II - Éléments de programmation (suite)

2) - Instructions conditionnelles

Une instruction conditionnelle s'exécute si et seulement si une condition est réalisée. Il est également possible d'exécuter des instructions différentes par disjonction avec un nombre quelconque de cas.

Syntaxe d'une instruction conditionnelle dans le cas général

```
if condition 1:
   Bloc d'instructions 1   # si la condition 1 est vraie alors le bloc d'instructions 1 est exécuté.
elif condition 2:
   Bloc d'instructions 2   # sinon, si condition 2 est vraie alors bloc d'instructions 2 est exécuté.
elif condition 3:
   Bloc d'instructions 3   # sinon, si condition 3 est vraie alors bloc d'instructions 3 est exécuté.
⋮
else:
   Bloc d'instructions n   # sinon le bloc d'instructions n est exécuté.
```

Les parties contenant `elif` et `else` sont facultatives.
Le symbole " : " signifie que l'on va commencer à définir le bloc d'instructions conditionnelles à partir de la ligne suivante par une suite d'instructions continument indentées par rapport au mot réservé *if* ou *elif* ou *else*.

Exemple 1 :

```
a = eval(input('Entrez la valeur de a : '))
b = eval(input('Entrez la valeur de b : '))
if a > b:
   mini = b
else:
   mini = a
print('Le minimum de {} et {} est : {}'.format(a,b,mini))
```

L'option elif permet de distinguer davantage de cas. Voyons cela sur un jeu amusant :

Exemple 2 :

```
a = int(input('Entrez un nombre entier entre 1 et 10 : '))
if a == 7:
   print('Vous avez gagné !')
elif a == 8 or a == 6:
   print("Vous avez presque gagné !")
else:
   print("Vous avez complètement perdu !")
```

3) - Fonctions

Python contient des fonctions prédéfinies. On peut compléter ces fonctions en important des modules ou en définissant ses propres fonctions.
Une fonction est une série d'instructions qui sont exécutées lorsqu'on appelle celle-ci. Une fois définie dans un fichier .py (appelé aussi module) et exécutée, une fonction peut être appelée (avec des paramètres effectifs) dans l'interpréteur de commandes ou dans un programme, comme n'importe quelle fonction prédéfinie ou importée.
Dans le bloc d'instructions d'une fonction, la commande facultative `return x` renvoie la valeur de `x` puis interrompt le programme.

Syntaxe de la définition d'une d'une fonction Python

```
def nom_de_la_fonction(liste des paramètres formels):
        bloc d'instructions de la fonction
```

Le symbole " : " signifie que l'on va commencer à définir la fonction à partir de la ligne suivante par une suite d'instructions continument indentées par rapport au mot réservé *def*.
L'exécution du script de la définition d'une fonction (sous réserve qu'aucune erreur n'ait été repérées) compile cette fonction. Le programme contenu dans une fonction compilée peut être exécuté par simple appel de la fonction accompagnée de paramètres numériques (appelés paramètres effectifs).
Une fonction peut n'avoir aucun paramètre.
Les paramètres formels se présentent sous la forme : `a,b,...,a1=u1,b1=v1,...`
les paramètres $a, b, \dots$ doivent être renseignés alors que les suivants $(a_1, b_1, \dots)$ sont définis par défaut et renseignés facultativement.
La commande *return* n'a de sens qu'à l'intérieur d'une fonction.

Exemple 3 :
Un professeur au système de notation très complexe dispose de notes s'écrivant sous forme décimale. Il souhaite arrondir ses notes au demi-point le plus proche, et il écrit pour cela une fonction Python qui automatise le procédé :

```
def arrondi(x):
    """renvoie l'arrondi de x au demi point le plus proche"""
    r=round(2*x)/2
    return r
```

On enregistre et exécute ce script dans un fichier .py et la fonction arrondi vient alors s'ajouter aux fonctions Python existantes.
Tester cette fonction pour différentes valeurs de `x` : 10, 7.2, 12.25, 5.5, 19.8, 14.75
La fonction help renvoie les spécifications ou docstrings de la fonction. Observez ce que donne la commande help(arrondi).

On dit que la fonction `arrondi` prend en paramètre ou argument la note `x` que l'on veut arrondir et retourne (ou renvoie) la note arrondie.
Essayez de comprendre ce que renvoie `round(x)`, au besoin en testant cette fonction sur les nombres `2, 3.4, 7.8, 6.5`.
Plus généralement, après avoir importé le module *math* (`from math import*`), exécuter les commandes `round(pi), round(pi,1), round(pi,2), round(pi,3)` puis décrire l'action de *round* lorsqu'on lui ajoute un deuxième paramètre.

EXERCICE 1 : Nombre de surjections

Soit $S_{p,n}$ le nombre de surjections de $[\![1,p]\!]$ dans $[\![1,n]\!]$ (inutile de connaitre la notion de surjection pour traiter cet exercice).

On pourrait démontrer par récurrence que $S_{p,n} = \begin{cases} \displaystyle\sum_{k=1}^{n}(-1)^{n-k}\binom{n}{k}k^p & \text{si } p \geqslant n \\ 0 & \text{si } p < n \end{cases}$

Écrire un script (ou une fonction) qui affiche (ou retourne) le nombre de surjections de $[\![1,p]\!]$ dans $[\![1,n]\!]$.

Pour le calcul de $\binom{n}{k}$ on écrira une fonction *binome* de paramètres *n, k* qui renvoie le nombre entier $\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$.
Pour minimiser les calculs on pourra commencer par remplacer k par $n-k$ si $k > \frac{n}{2}$.
On codera ensuite une boucle qui calcule concomitamment le numérateur et le dénominateur. On rappelle que `n//p` renvoie le quotient de la division euclidienne de n par p.

Vérification : si $p = 3$, on doit trouver 6 à la fois pour le cas $n = 2$ et pour le cas $n = 3$, tester aussi le cas $n = 4$ pour lequel on doit trouver 0. Pour $p = 4$ et $n = 2$ on doit trouver 14.

EXERCICE 2 : Factorielle, le retour
Écrire une fonction factorielle qui renvoie la factorielle d'un entier, et qui renvoie un message d'erreur si le nombre fourni en argument n'est pas un entier naturel (on pourra réutiliser les scripts des TD 1 et 2).

III - Listes

Une liste est un objet ordonné contenant d'autres objets de type quelconque délimités par des crochets et séparés par des virgules. Les objets contenus dans une liste n'ont pas forcément tous le même type.
L'accès au terme d'indice i d'une liste `lst` s'obtient par la commande `lst[i]` qui se comporte comme une variable : on peut afficher sa valeur, l'intégrer dans un programme, lui réaffecter une valeur.
Attention, le premier terme d'une liste est indexé par 0 (ou $-n$) et le dernier par $n-1$ (ou -1) où n est le nombre de termes de la liste.

Exemple 1 : Taper dans l'interpréteur de commandes chacune des six instructions suivie de la touche *Entrée* :

```
a = []  #a est la liste vide. La liste vide s'écrit aussi list()
b=[1, 3, 5, 7]  #b est une liste contenant des entiers
c=['assez',1,3.14,[1,8]] #c est une liste contenant une chaine, un entier, un flottant, une liste
a
b
c
```

Valider chacune des instructions suivantes par la touche *Entrée* et interpréter les résultats obtenus (y compris les erreurs).
`a[0], b[0], b[3], b[4], c[0], c[3], c[3][0], c[3][1], c[3][2]`

Fonctions à connaitre qui prennent en argument une liste `lst` sans la modifier (import random as rd)

Commandes	len(lst)	sum(lst)	max(lst)	min(lst)	rd.choice(lst)
Renvoie	Nombre de termes	Somme des termes	Maximum	Minimum	Terme choisi au hasard

Remarque. la fonction *len* s'applique également aux chaînes de caractères et aux tuples.

Méthodes qui modifient une liste *lst* (retenir les trois premières)

Commandes	Actions
`lst.append(a)`	Ajoute a à la fin de *lst* et ne renvoie rien du tout.
`lst.pop(i)`	Élimine le terme d'indice i de *lst* et le renvoie. Si i n'est pas renseigné, les opérations précédentes s'appliquent au dernier terme de *lst*.
`lst.remove(a)`	Élimine le premier a rencontré dans *lst* en la parcourant dans le sens normal et ne renvoie rien.
`lst.insert(i,a)`	Insère a dans *lst* au niveau du $i^{\text{ème}}$ terme et ne renvoie rien du tout.
`lst.sort()`	Classe `lst` dans l'ordre croissant et ne renvoie rien.
`lst.reverse()`	Inverse l'ordre des termes de `lst` et ne renvoie rien.
`lst.extend(lst1)`	Ajoute les éléments de *lst*1 à la fin de *lst*. Équivalent en place de `lst = lst + lst1`.

Concaténation de listes

Comme pour les chaines de caractères, on peut effectuer la concaténation (ou la mise bout à bout) de listes. Cette opération se code avec le symbole +.

La fonction *choice* du module *random* prend en paramètre une liste *lst* et renvoie un terme de cette liste au hasard.
La fonction *randint* du module *random* prend en arguments 2 entiers a et b et renvoie un entier au hasard entre a et b.

EXERCICE 3 : manipulation d'une liste d'entiers

Écrire une fonction *manip(lst,n)* qui ne renvoie rien et qui effectue les taches suivantes dans l'ordre prescrit :

1. enlève les termes de la liste *lst* égaux à n (*range(a,b,-1)* sont les entiers $a, a-1, \ldots, b+1$),
2. inverse l'ordre des termes de *lst*,
3. insert au hasard dans *lst* un entier au hasard entre 1 et 10,
4. retire au hasard un terme de *lst*,
5. insert à la fin de *lst* la somme des termes de *lst*.

Tester cette fonction avec les paramètres : `lst` contenant `[1,3,7,5,3,4]` et l'entier `3`.

Rappel concernant *break* : dès que le programme rencontre la commande ***break*** dans une boucle, celle-ci s'interrompt et le programme continue après la boucle. La commande équivalente pour les instructions conditionnelles est ***continue***.

EXERCICE 4 : Saisie de notes

Écrire un script Python qui demande à un candidat de saisir le nombre de notes ainsi que ces notes et qui affiche le texte 'Vous êtes éliminé' si au moins l'une des notes est éliminatoire (inférieure ou égale à la variable `note_elim`) ou bien la moyenne des notes si aucune note n'est éliminatoire. On pourra compléter le code suivant :

```
n = _____________________________
lst = []              # lst est initialisée à la liste vide
note_elim = eval(input("Entrer la valeur de la note éliminatoire (entrer -1 s'il n'y en a pas) : "))
elimine = False
for k in range(n):
    a = _____________________________________
    if ___________________:
        elimine = True        # une variable contenant un booléen est appelée drapeau
        break        # on sort de la boucle s'il y a une note éliminatoire même s'il restait beaucoup
                     # de notes à rentrer
    lst.append(_)
if elimine:
    print(____________________)
else:
    print(_____________)
```

Ce programme peut être sensiblement simplifié en l'écrivant sous la forme d'une fonction, en remplaçant l'instruction `break` par `return 'Ce candidat est éliminé'` et en supprimant ou modifiant certaines instructions. Effectuer cette modification.

EXERCICE 5 : Concaténation des listes des termes d'indices pairs et impairs

Écrire deux fonctions *scission_parite1* et *scission_parite2* d'argument *lst* qui renvoie *lst2, max(lst0) - min(lst0), max(lst1) - min(lst1)* où *lst0* est la liste des termes de *lst* d'indices pairs, *lst1* la liste des termes d'indices impairs et *lst2* la concaténation de *lst0* et *lst1*. Rappel : la concaténation des listes $[1,2,3]$ et $[4,5]$ est la liste $[1,2,3,4,5]$.

On procèdera de deux façons (n désigne le nombre de termes de lst) :

- Avec une seule boucle *for*. On fera varier k de 0 (inclus) à n (exclu) puis on discutera suivant la parité de k. On rappelle que `k%2` renvoie le reste de la division euclidienne de k par 2, par conséquent `k%2` renvoie 0 si `k` est pair et 1 si `k` est impair .
- Avec deux boucles *for*. Pour la première on fera varier k de 0 (inclus) à n (exclu) en parcourant les nombres pairs et pour la deuxième on fera varier k de 1 (inclus) à n (exclu) en parcourant les nombres impairs. On rappelle que $range(p,q,r)$ est la séquence des entiers de p (inclus) à q (exclu) avec un pas de défilement égal à r.

Vérifier qu'avec l'argument $[3,5,2,1,4]$ les deux fonctions renvoient $[3,2,4,5,1]$, 2, 4.

EXERCICE 6 : Jeu de devinette

Programmez un jeu tout simple consistant à faire deviner un nombre entier compris entre 1 et 50 au joueur. À chaque proposition faite par le joueur, le programme doit préciser si le nombre fourni est trop grand ou trop petit. Le jeu s'arrête lorsque le joueur a trouvé le bon nombre. Le nombre à trouver doit être généré aléatoirement par l'ordinateur (il varie donc à chaque jeu et le joueur ne le connait pas tant que le jeu n'est pas fini), utilisez pour cela la fonction `randint` avec les arguments $(1,50)$ de la bibliothèque importée `random`.

Petit raffinement possible : indiquez le nombre de tentatives qui ont été nécessaires pour que le joueur trouve la solution.

EXERCICE 7 : L'ordinateur joue contre lui-même (ce qui permet de simuler un grand nombre de parties)

La bibliothèque *random* contient la fonction *randint* de paramètres (n,p) qui renvoie un nombre entier au hasard entre l'entier n et l'entier p, elle contient également la fonction *choice* qui renvoie un terme d'une liste au hasard.

Programmer le jeu précédent en remplaçant le joueur humain par l'ordinateur.

On écrira quatre fonctions correspondant aux stratégies suivantes en indiquant le nombre de tentatives :

1. À chaque étape, l'ordinateur choisit aléatoirement un nombre entre 1 et 50 (stratégie sans mémoire).
2. À chaque étape, l'ordinateur choisit aléatoirement un nombre parmi ceux qui n'ont pas été choisis aux étapes précédentes. Une liste contenant les nombres non essayés évoluera grâce à la méthode *remove*. Un tirage au hasard dans cette liste se fera grâce à la fonction *choice* du module *random*.
3. À chaque étape, l'ordinateur choisit aléatoirement un nombre entre deux nombres a et b qui évoluent en prenant en compte la situation de chaque tentative par rapport au nombre à deviner (plus grand ou plus petit).
4. Même stratégie que la précédente en remplaçant le choix aléatoire dans $[\![a,b]\!]$ par le nombre entier le plus proche du milieu de l'intervalle $[a,b]$ (dichotomie).
5. Écrire une fonction *moy* qui renvoie les 4 moyennes du nombre de tentatives sur 10000 parties pour les 4 stratégies.
6. Vérification : appeler la fonction *moy*, en déduire un classement de ces quatre stratégies en fonction de leur efficacité. Donner approximativement et en moyenne le nombre de tentatives nécessaires pour trouver le nombre secret dans les stratégies 1 (sans mémoire) et 4 (dichotomique).

EXERCICE 8 : Algorithme et suite de Kaprekar

L'algorithme de Kaprekar consiste à associer à un nombre quelconque n un autre nombre $K(n)$ généré de la façon suivante :

- on forme le nombre n_1 en arrangeant les chiffres du nombre n dans l'ordre croissant et le nombre n_2 en les arrangeant dans l'ordre décroissant ;
- on définit $K(n) = n_2 - n_1$;

1. Écrire une fonction `digits(n)` qui renvoie la liste des chiffres de n rangés dans l'ordre croissant.
2. Écrire une fonction `conversion(lst)` qui renvoie l'entier correspondant à la liste de ses chiffres stockés dans `lst`.
3. Écrire une fonction `kaprekar(n)` qui renvoie $K(n)$.
4. Écrire une fonction `suiteK(n)` qui renvoie les termes itérés de l'algorithme de Kaprekar sous la forme d'une liste en commençant par n jusqu'à obtenir 0 ou un nombre égal à l'un des termes précédents (cycle).
5. Vérifier que `suiteK(24)` renvoie `[24,18,63,27,45,9,0]` et `suiteK(53955)` renvoie `[53955,59994,53955]`.

On pourra utiliser les commandes suivantes :

- `a in lst` indique par un booléen si l'objet `a` est dans la liste `lst`. Exemple : `1 in [2,3,8]` renvoie `False`.
- `str(n)` renvoie la chaîne de caractères des chiffres d'un entier n. Exemple : `str(292)` renvoie `"292"`.
- `list(ch)` renvoie la liste des caractères de la chaîne de caractères `ch`. Exemple : `list("292")` renvoie `["2","9","2"]`.
- `eval(ch)` renvoie l'évaluation numérique de la chaîne de caractères `ch`. Exemple : `eval("2")` renvoie `2`.
- `lst.sort()` range dans l'ordre croissant les termes de la liste `lst` qui sont tous des nombres ou tous des caractères.
  Exemple. Si `lst` contient `[2,1]` alors après la commande `lst.sort()` la liste `lst` contient `[1,2]`.
  Exemple. Si `lst` contient `["2","1"]` alors après la commande `lst.sort()` la liste `lst` contient `["1","2"]`.

On rappelle que `lst[-1]` est le dernier terme de `lst`, `lst[-2]` l'avant-dernier, `lst[-3]` l'antépénultième, etc.