Hoofdstuk 10

Strings

Tot nu toe gebruikten de meeste voorbeelden en opgaves getallen. Je hebt je misschien afgevraagd of programmeren vooral bedoeld is voor het manipuleren van getallen. In het dagelijks leven is het veel gebruikelijker om met tekstuele informatie om te gaan.

De reden dat de omgang met teksten was uitgesteld tot nu, is dat in programmeertalen het veel gemakkelijker is om met getallen om te gaan dan met teksten. Maar in dit hoofdstuk leg ik uit hoe je teksten kunt manipuleren met programma's.

In programmeeromgevingen worden teksten weergegeven door strings. Dit hoofdstuk geeft details over strings, en over functies die beschikbaar zijn om strings aan te pakken.

10.1 Herhaling

In hoofdstuk 3 heb ik strings kort geïntroduceerd. Ik heb uitgelegd dat een string een tekst is, die omsloten is door enkele of dubbele aanhalingstekens, en dat een string iedere lengte mag hebben, inclusief nul tekens lang. Het hoofdstuk legde ook uit dat je twee strings aan elkaar kunt plakken met behulp van de $+$, en dat je een string zichzelf kunt laten herhalen door middel van de $*$. Bijvoorbeeld:

```
s1 = "appel"
s2 = 'banaan'
print( s1 )
print( s2 )
print( s1 + s2 )
print( 3 * s1 )
print( s2 * 3 )
print( 2 * s1 + 2 * s2 )
```

Hoofdstuk 5 introduceerde de **format**() functie die strings op allerlei manieren kan formatteren. Ik gaf ook aan dat je de lengte van een string kunt bepalen met de **len**() functie.

String vergelijkingen heb ik uitgelegd in hoofdstuk 6; ik noemde specifiek het feit dat bij vergelijkingen tussen strings de alfabetische regels worden aangehouden, waarbij

hoofdletters altijd eerder in het alfabet staan dan kleine letters. Ik zal hier in dit hoofdstuk meer over zeggen. In hoofdstuk 6 gaf ik ook aan dat de **in** operator gebruikt kan worden om te testen of tekens of substrings voorkomen in een string.

Hoofdstuk 7 legde uit hoe je met een **for** loop alle tekens in een string kunt doorlopen.

```
s1 = "mango"
s2 = "banaan"
for letter in s1:
    if letter in s2:
        print( s1, "en", s2, "bevatten beide de letter", letter )
```

10.2 Strings over meerdere regels

In Python kunnen strings meerdere regels beslaan. Dat kan nuttig zijn wanneer je een erg lange string in je programma hebt, of als je de output van de string op een specifieke manier wilt formatteren. Dit kan op twee manieren bereikt worden:

- Met enkele of dubbele aanhalingstekens, en een indicatie dat de string doorloopt op de volgende regel door een backslash aan het einde van de regel te zetten.
- Met drievoudige enkele of dubbele aanhalingstekens.

Ik demonstreer eerst hoe het werkt met enkele of dubbele aanhalingstekens om de string:

listing1001.py

```
langestring = "Ik heb mijn buik vol van te worden behandeld \
als een schaap. Wat is de zin van op vakantie gaan als je \
gewoon maar een toerist bent die wordt rondgereden in een bus \
omringd door zweterige uilskuikens uit Den Haag en Rotterdam \
met hun honkbalpetjes en trainingspakken en radio's en \
De Telegraaf, klagend over de koffie - 'Oh, ze zetten geen \
lekker bakje hier, toch, niet zoals thuis' - en dan stoppen \
bij Mallorcaanse eettentjes waar ze friet en kroket verkopen \
en Heineken en calamaris met salade en ze zitten in hun \
katoenen overgooiers Nivea zonnebrand te spuiten over hun \
pafferige rauwe opgezwollen uitpuilende blubberbuiken 'omdat \
ze op de eerste dag wat teveel zon hebben gehad.'"
print( langestring )
```

Als je deze code uitvoert, zie je dat Python de string als een geheel interpreteert. De backslash (\) die aangeeft dat de string verder gaat op de volgende regel is algemener bruikbaar dan alleen voor strings: je kunt hem achter ieder Python statement zetten om aan te geven dat het statement verder gaat op de volgende regel. Dat kan nuttig zijn bij bijvoorbeeld lange berekeningen.

De aanbevolen manier om strings over meerdere regels te spreiden in Python is echter het gebruik van drievoudige aanhalingstekens. Ik heb in een eerder hoofdstuk aangegeven dat je die gebruikt om lang commentaar in de code op te nemen, maar feitelijk komt het erop

neer dat je dan een lange string midden in je programma zet. Zo'n string doet niks, tenzij je hem aan een variabele toekent. Hier is een voorbeeld van een string met drievoudige dubbele aanhalingstekens:

listing1002.py

```
langestring = """En je wordt onderworpen aan een eindeloze stroom
Hotel Miramars en Bellevues en Continentaals met hun moderne luxe
internationale studio's en Heineken van de tap en zwembaden vol
vette Duitse zakenlui die denken dat ze acrobaten zijn en
pyramides vormen en de kinderen bang maken en voordringen in de
rij en als je niet aan tafel zit om klokslag zeven dan mis je je
kop Knorr Romige Tomatensoep, het eerste item op het menu van de
Internationale Cuisine, en iedere donderdagavond is er een
amateuristisch theater in de bar, met een kleine verwijfde
Spanjool met smalle heupjes en een opblazen taart met haar haar
platgeboterd en een enorm achterwerk die Flamenco voor
Buitenlanders presenteren."""
print( langestring )
```

Het opvallende verschil tussen deze twee voorbeelden is dat in het eerste voorbeeld de string beschouwd werd als een lange, doorlopende serie tekens, terwijl in het tweede voorbeeld de verschillende regels op meerdere regels geprint wordt. De reden dat dat gebeurt in het tweede voorbeeld is dat er een onzichtbaar teken staat aan het einde van iedere regel, dat aangeeft dat Python naar een nieuwe regel moet gaan voordat verder geprint wordt. Dit is een zogeheten "newline" teken, en je kunt het expliciet in een string opnemen, door gebruik te maken van de code `"\n"`. Deze code moet je niet lezen als twee tekens, de backslash en de `"n"`, maar als een enkel "newline" teken. Je kunt met dit teken ervoor zorgen dat de output over meerdere regels geprint wordt. Dat kan zelfs als je de backslash aan het einde van een regel zet om aan te geven dat de string over meerdere regels verspreid is, als in het eerste voorbeeld. Bijvoorbeeld:

listing1003.py

```
langestring = "En een paar nasale secretaresses uit Middelburg\n\
met flubberende witte benen en buikloop die flirten met harige\n\
krombenige Spaanse obers genaamd Manuel en eens per week is er\n\
een excursie naar de plaatselijke Romeinse bouwval waar je\n\
cola kunt kopen en gesmolten ijsjes en natuurlijk Heineken en\n\
op een avond bezoek je het zogenaamde typische restaurant met\n\
rustieke uitstraling en plaatselijke atmosfeer en je zit naast\n\
een groepje toeristen uit Amstelveen die maar blijven zingen\n\
'Torremolinos, torremolinos' en klagen over het eten - 'Het is\n\
erg vettig hier, vind je niet?' - en je wordt in een hoek\n\
gedreven door een dronken groentenboer uit Hilversum met zijn\n\
instant fototoestel en plastic sandalen en het Algemeen\n\
Dagblad van afgelopen dinsdag en hij zaagt maar door en door\n\
en door over hoe meneer Jansen dit land moet besturen en\n\
hoeveel talen Frits Bolkestein wel niet spreekt en dan kotst\n\
hij zijn avondeten uit over de cocktails."
print( langestring )
```

Dit betekent dat als je niet die automatische "newlines" wilt hebben in een string die meerdere regels beslaat, je de eerste aanpak moet gebruiken, met de backslash aan het einde van iedere regel. Als je wel meerdere regels wilt, dan is de tweede aanpak waarschijnlijk het beste leesbaar.

10.3 Speciale tekens

`"\n"` is een "speciaal teken" (in het Engels heet dit een "escape sequence"). Speciale tekens in Python worden over het algemeen geschreven als een backslash gevolgd door een code. De code kan één of meerdere tekens beslaan. Python interpreteert zulke speciale tekens, als ze in een string staan, niet letterlijk.

Naast het "newline" teken `"\n"`, heb ik in hoofdstuk 3 ook de speciale tekens `"\'"` en `"\""` geïntroduceerd, die je kunt gebruiken om een enkel respectievelijk dubbel aanhalingsteken in een string op te nemen, ongeacht het type aanhalingstekens dat je om de string heen hebt gezet. Ik heb ook genoemd dat je `"\\"` kunt gebruiken om een "echte" backslash in de string op te nemen.

Naast deze zijn er nog diverse andere speciale tekens. De meeste zijn behoorlijk archaïsch en worden niet meer gebruikt op moderne computers, dus die kun je negeren. De twee die ik nog wil noemen zijn `"\t"` die een tabulatie (inspringing) in de string representeert, en `"\x`*nn*`"` waarbij *nn* staat voor twee hexadecimale cijfers, die het hexadecimale getal *nn* representeren. Bijvoorbeeld, `"\x20"` is het teken dat gerepresenteerd wordt door het hexadecimale getal 20, dat hetzelfde is als het decimale getal 32, wat een spatie is (dit leg ik later in dit hoofdstuk verder uit).

Voor het geval je nooit hebt geleerd hoe je moet tellen met hexadecimale getallen: Hexadecimale getallen gebruiken 16 verschillende cijfers, namelijk 0 tot en met 9 en A tot en met F. Een directe vertaling van hexadecimale cijfers naar decimale getallen stelt dat A gelijk is aan 10, B aan 11, etcetera. In decimale getallen wordt de waarde van een getal dat uit meerdere cijfers bestaat berekend door de cijfers te vermenigvuldigen met oplopende machten van 10, van rechts naar links; bijvoorbeeld, het getal 1426 is $6 + 2 * 10 + 4 * 100 + 1 * 1000$. Voor hexadecimale getallen doe je hetzelfde, maar vermenigvuldig je de cijfers met oplopende machten van 16; bijvoorbeeld, het hexadecimale getal $4AF2$ is $2 + 15 * 16 + 10 * 256 + 4 * 4096$. Programmeurs gebruiken graag hexadecimale getallen, omdat computers als kleinste rekeneenheid de "byte" gebruiken, en een byte kan 256 verschillende waardes bevatten; met andere woorden, een byte kan iedere waarde bevatten die je kunt uitdrukken met precies twee hexadecimale cijfers.

Waarom het nuttig kan zijn te weten hoe je hexadecimaal moet tellen en waarom je tekens in een string hexadecimaal zou willen representeren volgt later in dit boek.

10.4 Tekens in een string

Ik heb al meerdere keren laten zien dat een string een verzameling is van tekens in een specifieke volgorde. Je kunt de individuele tekens van een string middels indices benaderen.

10.4.1 String indices

Ieder teken in een string heeft een positie, en die positie kun je weergeven door het index nummer van de positie. De indices beginnen bij 0 en lopen op tot aan de lengte van de string. Hieronder zie je het woord "python" op de eerste regel, met op de tweede en derde regel indices voor ieder teken in deze string:

```
 p  y  t  h  o  n
 0  1  2  3  4  5
-6 -5 -4 -3 -2 -1
```

Zoals je kunt zien, kun je positieve indices gebruiken die beginnen met 0 bij de eerste letter van de string, en die oplopen tot het einde van de string. Je kunt ook negatieve indices gebruiken, die starten met -1 bij de laatste letter van de string, en die aflopen totdat de eerste letter van de string bereikt is.

De lengte van een string `s` kun je berekenen met **`len`**`(s)`; de laatste letter van de string heeft dus index **`len`**`(s)-1`. Met negatieve indices heeft de eerste letter van de string de index `-`**`len`**`(s)`.

Als een string is opgeslagen in een variabele, dan kun je de individuele letters van de string benaderen via de variabele naam en de index van de gevraagde letter tussen vierkante haken ([]) rechts ernaast.

```
fruit = "aardbei"
print( fruit[4] )
print( fruit[2] )
print( fruit[1] )
print( fruit[-4] )
print( fruit[-2] )
print( fruit[-5] )
print( fruit[-1] )
print( fruit[5] )
```

Je mag ook variabelen als indices gebruiken, en zelfs berekeningen of functie-aanroepen. Je moet er echter altijd voor zorgen dat berekeningen leiden tot integers, want floats kunnen niet als indices gebruikt worden. Hieronder staan een paar voorbeelden, waarvan de meeste zo ingewikkeld zijn dat ik geen reden zie om ze op deze manier in een programma te zetten. Maar ze laten zien wat de mogelijkheden zijn.

```
from math import sqrt

fruit = "aalbes"
x = 3

print( fruit[3-2] )
print( fruit[int( sqrt( 4 ) )] )
print( fruit[2**2] )
print( fruit[int( (x-len( fruit ))/3 )] )
print( fruit[-len( fruit )])
print( fruit[-x] )
```

In principe mag je een index ook gebruiken bij een string die niet in een variabele staat, bijvoorbeeld, `"aalbes"[3]` is de letter `"b"`. Het mag duidelijk zijn dat niemand dat ooit doet.

Naast enkele indices om letters in een string te benaderen, kun je ook substrings van een string benaderen door twee getallen tussen vierkante haken te zetten met een dubbele punt (:) ertussen. De eerste van deze getallen is de index waar de substring start, de tweede waar de substring eindigt. De substring is exclusief de letter die hoort bij de tweede index. Door het linkergetal weg te laten geef je aan dat de substring begint bij de start van de string (dus bij index 0). Door het rechtergetal weg te laten geef je aan dat de substring eindigt met het laatste teken van de string (inclusief dit laatste teken).

Als je probeert een teken van een string te benaderen met een index die buiten de string valt, krijg je een runtime error ("index out of bounds"). Als je een substring probeert te benaderen geldt die beperking niet; het is toegestaan om getallen te gebruiken die buiten het bereik van de string vallen.

```
fruit = "aalbes"
print( fruit[:] )
print( fruit[0:] )
print( fruit[:6] )
print( fruit[:100] )
print( fruit[:len( fruit )] )
print( fruit[1:-1] )
print( fruit[2], fruit[1:6] )
```

10.4.2 Strings doorlopen

Ik heb eerder uitgelegd hoe je de tekens van een string kunt doorlopen middels een **for** loop:

```
fruit = 'appel'
for teken in fruit:
    print( teken, '- ', end='' )
```

Nu je indices begrijpt, realiseer je je waarschijnlijk wel dat je die ook kunt gebruiken om een string te doorlopen:

listing1004.py

```
fruit = 'appel'

for i in range( 0, len( fruit ) ):
    print( fruit[i], "- ", end="" )
print()

i = 0
while i < len( fruit ):
    print( fruit[i], "- ", end="" )
    i += 1
```

Als je voldoende hebt aan toegang krijgen tot de individuele tekens in de string, is de eerste methode, waarbij de constructie **for** <teken> **in** <string> wordt gebruikt, verreweg het meest elegant en leesbaar. Maar soms moet je een probleem oplossen waarbij een andere methode nodig is.

Opgave Schrijf een programma dat van een string de indices print van alle klinkers (a, e, i, o, en u). Dit kan met een **for** loop of een **while** loop, maar de **while** lijkt iets geschikter.

Opgave Schrijf een programma dat twee strings gebruikt. Voor ieder teken in de eerste string dat in de tweede string precies hetzelfde teken heeft met precies dezelfde index, druk je het teken en de index af. Pas op voor een "index out of bounds" runtime error. Test met de strings `"The Holy Grail"` en `"Life of Brian"`.

Opgave Schrijf een functie die een string als argument krijgt, en die dan een nieuwe string retourneert die hetzelfde is als het argument, maar waarbij ieder teken dat geen letter is vervangen is door een spatie (bijvoorbeeld, de uitdrukking `"ph@t l00t"` wordt gewijzigd in `"ph t l t"`). Om zo'n functie te schrijven begin je met een lege string, en doorloopt de tekens van het argument één voor één. Als je een acceptabel teken tegenkomt, voeg je het toe aan de nieuwe string. Anders voeg je een spatie toe aan de nieuwe string. Je kunt testen of een teken acceptabel is met eenvoudige vergelijkingen, bijvoorbeeld, alle kleine letters kun je herkennen omdat ze de test `ch >= 'a' and ch <= 'z' True` maken.

10.4.3 Substrings met stappen

Substrings kunnen behalve een index voor begin en einde een derde argument krijgen, namelijk stapgrootte. Dit argument werkt equivalent aan het derde argument voor de **range**() functie. De syntax voor substrings is <string>[<begin>:<einde>:<stap>]. Indien niet opgegeven, is de stapgrootte 1.

Een veelgebruikte toepassing van de stapgrootte is het gebruik van een negatieve waarde om de string te inverteren.

```
fruit = "banaan"
print( fruit[::2] )
print( fruit[1::2] )
print( fruit[::-1] )
print( fruit[::-2] )
```

Het inverteren van een string via `[::-1]` is conceptueel gelijk aan het doorlopen van de string vanaf het laatste teken tot het eerst met achterwaartse stappen van grootte 1.

```
fruit = "banaan"
print( fruit[::-1] )
for i in range( 5, -1, -1 ):
    print( fruit[i] )
```

10.5 Strings zijn onveranderbaar

Een kerneigenschap van strings is dat ze onveranderbaar (Engels: "immutable") zijn. Dit betekent dat strings niet kunnen wijzigen. Bijvoorbeeld, je kunt niet een teken in een string wijzigen door er een nieuwe waarde aan toe te kennen. Ter demonstratie: de volgende code leidt tot een runtime error als je hem probeert uit te voeren:

```
fruit = "aaldbei"
fruit[2] = "r"  # Runtime error!
print( fruit )
```

Als je een wijziging wilt maken in een string, moet je een nieuwe string maken die de wijziging omvat; je kunt daarna de nieuwe string toekennen aan de bestaande variabele als je wilt. Bijvoorbeeld:

```
fruit = "aaldbei"
fruit = fruit[:2] + "r" + fruit[3:]
print( fruit )
```

De reden waarom strings onveranderbaar zijn, is te technisch om hier te bespreken. Onthoud alleen dat als je een string wilt wijzigen, je geen nieuwe waarde kunt toekennen aan een individueel teken uit de string. In plaats daarvan moet je de variabele die de string bevat geheel overschrijven.

10.6 `string` methodes

Er is een aantal methodes beschikbaar die ontworpen zijn om strings te bewerken. Al deze methodes worden toegepast op een string om een operatie uit te voeren. Omdat strings onveranderbaar zijn, zullen deze methodes nooit de string waarop ze werken wijzigen, maar ze retourneren in plaats daarvan een gewijzigde versie van de string.

Net als de **`format`**`()` methode die in hoofdstuk 5 besproken is, worden al de string methodes aangeroepen via de syntax `<string>.<methode>()`, met andere woorden, je specificeert de string waarop de methode moet werken, gevolgd door een punt, gevolgd door de methode. Je zult dit vanaf nu vaker tegenkomen, en de reden dat methodes op deze manier aangeroepen moeten worden volgt in latere hoofdstukken (20 en verder).

De meeste string methodes zijn geen deel van een module, en je kunt ze aanroepen zonder iets te moeten importeren. Er is een `string` module die bepaalde nuttige constanten en methodes bevat die je in je programma's kunt gebruiken, maar de methodes die ik hier noem kun je gebruiken zonder de `string` module te importeren.

10.6.1 `strip()`

`strip()` verwijdert spaties aan het begin en einde van een string, inclusief eventuele "newline" tekens en andere tekens die als spaties gezien kunnen worden. Als je iets anders dan spaties wilt verwijderen, kun je als parameter een string meegeven die bestaat uit alle te verwijderen tekens.

```
s = "    En nu iets heel anders \n    "
print( "["+s+"]" )
s = s.strip()
print( "["+s+"]" )
```

10.6.2 upper() en lower()

upper() creëert een versie van een string met alle letters als hoofdletters. lower() werkt op dezelfde manier, maar maakt van alle letters kleine letters. Geen van beide methodes heeft parameters.

```
s = "The Meaning of Life"
print( s )
print( s.upper() )
print( s.lower() )
```

10.6.3 find()

find() kun je gebruiken om in een string te zoeken naar de start-index van een bepaalde substring. Als parameter krijgt de methode de gezochte substring. Optioneel kan een tweede, numerieke parameter meegegeven worden die aangeeft bij welke index gestart moet worden met zoeken. Een optionele derde, numerieke parameter is de index waarbij het zoeken moet stoppen. Je krijgt de laagste index waarbij de substring gevonden wordt terug, of -1 als de substring niet voorkomt.

```
s = "Humpty Dumpty zat op de muur"
print( s.find( "zat" ) )
print( s.find( "t" ) )
print( s.find( "t", 12 ) )
print( s.find( "q" ) )
```

10.6.4 replace()

replace() vervangt alle instanties van een substring in een string door een andere substring. Als parameters krijgt het de substring die gezocht wordt, en de substring die als vervanging dient. Optioneel kan een derde, numerieke parameter meegegeven worden die aangeeft hoe vaak een vervanging moet plaatsvinden.

Ik wil hier nogmaals benadrukken dat strings onveranderbaar zijn, dus de replace() functie maakt niet echt vervangingen in de string; hij retourneert een nieuwe string die een kopie is van de originele string, waarbij de vervangingen zijn gemaakt.

```
s = 'Humpty Dumpty zat op de muur'
print( s.replace( 'zat op', 'viel van' ) )
```

10.6.5 split()

split() splitst een string op in woorden, gebaseerd op een gegeven teken of substring die als separator beschouwd wordt. De separator is een parameter, en als die niet is opgegeven, is de separator de spatie, wat inhoudt dat je een string inderdaad opsplitst in de afzonderlijke woorden (waarbij interpunctie die aan woorden vastzit beschouwd wordt als een onderdeel van de desbetreffende woorden). Als de separator meerdere keren naast elkaar staat, dan worden de extra separatoren genegeerd (dat wil zeggen dat met spaties als separator, het niet uitmaakt of er tussen twee woorden één spatie staat, of meerdere).

Het resultaat van deze opsplitsing is een "lijst" van woorden. Lijsten komen aan bod in hoofdstuk 12, dus ik ga er nu weinig over zeggen. Ik zeg alleen dat als je de afzonderlijke woorden in de lijst wilt benaderen, je de constructie **for** <woord> **in** <lijst> kunt gebruiken.

```
s = 'Humpty Dumpty      zat   op de muur   '
lijst = s.split()
for woord in lijst:
    print( woord )
```

Een nuttige toepassing van het opsplitsen van een string is dat je het kunt gebruiken om sommige basale bestandsformaten te decoderen. Bijvoorbeeld, een CSV (Comma-Separated Value) bestand is een eenvoudig formaat, waarbij iedere regel van het bestand bestaat uit waardes die van elkaar gescheiden zijn door komma's. De waardes kun je uit een regel halen middels de split() methode.[13]

```
csv = "2016,september,28,Data Processing,Tilburg University"
waardes = csv.split( ',' )
for waarde in waardes:
    print( waarde )
```

10.6.6 join()

join() is de tegenhanger van split(). join() plakt een lijst van woorden aaneen tot een string, waarbij de woorden in de string van elkaar gescheiden zijn middels een specifieke separator. Dit klinkt wellicht alsof dit een methode van lijsten zou moeten zijn, maar om historische redenen is het gedefinieerd als een string methode. Omdat alle string methodes worden aangeroepen als <string>.<methode>(), moet er een string staan voor de aanroep van join(). Die string is de separator die je wilt gebruiken, terwijl de parameter die je meegeeft de lijst is waarvan je de woorden aan elkaar wilt plakken. De retourwaarde is, als altijd, de resulterende string.

```
s = "Humpty;Dumpty;zat;op;de;muur"
lijst = s.split( ';' )
s = " ".join( lijst )
print( s )
```

[13] In werkelijkheid is het vaak iets ingewikkelder omdat er komma's kunnen staan in de waardes die zijn opgeslagen in het CSV bestand, dus het is afhankelijk van de inhoud van het bestand of de genoemde aanpak werkt. Ik ga meer zeggen over CSV bestanden in hoofdstuk 26.

10.6.7 Oefening

Opgave In de string `"Barbara had een bar, waar ze rabarbar verkocht, en die daarom de rabarbarbarbarabar werd genoemd."` is het woord `"rabarber"` verkeerd gespeld. Gebruik `replace()` om alle voorkomende gevallen van deze fout te verbeteren.

Opgave Neem de string `"Niemand verwacht de Spaanse Inquisitie!# In feite, zij die de Spaanse Inquisitie wel verwachten..."` en toon hem tot aan, maar niet inclusief, de hash mark (#). Gebruik `find()` om de index van de hash mark te bepalen.

Opgave Schrijf een programma dat een "schone" versie van alle woorden in de string print. Alle tekens die geen letter zijn, worden niet beschouwd als deel van een woord, maar als separator. Alle letters moeten in kleine letters worden omgezet. Bijvoorbeeld, de string `"Ik heb zo'n honger."` produceert vijf woorden, namelijk `"ik"`, `"heb"`, `"zo"`, `"n"`, en `"honger"`. Je kunt de functie die je eerder hebt geschreven voor het schoonmaken van strings hier gebruiken.

10.7 Codering van tekens

Alle computersystemen gebruiken een manier om tekens te coderen. De basis codering die door (vrijwel) ieder systeem ondersteund wordt is de standaard ASCII code. Dit is een 7-bits code, die 128 verschillende tekens kan weergeven. Een aantal van deze tekens (met name die met de laagste nummers) zijn controle tekens die een speciale functie hebben. De meeste hiervan zijn alleen nuttig voor ouderwetse computersystemen, maar de tabulatie, "newline," en "backspace" tekens zitten er ook tussen. Als je alleen de tekens gebruikt die op een toetsenbord met US configuratie voorkomen, beperk je je tot standaard ASCII tekens.

Vandaag de dag gebruiken veel systemen Unicode. Unicode ondersteunt veel meer tekens. Er zijn verschillende manieren om tekens op te slaan als Unicode tekens. De meest bekende is UTF-8, die één byte gebruikt voor ieder van de ASCII tekens, maar meerdere bytes voor alle andere tekens (een byte is een groep van 8 bits, waarbij iedere bit een 1 of een 0 bevat). Andere Unicode coderingen gebruiken meerdere bytes voor alle tekens. Python ondersteunt UTF-8, wat betekent dat het ook reguliere ASCII coderingen ondersteunt.

10.7.1 ASCII

Hieronder zie je de ASCII tabel. De enige tekens die ik heb weggelaten zijn de speciale tekens. Die hebben nummers nul tot en met 31, en 127. 32 is de spatie. Ik geef ook de hexadecimale code voor ieder teken weer naast de decimale code. Die worden in een later hoofdstuk relevant.

Zoals je kunt zien heeft ieder teken een getal. Om in een Python programma te achterhalen wat het nummer is van een teken, kun je de **`ord`**`()` functie gebruiken. **`ord`**`("A")`, bijvoorbeeld, retourneert het nummer van `"A"`, dat 65 is, zoals je kunt zien. De tegenhanger van **`ord`**`()` is de **`chr`**`()` functie. **`chr`**`()` krijgt een nummer als argument, en retourneert het teken dat hoort bij dat nummer. Bijvoorbeeld, **`chr`**`(65)` is de letter `"A"`.

DC	HX		DC	HX		DC	HX		DC	HX		DC	HX	
32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o			

Een vergelijking tussen strings waarin alleen deze tekens gebruikt worden, gebruikt de nummers van de teken om te bepalen welke van de strings “kleiner” is. Bijvoorbeeld, de string `"mango"` is groter dan de string `"mangaan"`, omdat het eerste verschil tussen de strings het vierde teken is, wat `"o"` is voor `"mango"` en `"a"` voor `"mangaan"`. Omdat het nummer voor `"o"` groter is dan het nummer voor `"a"`, wordt de string `"mango"` beschouwd als groter dan de string `"mangaan"`. Dit is feitelijk een alfabetische vergelijking. Als er tekens in de string voorkomen die geen letters zijn, kun je in de ASCII tabel nakijken welke als kleiner beschouwd worden. Zie hoe alle cijfers kleiner zijn dan letters.

```
print( ord( 'A' ) )
print( ord( 'a' ) )
print( chr( 65 ) )
print( chr( 97 ) )
print( "mango" > "mangaan" )
```

Je kunt deze nummers en tekens die ermee geassocieerd zijn gebruiken in allerhande nuttige berekeningen. Bijvoorbeeld, als je wilt weten welke de twaalfde letter na de `"g"` is, kun je dat als volgt berekenen:

```
print( "De twaalfde letter na g is", chr( ord( "g" )+12 ) )
```

Om een uitgebreider voorbeeld van wat je kunt doen met codes voor tekens te laten zien, is hier een programma dat de ASCII tabel genereert als matrix:

listing1005.py

```
print( ' ', end='' )
for i in range(16):
    if i < 10:
        print( ' '+chr( ord( '0' )+i ), end='' )
    else:
        print( ' '+chr( ord( 'A' )+i-10 ), end='' )
print()
```

```
for i in range( 2, 8 ):
    print( i, end='' )
    for j in range( 16 ):
        c = i*16+j
        print( ' '+chr( c ), end='' )
    print()
```

Ik prefereer het als je de functies **ord**() en **chr**() gebruikt in een programma waar je de codes van tekens moet gebruiken. Als je wilt refereren aan de code voor de letter "A", schrijf dan niet 65, maar schrijf in plaats daarvan **ord**("A"). 65 heeft alleen betekenis voor mensen die ASCII codes van buiten kennen, en je programma zou betekenisvol moeten zijn voor iedereen. Daarbovenop komt nog dat, hoewel ASCII een zeer wijd verbreide standaard is, er nog steeds computers zijn die andere manieren van het coderen van tekens gebruiken, dus de code voor "A" is niet noodzakelijkerwijs 65 (inderdaad, IBM, ik heb het over jou).

10.7.2 UTF-8

Python ondersteunt Unicode, specifiek de meeste gebruikelijke versie van Unicode, namelijk UTF-8. Dit betekent dat je allerhande "vreemde" tekens kunt gebruiken. Ik legde uit bij de beschrijving van functie en variabele namen dat je "letters" in die namen kunt gebruiken. Je nam toen waarschijnlijk aan dat ik "A" tot en met "Z" en "a" tot en met "z" bedoelde. Het grappige is dat het afhankelijk is van je computersysteem wat daadwerkelijk beschouwd wordt als letter. Bijvoorbeeld, als in je computer staat ingesteld dat de taal die je gebruikt Duits is, dan kun je letters gebruiken met umlauts. Ook het Nederlands heeft speciale letters. Ik raad je echter ten zeerste af dit soort speciale letters te gebruiken in namen voor functies en variabelen. Niet alleen zijn ze lastig te typen, maar ze maken ook je programma minder goed overdraagbaar naar andere systemen.

In UTF-8 kun je in strings de reguliere tekens opnemen precies als je zou verwachten. Je kunt ook speciale letters opnemen, maar die zien er meestal anders uit dan je zou verwachten. Omdat Python UTF-8 ondersteunt, moet je voorzichtig zijn als je teksten kopieert van, bijvoorbeeld, een tekstverwerker. Tekstverwerkers hebben de irritante gewoonte om tekens te veranderen in andere tekens, bijvoorbeeld "rechte" aanhalingstekens in "kromme," of een min-teken in een "dash." Als je zulke tekens kopieert in je programma, zal Python de tekens accepteren, maar zal ze dan niet beschouwen als, bijvoorbeeld, string begrenzingen.

Als je Unicode tekens in een string wilt opnemen, kun je dat doen met Unicode codes. Je moet dan het UTF-8 nummer van het teken kennen dat je wilt tonen. Je kunt dan de code \u*xxxx* opnemen, waarbij *xxxx* een hexadecimaal getal van vier hexadecimale cijfers is, om het corresponderende teken in de string te zetten. Bijvoorbeeld, de onderstaande code toont het Griekse alfabet:[14]

```
alpha = "\u0391"
for i in range( 25 ):
    print( chr( ord( alpha )+i ), end=" " )
```

[14] Er staat een vreemd teken in deze reeks Griekse letters, namelijk tussen de Rho aen de Sigma, dat gelijk is aan \u03A2, dat klaarblijkelijk geen legaal Unicode teken is.

Over het algemeen hoef je je niet druk te maken over teken coderingen. Ik raad je aan je te beperken tot ASCII waar mogelijk. Als je met Unicode tekens moet werken, gaan de zaken meestal automatisch goed, omdat Python Unicode ondersteunt. Af en toe zie ik vertalingsproblemen als van Unicode naar ASCII moet worden gegaan, wat meestal te maken heeft met bestandsverwerking. Het zal een tijdje duren voordat je dat soort problemen krijgt, en ik zal er meer over zeggen in hoofdstuk 16 en verder.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Strings
- Strings over meerdere regels
- Positive and negative indices voor strings
- Substrings
- Onveranderlijkheid van strings
- String methodes `strip()`, `upper()`, `lower()`, `find()`, `replace()`, `split()`, en `join()`
- Speciale tekens
- ASCII en UTF-8 coderingen

Opgaves

Opgave 10.1 Tel hoeveel er van iedere klinker (a, e, i, o, u) staan in een tekst string, en druk die teller af voor iedere klinker met een enkele geformatteerde string. Bedenk dat klinkers zowel hoofd- als kleine letters kunnen zijn.

Opgave 10.2 Hieronder staat een tekst met een aantal tekens tussen vierkante haken. Doorloop de tekst en druk alle tekens af die tussen vierkante haken staan.

exercise1002.py

```
tekst = """En ze stu[re]n [i]ngekleurde prentbriefkaarten van
plekken waarvan ze zich niet reali[s]eren dat ze er nooit
geweest zijn [a]an 'Iedereen op nummer 22, weer is prachti[g],
onz[e] kamer is aa[n]gekruisd. Missen jullie. E[t]en[ ]i[s]
vettig, maar we hebben een geweldig leuk restaurantje gevonden
in de achterstraatjes waar ze Heine[ke]n hebben en kaas en
uien chips en iemand die "Een beetje verliefd" speel[t] op een
a[c]cordeon' en je zit vier dagen vast op Schip[h]ol voor je
vijfdaagse vliegvakantie met niks anders te eten dan
uitgedroogde voorverpakte boterhammen..."""
```

Opgave 10.3 Druk een regel af met alle hoofdletters `"A"` tot en met `"Z"`. Druk eronder een regel af die of 13 letters afstand in het alfabet liggen ten opzichte van de letters erboven. Bijvoorbeeld, onder de `"A"` druk je de `"N"` af, onder de `"B"` druk je de `"O"` af, etcetera. Beschouw het alfabet als circulair, dat wil zeggen, na de `"Z"`, gaat het weer terug naar de `"A"`. Dit kan natuurlijk met twee print-commando's, maar probeer het te doen met loops en gebruik te maken van **`ord`**`()` en **`chr`**`()`.

Opgave 10.4 Tel in de tekst hieronder hoe vaak het woord "knap" voorkomt (via een programma, natuurlijk). Zowel hoofd- als kleine letters mogen worden gebruikt, en je moet wel bedenken dat het woord "knap" een zelfstanding woord moet zijn, en niet een deel van een ander woord. Hint: Als je netjes alle oefeningen hebt gedaan tot nu toe, heb je al een functie gebouwd die schone woorden uit een tekst haalt.

exercise1004.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 10.5 Schrijf een programma dat een string neemt en er een nieuwe string van maakt die precies dezelfde tekens bevat als de originele string, maar in volgorde van hun ASCII codes. Bijvoorbeeld, de string `"Hello, world!"` geeft als resultaat de string `" !,Hdellloorw"`. Dit kan vrij gemakkelijk gedaan worden met "list" functies, maar die komen pas aan bod in hoofdstuk 12, dus probeer het nu te doen met string manipulatie.

Opgave 10.6 Typische autocorrectie maakt de volgende wijzigingen: (1) als een woord begint met twee hoofdletters gevolgd door een kleine letter, dan wordt de tweede hoofdletter gewijzigd in een kleine letter; (2) als een zin een woord bevat dat onmiddellijk gevolgd wordt door hetzelfde woord, dan wordt het tweede woord verwijderd; (3) als een zin begint met een kleine letter, dan wordt die gewijzigd in een hoofdletter; (4) als een woord volledig uit hoofdletters bestaat, behalve de eerste letter die een kleine letter is, dan wordt de kleine letter een hoofdletter en alle hoofdletters kleine letters; en (5) als de zin de naam van een dag bevat die niet met een hoofdletter begint, dan wordt de eerste letter in een hoofdletter veranderd. Schrijf een programma dat een zin neemt en die volgens deze autocorrectie regels aanpast. Je kunt het met de string hieronder testen.

exercise1006.py

```
zin = "en zo gebeurde het dat dat onze toevallige ontmoeting \
met de EErwaarde aRTHUR BElling een ommekeer betekende in ons \
leven, en vanaf dat moment gingen we iedere zondag naar \
de kerk van Sint sIMPEL bij Roombroodje MEt Jam."
```