

INF101/INF131 : examen final

Mardi 19 Décembre 2017

- **Durée 3 heures.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait 5 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.** L'examen est donc noté sur 22. 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 Nombres narcissiques (5 points)

Un nombre narcissique (ou nombre d'Armstrong de première espèce) est un entier naturel n non nul qui est égal à la somme des puissances p -ièmes de ses chiffres en base dix, où p désigne le nombre de chiffres de n :

$$n = \sum_{k=0}^{p-1} x_k 10^k = \sum_{k=0}^{p-1} (x_k)^p \quad \text{avec } x_k \in \{0, \dots, 9\} \text{ et } x_{p-1} \neq 0.$$

Par exemple, tous les entiers de 1 à 9 sont narcissiques ; 153 est narcissique car $153 = 1^3 + 5^3 + 3^3$; 548834 est narcissique car $548834 = 5^6 + 4^6 + 8^6 + 8^6 + 3^6 + 4^6$.

1. Écrire une fonction **nombreChiffres** qui calcule et renvoie le nombre p de chiffres d'un entier naturel non nul x reçu en argument.
2. Écrire une fonction **sommePuissances** qui reçoit en argument un entier naturel x , et calcule et renvoie la somme des puissances p -ièmes de ses chiffres, où p est son nombre de chiffres (qui sera calculé avec la fonction précédente).
3. Écrire une fonction **narcissique** qui reçoit en argument un entier naturel x , et renvoie un booléen indiquant si ce nombre est narcissique.
4. Écrire une fonction **listeNarcissiques** qui reçoit en argument une borne maximum, et qui construit et renvoie la liste des nombres narcissiques inférieurs ou égaux à cette borne. On rappelle qu'un entier narcissique doit être non nul. Si la borne reçue en argument est inférieure ou égale à 0, la liste renvoyée doit donc être vide.
5. Écrire un programme principal qui demande une borne maximum à l'utilisateur, puis affiche la liste des nombres narcissiques inférieurs ou égaux à cette borne, sur une seule ligne, séparés par des virgules, avec un retour à la ligne final. Par exemple :

```
bmax? 10
1,2,3,4,5,6,7,8,9
bmax? 200
1,2,3,4,5,6,7,8,9,153
```

2 Estimation du nombre π (4 points)

Il existe plusieurs formules pour calculer des estimations du nombre π . On considère en particulier dans cette exercice les deux formules suivantes :

- $\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$: cette formule de la somme alternée des inverses des nombres impairs, s'approche très très lentement du nombre π .
- $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \dots$: cette somme des inverses des carrés des entiers converge plus rapidement que la formule précédente.

En Python, on peut obtenir la valeur de π grâce à `math.pi`, ce qui nous permettra de tester la précision du calcul des estimations. On appelle précision la différence en valeur absolue entre l'estimation calculée par la formule, et la valeur de π donnée par `math.pi`. *Par exemple : `estimPi1(1)` calcule une estimation de π valant 4. La précision de cette estimation est $|4 - \pi|$ c'est-à-dire environ 0.8585.*

1. Écrire une fonction `estimPi1` qui reçoit en paramètre une borne max (l'entier impair constituant le dénominateur de la dernière fraction de la somme), calcule et renvoie l'estimation correspondante de π selon la première formule. Attention la formule permet de calculer $\pi/4$ et pas π . *Par exemple : `estimPi1(7)` estime $\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}$ puis calcule et renvoie π à partir de cette valeur.*
2. Écrire une fonction `estimPi2` qui reçoit en paramètre une borne max n (l'entier dont le carré constitue le dénominateur de la dernière fraction de la somme), calcule et renvoie l'estimation correspondante de π selon la deuxième formule. Attention la formule permet de calculer $\pi^2/6$ et pas π . On rappelle que `math.sqrt` permet de calculer la racine carrée. *Par exemple : `estimPi2(6)` estime $\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \frac{1}{6^2}$, puis calcule et renvoie π à partir de cette valeur.*
3. Écrire une fonction `estimPrecision` qui reçoit en paramètre une précision souhaitée dans l'approximation de π (c'est-à-dire l'écart maximum souhaité, en valeur absolue, entre l'estimation calculée et la valeur de π obtenue avec `math.pi`), et appelle la fonction `estimPi2` ci-dessus pour calculer les estimations successives de π (avec des valeurs croissantes de n) jusqu'à obtenir la précision souhaitée. La fonction renvoie alors la borne max n nécessaire, et l'estimation de π obtenue. *Par exemple : `estimPrecision(0.1)` renvoie (10, 3.04936163598207) ; `estimPrecision(0.01)` renvoie (96, 3.131681463952958)*
4. Écrire un programme principal qui demande à l'utilisateur une précision p (nombre réel), puis affiche la borne max nécessaire pour atteindre cette précision, et l'estimation correspondante. Le programme doit ensuite proposer à l'utilisateur de rejouer avec une nouvelle précision, jusqu'à ce que l'utilisateur refuse.

3 Jeu de la vie de Conway (8 points)

Le jeu de la vie de Conway est un automate cellulaire dont les règles sont extrêmement simples. On considère une grille carrée de cellules qui peuvent être soit vivantes soit mortes. L'évolution de l'état d'une cellule dépend de son nombre de voisins vivantes parmi les 8 cellules qui l'entourent (cf figure 1):

- Une cellule morte qui a exactement 3 voisins vivantes devient vivante (elle naît);
- Une cellule vivante qui a exactement 2 ou 3 voisins vivantes reste vivante, sinon elle meurt.

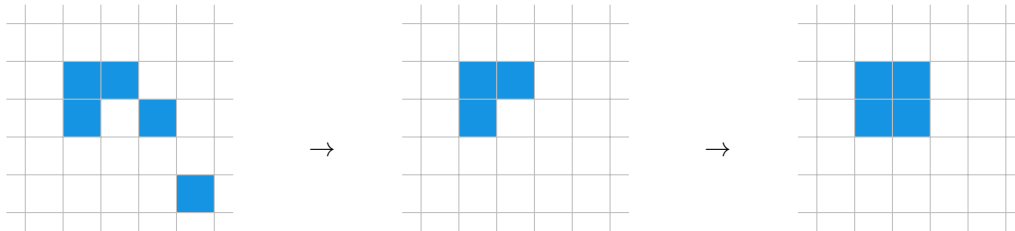


Figure 1: Un exemple d'évolution sur trois étapes d'un extrait de grille dans le jeu de la vie. Les cellules mortes sont en blanc, les cellules vivantes en bleu.

On choisit ici de représenter cette grille de la manière suivante :

- L'état de chaque cellule est un booléen : True pour vivante, False pour morte.
- La grille est une liste de listes : chaque élément de la liste est une liste contenant les états de toutes les cellules de cette ligne.
- On appelle taille de la grille le nombre de cellules par côté. Une grille de taille N est donc une liste de N listes de N éléments.

Par exemple la grille de gauche sur la figure 1 ci-dessus est de taille 5 et est représentée par la liste suivante :

```
[[False, False, False, False, False],
 [False, True,  True,  False, False],
 [False, True,  False, True,  False],
 [False, False, False, False, False],
 [False, False, False, False, True ]]
```

1. Écrire une fonction `initGrilleM(N)` qui reçoit en argument la taille N de la grille carrée, et initialise et renvoie une grille carrée de cette taille (N cellules par N cellules) contenant uniquement des cellules mortes. **Attention** à créer des listes différentes pour chaque ligne, et non pas des références à la même liste.
2. Écrire une fonction `initGrilleV(N,V)` qui appelle la fonction précédente pour créer une grille de taille N de cellules mortes, puis qui place **exactement** V cellules vivantes (à des positions au hasard), et renvoie cette grille. On suppose que V a une valeur correcte c'est-à-dire positif et inférieur ou égal à $N*N$. *Indice: il s'agit, pour chaque cellule vivante à placer, de tirer au hasard des coordonnées dans la grille, jusqu'à trouver une cellule morte, puis de rendre cette cellule vivante.*
3. Écrire une fonction `afficheSolution` qui reçoit en paramètre une grille et l'affiche de la manière suivante : les cellules vivantes sont représentées par le caractère '*' (étoile), et les cellules mortes par le caractère '.' (point), sans espace entre les valeurs. Par exemple la grille de booléens

ci-dessus (qui correspond à la grille de gauche sur la figure) sera affichée sous la forme suivante :

```
.....
.**..
.**.
.....
....*
```

4. Écrire une fonction `grilleEtendue` qui reçoit une grille de taille N , et qui calcule et renvoie une grille de taille $N+2$ en rajoutant des cellules mortes (valeur `False`) tout autour de la grille reçue. Cela permettra d'éviter les effets de bord dans le comptage des voisins vivants des cellules. *On rappelle les fonctions `insert` et `append` pour insérer un élément dans une liste ou à la fin d'une liste respectivement.* Par exemple pour la grille de taille 5 ci-dessus, cette fonction renvoie la grille de taille 7 suivante :

```
.....
.....
.**...
.**.*..
.....
.....*
.....
```

5. Écrire une fonction `compteVoisins(grille,i,j)` qui reçoit la grille de taille N , utilise la fonction ci-dessus pour calculer la grille étendue de taille $N+2$, puis compte et renvoie le nombre de voisins vivants de la cellule de la ligne d'indice i et colonne d'indice j de la grille. **Attention** : les indices reçus sont supposés corrects, et sont exprimés dans la grille de taille N , avant extension : ils sont donc compris entre 0 et $N-1$. Grâce à l'extension de la grille, toutes les cellules ont exactement 8 voisines et il n'est donc pas nécessaire de gérer des cas particuliers dans les coins ou sur les bords.

Par exemple pour la grille ci-dessus et les indices (0,1), la fonction compte les voisins vivants de la cellule de la 1e ligne et 2e colonne de la grille initiale (numérotées à partir de 0), c-à-d de la 2e ligne et 3e colonne dans la grille étendue, et renvoie 2 (une cellule vivante dessous, et une cellule vivante dessous à droite).

6. Écrire une fonction `evoluerGrille` qui reçoit la grille en argument, et crée et renvoie une nouvelle grille contenant le nouvel état de toutes les cellules après une évolution. La grille initiale n'est pas modifiée. Toutes les cellules évoluent **simultanément**, c'est-à-dire qu'on compte le nombre de voisins vivants dans l'ancienne grille (celle reçue en paramètre), afin de calculer le nouvel état qui sera stocké dans la nouvelle grille (celle renvoyée par la fonction). Pour créer la nouvelle grille, on pourra utiliser la fonction `initGrilleM` définie ci-dessus.
7. Écrire une fonction `nombreVivantes` qui reçoit une grille en argument, et parcourt cette grille pour calculer et renvoyer le nombre de cellules vivantes qu'elle contient.
8. Écrire un programme principal qui appelle les fonctions définies ci-dessus pour :
- Initialiser une grille carrée de taille 5 par 5, avec 7 cellules vivantes, et afficher cette grille initiale.
 - Faire évoluer cette grille pendant 10 tours, en affichant à chaque tour le numéro du tour, la nouvelle grille, et le nombre de cellules vivantes.
 - Ce programme s'arrête après les 10 tours, ou avant dès qu'il n'y a plus aucune cellule vivante dans la grille.

4 Dictionnaire de cryptage (5 points)

On s'intéresse dans cet exercice à des dictionnaires dont les clés sont les lettres de l'alphabet en minuscule, et dont les valeurs sont aussi des lettres minuscules de l'alphabet. Un tel dictionnaire associe chaque lettre à une lettre (elle-même ou une autre), chaque lettre n'apparaissant comme valeur que d'une seule clé, réalisant ainsi une permutation de l'alphabet. Cela permet donc de coder un mot ou un texte en remplaçant chaque lettre par une autre.

1. Écrire une fonction `initDico` qui crée et renvoie un dictionnaire qui associe aléatoirement à chaque lettre de l'alphabet une autre lettre. **Attention** : chaque lettre ne doit être utilisée qu'une seule fois. *Par exemple* si la clé "e" est associée à la valeur "a", alors aucune autre clé ne doit être associée à cette même valeur. Ce dictionnaire constitue donc une permutation de l'alphabet. On pourra par exemple utiliser la fonction `shuffle`.
2. Écrire une fonction `codeMot` qui reçoit un mot et un dictionnaire dont les clés et les valeurs sont des lettres de l'alphabet. Cette fonction calcule et renvoie ce mot codé par ce dictionnaire, c'est-à-dire dans lequel chaque lettre est remplacée par sa valeur dans le dictionnaire. On suppose que le mot reçu ne contient que des lettres minuscules.
3. Écrire une fonction `plusLointaine` qui reçoit en argument un tel dictionnaire, et qui calcule et renvoie quelle lettre (clé) est associée dans ce dictionnaire avec la lettre la plus lointaine, et la valeur de l'écart. Pour cela on calculera l'écart en valeur absolue entre la clé et la valeur associée (par exemple si la clé 'e', 5e lettre, est associée à la valeur 'a', 1e lettre, l'écart est de 4; si la clé 'i', 9e lettre, est associée à la valeur 'z', 26e lettre, l'écart est de 17). On rappelle les fonctions `ord` et `chr` permettant de convertir un caractère en code ASCII et vice-versa.
4. Écrire une fonction `dicoDecode` qui reçoit en argument un dictionnaire associant chaque lettre à son codage par une autre lettre, et qui génère et renvoie le dictionnaire permettant de **décoder** les mots codés avec ce dictionnaire. Par exemple à partir du dictionnaire 'a':'e', 'b':'c', 'c':'d', 'd':'a', 'e':'b', cette fonction renvoie le dictionnaire inversé 'a':'d', 'b':'e', 'c':'b', 'd':'c', 'e':'a'. Ainsi si on code un mot avec le premier dictionnaire, puis qu'on code le résultat avec le deuxième dictionnaire, on doit retomber sur le mot initial.
5. Écrire un programme principal qui :
 - Génère un dictionnaire aléatoire,
 - Calcule dans une variable `ll` la lettre qui est associée à la lettre la plus lointaine, et dans une variable `em` l'écart correspondant, puis affiche ces deux valeurs avec un texte clair
 - Demande à l'utilisateur un mot, le code avec ce dictionnaire et affiche le résultat,
 - Puis calcule le dictionnaire inverse, décode le mot et affiche le résultat.

Mémo Python - UE INF101 / INF131

Opérations sur les types

`type()` : pour connaître le type d'une variable
`int()` : permet la transformation en entier
`float()` : permet la transformation en flottant
`str()` : permet 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 >`
- 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

Opérateur 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> : division entière (quotient)
<code>%</code> : reste de la division entière (modulo)	

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"`

`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)

`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