Unit Testen

linked/18663/52557

1 Wat is testen? Waarom is het belangrijk?

Leerdoelen

- Je begrijpt wat testen is en waarom we het doen.
- Je herkent verschillende soorten testen.
- Je kunt een eenvoudige functie handmatig testen.

Uitleg

Testen betekent controleren of je code het juiste doet. Als je iets verandert, wil je zeker weten dat het nog steeds werkt. Denk aan:

- Een rekenmachine die ineens verkeerde antwoorden geeft.
- Een webshop die verkeerde prijzen toont.

Daarom testen we onze code – om fouten te voorkomen en vertrouwen te krijgen in wat we gemaakt hebben.

Testsoorten

Er zijn verschillende soorten testen. Hieronder zie je een overzicht:

Testsoort	Wat test je?	Voorbeeld
Unit test	Één functie of klein onderdeel	Klopt `bereken_korting()`?
Integratietest	Samenwerking tussen onderdelen	Voegt `bereken_korting()` + `bereken_verzendkosten()` alles goed op?

Testsoort	Wat test je?	Voorbeeld
Systeemtest	Het hele programma	Werkt de volledige webshop van begin tot eind?
Acceptatietest	Voldoet het aan de wensen van de gebruiker?	Kan de klant een product kiezen en succesvol bestellen?

In deze lessenserie focussen we op **unit tests**: kleine, duidelijke tests waarmee je functies controleert.

□□ Opdracht 1 – winkelmandje.py

Maak een nieuw Python-bestand aan met de naam `winkelmandje.py` en doe het volgende:

1. Schrijf een functie `bereken_totaalprijs(prijzen)` die een lijst van prijzen optelt.
2. Voorbeeld: `bereken_totaalprijs([2.50, 4.00, 3.25])` moet `9.75` opleveren.
3. Print het resultaat van je functie met verschillende lijsten.
4. Probeer ook een lege lijst – wat gebeurt er?

□□ Reflectie

- Welk van de testsoorten vind jij het belangrijkste? Leg in eigen woorden uit waarom!
- Wat gebeurt er als je iets fout typt in je functie?
- Wat als je functie 3 keer voorkomt in een groter programma – hoe test je dat?
- Hoe zou je dit automatisch kunnen controleren?

□□ Inleveren

- Lever het bestand `winkelmandje.py` in (.py).
- Beantwoord de drie reflectievragen in een apart .txt of .pdf bestand.

2 Je eerste unit test schrijven

□□eerdoelen

- Je kunt een eenvoudige test schrijven met de module `unittest`.
- Je weet wat `assertEqual()` doet.
- Je voert je testbestand uit en controleert of het slaagt.

□□ Uitleg

In de vorige les heb je een functie gemaakt die de totaalprijs berekent van een lijst met bedragen. Nu ga je leren hoe je automatisch kunt testen of die functie goed werkt.

Hiervoor gebruik je de module `unittest`. Je schrijft testfuncties waarin je verwacht wat de uitkomst moet zijn. Als die klopt, krijg je een groene melding. Als die fout is, zie je precies waar het misgaat.

Voorbeeldtest:

```
import unittest


def bereken_totaalprijs(prijzen):
    totaal = 0
    for prijs in prijzen:
        totaal += prijs
    return totaal


class TestTotaalprijs(unittest.TestCase):
    def test_drie_bedragen(self):
        self.assertEqual(bereken_totaalprijs([2.50, 4.00, 3.25]), 9.75)

    def test_lege_lijst(self):
        self.assertEqual(bereken_totaalprijs([]), 0)

    def test_een_bedrag(self):
        self.assertEqual(bereken_totaalprijs([5.00]), 5.00)

    # def test_nog_een_bedrag(self):
    #     self.assertEqual(bereken_totaalprijs([5.00, 2.00]), 8.00)

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

Je voert het testbestand uit met:

```
python totaalprijs_test.py
```

Opdracht 1

Maak regel 9 en 10 actief. Voer het bestand opneuw uit. Wat zie? Kna je verklaren wat er gebeurt?

Opdracht 2 – totaalprijs_test.py

Je gaat nu je eigen functie `bereken_totaalprijs()` uit les 1 testen.

1. Maak een nieuw bestand aan: `totaalprijs_test.py`.
2. Kopieer je functie `bereken_totaalprijs(prijzen)` uit `winkelmandje.py`.
3. Schrijf een testklasse `TestTotaalprijs` waarin je de functie test met `unittest`.
4. Schrijf minstens drie tests:
   - Lijst met meerdere bedragen (bijv. `[2.50, 3.50]`) → controleer of totaal klopt.
   - Lege lijst → verwacht `0`.
   - Lijst met één bedrag → verwacht dat bedrag.
5. Voer je bestand uit in de terminal met `python totaalprijs_test.py`.

Reflectie

- Wat gebeurt er als een test slaagt? En als hij faalt?
- Wat zijn de voordelen van automatisch testen vergeleken met handmatig printen?
- Hoe weet je zeker dat je test goed geschreven is?

Inleveren

- Lever een .txt of .pdf-bestand in met het antwoord op de vraag uit Opdracht 1.
- Lever het bestand `totaalprijs_test.py` in (.py).

3 Testen met lijsten en tekst

Leerdoelen

- Je leert functies testen die werken met tekst en lijsten.
- Je gebruikt `assertTrue()` en `assertFalse()`.
- Je test of een bepaald product in een lijst voorkomt.

Uitleg

Veel programma's werken met lijsten van gegevens, zoals producten of gebruikers. Het is belangrijk dat je functies goed testen of iets voorkomt in een lijst, of hoeveel elementen erin zitten.

Stel, je hebt een lijst met producten:

```
producten = ["kaas", "melk", "brood"]
```

Dan kun je bijvoorbeeld een functie maken die kijkt of een product voorkomt:

```
def zoek_product(lijst, naam):
    product_gevonden = gezochte_product_naam in product_lijst


   return product_gevonden
```

En die functie testen met:

```
self.assertTrue(zoek_product(["melk", "brood"], "melk"))
self.assertFalse(zoek_product(["kaas", "brood"], "cola"))
```

Opdracht 1 – producten_test.py

1. Maak een nieuw bestand `producten_test.py`.
2. Schrijf een functie `zoek_product(lijst, naam)` die `True` geeft als het product in de lijst staat.
3. Maak een testklasse `TestProductZoeken`.
4. Schrijf drie tests:

- Zoek naar een bestaand product → `assertTrue`
- Zoek naar een niet-bestaand product → `assertFalse`
- Zoek in een lege lijst → `assertFalse`

5. Voer je testbestand uit met `python producten_test.py`.

Uitbreiding (optioneel)

Maak ook een functie `tel_producten(lijst)` die het aantal producten telt, en schrijf daar een test voor met `assertEqual`.

□□ Reflectie

- Wat is het verschil tussen `assertTrue` en `assertEqual`?
- Waarom is het handig om een lege lijst te testen?
- Hoe weet je zeker dat je lijst-functies in alle gevallen goed werken?

□□ Inleveren

- Lever het bestand `producten_test.py` in (.py).
- Lever een reflectiedocument in (.txt of .pdf) met jouw antwoorden op de vragen.

4 Fouten en `assertRaises`

□□eerdoelen

- Je weet wat een foutmelding (exception) is in Python.
- Je leert testen of een functie op de juiste manier een fout geeft.
- Je gebruikt `assertRaises` in je testcode.

□□ Uitleg

In Python krijg je een **foutmelding** (ofwel: *exception*) als er iets misgaat, zoals delen door nul of een ongeldig type.

Stel je schrijft deze functie:

```
def deel(a, b):
    return a / b
```

Als je `deel(10, 0)` probeert, krijg je deze fout:

```
ZeroDivisionError: division by zero
```

In een goede functie wil je zulke fouten **voorspelbaar** en **controleerbaar** maken, bijvoorbeeld zo:

```
def deel(a, b):
    if b == 0:
        raise ValueError("Delen door nul is niet toegestaan")
    return a / b
```

Je kunt dan testen of de fout netjes wordt afgehandeld, met:

```
with self.assertRaises(ValueError):
    deel(10, 0)
```

□□ Opdracht 1 – deel_test.py

1. Maak een nieuw bestand `deel_test.py`.
2. Schrijf een functie `deel(a, b)` die:
   - Een `ValueError` geeft als `b == 0`
   - Anders de uitkomst van `a / b` teruggeeft
3. Maak een testklasse `TestDelen` met drie tests:
   - Deel 10 door 2 → verwacht `5.0`
   - Deel 9 door 3 → verwacht `3.0`
   - Deel door 0 → verwacht `ValueError` met `assertRaises`
4. Voer je tests uit met `python deel_test.py`.

Reflectie

- Waarom is het belangrijk om ook foutgevallen te testen?
- Wat is het verschil tussen een fout en een mislukte test?
- Hoe weet je of je functie veilig omgaat met verkeerde input?

Inleveren

- Lever het bestand `deel_test.py` in (.py).
- Lever een .txt of .pdf-bestand in met antwoorden op de reflectievragen.

5 Testen van klassen met `setUp()`

Leerdoelen

- Je weet wat een klasse is in Python.
- Je leert hoe je een klasse test met `unittest`.
- Je gebruikt `setUp()` om herhaling in je testcode te voorkomen.

Uitleg

In grotere programma's werk je vaak met **klassen**. Die bestaan uit eigenschappen (zoals naam en prijs) en methodes (zoals korting berekenen).

Bijvoorbeeld een klasse `Product`:

```
class Product:
    def __init__(self, naam, prijs):
        self.naam = naam
        self.prijs = prijs

    def prijs_incl_btw(self):
        return self.prijs * 1.21
```

Als je zo'n klasse vaak moet testen, wil je niet steeds opnieuw een object aanmaken in elke test. Daarom gebruik je `setUp()`:

```
class TestProduct(unittest.TestCase):
    def setUp(self):
        self.product = Product("muis", 10.00)

    def test_naam(self):
        self.assertEqual(self.product.naam, "muis")

    def test_prijs(self):
        self.assertEqual(self.product.prijs, 10.00)

    def test_prijs_incl_btw(self):
        self.assertAlmostEqual(self.product.prijs_incl_btw(), 12.10)
```

□□ Opdracht 1 – product_test.py

1. Maak een nieuw bestand `product_test.py`.
2. Schrijf een klasse `Product` met:
   - Eigenschappen `naam` en `prijs`
   - Methode `prijs_incl_btw()` die 21% btw toevoegt
3. Maak een testklasse `TestProduct` waarin je in `setUp()` een `Product` aanmaakt.
4. Voeg minimaal drie tests toe:
   - Test of de naam klopt
   - Test of de prijs klopt
   - Test of de prijs inclusief btw ongeveer klopt (gebruik `assertAlmostEqual`)

□□ Reflectie

- Wat is het voordeel van `setUp()` gebruiken?
- Wanneer gebruik je `assertAlmostEqual` in plaats van `assertEqual`?
- Hoe zou je meerdere producten kunnen testen zonder dubbel werk?

Inleveren

- Lever het bestand `product_test.py` in (.py).
- Lever een reflectiedocument in (.txt of .pdf) met jouw antwoorden op de vragen.

6 Edge cases en grenswaarden

Leerdoelen

- Je weet wat een edge case is.
- Je test functies met bijzondere of extreme invoer.
- Je denkt vooraf na over mogelijke problemen in je functie.

Uitleg

Een **edge case** is een grensgeval of een extreme situatie waarin je functie getest wordt. Zulke gevallen zorgen vaak voor fouten als je er niet vooraf aan denkt.

Voorbeelden van edge cases:

- Een lege lijst
- De waarde `0`
- Negatieve getallen
- Extreem grote of kleine getallen

Voorbeeld: verzendkosten berekenen

Stel, je schrijft een functie die verzendkosten berekent op basis van het aantal producten:

```
def bereken_verzendkosten(aantal):
    if aantal <= 0:
        raise ValueError("Aantal moet positief zijn")
    elif aantal <= 3:
        return 2.50
```

```
    elif aantal <= 10:
        return 5.00
    else:
        return 0.00
```

Je kunt dan tests schrijven voor grenswaarden zoals 3, 4, 10 en 11.

□□ Opdracht 1 – verzendkosten_test.py

1. Maak een nieuw bestand `verzendkosten_test.py`.
2. Schrijf een functie `bereken_verzendkosten(aantal)` met de volgende regels:
   - 1 t/m 3 producten → €2,50
   - 4 t/m 10 producten → €5,00
   - Meer dan 10 producten → gratis
   - 0 of minder → `ValueError`
3. Schrijf een testklasse met tests voor:
   - `bereken_verzendkosten(1)` → €2,50
   - `bereken_verzendkosten(3)` → €2,50
   - `bereken_verzendkosten(4)` → €5,00
   - `bereken_verzendkosten(10)` → €5,00
   - `bereken_verzendkosten(11)` → €0,00
   - `bereken_verzendkosten(0)` → foutmelding (`assertRaises`)

□□ Reflectie

- Waarom is het belangrijk om grenswaarden te testen?
- Wat zou er gebeuren als je `aantal == 0` niet test?
- Ken je een voorbeeld uit de praktijk waar een fout ontstond door een vergeten randgeval?

□□ Inleveren

- Lever het bestand `verzendkosten_test.py` in (.py).
- Lever een reflectiedocument in (.txt of .pdf) met je antwoorden.

7 Eindopdracht – Testen in de praktijk

Leerdoelen

- Je past toe wat je geleerd hebt over unit testen.
- Je denkt zelfstandig na over wat belangrijk is om te testen.
- Je levert een werkend testbestand in met meerdere tests.

Uitleg

In deze les werk je aan een kleine testset voor een programma naar keuze. Je kiest één van de twee scenario's hieronder. Je schrijft zelf de functies én de bijbehorende `unittest`-tests.

Scenario 1 – Webshop-functies

Je maakt een verzameling functies zoals:

- `bereken_totaalprijs(prijzen)`
- `bereken_korting(prijs, procent)`
- `bereken_verzendkosten(aantal)`

Je schrijft voor elk van deze functies minstens 2 tests, waaronder minstens 1 edge case per functie.

Scenario 2 – Quiz-controlefunctie

Je schrijft een functie `controleer_antwoord(gegeven, verwacht)` die controleert of het antwoord juist is (hoofdletterongevoelig en spaties tellen niet mee).

Voorbeeld:

```
controleer_antwoord(" Python ", "python")  → True
```

Schrijf meerdere tests voor deze functie:

- Juist antwoord met extra spaties
- Hoofdletters door elkaar
- Verkeerd antwoord → moet `False` geven

□□ Opdracht – Kies en test

1. Kies scenario 1 (webshop) of scenario 2 (quiz).
2. Maak een nieuw testbestand (bijvoorbeeld `webshop_test.py` of `quiz_test.py`).
3. Schrijf de functies zoals hierboven beschreven.
4. Maak een testklasse en voeg minimaal 6 goed werkende tests toe (minstens 2 per functie).
5. Zorg dat je ook minstens één foutmelding test met `assertRaises`.

□□ Reflectie

- Welke test vond je het lastigst om te schrijven? Waarom?
- Wat is volgens jou een goede test? En wat niet?
- Hoe zou je deze testset uitbreiden als je meer tijd had?

□□ Inleveren

- Lever je testbestand in (.py).
- Lever een reflectieverslag in (.txt of .pdf).
- Als je extra functionaliteit of creatieve toevoegingen hebt gemaakt: geef dat kort aan in je verslag.

Revision #10
Created 6 June 2025 13:00:36 by Max
Updated 14 June 2025 20:38:29 by Max