9 Enhetstesting

9.1 Læringsmål dette kapitel

- Forklare hva enhetstesting er og hvorfor det er viktig i programvareutvikling
- Bruke `assert` for enkel testing av funksjoner i Python
- Skrive strukturerte tester med `unittest`-rammeverket
- Organisere testkode i egne testklasser og testfiler
- Forstå hvordan testverktøy gir tilbakemelding ved feil
- Skille mellom kode og tester i små og større prosjekter
- Bruke `subTest` for å teste flere verdier i én testmetode
- Kjenne til vanlige `assert`-metoder i `unittest`

9.2 Innledning

Når vi skriver programmer er det lett å tenke at koden fungerer slik vi har tenkt – helt til den ikke gjør det. Derfor er testing en viktig del av utviklingsarbeidet, og *enhetstesting* er en metode som hjelper oss å sjekke at små deler av programmet (funksjoner, metoder, klasser) oppfører seg riktig.

Ideen om å teste programvare systematisk har utviklet seg over tid, og en av de som har hatt stor innflytelse er **Kent Beck**. Han var med på å utvikle både **JUnit** (for Java) og **Test-Driven Development (TDD)**, der man skriver testene *før* man skriver selve koden. Dette har vært med på å gjøre testing til en naturlig del av programmeringsprosessen – ikke bare noe man gjør til slutt.

I Python har vi flere rammeverk for testing, blant annet:

- doctest (tester basert på docstrings)
- unittest (standardbiblioteket)
- pytest (populært og fleksibelt)

Her skal vi bruke **unittest**, som følger med Python og derfor ikke krever ekstra installasjon. Det gir oss et strukturert rammeverk for å skrive og kjøre tester, og passer godt til å lære de grunnleggende prinsippene.

9.3 `assert` – banalt men basalt i testing av kode

Når vi tester kode, handler det i bunn og grunn om å sjekke at programmet gjør det vi forventer. En av de enkleste måtene å gjøre dette på i Python er med kommandoen `assert`.

`assert` er et **innebygd statement** i Python – ikke en funksjon – og brukes til å uttrykke en påstand som skal være sann. Hvis påstanden er usann, stopper programmet og gir en feilmelding. Dette gjør `assert` nyttig for enkel testing og feilsøking, spesielt når man jobber interaktivt i REPL:

```
>> assert 2 + 2 == 4
>>> assert "hello".upper() == "HELLO"
>>> assert len([1, 2, 3]) == 3
>>> assert 2 + 2 == 5
Traceback (most recent call last):
  ...
AssertionError
```

Som vi ser: hvis uttrykket er sant, skjer ingenting. Hvis det er usant, får vi en `AssertionError`. Dette gir oss en enkel måte å sjekke at koden fungerer som forventet.

9.3.1 Testing av funksjoner med assert

La oss si vi har en enkel funksjon som utfører addisjon og subtraksjon:

```
def calculate(a, b, operation):
    if operation == "add":
        return a + b
    elif operation == "sub":
        return a - b
    else:
        return None
```

Vi kan teste den slik:

```
assert calculate(2, 3, "add") == 5
assert calculate(5, 2, "sub") == 3
```

```
assert calculate(0, 0, "add") == 0
assert calculate(-1, -1, "sub") == 0
```

Dette fungerer fint for små tester, men blir fort uoversiktlig når vi har mange tester eller ønsker mer struktur.

9.4 **unittest** – strukturert testing med ***egne*** assert-metoder

Python har et innebygd testverktøy som heter **unittest**. Det gir oss en mer profesjonell måte å skrive tester på, med egne metoder som ligner på `assert`, men som gir bedre tilbakemeldinger og kan kjøres samlet.

Under følger et eksempel på en testklasse (alle testmetoder må være inne i en testklasse) med tre tester:

```
import unittest
from calculate import calculate # det vi skal teste

class TestCalculate(unittest.TestCase):
    def test_addition(self): # en testmetode
        self.assertEqual(calculate(2, 3, "add"), 5)

    def test_subtraction(self): # testmetode
        self.assertEqual(calculate(5, 2, "sub"), 3)

    def test_illegal_operation(self): # testmetode
        self.assertIsNone(calculate(1, 1, "mult"))

if __name__ == "__main__":
    unittest.main()
```

Første linje med kode importerer testrammeverket unittest.

Testene må ligge i en klasse som arver **`unittest.TestCase`**. La testene (som blir metoder i klassen) ha *beskrivende navn* som forteller hva vi skal teste.

Her bruker vi metoder som `self.assertEqual(...)` og `self.assertIsNone(...)` i stedet for Python sin innebygde

`assert`. Disse metodene gir mer informasjon når tester feiler, og gjør det enklere å organisere testene i klasser og filer.

Tilbakemeldingene vi får fra unittest rammeverket vil i tilfellet over være:

```
...
----------------------------------------------------
Ran 3 tests in 0.001s
OK
```

Dette betyr at alle tre testene (`test_addition`, `test_subtraction`, `test_illegal_operation`) ble kjørt, og ingen feilet.

La oss si du har en feil i én test, for eksempel:

```
self.assertEqual(calculate(2, 3, "add"), 6) # Feil forventet verdi
```

Da får du en feilmelding som viser hvilken test som feilet og hvorfor:

```
F..
======================================================
FAIL: test_addition (__main__.TestCalculate)
------------------------------------------------------
Traceback (most recent call last):
  File "test_calculate.py", line 8, in test_addition
    self.assertEqual(calculate(2, 3, "add"), 6)
AssertionError: 5 != 6
------------------------------------------------------
Ran 3 tests in 0.001s
FAILED (failures=1)
```

Forklaring:
`F..` betyr: første test feilet (`F`), de to neste passerte (`..`).

Du får en **traceback** som viser hvor feilen skjedde.

`AssertionError: 5 != 6` forteller at funksjonen returnerte `5`, men testen forventet `6`.

9.5 Hvor skal koden ligge i forhold til testene?

Når vi begynner å skrive tester for funksjoner og klasser, oppstår et praktisk spørsmål: **Hvor bør selve koden ligge, og hvor bør testene ligge?** Det finnes flere måter å organisere dette på, og valget avhenger ofte av prosjektets størrelse og kompleksitet.

9.5.1 Kode og tester i samme fil

I små prosjekter, eller når man lærer seg testing, er det helt greit å ha både koden og testene i **samme fil**. Dette gjør det enkelt å se sammenhengen mellom funksjonen og testen.

Eksempel:

```
def calculate(a, b, operation):
    if operation == "add":
        return a + b
    elif operation == "sub":
        return a - b
    else:
        return None

import unittest

class TestCalculate(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(calculate(2, 3, "add"), 5)
```

Ulempen med denne tilnærmingen er at det fort blir uoversiktlig når koden og testene vokser, og vi ønsker i prinsippet at kode og tester skal vedlikeholdes hver for seg.

9.5.2 Egen testfil

Det er mer ryddig å **skille testene fra selve koden**. Da legger man testene i en egen fil, ofte med navn som begynner med test_, for eksempel:

```
calculate.py          # Her ligger funksjonen
test_calculate.py     # Her ligger testene
```

I testfilen må man importere funksjonen:

```
from calculate import calculate
```

Selv om det er andre måter å organisere tester på, så vil vi bruke denne tilnærmingen. Det gir en enkel struktur, en enkel import setning, og vi kan starte testen på akkurat samme måte som vi kjører en ordinær .py fil.

9.6 Organisering i større / profesjonelle prosjekter

Kun til orientering, vi bruker ikke denne tilnærmingen.

Etter hvert som prosjekter vokser i størrelse og kompleksitet, blir det viktig å ha en mer gjennomtenkt struktur for hvordan både kildekode og testkode organiseres. I profesjonelle utviklingsmiljøer er det vanlig å skille tydelig mellom funksjonell kode og testkode, og å bruke en struktur som gjør det enkelt å vedlikeholde, utvide og kjøre tester på tvers av hele prosjektet.

Hvis vi relaterer dette til f eks kildekode for ei lærebok, som er kapittelinndelt, så ville strukturen kunne se slik ut:

```
bok/
├── src/
│   ├── kapittel1/
│   │   └── calculate.py
│   ├── kapittel2/
│   │   └── analyse.py
├── tests/
│   ├── kapittel1/
│   │   └── test_calculate.py
│   ├── kapittel2/
```

Her ligger all kildekode i en src/-katalog, og testene er organisert i en parallell tests/-katalog som speiler strukturen i src/. Dette gir god oversikt og gjør det enkelt å finne testene som hører til hver modul.

Fordeler med denne strukturen

- **Skalerbarhet**: Prosjektet kan vokse uten at strukturen blir uoversiktlig.
- **Separasjon av ansvar**: Kode og tester holdes adskilt, noe som gjør vedlikehold enklere.
- **Støtte for testverktøy**: Verktøy som unittest, pytest og CI/CD (*) -systemer forventer ofte en slik struktur.
- **Enklere testkjøring**: Du kan kjøre alle tester under ett.

(*) CI/CD er en samlebetegnelse for prosesser og verktøy som automatiserer bygging, testing og distribusjon av programvare. Målet er å gjøre utviklingen mer effektiv, pålitelig og rask.

9.7 Konvensjoner for navngiving

unittest, avhengig av konfigurering, er i stand til å finne testene dersom vi er konsekvent i navngivingen.

Vi velger å bruke følgende:

Filnavn med tester starter med test_

Eksempel: `test_calculate.py`

Testklasser må arve fra `unittest.TestCase`

Eksempel:

```
class TestCalculate(unittest.TestCase)
```

Testmetoder starter med `test_`

Eksempel:

```
def test_addition(self):
    self.assertEqual(calculate(2, 3, "add"), 5)
```

9.8 Eksempler på ulike unittest asserts

9.8.1 Test på omtrentlig verdi

Det vi tester:

```
# file: sc_09_02_area_circle.py

def area_circle(radius):
    return 3.14159 * radius * radius
```

Testen:

```
import unittest
from sc_09_02_area_circle import area_circle

class TestArea(unittest.TestCase):
    def test_area(self):
        self.assertAlmostEqual(area_circle(1), 3.14159, places=3)
        self.assertAlmostEqual(area_circle(2), 1.56736, places=3)

unittest.main()
```

Utskrift fra testen:

```
F
======================================================
FAIL: test_area (__main__.TestArea)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\...\tb_sc\ch_09_unit_testing\test_sc_09_02_area_circle.py",
line 8, in test_area
    self.assertAlmostEqual(area_circle(2), 12.56736, places=3)
AssertionError: 12.56636 != 12.56736 within 3 places
(0.0010000000000012221 difference)
----------------------------------------------------------------------
Ran 1 test in 0.000s


FAILED (failures=1)
```

Testen har to `assertAlmostEqual()`, den ene feiler fordi det er avvik i desimalposisjon 3.

Merk: I utskriften vil det kun komme ut en 'F' dersom det er flere asserts i en og samme test.

9.8.2 Test med flere verdier som input

Det vi tester (en feil er lagt inn i koden):

```
# file: sc_09_03_leapyear.py
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year // 400 == 0):
        return True
    else:
        return False
```

Her er det lagt inn en feil; - i siste test som *skal* være `(year % 400 == 0)` er % erstattet av //, hvilket medfører at 2000, som er et skuddår, blir underkjent.

Testen:
I testen under bruker vi en todimensjonal liste av tupler for å angi input og forventet verdi. Det er laget to testmetoder, `test_leap_years_v1` og `test_leap_years_v2` for å vise to forskjellige teknikker når vi bruker løkke.

```
# file: test_sc_09_03_leapyear.py
import unittest
from sc_09_03_leapyear import is_leap_year
class TestLeapYear(unittest.TestCase):
    test_data = [
            (2000, True),  # Divisible by 400
            (1900, False), # Divisible by 100 but not by 400
            (2004, True),  # Divisible by 4 but not by 100
            (2001, False), # Not divisible by 4
        ]
    def test_leap_years_v1(self):
        # Enkel løkke og egendefinert melding
        for year, expected in test_data:
           self.assertEqual(is_leap_year(year),
                            expected,
                            msg=f"Feil for år {year}")
```

```
    def test_leap_years_v2(self):
        # Med subTest og egendefinert melding
        for year, expected in test_data:
           with self.subTest(year=year):
               self.assertEqual(is_leap_year(year),
                                expected,
                                msg=f"Feil for år
{year}")

unittest.main()
```

Forklaring:

```
test_data = [
            (2000, True),  # Divisible by 400
            (1900, False), # Divisible by 100 but not by 400
            (2004, True),  # Divisible by 4 but not by 100
            (2001, False), # Not divisible by 4
        ]
```

...er en liste med tupler som representerer henholdsvis input og respons fra den funksjonen vi tester.

Løkka i **`test_leap_years_v1()`** vil ***stoppe*** når en feil oppstår, og eventuelle gjenværende testverdier i lista vil ikke bli utført.

Alle `assertXXX` metoder i unittest har en tredje parameter **`msg`** hvor en kan legge inn en brukerdefinert melding, hvilket er gjort her. Hvis ikke denne hadde vært lagt inn, ville vi ikke fått beskjed om hvilken verdi det var som feilet.

Løkka i **`test_leap_years_v2()`** vil ***ikke*** stoppe dersom en av verdiene gir feil før løkka er ferdig. Dette er på grunn av setningen `with self.subTest(year=year):.`

Den første `year` (til venstre for =) er navnet på parameteren som vises i test-rapporten. Den andre `year` (til høyre for =) er variabelen fra løkken.

`subTest` er en såkalt *context manager* og må brukes sammen med nøkkelordet `with`.

Fordelen med `subTest` er at alle testene kjøres, og du får en egen feilmelding for hver verdi som feiler, i stedet for at testen stopper ved første feil.

`subTest` kan også brukes i testmetoder med flere asserts, hvis du vil at alle testene skal kjøres selv om én feiler.

Så utskriften fra disse testene vil gi:

```
F
======================================================
FAIL: test_leap_years_v1 (__main__.TestLeapYear)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "... \tb_sc\ch_09_unit_testing\test_sc_09_03_leapyear.py", line 15, in test_leap_years_v1
    self.assertEqual(is_leap_year(year),
AssertionError: False != True : Feil for år 2000


======================================================
FAIL: test_leap_years_v2 (__main__.TestLeapYear) (year=2000)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "... tb_sc\ch_09_unit_testing\test_sc_09_03_leapyear.py", line 23, in test_leap_years_v2
    self.assertEqual(is_leap_year(year),
AssertionError: False != True : Feil for år 2000


----------------------------------------------------------------------
Ran 2 tests in 0.001s
```

La oss si vi legger inn en feil i `test_data` tabellen slik at siste test blir `2008, False` i stedet for `2001, False`.

Da vil `test_leap_years_v2` komme ut med *to* feil:

```
AssertionError: False != True : Feil for år 2000
```

AssertionError: True != False : Feil for år 2008

…mens den første testmetoden `test_leap_years_v2` ville blitt avbrutt etter første feil.

9.8.3 subTest

Du kan bruke `subTest` uten løkke, for eksempel når du vil teste flere relaterte påstander i samme testmetode og ønsker separat rapportering for hver, uten at testen avbrytes:

```
import unittest
class TestMath(unittest.TestCase):
    def test_simple(self):
        with self.subTest(msg="Test addisjon"):
            self.assertEqual(2 + 2, 5)
        with self.subTest(msg="Test multiplikasjon"):
            self.assertEqual(3 * 3, 8)
        with self.subTest(msg="Test subtraksjon"):
            self.assertEqual(5 - 2, 4)

if __name__ == "__main__":
    unittest.main()
```

Her er det feil i alle testene, men alle vil kjøre.

Utskrift (forkortet):

FAIL: test_simple (__main__.TestMath) [Test addisjon]

self.assertEqual(2 + 2, 5)

AssertionError: 4 != 5

FAIL: test_simple (__main__.TestMath) [Test multiplikasjon]

self.assertEqual(3 * 3, 8)

AssertionError: 9 != 8

FAIL: test_simple (__main__.TestMath) [Test subtraksjon]

self.assertEqual(5 - 2, 4)

AssertionError: 3 != 4

9.9 Oversikt unittest assert metoder

Tabellen under viser de mest brukte assert-metodene i Python sitt unittest-rammeverk.

Metode	Beskrivelse
`assertEqual(a, b)`	Tester at a == b
`assertNotEqual(a, b)`	Tester at a != b
`assertTrue(x)`	Tester at x er sann
`assertFalse(x)`	Tester at x er usann
`assertIsNone(x)`	Tester at x er None
`assertIsNotNone(x)`	Tester at x ikke er None
`assertAlmostEqual(a, b, tol)`	Tester at a og b er omtrent like (desimaltall, med toleranse som går på ulikhet ved desimal `tol`)
`assertGreater(a, b)`	Tester at a > b
`assertLess(a, b)`	Tester at a < b
`assertIn(a, b)`	Tester at a finnes i b (f.eks. element i liste eller tegn i streng)
`assertNotIn(a, b)`	Tester at a ikke finnes i b
`assertIsInstance(x, type)`	Tester at x er en instans av type
`assertRaises(error)`	Tester at en funksjon kaster en bestemt feil (krever with-blokk)

9.10setUp metoden

Det er viktig å forstå at testene vi lager kjøres *uavhengig* av hverandre, og i tilfelding rekkefølge. Det betyr at vi ikke kan vite tilstanden til et objekt som vi tester, med mindre vi tar spesielle forholdsregler.

Dersom vi lager metoden `setUp()`, *så blir den kjørt foran hver eneste test*. Her et eksempel hvor vi tester en klasse `Account`, og hvor vi sikrer oss at `balance` er den samme for alle testmetoder:

```
import unittest

class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def get_balance(self):
        return self.balance

class TestBankAccount(unittest.TestCase):
    def setUp(self):
        # Denne metoden kjøres før hver test
        self.account = BankAccount(balance=1000)

    def test_initial_balance(self):
        self.assertEqual(self.account.get_balance(), 1000)

    def test_deposit(self):
        self.account.deposit(500)
        self.assertEqual(self.account.get_balance(), 1500)

    def test_withdraw(self):
        self.account.withdraw(300)
        self.assertEqual(self.account.get_balance(), 700)

    def test_overdraw(self):
        with self.assertRaises(ValueError):
            self.account.withdraw(2000)

if __name__ == "__main__":
    unittest.main()
```

`unittest` har også en metode `tearDown()`, som kjøres *etter* hver testmetode.

9.11setUpClass metoden

`setUpClass()` er en metode i `unittest` som kjøres *én gang før alle testene i en testklasse*, i motsetning til `setUp()` som kjøres *før hver enkelt testmetode*. Den brukes når du har *ressurser som er dyre å opprette*, eller når du vil *initialisere noe én gang* og gjenbruke det i alle testene. Bruk `setUpClass`:

- Når du tester mot en database eller API og vil opprette én tilkobling
- Når du har store datastrukturer som skal brukes i alle tester
- Når du vil spare tid ved å unngå gjentatt initialisering

Her et eksempel hvor vi initialiserer en klasse med startdata som vi ikke har til hensikt å endre, slik at alle testmetodene bruker det samme datagrunnlaget:

```
import unittest
class DataProcessor:
    def __init__(self, data):
        self.data = data
    def count_items(self):
        return len(self.data)
    def has_item(self, item):
        return item in self.data

class TestDataProcessor(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # Kjøres én gang før alle testene
        cls.shared_data = ["apple", "banana", "cherry"]
        cls.processor = DataProcessor(cls.shared_data)

    def test_count(self):
        self.assertEqual(self.processor.count_items(), 3)

    def test_has_apple(self):
```

```
        self.assertTrue(self.processor.has_item("apple"))

    def test_has_orange(self):
        self.assertFalse(self.processor.has_item("orange"))

if __name__ == "__main__":
    unittest.main()
```

Merk:

- `setUpClass()` må være en *klassemetode* (`@classmethod`) og ta `cls` som parameter
- Du kan også bruke `tearDownClass()` for å rydde opp etterpå (f.eks. lukke databaseforbindelser)

`setUpClass()` egner seg altså når:

- *Alle testmetoder skal bruke samme data eller objekt*
- *Ingen av testene endrer på denne dataen* (eller endringene ikke påvirker andre tester)
- Du ønsker å *initialisere én gang* for ytelse eller struktur

Hvis testene *modifiserer* dataen som ble opprettet i `setUpClass()`, kan det føre til *avhengigheter mellom tester*, og det bør unngås. I slike tilfeller er `setUp()` tryggere.

9.12 Klassiske retningslinjer for gode unit tester

Prinsippene som nevnes under har vært sentrale i bøker som *The Art of Unit Testing* (Roy Osherove) og *Test-Driven Development* (Kent Beck):

Isolér tester
Hver test skal teste én ting og være uavhengig av andre tester. Unngå avhengigheter til eksterne systemer (database, nettverk osv.) – bruk mocks/stubs ved behov (mock / stubs er beskrevet i egne underkapitler)

Navngi tester beskrivende
Testnavn skal tydelig beskrive hva som testes og forventet utfall.

AAA-mønsteret (Arrange, Act, Assert)
Del opp testene i tre deler:

- *Arrange*: Sett opp testdata og miljø
- *Act*: Kjør funksjonen/metoden
- *Assert*: Sjekk at resultatet er som forventet

Arrange vil kanskje innebære at du lager `setUp()` og `setUpClass()` metodene (eller gjør datainitering i kodelinjene før Act + Assert), mens *Act / Assert* er selve testen og sjekking av resultatet i forhold til forventet.

Unngå logikk i tester
Testene skal være enkle og *ikke inneholde egne if-setninger* eller løkker (bortsett fra parametriserte tester).

Test kun én ting per test
Hver testmetode bør kun ha én årsak til å feile.

Hold testene små og raske
Unit tester skal kjøres ofte og raskt – de skal ikke være avhengige av store datasett eller lang kjøretid.

Gjør det lett å kjøre testene
Organiser testene / testene bør kunne kjøres med én kommando.

Test kun public / offentlig grensesnitt
Ikke test private metoder direkte – test gjennom det offentlige API-et.

Unngå duplisering i testkode
Gjenbruk setup / teardown og bruk hjelpefunksjoner der det gir mening.

Sørg for at testene er deterministiske
Testene skal alltid gi samme resultat hvis koden ikke er endret.

9.13 Test coverage

Test coverage (testdekning) er et mål på *hvor stor del av koden som blir kjørt* når du kjører testene dine. Det brukes for å vurdere hvor godt testene dine dekker funksjonaliteten i programmet.

Testdekning gir ikke garanti for at koden er feilfri, men høy dekning kan indikere at koden er godt testet.

Typer test coverage

De vanligste typene er:

- **Line coverage**: Hvor mange linjer av koden som faktisk blir kjørt.
- **Branch coverage**: Om alle grener i if/else-setninger blir testet.
- **Function/method coverage**: Om alle funksjoner/metoder blir kalt.
- **Condition coverage**: Om alle boolske uttrykk evalueres både til True og False.

9.13.1 Line coverage

Definisjon: Måler *hvor mange linjer* av koden som blir kjørt under testing.

Eksempel:

```
def greet(name):
    print(f"Hello, {name}!")
```

Test:

```
def test_greet():
    greet("Alice")
```

Hvis testen kjører `greet("Alice")`, blir linjen med `print(...)` dekket.

9.13.2 Branch coverage

Definisjon: Måler om *alle grener* i kontrollstrukturer (f.eks. if/else) blir testet.

Eksempel:

```
def is_even(n):
    if n % 2 == 0:
```

```
        return True
    else:
        return False
```

Test:

```
def test_even():
    assert is_even(2) == True
```

Branch coverage er **ikke komplett** her, fordi `else`-grenen ikke blir testet. For full branch coverage må vi også teste `is_even(3)`.

9.13.3 Function/method coverage

Definisjon: Måler om *alle funksjoner/metoder* i programmet blir kalt under testing.

Eksempel:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Test:

```
def test_add():
    assert add(2, 3) == 5
```

subtract() blir ikke kalt, altså ikke dekket.

For full function coverage må vi også teste `subtract()`.

9.13.4 Condition coverage

Definisjon: Måler om *alle boolske uttrykk* evalueres både til `True` og `False`.

Eksempel:

```
def is_valid(x):
```

```
    return x > 0 and x < 10
```

Test:

```
def test_valid():
    assert is_valid(5) == True
```

Her evalueres x > 0 og x < 10 begge til True.

For full condition coverage, må vi teste:

- x <= 0 → False and True
- x >= 10 → True and False
- x = 5 → True and True

9.13.5Installere coverage

For å installere coverage, kjør denne kommandoen i terminalvinduet:

```
pip install coverage
```

9.13.6 Eksempelkode og tester

```
# file: sc_09_05_coverage_examples.py
def greet(name):
    return f"Hello, {name}!"

def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

```
def is_valid(x):
    return x > 0 and x < 10
```

Tester:

```
# file: test_sc_09_05_coverage_examples.py
import unittest
from sc_09_05_coverage_examples import greet, is_even, add,
subtract, is_valid

class TestCoverageExamples(unittest.TestCase):

    # Line coverage: tester at linjen i greet() blir kjørt
    def test_greet(self):
        self.assertEqual(greet("Alice"), "Hello, Alice!")

    # Branch coverage: tester både if og else i is_even()
    def test_is_even_true(self):
        self.assertTrue(is_even(4))

    def test_is_even_false(self):
        self.assertFalse(is_even(3))

    # Function/method coverage: tester add(), men ikke
subtract()
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

    # Condition coverage: tester is_valid() med ulike
sannhetsverdier
    def test_is_valid_true(self):
        self.assertTrue(is_valid(5))  # True and True

    def test_is_valid_false_low(self):
        self.assertFalse(is_valid(-1))  # False and True

    def test_is_valid_false_high(self):
        self.assertFalse(is_valid(15))  # True and False

if __name__ == "__main__":
    unittest.main()
```

9.13.7 Kjøre coverage

(I vår eksempelkode er det bare test av metoden `subtract` som mangler.)

For å kjøre coverage på fila med testene, gå til katalogen der testfila ligger og skriv:

```
coverage run test_sc_09_05_coverage_examples.py
```

For å få en rapport, skriv:

```
coverage report
```

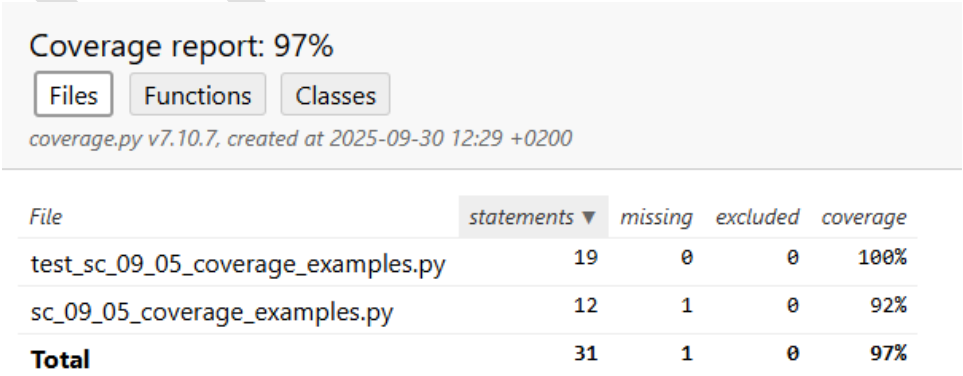
..som vil gi denne utskriften:

```
Name                                          Stmts   Miss  Cover
-----------------------------------------------------------------
sc_09_05_coverage_examples.py                    12      1    92%
test_sc_09_05_coverage_examples.py               19      0   100%
-----------------------------------------------------------------
TOTAL                                            31      1    97%
```

Kommandoen

coverage html

..vil lage en katalog `htmlcov` på den katalogen vi står på. Her ligger rapporter som viser resultatet, her fila index.html:



Filnavnene og knappene er lenker som viser nye html filer med detaljer om manglende coverage.

For å kjøre coverage på *alle* testfiler i en katalog, kjør denne kommandoen:

```
coverage run -m unittest discover
```

9.14 Ting vi ikke går i dybden på

I dette kapittelet har vi fokusert på grunnleggende testing med unittest, og hvordan man skriver og organiserer enkle tester. Det finnes imidlertid flere begreper og funksjoner i testverktøyet som er nyttige å kjenne til, selv om vi ikke går i detalj på dem her.

9.14.1 Testklasser

Alle testene vi har skrevet ligger i en **testklasse** som arver fra unittest.TestCase. Dette er en struktur som gjør det mulig å samle flere testmetoder som tester samme funksjon eller modul.

Eksempel:

```
import unittest
class TestCalculate(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(calculate(2, 3, "add"), 5)
    def test_subtraction(self):
        self.assertEqual(calculate(5, 2, "sub"), 3)
```

Du kan gruppere andre tester som høres logisk sammen i flere (andre) testklasser.

9.14.2 Test suite

En **test suite** er en samling av tester som kan kjøres samlet. Dette brukes ofte når man ønsker å gruppere tester på en mer kontrollert måte enn det `unittest.main()` gir.

Eksempel (som vi ikke går videre med):

```
suite = unittest.TestSuite()
suite.addTest(TestCalculate('test_addition'))
suite.addTest(TestCalculate('test_subtraction'))
unittest.TextTestRunner().run(suite)
```

I større prosjekter kan test suites brukes til å kjøre utvalgte tester, eller til å kombinere tester fra flere filer.

9.14.3Test discovery

Test discovery betyr at unittest automatisk finner og kjører testfiler og testmetoder, så lenge du følger visse navnekonvensjoner:

- Filnavn må starte med test_
- Testklasser må arve fra unittest.TestCase
- Testmetoder må starte med test_

Du kan starte test discovery fra terminalen:

```
python -m unittest discover
```

Denne kommandoen:

- Søker etter testfiler i **nåværende katalog** (og underkataloger, med mindre du spesifiserer noe annet).
- Kjører alle testmetoder som følger konvensjonene.