Université de Lille — FACULTÉ DES SCIENCES ET TECHNOLOGIES — Département Informatique

L1 MIASH/SESI/PEIP - 2020–2021

Algorithmes et Programmation 1

DS - 19/05/2021

durée 2h

Ce devoir contient quatre exercices indépendants. Indiquez votre parcours et le numéro de votre groupe sur votre copie.

Sauf mention explicite du contraire, il n'est pas attendu de documentation complète pour les fonctions demandées mais uniquement les **conditions d'utilisation (CU)**. En revanche, toute autre fonction non explicitement demandée, mais dont vous estimez avoir besoin, doit être documentée.

Exercice 1 Foobarnacci

On considère la variable globale `FOO` et la fonction `bar` définies ci dessous :

Fonction mystérieuse

```
1  FOO = {0: 1, 1: 1}
2
3  @trace
4  def bar(n):
5      assert n >= 0
6      if n in FOO:
7          res = FOO[n]
8      else:
9          res = bar(n-1) + bar(n-2)
10         FOO[n] = res
11     return res
```

Question 1 Donnez la trace obtenue lors de l'appel de `bar(5)`, ainsi que la valeur de `FOO` à la fin de l'appel.

Exercice 2 Commande Unix `wc`

La commande Unix `wc` permet de compter le nombre de lignes, de mots et de caractères dans un fichier texte.

Supposons que le fichier `dsf/epitaphe.txt` contiennent les deux lignes suivantes :

```
Ci-gît le Seigneur de La Palice
S'il n'était mort il ferait encore envie
```

En utilisant la commande `wc -l dsf/epitaphe.txt`, on obtient le nombre 2, car ce fichier contient 2 lignes.

Question 1 Définissez une fonction `wc_l`, paramétrée par une chaîne `nom` et renvoyant le nombre de lignes du fichier dont le nom est contenu dans `nom`.

```
>>> wc_l('dsf/epitaphe.txt')
2
```

Exercice 3 Notes de devoir surveillé

Les parties de cet exercice peuvent être résolues indépendemment.

Lors de la correction du devoir surveillé, le barème n'est pas encore fixé et les enseignants renseignent dans un fichier csv les résultats.

Voici les trois premières lignes d'un tel fichier :

```
"NIP";"Nom";"Prénom";"Q1";"Q2";"Q3";"Q4";"CU"
"98765432";"Calbuth";"Raymond";0;0.5;0;0.25;0.5
"42036656";"Thergal";"Jean-Claude";0;1;0;0;0
```

La première ligne contient les *noms de colonnes* séparés par des points virgules ;. Les lignes suivantes contiennent chacune une *série de données*, qui est une séquence d'éléments séparés par des points virgules ; et associés aux noms de colonnes.

Dans les séries de données, on trouve le NIP, le nom, le prénom, et les pourcentages de points dans chacune des questions.

Par exemple, la deuxième ligne indique que l'étudiant Raymond Calbuth (NIP 98765432) a eu 0% des points de la question 1 ; 50% des points de la question 2 ; 0% des points de la question 3 ; 25% des points de la question 4 et 50% des points pour les CU.

Question 1 Définissez une fonction nommée `dictionnaire` paramétrée par deux listes de chaînes `cles` et `valeurs`, et renvoyant un dictionnaire dont :

- les clés sont les éléments de `cles` auxquels on a retiré les guillemets ;
- les valeurs associées à ces clés sont obtenues :
  - si l'élément de `valeur` est encadré par des guillemets, alors la valeur sera cet élément auquel on a retiré les guillemets ;
  - si l'élément de `valeur` n'est pas encadré par des guillemets, alors la valeur sera la conversion en flottant de cet élément.

```
>>> dictionnaire(['"cle 1"', '"cle 2"', '"cle 3"'], ['"val 1"', '2', '3.0'])
{'cle 1': 'val 1', 'cle 2': 2.0, 'cle 3': 3.0}
```

Question 2 Définissez une fonction `lire_csv` qui lit le fichier dont le nom est contenu dans la chaîne passée en paramètre et qui renvoie une liste de dictionnaires dont les clés sont les noms de colonnes et les valeurs associées les éléments de la série de données de la ligne.

Par exemple, la liste obtenue pour le fichier donné en exemple sera :

```
[{"NIP": "98765432", "Nom": "Calbuth", "Prénom": "Raymond", "Q1": 0.0,
  "Q2": 0.5, "Q3": 0.0, "Q4": 0.25, "CU": 0.5},
 {"NIP": "42036656", "Nom": "Thergal", "Prénom": "Jean-Claude", "Q1": 0.0,
  "Q2": 1.0, "Q3": 0.0, "Q4": 0.0, "CU": 0.0},
 ...
]
```

Pour le calcul de la note finale, le barème est représenté par un dictionnaire dont les clés sont les noms des questions et les valeurs associées sont les nombres de points.

Question 3 Définissez une fonction `note_finale` paramétrée par deux dictionnaires, le premier représentant un barème, le second des pourcentages de notes, et renvoyant la note obtenue.

```
>>> note_finale({"Q1": 4, "Q2": 4, "Q3": 5, "Q4": 4, "CU": 1},
...             {"NIP": "98765432", "Nom": "Calbuth", "Prénom": "Raymond",
...              "Q1": 0.0, "Q2": 0.5, "Q3": 0.0, "Q4": 0.25, "CU": 0.5})
3.5
```

(En effet, 0% de 4 points plus 50% de 4 points plus 0% de 5 points plus 25% de 4 points plus 50% de 1 points font 3,5 points.)

Question 4 Définissez une fonction `liste_notes` paramétrée par un dictionnaire `bareme` et une liste de dictionnaires dont les clés sont `'NIP'`, `'Prénom'`, `'Nom'` et toutes les clés de barème, et qui renvoie une liste de dictionnaires dont les clés sont `'NIP'`, `'Prénom'`, `'Nom'` et `'Note finale'` qui contient les notes finales calculées à partir du barème.

La liste renvoyée devra être triée par ordre croissant de nom, puis de prénom.

Exercice 4 Élément majoritaire

On s'intéresse dans cet exercice au problème suivant :
Dans une liste `l`, existe-t-il un élément majoritaire ?
C'est-à-dire, si n est le nombre d'éléments de la liste, existe-t-il un élément x dont le nombre d'occurrences n_x vérifie $n_x > \frac{n}{2}$.

Question 1 Réaliser une fonction `nb_occurrences(l, x, a, b, cmp)` prenant en paramètre une liste `l`, un élément `x` et une fonction de comparaison `cmp` et qui renvoie le nombre d'occurrences de `x` dans la liste `l[a:b]`. On suppose que `x` est comparable avec les éléments de `l` avec la fonction `cmp`.

Question 2 Peut-il y avoir plus de deux éléments différents majoritaires ?
En déduire un algorithme `a_majoritaire_v1(l, cmp)` paramatré par une liste `l` et une fonction de comparaison `cmp` et qui renvoie `True` si la liste possède un élément majoritaire et `False` sinon. Les éléments de la liste peuvent être mutables.

```
>>> a_majoritaire_v1([1,2,1,4,1], compare)
True
>>> a_majoritaire_v1([1,2,1,2], compare)
False
```

Question 3 Donner le nombre de comparaisons effectuées par votre fonction `a_majoritaire_v1` en fonction du nombre n d'éléments de la liste dans le meilleurs et dans le pire des cas.

Question 4 Dans cette question, on n'utilise pas de fonction de comparaison. Lorsque les éléments de la liste ne sont pas mutables, alors il est possible d'améliorer le nombre de comparaisons dans le pire des cas en utilisant un dictionnaire.
Rappeler la méthode utilisée permettant d'effectuer n tests d'appartenance à un dictionnaire.

Question 5 Lorsque les éléments de la liste sont mutables, la méthode précédente ne peut plus être utilisée. On propose ici une méthode récursive fournie en pseudo-code.

Entrée : une liste li, deux indices a et b , une fonction de comparaison cmp
Sortie : renvoie un couple (x, n_x) si x admet un élément majoritaire, $(None, 0)$ sinon.

1: **Function** a_majoritaire_v2(li, a, b, cmp)
2: initialiser res à `(None, 0)`
3: **si** $b - a > 1$ **alors**
4: $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
5: $(x, n_x) \leftarrow$ a_majoritaire_v2(li, a, m, cmp)
6: $(y, n_y) \leftarrow$ a_majoritaire_v2(li, m, b, cmp)
7: **si** $n_x > 0$ **alors**
8: $n'_x \leftarrow n_x +$ nb_occurrences(li, x, m, b, cmp)
9: **si** $2 \times n'_x > b - a$ **alors**
10: $res \leftarrow (x, n'_x)$
11: **fin si**
12: **fin si**
13: **si** $n_y > 0$ **alors**
14: $n'_y \leftarrow n_y +$ nb_occurrences(li, y, a, m, cmp)
15: **si** $2 \times n'_y > b - a$ **alors**
16: $res \leftarrow (y, n'_y)$
17: **fin si**
18: **fin si**
19: **sinon**
20: **si** $b - a = 1$ **alors**
21: $res \leftarrow (li[a], 1)$
22: **fin si**
23: **fin si**
24: **Renvoyer** res
25: **EndFunction**

Réaliser une implantation de cet algorithme en Python.

Mémento Bases Python 3

Documentation officielle Python sur http://docs.python.org/py3k

Types de base

entier, flottant, booléen, chaîne

```
  int 783    0    -192
float 9.23   0.0   -1.7e-6        (-1.7e-6 = 10^-6)
 bool True   False
  str "Un\nDeux"   'L\'âme'
```

retour à la ligne — ' échappé

multiligne : `"""X\tY\tZ 1\t2\t3"""` (tabulation)

non modifiable, séquence ordonnée de caractères

Types Conteneurs

- séquences ordonnées, accès index rapide, valeurs répétables

```
  list [1,5,9]   ["x",11,8.9]   ["mot"]   []
 tuple (1,5,9)   11,"y",7.4     ("mot",)  ()
```

non modifiable → `str` en tant que séquence ordonnée de caractères

expression juste avec des virgules

- sans ordre *a priori*, clé unique, accès par clé rapide ; clés = types de base ou tuples

```
  dict {"clé":"valeur"}                          {}
```

dictionnaire *couples clé/valeur*

```
       {1:"un",3:"trois",2:"deux",3.14:"π"}
```

ensemble

```
   set {"clé1","clé2"}   {1,9,3,0}   set()
```

Identificateurs

pour noms de variables, fonctions, modules, classes…

`a..zA..Z_` suivi de `a..zA..Z_0..9`

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

☺ `a toto x7 y_max BigOne`
☹ ~~`8y and`~~

Affectation de variables

```
x = 1.2+8+sin(0)
```

valeur ou expression de calcul

nom de variable (identificateur)

```
y,z,r = 9.2,-7.6,"bad"
```

noms de variables — conteneur de plusieurs valeurs (ici un tuple)

`x+=3` incrémentation, `x-=2` décrémentation

`x=None` valeur constante « non défini »

Conversions

`type(`*expression*`)`

```
int("15")        on peut spécifier la base du nombre entier en 2nd paramètre
int(15.56)       troncature de la partie décimale (round(15.56) pour entier arrondi)
float("-11.24e8")
str(78.3)        et pour avoir la représentation littérale → repr("Texte")
```

voir au verso le formatage de chaînes, qui permet un contrôle fin

`bool` → utiliser des comparateurs (avec `==`, `!=`, `<`, `>`, …), résultat logique booléen

```
list("abc") → ['a','b','c']        (utilise chaque élément de la séquence en paramètre)
dict([(3,"trois"),(1,"un")]) → {1:'un',3:'trois'}
set(["un","deux"]) → {'un','deux'}  (utilise chaque élément de la séquence en paramètre)
":".join(['toto','12','pswd']) → 'toto:12:pswd'
```

chaîne de jointure — séquence de chaînes

```
"des mots espacés".split() → ['des','mots','espacés']
"1,4,8,2".split(",") → ['1','4','8','2']
```

chaîne de séparation

Indexation des séquences

pour les listes, tuples, chaînes de caractères,…

```
index négatif   -6   -5   -4    -3    -2    -1
index positif    0    1    2     3     4     5
lst=           [11,  67,  "abc", 3.14, 42,  1968]
tranche positive 0    1    2     3     4     5     6
tranche négative -6   -5   -4    -3    -2    -1
```

`len(lst)` → 6

accès individuel aux éléments par `[`*index*`]`

`lst[1]`→67 `lst[0]`→11 *le premier*

`lst[-2]`→42 `lst[-1]`→1968 *le dernier*

accès à des sous-séquences par `[`*tranche début*`:`*tranche fin*`:`*pas*`]`

```
lst[:-1]→[11,67,"abc",3.14,42]
lst[1:-1]→[67,"abc",3.14,42]
lst[::2]→[11,"abc",42]
lst[:]→[11,67,"abc",3.14,42,1968]
lst[1:3]→[67,"abc"]
lst[-3:-1]→[3.14,42]
lst[:3]→[11,67,"abc"]
lst[4:]→[42,1968]
```

Indication de tranche manquante → à partir du début / jusqu'à la fin.

Sur les séquences modifiables, utilisable pour suppression `del lst[3:5]` *et modification par affectation* `lst[1:4]=['hop',9]`

Logique booléenne

Comparateurs: `<` `>` `<=` `>=` `==` `!=` (≤ ≥ = ≠)

`a and b` et logique — *les deux en même temps*

`a or b` ou logique — *l'un ou l'autre ou les deux*

`not a` non logique

`True` valeur constante vrai

`False` valeur constante faux

Blocs d'instructions

instruction parente :
  bloc d'instructions 1…
instruction parente :
  bloc d'instructions 2…
instruction suivante après bloc 1

indentation !

Instruction conditionnelle

bloc d'instructions exécuté uniquement si une condition est vraie

`if` *expression logique* `:`
→ *bloc d'instructions*

combinable avec des sinon si, sinon si... et un seul sinon final, exemple :

```
if x==42:
    # bloc si expression logique x==42 vraie
    print("vérité vraie")
elif x>0:
    # bloc sinon si expression logique x>0 vraie
    print("positivons")
elif bTermine:
    # bloc sinon si variable booléenne bTermine vraie
    print("ah, c'est fini")
else:
    # bloc sinon des autres cas restants
    print("ça veut pas")
```

Maths

nombres flottants… valeurs approchées ! — *angles en radians*

Opérateurs: `+` `-` `*` `/` `//` `%` `**` (× ÷ ÷ entière, reste ÷, a^b)

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0   √
log(e**2)→2.0   etc. (cf doc)
```

Instruction boucle conditionnelle

bloc d'instructions exécuté tant que la condition est vraie

```
while expression logique:
    bloc d'instructions
```

```
s = 0
i = 1
```
*initialisations **avant** la boucle*

condition avec au moins une valeur variable (ici `i`)

```
while i <= 100:
    # bloc exécuté tant que i ≤ 100
    s = s + i**2
    i = i + 1
```
faire varier la variable de condition !

```
print("somme:",s)
```
résultat de calcul après la boucle

$$s = \sum_{i=1}^{i=100} i^2$$

attention aux boucles sans fin !

Contrôle de boucle

`break` *sortie immédiate*

`continue` *itération suivante*

Instruction boucle itérative

bloc d'instructions exécuté pour chaque élément d'un conteneur ou d'un itérateur

```
for variable in séquence:
    bloc d'instructions
```

Parcours des **valeurs** de la séquence

```
s = "Du texte"
cpt = 0
```
*initialisations **avant** la boucle*

variable de boucle, valeur gérée par l'instruction `for`

```
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```
Comptage du nombre de `e` dans la chaîne.

boucle sur dict/set = boucle sur séquence des clés

utilisation des tranches pour parcourir un sous-ensemble de la séquence

Parcours des **index** de la séquence

- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11,18,9,12,23,4,17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val> 15:
        perdu.append(val)
        lst[idx] = 15
print("modif:",lst,"-modif:",perdu)
```
Bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

Parcours simultané **index** et **valeur** de la séquence:

```
for idx,val in enumerate(lst):
```

Affichage / Saisie

```
print("v=",3,"cm :",x,",",y+4)
```

éléments à afficher : valeurs littérales, variables, expressions

Options de `print`:

- `sep=" "` (séparateur d'éléments, défaut espace)
- `end="\n"` (fin d'affichage, défaut fin de ligne)
- `file=f` (print vers fichier, défaut sortie standard)

```
s = input("Directives:")
```

`input` retourne toujours une **chaîne**, la convertir vers le type désiré (cf encadré *Conversions* au recto).

Opérations sur conteneurs

```
len(c) → nb d'éléments
min(c)  max(c)  sum(c)
sorted(c) → copie triée
val in c → booléen, opérateur in de test de présence (not in d'absence)
enumerate(c) → itérateur sur (index,valeur)
```

*Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les **clés**.*

*Spécifique aux **conteneurs de séquences** (listes, tuples, chaînes) :*

```
reversed(c) → itérateur inversé   c*5 → duplication   c+c2 → concaténation
c.index(val) → position           c.count(val) → nb d'occurences
```

Opérations sur listes

modification de la liste originale

`lst.append(item)`	ajout d'un élément à la fin
`lst.extend(seq)`	ajout d'une séquence d'éléments à la fin
`lst.insert(idx,val)`	insertion d'un élément à une position
`lst.remove(val)`	suppression d'un élément à partir de sa valeur
`lst.pop(idx)`	suppression de l'élément à une position et retour de la valeur
`lst.sort()` `lst.reverse()`	tri / inversion de la liste *sur place*

Opérations sur dictionnaires

```
d[clé]=valeur      d.clear()
d[clé]→valeur      del d[clé]
d.update(d2)   mise à jour/ajout des couples
d.keys()
d.values()     vues sur les clés, valeurs, couples
d.items()
d.pop(clé)
```

Opérations sur ensembles

Opérateurs:

- `|` → union (caractère barre verticale)
- `&` → intersection
- `-` `^` → différence/diff symétrique
- `<` `<=` `>` `>=` → relations d'inclusion

```
s.update(s2)
s.add(clé)  s.remove(clé)
s.discard(clé)
```

Fichiers

stockage de données sur disque, et relecture

```
f = open("fic.txt","w",encoding="utf8")
```

variable fichier pour les opérations

nom du fichier sur le disque (+chemin…)

mode d'ouverture
- `'r'` lecture (read)
- `'w'` écriture (write)
- `'a'` ajout (append)…

encodage des caractères pour les fichiers textes: `utf8` `ascii` `latin1` …

cf fonctions des modules `os` et `os.path`

en **écriture**

```
f.write("coucou")
```

*fichier texte → lecture / écriture de **chaînes** uniquement, convertir de/vers le type désiré*

en **lecture**

```
s = f.read(4)
```
chaîne vide si fin de fichier

si nb de caractères pas précisé, lit tout le fichier

```
s = f.readline()
```
lecture ligne suivante

```
f.close()
```
ne pas oublier de refermer le fichier après son utilisation !

Fermeture automatique Pythonesque : `with open(…) as f:`

très courant : boucle itérative de lecture des lignes d'un fichier texte :

```
for ligne in f :
    bloc de traitement de la ligne
```

Génération de séquences d'entiers

très utilisé pour les boucles itératives for

```
range([début,] fin [,pas])
```
par défaut 0 — non compris

```
range(5)   → 0 1 2 3 4
range(3,8) → 3 4 5 6 7
range(2,12,3) → 2 5 8 11
```

`range` retourne un « générateur », faire une conversion en liste pour voir les valeurs, par exemple:

```
print(list(range(4)))
```

Définition de fonction

nom de la fonction (identificateur)

paramètres nommés

```
def nomfct(p_x,p_y,p_z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```
valeur résultat de l'appel. si pas de résultat calculé à retourner : `return None`

les paramètres et toutes les variables de ce bloc n'existent que *dans* le bloc et *pendant* l'appel à la fonction (« *boite noire* »)

Appel de fonction

```
r = nomfct(3,i+2,2*i)
```
un argument par paramètre

récupération du résultat retourné (si nécessaire)

Formatage de chaînes

directives de formatage — valeurs à formater

```
"modele{} {} {}".format(x,y,r) → str
"{sélection:formatage!conversion}"
```

- **Sélection** :
  `2`
  `x`
  `0.nom`
  `4[clé]`
  `0[2]`
- **Formatage** :

car-rempl. *alignement* *signe* *larg.mini*`.`*précision~larg.max* *type*

`< > ^ =` `+ - espace` `0` au début pour remplissage avec des 0

entiers: `b` binaire, `c` caractère, `d` décimal (défaut), `o` octal, `x` ou `X` hexa…

flottant: `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut), `%` pourcentage

chaîne : `s` …

- **Conversion** : `s` (texte lisible) ou `r` (représentation littérale)

Exemples

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("L'ame")
→'"L\'ame"'
```