

# OBJEKTORIENTERT PROGRAMMERING MED PYTHON



# JOAKIM LIER, CISCO SYSTEMS NORWAY



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer
- Veldig god til å si ja



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer
- Veldig god til å si ja
- Studerte Informatikk ved UiO



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer
- Veldig god til å si ja
- Studerte Informatikk ved UiO
- Og så datateknologi ved NTNU



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer
- Veldig god til å si ja
- Studerte Informatikk ved UiO
- Og så datateknologi ved NTNU
- Drev mye med high performance computing



# JOAKIM LIER, CISCO SYSTEMS NORWAY

- Jobber som blant annet software engineer
- Veldig god til å si ja
- Studerte Informatikk ved UiO
- Og så datateknologi ved NTNU
- Drev mye med high performance computing
- og programmeringsspråk



❤️ Programmeringsspråk ❤️



# PLAN FOR DAGEN!

*Zero to hero*

Vi skal jobbe oss opp fra det grunnleggende og opp til å kanskje løse noen ekte problemer!



# PYTHON

Vi skal bruke python som språk for å lære om  
Objektorientert programmering (OOP).

Vi skal ikke lære python, så jeg antar at dere kan det

*Hvis det er noe dere ikke forstår, så bare  
spør!*

Detaljer som kun gjelder python er markert med 



# QUIZ!

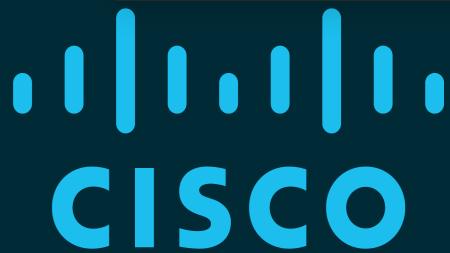
```
a = 2  
b = 4  
c = a + b  
print(c)
```



# QUIZ!

```
liste = [1]
liste.append(12)
liste.append('b')

for element in liste:
    if element == 1:
        continue
    else:
        break
```



# QUIZ!

```
def business(important, stuff):  
    return {important: stuff}  
  
print(business("jobb", "cisco"))
```



# QUIZ!

```
a = 3  
assert a == 2, f"A burde vært 2, men er {a}"
```



# OBJEKTORIENTERT PROGRAMMERING

OOP er et paradigme innen programmering.



*Paradigme blir brukt som betegnelse på  
særlige regler innenfor vitenskapelige  
disipliner*

wikipedia



*Et mønster å løse problemer på*

*Meg*



# OBJEKTORIENTERT PROGRAMMERING

Mange språk lar deg skrive objektorientert kode

Mange språk lar deg *ikke* skrive objektorientert kode

Mange språk lar deg kombinere paradigmer





Python er en god blanding av mye



I noen av oppgavene og kodesnuttene vi skal se på er  
OOP antageligvis ikke den beste løsningen

Men vi gjør det for å lære



# ENCAPSULATION, INHERITANCE, POLYMORPHISM

Objektorientert programmering kan beskrives med 3 egenskaper.



# ENCAPSULATION, INHERITANCE, POLYMORPHISM

Objektorientert programmering kan beskrives med 3 egenskaper.

- Encapsulation (Innkapsulering)



# ENCAPSULATION, INHERITANCE, POLYMORPHISM

Objektorientert programmering kan beskrives med 3 egenskaper.

- Encapsulation (Innkapsulering)
- Inheritance (Arv)



# ENCAPSULATION, INHERITANCE, POLYMORPHISM

Objektorientert programmering kan beskrives med 3 egenskaper.

- Encapsulation (Innkapsulering)
- Inheritance (Arv)
- Polymorphism (polymorfisme)



# ENCAPSULATION

Litt motivasjon for å se problemet



```
person1_fornavn = "Joakim"
person1_etternavn = "Lier"
person1_alder = 28
person1_jobb = "Cisco"
person2_fornavn = "Elev"
person2_etternavn = "Studentesen"
person2_alder = 17
person2_skole = "Akademiet"
personliste = [person1_fornavn,
                person1_etternavn,
                person1_alder,
                person1_jobb,
                person2_fornavn,
                person2_etternavn,
                person2_alder,
                person2_skole]
```





```
1 person1 = ["Joakim", "Lier", 28, "Cisco"]
2 person2 = ["Elev", "Studentesen", 17, "Akademiet"]
3
4 personliste = [person1, person2]
```



```
1 person1 = ["Joakim", "Lier", 28, "Cisco"]
2 person2 = ["Elev", "Studentesen", 17, "Akademiet"]
3
4 personliste = [person1, person2]
5
6 # hva er dette?
7 print(personliste[1][2])
```



```
1 person1 = ["Joakim", "Lier", 28, "Cisco"]
2 person2 = ["Elev", "Studentesen", 17, "Akademiet"]
3
4 personliste = [person1, person2]
5
6 def get_job(person):
7     return person[3]
8
9 print(get_job(personliste[0]))
```



```
1 person1 = ["Joakim", "Lier", 28, "Cisco"]
2 person2 = ["Elev", "Studentesen", 17, "Akademiet"]
3
4 personliste = [person1, person2]
5
6 def get_job(person):
7     return person[3]
8
9 print(get_job(personliste[1])) # Hvorfor er dette feil?
```



```
1 person1 = ["Joakim", "Lier", 28, "Cisco"]
2 person2 = ["Elev", "Studentesen", 17, "Akademiet"]
3
4 personliste = [person1, person2]
5
6 def get_job(person):
7     return person[3]
8
9 print(get_job(personliste[1])) # Hvorfor er dette feil?
```



# ENCAPSULATION - KLASSE

En klasse er en definisjon, eller "oppskrift", på en ting.



Hittil har vi definert en person som "noe" med:

- et fornavn
- et etternavn
- en alder
- en jobb eller skole



Dette har vi prøvd å få frem ved bruk av lister  
Men klasser er laget for å løse akkurat dette



# ENCAPSULATION - KLASSE

En klasse inneholder informasjonen som hører til en ting

- Egen data
- Egne funksjoner



# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

• Tydelig hva som definerer en person

• Tydelig hva som menes med koden



# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

• Tydelig hva som definerer en person

• Tydelig hva som menes med koden



# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

• Tydelig hva som definerer en person

• Tydelig hva som menes med koden

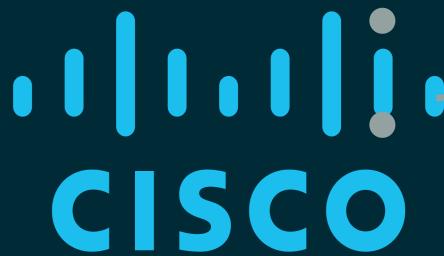


# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

• Tydelig hva som definerer en person

• Tydelig hva som menes med koden

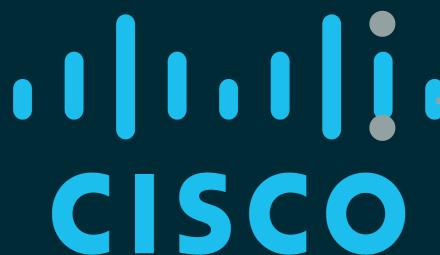


# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

Tydelig hva som definerer en person

Tydelig hva som menes med koden

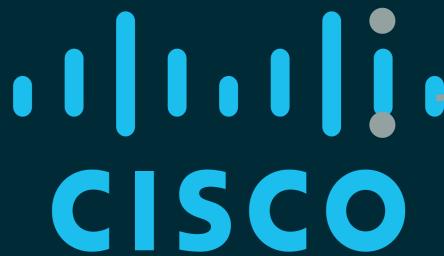


# ENCAPSULATION - KLASSE

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 person1 = Person(  
10    fornavn = "Joakim",  
11    etternavn = "Lier",  
12    alder = 28,  
13    jobb = "Cisco")  
14 print(person1.fornavn)
```

• Tydelig hva som definerer en person

• Tydelig hva som menes med koden



# Vi bruker punktum for å få tak i ting inni en klasse

```
1 class Person:  
2     def __init__(self, fornavn, etternavn, alder, jobb):  
3         self.fornavn = fornavn  
4         self.etternavn = etternavn  
5         self.alder = alder  
6         self.jobb = jobb  
7  
8 person1 = Person("Joakim", "Lier", 28, "Cisco")  
9 print(person1.fornavn)  
10 print(person1.etternavn)  
11 print(person1.alder)  
12 print(person1.jobb)
```



# ENCAPSULATION - KLASSE

```
(fornavn: str, etternavn: str, alder: int, jobb: str) -> None  
person1 = Person(
```



# KLASSER ELLER OBJEKTER?

Klasser er definisjonen på noe vi kan lage

Objekter er noe laget ut fra definisjonen



*Litt som forholdet mellom en oppskrift på  
kjeks og kjeksen du lagde fra oppskriften!*



Vi sier at et objekt er en instans av en klasse



# KLASSER ELLER OBJEKTER?

```
1 class Person: # Klasse
2     def __init__(self, fornavn, etternavn, alder, jobb):
3         self.fornavn = fornavn
4         self.etternavn = etternavn
5         self.alder = alder
6         self.jobb = jobb
7
8 person1 = Person("Joakim", "Lier", 28, "Cisco") # Objekt
9 person2 = Person("Elev", "Studentesen",
10    17, "Akademiet") # Objekt
```



# KLASSER ELLER OBJEKTER?

```
1 class Person: # Klasse
2     def __init__(self, fornavn, etternavn, alder, jobb):
3         self.fornavn = fornavn
4         self.etternavn = etternavn
5         self.alder = alder
6         self.jobb = jobb
7
8 person1 = Person("Joakim", "Lier", 28, "Cisco") # Objekt
9 person2 = Person("Elev", "Studentesen",
10    17, "Akademiet") # Objekt
```



# KLASSER ELLER OBJEKTER?

```
1 class Person: # Klasse
2     def __init__(self, fornavn, etternavn, alder, jobb):
3         self.fornavn = fornavn
4         self.etternavn = etternavn
5         self.alder = alder
6         self.jobb = jobb
7
8 person1 = Person("Joakim", "Lier", 28, "Cisco") # Objekt
9 person2 = Person("Elev", "Studentesen",
10    17, "Akademiet") # Objekt
```



# KLASSER ELLER OBJEKTER?

```
1 class Person: # Klasse
2     def __init__(self, fornavn, etternavn, alder, jobb):
3         self.fornavn = fornavn
4         self.etternavn = etternavn
5         self.alder = alder
6         self.jobb = jobb
7
8 person1 = Person("Joakim", "Lier", 28, "Cisco") # Objekt
9 person2 = Person("Elev", "Studentesen",
10    17, "Akademiet") # Objekt
```



# KLASSER ELLER OBJEKTER?

Dette betyr at man kan lage mange objekter fra en klasse

*Akkurat som du kan lage mange kjeks fra  
en oppskrift*



Samtidig trenger man bare en klasse for hver ting du vil beskrive

*Du trenger jo ikke flere oppskrifter på samme kjeksen*



# FUNKSJONER

Hittil har vi for det meste sett på å pakke inn variabler



Vi kan også lage egne funksjoner for klassene våre

Vi har sett ett eksempel på det så langt:

```
class Person:  
    def __init__(self, ...
```





Funksjoner som er omringet av \_\_ kommer vi tilbake til.



# FUNKSJONER

```
1 class Person:  
2     def er_like_gammel_som(self, andre):  
3         return self.alder == andre.alder  
4  
5 person1 = Person("Joakim", "Lier", 28, "Cisco")  
6 person2 = Person("Elev", "Studentesen",  
7                  "Akademiet")  
8 print(person1.er_like_gammel_som(person2)) # Bra!  
9 print(er_like_gammel_som(person2))          # gir ikke mening  
10                                         # trenger objekt
```



# FUNKSJONER

```
1 class Person:  
2     def er_like_gammel_som(self, andre):  
3         return self.alder == andre.alder  
4  
5 person1 = Person("Joakim", "Lier", 28, "Cisco")  
6 person2 = Person("Elev", "Studentesen",  
7                  17, "Akademiet")  
8 print(person1.er_like_gammel_som(person2)) # Bra!  
9 print(er_like_gammel_som(person2))          # gir ikke mening  
10                                         # trenger objekt
```



# FUNKSJONER

```
1 class Person:  
2     def er_like_gammel_som(self, andre):  
3         return self.alder == andre.alder  
4  
5 person1 = Person("Joakim", "Lier", 28, "Cisco")  
6 person2 = Person("Elev", "Studentesen",  
7                  17, "Akademiet")  
8 print(person1.er_like_gammel_som(person2)) # Bra!  
9 print(er_like_gammel_som(person2))          # gir ikke mening  
10                                         # trenger objekt
```



# FUNKSJONER - SELF



I Python må funksjoner som tilhører en klasse må ta inn parameteren `self`.

```
1 class Person:  
2     # Legg merke til self parameteren  
3     def er_over_20(self):  
4         return self.alder > 20  
5  
6 person1 = Person("Joakim", "Lier", 28, "Cisco")  
7 # legg merke til at det er 0 parametre under  
8 print(person1.er_over_20())
```



Denne blir automagisk sendt inn når du kaller på  
funksjonen

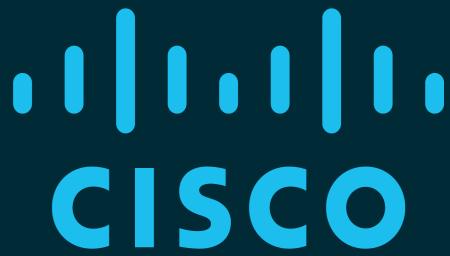
Gir funksjonen en måte å kunne lese sine egne  
variabler



*automagisk = magisk og automatisk*



```
1 class Person:  
2     def er_over_20(self):  
3         # feil!  
4         # ikke self, da vet ikke python  
5         # hvilken alder du snakker om  
6     return alder > 20
```



```
1 class Person:  
2     def er_over_20(self):  
3         # riktig!  
4         # her bruker vi self, da vet  
5         # python hvor den skal lete  
6         return self.alder > 20
```



# FUNKSJONER, MEDLEMMER ELLER METODER?

Vi sier at en funksjon eller variabel som tilhører en klasse er et *medlem*

Noen ganger brukes ordet *metode* istedet for medlemsfunksjon



# — FUNKSJONER —



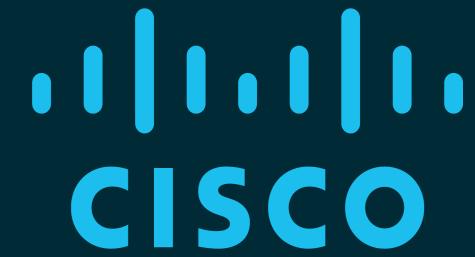
De kalles "magiske metoder"

Er ikke ment til å brukes direkte



*Magic methods python, dunder  
python*

Python bruker dem automagisk i visse situasjoner





Denne kalles hver gang du lager et objekt!

```
1 class Person:  
2     def __init__(self):  
3         print("Nå ble jeg laget!")  
4  
5 person1 = Person() # Dette printer "Nå ble jeg laget!"
```

Kalles ofte en konstruktør i andre språk



`__STR__` 

Kort for "string"

Lar deg velge en tekst-beskrivelse av objektet, feks når  
du printer

returnerer en string



```
1 class Dårlig:  
2     pass  
3  
4 class Bra:  
5     def __str__(self) -> str:  
6         return "Jeg er bra!"  
7  
8 dårlig = Dårlig()  
9 bra = Bra()  
10 print(dårlig)  
11 # printer "<__main__.Dårlig object at 0x7fda01998730>"  
12 print(bra)  
13 # printer "Jeg er bra!"
```



`_add_` og `_ge_` for matematiske operasjoner  
`_call_` for å lage objekter som oppfører seg som funksjoner  
`_enter_` og `_exit_` for å kunne bruke `with`-nøkkelordet



# OPPSUMMERING

Vi bruker klasser for å samle informasjon og funksjoner  
som hører sammen



Vi oppretter objekter basert på disse klassene, så vi lett kan jobbe med dem. Nå kan vi ha en variabel som beskriver en person, istedet for flere!



# ARV

Noen ganger kan flere klasser ha mye til felles, men må likevel holdes separat

En måte å skille ut det de har til felles kalles arv



Dette betyr at hvis en klasse arver fra en annen, vil barnet få de samme medlemmene som forelderen.

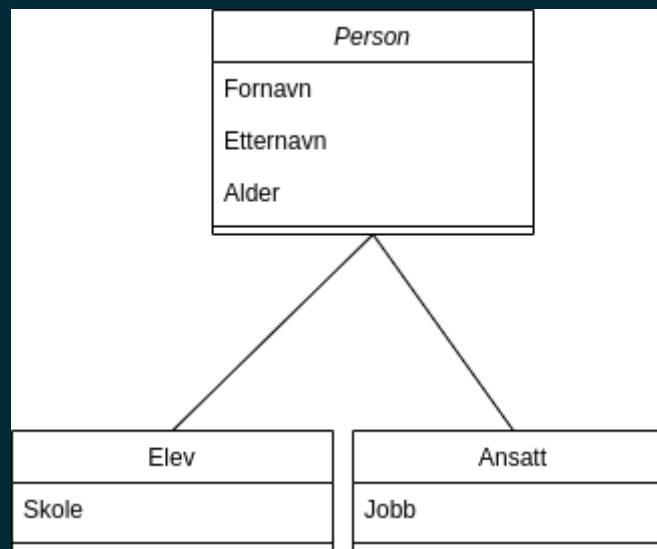


En klasse som arves fra kalles foreldreklasse,  
grunnklasse, superklasse



En klasse som arver kalles barnekasse, derivert klasse,  
subklasse





```
1 #Grunnkasse
2 class Person:
3     def __init__(self, fornavn,
4                  etternavn,
5                  alder):
6         self.fornavn = fornavn
7         self.etternavn = etternavn
8         self.alder = alder
9
10    def si_hallo(self):
11        print("hallo")
```



```
1 class Elev(Person): # Arver fra Person
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, skole):
5         super().__init__(fornavn, etternavn, alder)
6         self.skole = skole
7
8 class Ansatt(Person): # Arver fra Person
9     def __init__(self, fornavn,
10                  etternavn,
11                  alder, jobb):
12        super().__init__(fornavn, etternavn, alder)
13        self.jobb = jobb
```



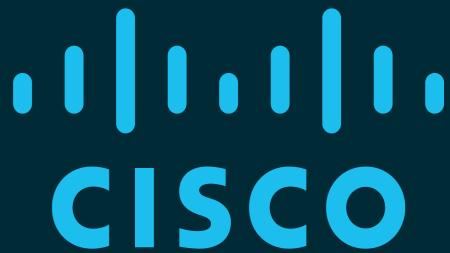
```
1 class Elev(Person): # Arver fra Person
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, skole):
5         super().__init__(fornavn, etternavn, alder)
6         self.skole = skole
7
8 class Ansatt(Person): # Arver fra Person
9     def __init__(self, fornavn,
10                  etternavn,
11                  alder, jobb):
12         super().__init__(fornavn, etternavn, alder)
13         self.jobb = jobb
```



```
1 class Elev(Person): # Arver fra Person
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, skole):
5         super().__init__(fornavn, etternavn, alder)
6         self.skole = skole
7
8 class Ansatt(Person): # Arver fra Person
9     def __init__(self, fornavn,
10                  etternavn,
11                  alder, jobb):
12        super().__init__(fornavn, etternavn, alder)
13        self.jobb = jobb
```



```
elev1 = Elev("Elev",
              "Studentesen",
              18,
              "Akademiet")
# Kan fortsatt bruke medlemmene til Person
print(elev1.fornavn)
elev1.si_hallo()
# Men har nå også et medlem som heter skole!
print(elev1.skole)
```





```
1 class Ansatt(Person):
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, jobb):
5         super().__init__(fornavn, etternavn, alder)
```

super() gir oss foreldreklassen, så vi kan kjøre  
forelderens \_\_init\_\_



```
1 class Ansatt(Person):
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, jobb):
5         super().__init__(fornavn, etternavn, alder)
```

super() gir oss foreldreklassen, så vi kan kjøre  
forelderens \_\_init\_\_





```
1 class Ansatt(Person):
2     def __init__(self, fornavn,
3                  etternavn,
4                  alder, jobb):
5         super().__init__(fornavn, etternavn, alder)
```

super() gir oss foreldreklassen, så vi kan kjøre  
forelderens \_\_init\_\_

*Hvorfor må vi bruke super(), og ikke bare  
si \_\_init\_\_( . . )?*



For å bygge intuisjon om når dette kan bli nyttig:



Det er mye som er møbler, men er alle møbler bord?

En stol er ikke et bord, men begge er møbler



Da trenger vi kanskje en møbel-klasse og en stol- og  
bord-klasse som arver fra denne

Selv om begge er laget av tre, har fire ben og selges på  
Ikea



Her på jobb tenker vi masse på møter.



Webex-, google-, microsoft teams- og zoom-møter har  
mye til felles

Alle har en start og en slutt og deltakerlister



En grunnleggende møtekasse som de forskjellige  
møtetyppene arver fra kan være lurt!

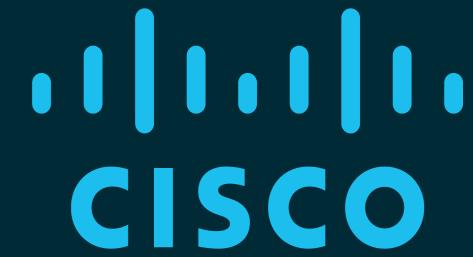


I favorittdataspillet ditt er det sikkert flere figurer som oppfører seg annerledes.



Kanskje det finnes en felles "figur"-klasse som arves fra?

Da slipper de å programmere alt alle figurer har til felles på nytt!



OBS! 

I oppgavene vil dere se

```
class Arv(Noe):  
    pass
```

Dette betyr at klassen `Arv` arver fra klassen `Noe`, men endrer ingenting så den er veldig lik `Noe`.

Dette brukes for å lage et startpunkt for dere, men er ellers ofte en nytteløs ting å gjøre :)



# POLYMORFISME

Et stort og skummelt ord



Men ganske greit konsept!



*Poly - flere*

*morphism - form*

Flere former!



Så hva betyr det i kode?



En funksjon med samme navn kan gjøre forskjellige ting



## Kodeeksempel:

```
class Dyr:  
    def snakk(self):  
        print("Jeg vet ikke hva jeg er!")  
  
class Hund(Dyr):  
    def snakk(self):  
        print("Voff")  
  
class Katt(Dyr):  
    def snakk(self):  
        print("Mjau")
```



```
dyreliste = [Hund(), Hund(), Katt(), Hund()]
for dyr in dyreliste:
    dyr.snakk()
```

```
→ ~ python3 dyr.py
Voff
Voff
Mjau
Voff
```



Hver type dyr finner sin riktige funksjon, selv om alle  
heter snakk



Hvis en klasse ikke lager sin egen snakk, så brukes forelderens snakk

```
class Axolotl(Dyr):
    pass
axolotl = Axolotl()
axolotl.snakk()
```

```
→ ~ python3 axolotl.py
Jeg vet ikke hva jeg er!
```



Polymorfisme er en arvet egenskap som barnet har gjort på sin måte.



For eksempel, så kan dere alle snakke  
Men dere snakker kanskje ikke helt likt som foreldrene  
deres



# OPPSUMMERING!



*Objektorientert programmering handler om å lage klasser*



Klasser er en oppskrift på et *objekt*.  
Beskriver variabler og funksjoner som hører til  
Disse kalles *medlemmer*



Et objekt er en instansiert klasse.

De kan for eksempel gjenspeile ekte ting som  
mennesker

Samler informasjon til en mer håndterbar ting



*Prøv å huske analogien oppskrift på kjeks  
og kjeks bakt etter oppskriften*



En klasse kan *arve* fra en annen

Da får den alle medlemmene til forelderen.



Polymorfisme er når en barnekasse endrer på en arvet funksjon

Den gjør noe liknende forelderen sin, men på sin egen måte.



Ved å bruke disse egenskapene kan vi skrive  
programmer med mennesker, biler og møbler



Uten å glemme at ulike mennesker, biler og møbler er  
forskjellige



