

Broodkruimels

Een element uit een set halen

Om een onbepaald element uit de set te halen kan gebruik gemaakt worden van de `.pop()` functie. Hierbij zal er een element uit de set verwijderd en teruggegeven worden. Merk hierbij op dat je niet op voorhand weet welk element dit zal zijn, maar dat dit ook niet echt willekeurig is.

```
>>> een_set = {'ABC', 'abc', '123'}
>>> een_set.pop()
'abc'
>>> een_set
{'123', 'ABC'}
```

Wanneer je een bepaald element uit een set wilt halen kan de `.remove()` functie gebruikt worden.

```
>>> een_set = {'ABC', 'abc', '123'}
>>> een_set.remove('123')
>>> een_set
{'abc', 'ABC'}
```

Specifieke info

Om het algoritme dat beschreven staat te verduidelijken geven we een woordje uitleg bij elke stap:

1. initialisatie $\mathcal{B} = \emptyset, \mathcal{R} = \{(0, 0)\}$

In Python kunnen we dit schrijven als:

```
B = set()
R = {(0,0)}
```

2. haal een willekeurig punt p uit $\mathcal{R}$

Hiervoor kan je de `.pop()` functie gebruiken zoals beschreven in deze tips-en-tricks.

3. voeg p toe aan $\mathcal{B}$

Dit kan met de `.add()` functie.

4. voeg de burens van p toe aan $\mathcal{R}$ als die burens bereikbaar zijn en nog niet in $\mathcal{B}$ zitten

Hiervoor wil je jouw zelf geschreven functie `buren(...)` aanroepen. Voor elk element uit het resultaat van deze functie kan je controlleren of het al in $\mathcal{B}$ aanwezig is. Indien niet, dan voegen we deze buur toe aan $\mathcal{R}$.

5. herhaal vanaf stap 2 totdat $\mathcal{R} = \emptyset$

In Python-termen betekent dit dus een `while`-lus rond stappen 2 tot en met 4.

De heilige birmaan

Waarden toekennen aan sleutels in een dictionary

Bij het opbouwen van een dictionary is het soms nodig om een waarde aan een bepaalde sleutel toe te kennen, rekening houdend met feit dat er eventueel reeds een waarde met die sleutel geassocieerd is. Hierbij is het vaak zo dat je een nieuw sleutel/waarde paar moet toevoegen indien de sleutel nog niet gebruikt werd in de dictionary, en dat je een bestaand sleutel/waarde paar moet bijwerken indien de sleutel wel reeds gebruikt werd in de dictionary.



Figure 1: updaten dictionary

Deze techniek kan onder meer gebruikt worden om frequentietabellen op te bouwen. Frequentietabellen zijn dictionaries waarvoor iedere sleutel een waarde heeft die overeenkomt met het aantal keer dat die sleutel voorkomt in een containerobject (bv. een lijst, een tuple of een verzameling).

```

>>> lijst = ['R', 'S', 'E', 'E', 'N', 'T', 'E', 'I', 'L', 'D', 'I']
>>> frequentietabel(lijst)
{'E': 3, 'S': 1, 'D': 1, 'N': 1, 'T': 1, 'R': 1, 'L': 1, 'I': 2}

```

Bovenstaande techniek kan gebruikt worden om de functie `frequentietabel` te implementeren.

```

def frequentietabel(lijst):
    tabel = {}                # lege dictionary aanmaken
    for element in lijst:
        if element not in tabel:
            tabel[element] = 0 # element toevoegen aan dictionary met initiële waarde 0
            tabel[element] += 1 # waarde geassocieerd met element bijwerken

```

In dit geval kan je echter ook gebruik maken van de dictionary-methode `get`. Deze methode vraagt de waarde op die in de dictionary geassocieerd is met een bepaalde sleutel. In tegenstelling tot het indexeren van een dictionary aan de hand van vierkante haakjes om de waarde op te halen die geassocieerd is met een bepaalde sleutel, zal de methode `get` niet resulteren in een `KeyError` indien de sleutel niet voorkomt in de dictionary. In plaats daarvan zal de methode `get` standaard de waarde `None` teruggeven. Als je een waarde doorgeeft aan de tweede optionele parameter, dan zal die waarde als standaardwaarde teruggegeven worden als de methode `get` de sleutel niet terugvindt in de dictionary.

```

>>> d = {'A': 1, 'B': 2, 'C': 3}
>>> d['A']
1
>>> d.get('A')
1
>>> d['D']
Traceback (most recent call last):
  KeyError: 'D'
>>> d.get('D')
None
>>> d.get('D', 0)
0

```

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.

Dictionaries gebruiken om if statements te vermijden

Soms heb je in je code zeer lange `if-elif-else` statements waarbij alle gevallen eigenlijk hetzelfde doen, maar er één variabele is die verschilt voor elk geval. In dergelijke gevallen is het meestal beter om de waarde van die variabele op te zoeken in een dictionary, en de geassocieerde waarde te gebruiken als resultaat.

```
>>> if x == 'A':
...     y += 3
```

```
... elif x == 'B':  
...     y += 1  
... elif x == 'C':  
...     y += 7
```

Deze code kan korter geschreven worden door gebruik te maken van een dictionary. Hierbij zijn de sleutels van de dictionary de verschillende waarden voor `x` en de waarde die bij iedere sleutel hoort komt overeen met de waarde die bij `y` geteld moet worden.

```
>>> afbeelding = {'A': 3, 'B': 1, 'C': 7}  
>>> y += afbeelding[x]
```

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'}
```