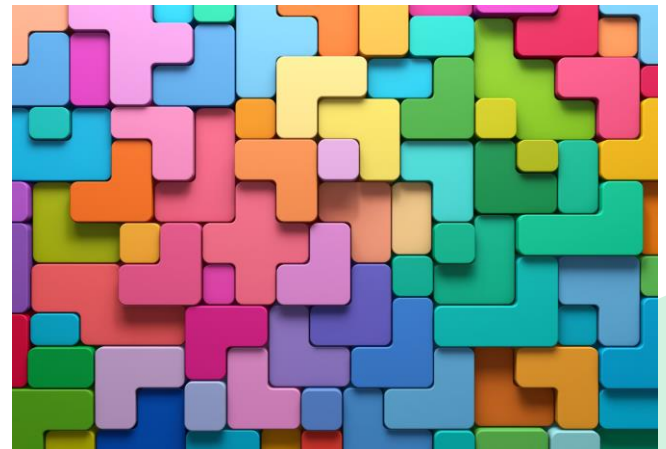
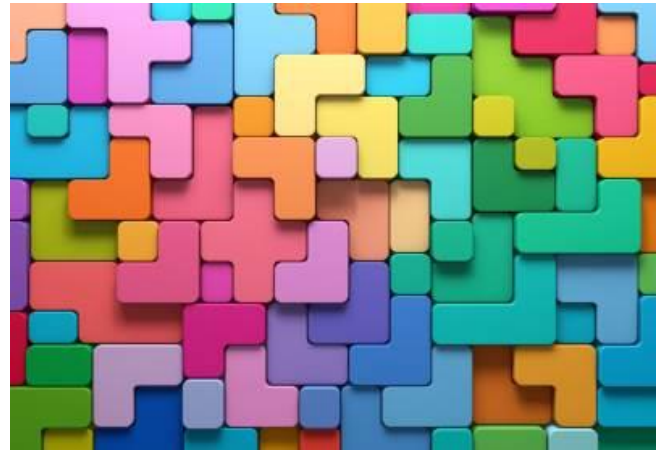


O O P

FORELESNING 5

ONSDAG 4/9



○ En liten rettelse om **Optional[...]**

Når jeg kjører mypy på calculator.py, så får jeg feilmeldingen `Unsupported operand types for +("None" and "int") [operator]`. Årsaken jeg har kommet fram til, er at jeg har en frivillig variabel i funksjonsdefinisjonen, definert ved `N: Optional[int] = 20`, og mypy klager på at det strengt tatt gir enten `None` eller `int`, hvorav den ikke liker at jeg adderer `None`. Hvordan kan jeg løse dette problemet?

Hvis jeg i stedet skriver `N: int = 20`, så klager ikke mypy. Er dette greit å bruke i stedet for `N: Optional[int] = 20`?

oddps today

Jeg synes det er en god løsning der. Egentlig har jeg oppdaget at vi bruker `Optional` litt feil. `Optional` gir mening for

```
def funksjon(x: Optional[int] = None):
```

men blir litt tullete å bruke for

```
def funksjon(x: int = 42):
```

dvs. hvis verdien blir en `int` uansett om du gir parameteren en verdi eller ikke, så trenger du ikke bruke `Optional` der.

`Optional[str]` betyr egentlig det samme som `Union[str, NoneType]` hvor `NoneType` logisk nok er typen til `None`.



- Den frivillige oppgaven:
Dette er ikke midpunktregelen

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} \frac{(x_{i+1} - x_i)}{2} (f(x_i) + f(x_{i+1})) = \frac{h}{2} \sum_{i=0}^{n-1} (f(x_i) + f(x_{i+1}))$$



- Tips: Insights → Network på GitHub



○ Prosjekt 0: Vurderingsrubrikk

Det litt typiske er at de som forstår mye tror de ligger dårligere an enn de gjør...



...og at de som virkelig trenger å ta det mer på alvor tror de ligger bedre an enn de gjør...



NYE TING I TIMEPLANEN



Bilde: bing image creator

○ Læremål: Avansert bruk av Python

- Gjøre programmene lettere å lese og finne fram i
- *Objektorientert programmering* er en måte å tenke på når vi skriver programmer som åpner nye muligheter
- (+ mer de kommende ukene)



○ Motivasjon: Hvorfor objektorientering?

- En måte å tenke på som organiserer programmer og gjør ting mer oversiktlig (i mange tilfeller)
- I stedet for *passive* samlinger med data (list, dict ++) som må sendes inn til funksjoner for å bli gjort noe med...
- ...kan man lage litt mer *selvstendige* objekter som i tillegg til å inneholde data selv vet hva de skal gjøre med disse dataene
- passiv: **resultat = gjør_noe_med(en_samling)**
- selvstendig: **resultat = et_objekt.gjør_noe_selv()**



○ *Må* man bruke objektorientering, da?

- Nei, man *kan* ha alt av data i lister eller dictionaries
- Og lage funksjoner til alt man skal gjøre med dem





```
uten_objekt.py > ...
1  from kortstokk import lag_kortstokk, trekk_kort
2
3  kortstokk = lag_kortstokk()
4  tilfeldig_kort = trekk_kort(kortstokk)
5
6  print()
7  print(tilfeldig_kort)
8  print()
9  print("type:", type(tilfeldig_kort))
10 print("farge:", tilfeldig_kort["farge"])
11 print("verdi:", tilfeldig_kort["verdi"])
12 print()
```



```
(base) PS C:\GitHub\test> python uten_objekt.py

{'farge': 'rød', 'symbol': 'ruter', 'verdi': 11, 'tekst': 'J'}

type: <class 'dict'>
farge: rød
verdi: 11
```

```
med_objekt.py > ...
1  from kortstokk import Kortstokk
2
3  kortstokk = Kortstokk()
4  tilfeldig_kort = kortstokk.trekk()
5
6  print()
7  print(tilfeldig_kort)
8  print()
9  print("type:", type(tilfeldig_kort))
10 print("farge:", tilfeldig_kort.farge)
11 print("verdi:", tilfeldig_kort.verdi)
12 print()
```

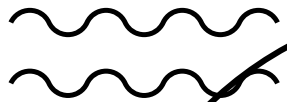
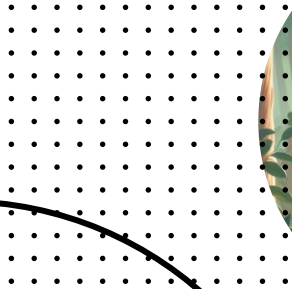


```
(base) PS C:\GitHub\test> python med_objekt.py

kløver K

type: <class 'kortstokk.Kort'>
farge: svart
verdi: 13
```





De fire
fordelene med
OOP
(de fire
pillarene)

- Abstraksjon
- Innkapsling
- Arv
- Polymorfisme

○ Abstraksjon

- I stedet for å ha 20 ulike funksjoner som gjør ting med kortstokker å holde styr på...
- ...samle alle funksjonene som representerer en Kortstokk i en klasse som tar seg av alt dette
- (og de som representerer et enkelt Kort i en annen klasse)
- Da er ting som naturlig hører sammen på samme sted
- Importerer da 2 klasser i stedet for 20 funksjoner



○ Innkapsling

- Med en ordbok vil *alle* data være synlig for programmet som bruker den
- I en klasse kan vi gjemme bort interne data og funksjoner som "omverdenen" ikke trenger å vite noe om
- Reduserer kompleksitet: fokus på *hva*, ikke *hvordan*
- *Grensesnittet* til klassen er navnet på de offentlige metodene (funksjonene) og variablene som annen kode skal bruke
- Resten er internt (bare interessant for klassen selv)



○ Arv

- Vi kan lage generelle klasser som fungerer i mange tilfeller (Kort)
- Og mer spesialiserte klasser som er tilpasset en bestemt bruk (KlassiskKort, PokemonKort)
- Alt som er felles for disse klassene er samlet i "moderklassen" Kort, og *arves* av de spesialiserte klassene (mer oversiktelig)
- De spesialiserte klassene inneholder da kun ekstra funksjoner og data som bare er relevante til deres spesielle bruk



○ Polymorfisme

- De spesielle klassene teller også som eksemplarer av "moderklassen" sin (men ikke omvendt)
 - Et KlassiskKort teller som et Kort
 - Et PokemonKort teller som et Kort
 - Men et Kort teller ikke som et PokemonKort!
- "Søskenklasser" teller ikke som hverandres type
 - et KlassiskKort teller ikke som et PokemonKort



○ Husk hvordan exceptions fungerte:

- Teller en `ArithmeticError` som en `ZeroDivisionError`?
- Teller en `ZeroDivisionError` som en `Exception`?
- Teller en `OverflowError` som en `ZeroDivisionError`?
- Hvordan fange opp både `OverflowError` og `ZeroDivisionError`, men ikke `TabError` (i en og samme **except**-blokk)?

```
+-- Exception
+-- StandardError
|   +-- ArithmeticError
|       +-- FloatingPointError
|       +-- OverflowError
|       +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- EnvironmentError
|   +-- IOError
+-- EOFError
+-- ImportError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- NameError
+-- RuntimeError
|   +-- NotImplementedError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
```

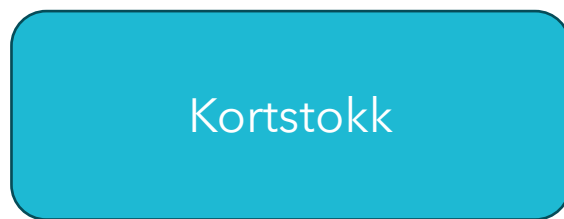




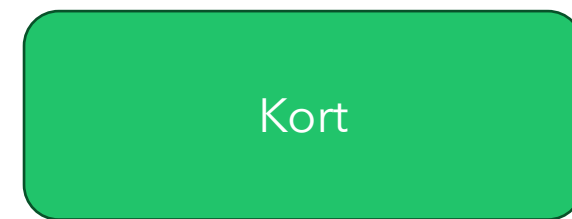
LIVEKODING:
KLASSER

○ Klassediagram

Generelle
klasser



inneholder



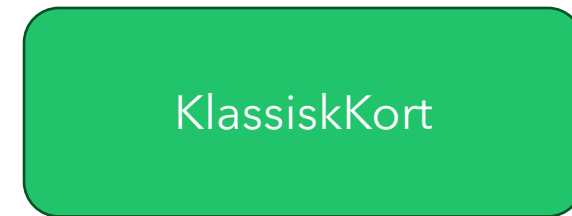
arv

arv

Mer
spesialiserte
klasser



inneholder



○ Metoder

- En vanlig funksjon kalles slik:
 - **en_returnverdi = en_funksjon(en_parameter)**
- Men vi kan også la et objekt kalle en av sine funksjoner for oss
 - **en_returverdi = et_objekt.en_metode()**
- En slik funksjon kalles en *metode*



○ self

- Klassen er felles oppskrift for *alle* objekter av en type
- Men når vi kaller en *metode* så trenger vi ofte å vite akkurat hvilket objekt som kalte metoden
- Ulike objekter av samme type kan ha helt ulike data (for eksempel tallverdi og farge på et spillkort)
- Derfor inneholder alle metoder en ekstra parameter som ikke brukes når vi kaller metoden: **self**



○ self

- Når en metode kalles:
 - **et_objekt.gjør_noe()**
 - **et_objekt.gjør_noe_med(et_argument)**
- gjør Python egentlig dette:
 - **KlassenTilObjektet.gjør_noe(et_objekt)**
 - **KlassenTilObjektet.gjør_noe_med(et_objekt, et_argument)**
- Derfor trenger vi den ekstra parameteren
 - **def gjør_noe_med(self, en_parameter):**

