

## INF101/INF131 : examen 2e session

Lundi 18 Juin 2018

- **Durée 2 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait 2 pages.
- Respectez **strictement** les consignes de l'énoncé (noms des fonctions et variables, format d'affichage...)
- Les exercices sont indépendants et ne sont **pas** dans l'ordre de difficulté. Vous pouvez sauter des exercices et des questions. Vous pouvez supposer existantes les fonctions des questions précédentes et les utiliser, même si vous n'avez pas répondu à la question correspondante.
- Il est inutile de filtrer les entrées lorsque ce n'est pas explicitement demandé (elles seront supposées correctes).
- **Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc, est interdite.**
- **Chaque question vaut 1 point.** Le barème est indicatif. La qualité de la rédaction et de la présentation sera prise en compte. Pensez à commenter vos programmes pour les rendre plus lisibles. Attention à l'indentation. Répondez à **chaque exercice sur une feuille séparée.**

## 1 Triplets de Pythagore (5 points)

Un triplet de Pythagore est un ensemble de 3 entiers naturels,  $a < b < c$ , tels que  $a^2 + b^2 = c^2$ . Par exemple,  $3^2 + 4^2 = 9 + 16 = 25 = 5^2$ .

1. Écrire une fonction `triplets` qui calcule et renvoie la liste des triplets de Pythagore tels que  $a$  est inférieur ou égal à une borne maximale reçue en paramètre.
2. Écrire une fonction `tripletsSomme` qui reçoit en paramètre un entier  $s$ , et affiche le triplet de Pythagore tel que  $a + b + c = s$  s'il existe, ou un message d'erreur sinon. On se servira de la fonction précédente. *Indice: quelle est la valeur maximum de  $a$  pour une somme  $s$  donnée ?*
3. Écrire un programme principal qui demande à l'utilisateur une somme, et utilise la fonction précédente pour afficher le triplet de Pythagore dont la somme correspond à celle demandée (s'il existe).

## 2 Codage de mots (4 points)

Dans cet exercice, on appelle 'mot' une chaîne de caractères constituée uniquement de lettres minuscules de l'alphabet.

- Écrire une fonction `codage1` qui reçoit en paramètre un mot, et renvoie la somme des positions de ses lettres dans l'alphabet. Par exemple pour le mot 'salut', on renverra 73 ( $19+1+12+21+20$ ).
- Écrire une fonction `codage2` qui reçoit en paramètre un mot et renvoie le mot obtenu en remplaçant chaque lettre par la suivante (le 'z' devient un 'a'). Par exemple 'salut' devient 'tbmvu'.
- Écrire une fonction `codage3` qui reçoit en paramètre un entier strictement positif, et qui renvoie le mot constitué des lettres dont la position correspond à chaque chiffre, en comptant à partir de 0 (il ne peut donc y avoir que des lettres entre a, position 0, et j, position 9). Par ex : 1203 devient 'bcad'; 2708 devient 'chai'. *Attention au sens !*
- Écrire une fonction `code4` qui reçoit en paramètre un mot, et renvoie un mot composé des mêmes lettres mais mélangées dans un ordre aléatoire. *Indice: on pourra utiliser la fonction `shuffle` du module `random`, sans oublier de l'importer.*

### 3 Démineur (6 points)

On s'intéresse à une grille de démineur carrée, représentée par une liste de listes d'entiers. Une case vide est représentée par l'entier 0, une case minée par l'entier 1. Pour éviter les effets de bord, on stocke une grille de démineur de taille  $n$  dans une liste de  $n+2$  listes de  $n+2$  entiers (les entiers placés sur les bordures de la grille valent nécessairement 0).

1. Écrire une fonction `initGrille` qui reçoit en paramètre un entier  $n$ , calcule et renvoie une liste de listes de taille  $n+2$ , ne contenant que des entiers 0.
2. Écrire une fonction `placeMines` qui reçoit en paramètre une liste de listes et un entier  $x$  (nombre de mines). Cette fonction ajoute exactement  $x$  mines dans la grille, en évitant les bordures. Cette fonction modifie directement la grille reçue en paramètre, elle ne renvoie rien. *Indice: on pourra tirer une position vide au hasard avec `random.randint(inf, sup)`.*
3. Écrire une fonction `affiche` qui reçoit en paramètre une grille d'entiers, et l'affiche sous forme d'un carré, les entiers de chaque ligne séparés par un espace, avec un retour à la ligne entre chaque ligne.
4. Écrire une fonction `lignePlusMinee` qui reçoit en paramètre une grille, cherche la ligne contenant le plus de mines, et renvoie le numéro de cette ligne et le nombre de mines qu'elle contient.
5. Écrire une fonction `ligneNonMinee` qui reçoit en paramètre une grille carrée, cherche la première ligne ne contenant aucune mine, et renvoie son numéro, ou -1 si une telle ligne n'existe pas.
6. Écrire un programme principal qui demande à l'utilisateur la taille de la grille de démineur et le nombre de mines, utilise les fonctions ci-dessus pour initialiser la grille, y placer le bon nombre de mines, et l'afficher sous forme d'un carré. Le programme affiche ensuite un message indiquant le numéro de la ligne la plus minée et son nombre de mines, et le numéro de la première ligne non minée si elle existe.

### 4 Dictionnaire de traduction (5 points)

1. Écrire une fonction `initListeMots` qui ne reçoit aucun paramètre, demande à l'utilisateur de saisir des mots français en terminant par 'stop', enregistre tous ces mots dans une liste (sauf 'stop'), et renvoie cette liste.
2. Écrire une fonction `initDicoAnglais` qui reçoit une liste de mots français, demande à l'utilisateur la traduction de chaque mot en anglais, et initialise un dictionnaire français-anglais (clé = mot français, valeur = mot anglais).
3. Écrire une fonction `supprimerDoublons` qui reçoit en paramètres un dictionnaire français-anglais et une liste de mots français; elle parcourt les mots français pour vérifier qu'il n'y a pas plusieurs mots ayant la même traduction dans le dictionnaire reçu. Si c'est le cas, alors les doublons sont supprimés du dictionnaire (on ne garde que le premier mot ayant cette traduction). Cette fonction ne renvoie rien, elle modifie le dictionnaire reçu en paramètre (effet de bord). *Indice: on a besoin de parcourir la liste des mots plutôt que les clés du dictionnaire, car on n'a pas le droit de modifier le dictionnaire pendant qu'on le parcourt.*
4. Écrire une fonction `inverseDico` qui reçoit en paramètre un dictionnaire français-anglais, et l'inverse pour créer un dictionnaire anglais-français (*on suppose qu'il n'y a pas plusieurs mots ayant la même traduction*).
5. Écrire un programme principal qui utilise les fonctions précédentes pour :
  - Créer un dictionnaire nommé `fr2en` associant des mots français (saisis par l'utilisateur) à leur traduction anglaise (saisie par l'utilisateur).
  - Créer un dictionnaire nommé `en2fr` en inversant le dictionnaire ci-dessus. On prendra soin de supprimer les doublons avant l'inversion.
  - Demander à l'utilisateur de saisir un mot et une langue ('en' ou 'fr'), puis utiliser le bon dictionnaire pour traduire le mot dans l'autre langue, et afficher le résultat.
  - Lui proposer de rejouer avec un autre mot et langue, jusqu'à ce qu'il refuse de continuer.
  - Afficher 'fin du programme' avant de se terminer.

# 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>&lt;</code> inférieur,	<code>&lt;=</code> inférieur ou égal
<code>&gt;</code> supérieur,	<code>&gt;=</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 (récursive)

## 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 (récursive)

## 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