Algemeen

Hergebruik van bestaande functionaliteit

Functies zijn controlestructuren die handig zijn bij het vermijden van **codeduplicatie**. Codeduplicatie is het fenomeen waarbij hetzelfde of een gelijkaardig stuk code op twee of meer plaatsen in je broncode voorkomt. Het is altijd een goed idee om codeduplicatie te vermijden.

Hou daarom altijd rekening met het feit dat je vanuit een functie ook andere functies kunt aanroepen. Soms staat het uitdrukkelijk vermeld in de opgave dat je bij de implementatie van een functie, een andere functie (die je eerder al geïmplementeerd hebt) moet hergebruiken. Maar ook als een dergelijke uitdrukkelijke vermelding niet in de opgave staat, blijft het de bedoeling dat je op zoek gaat om zoveel mogelijk bestaande functionaliteit (die je bijvoorbeeld in andere functies hebt geïmplementeerd) probeert te hergebruiken bij het implementeren van je functies.

Stel bijvoorbeeld dat je gevraagd wordt om twee functies te implementeren: `maxsom` en `minverschil`. Aan de eerste functie `maxsom` moeten drie argumenten doorgegeven worden, en moet de functie een Booleaanse waarde (`bool`) teruggeven die uitdrukt of de som van de eerste twee argumenten al dan niet kleiner is dan de waarde van het derde argument. Aan de tweede functie `minverschil` moeten drie argumenten doorgegeven worden, en moet de functie een Booleaanse waarde (`bool`) teruggeven die uitdrukt of de absolute waarde van het verschil van de eerste twee argumenten al dan niet groter is dan de waarde van het derde argument. Deze twee functies kunnen op de volgende manier geïmplementeerd worden.

```
def maxsom(x, y, a):
    return x + y < a

def minverschil(x, y, b):
    return abs(x - y) > b
```

Als je nu bovendien ook nog een derde functie `minmax` moet schrijven waaraan vier argumenten moeten doorgegeven worden, en die moet nagaan of de som van de eerste twee argumenten kleiner is dan het derde argumenten, en de absolute waarde van het verschil van de eerste twee argumenten groter is dan de waarde van het vierde argument, dan zou je deze functie op de volgende manier kunnen implementeren.

```
def minmax(x, y, a, b):
    return x + y < a and abs(x - y) > b
```

Hiermee ben je natuurlijk het *warm water* opnieuw aan het uitvinden. In dit geval is het impliciet de bedoeling dat je de functie `minmax` kunt implementeren door hergebruik te maken van de implementaties van de functies `maxsom` en `minverschil`. Op die manier kan je de implementatie van de functie eenvoudigweg als volgt neerschrijven.

```
def minmax(x, y, a, b):
    return maxsom(x, y, a) and minverschil(x, y, b)
```

Als je nu een wijziging zou aanbrengen aan je implementatie van de functie `maxsom` (omdat je bijvoorbeeld een efficiëntere oplossingsmethode gevonden hebt, of omdat er een *bug* in de implementatie bleek te zitten), dan moet je dit maar op één plaats aanpassen, en niet op twee plaatsen als je de code ook had gekopieerd in de functie `minmax`.

Itereren over de posities en de elementen van een collectie

De ingebouwde functie `enumerate` kan gebruikt worden om een iterator op te vragen van collecties (*iterables*: objecten van gegevenstypes zoals `str`, `list`, `tuple`, bestanden, ...), die telkens zowel de positie en het volgende element uit de collectie teruggeeft. Onderstaand voorbeeld geeft bijvoorbeeld aan hoe je voor een string tegelijkertijd de posities en de karakters op die posities kunt overlopen.

```
>>> for index, karakter in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'karakter: {karakter}')
...
index: 0
karakter: a
index: 1
karakter: b
index: 2
karakter: c
```

Je kan dit bijvoorbeeld gebruiken als je synchroon de karakters van twee strings wil overlopen.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for index, karakter in enumerate(eerste):
...    print(f'{karakter}-{tweede[index]}')
...
a-d
b-e
c-f
```

In dat geval is het echter aangewezen om gebruik te maken van de ingebouwde functie `zip`, die net haar bestaansrecht haalt uit het feit dat ze een iterator teruggeeft voor twee of meer collecties.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for karakter1, karakter2 in zip(eerste, tweede):
...    print(f'{karakter1}-{karakter2}')
...
a-d
b-e
c-f
```

Strings opsplitsen in verschillende delen

Om een string op te splitsen in verschillende delen, is het soms handig om gebruik te maken van de stringmethode `split`. Aan deze methode moet je een argument doorgeven, dat aangeeft welke reeks opeenvolgende karakters moet gebruikt worden om de string (waarop je de methode aanroept) in stukken te breken. Deze reeks opeenvolgende karakters wordt vaak *het scheidingsteken* genoemd.

```
>>> string = 'a-b-c-d'
>>> string.split('-')
['a', 'b', 'c', 'd']
```

Standaard wordt de string op alle plaatsen gesplitst waar de reeks opeenvolgende karakters voorkomt, en worden alle delen waarin de string werd opgesplitst in een lijst gestopt, die door de methode wordt teruggegeven. Als je geen argument meegeeft aan de methode, dan splitst die standaard de string op waar er een reeks opeenvolgende witruimtekarakters voorkomt (spaties, tabs, newlines, ...). De stringmethode heeft ook nog een optioneel argument dat je kan gebruiken om aan te geven in hoeveel stukken de string maximaal mag gesplitst worden.

Omdat de stringmethode `split` een lijst teruggeeft, kan je ook makkelijk rechtstreeks over deze lijst itereren met een `for`-lus.

```
>>> string = 'a-b-c-d'
>>> for element in string.split('-'):
```

```
...     print(element)
...
a
b
c
d
```

Functies/methoden: return vs print

Bij opgaven waarbij er gevraagd wordt om functies (vanaf reeks 05) of methoden (vanaf reeks 10) te implementeren, moet je heel goed opletten of er gevraagd wordt dat de functie/methode een resultaat moet **teruggeven**, dan wel dat de functie/methode een resultaat moet **uitschrijven**. Om een functie/methode een resultaat te laten teruggeven, maak je gebruik van een `return` statement. Om een functie een functie/methode een resultaat te laten uitschrijven naar *standaard uitvoer* (afgekort als `stdout`), maak je gebruik van de ingebouwde functie `print`.

In de opgave C-som (reeks 05) wordt er bijvoorbeeld gevraagd om een functie `csom` te schrijven die een resultaat moet **teruggeven**. Een mogelijke correcte implementatie van deze functie is

```python
def csom(getal):
    return getal % 9
```

waarbij een `return` statement gebruikt wordt om een berekende waarde te laten teruggeven door de functie. Stel dat we in plaats van een waarde terug te geven, verkeerdelijk gebruik zouden maken van een `print` statement om de berekende waarde uit te schrijven, en dat we de volgende oplossing zouden indien voor deze opgave.

```python
def csom(getal):
    print(getal % 9)
```

Deze oplossing zal een **verkeerd antwoord** opleveren, waarbij we de volgende informatie te zien krijgen op de feedbackpagina.

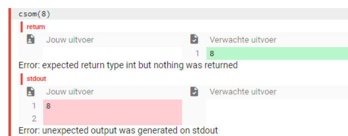

Figure 1: return vs print

Hierbij worden twee opmerkingen geformuleerd. De eerste opmerking geeft aan dat er verwacht werd dat de functie een integer waarde zou teruggeven (de waarde `8`), maar dat de functie niets heeft teruggegeven (of meer specifiek, dat de functie de waarde `None` heeft teruggegeven). Dit is de betekenis van de foutmelding

```
Error: expected return type int but nothing was returned
```

Bovendien komt er nog een tweede opmerking die aangeeft dat de functie iets heeft uitgeschreven (naar *standard output*, of kortweg *stdout*) terwijl dat helemaal niet verwacht werd. Dit is de betekenis van de foutmelding

```
Error: unexpected output was generated on stdout
```

Als je een `return` statement had gebruikt in plaats van de `print` functie, dan waren beide foutmeldingen verdwenen en kreeg je een **correct antwoord**. Merk op dat resultaten die door een functie/methode worden teruggegeven, in de feedbacktabel gemarkeerd worden met de tekst `return`. Resultaten die door een functie/methode worden uitgeschreven, worden in de feedbacktabel gemarkeerd met de tekst `stdout`.

Letters omzetten naar hun getalwaarde

In Python heeft elk karakter een getalwaarde (`int`). Zo heeft het vraagteken (`?`) als getalwaarde 63. Deze waarde kan bekomen worden met behulp van de ingebouwde functie `ord`.

```
>>> ord('?')
63
```

Het interessante aan deze getalwaarden is dat de opeenvolgende letters van het alfabet ook opeenvolgende getalwaarden hebben. Dit geldt zowel voor kleine letters als voor hoofdletters. Zo heeft de letter `a` als getalwaarde 97, waaruit we kunnen afleiden dat de letter `b` als getalwaarde 98 moet hebben. Met die wetenschap kunnen we dus op een eenvoudige manier de positie van een letter in het alfabet bepalen.

```
>>> letter = 'd'
>>> ord(letter)
100
>>> ord('a')
97
>>> ord(letter) - ord('a') + 1   # d is de 4e letter van het alfabet
4
>>> ord('z') - ord('a') + 1      # z is de 26e letter van het alfabet
26
```

ISBN

Gegevenstype controleren met `isinstance`

Om na te gaan of een gegeven object een bepaald gegevenstype heeft, kan je natuurlijk gebruikmaken van de ingebouwde functie `type(o)` die het gegevenstype van het object `o` teruggeeft. Het is echter beter om hiervoor de ingebouwde functie `isinstance(o, t)` te gebruiken. Deze functie geeft een Booleaanse waarde terug die aangeeft of het object `o` al dan niet behoort tot het type `t`.

```
>>> type(3) == int
True
>>> isinstance(3.14, int)
False
>>> isinstance(3.14, float)
True
>>> isinstance([1, 2, 3], list)
True
```

Als je wil controleren of een gegeven object 1 van meerdere types is, kun je de volgende syntax gebruiken. Hierbij lijst je de mogelijke toegelaten types op in een tuple.

```
>>> isinstance(3, (str, int))
True
>>> isinstance('a', (str, int))
True
>> isinstance(['a'], (str, int))
False
```