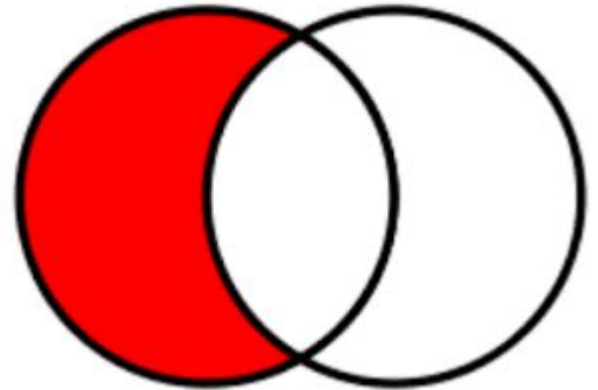
```
x = {1, 2, 3}
```

```
y = {2, 3, 4}
```

```
print(x - y)
```



```
{1}
```



# Mengde-metode .union()

`x.union(y)` gir oss en mengde som inneholder alle elementer fra to mengder `x` og `y`.

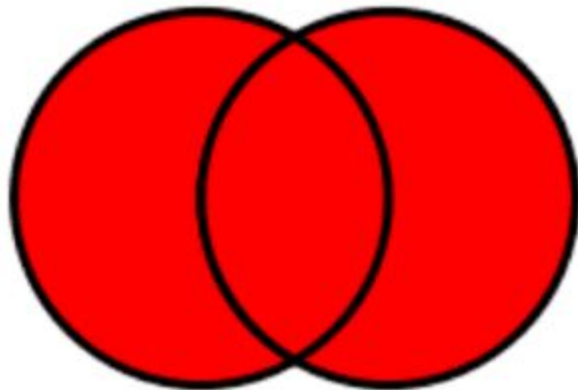
```
x = {1, 2, 3}
```

```
y = {2, 3, 4}
```

```
print(x.union(y))
```



```
{1, 2, 3, 4}
```





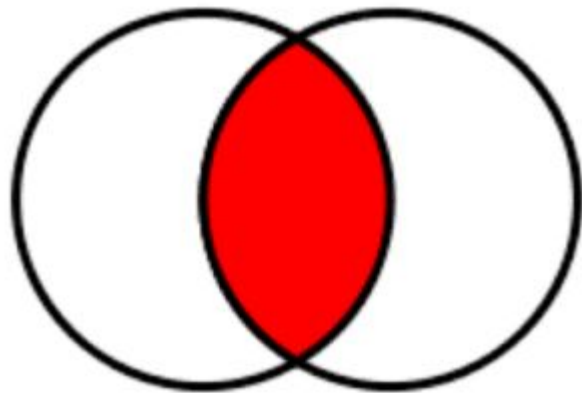
# Mengde-metode .intersection()

`x.intersection(y)` gir oss en mengde som kun inneholder elementer som to mengder `x` og `y` har til felles.

```
x = {1, 2, 3}
```

```
y = {2, 3, 4}
```

```
print(x.intersection(y)) → {2, 3}
```



# Mengde-metode .issubset()

`x.issubset(y)` returnerer True dersom alle elementene i mengden `x` eksisterer i mengden `y`. Ellers returnerer den False.

```
x = {1, 2, 3}
```

```
y = {2, 3, 4}
```

```
print(x.issubset(y))
```

 False

```
x = {2, 3}
```

```
y = {2, 3, 4}
```

```
print(x.issubset(y))
```

 True

# Oppgave 5

Ved å bruke mengder, skriv et program som teller hvor mange unike ord som finnes i en tekst.

Bruk gjerne koden til høyre for å komme i gang.

```
text = "hello test one two three test"
```

```
def unique_words(text):
```

```
    # Code goes here
```

```
    ...
```

```
print(unique_words(text))
```

# 9 - Filer

# Jobbe med filer

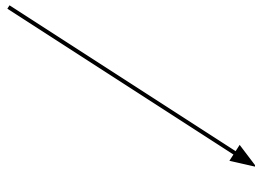
I følgende eksempler skal vi bruke en test-fil som ser slik ut:

```
≡ test.txt
```

```
1   dette er linje 1
2   så kommer linje 2
3   og til slutt linje 3
4
```

# Gå gjennom fil linje for linje

```
with open("test.txt", "r", encoding="utf-8") as f:  
    for line in f:  
        print(repr(line))
```



```
'dette er linje 1\n'  
'så kommer linje 2\n'  
'og til slutt linje 3\n'
```

# Lese en fil som *en* streng

```
with open("test.txt", "r", encoding="utf-8") as f:  
    data = f.read()  
    print(repr(data))
```



'dette er linje 1\nså kommer linje 2\nog til slutt linje 3\n'

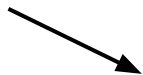


# En liste med hver linje som element

```
with open("test.txt", "r", encoding="utf-8") as f:
```

```
    lines = f.readlines()
```

```
    print(lines)
```

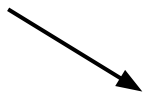


['dette er linje 1\n', 'så kommer linje 2\n', 'og til slutt linje 3\n']

```
with open("test.txt", "r", encoding="utf-8") as f:
```

```
    lines = f.read().splitlines()
```

```
    print(lines)
```



['dette er linje 1', 'så kommer linje 2', 'og til slutt linje 3']



# En liste med alle ord som element

```
with open("test.txt", "r", encoding="utf-8") as f:  
    words = f.read().split()  
    print(words)
```



```
['dette', 'er', 'linje', '1', 'så', 'kommer', 'linje', '2', 'og', 'til', 'slutt', 'linje', '3']
```

# Hente ut en linje av gangen

```
with open("test.txt", "r", encoding="utf-8") as f:
```

```
    line_one = f.readline()
```

```
    line_two = f.readline()
```

```
    the_rest = f.read()
```

```
print(repr(line_one))
```

```
print(repr(line_two))
```

```
print(repr(the_rest))
```



'dette er linje 1\n'  
'så kommer linje 2\n'  
'og til slutt linje 3\n'

# Skrive inn i fil (fjerner tidligere innhold)

```
with open("test.txt", "w", encoding="utf-8") as f:  
    f.write("Hello!")
```



```
≡ test.txt  
1   dette er linje 1  
2   så kommer linje 2  
3   og til slutt linje 3  
4
```



```
≡ test.txt  
1   Hello!
```

# Skrive inn i fil (legger til på slutten)

```
with open("test.txt", "a", encoding="utf-8") as f:  
    f.write("Hello!")
```

≡ test.txt

```
1  dette er linje 1  
2  så kommer linje 2  
3  og til slutt linje 3  
4
```



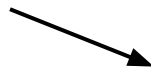
≡ test.txt

```
1  dette er linje 1  
2  så kommer linje 2  
3  og til slutt linje 3  
4  Hello!
```

# Skrive liste med strenger inn i fil

```
lines = ['linje 1\n', 'linje 2\n', 'linje 3\n']
```

```
with open("test.txt", "w", encoding="utf-8") as f:  
    f.writelines(lines)
```




≡ test.txt

```
1  linje 1  
2  linje 2  
3  linje 3  
4
```

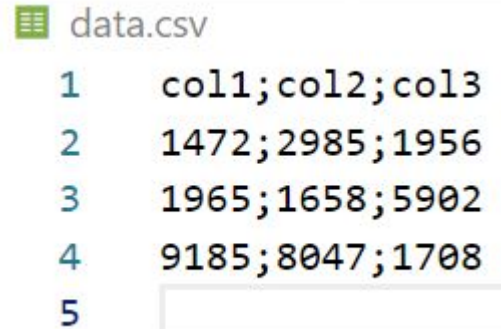
# CSV fil: Gå gjennom alle radene

```
import csv
```

```
with open('data.csv', 'r', encoding='utf-8') as file:  
    reader = csv.reader(file, delimiter=';')  
    for row in reader:  
        print(row)
```



```
['col1', 'col2', 'col3']  
['1472', '2985', '1956']  
['1965', '1658', '5902']  
['9185', '8047', '1708']
```



```
data.csv  
1 col1;col2;col3  
2 1472;2985;1956  
3 1965;1658;5902  
4 9185;8047;1708  
5
```

# CSV fil: Skrive inn en rad

```
with open('data.csv', 'w', encoding='utf-8') as file:  
    writer = csv.writer(file, delimiter=';')  
    writer.writerow(['one', 'two', 'three'])
```

data.csv

1	col1;col2;col3
2	1472;2985;1956
3	1965;1658;5902
4	9185;8047;1708
5	





data.csv

1	one;two;three
2	

# CSV fil: Skrive inn flere rader

```
rows = [['col1', 'col2', 'col3'],  
        ['1472', '2985', '1956'],  
        ['1965', '1658', '5902'],  
        ['9185', '8047', '1708']]
```

```
with open('data.csv', 'w', encoding='utf-8', newline='') as file:  
    writer = csv.writer(file, delimiter=';')  
    writer.writerows(rows)
```

 data.csv  
1 one;two;three  
2 data.csv  
1 col1;col2;col3  
2 1472;2985;1956  
3 1965;1658;5902  
4 9185;8047;1708  
5



# 10 - Exceptions



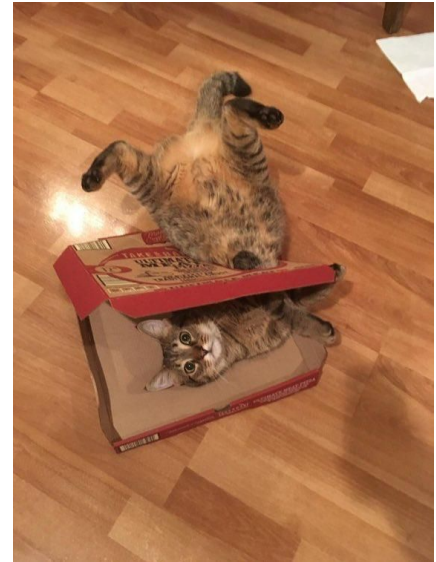
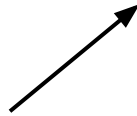
# try / except

Try og except gir oss en måte å håndtere kræsje på. Hvis koden i try blokken leder til error kjører python koden i except i stedet for å kræsje.

For eksempel:

```
try:  
    print(5/0)  
except:  
    print("Something went wrong!")
```

Something went wrong!



# try / except - ValueError

Et godt eksempel på try / except er når vi vil få inn et heltall fra en bruker. Hvis vi gjør det slik som vi har gjort det før, får vi error dersom brukeren skriver inn noe som ikke kan konverteres til en int.

```
number = int( input("Enter a whole number: ") )
```

Derfor lønner det seg å bruke try / except her. Det kan se slik ut:

```
while True:
    try:
        number = int( input("Enter a whole number: ") )
        break
    except ValueError:
        print("That's not a whole number!")
```

# Exceptions

Andre viktige exceptions inkluderer  
KeyError, IndexError og FileNotFoundError

# 11 - Moduler

# Egne moduler

main.py

```
1 import my_module
2
3 my_module.panic()
```

→ baXftgqT!!

my\_module.py > ...

```
1 import random
2 import string
3
4 def panic():
5     print(''.join(random.choices(string.ascii_letters, k=8)) + '!!!')
```



# Egne moduler

Det kan være problematisk å ha kode i det globale scope når man lager en modul. Dette er fordi all den koden vil kjøre når man importerer modulen i en annen fil. Derfor plasserer vi denne koden i en if-setning som ser slik ut:

```
def hello():
```

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



```
hello() # testing my module
```

```
def hello():
```

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



```
if __name__ == "__main__":
```

```
    hello() # testing my module
```

# Standardbiblioteket

Standardbiblioteket er flere verktøy som vi kan bruke i Python. Men disse må først importeres. Noen populære biblioteker er random, math, datetime og csv.

```
import math
```

```
from random import choice
```

```
r = 4
```

```
print(choice(["red", "green", "blue"]))
```

```
print(math.pi*r**2)
```

Vanligvis pleier man å importere alt man trenger øverst i filen.



# Eksterne pakker

Pakker som ikke ligger i standardbiblioteket må installeres. Da bruker vi pip. Matplotlib er et eksempel på en ekstern pakke. Instruksjoner for å installere den finnes på deres nettside:

[https://matplotlib.org/stable/users/getting\\_started/](https://matplotlib.org/stable/users/getting_started/)

# Plotting av grafer

Det å plote grafer er ganske greit. Man trenger en liste for x'er og en liste for y'er. Disse to listene må være like lange. Tallet i indeks 0 av x-listen hører sammen med det samme elementet fra y-listen.

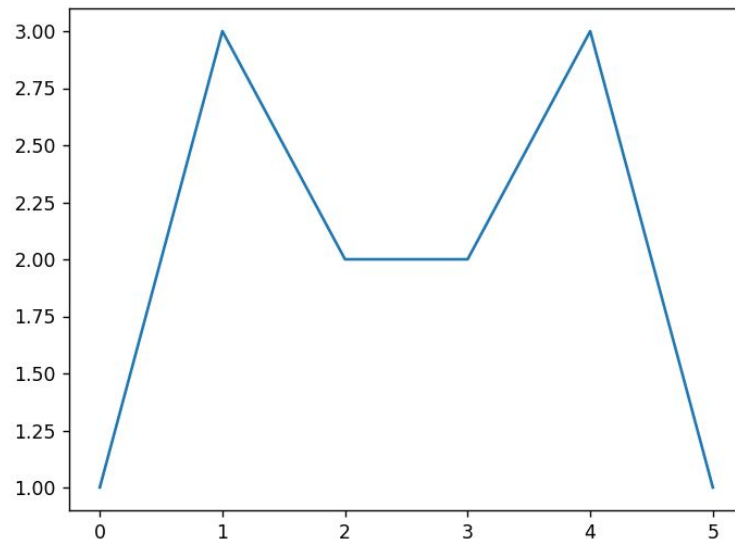
```
import matplotlib.pyplot as plt
```

```
xs = [0, 1, 2, 3, 4, 5]
```

```
ys = [1, 3, 2, 2, 3, 1]
```

```
plt.plot(xs, ys)
```

```
plt.show()
```



# Lage et søylediagram

For å tegne et søylediagram trenger vi kun endre på en linje, nemlig bytte `.plot()` med `.bar()`

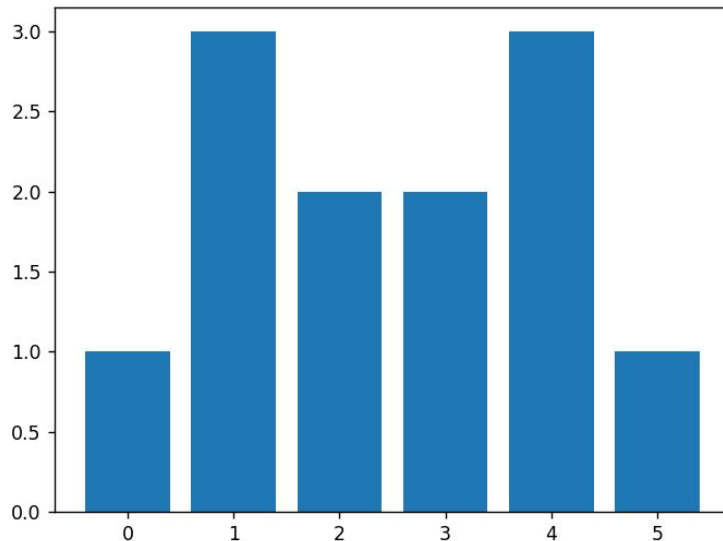
```
import matplotlib.pyplot as plt
```

```
xs = [0, 1, 2, 3, 4, 5]
```

```
ys = [1, 3, 2, 2, 3, 1]
```

```
plt.bar(xs, ys)
```

```
plt.show()
```



# Plotting av spredningsdiagram

For å plotte en spredningsdiagram kan vi skrive en for-løkke med `zip()` og bruke `.scatter()` metoden, slik:

Hvis du jobber med mye data, kan `.scatter()` være veldig treg. Derfor kan du alternativt bruke `.plot()` metoden slik som vist i kommentaren.

```
import matplotlib.pyplot as plt
```

```
xs = [7, 1, 9, 2, 5, 1, 3, 5, 8, 2, 0, 4]
```

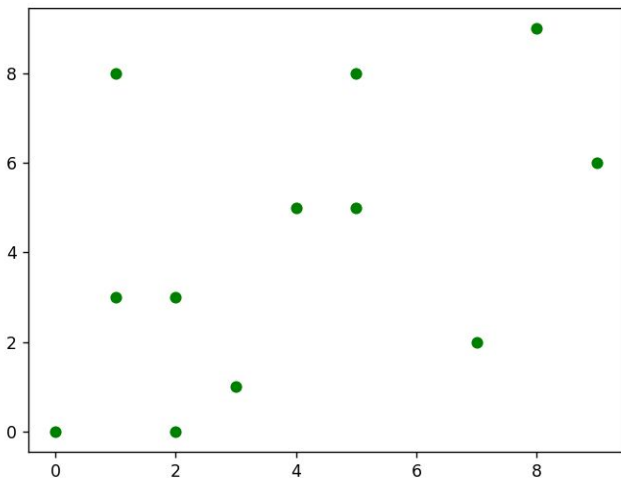
```
ys = [2, 3, 6, 0, 5, 8, 1, 8, 9, 3, 0, 5]
```

```
for x, y in zip(xs, ys):
```

```
    plt.scatter(x, y, color="green")
```

```
    # plt.plot(x, y, "o", color="green")
```

```
plt.show()
```



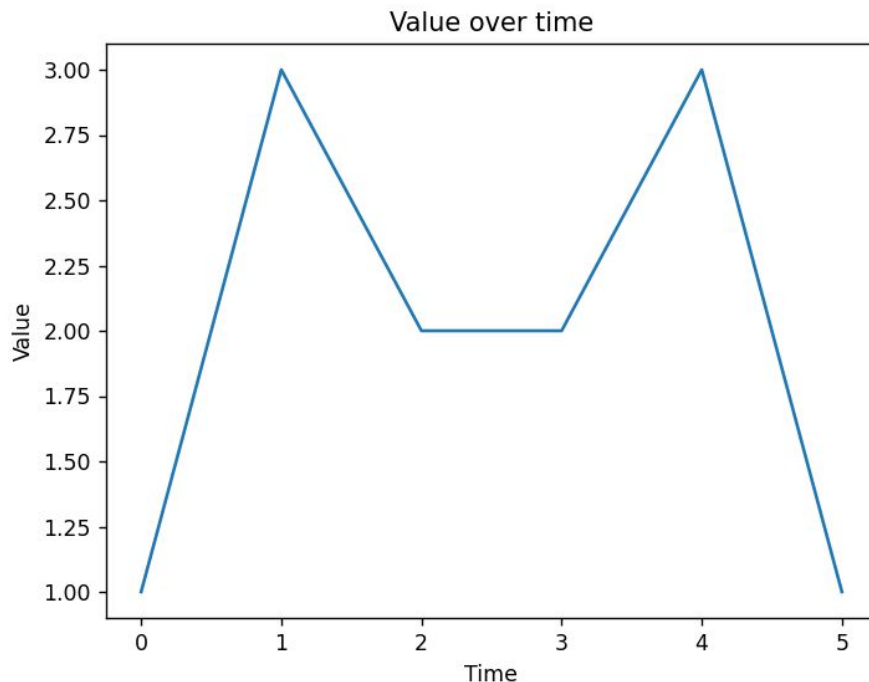
Metoder som `.title()` `.xlabel()` `.ylabel()` og `.legend()` lar oss merke viktig informasjon.

Dette gjøres før `.show()`

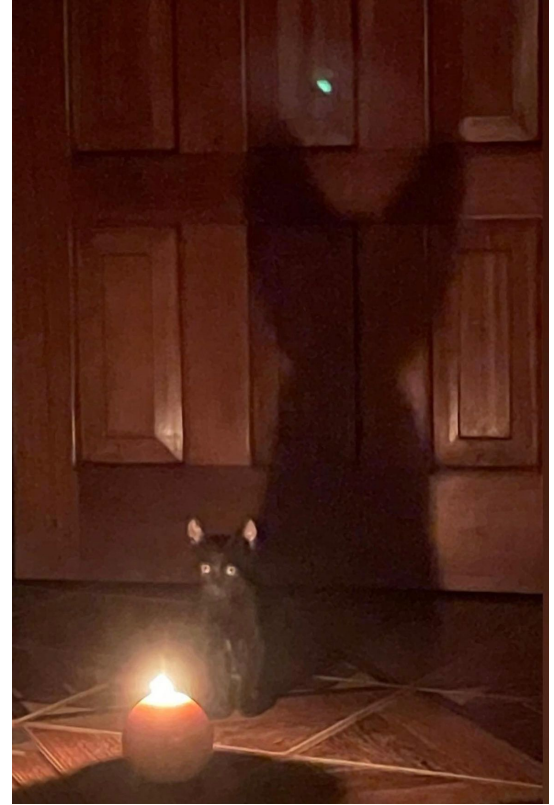
```
plt.title("Value over time")
```

```
plt.xlabel("Time")
```

```
plt.ylabel("Value")
```



# 12 - Problemløsning



# Bruk gode ressurser

Finn og bruk gode ressurser på eksamen. Dette kan være dokumentasjon eller forklaringer på enkelte ting du sliter med.

En god referanseside:

[https://www.w3schools.com/python/python\\_reference.asp](https://www.w3schools.com/python/python_reference.asp)

Her finner du lister over metoder, innebygde funksjoner, osv.

Husk at plagiat telles som fusk!

# repr()

Denne funksjonen kan vi bruke for å se akkurat hvordan en bestemt streng ser ut. Hvis du er i tvil om hvordan en streng ser ut, bruk den!

Om vi printer en streng uten repr() vil all formatering aktiveres

```
streng = "Hello \nWorld "
```

```
print(streng)
```



Hello  
World

Og med bruk av repr():

```
print(repr(streng))
```



'Hello \nWorld '



# Kommenter koden din!

Dette kan gi deg noen poeng som du ellers ikke ville fått.  
Sensor kan enklere forstå hvordan du har tenkt, og det  
hjelper ofte med å forstå hvilke datatyper man jobber med  
til enhver til.

Gjennomgang av eksamensoppgaver

Lykke til på eksamen!