1 La programmation structurée

sur l'exemple du calcul de la date du lendemain...

Le problème s'énonce : on se donne une date sous la forme de trois entiers `(jour,mois,an)`, par exemple `(31,3,1995)` pour *le 31 mars 1995* et il faut que la fonction `lendemain` retourne le lendemain de ce jour, par exemple `(1,4,1995)` pour *le 1er avril 1995*.

La phrase qui précède définit ce que le programme doit faire. Une telle description est appelée la *spécification* du programme. Passer d'une spécification en langue naturelle à un programme d'ordinateur n'est pas toujours facile *a priori*, cependant dans presque tous les cas la technique de *programmation structurée* permet d'aboutir à une solution élégante sans trop de difficulté.

Face à un problème, la technique de programmation structurée consiste, d'une certaine façon, à toujours choisir la solution de facilité en premier lieu. Non seulement ce n'est pas désagréable..., mais en plus, après avoir « inversé l'ordre de programmation » comme on le verra plus loin, on aboutit à un style de programmation extrêmement clair et facile à lire.

Essayons donc. Comme déjà mentionné, il est souvent plus facile de commencer par éliminer les cas d'erreur car il n'y a pour eux aucun résultat à fournir. Ici c'est lorsque le triplet ne représente pas une date :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > ??? :
...         print("La date fournie n'existe pas !")
...     else :
...         .... la suite ....
```

Bon, on tombe sur une première difficulté (les `???`) car le nombre maximal de jours possibles dépend du mois et c'est un calcul compliqué...

C'est là qu'intervient la programmation structurée : ***c'est compliqué ? qu'à cela ne tienne, on suppose qu'il existe déjà une fonction qui résoud le problème !***

Dans l'exemple qui nous occupe, on suppose donc qu'il existe une fonction `nbjours` qui prend en entrée le numéro du mois et retourne en résultat le nombre de jours de ce mois. Dès lors, la gestion des cas d'erreur est résolue et passons à « `.... la suite ....` »

Là, il y a plusieurs cas, selon qu'on est en fin de mois ou pas, en fin d'année ou pas. Le plus facile est naturellement lorsqu'il suffit d'ajouter 1 au jour. La programmation structurée, qui repousse comme on l'a dit toutes les difficultés à plus tard, nous dit de commencer par ce cas le plus facile.

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
...         return( jour+1 , mois, an )
...     else :
...         .... la suite ....
```

Jusque là, on s'en est sorti sans mettre trop de jus de cerveau, on imagine donc que les difficultés vont commencer dans « `.... la suite ....` » (ou bien dans la programmation de `nbjours`). Puisqu'on est maintenant dans le `else` et que la date est correcte, c'est que nous sommes le dernier jour du mois. Bref, finalement, si nous ne sommes pas en décembre ce sera facile car il suffit d'ajouter 1 au mois et de revenir au premier du mois. La programmation structurée nous dit donc de commencer par ce cas facile.

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
```

```
...         return( jour+1 , mois, an )
...     elif mois < 12 :
...         return( 1 , mois+1 , an )
...     else :
...         .... la suite ....
```

Les difficultés seraient-elles derrière ce dernier `else` ? pas vraiment puisque derrière ce dernier `else` nous sommes nécessairement le 31 décembre, il suffit donc de passer au premier janvier de l'année d'après :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
...         return( jour+1 , mois, an )
...     elil mois < 12 :
...         return( 1 , mois+1 , an )
...     else :
...         return( 1 , 1 , an+1 )
```

C'était donc simple finalement : en passant en revue un à un tous les cas les plus faciles, on arrive à la conclusion qu'il n'y a pas de cas difficile !

OK, la difficulté sera donc sans doute de programmer `nbjours`... La démarche de programmation structurée consiste, de manière assez naturelle, à ré-appliquer la même technique. On commence par la spécification :

> Le sous-problème s'énonce : on se donne un mois sous forme d'un entier compris entre 1 et 12 et la fonction `nbjours` doit renvoyer le nombre de jours de ce mois.

Programmation structurée : on commence par les cas faciles, c'est-à-dire les mois de 31 jours et les mois de 30 jours :

```
>>> def nbjours(m) :
...     if m in [4,6,9,11] :
...         return(30)
...     elif m != 2 :
...         return(31)
...     else :
...         .... aie aie aie ....
```

Là, on découvre qu'on a été un peu optimiste : les autres mois étaient faciles mais le mois de février est insoluble en l'état car le nombre de jours du mois de février dépend de l'année.

Donc il nous manque une donnée ; il faut que `nbjours` prenne aussi l'année en argument ! Cela nous obligera à modifier la manière d'appeler la fonction `nbjours` dans la fonction `lendemain` :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois,an) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois,an) :
...         return( jour+1 , mois, an )
...     elil mois < 12 :
...         return( 1 , mois+1 , an )
...     else :
...         return( 1 , 1 , an+1 )
```

et reprenons la résolution du sous-problème de `nbjours`... Là encore, on cherche à ne faire que des choses faciles. Si l'année est bissextile, il y a 29 jours, sinon il y en a 28. Ce qui est difficile, c'est de savoir si l'année est bissextile.

```
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
```

```
...         return(31)
...     elif ??? :
...         return(29)
...     else :
...         return(28)
```

On commence à en avoir l'habitude maintenant : ***c'est compliqué ? qu'à cela ne tienne, on suppose qu'il existe déjà une fonction qui résoud le problème !***
On suppose donc qu'il existe une fonction `bissextile` qui résoud la question. Cette fonction devra renvoyer `True` si l'année est bissextile et `False` sinon.

```
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
...         return(31)
...     elif bissextile(a) :
...         return(29)
...     else :
...         return(28)
```

C'était donc simple finalement. On imagine donc que c'est la fonction `bissextile` qui sera difficile à programmer...

La programmation structurée nous dit de commencer par spécifier cette fonction. Là, on fait un clic sur wikipedia par exemple et on finit par spécifier :

> Le sous-sous-problème s'énonce : on se donne une année sous la forme d'un nombre entier positif et la fonction `bissextile` doit retourner un booléen qui dit si l'année est bissextile. Une année est bissextile :
> — si elle est divisible par 4
> — mais qu'elle n'est pas divisible par 100
> — sauf que si elle est divisible par 400 alors elle est quand même bissextile.

Heu... bon, la programmation structurée dit de commencer par les cas faciles : si une année n'est pas divisible par 4, on est sûr qu'elle n'est pas bissextile.

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     else :
...         .... la suite ....
```

Il ne reste donc plus qu'à traiter dans `la suite` que les années divisibles par 4. C'est facile quand l'année est divisible par 400 car elle est dans ce cas toujours bissextile.

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
...     else :
...         .... la suite ....
```

Maintenant, une fois traitées les années multiples de 400, les années multiples de 4 sont bissextiles sauf si elles sont divisibles par 100 :

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
```

```
...     if a % 100 == 0 :
...         return(False)
...     else :
...         return(True)
```

Evidemment, Python n'est pas capable « d'attendre » qu'on ait programmé `nbjours` et `bissextile` pour traiter `lendemain`. Il faut donc inverser l'ordre de programmation. On programme donc en sens inverse de la manière dont on a réfléchi [1] :

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
...     else :
...         return( not (a % 100 == 0) )
...
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
...         return(31)
...     elif bissextile(a) :
...         return(29)
...     else :
...         return(28)
...
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois,an) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois,an) :
...         return( jour+1 , mois, an )
...     elil mois < 12 :
...         return( 1 , mois+1 , an )
...     else :
...         return( 1 , 1 , an+1 )
```

Voilà, c'est résolu. La programmation structurée est fondée sur un mode de pensée plutôt confortable pour l'esprit : on commence par faire ce qui est simple, et on suppose déjà résolus (par des fonctions) les problèmes qui paraissent difficiles. On attaque ensuite chacune des fonctions qu'on a laissées de côté en suivant le même mode de pensée... et on finit par découvrir qu'on arrive au bout du problème sans jamais avoir eu à résoudre de problème très complexe !

Remarque subsidiaire : comparez le nombre de lignes du programme précédent et le nombre de millions de dollars qu'a coûté le fameux « bug de l'an 2000 »...

2 Les dictionnaires

Dans l'exemple de la date du lendemain précédent, il serait facile de connaître le nom d'un mois à partir de son numéro. En effet, si on considère la liste suivante :

```
>>> leMois = [ "janvier" , "février" , "mars" , "avril" , "mai , "juin" , "juillet" , "aout" ,
               "septembre" , "octobre" , "novembre" , "décembre" ]
```

et si la variable `mois` contient un entier compris entre 1 et 12, alors `leMois[mois-1]` fournit la chaîne de caractère donnant le nom de ce mois.

1. Notez la petite simplification de `bissextile` qu'on a faite au passage

Cependant, l'inverse est beaucoup moins pratique. Etant donné un nom de mois, trouver son numéro suppose de parcourir la liste, comme en TD avec la fonction `indice`. Dans l'idéal, il faut pouvoir *associer* à chaque nom de mois son numéro.

La structure de données qui mémorise cela est appelée un « dictionnaire » en Python et son type est `dict`.

L'ensemble des dictionnaires est constitué des expressions de la forme

$$\texttt{\{ } e_1\texttt{:}d_1 \texttt{ ,..., } e_n\texttt{:}d_n \texttt{ \}}$$

où les e_i sont des données *élémentaires* quelconques et les d_i sont des données quelconques. Par donnée élémentaire, on entend ici une donnée qui n'est pas elle-même une liste ou un dictionnaire.

Les e_i sont appelés les *éléments* du dictionnaires et les d_1 ne sont que les informations associées à ces éléments.

```
>>> numMois = { "janvier":1 , "Janvier":1 , "janv":1 , "janv.":1 ,
                "février":2 , "Février":2 , "fév":2 , "fev":2 , "fév.":2 , "fev.":2 ,
                ...etc... }
>>> numMois[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> numMois["fev"]
2
```

On constate que l'on n'accède pas au contenu d'un dictionnaire par un indice entier donnant la position dans le dictionnaire mais, et c'est plus simple, par le nom des éléments appartenant au dictionnaire.

ATTENTION : l'accès à un dictionnaire se fait par les éléments, pas par les informations qui lui sont associées.

On peut compléter ou modifier un dictionnaire avec une instruction d'affectation ; par exemple :

```
>>> coursesPrecises = {'baguette': 2, 'saumon': 2, 'vin': 1}
>>> coursesPrecises
{'baguette': 2, 'saumon': 2, 'vin': 1}
>>> coursesPrecises["saumon"]=1
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1}
>>> coursesPrecises["gateau"]=8
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1, 'gateau': 8}
```

On peut aussi supprimer un élément (et son information associée) avec `del`, obtenir le nombre d'éléments avec `len` et tester si un élément est dans le dictionnaire avec `in`.

```
>>> del coursesPrecises["vin"]
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'gateau': 8}
>>> len(coursesPrecises)
3
>>> "gateau" in coursesPrecises
True
>>> "chou" in coursesPrecises
False
```

Enfin, on peut *itérer* un bloc d'instructions sur tous les éléments d'un dictionnaire, un peu comme avec un `while` sur une chaîne de caractères ou sur une liste. Pour cela on utilise `for...in...`

```
>>> def total (d) :
...     s=0
```

```
...     for e in d :
...         s = s + d[e]
...     return(s)
...
>>> total(coursesPrecises)
11
```

BIS REPETITA : là encore, attention, les opérations d'affectation, `del`, `in` et `for` se basent sur les *éléments* des dictionnaires et non pas sur leurs *informations* associées !

Note : un dictionnaire avec des éléments ayant tous le même type et des informations ayant toutes le même type, c'est souvent mieux !