Zoemzinnen

Algemene info

Functies met een variabel aantal argumenten

Bij het definiëren van een functie leg je in principe vast hoeveel argumenten er aan de functie moeten doorgegeven worden. Dit aantal correspondeert met het aantal parameters die opgegeven worden bij de definitie van de functie. Aan onderstaande functie moeten bijvoorbeeld twee argumenten doorgegeven worden, en zal de functie het resultaat van de optelling van de twee doorgegeven objecten teruggeven.

```
>>> def som(term1, term2):
...     return term1 + term2
...
>>> som(1, 2)
3
```

Stel echter dat we een functie willen schrijven waaraan men nul of meer argumenten moet kunnen doorgeven, waarop de functie het resultaat van de som van alle doorgegeven objecten moet teruggeven. Dit kan door een parameter te laten voorafgaan door een asterisk (`*`). Aan deze parameter zal een tuple toegekend worden dat alle argumenten bevat die positioneel werden doorgegeven bij het aanroepen van de functie en die niet aan andere parmaters werden toegekend.

```
>>> def som(*termen):
...     totaal = 0
...     for term in termen:
...         totaal += term
...     return totaal
...
>>> som(1, 2)
3
>>> som(1, 2, 3)
6
>>> som(1, 2, 3, 4)
10
```

In dit geval zullen alle argumenten die aan de functie `som` doorgegeven worden, gebundeld worden in een tuple dat wordt toegekend aan de lokale variabele `termen`. Het is ook mogelijk om andere parameters op te geven bij de definitie van de functie, zolang de parameter die voorafgegaan wordt door een asterisk als laatste parameter gedefinieerd wordt. Er kan ook slechts één parameter voorkomen die voorafgegaan wordt door een asterisk.

Hieronder definiëren we bijvoorbeeld een functie `som` waaraan minstens twee argumenten moeten doorgegeven worden. De functie geeft nog altijd de som van alle argumenten terug die aan de functie worden doorgegeven.

```
>>> def som(term1, term2, *termen):
...     totaal = term1 + term2
...     for term in *termen:
...         totaal += term
...     return totaal
...
>>> som(1, 2)
3
```

```
>>> som(1, 2, 3)
6
>>> som(1, 2, 3, 4)
10
```

Dawkins' wezel

Algemene info

De module `random`

De module random uit The Python Standard Library kan gebruikt worden om willekeur toe te voegen aan je Python code. Deze module bevat onder andere de volgende functies.

functie	korte omschrijving
`random()`	genereert een willekeurig reëel getal uit het interval $[0, 1[$
`randint(a, b)`	genereert een willekeurig geheel getal uit het interval $[a, b]$
`choice(s)`	kiest een willekeurig element uit de sequentie `s`
`sample(s, k)`	kiest willekeurig `k` verschillende elementen uit de sequentie `s`
`shuffle(l)`	schudt de elementen van de lijst l willekeurig door elkaar

Hieronder staan alvast enkele voorbeelden.

```
>>> import random

>>> random.random()
0.954131645221452
>>> random.random()
0.3548429482674793

>>> random.randint(3, 10)
5
>>> random.randint(3, 10)
8

>>> lijst = ['a', 'b', 'c']
>>> random.choice(lijst)
'b'
>>> random.choice(lijst)
'a'
>>> lijst
['a', 'b', 'c']

>>> random.sample(lijst, 2)
['a', 'c']
>>> random.sample(lijst, 2)
['b', 'a']
>>> lijst
['a', 'b', 'c']
```

```
>>> random.shuffle(lijst)
>>> lijst
['c', 'a', 'b']
```

Onvoorspelbare verjaardagen

Algemene info

De module `datetime`

De module datetime uit The Python Standard Library definieert een aantal nieuwe gegevenstypes die datums (`datetime.date` objecten) en tijdsintervallen (`datetime.timedelta` objecten) voorstellen. Hieronder alvast enkele voorbeelden.

```
>>> from datetime import date
>>> geboortedatum = date(1990, 10, 3)
>>> geboortedatum = date(day=3, month=10, year=1990)
>>> geboortedatum.day          # day is een eigenschap
3
>>> geboortedatum.month        # month is een eigenschap
10
>>> geboortedatum.year         # year is een eigenschap
1990
>>> geboortedatum.weekday()    # weekday is een methode !!
2
>>> vandaag = date.today()
>>> vandaag
datetime.date(2015, 11, 10)    # uitgevoerd op 11 oktober 2015
>>> from datetime import timedelta
>>> morgen = vandaag + timedelta(1)
>>> morgen
datetime.date(2015, 11, 11)
>>> verschil = morgen - vandaag
>>> type(verschil)
datetime.timedelta
>>> verschil.days
1
```

Algemeen

Verzamelingen en dictionaries in doctests

De elementen van een verzameling en de sleutels van een dictionary hebben geen vaste volgorde. Dit betekent dat twee verzamelingen of twee dictionaries gelijk zijn, ongeacht de volgorde waarin de elementen/sleutels voorkomen bij de definitie ervan.

```
>>> {1, 3, 2, 4} == {4, 3, 1, 2}
True
>>> {'A': 1, 'B': 2, 'C': 3} == {'B': 2, 'A': 1, 'C': 3}
True
```

Verzamelingen en dictionaries kunnen echter problemen opleveren als je werkt met doctests, omdat die bij het vergelijken van de verwachte en gegenereerde uitvoer als volgt te werk gaan: de verwachte uitvoer wordt uit de docstring geëxtraheerd als een string, en de waarde die door de functie wordt teruggegeven of die als resultaat bekomen wordt bij de evaluatie van een expressie, wordt omgezet naar een string. Deze twee strings worden met elkaar vergeleken (als string, niet als verzameling of als dictionary). Bij het vergelijken van strings speelt de volgorde van de karakters wel een rol, en hier zit net het probleem.

```
>>> d1 = {'A': 1, 'B': 2, 'C': 3}
>>> s1 = str(d1)   # uitgevoerd op computer 1
>>> d1
"{'A': 1, 'C': 3, 'B': 2}"
>>> d2 = {'A': 1, 'B': 2, 'C': 3}
>>> s2 = str(d2)   # uitgevoerd op computer 2
"{'A': 1, 'B': 2, 'C': 3}"
>>> d1 == d2
True
>>> s1 == s2
False
```

Hoewel de dictionaries `d1` en `d2` dezelfde waarde hebben, geeft de doctest aan dat de stringvoorstellingen van deze twee dictionaries verschillend zijn. Het omzetten van een dictionary naar een string kan immers afhangen van de computer waarop je de code uitvoert, van de versie van Python die gebruikt wordt en kan zelfs verschillen als je het een paar keer na elkaar uitvoert op dezelfde computer met dezelfde Python versie. Het probleem kan opgelost worden door ervoor te zorgen dat de doctest geen strings maar elkaar vergelijkt, maar rechtstreeks de verzamelingen of dictionaries met elkaar vergelijkt. Als je dus oorspronkelijk een doctest krijgt van de volgende vorm

```
>>> functie(parameter1, parameter2)
{'A' : 1, 'B': 2, 'C': 3}
```

dan kan je deze doctest beter herschrijven als

```
>>> functie(parameter1, parameter2) == {'A' : 1, 'B': 2, 'C': 3}
True
```

Indien je je oplossing indient op Dodona, dan zullen de resultaten wel degelijk als verzamelingen of dictionaries geïnterpreteerd worden, en niet als strings. Daar doet het probleem zich dus niet voor.

Busroddels

Algemene info

Operatoren voor verzamelingen

In Python is het eenvoudig om de unie, de doorsnede en het verschil te bepalen van twee verzamelingen.

```
>>> verzameling1 = {'A', 'B', 'C'}
>>> verzameling2 = {'C', 'D', 'E'}
>>> verzameling1 & verzameling2        # doorsnede
{'C'}
>>> verzameling1 | verzameling2        # unie
{'A', 'B', 'C', 'D', 'E'}
>>> verzameling1 - verzameling2        # verschil
{'A', 'B'}
>>> verzameling2 - verzameling1        # verschil is assymetrisch
{'D', 'E'}
```

Evendeling

Specifieke info

De methode samenhangend kan best geïmplementeerd worden met een flood fill algoritme.

We starten hiervoor met een willekeurig element e uit onze container S en plaatsen het in een lijst L. L zal de lijst zijn die alle elementen bevat die we als bereikbaar gelabeld hebben, maar waarvan de buren nog niet noodzakelijk gelabeld zijn. Voeg e toe aan L. Vervolgens blijven we de volgende procedure uitvoeren tot L leeg is.

- Neem het eerste element f uit L
- Voor alle niet gelabelde buren b van f in S
  - Voeg b toe aan L
  - Label b als bereikbaar
- Verwijder f uit L

Als L leeg is, zijn alle elementen uit S die bereikbaar zijn vanuit e gelabeld. Als er dus nog elementen in S zijn die niet gelabeld zijn, is onze container niet samenhangend.

Tips:

- Je kunt de container S best omvormen naar een lijst
- In plaats van een element te labelen in S kan het ook verwijderd worden uit S. Dit kun je efficiënt doen door zijn waarde naar None te veranderen in de S.

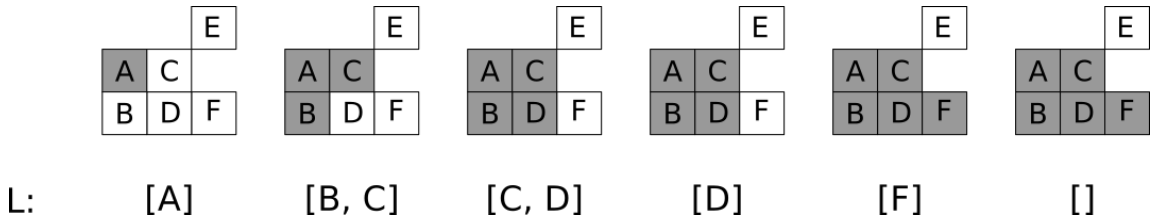

Figure 1: Flood fill