

# IN1000 Repetisjon

Sammendrag av hele semesteret
Hva er de viktigste tingene?

# Hva skal vi med funksjoner?

- Hensikten er å gjøre programmer mer oversiktelige og lettere å skrive og lese for mennesker
- Funksjoner et er verktøy til å redusere kompleksitet
- Vi kan *bruke* funksjoner uten å tenke på *hvordan* de virker innvendig
- *print*-funksjonen i Python er egentlig 100 linjer skrevet i C (et annet programmeringsspråk), men det har ikke vi trengt å tenke på – det viktigste for oss er å bruke den riktig
- **Hva** funksjonen gjør og **hvordan den brukes riktig** er viktig når vi bruker en funksjon
- **Hvordan** funksjonen gjør det den gjør er bare viktig når vi lager / endrer funksjonen

# Hva skal vi med objekter?

- Hensikten er å gjøre programmer mer oversiktelige og lettere å skrive og lese for mennesker (akkurat som med funksjoner)
- Også objekter et er verktøy til å redusere kompleksitet
- Vi kan *bruke* objekter uten å tenke på *hvordan* de virker innvendig
- Litt som at vi kan bruke en bil uten at vi trenger å kunne bygge eller reparere en
- **Hva** objektet gjør og **hvordan det brukes riktig** er viktig når vi bruker et objekt
- **Hvordan** objektet gjør det det gjør er bare viktig når vi lager / endrer oppskriften på hvordan denne typen objekt skal lages

# Objekter inneholder data

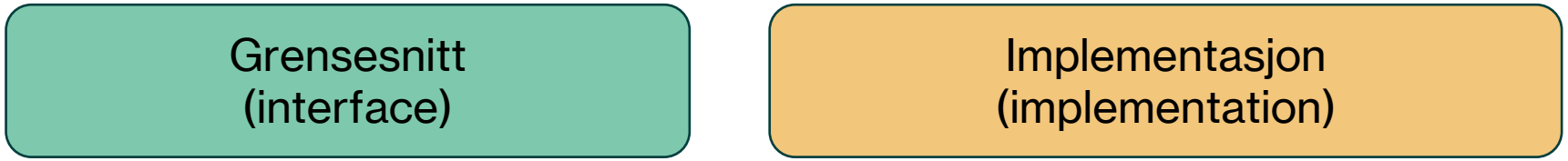
- Et objekt kan ha mange instansvariabler med data
- På denne måten kan vi samle data som hører sammen i ett objekt
- Da kan vi jobbe med dette objektet i stedet for å jobbe med datavariablene hver for seg
- I stedet for
  - en variabel med symbolet til kortet (str), for eksempel “kløver”
  - en variabel med tallverdien til kortet (int), for eksempel 4
- Så har vi ett *Spillkort*-objekt som inneholder alt dette
- Lettere å se helheten, ikke bare detaljene!

# Objekter er mer enn bare passive data – de kan også gjøre ting

- I tillegg til data kan objekter gjøre ting selv
- Ikke fordi de er intelligente, men fordi vi (eller noen) har laget en oppskrift på hvordan de skal gjøre det
- For eksempel vet en liste hvordan den skal sortere seg selv (så lenge innholdet i listen er av en type som kan sorteres)
- Å gi objektet til en **funksjon** er som å behandle objektet som en baby som ikke klarer å gjøre noe selv
  - for eksempel: *sorted(listeobjekt)* *# hei, funksjon, sortér dette & gi meg resultatet!*
  - Men da trenger vi jo en funksjon i tillegg til selve objektet…
- Å selv gi objektet ansvar for å kunne gjøre ting på egen hånd er som å behandle det som et større barn, som kan gjøre ting på egen hånd
  - for eksempel: *listeobjekt.sort()* *# hei, listeobjekt, sortér deg selv!*

# Objektorientering er en måte å *tenke* på

- For datamaskinen har det lite å si om vi bruker objekter eller ikke – maskinkoden er ikke så forskjellig
- Men for oss mennesker kan denne måten å tenke på (dataobjekter som kan gjøre ting selv) gjøre programmene enklere å skrive, lese og forstå
- Hjelper oss å få oversikt over store programmer
- Skiller mellom hva objektet gjør (sett utenfra) og hvordan det gjør det (sett innenfra)




- Alt vi har gjort i IN1000 til nå har vært å lære å bruke grensesnittet til noen typer objekter riktig

# Innkapsling

- Det viktige for å bruke klassen rett er i **grensesnittet**
  - metoder (med parametre, returverdier)
  - instansvariabler
  - dokumentasjon
- Detaljene som bare er viktig internt er **innkapslet** (“gjemt bort”) inne i objektet (**implementasjonen**), så vi kan bruke objektet uten å tenke på disse
  - men vi må inn her hvis vi skal endre på / reparere hvordan det virker

# Syntaks for objekter, oppsummert

- Hvordan finner vi typen til et objekt (når *x* er en variabel)?
  - *print(“x har typen”, type(x))*
- Hvordan lager vi et nytt objekt av typen (eller klassen) *HjemmelagetObjekt*?
  - *x = HjemmelagetObjekt()*
- Noen typer objekter må i tillegg ha **argumenter** for å lages, akkurat som noen funksjoner må ha det (det går an å tenke på *HjemmelagetHeltall* som en slags funksjon som returnerer et nytt objekt av typen *HjemmelagetHeltall*)
  - *y = HjemmelagetHeltall(100)*
- Objekter kan ha variabler (**instansvariabler**): *z = y.verdi*
- Objekter kan ha egne funksjoner (**metoder**) som lar objektet gjøre noe: *y.doble_verdi()*

# To typer objekter vi møter ofte

- Enkle **dataobjekter** som bare har instansvariabler med data, og ingen metoder
  - Eksempel: Et *Spillkort*-objekt, som bare har *.symbol* (hjerter, ruter, kløver, spar) og *.verdi* (2-9)
- En **samling** av flere objekter i en bestemt **rekkefølge**
  - Eksempel: Et *Kortstokk*-objekt, som inneholder mange *Spillkort*-objekter
  - Når vi skal trekke et kort skal vi ta det første (øverste) i bunken
- Det viktigste her er hvordan vi får disse objektene til å fungere riktig *sammen*
  - Mindre viktig, men også relevant: Hvordan vi lager en ny type objekt i Python

# En klasse → mange objekter

- En **klasse** er oppskriften vi bruker på å lage et objekt, som definerer alle metodene objekter av denne typen har
- Det vil si at **klasse** og **type** nesten betyr det samme: alle objekter av samme type er laget med samme oppskrift (klasse)
- Vi kan lage mange kaker av samme oppskrift, og mange objekter av samme klasse
- Disse objektene kan likevel være i forskjellige **tilstander** med ulike **data**: Selv om både *a* og *b* er av klassen **str**, så kan verdien til *a være “Hei”* og verdien til *b* være *“Slartibartfast spiller digeridoo”*, for eksempel
- Litt som at vi kan ha en kake som er halvspist og pyntet med nonstop og en pyntet med seigmenn som det bare er tatt en liten bit av (men begge er sjokoladekaker)

# Variabel → referanse → verdi (objekt)

En **variabel** er et menneskevennlig navn vi gir til noe i programmet vårt

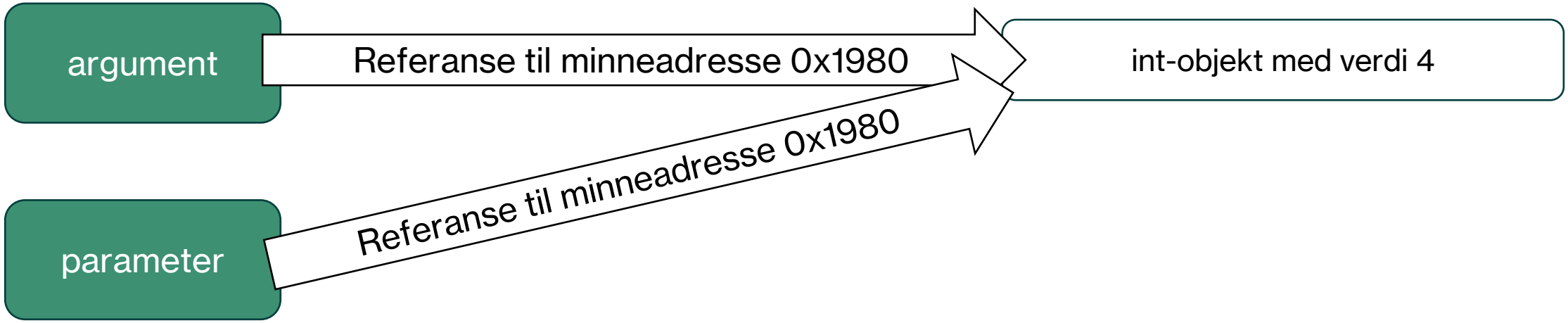
Variabelen inneholder en **referanse** (en minne-adresse) som forteller maskinen hvor den skal slå opp i minnet når vi bruker variabelen senere

På denne minne-adressen finner vi et **objekt** (som vi kaller variabelens **verdi**)

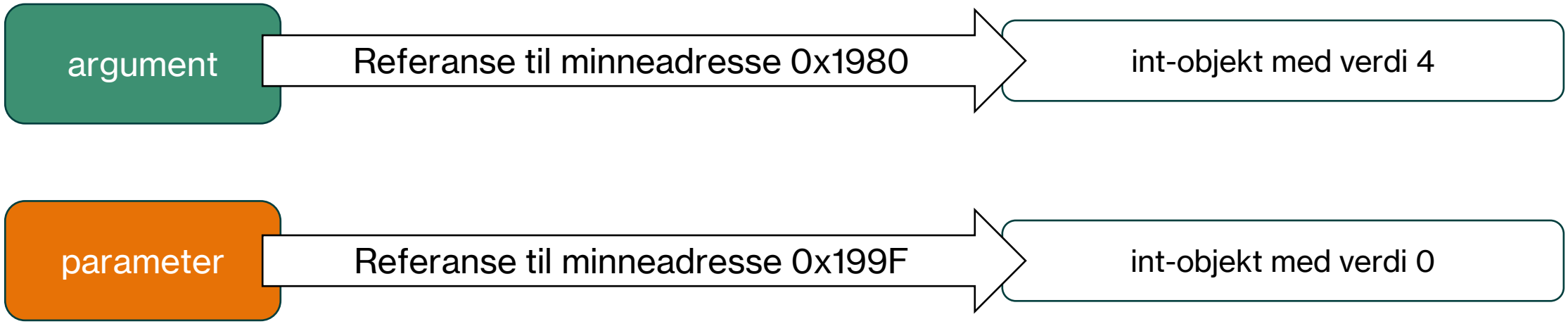
Objektet har en bestemt **type** (eller **klasse**) som bestemmer hvilke **metoder** dette objektet har (ting det selv vet hvordan det skal gjøre)

Objektet kan ha **instansvariabler**, som har referanser til *andre* objekter, osv...

# Ulike variabler kan ha lik referanse (til samme objekt)



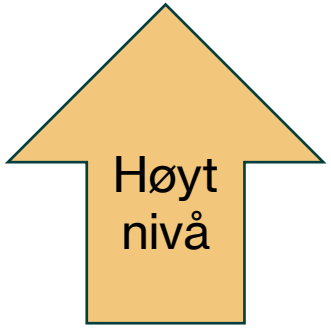
# Men endrer vi referansen til én variabel, endres ikke de andre variablene



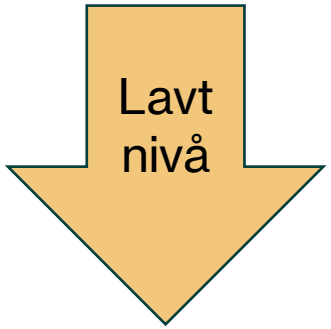
# Funksjonskall i Python

- Når vi kaller Python-funksjoner **kopieres ikke** objektene vi sender inn som argument til funksjonen – det lages ikke nye objekter
- I stedet får funksjonen en **referanse** (minneadresse) til objektene
- Hvis vi gir et objekt som kan endre tilstand (f.eks. liste) som argument og **endrer** dette inni funksjonen, så vil også variabelen i hovedprogrammet påvirkes av endringen
- Dette kalles **call by reference**
- I andre programmeringsspråk kan vi støte på noe som heter **call by value**, som vil si at funksjoner noen ganger helt automatisk lager sin egen kopi av objekter
- I slike tilfeller vil endringer på kopien ikke påvirke originalen
- (vi **kan** lage kopier av objekter i Python også, men da må vi gjøre det manuelt)

# Tre nivå å lese programkode på



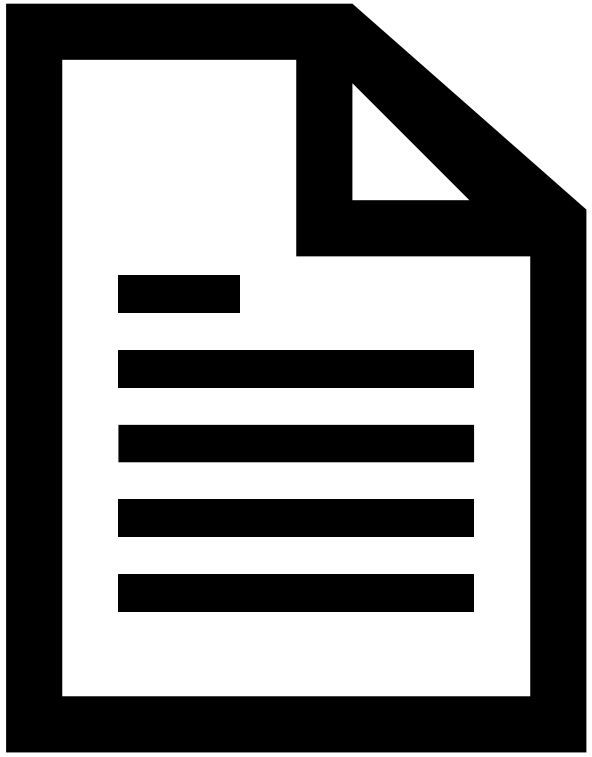

- Vi startet forelesningen på **menneskelig nivå**: Tenker på objektene som klumper med data som kan gjøre enkelte ting selv
  - Opptatt av **hva** objektene gjør (grensenitt), ikke **hvordan** de gjør det (implementasjon)
  - “Hva handler dette programmet om?” (navn på variabler / funksjoner til stor hjelp her!)
- Vi fortsatte på **syntaks-nivå**: Hvordan skriver vi Python-koden for å gjøre dette
  - Forrige forelesning: Skille mellom funksjon, metode og instansvariabel
  - Denne forelesningen: Hvordan skrive en klasse (implementasjon)
- Og avsluttet på **maskin-nivå**: Hvordan er tilstanden til programmer og objekter inne i maskinen akkurat nå (implementasjon)
  - Og hvordan endrer tilstanden seg når en ny kodelinje kjøres
  - Python Tutor er et godt hjelpemiddel for å øve opp dette



# Den virkelig store misforståelsen i IN1000

- “Du kommer langt på eksamen med å kunne skrive riktig syntaks for å lage en klasse”
- Nei
- Det har bare vært 5% av disse to forelesningene, og ca. 5% av poengene på eksamen
- **Å skrive en enkel klasse er svært enkelt, og belønningen er tilsvarende enkel**
- “Hvis syntaks-nivået alene er så lite viktig, hva er viktig da?”
- Å kunne **bruke** forskjellige objekter som lages **sammen** og utnytte det hvert objekt kan gjøre **selv** (tjenestene det tilbyr) for å løse problemer effektivt (menneskelig nivå)
- Å kunne forstå hvordan objektene skal **kobles sammen** og hvilke **datastrukturer** (lister, ordbøker etc.) som er fornuftige å bruke internt i en klasse (maskin-nivå)
- Begge disse tingene er fokus videre for de neste forelesningene

# Livekoding: begreper.py



# 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)*

# 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?

# 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**

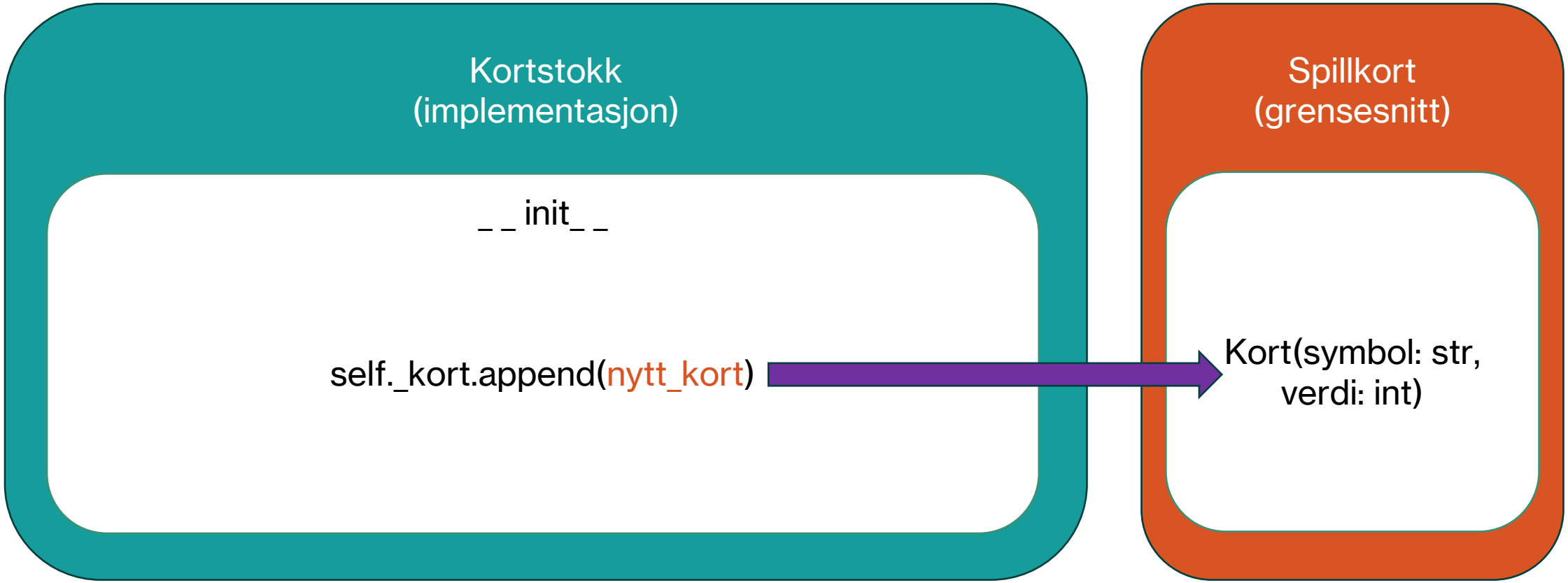
# Noen magiske metoder

| 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)** |

# 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**

# Se en klasse innenfra og en annen utenfra – samtidig



# Oblig 8 → Prøveeksamen

- Oppgave 4 blir endret til prøve-eksamen, for å bli mer lik avsluttende eksamen
- I stedet for at implementasjonen (private instansvariabler / metoder) forklares for deg, må du (i noen deloppgaver) selv finne ut hva du trenger
- Akkurat som i oblig 7: Trenger jeg en eller flere referanser? Trenger jeg en liste eller ordbok?
- Vær forberedt på at til noen klasser vil du bare få et grensesnitt – det kan være flere måter å implementere det på, og det er opp til deg å finne en som funker bra...
- ...akkurat som i en utvikler-jobb