

INF101/INF131 : devoir surveillé

Vendredi 26 Octobre 2018

- **Durée 2 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait 4 pages.
- Respectez **strictement** les consignes de l'énoncé (noms de variables, format d'affichage...)
- Pensez à commenter vos programmes pour les rendre plus lisibles.
- Les exercices sont indépendants, et en ordre croissant de difficulté.
- Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc est interdite.
- Le barème est indicatif. La qualité de la rédaction et de la présentation sera prise en compte.

1 EXERCICE 1 : BOOLÉENS (3 points)

On suppose que toutes les variables nécessaires sont bien initialisées.

1. Écrire les expressions booléennes vraies si:
 - a est strictement positif et multiple soit de b soit de c.
 - La variable x contient une lettre de l'alphabet en minuscule, et la variable y contient la même lettre en majuscule. Indice : utiliser la fonction `ord()`. Remarque : attention, x et y sont des chaînes de caractères, et peuvent être de longueur quelconque.
 - La variable `rep` contient la réponse correcte à la question "a,b,c sont-ils dans l'ordre croissant ? répondez par o ou n". *Indice : rep contient donc soit "o", soit "n", et il faut vérifier que cette réponse donnée par l'utilisateur correspond bien à ce qu'il aurait dû répondre étant donné l'ordre de a,b,c.*
2. Écrire la négation des 3 expressions booléennes ci-dessus, en simplifiant au maximum (pas d'opérateur `not`).

2 EXERCICE 2 : FONCTIONS (2 points)

On donne le code suivant :

```
def ma_fonction(a,b) :
    somme = a+b
    produit = a*b
    a = a+1
    print("somme =",somme,"et produit =",produit)

# programme principal
a = ma_fonction(3,7)
print(a)
```

Répondez aux questions suivantes en expliquant vos réponses.

1. Quelles sont les variables locales de la fonction `ma_fonction` ?
2. Quelle est la valeur de retour de cette fonction ?
3. Quels sont ses effets de bord ?
4. Qu'affiche l'instruction `print(a)` du programme principal ?

3 EXERCICE 3 - ALPHABET (4 points)

Lisez bien l'exercice en entier avant de répondre ! Écrire un programme qui :

- Demande à l'utilisateur une lettre de l'alphabet minuscule.
- La filtre jusqu'à ce qu'il s'agisse bien d'une lettre minuscule.
- Calcule et affiche sa position dans l'alphabet.
- Calcule et affiche la majuscule de même position.
- Propose de rejouer ; tant que l'utilisateur accepte le programme recommence au début (avec une nouvelle lettre)
- Se termine quand l'utilisateur refuse de rejouer, en affichant "fin du programme"

Remarque importante : dans cet exercice on interdit d'utiliser les fonctions prédéfinies Python : `lower`, `upper`, `isalpha`. Vous pouvez utiliser les fonctions `chr` et `ord`.

Exemple d'exécution :

```
Entre une lettre minuscule : @
Erreur, entre une lettre minuscule : T
Erreur, entre une lettre minuscule : a
Position de a : 1
Majuscule correspondante : A
Rejouer ? oui
Entre une lettre minuscule : e
Position de e : 5
Majuscule correspondante : E
Rejouer ? non
Fin du programme
```

4 EXERCICE 4 - BOUCLES (3 points)

On donne le code à trous suivant :

```
i = <init>
while <condition> :
    print(<valeur>)
    i = <màj>
```

Donner les valeurs des 4 éléments manquants (init, condition, valeur, mise à jour), pour produire les affichages suivants :

1. Les entiers impairs entre 1 (inclus) et 1000, sur une seule ligne, séparés par un espace :
1 3 5 7 ... 997 999
2. Les entiers strictement positifs strictement inférieurs à 100, en ordre décroissant, séparés par un espace : 99 98 97 ... 2 1
3. Les puissances de 3, pour les exposants entre 0 et 9 (inclus), un résultat par ligne :
1
3
9
27
81
...

5 EXERCICE 5 - JEU DU PENDU (8 points)

On veut coder un jeu du pendu, qui consiste à faire deviner un mot secret lettre par lettre. Le joueur a droit à un nombre limité d'erreurs avant de perdre la partie. A chaque tour les lettres déjà trouvées du mot sont affichées à leur place dans le mot, les autres sont masquées et remplacées par un tiret ; le programme affiche aussi la liste des lettres déjà essayées, pour éviter au joueur de faire plusieurs fois la même proposition erronée (on supposera dans la suite que le joueur ne propose pas plusieurs fois la même lettre). Le joueur gagne quand il a trouvé toutes les lettres (avant d'avoir fait trop d'erreurs) et donc le mot. Il perd s'il atteint le nombre maximum d'erreurs autorisées avant d'avoir trouvé le mot complet. On suppose que le mot secret est entièrement en minuscules, et que le joueur ne saisit que des lettres minuscules. On ne demande donc pas de le vérifier.

Voir les exemples d'exécution après les questions. On vous demande de respecter exactement les affichages de ces exemples.

1. Écrire une fonction `affiche` qui reçoit en arguments `mot` (le mot secret à deviner) et `lettres` (la liste des lettres déjà essayées par l'utilisateur) ; cette fonction affiche le mot tel que doit le voir le joueur, les lettres déjà essayées sont affichées à leur place, les autres sont remplacées par un tiret, les éléments sont séparés par des points, et on affiche un retour à la ligne final après le mot. *Par exemple pour le mot "bonjour", si la liste des essais contient les lettres "o" et "p", cette fonction doit afficher "-.o.-.-.o.-.-.", comme dans l'exemple d'exécution ci-dessous.*
2. Écrire une fonction `liste_erreurs` qui reçoit en arguments le mot secret et la liste des lettres essayées, et calcule et renvoie la liste des erreurs, c'est-à-dire des lettres essayées par le joueur mais qui ne sont pas dans le mot secret. *Par exemple pour le mot "bonjour", si la liste des essais contient les lettres "o" et "z", alors la liste des erreurs ne contient que la lettre "z".*
3. Écrire une fonction `nb_erreurs` qui reçoit en arguments le mot secret et la liste des lettres essayées, et renvoie le nombre d'erreurs déjà faites par le joueur. *Indice : utiliser la fonction précédente.*
4. Écrire une fonction booléenne `gagne` qui reçoit en arguments le mot secret et la liste des lettres essayées, et renvoie un booléen indiquant si le joueur a trouvé toutes les lettres du mot secret (s'il a donc gagné).
5. Écrire une fonction `tour_de_jeu` qui reçoit le mot secret et la liste des essais, et gère un tour de jeu en utilisant les fonctions ci-dessus. Cette fonction doit :
 - Afficher le numéro du tour (à partir de 1) ;
 - Afficher le mot tel que doit le voir le joueur (uniquement les lettres déjà trouvées, les autres sont masquées) ;
 - Afficher les lettres déjà essayées (les erreurs) ;
 - Demander à l'utilisateur de proposer une lettre (bonus : la filtrer pour qu'il s'agisse bien d'une lettre, voire pour qu'elle n'ait pas été déjà proposée avant) ;
 - Ré-afficher le mot, avec éventuellement une lettre en plus ;
 - Bonus : si la lettre n'est pas dans le mot, afficher un message d'erreur (indiquer combien d'erreurs ont déjà été commises) au lieu de ré-afficher le mot (puisque'il n'a pas changé) ;
 - Renvoyer la liste d'essais mise à jour.
6. Écrire le programme principal qui boucle tant que l'utilisateur n'a ni gagné (trouvé le mot) ni perdu (épuisé le nombre d'erreurs autorisées). Ce programme utilise les fonctions définies ci-dessus. On fixera le nombre d'erreurs autorisées à 7, et le mot secret devra être entré par l'utilisateur (on suppose qu'il s'agit d'un 2e joueur). Ce programme se termine en affichant un message de victoire (avec le nombre d'erreurs) ou de défaite (avec la solution), comme dans les exemples ci-dessous.

Exemple d'exécution souhaitée (victoire) : *Exemple d'exécution souhaitée (défaite)*

Tour 1

-.-.-.-.-

Essais faux : []

Lettre ? : a

Erreur 1 , tu as encore droit a 6 erreurs

Tour 2

-.-.-.-.-

Essais faux : ['a']

Lettre ? : b

b.-.-.-.-

Tour 3

b.-.-.-.-

Essais faux : ['a']

Lettre ? : o

b.o.-.-.-

...

Tour 7

b.o.n.j.o.u._.

Essais faux : ['a']

Lettre ? : r

b.o.n.j.o.u.r.

Bravo, gagne avec seulement 1 erreurs

...

Tour 6

.o..o._.r.

Essais faux : ['e', 't', 'z']

Lettre ? : b

b.o._.o._.r.

Tour 7

b.o._.o._.r.

Essais faux : ['e', 't', 'z']

Lettre ? : p

Erreur 4 , tu as encore droit a 3 erreurs

Tour 8

b.o._.o._.r.

Essais faux : ['e', 'p', 't', 'z']

Lettre ? : w

Erreur 5 , tu as encore droit a 2 erreurs

Tour 9

b.o._.o._.r.

Essais faux : ['e', 'p', 't', 'w', 'z']

Lettre ? : q

Erreur 6 , tu as encore droit a 1 erreurs

Tour 10

b.o._.o._.r.

Essais faux : ['e', 'p', 'q', 't', 'w', 'z']

Lettre ? : i

Erreur 7 , tu as encore droit a 0 erreurs

Perdu, le mot etait : bonjour

Mémo Python - UE INF101 / INF131 / INF204

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

Définition d'une fonction

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

Fonction qui renvoie la valeur ou variable `v`.

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

| | |
|------------------------------|--------------------------------------|
| <code>==</code> égalité | <code>!=</code> différence |
| <code><</code> inférieur, | <code><=</code> inférieur ou égal |
| <code>></code> supérieur, | <code>>=</code> supérieur ou égal |

Instructions conditionnelles

```
if condition :  
    instructions
```

```
if condition :  
    instructions  
else :  
    instructions
```

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

Opérateurs arithmétiques

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

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` ($i \geq 0$).
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 éléments 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`)

`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 (réursive)

Aléatoire

`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

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`.

`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 (réursive)

Gestion des fichiers

`f=open('data.txt')`: ouvrir un fichier en lecture seule
`f=open('data.txt', 'w')`: ouvre un fichier en écriture (attention s'il existe il est écrasé, sinon il est créé)
`f=open('data.txt', 'a')`: ouvre un fichier en écriture (ajoute le texte à la fin)

`texte = f.read()`: lire tout le fichier en une seule fois
`lignes = f.readlines()`: lire en 1 fois toutes les lignes du fichier et les stocker dans une liste (un élém=une ligne)
`for ligne in f:`
 `instructions`

Lire le fichier ligne par ligne dans une boucle `for`

`f.write(texte)`: écrire dans un fichier (`texte` doit obligatoirement être une `string`).
Ne saute pas de ligne automatiquement à la fin du texte.
'\n' code un saut de ligne.

`f.close()`: ferme un fichier