Hoofdstuk 13

Dictionaries

Strings, tuples en lists zijn geordende data structuren, wat inhoudt dat ze via indices benaderd kunnen worden. Maar niet alle data verzamelingn hebben een natuurlijke manier van numeriek geordend zijn, en deze kunnen dus niet (gemakkelijk) geïndiceerd worden. Python biedt "dictionaries" als een manier om ongeordende data te structureren.

13.1 Dictionary basis

Een "dictionary" (letterlijk: "woordenboek," maar die vergelijking loopt spaak in Python) is een ongeordende data structuur die een verzameling elementen bevat. Om een element te vinden, moet je de "key" ("sleutel") van het element kennen.

In de grond is een dictionary een verzameling van "keys" met geassocieerde waardes. Ieder onveranderbaar data type mag gebruikt worden als key. Een veelgebruikt data type dat als key wordt ingezet is de string.

Dictionaries creëer middels accolades `{}`, vergelijkbaar met hoe je lists creëert met vierkante haken. Een lege dictionary maak je door een assignment aan een variabele te doen met `{}`. Je kunt een dictionary met inhoud creëren door ieder element dat je erin wilt hebben tussen de accolades te zetten, met als syntax `<key>:<value>`, en komma's tussen de elementen.

Hieronder bouw ik een dictionary fruitmand, met drie elementen, namelijk de key `"appel"` met waarde 3, de key `"banaan"` met waarde 5, en de key `"kers"` met waarde 50.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
```

Om een waarde te vinden die hoort bij een specifieke sleutel, gebruik je dezelfde syntax als voor een list, behalve dat waar je bij een list de index schrijft, je bij een dictionary de gezochte key schrijft.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand["banaan"] )
```

Je kunt via een **for** loop een dictionary doorlopen. De variabele in de **for** loop krijgt de waardes van de keys.

listing1301.py

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
for key in fruitmand:
    print( "{}:{}".format( key, fruitmand[key] ))
```

Als je een dictionary element probeert te benaderen met een key die niet voorkomt in de dictionary, krijg je een runtime error. Maar als je een nieuw element wilt toevoegen, kun je dat eenvoudigweg doen door een waarde toe te kennen aan een dictionary element met de nieuwe key. Bijvoorbeeld, om een `"mango"` toe te voegen aan de `fruitmand`, doe je het volgende:

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand )
fruitmand["mango"] = 1
print( fruitmand )
```

Op dezelfde wijze kun je een bestaand dictionary element overschrijven.

Om een element te verwijderen uit een dictionary, gebruik je het gereserveerde woord **del**, net zoals je het gebruikt met lists.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand )
del fruitmand["banaan"]
print( fruitmand )
```

Je kunt het aantal elementen in de dictionary bepalen met de **len**() functie.

Snap je overigens in welke volgorde de dictionary de elementen aanbiedt als je de inhoud van de dictionary print? Denk er even over na.

Het antwoord is: de volgorde is willekeurig. Ik zei dat bij het begin van het hoofdstuk: dictionaries zijn ongeordend. Ik kan je niet eens zeggen wat voor volgorde je op je scherm ziet als een de code hierboven uitvoert, want de volgorde kan verschillen tussen computers, besturingssystemen, en versies van Python. Er is een zekere structuur in de ordening, maar niet een die je zou kunnen (of willen) voorspellen. Door voldoende nieuwe elementen toe te voegen, kan de volgorde zelfs plotseling drastisch wijzigen.

Omdat dictionaries ongeordend zijn, zijn vele concepten die gelden voor lists, niet van toepassing op dictionaries. Bijvoorbeeld, je kunt geen “subdictionary” maken door een key-bereik te definiëren, je kunt kunt een dictionary niet “sorteren” of “inverteren.” Dictionaries zijn daarom wat beperkt, maar ze kunnen nuttig zijn.

13.2 Dictionary methodes

Hieronder staan de dictionary methodes die het meest gebruikt worden.

13.2.1 copy()

Net als met lists, kun je een variabele die een dictionary bevat via de assignment operator aan een andere variabele koppelen, en daarmee maak je dan een alias voor de dictionary (als je je dat niet herinnert, moet je er hoofdstuk 12 nog eens op naslaan). Het is niet mogelijk dezelfde "truuk" die bij lists bestaat te gebruiken om een kopie te maken. In plaats daarvan hebben dictionaries een methode copy() die een kopie van de dictionary maakt en retourneert.

listing1302.py

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
fruitmandalias = fruitmand
fruitmandcopy = fruitmand.copy()

print( id( fruitmand ) )
print( id( fruitmandalias ) )
print( id( fruitmandcopy ) )
```

Merk op dat een dergelijke kopie een ondiepe kopie is (zie hoofdstuk 12 als je je het verschil tussen ondiepe en diepe kopieën niet herinnert). Als je een diepe kopie wilt maken, moet je de deepcopy() functie van de copy module gebruiken.

13.2.2 keys(), values(), and items()

De methode keys() levert een iterator die alle keys van de dictionary genereert. De methode values() levert een iterator die alle waardes van een dictionary genereert. De methode items() levert een iterator die 2-tuples genereert die alle keys en waardes van de dictionary bevatten.

Ik zeg specifiek dat deze methodes iterators retourneren en niet lists. Als je wat ze retourneren in lists wil veranderen, moet je list casting gebruiken (zie hoofdstuk 12).

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( list( fruitmand.keys() ) )
print( list( fruitmand.values() ) )
print( list( fruitmand.items() ) )
```

Op dit punt vraag je je wellicht af wanneer je een iterator kunt gebruiken. Je gebruikt iterators met name in **for** loops (maar je kunt ze ook gebruiken als argumenten voor de functies **max**(), **min**() en **sum**()).

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}
for key in fruitmand.keys():
    print( "{}:{}".format( key, fruitmand[key] ) )
print( sum( fruitmand.values() ) )
```

Omdat deze code een onvoorspelbare volgorde voor de keys doorloopt, wil je ze meestal sorteren voordat je ze in de loop verwerkt. Omdat keys() geen list maar een iterator levert,

kun je ze niet direct sorteren, maar moet je het resultaat van `keys()` eerst met een list casting in een list omzetten. Daarna kun je de list sorteren.

listing1303.py

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}
keylist = list( fruitmand.keys() )
keylist.sort()
for key in keylist:
    print( "{}:{}".format( key, fruitmand[key] ) )
```

Je kunt niet direct de `sort()` methode toepassen op de list casting, met andere woorden, `keylist = list( fruitmand.key() ).sort()` werkt niet. Je moet eerst de list creëren, en dan pas sorteren. Je kunt ook niet schrijven `for key in keylist.sort()`, omdat de `sort()` methode geen retourwaarde heeft.

Als je je afvraagt waarom Python iterators boven lists prefereert: het antwoord daarop is dat iterators meer generiek bruikbaar zijn en minder geheugen gebruiken. Het zijn "luie" methodes, omdat ze alleen een item produceren als erom gevraagd wordt.

13.2.3 get()

De `get()` methode kun je gebruiken om een waarde uit de dictionary te halen zelfs als je niet weet of de key die je zoekt wel in de dictionary zit. Je roept de `get()` methode aan met de key die je zoekt als argument, en het geeft de corresponderende waarde terug als de key bestaat, of de speciale waarde **`None`** als de key niet bestaat in de dictionary. Als je in plaats van **`None`** een andere waarde terug wilt krijgen als de key niet bestaat, dan kun je die waarde meegeven als het tweede, optionele argument.

listing1304.py

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}

appel = fruitmand.get( "appel" )
if appel:
    print( "appel is in de mand" )
else:
    print( "geen appels in de mand")

aardbei = fruitmand.get( "aardbei" )
if aardbei:
    print( "aardbei is in de mand" )
else:
    print( "geen aardbei in de mand")

banaan = fruitmand.get( "banaan", 0 )
print( "aantal bananen in de mand:", banaan )

aardbei = fruitmand.get( "aardbei", 0 )
print( "aantal aarbeien in de mand:", aardbei )
```

Voer de code hierboven uit en bestudeer de uitkomst, omdat wat de code demonstreert over de `get()` methode erg nuttig is. Stel je voor dat een collectie van items hebt met corresponderende hoeveelheden, bijvoorbeeld, de inhoud van een fruitmand waarbij de keys de namen van fruit zijn, en de waardes de hoeveelheden. Als je in de `fruitmand` dictionary zoekt met de `get()` methode en als tweede argument een nul, kun je naar een willekeurig stuk fruit zoeken in de mand zonder dat je eerst moet controleren of het bestaat in de mand, want als je naar een fruitnaam vraagt die er niet als key in voorkomt, krijg je nul terug, en dat is precies wat je wilt zien.

13.2.4 Oefening

Opgave De code hieronder bevat een list met woorden. Bouw een dictionary die al deze woorden als key heeft, en als waarde hoe vaak het woord voorkomt in de list. Print daarna de woorden met hun hoeveelheden.

listing1305.py

```
woordlist = ["appel","doerian","banaan","doerian","appel","kers",
    "kers","mango","appel","appel","kers","doerian","banaan",
    "appel","appel","appel","appel","banaan","appel"]
```

Opgave De code hieronder bevat een string met woorden gescheiden door komma's. Bouw een dictionary die al deze woorden als key heeft, en als waarde hoe vaak het woord voorkomt in de list. Print daarna de woorden met hun hoeveelheden.

listing1306.py

```
tekst = "appel,doerian,banaan,doerian,appel,kers,kers,mango," + \
    "appel,appel,kers,doerian,banaan,appel,appel,appel," + \
    "appel,banaan,appel"
```

Opgave De code hieronder bevat een kleine dictionary die vertalingen bevat van Engelse woorden naar Nederlandse. Schrijf een programma dat met behulp van deze dictionary een woord-voor-woord vertaling maakt van de tekst eronder. Een woord dat niet in de dictionary voorkomt, vertaal je niet. De dictionary gebruikt alleen kleine letters, en je kunt de hele tekst naar kleine letters omzetten voordat je de vertaling maakt. Het is aardig als je interpunctie in stand houdt in de vertaling, maar je mag die ook weglaten (het is best veel werk om de interpunctie intact te houden en het heeft niks van doen met dictionaries, dus het is niet belangrijk – het is ook veel gemakkelijker om dit te doen als je eenmaal geleerd hebt om te gaan met reguliere expressies, in hoofdstuk 25). Ik besef heel goed dat de vertaling uitermate slecht is, maar daar gaat het niet om.

listing1307.py

```
engels_nederlands = { "last":"laatst", "week":"week", "the":"de",
    "royal":"koninklijk", "festival":"feest", "hall":"hal",
    "saw":"zaag", "first":"eerst", "performance":"optreden",
    "of":"van", "a":"een", "new":"nieuw", "symphony":"symphonie",
```

```
    "by":"bij", "one":"een", "world":"wereld", "leading":
    "leidend", "modern":"modern", "composer":"componist",
    "composers":"componisten", "two":"twee", "shed":"schuur",
    "sheds":"schuren" }

zin = "Last week The Royal Festival Hall saw the first \
performance of a new symphony by one of the world's leading \
modern composers, Arthur \"Two-Sheds\" Jackson."
```

13.3 Keys

Zoals ik aangaf kan ieder onveranderbaar data type een dictionary key zijn. Dat betekent dat je strings, integers, en floats kunt gebruiken als keys. Je herinnert je wellicht dat tuples ook onveranderbaar zijn, wat betekent dat ook tuples als keys gebruikt kunnen worden. Dat is soms nuttig.

Een eenvoudig voorbeeld van het nut van tuples als keys is een dictionary waarbij je informatie wilt opslaan die geassocieerd is met punten in een 2-dimensionale ruimte (ik besprak dit in hoofdstuk 11). Er is geen goede manier waarmee je een 2-dimensionaal punt kunt opslaan als een enkel getal of string. Het is niet onmogelijk (je kunt bijvoorbeeld het getallenpaar omzetten naar hun string-representatie en ze met een komma ertussen tot één string maken) maar het wordt al snel verwarrend (bijvoorbeeld, de strings `"2,3"`, `"2, 3"`, `"+2,+3"`, en `"02,03"` zouden alle dezelfde tuple representeren, terwijl het verschillende keys zijn).

13.4 Opslaan van complexe waardes

Tot op dit moment heb ik alleen gesproken over het opslaan bij een key in een dictionary van een enkele waarde van een enkel data type. Het is echter ook mogelijk om complexe waardes op te slaan in een dictionary. De waardes kunnen willekeurige Python objecten zijn. Bijvoorbeeld, je kunt bij iedere key een list opslaan. Hieronder staat een dictionary waarbij ik studenten opsla die een cursus volgen. De cursus wordt geïdentificeerd door het cursusnummer. De studenten worden geïdentificeerd door hun studentnummers.

listing1308.py

```
curses = {
    '880254':['u123456', 'u383213', 'u234178'],
    '822177':['u123456', 'u223416', 'u234178'],
    '822164':['u123456', 'u223416', 'u383213', 'u234178']}

for c in curses:
    print( c )
    for s in curses[c]:
        print( s, end=" " )
    print()
```

Stel je voor dat ik niet alleen de studentnummers per cursus wil opslaan, maar ook de cursusnaam, de cursus ECTS (ECTS is een standaard voor studiepunten), en voor iedere student een eindcijfer. Je kunt dat (bijvoorbeeld) doen door als waarde bij een cursusnummer een dictionary op te nemen, met drie keys: `"naam"`, `"ects"`, en `"studenten"`. De waarde bij `"naam"` is de cursusnaam als een string, de waarde voor `"ects"` is een integer, en de waarde voor `"studenten"` is een andere dictionary, die studentnummers als keys gebruikt en eindcijfers als waardes.

listing1309.py

```
curses = {
    '880254':
    { "naam":"Onderzoeksvaardigheden Data Processing", "ects":3,
        "studenten":{'u123456':8, 'u383213':7.5, 'u234178':6} },
    '822177':
    { "naam":"Logica", "ects":6,
        "studenten":{'u123456':5, 'u223416':7, 'u234178':9} },
    '822164':
    { "naam":"Computer Games", "ects":6,
        "studenten":{'u123456':7.5, 'u223416':9 } } }

for c in curses:
    print( "{}: {} ({})".format( c, curses[c]["naam"],
        curses[c]["ects"] ) )
    for s in curses[c]["studenten"]:
        print( "{}: {}".format( s, curses[c]["studenten"][s] ) )
    print()
```

Data structuren kunnen nog complexer worden dan dit als je dat wilt. Echter, als je inderdaad overweegt om Python programma's te maken voor dit soort data structuren, doe je er goed aan om eerst object oriëntatie te bestuderen (hoofdstuk 20 en verder) en waarschijnlijk een aparte cursus over databases te volgen.

13.5 Snelheid

Lists en dictionaries zijn de twee meest-gebruikte data structuren in Python. Het is vaak duidelijk welk van de twee je moet gebruiken bij een probleem, maar het kan nuttig zijn om iets te weten over hoe Python deze data structuren verwerkt voor het geval je een keus hebt.

Stel je voor dat je een groot aantal getallen uit een bestand moet lezen. De getallen zijn allemaal verschillend en kunnen willekeurig groot zijn. Je moet later getallen van een andere lijst vergelijken met de getallen die je uit het bestand hebt gelezen.

Moet je een list of een dictionary gebruiken om de getallen die je inleest op te slaan? Omdat het alleen maar getallen zijn en geen extra data, lijkt een list de beste optie. Er is echter een probleem dat optreedt als je hier een list gebruikt. Bekijk de volgende code, die een list creëert met 10000 getallen, en daarna bekijkt of 10000 andere getallen in de list voorkomen (wat geldt voor geen van de getallen).

listing1310.py

```
from datetime import datetime

numlist = []
for i in range( 10000 ):
    numlist.append( i )

start = datetime.now()
teller = 0
for i in range( 10000, 20000 ):
    if i in numlist:
        teller += 1
eind = datetime.now()

print( "{}.{} seconden om {} nummers te vinden".format(
    (eind-start).seconds, (eind-start).microseconds, teller ) )
```

Hier is code die hetzelfde doet, maar de getallen inleest in een dictionary, waarbij simpelweg voor ieder ingelezen getal een waarde van 1 in de dictionary wordt opslagen.

listing1311.py

```
from datetime import datetime

numdict = {}
for i in range( 10000 ):
    numdict[i] = 1

start = datetime.now()
teller = 0
for i in range( 10000, 20000 ):
    if i in numdict:
        teller += 1
eind = datetime.now()

print( "{}.{} seconden om {} nummers te vinden".format(
    (eind-start).seconds, (eind-start).microseconds, teller ) )
```

Als je deze code uitvoert, zul je zien dat voor de dictionary het antwoord vrijwel onmiddellijk volgt, maar dat het voor een list wat langer duurt. De reden is dat ik met behulp van de **in** operator controleer of een getal voorkomt in de list of de dictionary. Voor een list betekent dat dat Python de hele list sequentieel doorzoekt, totdat het het getal vindt of het einde van de list bereikt wordt. Dit houdt in dat Python 10000 keer 10000 getallen controleert (omdat het geen enkel getal kan vinden), ofwel 100 miljoen getallen.

Voor een dictionary is het zoeken van een key veel sneller. Python kan erg snel vaststellen of een key wel of niet in een dictionary zit.[18] Meestal is het controleren van een handjevol getallen genoeg. Daarom is de dictionary code veel, veel sneller.

[18] Python slaat de keys voor een dictionary op in een zogeheten "hash tabel." Ik leg de details daarvan niet uit, maar weet dat dit het mogelijk maakt om keys heel snel op te zoeken, ten koste van wat geheugengebruik.

Je denkt misschien dat een paar seconden zoektijd voor de list nog steeds erg weinig is, maar de zoektijd neemt kwadratisch toe met de hoeveelheid data. Afhankelijk van het soort en de omvang van het probleem, kan een dictionary zeer te prefereren zijn boven een list.

Lists nemen wel minder geheugen in beslag dan dictionaries, en als je een list direct kunt benaderen via een index, kunnen lists zeker sneller zijn dan dictionaries. Bijvoorbeeld, in bovenstaande probleem, als ik zou weten dat de list gesorteerd is dan kan ik getallen op een veel slimmere manier vinden dan via de **`in`** operator (ongeveer 14 getallen controleren volstaat) – in dat geval is het gebruik van een list misschien sneller dan het gebruik van een dictionary.

Onthoud hiervan dat een list snel is als je de elementen via hun index kunt benaderen, terwijl een dictionary een betere keuze is als de enige manier om iets te zoeken is de elementen te scannen. De **`in`** operator lijkt gemakkelijk en is leesbaar, maar als je hem gebruikt om iets te zoeken in een lange list, dan ben je verkeerd bezig.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Dictionaries
- Dictionary keys en waardes
- Dictionary methodes `copy()`, `keys()`, `values()`, `items()`, en `get()`
- Complexe dictionaries
- Snelheid verschillen tussen lists en dictionaries

Excercises

Opgave 13.1 Schrijf een programma dat een tekst neemt (bijvoorbeeld de tekst hieronder), de tekst splitst in woorden (waarbij alles dat geen letter is beschouwd wordt als een woord-scheider), en een dictionary bouwt die voor ieder woord (case-insensitive) opslaat hoe vaak het woord voorkomt in de tekst. Print dan alle woorden met hun hoeveelheden in alfabetische volgorde.

exercise1301.py

```
tekst = """Kapper Knap, de knappe kapper, knipt en kapt heel
knap, maar de knecht van kapper Knap, de knappe kapper, knipt
en kapt nog knapper dan kapper Knap, de knappe kapper."""
```

Opgave 13.2 De code hieronder bevat een list van films. Voor iedere film is er ook een list met scores. Verander deze code zo dat het alle data opslaat in één dictionary, en gebruik dan de dictionary om de gemiddelde score voor iedere film af te drukken, afgerond op één decimaal.

exercise1302.py

```
films = ["Monty Python and the Holy Grail",
         "Monty Python's Life of Brian",
         "Monty Python's Meaning of Life",
         "And Now For Something Completely Different"]

grail_scores = [ 9, 10, 9.5, 8.5, 3, 7.5,8 ]
brian_scores = [ 10, 10, 0, 9, 1, 8, 7.5, 8, 6, 9 ]
life_scores = [ 7, 6, 5 ]
different_scores = [ 6, 5, 6, 6 ]
```

Opgave 13.3 Een bibliotheek heeft boeken. Ieder boek heeft een schrijver, die je kunt identificeren door achter- en voornaam. Boeken hebben een titel. Boeken hebben ook een locatienummer dat aangeeft waar ze staan in de bibliotheek. De bibliothecaris wil kunnen vinden waar een boek staat als hij de schrijver en titel kent, en wil ook alle boeken kunnen afdrukken die van een bepaalde schrijver zijn. Welke data structuur kun je gebruiken om de boeken in op te slaan?