

INF101/INF131 : examen final

Lundi 17 Décembre 2018

- **Durée 3 heures.**
- **Aucun document autorisé. L'utilisation d'une calculatrice, d'un téléphone, etc, est interdite.**
- **Lisez toutes les questions avant de commencer !** Le sujet fait 5 pages.
- 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.
- Respectez **strictement** les consignes de l'énoncé (noms des fonctions et variables, format d'affichage...). Il est inutile de filtrer les entrées lorsque ce n'est pas explicitement demandé (elles seront supposées correctes). Pensez aussi à commenter vos programmes pour les rendre plus lisibles.
- Le barème est indicatif.
- La qualité de la rédaction et de la présentation est prise en compte. En particulier, ne mélangez pas vos réponses à plusieurs exercices. Si vous devez revenir sur un exercice plus tard, laissez suffisamment d'espace. **Les morceaux d'exercice éparpillés ne seront pas pris en compte.**

1 Booléens (3 points)

1. Écrire les expressions booléennes correspondant aux phrases suivantes :
 - (a) La chaîne de caractères c ne contient aucune voyelle minuscule
 - (b) L'entier naturel n a au maximum k chiffres
2. Traduire en français courant (compréhensible par un non informaticien) les expressions booléennes suivantes :
 - (a) $a \% b == 0$ or $b \% a == 0$
 - (b) $\text{len}(c) == 1$ and $'a' <= c <= 'z'$
3. Donner les négations (simplifiées au maximum, sans opérateur *not*) des expressions booléennes suivantes :
 - (a) $(a \% 2 == 0$ and $c == 'oui')$ or $(a \% 2 == 1$ and $c == 'non')$
 - (b) $a < b < c$ and rep in ['oui', 'non']

2 Suite de Fibonacci et nombre d'or (4 points)

Le nombre d'or est une proportion (supposée 'parfaite' ou 'divine') utilisée en géométrie. Sa valeur est exactement $\frac{1+\sqrt{5}}{2}$ (soit environ 1.61803398875).

La suite de Fibonacci est définie par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \forall n > 1 \end{cases}$$

Elle peut être utilisée pour calculer le nombre d'or en divisant un terme par le précédent :

$$NB_OR \approx \frac{F_n}{F_{n-1}}$$

Plus l'indice n du terme augmente, plus la précision du calcul est grande (c'est-à-dire plus la différence – en valeur absolue – entre la valeur du nombre d'or et la valeur de l'approximation est faible).

1. Écrire une fonction **fibonacci(n)** qui reçoit en paramètre un entier n et renvoie le n^{ieme} terme de la suite de Fibonacci.
2. Écrire une fonction **ecartApprox(n)** qui reçoit en paramètre un entier n et qui renvoie l'écart (en valeur absolue) entre l'approximation de rang n du nombre d'or, et sa valeur "exacte" (donnée ci-dessus). *Par exemple pour $n=2$, l'approximation de rang 2 vaut $1/1$ donc 1; pour $n=3$, l'approximation vaut F_3/F_2 donc $2/1$ donc 2.*
3. Écrire une fonction **rangApproxPrecise(p)** qui reçoit en paramètre un réel p représentant l'écart maximal souhaité entre l'approximation et la valeur exacte. Cette fonction calcule et renvoie le rang n de la suite de Fibonacci permettant d'obtenir une approximation avec la bonne précision.
4. Écrire un programme principal qui :
 - Demande à l'utilisateur une précision réelle p et la filtre pour qu'elle soit comprise entre 0.0001 et 1 ;
 - Calcule le rang n permettant d'obtenir cette précision ;
 - Affiche ce rang, l'estimation obtenue, et la valeur de l'écart avec le nombre d'or, dans un message clair sous la forme : `'L'approximation de rang XXX vaut XXX; cela représente un écart de XXX avec le nombre d'or'` (en remplaçant les XXX par les valeurs calculées).

3 Codage de César (3 points)

Le codage de César consiste à décaler chaque lettre d'un texte pour la remplacer par la lettre située x positions (en général 3) plus loin dans l'alphabet. Dans cet exercice on va améliorer légèrement ce codage en utilisant une valeur de décalage différente pour chaque mot du texte : dans chaque phrase, le premier mot sera codé avec un décalage de 1 position, le deuxième mot avec un décalage de 2 positions, etc.

On suppose que le texte ne contient que des lettres non accentuées, et que les phrases sont toutes séparées par un point (pas de phrases interrogatives ou exclamatives). Dans le codage, les lettres seront décalées, et les caractères non alphabétiques (points, virgules, etc) seront laissés inchangés.

1. Codage mot:

- Écrire une fonction `decaleLettre(lettre,d)` qui reçoit en arguments une lettre (minuscule ou majuscule) de l'alphabet, et un entier d (qu'on suppose compris entre 1 et 26), et qui renvoie la lettre située d positions plus loin dans l'alphabet. Attention, il ne faut pas modifier la casse. *Par exemple, la lettre x en minuscule décalée de 3 positions devient un a, toujours en minuscule. La lettre Y majuscule décalée de 5 positions devient un D majuscule. La lettre b minuscule décalée de 3 positions devient un e minuscule.*
- Écrire une fonction `codeMot(mot,x)` qui reçoit en arguments un mot et un entier x , code ce mot par décalage de x positions, et renvoie une chaîne de caractères correspondant au mot ainsi codé. *Par exemple `codeMot('Hello',3)` renvoie la chaîne 'Khoor'.*

2. Écrire une fonction `codePhrase(p)` qui reçoit une phrase et qui renvoie la phrase codée comme expliqué ci-dessus : le premier mot est codé avec un décalage de 1, le 2e mot avec un décalage de 2, etc; les espaces et autres caractères non alphabétiques ne sont pas modifiés. On utilisera la fonction `codeMot` codée précédemment.
3. Écrire une fonction `motPlusLong(p)` qui reçoit une phrase et qui renvoie le mot le plus long de cette phrase et la taille de ce mot.

4 Calcul de polynômes (6 points)

Dans cet exercice, on représente un polynôme de degré n par une liste de $n+1$ éléments représentant les coefficients de ses différents éléments. On suppose que les coefficients sont des entiers. *Par exemple le polynôme de degré 5 suivant: $6x^5 + x^3 + x^2 - 7x + 1$ est représenté par la liste de 6 entiers relatifs suivante: $[1, -7, 1, 1, 0, 6]$. Le polynôme de degré 2 suivant: $3x^2 - 5$ est représenté par la liste de 3 entiers suivante: $[-5, 0, 3]$.*

Le dernier élément de la liste est donc forcément non nul, il correspond au coefficient du monôme de plus haut degré. Il peut y avoir des 0 dans la liste, mais on ne stocke pas les 0 'inutiles' en fin de liste. *Par exemple $[1, 1, 1, 0, 0]$ et $[1, 1, 1]$ représentent le même polynôme $x^2 + x + 1$, les coefficients 0 pour les monômes de degré 3 et 4 sont inutiles.* Ainsi le polynôme nul est donc représenté par une liste vide.

1. Écrire une fonction `lire` qui ne prend pas d'arguments, qui demande à l'utilisateur le degré du polynôme, puis lui demande les différents coefficients dans l'ordre décroissant des degrés, et qui renvoie la liste représentant ce polynôme. Par exemple:

```
>>> lire()
Degré du polynome ? 2
Coeff du monome de degre 2 ? 1
Coeff du monome de degre 1 ? -3
Coeff du monome de degre 0 ? 5
[5,-3,1]
```

2. Écrire une fonction `evaluer` qui reçoit en paramètre une liste d'entiers `p` et un réel `x`, qui calcule la valeur au point `x` du polynôme représenté par cette liste `p`, et qui renvoie cette valeur. *Par exemple, l'évaluation du polynôme représenté par la liste $[1, 5, -3]$ au point 2 vaut $1 * 2^2 + 5 * 2 - 3$ donc 11.*
3. Écrire une fonction `simplifier` qui reçoit en paramètre une liste d'entiers `p` représentant un polynôme, et qui la simplifie en retirant les éventuels 0 inutiles. Cette fonction modifie directement la liste, et elle renvoie le degré du polynôme. *Par exemple:*

```
>>> l = [1,0,5,0,0]
>>> simplifier(l)
2
>>> l
[1, 0, 3]
```

4. Écrire une fonction `somme` qui reçoit en paramètre deux listes `p1` et `p2` représentant 2 polynômes, et qui renvoie une liste `p` représentant leur somme. Cette fonction ne doit pas modifier les polynômes reçus en arguments. Le polynôme renvoyé ne doit pas contenir de 0 inutiles.
5. Écrire une fonction `produit` qui reçoit en paramètre deux listes `p1` et `p2` représentant 2 polynômes, et qui renvoie une liste `p` représentant leur produit. Cette fonction ne doit pas modifier les polynômes reçus en arguments. Le polynôme renvoyé ne doit pas contenir de 0 inutiles.
6. Écrire une fonction `derivee` qui reçoit une liste représentant un polynôme, calcule et renvoie une nouvelle liste représentant la dérivée de ce polynôme. *Par exemple la dérivée du polynôme $3x^2 + x + 2$, représenté par la liste $[2, 1, 3]$ est le polynôme $6x + 1$ représenté par la liste $[1, 6]$.*
7. Écrire un programme principal qui demande à l'utilisateur de saisir 2 polynômes, et calcule et affiche leur produit, leur somme, et leur dérivée (sous forme de listes). Ce programme utilise les fonctions définies ci-dessus.

5 Calculatrice préhistorique (4 points)

Dans cet exercice, nous allons recréer tous les opérateurs arithmétiques sur les **entiers naturels** à partir d'une seule fonction de départ (la soustraction). On interdit donc d'utiliser les opérateurs arithmétiques de Python (+, -, *, /, **, //), ainsi que les fonctions du module math (valeur absolue, factorielle, etc). On suppose qu'on dispose d'une fonction *soustraction(a, b)* qui renvoie la différence entre a et b (a-b). Cette fonction nous suffit pour définir toute l'arithmétique de base pour les entiers et plus encore !

1. Opérateurs simples:

- En utilisant la fonction *soustraction*, définir la fonction *addition(a, b)*, qui calcule la somme de ses arguments. On rappelle qu'il ne faut utiliser **aucun** opérateur arithmétique de Python.
- En utilisant la fonction *addition*, définir la fonction *multiplication(a, b)* qui calcule le produit de ses arguments.

2. Puissance et factorielle:

- Définir la fonction *puissance(a, n)* qui renvoie a^n (sans utiliser l'opérateur ** de Python). Les arguments a et n sont des entiers naturels.
- Définir la fonction *factorielle(a)* qui renvoie la factorielle de son argument (sans utiliser la fonction *factorial* du module *math*).

3. En utilisant les fonctions précédemment définies, écrire une fonction *division(a, b)* qui retourne deux valeurs: le quotient de la division entière de a par b, ainsi que le reste de cette division. On rappelle que a et b sont des entiers naturels.

4. Écrire le programme principal de notre calculatrice, qui :

- Lit (et filtre) un premier entier naturel A
- Lit un opérateur parmi ceux supportés ("+": addition; "-": soustraction; "*": multiplication; "/": division; "**": puissance; "!": factorielle), avec filtrage tant que l'opérateur saisi est inconnu
- Si nécessaire (et uniquement si nécessaire) lit (et filtre) un deuxième entier naturel B (par exemple la factorielle ne nécessite pas de 2e opérande)
- Affiche le résultat du calcul demandé
- Se relance et attend que l'utilisateur saisisse un nouveau calcul

Exemples d'exécution:

A=5	A=5
+	!
B=8	120
13	A=5
	\$
	Opérateur inconnu, on recommence !
A=12	
/	A=5
B=-5	**
Erreur, B=5	B=5
(2, 2)	3125

Mémo Python - UE INF101 / INF131 / INF204 - version 2018

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)
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)

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