INF101 et INF104 : examen final

10 Janvier 2022

- **Durée 2 heures.** Le sujet fait 3 pages. Aucun document, ni calculette/téléphone/...
- **Lisez bien ces consignes avant de commencer !**
- Respectez **strictement** les consignes de l'énoncé (noms de fonctions/variables, format d'affichage...)
- On interdit d'utiliser `break` et `continue`, ainsi que toutes fonctions non vues en cours.
- Un programme qui fonctionne mais ne respecte pas toutes les consignes ne rapporte **pas** de points. Un programme qui 'fonctionne' mais de manière manifestement sous-optimale ne rapporte pas tous les points.
- Vos programmes doivent être **LISIBLES**: indentation correcte, commentaires, pas de raccourcis...
- Les exercices sont indépendants, les questions dans chaque exercice aussi. Vous pouvez toujours utiliser une fonction d'une question précédente même sans l'avoir écrite.
- Le barème est **indicatif**. La qualité de la rédaction et de la présentation sera prise en compte. Les questions étoilées sont plus difficiles et valent plus de points.
- Répondez aux exercices **DANS l'ORDRE**. Laissez de la place si vous voulez revenir à un exercice plus tard. Évitez absolument les réponses éparpillées et les exercices mélangés entre eux.

1 EXERCICE 1 : BOOLEENS (2 points)

On suppose que toutes les variables sont initialisées avec les types suivants : x est une chaîne de caractères; a, b, c sont des entiers; d est un dictionnaire; li est une liste. Écrire les expressions booléennes correspondant aux affirmations suivantes:

1. a est un multiple de b et est inférieur ou égal à c
2. x est un caractère non alphabétique
3. x est une clé du dictionnaire d
4. li est une liste non vide et son dernier élément est soit a soit b soit c

2 EXERCICE 2 : MOTS ET TRI, LE RETOUR (6 points)

On veut écrire un programme qui lit un par un des mots au clavier et les enregistre dans une liste triée selon l'algorithme du tri par insertion. L'écriture du programme suivra les questions ci-dessous. L'exécution du programme doit produire le résultat suivant :

```
Exemple d'exécution :
>
Tape des mots (stop pour arrêter)
? bateau
? velo
? voiture
? avion
? chat
? stop
Tes mots dans l'ordre : chat; velo; avion; bateau; voiture.
Rejouer? : yep
réponse incorrecte, rejouer? : ouiii
réponse incorrecte, rejouer? : oui
Tape des mots (stop pour arrêter)
? bonjour
? stop
Tes mots dans l'ordre : bonjour.
Rejouer? : non
Au-revoir
```

1. Écrire une fonction `verifierMot(mot)` qui reçoit en argument une chaîne de caractères *mot*, et qui **renvoie un booléen** indiquant si *mot* contient uniquement des caractères alphabétiques minuscules. La fonction doit arrêter la vérification dès que possible.

Tournez la page

2. Écrire une fonction `lireFiltrer()` qui ne reçoit aucun argument, qui lit une chaîne de caractères au clavier, la **filtre** (grâce à la fonction précédente) jusqu'à ce qu'elle soit bien constituée uniquement de lettres minuscules, et la **renvoie** alors.

3. Écrire une fonction `insererMot(mot,liste)` qui reçoit en argument un mot et une liste de mots déjà triée par ordre croissant de taille (nombre de caractères), et par ordre alphabétique pour les mots de même taille. Cette fonction **modifie** la liste en y insérant le mot au bon endroit pour que la liste reste triée. La fonction ne **renvoie rien**.

4. Écrire une fonction `triLecture()` qui ne reçoit aucun argument, et qui lit et filtre des mots au clavier (grâce à la fonction `lireFiltrer`) jusqu'à lire le mot "stop" (qui arrête la lecture et n'est pas compté dans la suite). Cette fonction doit créer au fur et à mesure une liste triée de tous les mots lus, triés dans l'**ordre croissant de taille** (nombre de caractères; les mots de même taille en ordre alphabétique). On ne doit créer qu'**une seule** liste. Cette liste **doit être maintenue triée à tout moment**, en insérant chaque nouveau mot lu directement au bon endroit, grâce à la fonction précédente. A la fin de la lecture, la fonction renvoie la liste (triée) de mots lus. *On **interdit** bien sûr de trier la liste entière a posteriori, avec la fonction `sort` ou tout autre moyen.*

5. Écrire une fonction `afficheMots(li)` qui reçoit une liste de mots et les affiche sur une ligne, séparés par un point virgule, terminée par un point et un retour à la ligne *(cf exemple).*

6. Écrire un programme principal qui demande à l'utilisateur de taper des mots puis qui affiche la liste triée, en appelant les fonctions ci-dessus (aucune réécriture inutile de code). Le programme doit ensuite proposer de rejouer (réponse "oui" ou "non"), et recommencer tant que l'utilisateur accepte de rejouer (réponse "oui" exactement). Respecter **exactement** le format de l'exemple.

3 EXERCICE 3 : NOMBRES BRESILIENS (12 points)

Un entier est premier s'il n'admet pas d'autre diviseur que 1 et lui-même. Un entier peut se décomposer de manière unique comme produit de facteurs premiers.

1. Écrire une fonction `estPremier(n)` qui vérifie si un entier n est premier, et renvoie un booléen (vrai si premier, faux sinon).

2. [**] Écrire une fonction `facteursPpremiers(n)` qui décompose un entier n en la liste de ses facteurs premiers (chacun apparaît autant de fois dans la liste que sa puissance). *Par exemple, 8 ($= 2^3$) se décompose en [2,2,2]; 18 ($= 2 * 3^2$) se décompose en [2,3,3], 17 se décompose en [17].*

Un entier n est dit brésilien s'il existe une base $1 < b < n - 1$ telle que n s'écrit avec des chiffres tous identiques dans la base b. Par exemple 8 est brésilien car il s'écrit 22 en base 3 (2 * 3 + 2). **Remarque :** la condition $b < n$–1 est indispensable, car tout entier n s'écrit 11 en base $n - 1$ et donc tout nombre serait brésilien.

3. Écrire une fonction `chiffresEgaux(ns)` qui vérifie si la **chaîne de caractères** *ns* reçue en argument (qui représente un entier n dans une certaine base $b < n - 1$) a bien tous ses caractères égaux. La fonction renvoie un booléen (vrai si tous les caractères égaux, faux sinon). *Par exemple vrai pour "111" ou "AA", faux pour "1B".*

4. Écrire une fonction `bresilien(n)` qui teste si un entier n est brésilien. Si le nombre est brésilien, la fonction doit **s'arrêter dès que possible** (première base trouvée avec une écriture aux chiffres égaux). La fonction doit **afficher** cette base, et **afficher** l'écriture de n dans cette base. Si n n'est pas brésilien, la fonction n'affiche rien. Dans tous les cas, la fonction **renvoie** un booléen indiquant si le nombre est brésilien (vrai) ou non (faux). Pour le changement de base, on utilisera la fonction `numpy.base_repr(n,b)` qui renvoie une chaîne de caractères représentant n dans la base b. **Attention**: jusqu'à quelle base faut-il tester pour conclure que n n'est pas brésilien?

On s'intéresse maintenant à la suite des nombres brésiliens, et la suite des nombres **premiers** brésiliens.

5. Écrire une fonction `listeBresiliens(sup)` qui renvoie la liste des entiers brésiliens inférieurs ou égaux à une borne supérieure *sup* reçue en paramètre. On appellera pour cela la fonction `bresilien`.

6. Écrire une fonction `listePremiersBresiliens(x)` qui renvoie la liste des x premiers nombres **premiers** brésiliens (à la fois premiers et brésiliens). La liste renvoyée doit donc contenir exactement x entiers. On appellera pour cela les fonctions précédentes `bresilien` et `estPremier`. Par exemple pour $x = 5$ cette liste contient 7, 13, 31, 43, 73.

Un nombre peut être plusieurs fois brésilien s'il existe plusieurs bases différentes (toutes strictement inférieures à $n - 1$) dans lesquelles son écriture comporte des chiffres tous égaux. Si le nombre admet des écritures à chiffres égaux dans k bases différentes, on dit qu'il est k-brésilien. Par exemple 40 est 4-brésilien car il s'écrit '1111' en base 3, '55' en base 7, '44' en base 9, et '22' en base 19, et qu'il n'a pas d'autre écriture avec les chiffres tous égaux dans les autres bases.

7. Écrire une fonction `dicoBresilien(n)` qui **renvoie** un dictionnaire associant chacune des bases dans lesquelles l'entier n s'écrit avec des chiffres tous égaux (et **uniquement ces** bases), à l'écriture de n dans cette base. Les clés sont les bases, les valeurs sont l'écriture de n dans ces bases. Par exemple, 31 est 2-brésilien, car il s'écrit "11111" en base 2 et "111" en base 5, donc le dictionnaire renvoyé sera {`2:"11111",5:"111"`}. On remarque que ce dictionnaire ne contient **que** ces 2 écritures aux chiffres égaux.

Il existe pour chaque k une séquence de nombres k-brésiliens. La suite des **plus petits nombres k-brésiliens** (pour chaque valeur successive de $k >= 1$) débute par 7 (plus petit entier 1-brésilien), 15 (plus petit entier 2-brésilien), 24 (plus petit entier 3-brésilien), 40 (plus petit entier 4-brésilien), 60 (plus petit 5-brésilien), 144 (plus petit 6-brésilien), 120 (plus petit 7-brésilien), etc. Remarquez que cette séquence n'est **pas** croissante.

8. [**] Écrire une fonction `miniKBresiliens(maxk)` qui renvoie la liste des plus petits nombres k-brésiliens pour toutes les valeurs de k entre 1 et $maxk$ inclus.

Un entier n est **hautement brésilien** s'il possède davantage d'écritures le définissant comme nombre brésilien que n'importe quel entier inférieur à lui. Par exemple 120 est hautement brésilien car il est 7-brésilien et qu'aucun nombre inférieur à 120 n'est plus que 6-brésilien. Par contre 144 n'est pas hautement brésilien car il est seulement 6-brésilien, alors que 120 est plus petit que 144 mais est 7-brésilien.

9. [**] Écrire une fonction `hautementBresiliens(n)` qui renvoie la liste des n premiers nombres hautement brésiliens (plus brésiliens que les nombres avant eux). Par exemple pour $n = 6$ la liste renvoyée est constituée des 6 premiers nombres hautement brésiliens : 7, 15, 24, 40, 60, 120.

FIN

Mémo Python - UE INF101 / INF104 / INF131 - version 2021

Opérations sur les types

`type()` : pour connaître le type d'une variable
`int()` : transformation en entier
`float()` : transformation en flottant
`str()` : transformation en chaîne de caractères

Infini

`float('inf')` : valeur infinie positive ($+\infty$)
`float('-inf')` : valeur infinie négative ($-\infty$)

Écriture dans la console

```
print(a1,a2,...,an, sep=xx, end=yy)
```

- Pour imprimer une suite d'arguments de a1 à an
- sep : permet de définir le séparateur affiché entre chaque argument (optionnel, par défaut " ")
- end : permet de définir ce qui sera affiché à la fin (optionnel, par défaut : saut de ligne)

Lecture dans la console

```
res = input(message)
```

- Pour lire une suite de caractères au clavier terminée par $< Enter >$
- Ne pas oublier de transformer la chaîne en entier (int) ou réel (float) si nécessaire.
- La chaîne de caractères résultante doit être affectée (ici à la variable `res`).
- L'argument est optionnel : c'est un message explicatif destiné à l'utilisateur

Opérateurs booléens

```
and: et logique     or: ou logique
not: négation
```

Opérateurs de comparaison

```
== égalité              != différence
<  inférieur,           <= inférieur ou égal
>  supérieur,           >= supérieur ou égal
```

Opérateurs arithmétiques

```
+ : addition,          - : soustraction
* : multiplication,    ** : puissance,
/ : division,          // : quotient div entière,
% : reste de la division entière (modulo)
```

Fonctions arithmétiques

```
abs(x): valeur absolue
math.sqrt(x): racine carrée
```

Aléatoire: module random

`random.randint(inf,sup)`: entier aléatoire entre bornes inf et sup incluses
`random.shuffle(maListe)`: mélange la liste (effet de bord), ne renvoie rien
`random.choice(maListe)`: renvoie un élément au hasard de la liste

Instructions conditionnelles

```
if condition :
    instructions

if condition :
    instructions
else :
    instructions
```

```
if condition1 :
    instructions
elif condition2 :
    instructions
else :
    instructions
```

Caractères

`ord(c)` : renvoie le code ASCII du caractère c
`chr(a)` : renvoie le caractère de code ASCII a

Chaînes de caractères

`len(s)` : renvoie la longueur de la chaîne s
`s1+s2` : concatène les chaînes s1 et s2
`s*n` : construit la répétition de n fois la chaîne s
exemple : "ta"* 3 donne "tatata"
`list(chaine)` : renvoie la liste des caractères de la chaîne
`ch.split(arg)` : retourne la liste des sous-chaînes de *ch*, en coupant à chaque occurrence de `arg` (par défaut `arg=" "`)
`ch.join(liste)` : concatène les chaînes de `liste`, en utilisant `ch` comme séparateur, et renvoie la chaîne résultante
`ch.upper()`: passe `ch` en majuscules
`ch.lower()`: passe `ch` en minuscules

Itération tant que

```
while condition :
    instructions
```

Itération for, et range

```
for e in conteneur :
     instructions

for var in range (deb, fin, pas) :
     instructions
```

Itère les instructions avec e prenant chaque valeur dans le conteneur (liste, chaîne ou dictionnaire) ; ou avec var prenant les valeurs entre `deb` et `fin` avec un `pas` donné.

`range(a)` : séquence des valeurs [0, a[
`range (b,c)` : séquence des valeurs [b, c[(pas=1, $c > b$)
`range (b, c, g)` : idem avec un pas = g
`range(b,c,-1)`: valeurs décroissantes de b (incl.) à c (excl.), pas=-1 ($c < b$)

Listes

`maListe = []`: création d'une liste vide
`maListe = [e1,e2,e3]` : création d'une liste, ici à 3 éléments e1, e2, et e3

`maListe[i]`: obtenir l'élément à l'index i ($i >= 0$).
Les éléments sont indexés à partir de 0. Si $i < 0$, les éléments sont accédés à partir de la fin de la liste. Ex : `maListe[-1]` permet d'accéder au dernier élément de la liste

`maListe.append(elem)`: ajoute un élément à la fin
`maListe.extend(liste2)`: ajout de tous les élements de la liste `liste2` à la fin de la liste `maListe`
`maListe.insert(i,elem)`: ajout d'un élément à l'index i

`res = maListe.pop(index)`: retire l'élément présent à la position `index` et le renvoie, ici dans la variable `res`
`maListe.remove(element)`: retire l'élément donné (le premier trouvé)

`len(maListe)`: nombre d'éléments d'une liste
`elem in maListe`: teste si un élément est dans une liste (renvoie True ou False)
`maListe.index(elem)`: renvoie l'index (la position) d'un élément dans une liste (ValueError si absent)

`l2 = maListe`: crée un synonyme (2ème nom pour la liste)
`l3 = list(maListe)`: crée une copie de surface (un clone)
`l4 = copy.deepcopy(maListe)`: crée une copie profonde (récursive)

Définition d'une fonction

```
def nomFonction(arg) :
   instructions
   return v
```

Fonction qui renvoie la valeur ou variable v.

Dictionnaires

`monDico = {}` : création d'un dictionnaire vide
`monDico = { c1:v1, c2:v2, c3:v3 }` : création d'un dictionnaire, ici à 3 entrées (clé c1 avec valeur v1, etc)

`e = monDico[c1]` : les valeurs du dictionnaire sont accessibles par leurs clés. Ici, e prendra la valeur v1. Provoque une erreur si la clé n'existe pas.

`monDico[c3] = v3`: ajoute une nouvelle valeur au dictionnaire (ici v3) avec une clé (ici c3). Si la clé existe déjà, la valeur associée est modifiée.

`del monDico[C3]`: supprime une association dans le dictionnaire. La clé doit exister.

`c in monDico`: vérifie l'existence d'une clé dans le dictionnaire, renvoie True ou False.

`len(monDico)`: longueur d'un dictionnaire.

`dic2 = monDico`: crée un synonyme (2ème nom au dico)
`dic3 = dict(monDico)`: crée une copie de surface (clone)
`dic4 = copy.deepcopy(monDico)`: crée une copie profonde (récursive)