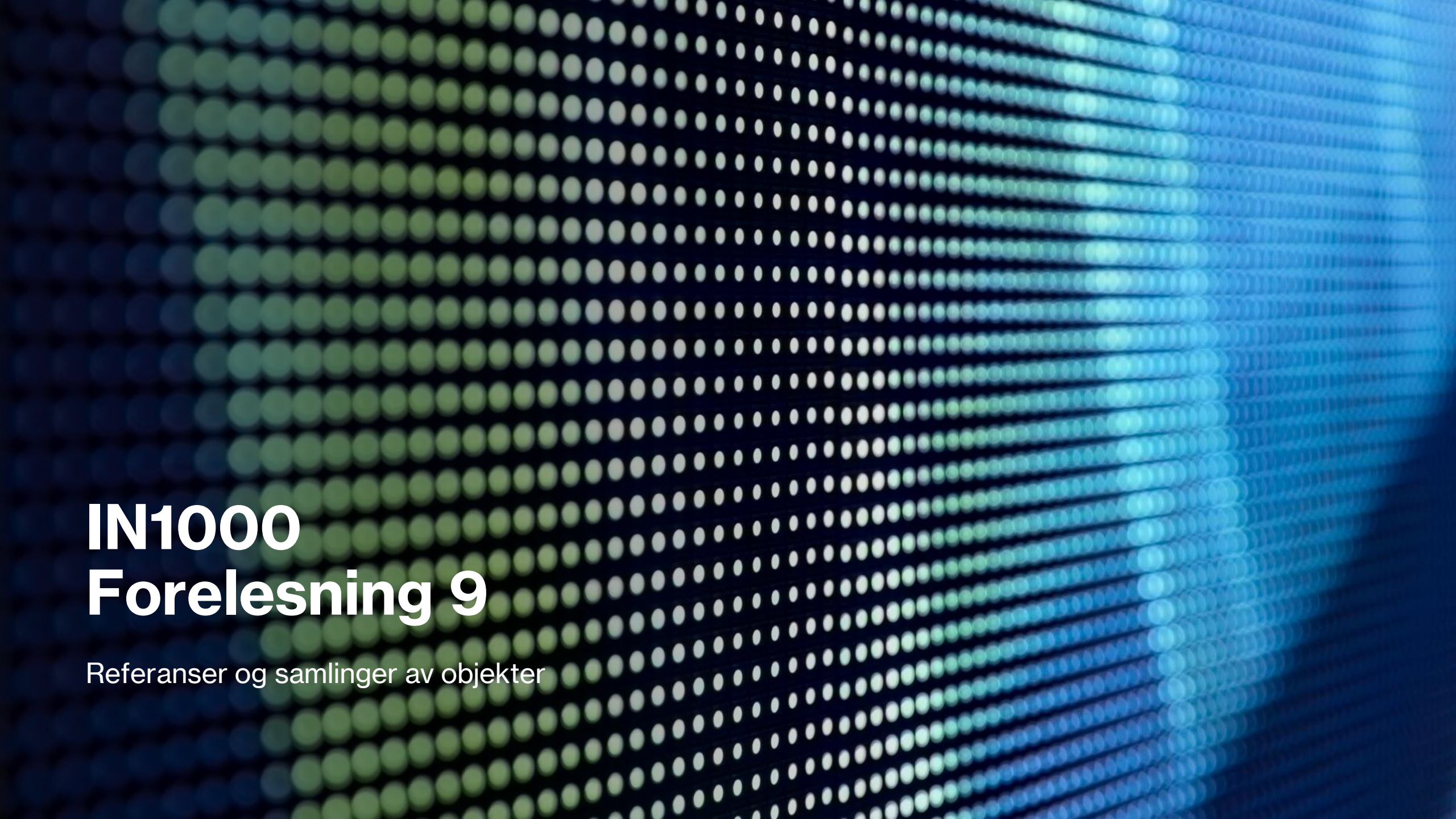


IN1000 Forelesning 9

Referanser og samlinger av objekter

Mer om parameteren *self*

- Husk at det *aktive* objektet som kaller metoden sin er separat fra objekter som sendes inn *passivt* som argumenter når en metode kalles
  - *aktivt_objekt.metode(passivt_objekt_1, passivt_objekt_2)*
  - Her er det 3 objekter, 1 aktivt og 2 passive
- Når vi definerer metoden har vi alltid en ekstra parameter *self* som gir tilgang til det aktive objektet og dets data
  - *def metode(self, parameter_1, parameter_2):*
  - Her er det 3 objekter – *self* er det aktive, de andre 2 passive
- Tilgang til det aktive objektets data ser forskjellig ut fra to ulike steder
  - Fra hovedprogrammet: *print(aktivt_objekt.instansvariabel)*
  - Fra innsiden av metoden: *print(self.instansvariabel)*

Relevante læremål

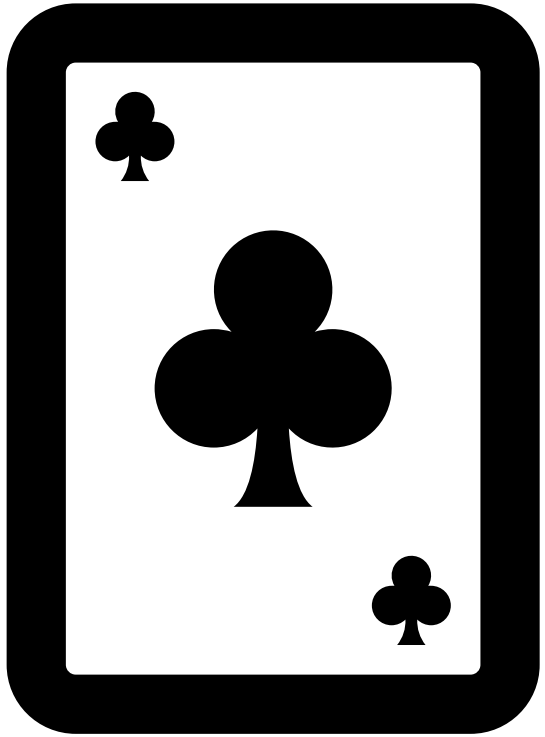
Hva lærer du?

Etter å ha tatt IN1000:

- forstår du prinsippene for objektorientert programmering og kan benytte disse til å skrive enklere objektorienterte programmer
- kan du programmere i programmeringsspråket Python og kan bruke dette til å løse mindre problemer ved hjelp av valg, løkker, funksjoner, lister, klasser og objekter
- kan du skrive oversiktlige og lesbare programmer
- er du i stand til å sette deg inn i andres programmer, finne eventuelle feil i dem og modifisere dem

Samlinger og for-løkker

- **for element in liste**
  - element = liste[0] → element = liste[1] → element = liste[2] → ...
- **for nøkkel in ordbok**
  - Ordbøker har både nøkler og verdier
  - En for-løkke som den over går kun gjennom *nøklene* (ikke verdiene, ikke begge deler)
  - Verdiene kan vi få tak i (inni løkka) med: **verdi = ordbok[nøkkel]**
- Hvordan kan vi få en for-løkke til å gå gjennom en hjemmelaget samling som **Kortstokk** fra forrige forelesning?



Livekoding: Løkker og samlinger

return og yield

- **return**
  - Kjøres bare én gang
  - Alt funksjonen skal returnere må med der
  - Avslutter funksjonen etterpå – ingen flere kodelinjer fra funksjonen kjøres
- **yield**
  - Kjøres flere ganger
  - Kan returnere én og én ting av gangen (f.eks. hver gang løkke trenger å gå videre)
  - *Pauser* funksjonen i stedet for å avslutte den
  - Funksjonen *fortsetter der den slapp* neste gang vi trenger en returverdi fra den

“Magiske” metoder

- Metoder som kjøres automatisk av Python når noe bestemt skjer
- Python leter etter bestemte metodenavn
- Hvis den finner en slik metode, kjøres den når noe bestemt skjer
- Hvis den ikke finnes...
  - ...vil det noen ganger håndteres på en eller annen standard måte: **print(objekt)**
  - ...vil det andre ganger komme en feilmelding som sier at metoden mangler: **for x in objekt**

Noen magiske metoder

Metodenavn	Kalles “automagisk” når...	Hvis metoden da mangler...
__init__	Objektet nettopp er laget	Får objektet ingen instansvariabler definert fra start av (ikke bare mangler verdiene, men variablene er heller ikke definert)
__str__	Man kaller **print(objekt)** eller **str(objekt)** for å gjøre objektet om til en **str**	Skrives typenavnet og minneadressen ut: **<kortstokk.Spillkort object at 0x000001F0C1323890>**
__iter__	Man bruker en for-løkke på objektet: **for x in objekt**	**TypeError: 'Kortstokk' object is not iterable**
__eq__	Man sjekker om to objekter er like: **objekt1 == objekt2**	Sjekker Python om variablene peker på samme minneadresse (dvs. det samme objektet): **id(objekt1) == id(objekt2)**

Noen magiske metoder trenger ekstra parametre og riktige returverdier

- **def __str__(self)**
  - Må returnere en **str**, ellers vil du se feilmeldinger av typen **TypeError: __str__ returned non-string (type NoneType)**
- **def __iter__(self)**
  - Må inneholde **yield**, ellers vil du se feilmeldinger av typen **TypeError: iter() returned non-iterator of type 'Spillkort'**
- **def __eq__(self, other)**
  - Med objekt1 == objekt2 er **self** det første, aktive, objektet (objekt1)...
  - ...og **other** det andre, passive objektet

Konstruktørens parametre styrer hvilke argumenter vi må lage objektet med

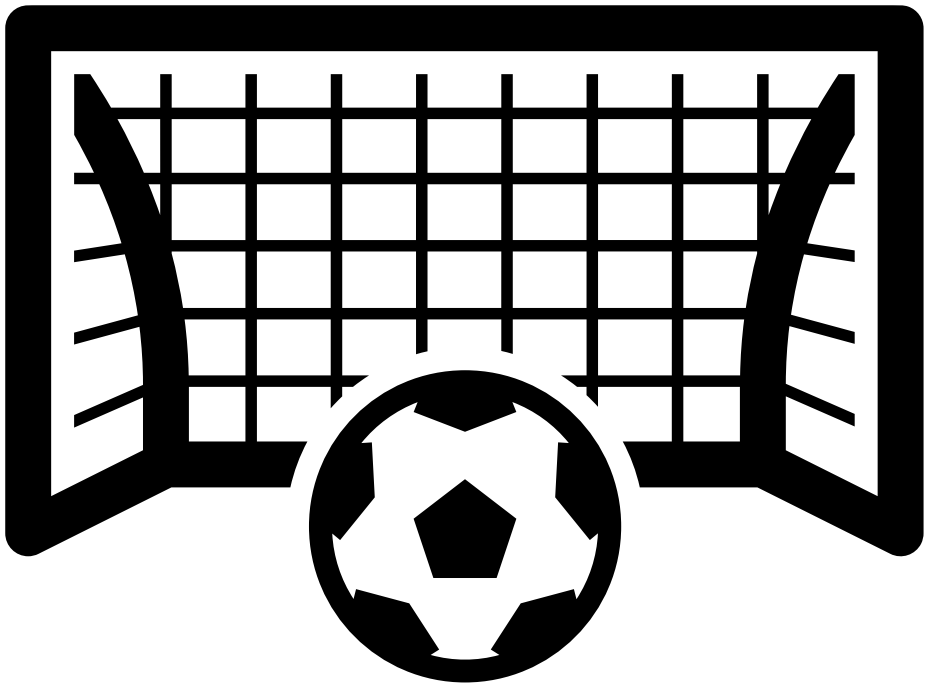
- **def __init__(self)** betyr at vi kan lage objektet med **variabel = HjemmelagetObjekt()**
- **def __init__(self, tall)** kan bety **variabel = HjemmelagetObjekt(8)**
- **def __init__(self, tall1, tall2)** kan bety **variabel = HjemmelagetObjekt(40, 80)**
- Et viktig poeng er at parameternavnet er et hint til oss mennesker
- Python sjekker ikke om objektet har en bestemt (eller riktig) type!
- **def __init__(self, tall)** kan like gjerne kalles med **variabel = HjemmelagetObjekt(“Ole”)**
- Men i konstruktøren kan vi sjekke om parameteren **tall** virkelig er et heltall
- **if type(tall) is not int:**
  - **tall = 0** # standardverdi hvis tall har feil type

is / is not sjekker om to variabler peker på samme objekt eller ikke

- **liste1 = [1, 2, 3]** # minneadresse 1980
- **liste2 = [1, 2, 3]** # minneadresse 2026
- **print(liste1 is liste2)**
  - **False**
- **print(liste1 == liste2)**
  - **True**
- **__eq__**-metoden til lister er nemlig laget slik at den returnerer **True** hvis listene inneholder like (sjekkes med ==) elementer
  - Sjekker dessuten at elementene har samme *type* og at begge listene har like *mange* elementer
- **objekt1 is objekt2** betyr det samme som **id(objekt1) == id(objekt2)**
  altså at variablene peker på samme minneadresse

is brukes vanligvis til å sammenligne med None og undersøke typen til et objekt

- Bruker vi **objekt == None**, kaller vi **__eq__**-metoden til klassen
- Dette kan kan godt returnere **True** selv om det andre objektet ikke er **None**
- **def __eq__(self, other):**
  - **return True** # ganske merkelig ting å gjøre (objektet er lik alle andre objekter), men fullt mulig
- For å være *sikre* på at et objekt virkelig er **None**-objektet, må vi bruke **objekt is None**, som ikke overstyres av en **__eq__**-metode
- **None**-objektet er unikt (vi vil aldri ha 2 forskjellige **None**-verdier)
- Samme logikk gjelder for å sammenligne typer: **type(objekt) is int** er vanligvis veien å gå for å sjekke om noe er et heltall
  - Igjen er noe som kalles arv et unntak, og for de som liker å bevege seg utenfor pensum...
  - ....vil funksjonen **isinstance(objekt, int)** fungere også for typer som *arver* fra **int**

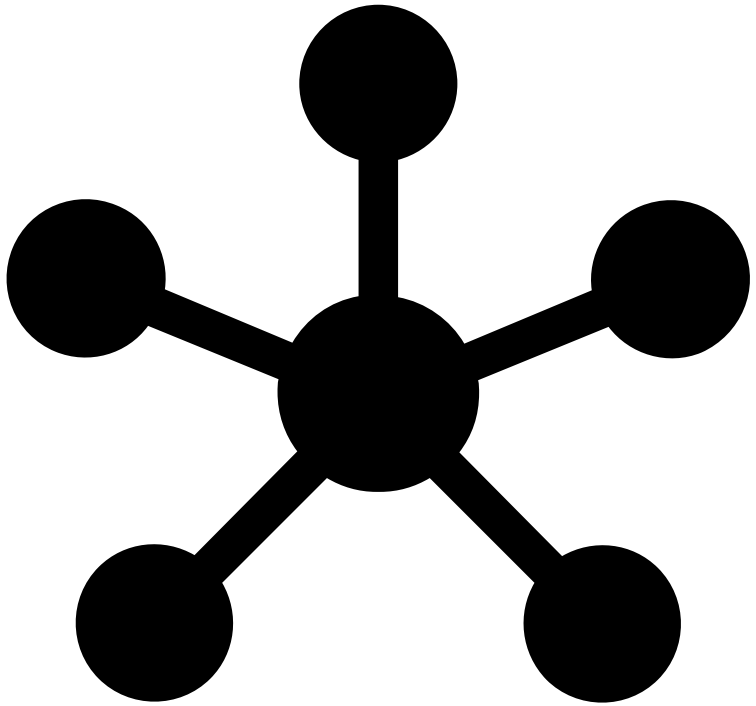


Livekoding: Tilfeldige tall

Roller til instansvariabler

- Enkel verdi: **int / float / str / bool**
- Samling med mange objekter som dette objektet “inneholder”:
  - Henter objektene direkte: **list**
  - Henter objektene med oppslag på navn, ID eller lignende: **dict**
- Referanse til et objekt som dette objektet “inneholder”:
  - Et **Sesong**-objekt kan inneholde et **Tabell**-objekt (ikke en del av oblig 7)
- Bakover-referanse til et objekt som inneholder dette objektet
  - Et **Runde**-objekt inneholder en referanse til **Sesong**-objektet som har en referanse til det
  - Altså en toveis referanse
  - Dette er fordi **Runde**-objektet trenger tilgang til hvilke **Lag** som ligger i en **Sesong**

Livekoding: Toveis koblinger mellom objekter



Referanser i oblig 7, oppgave 2

- Skal referansevariablene peke på ett enkelt objekt, en liste med objekter eller en ordbok med objekter?
- En **Sesong** skal ha referanse til mange **Runde**-objekter, og du skal kunne slå opp på rundenummer
- En **Sesong** skal ha referanse til mange **Lag**-objekter (lest fra fil), og du skal kunne slå opp på lagnavn
- En **Runde** skal ha en referanse tilbake til **Sesong**-objektet den tilhører
- En **Runde** skal ha referanse til mange **Kamp**-objekter
- En **Kamp** skal ha referanse til to **Lag**-objekter (hjemme- og bortelag)

Hvordan beskriver vi typer til samlinger av objekter?

- Liste som kan inneholde mange ukjente typer: **list**
- Liste som inneholder bare én type og aldri noe annet: **list[Kamp]**
- Liste hvor alle typene er enten **int** eller **str** og aldri noe annet: **list[int | str]**
- Ordbok hvor nøkler og verdier begge kan inneholde mange ukjente typer: **dict**
- Ordbok hvor nøkler og verdier alltid har helt bestemte typer: **dict[str, Sesong]**
  - I det siste tilfellet har *nøklene* alltid type **str** og *verdiene* alltid type **Sesong** (nøklene kommer først og verdiene sist)
- Hvis du kjenner til noe som kalles *type hinting* i Python (ikke pensum), så er det akkurat det samme formatet som vi bruker her