

Algemeen

Funcities/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

```
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 indienen voor deze opgave.

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

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

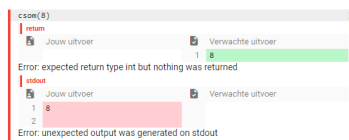


Figure 1: return vs print

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**.

Hergebruik van bestaande functionaliteit

Funcities 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 hierbij ook rekening met het feit dat je vanuit een functie ook andere functies kunt aanroepen. Soms staat het expliciet vermeld in de opgave dat je bij de implementatie van een functie, een andere functie (die je eerder reeds geïmplementeerd hebt) moet hergebruiken. Maar ook als een dergelijke expliciete 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 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 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 argument, 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 elementen én hun posities van sequentietypes

De ingebouwde functie `enumerate` kan gebruikt worden om een iterator op te vragen van een sequentietype (*sequence type*: strings, lijsten, tuples, bestanden, ...), die zowel de positie als de waarde van het volgende element van het type teruggeeft. Onderstaand voorbeeld geeft bijvoorbeeld aan hoe je tegelijkertijd de posities en de karakters op die posities kunt overlopen voor een string.

```
>>> 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 iterateerbare objecten.

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