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`.

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

```
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.

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

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

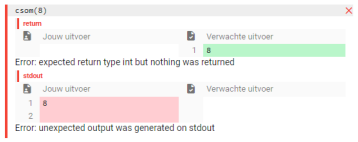

Figure 1: return vs print

Hierbij worden twee opmerkingen geformuleerd. De eerste opmerking geeft aan dat er verwacht werd dat de functie een integer (`int`) 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

Elk karakter heeft 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 de ingebouwde functie `type(o)` gebruiken 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 (`bool`) 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 één van meerdere types is, kun je de volgende syntax gebruiken. Hierbij lijst je de mogelijke toegelaten types op in een tuple (`tuple`).

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

Humble-Nishiyama gokspel

Controle of bepaalde voorwaarden gelden

Soms moet je in een programma uitdrukkelijk nagaan of er aan bepaalde voorwaarden voldaan is, en moet je programma reageren als één van de voorwaarden geschonden is. Eén van de manieren waarop je dit kunt doen is door een `assert` statement te gebruiken.

```
>>> x = 2
>>> y = 2
>>> assert x == y, 'beide getallen zijn niet gelijk'
>>> x = 1
>>> assert x == y, 'beide getallen zijn niet gelijk'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: beide getallen zijn niet gelijk
```

De algemene syntaxis van een `assert` statement is

```
assert <voorwaarde>, <boodschap>
```

Het `assert` statement controleert of er aan de voorwaarde voldaan is. Als dat niet het geval is, dan zal er een `AssertionError` opgeworpen worden met de boodschap (`str`) die wordt meegegeven op het einde van het

`assert` statement. Als deze uitzondering (Engels: *exception*) niet wordt opgevangen (wat voor deze cursus altijd het geval zal zijn), dan wordt de uitvoer van de code ook beëindigd (*runtime error*) met een melding van de `AssertionError`.

Rövarspråket

Specifieke info

Een strategie die je voor deze opgave kunt gebruiken, bestaat erin dat je de verwerking van een groep medeklinkers uitstelt tot op het ogenblik dat je een niet-medeklinker tegenkomt in de string (waardoor je weet dat de voorgaande groep medeklinkers compleet is). Dit wordt geïllustreerd aan de hand van het onderstaand voorbeeld, waarbij de string `abcdefg` wordt omgezet naar Rövarspråket.

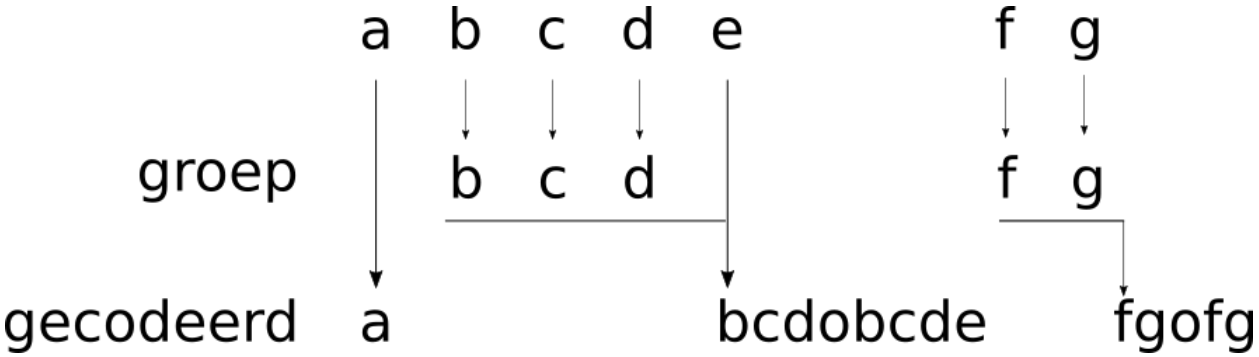

Figure 2: Rövarspråket

Hierbij wordt de string die moet gecodeerd worden karakter per karakter overlopen. Als hierbij een medeklinker wordt tegengekomen, dan wordt die niet onmiddellijk toegevoegd aan de gecodeerde string, maar wordt die toegevoegd aan de groep opeenvolgende medeklinkers. Op het moment dat er een niet-medeklinker wordt tegengekomen, dan sluit deze de groep van opeenvolgende medeklinkers af. Indien er zich een groep gevormd had, dan wordt die eerst aan de gecodeerde string toegevoegd, gevolgd door de letter `o` en nog een herhaling van de groep opeenvolgende medeklinkers. Pas daarna wordt de niet-medeklinker aan de gecodeerde groep toegevoegd, en wordt er een nieuwe groep medeklinkers gevormd.

Nadat alle karakters van de te coderen string overlopen zijn, kan het zijn dat er zich nog een groep van opeenvolgende medeklinkers gevormd heeft die niet aan de gecodeerde string werd toegevoegd. Dit is het geval wanneer de string eindigt op een medeklinker. In het voorbeeld is dit het geval voor de groep `fg`. Indien dit geval is, dan moet deze groep ook nog toegevoegd worden aan de gecodeerde string, gevolgd door de letter `o` en nog een herhaling van de groep opeenvolgende medeklinkers.