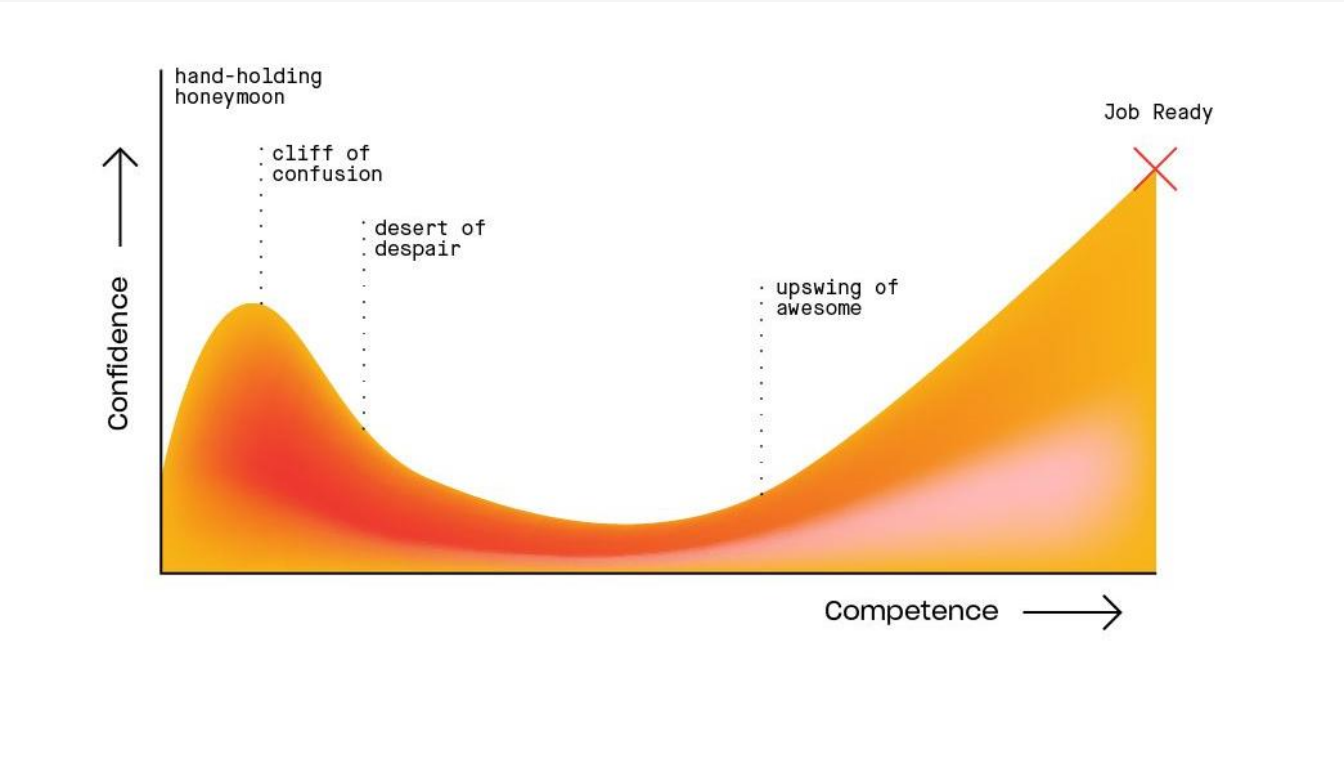
Thinking like a programmer

One of the hardest things to learn in programming is not the syntax you need to learn, but how to apply it to solve real-world problems. You need to start thinking like a programmer — this generally involves looking at descriptions of what your program needs to do, working out what code features are needed to achieve those things, and how to make them work together.

oblig 8

This requires a mixture of hard work, experience with the programming syntax, and practice — plus a bit of creativity. The more you code, the better you'll get at it. We can't promise that you'll develop "programmer brain" in five minutes, but we will give you plenty of opportunities to practice thinking like a programmer throughout the course.



https://www.thinkful.com/blog/why-learning-to-code-is-so-damn-hard/

hvorfor?

- vi lærer mer, men blir samtidig mer klar over alt vi *ikke* kan enda
- når utfordringene øker i takt med at vi blir flinkere, kan det føles som om vi står stille
- og når ting blir mer komplekse (som i oblig 8), kan det brått føles som om vi blir *mindre* flinke

hva er løsningen?

- tid og tålmodighet
- realistiske forventninger
- øvelse, øvelse, øvelse
- fokusere på *forståelse*
- redusere press på seg selv
- lære strategier for å navigere kompleksitet
- spørre om hjelp
- **se tilbake, legge merke til hvor mye vi har lært!**

in1000
uke 13

klassedesdign, abstraksjon og feilsøking

læremål

0

abstraksjon

helhet og detaljer

1

klassedesign

hvordan få delene til å virke sammen

2

feilsøking

hvilke verktøy har vi?

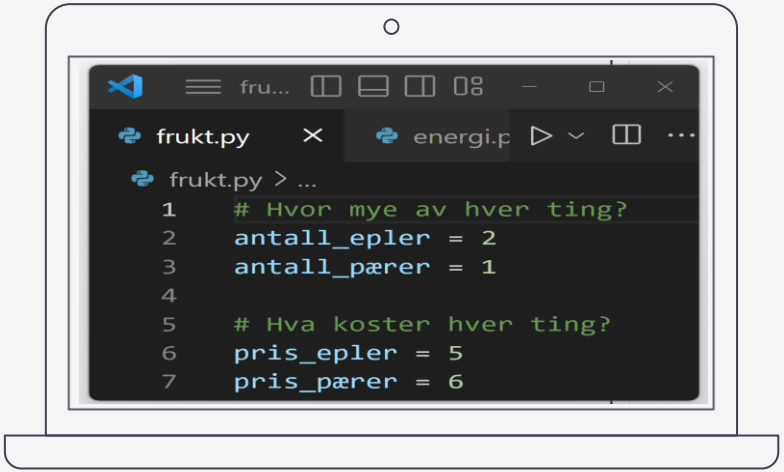
0

abstraksjon

å kunne se skogen for bare trær

"**Managing complexity** is the most important technical topic in software development."

Steve McConnell, *Code Complete*



live-koding

få oversikt ved å skjule detaljer og fokusere på grensesnitt

tabell.py + statistikk.py
(oblig 8)

Statistikk

Metodenavn	**Parametre (utenom *self*)**	**Retur-verdi**	**Beskrivelse**
`__init__`	`lagliste: list[Lag]`	-	Konstruktør. Alle lagene får 0 av alle typer plasseringer.
`lagre_rangering`	`rangering: list[Lag]`	-	Lagrer rangeringen etter at én sesong er spilt
`print_plasseringer`	-	-	Skriver ut antall plasseringer for hvert lag (etter alle sesonger spilt)

Tabell

Metodenavn	**Parametre (-*self*)**	**Retur-verdi**	**Beskrivelse**
`__init__`	`lagliste: list[Lag]`	-	Konstruktør. Alt i tabellen blir satt til 0.
`legg_til_resultat`	`kamp: Kamp`	-	Legger til resultatet av en spilt kamp i tabellen.
`oppdater_rangering`	-	-	Sorterer lagene etter poeng, målforskjell og evt. navn.
`hent_rangering`	-	`list [Lag]`	Returnerer rekkefølgen til lagene i tabellen.
`print_tabell`	-	-	Skriver ut tabellen til terminalen.

vi trenger ikke tenke på **alle** detaljene samtidig, bare de som er viktige på nivået vi er på akkurat nå

1

klassedesign

hvordan få delene til å fungere godt sammen

innkapsling: hver del får ansvar for hver sine ting

- **hovedprogram (+ statistikk):** regne sannsynlighet over mange sesonger
- **sesong (+ tabell):** oversikt over alle kamper og resultater i én sesong
- **kamp:** hvordan spilles en enkelt kamp (hvordan lages resultatet)
- **lag:** informasjon om lagene

hva trenger hovedprogrammet?

- tilgang til Sesong-objektene for å lage og spille sesonger, og til å hente ut plasseringene
- tilgang til Lag-objektene for å vite hvilke lag det skal lages statistikk over (lagliste)
- Kamp-objektene håndteres av Sesong, og vi trenger ikke tenke på dem på dette nivået – mindre komplisert ☺

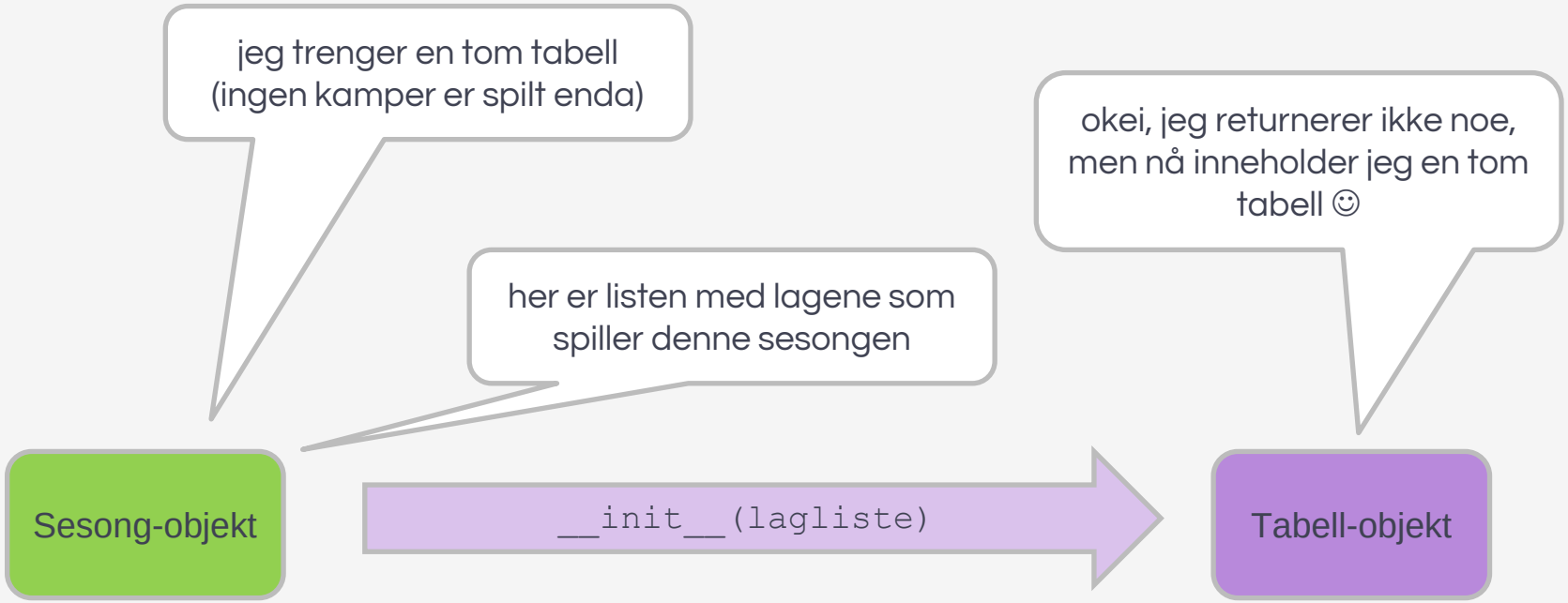
hva trenger Sesong-objektene?

- tilgang til Kamp-objektene for å lage og spille kampene i hver runde og hente ut resultater
- tilgang til Lag-objektene for å vite hvilke lag som skal inn i tabellen
- vi må sørge for at **grensesnittene** lar klassene motta (parametre) og sende fra seg (returverdier) alt de andre klassene trenger

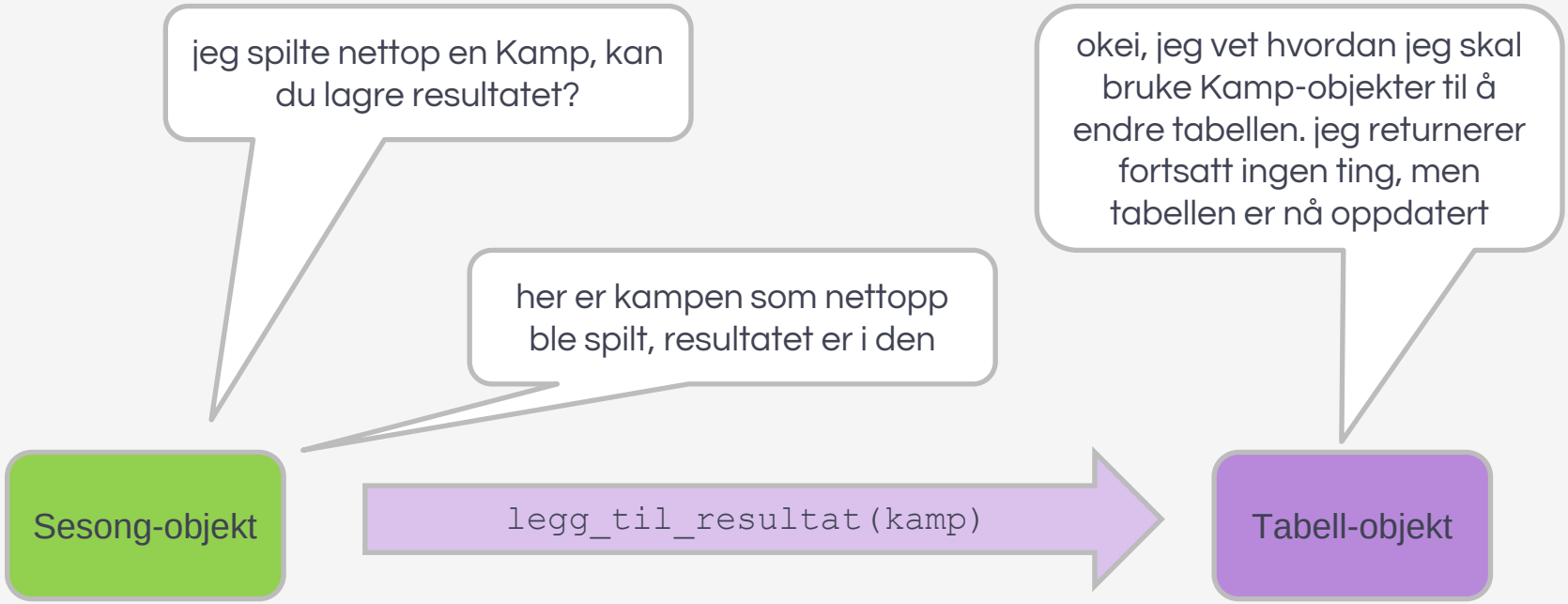
eksempel: Tabell-klassens grensesnitt

Metodenavn	**Parametre (-*self*)**	**Retur-verdi**	**Beskrivelse**
`__init__`	`lagliste: list[Lag]`	-	Konstruktør. Alt i tabellen blir satt til 0.
`legg_til_resultat`	`kamp: Kamp`	-	Legger til resultatet av en spilt kamp i tabellen.
`oppdater_rangering`	-	-	Sorterer lagene etter poeng, målforskjell og evt. navn.
`hent_rangering`	-	`list [Lag]`	Returnerer rekkefølgen til lagene i tabellen.
`print_tabell`	-	-	Skriver ut tabellen til terminalen.

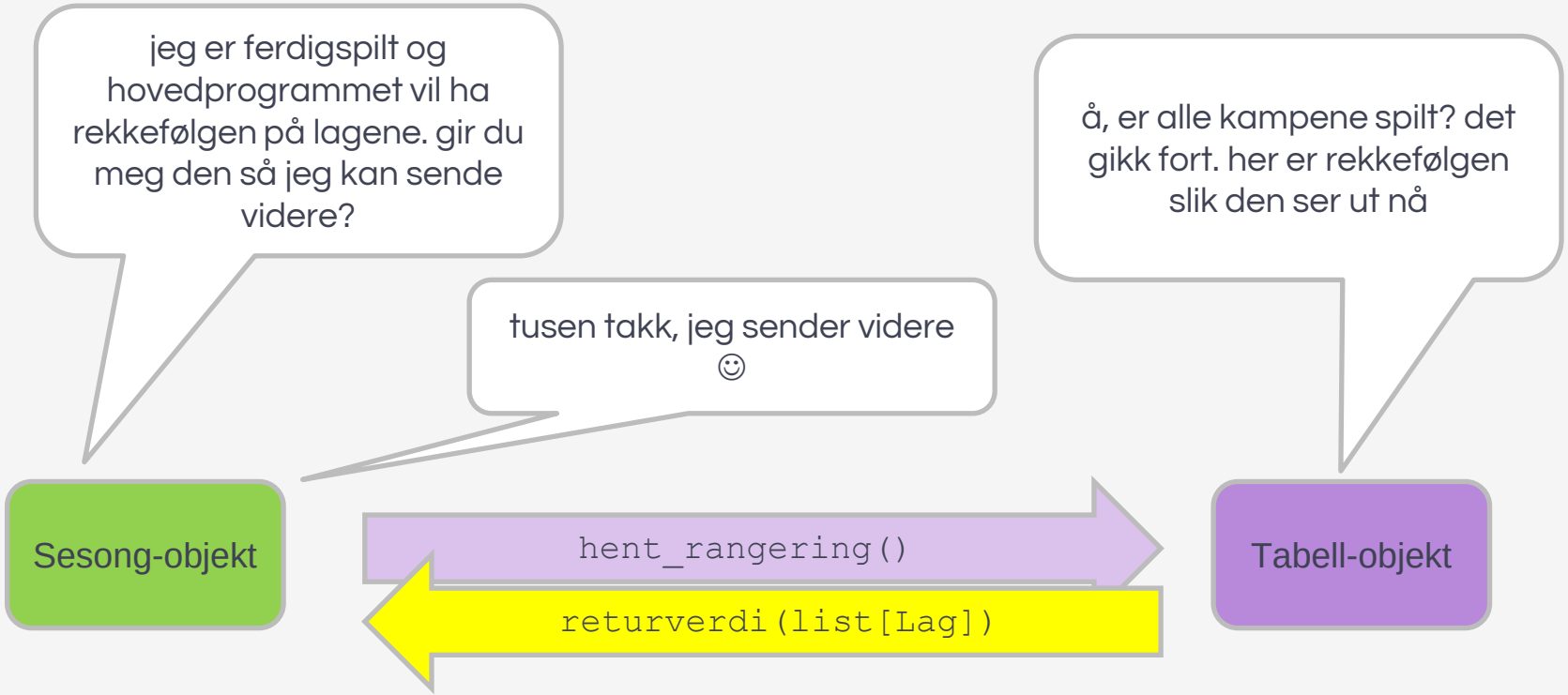
bruk av grensesnittet fra Sesong



bruk av grensesnittet fra Sesong

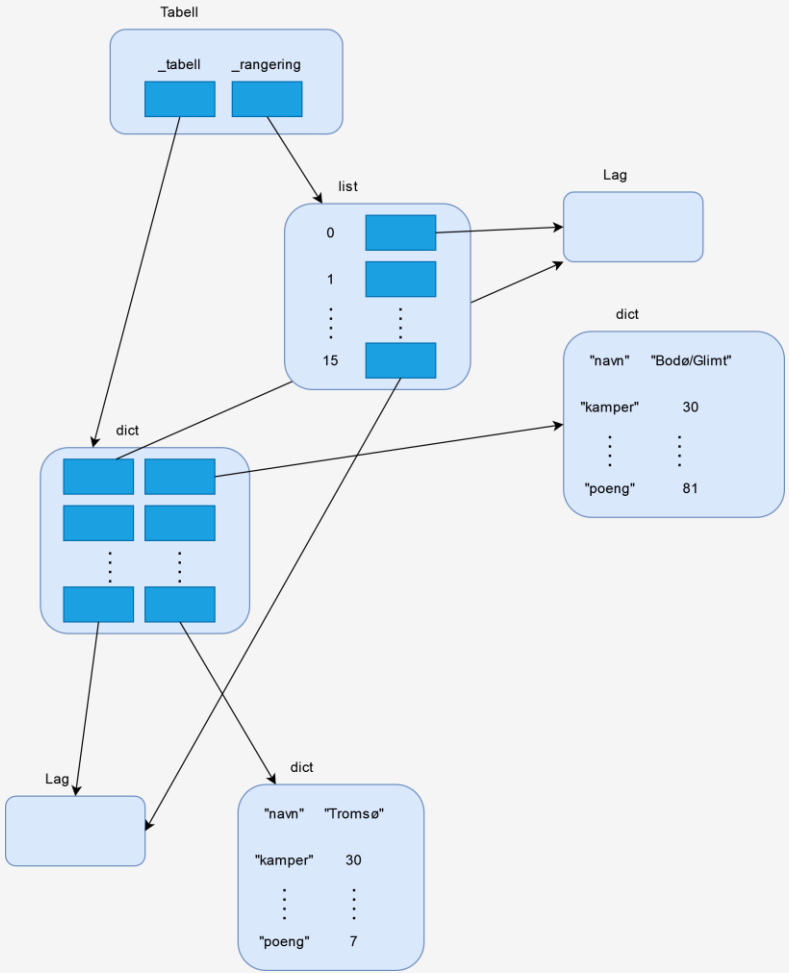


bruk av grensesnittet fra Sesong



med dette grensesnittet trenger ikke Sesong-objektet å vite (eller tenke på) hvordan Tabell-objektet fungerer innvendig

(komplisert)



2

feilsøking

hva gjør vi når ting ikke fungerer?

feilsøking (debugging)

- “det kan da ikke skje!”
- “det skjer i alle fall ikke på *min* maskin”
- “det burde ikke skje”
- “hvorfor i all verden skjer det der?”
- “åh, jeg skjønner…”
- “hvordan i alle dager virket dette i utgangspunktet?”

feil er en uunngåelig del av prosessen

- syntaksfeil – gir feilmelding (mer eller mindre hjelpsom)
- logiske feil – programmet kjører uten feilmelding, men gjør ikke det vi ville det skulle gjøre

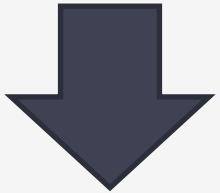
vi trenger ikke bare å finne ut **om** noe er feil, men også **hva** feilen er

hvordan finner vi feil?

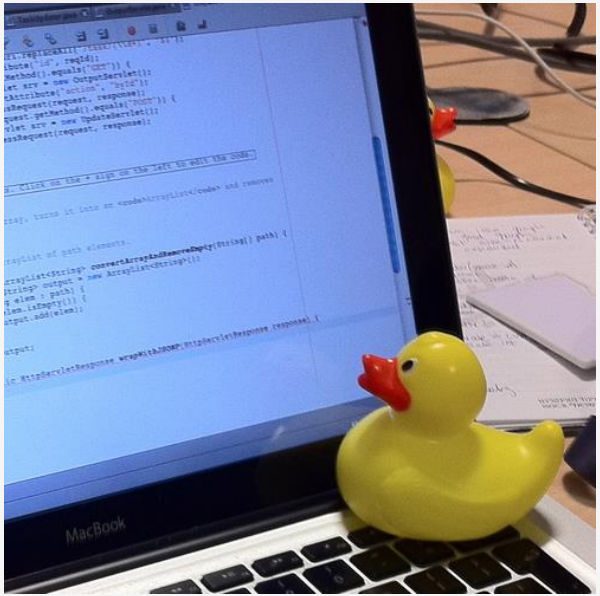
- “rubber duck debugging”
- debugger (vs code/python tutor)
- assert
- print

vanskelige feil å finne

enkle feil å finne

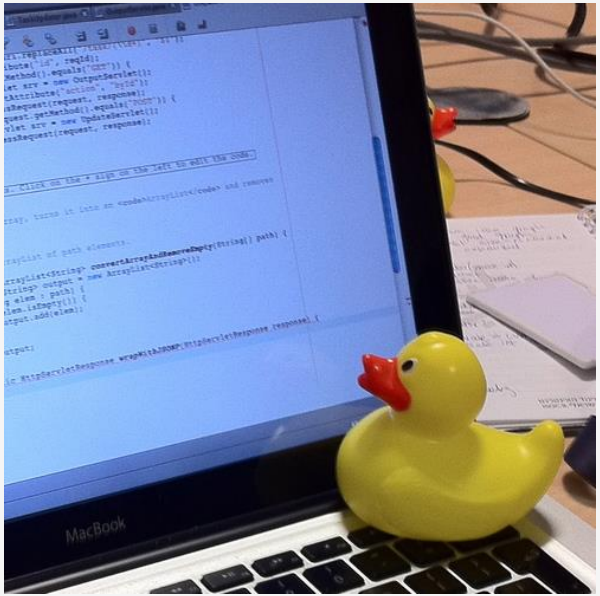


“rubber duck debugging”:
forklare problemet høyt til en gummiand (e.l.)



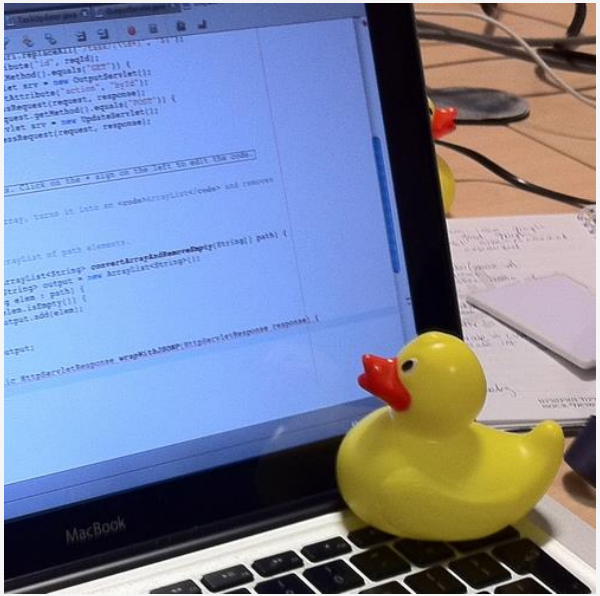
https://en.wikipedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg

hva er det du vil koden skal gjøre?
hva er det den *faktisk* gjør?

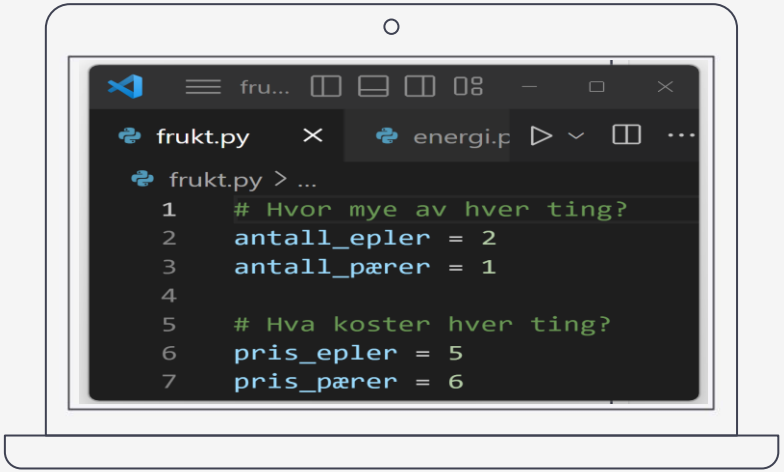


https://en.wikipedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg

å *forklare* problemet ordentlig grundig setter oss ofte på sporet av løsningen!



https://en.wikipedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg



live-koding

bruke debuggeren i vs code til å finne feil i kode som er i flere filer

hovedprogram.py + sesong.py
(oblig 8)

automatiske tester (assert)

```
kamp = sesong._runder[0][0]
assert isinstance(kamp, Kamp), f"Må bruke Kamp objekter i Sesong, ikke {type(kamp)}"
assert isinstance(kamp.hjemmelag(), Lag), f"Kamp-objektet opprettes med Lag-objekter"
assert isinstance(kamp.bortelag(), Lag), f"Kamp-objektet opprettes med Lag-objekter"
```

her: tester at objeket er av riktig type

hvis vi gjør en endring i programmet som gir et objekt feil type, vil disse testene si fra om det

obs: krever at vi forutser på forhånd hva som kan gå galt

andre viktige begreper nevnt i dag:

- **abstraksjon:** tenke på flere nivå
- **grensesnitt:** hvordan kommunisere med andre nivå (andre klasser/hovedprogram)
- **debugging:** finne feil i programmer
- **syntaksfeil:** feil som gir feilmelding
- **logisk feil:** feil som *ikke* gir feilmelding
- **"gummiand-feilsøking":** finne feil ved å forklare problemet grundig til noen andre


slidesgo