**Technieken dag 3**

**Programma ontwerp**

Het programma is nu al doende tot stand gekomen. Bij een iets groter programma is dit al snel verwarrend. Dan is een eerste ontwerpfase nodig.

Programma ontwerp kan *top down*, *bottom up* of met een tussenvorm van beide technieken plaatsvinden.

Bij top down maak je eerst een “schets” van de hoofdtaken van het programma. Daarna ga je deze functies uitwerken (in Python heet het allemaal “functie”, in andere talen b.v. “subroutine” of “procedure”.

Bij *bottom up* focus je eerst op het uitwerken en testen van bepaalde deelfuncties die je meteen al onderkent. Hier is dat bijvoorbeeld:

* verdeel de invoerregels in twee helften
* zoek het gemeenschappelijke element
* bereken de waarde daarvan

Meestal werk je in een mengvorm: stukje hoofdlijn, functie uitwerken, nog een stukje hoofdlijn, enz. Vaak verschuiven al doende sommige acties van de hoofdlijn naar een aparte functie.

**Doorsnede (intersection) van lijsten**

In deze puzzel kun je experimenteren met twee manieren om gemeenschappelijke elementen in twee lijsten op te sporen: via sets of direct in de lijsten in een *for loop*.

De set doorsnede (m.b.v. de *method* “intersection” of de “&” operator) levert een set op met unieke elementen (in dit geval: 1 uniek element, omdat er maar 1 letter in beide delen voorkomt).

De *loop* over de lijsten geeft een lijst van **alle** (occurrences van) gemeenschappelijke elementen. (sommige invoerregels bevaten links en rechts meerdere malen het gemeenschappelijke element).

Voordeel van het gebruik van de “set1 & set2” methode is performance: dit is vele malen sneller dan een zelfgemaakte *loop* over de lijsten, omdat het onderliggende mechanisme direct is geïmplementeerd in machinecode.

Zie apart testprogramma om de performance van beide te vergelijken   
  
Als je *alle* occurrences wilt vinden, dan is een set onbruikbaar, omdat die automatisch ontdubbelt. Je moet dan toch rechtstreeks de lijsten vergelijken. Als je dat hier doet, krijg je soms een lijst met één of meer herhalingen van dezelfde letter.

**Slicing**

Bij het opdelen van een rugzak vind ik slicing m.b. v. de hulpvariabele “eindPos” leesbaarder.

In mijn oplossing is de eerste “0” en de laatste “len(regel)” overbodig.

**Functie die meer dan 1 waarde aflevert**

De functie “deelRugzak” levert twee waarden op: “linkerDeel” en “rechterDeel”.

Bij aanroep kun je die beide afzonderlijk laten vullen:

linkerDeel, rechterDeel = deelRugzak(regel)

Je kunt ook 1 variabele ermee laten vullen:

beideDelen = deelRugzak(regel)

Het resultaat is dan een lijst met twee sublijsten, waar de afzonderlijke delen in staan.

**ASCII codes**

Om de waarde van een letter (“prioriteit” in de puzzel) te berekenen kun je eenvoudig de ASCII codes gebruiken met de functie “ord”:

ascii = ord(letter)

if ascii > 96:

prio = ascii - 96

else:

prio = ascii - 64 + 26

(mijn berekening is iets omslachtiger, omdat ik in eerste instantie uitging van hoofdletters eerst).

**Dictionary**

De waarde van de letters opzoeken in een lijst met [key, waarde] paren is een aardige oefening in lijstgebruik.

Anderzijds is dan de *dictionary* een meer geëigende oplossing: priority = {‘a’:1, ‘b’:2, .... ‘Z’:52}.

Een dictionary werkt als een geïndexeerde tabel. Met de key kun je de gezochte waarde direct opzoeken. In deze puzzel ziet dat er zo uit:

priority = {'a':'1','b':'2','c':'3','d':'4','e':'5','f':'6' \

,'g':'7','h':'8','i':'9','j':'10','k':'11','l':'12' \

,'m':'13','n':'14','o':'15','p':'16','q':'17','r':'18' \

,'s':'19','t':'20','u':'21','v':'22','w':'23','x':'24' \

,'y':'25','z':'26','A':'27','B':'28','C':'29','D':'30' \

,'E':'31','F':'32','G':'33','H':'34','I':'35','J':'36' \

,'K':'37','L':'38','M':'39','N':'40','O':'41','P':'42' \

,'Q':'43','R':'44','S':'45','T':'46','U':'47','V':'48' \

,'W':'49','X':'50','Y':'51','Z':'52'}

In het hoofdprogramma kun je dan bijvoorbeeld direct zeggen:

totaal += **int(priority[item[0]])**

of:

priorityList.append( **int(priority[commonElementList[0]])** )

De dictionary kan handmatig worden ingetypt, maar je kunt hem ook eenvoudig genereren:

priority = dict() # definieer lege dictionary

waarde = 1 # a - z = 1 - 26

for letter in range(ord("a"), ord("z") + 1):

**priority[chr(letter)] = waarde**

waarde += 1

waarde = 27 # A - Z = 27 - 52

for letter in range(ord("A"), ord("Z") + 1):

**priority [chr(letter)] = waarde**

waarde += 1

print(priority)

***Loops* en tellers**

In vrijwel elk programma komt het voor: eén of meer acties wordt een aantal keren herhaald. Het meest gebruikte mechanisme om dit te doen is de ***loop (*of: de *lus).***

Er zijn twee hoofdtypen:

* de ***data driven loop***Hierbij ligt de conditie voor de herhaling in de gegevens die worden verwerkt. In Python is een bekend voorbeeld de hoofdverwerkingsloop:

for line in file:

De verwerking loopt door totdat er geen invoerregels meer zijn in “file”.

* de ***counter driven loop***

Hierbij hangt de uit te voeren actie af van de waarde van een teller. Daarbij kan van alles misgaan: de loop start te vroeg of te laat, eindigt te vroeg of te laat of de loop verwerkt de verkeerde gegevens. Deel 2 van dag 3 bevat een oefening om dit te illustreren, doordat de invoer in groepjes van 3 moet worden verwerkt.

Je kunt een *counter based loop* op verschillende manieren construeren. Eén belangrijke keuze is of je de teller met 0 of met 1 laat beginnen (*zero based indexing* of *one based indexing*)*.* Voor beide methodes zijn principiële voor- en tegenstanders te vinden, maar zolang je maar precies weet wat je doet zijn wat mij betreft beide varianten mogelijk. Je kunt op praktische gronden voor één van beide kiezen.  
  
Een andere balangrijke keuze is of je de teller **voor** de uit te voeren actieophoogt (*pre increment)* of **na** deactie (*post increment*). In combinatie met de zero based/one based dimensie levert dit een matrix op met interessante mogelijkheden om fouten te maken!