testsinpython

January 27, 2025

1 Test i Python med unittest og doctests - Af Henrik Sterner (henrik.sterner@gmail.com)

I Python har vi mulighed for at lave flere forskellige tests. Vi kan bruge kommandoen `assert`, som tjekker om en betingelse er sand. Vi kan lave unittests, der er en indbygget testfunktion i Python, som tester om en funktion virker som den skal. Vi kan også lave doctests, der er en test, der er skrevet i docstrings-formatet. Vi kan også lave en test, der er skrevet i en fil, og som vi kan køre fra terminalen.

Vi indleder med et afsnit om test, de forskellige typer af test og hvorfor det er så vigtigt at teste. Efterfølgende ser vi på hvorledes man kan lave forskellige former for tests i Python ved brug af assertions, unittests og doctests.

1.1 Test - hvorfor og hvordan?

Test er helt central i al softwareudvikling og programmering. I praksis handler det om at sikre, at ens system eller program rent faktisk fungerer efter hensigten og overholder de krav som man har opstillet.

Ikke desto mindre har det historisk set været nedprioriteret og ofte betragtet som en adskilt del eller en eftertanke i udviklingsprocessen.

Stadig den dag i dag er det ikke unormalt, at man først helt til sidst i produktionsfasen tester produktet for fejl med det resultat, at pga et meget lille vindue til at teste systemet, ender man man med sende produkter på markedet med mange fejl, fordi testerne ikke har haft tid nok til at fange alle fejl. Et eksempel herpå er indenfor spiludviklingsbranchen hvor det nærmest er blevet tradition, at man på release date eller kort bagefter frigiver store patches, som retter mange fejl. Det koster måske primært noget frustation hos spillerne mere end, at det går udover salgene. Et andet eksempel hvor det ikke er tilfældet er bilindustrien, som i de senere år i stadig højere grad, har måtte tilbagekalde biler i tusindvis på grund af en softwaredefekt med airbag eller lignende. Det koster producenterne millioner eller milliarder af kroner ved ikke at have gode teknikker for testning og skaber samtidig utilfredse kunder, der kan forstærke et i forvejen dårligt ry.

Ved at indføre tidlig og kontinuerlig testning kan man også langt bedre justere ikke kun på ukorrekt eller fejlbehæftet adfærd og sikkerhedsproblemer, men også justere systemets overordnede arkitektur. Sidstnævnte kan blive svære jo længere vi når i processen.

1.1.1 Forskellige typer af test

Der findes en lang række forskellige former for afhængig af hvad formålet er. Herunder gennemgås nogle af de centrale former for test:

- Accept testning: Virker det overordnede system rent faktisk efter hensigten
- Integration testning: Kan de enkelte komponenter i systemet og arkitekturen rent fakiske operere og kommunikere som ønsket
- Enhedstestning: Virker de enkelt enheder såsom funktioner og klasser efter hensigten.
- Performance test: Hvordan performer systemet under forskellig brug
- Bagudkompatibilitets test: Fungerer opdateret systemer med tidligere versioner
- Stress test: Hvor meget kan systemet klare inden det stopper med at fungere.
- Brugergrænseflade test: Hvordan tager brugeren af systemet i mod systemet og er der en fornuftig interaktion mellem bruger og system

I praksis kan man implementere test i udviklingsprocessen på en lang række måder. Det kunne eksempelvis være:

- Manuel testning: En person tester systemet manuelt
- Automatiseret testnin: En person skriver test, som kan køres automatisk
- Kontinuerlig testning: Testning sker løbende og automatisk. Kan være både manuel og automatiseret.

I det følgende vil vi gennemgå disse former for test og hvordan de kan implementeres i praksis.

1.1.2 Manuel testning

Enhver programmør eller team af programmører udfører manuel testning på regulær basis. Formålet med manuel testing, som normalt udføres af en eller flere personer, er at lokalisere fejl og mangler i koden/produktet. Man kan inddele denne form for testning i tre dele:

Sort boks testning Her testes produktet af software testere, som ikke nødvendigvis kender den interne struktur af koden eller programmet. Fokus er på at teste funktionaliteten af systemet. Kan systemet det som der forventes. Typisk er denne form for test baseret på en kravspecifikation, hvor der testes for hvorvidt de enkelte krav er opfyldt. Typisk stilles der ikke krav til hvorvidt testeren kan kode eller ej. Forestiller man sig eksempelvis, at et login-system med et brugernavn og password skal testes, så vil man her typisk kunne teste hvorvidt login fungerer som det skal. Dvs. logger systemet ind når man skriver korrekte login oplysninger og omvendt giver det korrekte beskeder, hvis man bruger forkert login? Hvis loginsystemet er blevet videreudviklet kan man også her teste, hvorvidt systemet stadig fungerer som det skal.

Hvid boks testning Hvid boks testning foretages typisk af softwareudviklere hvor testeren har viden om programmets interne struktur. Man tester her programmet på kodeniveau, hvor de enkelte funktioner og programstumper logisk testes for hvorvidt de gør som de skal. Forestiller man sig igen et login-system vil fokus i hvid boks testning være på om flowet i koden er korrekt. Et login system består typisk af et opslag i et sæt af brugere og deres respektive passwords (løkke eller forespørgsel i database), et tjek om det respektive brugernavn og password stemmer overens med det indtastede (betinget udførsel) og at man videresendes på korrekt vis. Fokus er mao på at teste hvorvidt opslaget og den betinget udførsel fungerer korrekt.

Grå boks testning Grå boks testning kombinerer hvid boks og sort boks testning i den forstand, at man her antager, at testeren af systemet har partiel viden om systemet. Man ser denne form testning anvendt i bl.a. webapplikationer, hvor testeren kan tilpasse og justere html-koden. F.eks. kunne man i casen med et login-system lade testeren redigere i html koden, hvis linket til systemet ikke fungerer optimalt. Herved kan udviklerne i højere grad koncentere sig om at teste den mere komplekse del af systemet.

1.1.3 Automatiseret testning

Til forskel for manuel testning, hvor et menneske står for at teste systemet, så gør man i automatiseret testning brug af programmer til at teste ens produkt. Ulempen ved manuel testning er, at det ofte kræver mange ressourcer både i form af tid og mennesker, hvis hele workflowet i et system skal testes. Og det koster mange penge. Her kommer automatiseret test til rådighed, fordi det kan udføres uden indblanding fra mennesker og man kan køre det når som helst. Selvom natten. Man kan genbruge og gentage test lige så ofte, som man finder det nødvendigt. På den måde faciliteres et mere præcist billede systemets tilstand i fht fejl og mangler. Ydermere kan manuel testning være en ret kedelig proces i længden, og herved opstår muligheden for fejl i højere grad end hvis et system står for processen. Eksempelvis kan man forestille sig et system, som skal understøtte flere sprog. Her vil det hurtigt være en lang, kedelig og næsten triviel process for et menneske at teste hvorvidt systemet opfører sig ens og fremstår korrekt i de forskellige sprog.

Typisk bruges automatiseret testning derfor når testning skal gentages ofte, og at det er kedeligt, svært og tidskrævende at gøre manuelt, men også når testningen involverer en kritisk del af systemet. Kritiske systemer kan være alt fra sundhedsplatforme, banksystemer til systemer, der håndterer hardware eller robotter. Et eksempel på hvor man i den grad kunne have gjort brug af automatiseret test var NASAs og firmaet Lockheed Martins satelit, som blev sendt afsted i 1998 for at gå i kredsløb omkring Mars. Den kostede flere milliarder at få sendt afsted og over et år i rejsetid før satelitten etablerede kredsløbet omkring Mars. Få minutter efter bragede satelitten ind i Mars. Det viste sig efterfølgende, at fejlen lå i, at to af systemerne udviklet af hhv Nasa og Lockheed Martin brugte forskellige afsandsmetrikker.

1.2 Assertions: En simpel test

En assertion er en simpel test, der tjekker om en betingelse er sand. Hvis betingelsen er falsk, vil programmet stoppe og give en AssertionError.

```
assert 1 == 1
assert 1 == 2
```

Vi kan også skrive en besked, der bliver vist, hvis betingelsen er falsk.

```
assert 1 == 2, "1 er ikke lig med 2"
```

Vi kan bruge assertions til at teste om en funktion virker som den skal.

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

assert add(1, 2) == 3
assert add(1, 2) == 4
```

Vi kan også indkapsle vores assertions i en funktion, så vi kan teste flere ting på en gang.

```python
def test_add():
    assert add(1, 2) == 3
    assert add(1, 2) == 4

test_add()
```

Vi kan sågar lave en test, der tester om en exception bliver kastet.

```python
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("division by zero")
    return a / b

def test_divide():
    assert divide(1, 2) == 0.5
    assert divide(1, 0) == 0

test_divide()
```

Her vil testen fejle, fordi vi forventer at der bliver kastet en exception. Der findes en lang række forskellige exceptions, som vi kan teste for. Herunder en liste over de mest almindelige exceptions:

- `ZeroDivisionError`: Kastes når vi forsøger at dividere med 0.
- `ValueError`: Kastes når en funktion modtager en ugyldig værdi.
- `TypeError`: Kastes når en funktion modtager en ugyldig type.
- `IndexError`: Kastes når vi forsøger at tilgå et element i en liste, der ikke eksisterer.
- `KeyError`: Kastes når vi forsøger at tilgå en nøgle i et dictionary, der ikke eksisterer.
- `AssertionError`: Kastes når en assertion fejler.

Generelt set er det en god idé at teste for exceptions, da det kan hjælpe os med at finde fejl i vores kode. Vi kan gøre det ved at bruge `try` og `except` blokke. Herunder et eksempel:

```python
def test_divide():
    assert divide(1, 2) == 0.5
    try:
        divide(1, 0)
    except ZeroDivisionError:
        pass
    else:
        assert False
```

Her tester vi om der bliver kastet en `ZeroDivisionError`, når vi forsøger at dividere med 0. Hvis der ikke bliver kastet en exception, vil testen fejle.

Ved at bruge løkker og en liste af tests, kan vi teste flere funktioner på en gang.

```python
def test_all():
    tests = [test_add, test_divide]
    for test in tests:
        test()

test_all()
```

Bemærk at vi ikke får nogen output, hvis alle tests er succesfulde. Hvis en test fejler, vil vi få en AssertionError.

1.2.1 Opgaver til assertions

1. Skriv en funktion `multiply(a, b)`, der tager to tal som input og returnerer produktet af de to tal. Test funktionen med assertions. Undersøg hvad der sker, hvis du forsøger at multiplicere to strings og tag højde for det med en exception.
2. Skriv en funktion `subtract(a, b)`, der tager to tal som input og returnerer differensen af de to tal. Test funktionen med assertions.
3. Skriv en funktion `power(a, b)`, der tager to tal som input og returnerer a opløftet i b. Test funktionen med assertions.
4. Skriv en funktion `factorial(n)`, der tager et tal som input og returnerer n!. Test funktionen med assertions. Tag højde for at n kan være negativ og returner en passende exception.
5. Skriv en funktion der summer alle tal i en liste. Test funktionen med assertions. Undersøg hvad der sker, hvis listen indeholder strings og tag højde for det med en exception.
6. Skriv en funktion der tager et naturtal n som input og returnerer det n'te Fibonacci-tal. Test funktionen med assertions.
7. Skriv en funktion der tager en liste som input og returnerer det største tal i listen. Test funktionen med assertions.
8. Skriv en funktion der undersøger om en liste er sorteret. Test funktionen med assertions.
9. Skriv en funktion der tager en liste som input og returnerer en ny liste med de unikke elementer. Test funktionen med assertions.
10. Skriv en funktion der tager en liste som input og returnerer en ny liste med de elementer der optræder mere end én gang. Test funktionen med assertions. Undersøg hvad der sker, hvis listen indeholder strings og tag højde for det med en exception.

1.3 Unittests i Python

En unittest er en test, der er skrevet i Python, som tester om en funktion virker som den skal. En unittest er en del af Python standardbibliotek, og vi kan derfor bruge den uden at skulle installere noget. Fordelene ved unittests er, at de er nemme at skrive, og at de er nemme at køre. Ulempen er, at de kan være svære at læse, og at de kan være svære at vedligeholde.

For at lave en unittest skal vi importere unittest modulet, og så skal vi lave en klasse, der nedarver fra unittest.TestCase. I denne klasse skal vi lave en række metoder, der starter med test_. Disse metoder er de tests, der skal køres.

Vi starter med at importere unittest modulet:

```
import unittest
```

Derefter laver vi en klasse, der nedarver fra unittest.TestCase:

```
class TestStringMethods(unittest.TestCase):
```

I denne klasse laver vi en række metoder, der starter med test_. Disse metoder er de tests, der skal køres. I disse metoder bruger vi assert metoden til at teste om en funktion virker som den skal. Hvis assert metoden fejler, så vil testen fejle.

1.4 Eksempel på unittest: En funktion, der lægger to tal sammen

Vi starter med et relativt simpelt eksempel. Vi vil lave en funktion, der lægger to tal sammen. Vi vil teste om funktionen virker som den skal ved at lave en unittest:

[7]:
```
import unittest
def add(a, b):
    return a + b

class TestAdd(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(0, 0), 0)
        self.assertEqual(add(-1, -1), -2)
        self.assertEqual(add(1, -1), 0)
```

Her ser vi, at vi har lavet en klasse, der nedarver fra unittest.TestCase. I denne klasse har vi lavet en metode, der starter med test_. I denne metode bruger vi assertEqual metoden til at teste om funktionen add virker som den skal. Hvis assertEqual metoden fejler, så vil testen fejle.

Bemærk assertEqual metoden. Den tager to argumenter. Det første argument er det forventede resultat, og det andet argument er det faktiske resultat. Hvis de to argumenter er ens, så vil testen lykkes. Hvis de to argumenter ikke er ens, så vil testen fejle.

Vi kan køre testen i jupyter ved at skrive:

```
unittest.main(argv=[''], exit=False)
```

unittest.main() metoden kører alle tests i klassen. argv=[’’] argumentet fortæller unittest.main() metoden, at vi ikke vil have nogen argumenter. exit=False argumentet fortæller unittest.main() metoden, at vi ikke vil have unittest.main() metoden til at afslutte programmet.

Vi kan også køre testen i terminalen ved at skrive:

```
python -m unittest test_add.py
```

Herunder prøver vi at køre testen i jupyter:

[8]:
```
unittest.main(argv=[''], exit=False)
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

[8]:
```
<unittest.main.TestProgram at 0x1e4c5d83610>
```

1.5 Eksempel på unittest: Tælle alle ord i en tekst

Vi vil nu lave en lidt mere kompleks test. Vi vil lave en funktion, der finder de unikke ord i en tekst, og tæller hvor mange gange hvert ord optræder. Vi vil teste om funktionen virker som den

skal ved at lave en unittest:

[9]:
```python
import unittest

def count_unique_words(text):
    words = text.split()
    unique_words = set(words)
    word_count = {}
    for word in unique_words:
        word_count[word] = words.count(word)
    return word_count

class TestStringMethods(unittest.TestCase):
    def test_count_unique_words(self):
        text = "hello world hello"
        expected = {"hello": 2, "world": 1}
        self.assertEqual(count_unique_words(text), expected)

    def test_count_unique_words_empty(self):
        text = ""
        expected = {}
        self.assertEqual(count_unique_words(text), expected)



unittest.main(argv=[''], exit=False)
```

```
..
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

[9]:
```
<unittest.main.TestProgram at 0x1e4c6fc65d0>
```

1.6 Opgaver til unittests

1. Skriv en unittest til funktionen `multiply(a, b)`, der tager to tal som input og returnerer produktet af de to tal.
2. Skriv en funktion der afgør om et tal er et primtal. Skriv en unittest til funktionen.
3. Skriv en funktion der tager en liste som input og returnerer det største tal i listen. Skriv en unittest til funktionen.
4. Skriv en funktion der tager en liste som input og returnerer en ny liste med de unikke elementer. Skriv en unittest til funktionen.
5. Skriv en funktion der tager en liste som input og returnerer en ny liste med de elementer der optræder mere end én gang. Skriv en unittest til funktionen.
6. Skriv en funktion der tager en liste som input og returnerer summen af alle tal
7. Skriv en funktion der tager en liste som input og returnerer gennemsnittet af alle tal
8. Skriv en funktion der tager en liste som input og returnerer medianen af alle tal

9. Skriv en funktion der tager en liste som input og returnerer det n'te Fibonacci-tal
10. Skriv en funktion der tager en liste som input og returnerer det største tal i listen

1.7 Docstrings og doctests

En docstring er en streng, der står øverst i en funktion, og som beskriver hvad funktionen gør. En docstring er en god måde at dokumentere sin kode på, og det gør det nemmere for andre at forstå hvad koden gør.

Herunder ser vi et eksempel på en docstring:

```
def add(a, b):
    """
    This function adds two numbers together.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of the two numbers.
    """
    return a + b
```

Når vi har skrevet en docstring, kan vi bruge doctests til at teste om funktionen virker som den skal. En doctest er en test, der er skrevet i docstrings-formatet:

```
def add(a, b):
    """
    This function adds two numbers together.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of the two numbers.

    >>> add(1, 2)
    3
    >>> add(1, 2)
    4
    """
    return a + b
```

Her bemærker vi, at vi har skrevet to tests i docstrings-formatet. Hvis vi kører funktionen med doctests, vil vi få en AssertionError, fordi den anden test fejler.

Vi kan køre doctests ved at skrive:

```
if __name__ == "__main__":
```

```
    import doctest
    doctest.testmod()
```

Herunder ser vi et eksempel på en funktion, der tager en liste som input og returnerer det største tal i listen:

```
def max_list(lst):
    """
    This function takes a list as input and returns the largest number in the list.

    Parameters:
    lst (list): The list of numbers.

    Returns:
    int: The largest number in the list.

    >>> max_list([1, 2, 3, 4, 5])
    5
    >>> max_list([5, 4, 3, 2, 1])
    5
    """
    return max(lst)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Herunder ser vi et eksempel på en funktion, der tager en liste som input og returnerer en ny liste med de unikke elementer:

```
def unique_list(lst):
    """
    This function takes a list as input and returns a new list with the unique elements.

    Parameters:
    lst (list): The list of numbers.

    Returns:
    list: A new list with the unique elements.

    >>> unique_list([1, 2, 3, 4, 5])
    [1, 2, 3, 4, 5]
    >>> unique_list([1, 1, 2, 2, 3, 3, 4, 4, 5, 5])
    [1, 2, 3, 4, 5]
    """
    return list(set(lst))

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

1.8 Opgaver til doctests

Skriv docstrings og doctests til følgende funktioner: 1. En funktion der tager en liste som input og returnerer summen af alle tal 2. En funktion der afgør om størrelsen af en liste er et lige tal 3. En funktion der afgør om en streng er et palindrom 4. En funktion der tager en liste som input og returnerer gennemsnittet af alle tal 5. En funktion der undersøger om en liste er sorteret 6. En funktion der tager en liste som input og returnerer en ny liste med de unikke elementer 7. En funktion der tager en liste som input og returnerer en ny liste med de elementer der optræder mere end én gang 8. En funktion der tager en liste som input og returnerer det n'te Fibonacci-tal 9. En funktion der tager en liste som input og returnerer det største tal i listen 10. En funktion der tager en liste som input og returnerer medianen af alle tal

1.9 Debugging

Når vi skriver tests, kan vi bruge dem til at finde fejl i vores kode. Hvis en test fejler, kan vi bruge testen til at finde ud af, hvad der er galt. Det er her debugging kommer ind i billedet.

Debug eller debugging er en teknik, der bruges til at finde og rette fejl i kode. Der findes mange forskellige værktøjer til debugging, herunder print statements, debuggeren og logfiler.

1.9.1 Print instruktioner

Print instruktioner er en simpel måde at finde fejl i kode på. Vi kan bruge print til at se, hvad der sker i koden, når vi køer den, og vi kan bruge dem til at finde ud af, hvad der er galt. Herunder et eksempel på hvordan vi kan bruge print til at finde fejl i kode. Funktionen herunder tager en liste som input og returnerer det største tal i listen:

```
def max_list(lst):
    max_num = lst[0]
    for num in lst:
        if num > max_num:
            max_num = num
            print(max_num) # printer løbende det største tal
    return max_num
```

Hvis vi kører koden, vil vi se, at print statementet printer det største tal i listen, hver gang vi finder et nyt større tal.

Ulempen ved print instruktioner er, at de kan være svære at fjerne, når vi har fundet fejlen. Derudover kan de også være svære at læse, hvis vi har mange print instruktioner i koden.

Opgaver til print instruktioner

1. Herunder et program, hvor der er to mindre fejl. Brug print instruktioner til at finde fejlen i koden:

```
# Programmet tager et naturligt tal som input og returnerer True, hvis tallet er et primtal, o
def isPrime(n):
    if n < 2:
        return False
    for i in range(2, n+1):
        if n / i == 0:
```

```
            return False
    return True
```

2. Herunder et program, hvor der er en mindre fejl. Brug print instruktioner til at finde fejlen i koden:

```
# Funktionen skal de første n tal
def sum_of_squares(n):
    total = 0
    for i in range(n):
        total += i * i
    return total
```

3. Herunder et program, hvor der er en mindre fejl. Brug print instruktioner til at finde fejlen i koden:

```
# Funktionen skal generere en liste med de første n Fibonacci-tal
def fibonacci_sequence(n):
    sequence = [0, 1]
    for i in range(2, n):
        next_value = sequence[i - 1] + sequence[i - 2]
        sequence.append(next_value)
    return sequence
```

1.9.2 Breakpoints i koden

Et breakpoint er et punkt i koden, hvor programmet stopper eller nærmere pauser, så vi kan undersøge, hvad der er galt. Det betyder, at vi kan aflæse værdierne af variablerne på et bestemt tidspunkt i koden. Vi kan bruge breakpoints til at finde fejl i kode, og vi kan bruge dem til at se, hvad der sker i koden, når vi kører den.

Langt de fleste IDE'er har indbygget debugger, som vi kan bruge til at sætte breakpoints i koden, men Python har også et indbygget debugger, som vi kan bruge til at sætte breakpoints i koden. Vi kan bruge `pdb` modulet til at sætte breakpoints i koden:

```
import pdb
```

Herunder et eksempel på hvordan vi kan bruge breakpoints til at finde fejl i kode. Funktionen herunder tager en liste som input og returnerer det største tal i listen:

```
import pdb

def max_list(lst):
    max_num = lst[0]
    for num in lst:
        if num > max_num:
            max_num = num
            pdb.set_trace() # sætter et breakpoint
    return max_num
```

Hvis vi kører koden, vil vi se, at programmet stopper ved breakpointet, og vi kan undersøge, hvad der er galt. Vi kan bruge `n` til at køre programmet videre, og vi kan bruge `q` til at afslutte

programmet.

Nogle af de mest almindelige kommandoer i `pdb` modulet er:

- `n`: Kører programmet videre.
- `q`: Afslutter programmet.
- `c`: Fortsætter programmet indtil næste breakpoint.
- `l`: Viser koden omkring breakpointet.
- `p`: Printer værdien af en variabel.
- `s`: Træder ind i en funktion.
- `r`: Kører programmet indtil funktionen er færdig.

Opaver til breakpoints

1. Herunder et program, hvor der er to mindre fejl. Brug breakpoints til at finde fejlen i koden:

```python
def sum_list(numbers):
    total = 0
    for num in numbers:
        total += num
    return total

# Test programmet
numbers = [1, 2, 3, 4, 5]
print(sum_list(numbers))  # Forventet output: 15
```

2. Herunder et program, hvor der er en mindre fejl. Brug breakpoints til at finde fejlen i koden:

```python
def count_occurrences(lst, element):
    count = 0
    for item in lst:
        if item == element:
            count += 1
    return count

# Test funktionen
lst = [1, 2, 3, 4, 2, 2, 5, 2]
element = 2
print(count_occurrences(lst, element))  # Forventet output: 4
```

3. Herunder et program, hvor der er en mindre fejl. Brug breakpoints til at finde fejlen i koden:

```python
def bubble_sort(numbers):
    n = len(numbers)
    for i in range(n):
        for j in range(0, n-i-1):
            if numbers[j] > numbers[j+1]:
                numbers[j], numbers[j+1] = numbers[j+1], numbers[j]
    return numbers

# Test funktionen
```

```
numbers = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(numbers))  # Forventet output: [11, 12, 22, 25, 34, 64, 90]
```

1.9.3 Logfiler

En logfil er en fil, der indeholder information om, hvad der sker i programmet, når vi kører det. Vi kan bruge logfiler til at finde fejl i kode, og vi kan bruge dem til at se, hvad der sker i koden, når vi kører den.

Vi kan bruge `logging` modulet til at lave logfiler i Python:

```
import logging
```

Herunder et eksempel på hvordan vi kan bruge logfiler til at finde fejl i kode. Funktionen herunder tager en liste som input og returnerer det største tal i listen:

```
import logging

def max_list(lst):
    logging.basicConfig(filename='example.log', level=logging.DEBUG)
    max_num = lst[0]
    for num in lst:
        if num > max_num:
            max_num = num
            logging.debug(f"max_num: {max_num}")
    return max_num
```

Hvis vi kører koden, vil vi se, at programmet skriver information til logfilen, og vi kan undersøge, hvad der er galt. Vi kan bruge `logging.debug()` metoden til at skrive information til logfilen, og vi kan bruge `logging.error()` metoden til at skrive fejlmeddelelser til logfilen.

Man kan også logge til konsollen ved at bruge `logging.StreamHandler()` metoden:

```
import logging

def max_list(lst):
    logging.basicConfig(level=logging.DEBUG)
    console = logging.StreamHandler()
    console.setLevel(logging.INFO)
    logging.getLogger('').addHandler(console)
    max_num = lst[0]
    for num in lst:
        if num > max_num:
            max_num = num
            logging.debug(f"max_num: {max_num}")
    return max_num
```

Opgaver til logfiler

1. Herunder et program, hvor du kan brug logfiler til at finde fejl i koden:

```
import logging


def calculate_average(numbers):
    logging.debug(f"Input numbers: {numbers}")
    total = sum(numbers)
    logging.debug(f"Total sum: {total}")
    count = len(numbers)
    logging.debug(f"Number of elements: {count}")
    average = total / count
    logging.debug(f"Calculated average: {average}")
    return average

# Test funktionen
numbers = [1, 2, 3, 4, 5]
print(calculate_average(numbers))  # Forventet output: 3.0
```

2. Herunder et program, hvor du kan brug logfiler til at finde fejl i koden. Funktionen tager to tal som input og returnerer største fælles divisor:

```
import logging

logging.basicConfig(level=logging.DEBUG)

def gcd(a, b):
    logging.debug(f"Starting GCD calculation for a={a}, b={b}")
    while b != 0:
        logging.debug(f"Inside loop: a={a}, b={b}")
        a, b = b, a % b
        logging.debug(f"Updated values: a={a}, b={b}")
    logging.debug(f"Final GCD: {a}")
    return a

# Test funktionen
print(gcd(60, 48))  # Forventet output: 12
print(gcd(48, 60))  # Forventet output: 12
print(gcd(-17, 23))  # Forventet output: 1
```