SYNTAXE, NOTIONS et ASTUCES :

Lecture/écriture rapide :

Lecture rapide

Jusqu'à présent, pour lire une ligne de l'entrée, vous utilisiez :

```
ligne = input()
```

Ensuite, vous pouviez la convertir en un ou plusieurs entiers ou tout autre action sur cette ligne de l'entrée.

Le problème est que la fonction *input* est très lente car elle gère des fonctionnalités avancées destinées à une interface en ligne de commande (donc sans intérêt dans nos exercices). Ainsi, pour certains des problèmes qui vont suivre, elle ne vous permettra pas d'être suffisamment efficace.

Il faut donc utiliser une fonction plus basique. Elle se trouve dans le module *sys*, qu'il faut importer :

```
import sys
```

Puis après, on utilise :

```
ligne = sys.stdin.readline()
```

Cette nouvelle fonction est jusqu'à **8 fois** plus rapide que la précédente ! Tout comme *input*, elle retourne la prochaine ligne sur l'entrée ; par contre elle laisse le retour à la ligne (`\n`) à la fin. Si vous convertissez la chaîne en une ou plusieurs valeurs (entiers, flottants), cela ne devrait pas vous déranger ; sinon, faites attention.

Notez que l'on peut aussi plus simplement remplacer la fonction *input* par *sys.stdin.readline* au tout début de son programme :

```
import sys
input = sys.stdin.readline
```

Écriture rapide

Il est possible de faire plus rapide que *print* mais on ne gagne que 50% de temps, ce qui n'est pas si intéressant. Cela peut cependant servir parfois ; ainsi, voici comment faire :

```
import sys

texte = "ABCDE"
entier = 42

sys.stdout.write("Texte : ")
sys.stdout.write(texte)
sys.stdout.write(str(entier) + "\n")
```

```
Texte : ABCDE42
```

Il faut donc utiliser *sys.stdout.write*. Cette fonction prend un et un seul paramètre (alors que *print* pouvait en prendre n'importe quelle quantité, et comportait les deux options *sep* et *end*) : la chaîne de caractères à afficher. Pour afficher une valeur, il faut donc la convertir en chaîne de caractères avec la fonction *str*. Si l'on veut afficher plusieurs textes, il faut appeler plusieurs fois la fonction, ou concaténer les chaînes de caractères à afficher. Notez que cette fonction n'insère pas de retour à la ligne à la fin de l'affichage comme le faisait *print* : il faut donc rajouter le `\n` soi-même si l'on s'en sert.

UTILISER une fonction main :

L'interpréteur Python met beaucoup de temps à exécuter des instructions qui se trouvent en dehors de toute fonction. Dans le programme suivant :

```
def dedans(nbFois):
    for numFois in range(nbFois):
        print("Dedans")

nbFois = int(input())
for numFois in range(nbFois):
    print("Dehors")
    dedans(numFois)
```

les instructions en dehors de la fonction peuvent mettre plus de temps à s'exécuter que celles qui se trouvent à l'intérieur ! En effet, le programme qui exécute les scripts Python effectue beaucoup plus de traitements et de mise en contexte pour les instructions du corps global (nous vous décrirons peut-être lesquelles un de ces jours).

Pour remédier à cela, il suffit de mettre les instructions du corps global dans une fonction, que l'on nommera *main* pour se conformer aux langages exigeant une telle fonction (C, C++ et Java par exemple). On appellera donc cette fonction dans le corps global, et ce sera la seule instruction s'y trouvant :

```
def dedans(nbFois):
    for numFois in range(nbFois):
        print("Dedans")

def main():
    nbFois = int(input())
    for numFois in range(nbFois):
        print("Dehors")
        dedans(numFois)

main()
```

Cela peut nettement accélérer certains programmes et vous sera nécessaire dans certains exercices.

Modification raccourcie :

Lorsque l'on veut modifier une variable en fonction de sa propre valeur, la notation est un peu lourde. Prenons par exemple cette instruction qui permet d'augmenter la variable *nombre* de 1, ce que l'on a fait tant de fois :

```
nombre = nombre + 1;
```

Le langage Python propose en fait un opérateur combinant l'addition et l'affectation : `+=`. Il existe également de tels opérateurs pour tous les calculs arithmétiques du langage (et même d'autres opérations). Nous vous en présentons ci-dessous ceux qui correspondent aux opérations que nous avons utilisées jusqu'à présent.

```
nombre += ajout
nombre -= retrait
nombre *= facteur
nombre **= exposante
nombre /= diviseur
nombre //= diviseur  # pour des entiers
nombre %= modulateur
```

Observez alors que si l'on écrit :

```
lapins *= lapins + agitation
```

pour appliquer la croissance d'une population de lapins, on multiplie *lapins* par *lapins* + *agitation*. C'est donc équivalent à une affectation avec des parenthèses :

```
lapins = lapins * (lapins + agitation)
```

Conclusion

Nous emploierons désormais ces opérateurs dans les cours et corrections. Nous vous encourageons à vous en servir également pour rendre votre code plus clair et plus concis.

OPERATIONS VECTORIELLES :

Concaténation rapide de liste/tableau

Le langage Python est optimisé pour traiter des listes de données globalement. Ainsi certaines opérations peuvent aller des dizaines, voir des centaines de fois plus vite si l'on en tient compte. Par exemple :

```
nbItems = 5*1000
liste = []
for item in range(nbItems):
   liste = liste + [item % 10]
```

est très lent alors que

```
nbItems = 5*1000
liste = [item % 10 for item in range(nbItems)]
```

ou

```
nbItems = 5*1000
liste = []
for item in range(nbItems):
   liste.append(item % 10)
```

est 100 fois plus rapide. Une alternative également rapide est :

```
nbItems = 5*1000
liste = [None] * nbItems
for item in range(nbItems):
   liste[item] = item % 10
```

Il faut comprendre que les tableaux en Python ne sont ni vraiment des tableaux, ni des listes, mais un compromis entre les deux. Ajouter un élément à une liste par `liste + [element]`, ne provoque pas seulement des allocations de mémoire mais la recopie du tableau/liste dans son intégralité.

Par contre, dans les deux cas rapides, tout se passe comme si la mémoire était allouée en une seule fois et un seul bloc. Il n'y a aucune recopie inutile. Dans la réalité, c'est plus compliqué mais nous ne souhaitons pas l'expliquer ici.

Itérateurs

On peut penser que ces méthodes sont possibles car on peut prévoir dès le départ la taille du tableau. Que faire si l'on ne peut prévoir cette taille à l'avance ?

Dans le cas où cela est raisonnable on peut commencer par sur-dimensionner le tableau pour ensuite le recopier en une seule fois à sa plus juste taille :

```
maxItems = 5*1000
listeTemp = [None] * maxItems
nbItems = 0
for item in range(maxItems):
   if item % 7 == 0:
      listeTemp[nbItems] = item % 10
      nbItems = nbItems + 1
liste = [None] * nbItems
for item in range(nbItems):
   liste[item] = listeTemp[item]
listeTemp = []
```

Il y a un moyen plus « pythonesque » d'obtenir exactement le même résultat, plus rapide et plus lisible.

```
maxItems = 5*1000
liste = [item % 10 for item in range(maxItems) if item % 7 == 0]
```

Et que faire si les éléments à mettre dans la liste sont difficiles à construire et à énumérer ? C'est là qu'intervient une des constructions les plus puissantes de Python, les *itérateurs*. Pour comprendre comment ils fonctionnent le mieux est de regarder un exemple.

```
def maListe(nbItems):
   for item in range(nbItems):
      if item % 7 == 0:
         yield item % 10

maxItems = 5*1000
liste = [item for item in maListe(maxItems)]
```

Ce programme calcule exactement la même liste que le précédent et aussi efficacement. Par contre le mécanisme est beaucoup plus puissant car l'itérateur `maListe()` peut être très compliqué. L'instruction `yield` en est la clef, elle ressemble à une instruction `return` à ceci près que lors de l'itération le code sera appelé plusieurs fois et reprendra juste après la dernière instruction `yield`. Cette instruction ne fait donc pas que renvoyer un résultat, elle mémorise également l'état d'exécution du calcul pour pouvoir ensuite le continuer !

Notons que l'on aurait pu remplacer la dernière ligne par le plus simple :

```
liste = list( maListe(maxItems) )
```

On utilise souvent la fonction `map` en liaison avec un itérateur pour en fabriquer un autre plus complexe. Ainsi le résultat de :

```
print( list( map( str, range(10) ) ) )
```

est

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Voyons cela : `range(10)` est un itérateur qui génère les entiers de 0 à 9, `map( str, range(10) )` applique la fonction `str` à tous ces entiers et est donc un itérateur qui génère les chaines de caractères de '0' à '9', finalement `list( map( str, range(10) ) )` construit une liste à partir de ces chaines.

Concaténation rapide de chaines de caractères

La concaténation des chaines de caractères pose le même problème que celle des listes. Cela est d'autant plus gênant que la fonction `print` est lente. On aimerait donc l'appeler rarement avec de longues chaines plutôt que souvent avec de petites chaines. Or si l'on fabrique une chaine assez longue avec de multiples utilisations de `+` cela prend encore plus de temps.

Il y a une solution à ce problème. L'utilisation de la fonction `join` qui construit une chaine de caractères à partir de fragments par un simple parcours. Ainsi

```
nbItems = 10*1000
for item in range(nbItems):
   print(item)
```

est lent et

```
nbItems = 10*1000
message = ""
for item in range(nbItems):
   message = message + str(item) + "\n"
print(message,end="")
```

est pire mais

```
nbItems = 10*1000
message = "\n".join( map( str, range(nbItems) ) )
print(message)
```

est bien plus rapide !

Affichage d'un grand nombre d'objets

Dans le cas où l'on désire afficher un tableau il y a une méthode plus simple. La fonction `print` admet la syntaxe étendue

```
print(item1,item2,...,sep=séparateur,end=affichage_en_fin)
```

et par défaut

```
print(item1,item2,...,sep=" ",end="\n")
```

le séparateur est un blanc et l'affichage se termine par un passage à la ligne.

Or, si `f` est une fonction les codes

```
f(1,2,3,4,5)
```

et

```
liste = [1,2,3,4,5]
f(*liste)
```

sont équivalents.

On pourra donc afficher une liste à raison d'un élément par ligne en écrivant

```
print(*liste,sep="\n")
```

et sur une seule ligne en séparant les éléments par un blanc par

```
print(*liste)
```

LIMITE DU NOMBRE DE RECURSIONS :

Lorsqu'une fonction appelle une autre fonction qui elle-même en appelle une autre et ainsi de suite, on s'intéresse au nombre maximum d'appels qui peuvent s'enchaîner ainsi les uns dans les autres. Cette question prend de l'importance lorsque nous faisons appel à la récursivité.

Par défaut, en Python, on ne peut faire que 1000 appels récursifs. Si vous avez besoin de faire plus d'appels, ajoutez ceci au début de votre programme :

```
import sys
sys.setrecursionlimit(1000)
```

en remplaçant 1000 par la bonne valeur. Il faudra peut-être la multiplier par 2, par 10 ou par 100, à vous de le déterminer.

Notez cependant qu'en pratique, il faut essayer d'éviter de fonctionner ainsi ; il est généralement préférable d'éviter les appels récursifs et d'utiliser des boucles à la place. N'utilisez cette technique que pour vous y exercer, ou alors quand vous n'avez pas d'autre choix.