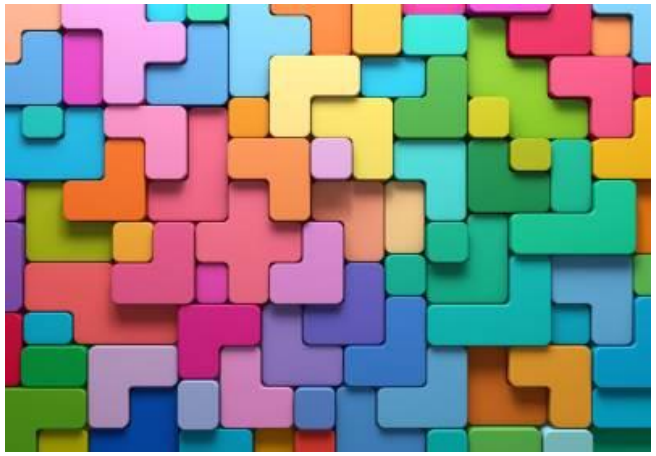
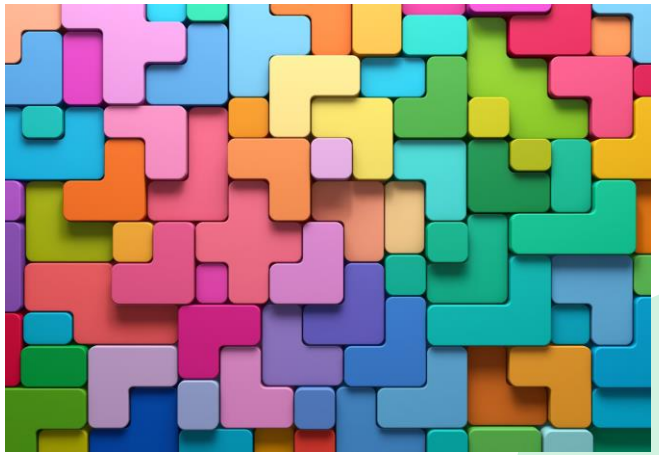
LESBAR KODE OG TESTING

FORELESNING 3

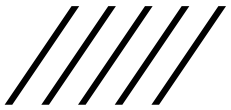
ONSDAG 28/8





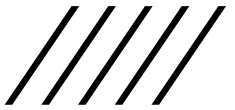
OBS: Oppdateringer på prosjekter

- Som dere har sett ligger prosjekt 0 i et eget GitHub-repository
- Følg med på oppdateringer (nye commits) underveies
- Retter feil / uklartheter (helt fram til fristen)
- Kan komme mindre endringer av selve oppgavene (til og med vi er ferdig med å gjennomgå lærestoffet til prosjektet)
- For prosjekt 0 vil det ikke bli endringer (annet enn feilretting) etter denne uken – fra og med neste uke skal dette være "låst"



Til dere som bruker git direkte i terminal

- Kanskje først og fremst aktuelt på Linux, selv om det skal være mulig å bruke GitHub Desktop også der om man vil
- Dere trenger en SSH-nøkkel som beskrevet i denne guiden (som også forklarer litt om hvordan git i termialen brukes)



Læremål: Avansert bruk av Python

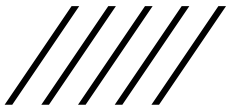
- Gjøre programmene lettere å lese og finne fram I
- (+ mer de kommende ukene)

```
 #Ugliest Code Ever
x = 0
y = 4+3
z=[1]*10
while x<y-1:
   x=x+1
for i in range(0,10,1):

 if i % 3==0:print(i,end=' ')
else:
  if i%2==0:print(i**2,end='   ')
    if(i%5==0):print('foo',end=',')
 k =[2, 3, "string",[1,2],{'key': 1}, (1, 2)]
def itDoesNothing():
                              pass
for j in range(len(z) - 1):
 z[ j ] = j
for key in k:
 if isinstance(key, int):
  print ( "int",key)
 elif isinstance(key,
str):print ("str", key)
 else:
  (lambda x:print('other',x))(key)
 x =0
if 1>0: print('foo')
 elif 0<1:
       print('bar')
k = "some-string"
def anotherFunc(asdf):
  for i in range(0,2):
     print(i, end='"')
    return 'done'
print( anotherFunc ( x))
```

```
# Calculate and print the Fibonacci sequence up to n terms

def fibonacci(n):
    """
    Generate a Fibonacci sequence of n terms.
    :param n: number of terms in the Fibonacci sequence
    :return: list of Fibonacci numbers
    """
    fib_sequence = [0, 1]

    for i in range(2, n):
        next_term = fib_sequence[i - 1] + fib_sequence[i - 2]
        fib_sequence.append(next_term)

    return fib_sequence

def print_fibonacci_sequence(n):
    """
    Print the Fibonacci sequence in a formatted way.
    :param n: number of terms to print
    """
    sequence = fibonacci(n)
    print("Fibonacci sequence up to", n, "terms:")
    for num in sequence:
        print(num, end=' ')
    print()

# Main code execution
if __name__ == "__main__":
    number_of_terms = 10  # Feel free to change this number
    print_fibonacci_sequence(number_of_terms)
```

Motivasjon: Hvorfor kodestil, dokumentasjon og type-hinting?

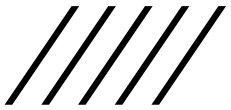
- *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"* - Martin Fowler
- Programmer som ikke er lette å lese og forstå er de som ikke blir brukt eller videreutviklet – lettere å finne opp kruttet på nytt
- Å vite typer på variabler hjelper på dette
- (I C++ og andre språk *må* vi gi variabler typer!)



Hva er kodestil

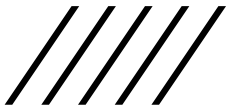
- Hvilken rekkefølge gjør vi ting i?
- Hvilke navn gir vi variabler, funksjoner og klasser?
- Hvordan bruker vi *whitespace* (mellomrom, linjeskift, …)?
- Hvordan kommenterer vi koden?

- Her er det mye som har lite å si for datamaskinen, men som har stor betydning for brukere, testere og de som skal videreutvikle programmet

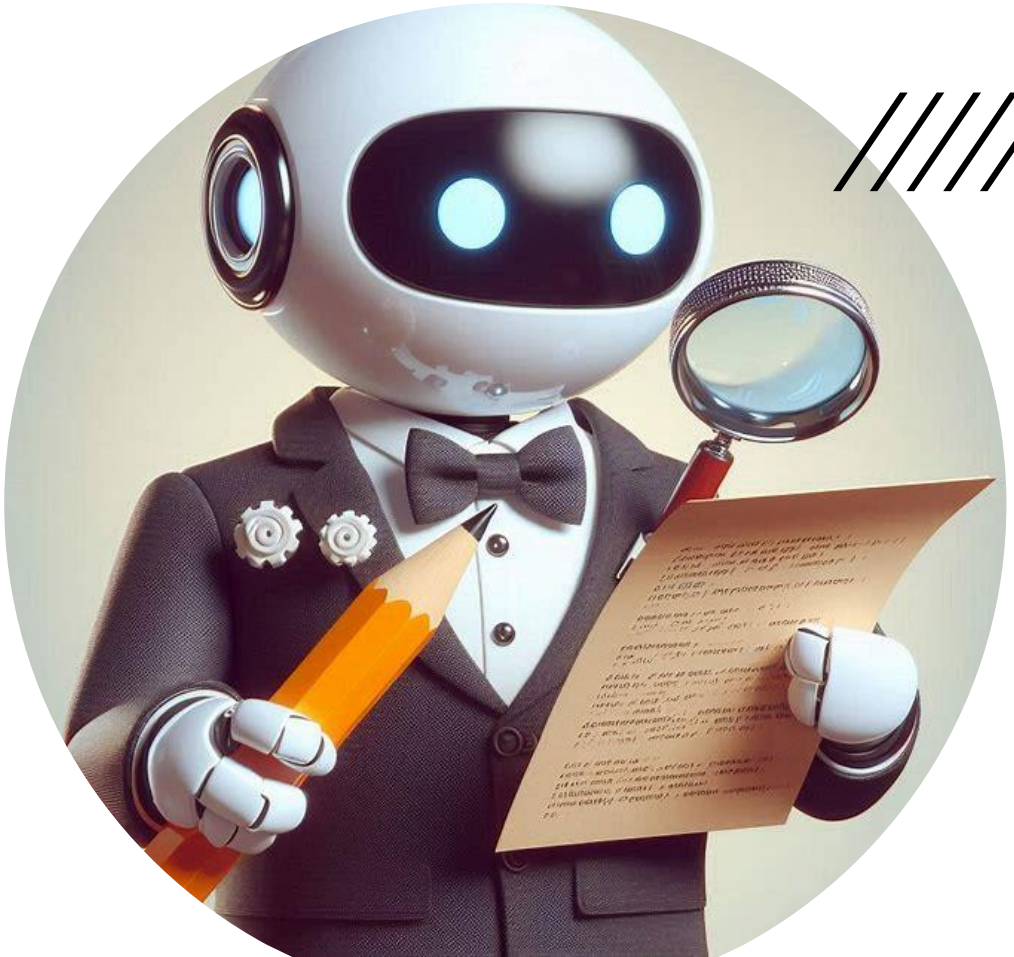


PEP8 og andre guider til kodestil

- PEP8 er en veiledning til "best practices" laget av Python-utviklere
- Må ikke følges strengt, og det finnes andre veiledninger man også kan følge isteden
- Å ha et bevisst forhold til *hvorfor* disse anbefalingene er slik de er, er kanskje det viktigste
- **OBS: Dere må følge dagens anbefalinger på variabel- og funksjonsnavn i prosjekt 0!**



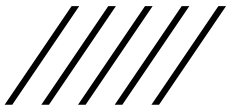
LIVEKODING: KODESTIL



(bilde: bing image creator)

Den viktigste anbefalingen

- **Ikke følg disse anbefalingene dersom koden blir *mindre* leselig som et resultat av at du gjør det**
- (ingen regel uten unntak – vit når du kan og bør bryte reglene)
- PEP8 (og lignende veiledninger) er ikke endestasjonen for vakker og forståelig kode – kun et sted å starte
- For spesielt interesserte som vil gå i dybden: Beyond PEP8 (ikke pensum)



Hva er vakker kode?

Don't let people fool you into thinking that beautiful code is the following:

- clever algorithms
- sneaky language features
- solving a problem with the least amount of key strokes

Because it's not. Code like that is *cute*, and it's certainly worth a glance, but it's not the kind of code you want to settle down with.

https://softwareengineering.stackexchange.com/questions/207929/what-is-beautiful-code

Trevor Hickey
1,023 ● 1 ● 10 ● 16

Hva er vakker kode?

And you know that fancy recursive meta templated static polymorphism that inherits variatic lambdas-- or whatever it was you read about online? You might be eager to jump on innovative and nifty tricks without a clear reason to use them. But code that pushes the boundaries of a language aren't beautiful either.

They're *sexy*.
Tons of fun, but ask yourself this: Do I really want to spend time exploring the anatomy of this language, or do I want to work together with a language and build something beautiful? After all, a programming language is merely the tool to create.

https://softwareengineering.stackexchange.com/questions/207929/what-is-beautiful-code

Trevor Hickey
1,023 ● 1 ● 10 ● 16

Hva er vakker kode?

So what is *beautiful* code then?

Beautiful code = maintainable code. THAT'S IT!
THAT'S THE FORMULA!

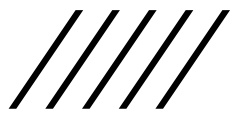
If you can write something, come back to it a few months later, and continue making progress on it, then that's beautiful. If a year later you realize that you want to add functionality as well as tweak an existing feature, and you manage to do it with relative ease, then THAT'S beautiful. If other people can step into your codebase and quickly figure out what's going on because things are organized, they'll have more hair, and also be beautiful.

https://softwareengineering.stackexchange.com/questions/207929/what-is-beautiful-code

Trevor Hickey
1,023 ● 1 ● 10 ● 16

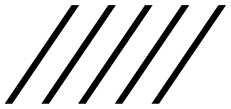
Dokumentasjon

- Se for deg at du er en annen person som åpner ditt GitHub-repository for første gang
- Vil de umiddelbart skjønner hva koden din gjør?
- Vil du *selv* skjønne det etter lang tids fravær fra denne koden?
- Det er begrenset hva koden selv (inkludert navn på variabler, funksjoner osv.) kan fortelle oss
- **Forklar *hva* koden gjør, ikke (bare) *hvordan* den gjør det**
- (hvis du fant en løsning på nett, kommenter med adressen)



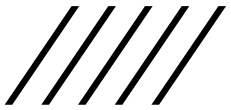
README.md

- En god README-fil gir et overblikk over koden i en mappe
- På GitHub-nettsiden vil README.md (skrevet i Markdown) vises automatisk under filene i en mappe
- Men bortsett fra det er det viktig at dokumentasjonen er i samme fil som selve koden – hvorfor?
- For at vi ikke skal glemme å oppdatere dokumentasjonen når vi oppdaterer koden!



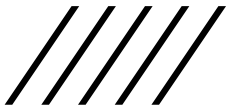
"""Docstrings"""

- Livekoding: Vi går tilbake til det vi kodet før i dag og legger til docstrings på alle funksjoner og klasser
- Docstrings er ikke kommentarer, men har samme rolle
- Livekoding: Brukere / andre utviklere kan slå opp dokumentasjon direkte i terminalen
- Verktøy som Sphinx (ikke pensum) kan automatisk lese docstrings og lage dokumentasjon som PDF eller nettside…
- …som andre kan lese etterpå



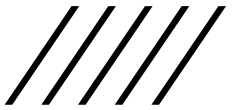
Python: Frihet under ansvar

- Python er designet slik at variabler ikke må ha en bestemt type
- Historisk sett er dette er relativt nytt og litt spesielt
  (i C++ må vi låse alle variabler til en eller annen type)
- Det gir oss frihet – vi kan ha variabler hvor vi ikke vet hva typen skal være (den kan være hva enn brukerne bestemmer når de kjører programmet)
- Men det kan også gjøre koden vanskeligere å lese og feil vanskeligere å finne (objekter kan ha feil type uten at vi vet det)



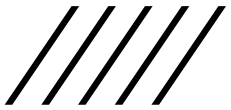
Type-annotasjoner (type-hinting)

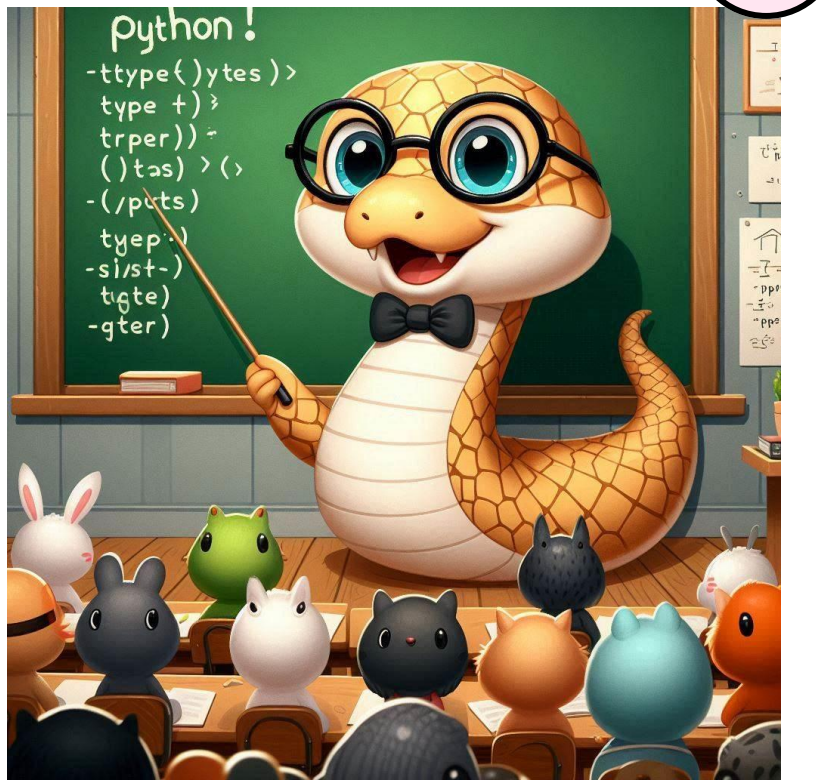
- Selv om Python ikke *krever* at vi oppgir typer til variabler, har vi likevel *lov* til å gi beskjed om hvilken type vi forventer!
- Kan gjøres på alle variabler, men vi skal hovedsakelig bruke det til parametre og returverdier i funksjoner
  (dette er et av kravene for å bestå prosjekt 0!)



Hvorfor?

- **Dokumentasjon:** Sammen med docstrings forklarer type-hintene hva som er inndata og utdata til en funksjon
- **Editor:** Vi kan få ekstra hjelp fra editoren når den vet typen til en variabel (kan slå opp metoder denne typen har)
- **Feilsøking:** Vi kan bruke en statisk type-sjekker som **mypy** til å se om noen objekter har fått andre typer enn forventet, som en del av testing/debugging





LIVEKODING: TYPE-HINTING

(BILDE: BING IMAGE CREATOR)

Etter forelesningen

- Hvis du ikke har gjort det enda – logg inn på github.uio.no (for å få GitHub-bruker) og send melding på Mattermost om at du er klar for å få repository (repo) til Prosjekt 0
- De første ukesoppgavene ligger også ute, se gjerne på dem parallelt med prosjektet