Løsningsforslag

Runde 3

3.7 Enhetstesting

3.7.1

a) Hva er hensikten med enhetstesting?
Hensikten med enhetstesting er å sjekke at små deler av koden fungerer som forventet og isolert fra andre deler av programmet.

b) Hva er en testenhet?
En testenhet er normalt en funksjon, metode eller klasse som testes individuelt.

c) Hvordan kan vi organisere testtilfeller i pytest?
Testtilfeller kan organiseres i funksjoner med navn som begynner med `test_`, og disse funksjonene plasseres i testmoduler som også begynner med `test_`. Det kan også være metoder med navn som begynner med test_, inni klasser som begynner med Test, inni moduler som begynner med test:_

d) Hvilke ulike nivåer kan vi velge å kjøre tester på ved hjelp av pytest?
Vi kan velge å kjøre tester på forskjellige nivåer, inkludert enkeltfunksjoner (inkl. metoder), klasser, moduler, mapper, og til og med hele prosjekter. Dette gir oss fleksibiliteten til å teste både små og store deler av koden vår.

e) Hva er en *fixture* i pytest, og hvordan brukes den?
Det er en funksjon som brukes til å sette opp og rydde opp i forhold som kreves for testing av en enhet. Funksjonen dekoreres med `@pytest.fixture` og returnerer eksempelvis en instans eller en variabel som senere kan brukes som argument i en eller flere testfunksjoner.

f) Hvorfor kan det være nyttig å etterligne deler av koden under testing?
Det kan være nyttig når vi vil kontrollere eller simulere deler av koden som normalt ville være utenfor vår kontroll, for eksempel eksterne tjenester eller tilfeldig genererte verdier.

g) Hvilke funksjoner i unittest-superklassen brukes for forberedelser og opprydding i testtilfellene?
`setUp` og `tearDown` brukes for forberedelser og opprydding før og etter hver testmetode.

h) Hvordan kan vi lage en HTML-rapport med testresultater ved hjelp av pytest-html?
`pytest --html=rapport.html`, eventuelt med `--self-contained-html`

3.7.2

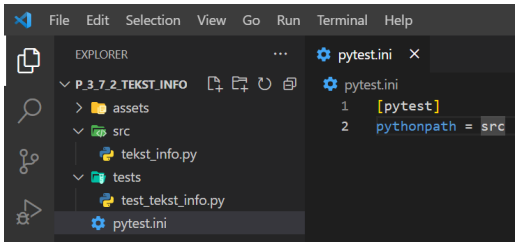
I denne oppgaven skal vi lage en klasse med metoder for å analysere tekst og deretter teste denne klassen ved hjelp av Pytest.

1. Opprett mappen tekstanalyse
2. I tekstanalyse-mappen, opprett
   a. To undermapper: src og tests , se pytest Good Integration Practices
   b. Filen pytest.ini med innholdet:

      [pytest]

      pythonpath = src

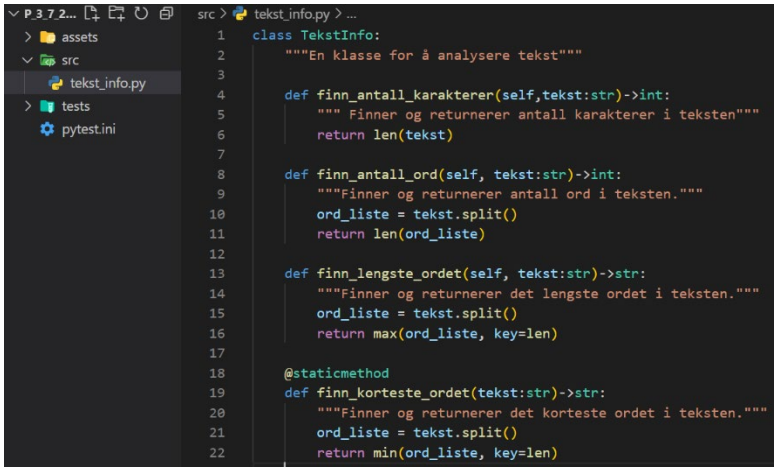
      for at pytest skal finne programmet



Se mappen p_3_7_2_tekstanalyse

3. I src-mappen, opprett filen tekst_info.py med klassen TekstInfo hvor følgende metoder skal implementers:
   a. finn_antall_karakterer(tekst:str)->int:
   b. finn_antall_ord(tekst:str)->int:
   c. finn_lengste_ord(tekst:str)->str:
   d. finn_korteste_ord(tekst:str)->str:

      Metodene tar inn en tekststreng som argument og returner henholdsvis antall karakterer, antall ord, lengste og korteste ord i teksten.



Vi har valgt å inkludere en statisk metode som kan kalles på en lignende måte som en klassemetode, uten krav om å opprette en instans først. Vi kan bruke klassens navn med stor forbokstav foran metodenavnet. Statiske metoder krever ikke en første parameter som self i objektmetoder eller cls i klassemetoder. De har heller ikke tilgang til verken klassevariabler eller objektvariabler. Se filen tekst_info.py i src-mappen

4. I tests-mappen, opprett filen test_tekst_info.py og implementer en test for hver metode ved å bruke pytest.

```python
import pytest
from tekst_info import TekstInfo

# Uten fixtures
def test_finn_antall_karakterer():
    ti = TekstInfo()
    tekst = "Dette er en testsetning"
    assert ti.finn_antall_karakterer(tekst) == 23

# Opprett en tekst vi kan bruke i testene
@pytest.fixture
def tekst():
    return "Dette er en setningen vi skal bruke i resten av testene"

def test_finn_antall_ord(tekst):
    ti = TekstInfo()
    assert ti.finn_antall_ord(tekst) == 11

# Opprett en instans vi kan bruke i testene
@pytest.fixture
def tekstinfo():
    return TekstInfo()

def test_finn_lengste_ordet(tekstinfo, tekst):
    assert tekstinfo.finn_lengste_ordet(tekst) == "setningen"

# Med statiske metoder slipper vi å opprette en instans
def test_finn_korteste_ordet(tekst):
    assert TekstInfo.finn_korteste_ordet(tekst) == "i"
```

I den første testen oppretter vi en TekstInfo-instans, en tekstvariabel, og kaller metoden i assert-setningen. Selv om vi kunne fortsatt med denne tilnærmingen, har vi valgt å vise flere alternativer.

Deretter bruker vi en *fixture* til å initialisere en tekstvariabel som vi benytter gjennom resten av testene.

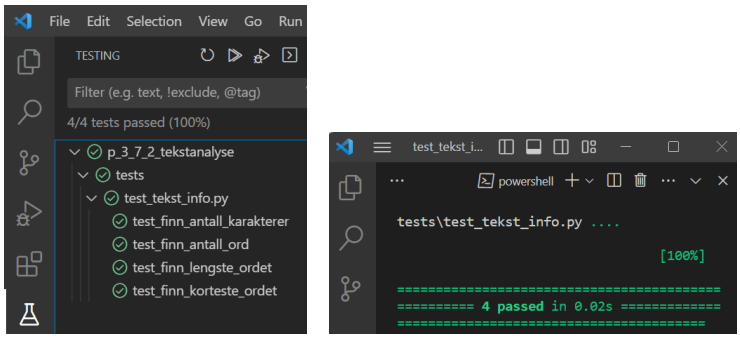
I den andre testen bruker vi denne variabelen som et argument.

Deretter bruker vi en annen *fixture* til å opprette en instans av TekstInfo-klassen, for å unngå gjentagelse i hver test.

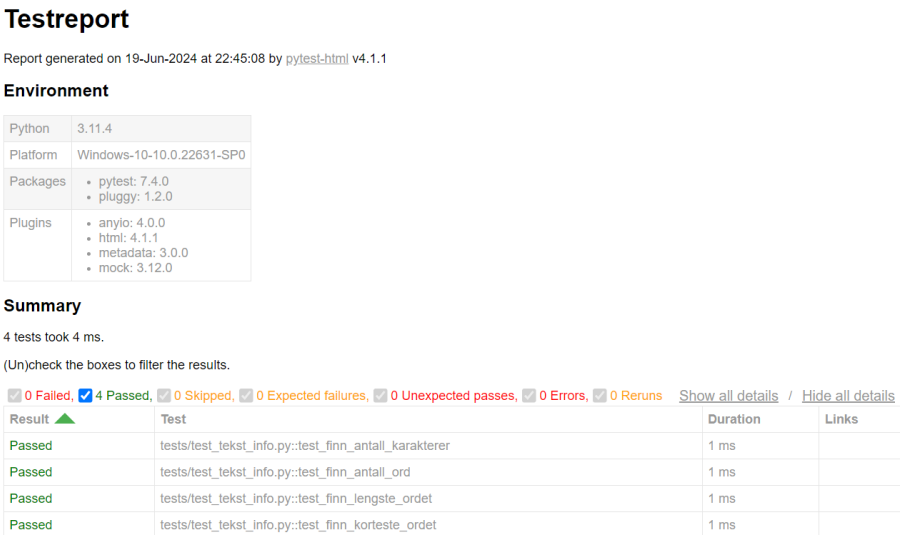
Denne instansen bruker vi i den tredje testen sammen med tekstvariabelen.

Til slutt viser vi hvordan bruken av statiske metoder forenkler koden i denne sammenhengen. Se filen test_tekst_info.py i tests-mappen

5. Ta skjermbilde etter å ha kjørt testene både med GUI i VS Code og CLI i terminalvinduet. Vis disse skjermbildene som en del av besvarelsen din.



6. Bruk pytest-html til å generere en HTML-testrapport med dokumentasjon for testresultatene. Se filen testreport.html.



Testreport

Report generated on 19-Jun-2024 at 22:45:08 by pytest-html v4.1.1

Environment

Python	3.11.4
Platform	Windows-10-10.0.22631-SP0
Packages	• pytest: 7.4.0 • pluggy: 1.2.0
Plugins	• anyio: 4.0.0 • html: 4.1.1 • metadata: 3.0.0 • mock: 3.12.0

Summary

4 tests took 4 ms.

(Un)check the boxes to filter the results.

0 Failed, 4 Passed, 0 Skipped, 0 Expected failures, 0 Unexpected passes, 0 Errors, 0 Reruns Show all details / Hide all details

Result	Test	Duration	Links
Passed	tests/test_tekst_info.py::test_finn_antall_karakterer	1 ms	
Passed	tests/test_tekst_info.py::test_finn_antall_ord	1 ms	
Passed	tests/test_tekst_info.py::test_finn_lengste_ordet	1 ms	
Passed	tests/test_tekst_info.py::test_finn_korteste_ordet	1 ms	

3.7.3

Oppgave 6 IT-2 Eksamen Vår 2024

Vi ønsker å lage en liten kalkulator for de fire regneoperasjonene – minus, pluss, gange og dele. Nedenfor finner du pseudokode som beskriver en del av en løsning ved hjelp av fire funksjoner, vist til høyre.

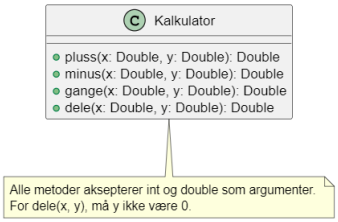
```
FUNCTION pluss(a, b)
    RETURN a + b
ENDFUNCTION

FUNCTION minus(a, b)
    RETURN a - b
ENDFUNCTION

FUNCTION gange(a, b)
    RETURN a * b
ENDFUNCTION

FUNCTION dele(a, b)
    RETURN a / b
ENDFUNCTION
```

a) Lag et klassediagram for en tilsvarende klasse kalt Kalkulator til bruk i en objektorientert løsning.



Se 6a_kalkulator.png

b) Implementer klassen i ditt programmeringsspråk.

```python
class Kalkulator:
    def pluss(self, a, b):
        return a + b

    def minus(self, a, b):
        return a - b

    def gange(self, a, b):
        return a * b

    def dele(self, a, b):
        return a / b
```

Se filen 6b_kalkulator.py

c) Implementer et egnet testprogram for å teste løsningen, og identifiser mulige feil og unntak. Kjør testene med pytest

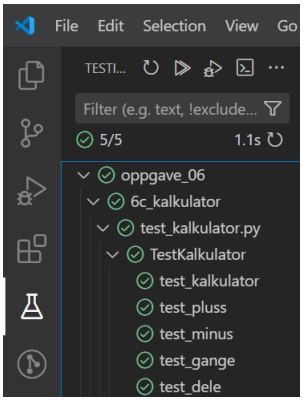
```python
def test_kalkulator(self, kalkulator):
    assert kalkulator is not None
    assert isinstance(kalkulator, Kalkulator)

def test_pluss(self, kalkulator):
    assert kalkulator.pluss(1, 2) == 3
    # Feil type
    with pytest.raises(TypeError):
        kalkulator.pluss(2, "3")

def test_minus(self, kalkulator):
    assert kalkulator.minus(5, 2) == 3
    # For få argumenter
    with pytest.raises(TypeError):
        kalkulator.minus(8)

def test_gange(self, kalkulator):
    assert kalkulator.gange(8, 13) == 104
    assert kalkulator.gange(math.pi, 2) \
        == pytest.approx(6.28, rel=0.001)

def test_dele(self, kalkulator):
    assert kalkulator.dele(10, 2) == 5
    # Divisjon med 0
    with pytest.raises(ZeroDivisionError):
        kalkulator.dele(5, 0)
```



Se filene test_kalkulator.py og testkjøring.png i mappen 6c_kalkulator

Mulige feil:

- Argumentene kan være noe annet enn float eller int.
- Vi kan ikke dele på 0

d) Implementer nødvendig håndtering av mulige feil og unntak.

```python
class Kalkulator:
    def sjekk_input(self, a, b):
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Feil: Inndata må være tall.")

    def pluss(self, a: float, b: float) -> float:
        self.sjekk_input(a, b)
        return a + b

    def minus(self, a: float, b: float) -> float:
        self.sjekk_input(a, b)
        return a - b

    def gange(self, a: float, b: float) -> float:
        self.sjekk_input(a, b)
        return a * b

    def dele(self, a: float, b: float) -> float:
        self.sjekk_input(a, b)
        if b == 0:
            raise ZeroDivisionError("Feil: Divisjon med null er ikke tillatt.")
        return a / b
```

Se filen 6d_kalkulator.py

```python
# Eksempel på bruk
if __name__ == "__main__":
    kalkulator = Kalkulator()
    try:
        print(f"Addisjon: 2 + 3 = {kalkulator.pluss(2, 3)}")
        print(f"Subtraksjon: 5 - 3 = {kalkulator.minus(5, 3)}")
        print(f"Multiplikasjon: 5 * 8 = {kalkulator.gange(5, 8)}")
        print(f"Divisjon: 13 / 8 = {kalkulator.dele(13, 8):.2f}")
        print(f"Divisjon: 5 / 0 = {kalkulator.dele(5, 0)}")
    except ZeroDivisionError as e:
        print(e)
    except TypeError as e:
        print(e)
    try:
        print(f"Ugyldig inndata: 5 + 'a' = {kalkulator.pluss(5, 'a')}")
    except TypeError as e:
        print(e)
    try:
        print(f"Feil antall argumenter: 2 - = {kalkulator.minus(2)}")
    except TypeError as e:
        print("Feil: Kalkulator.minus() mangler 1 argument: 'b'")
```

```
Addisjon: 2 + 3 = 5
Subtraksjon: 5 - 3 = 2
Multiplikasjon: 5 * 8 = 40
Divisjon: 13 / 8 = 1.62
Feil: Divisjon med null er ikke tillatt.
Feil: Inndata må være tall.
Feil: Kalkulator.minus() mangler 1 argument: 'b'
```

