UNIVERSITÉ LIBRE DE BRUXELLES Juin 2015
École polytechnique de Bruxelles

INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Interrogation de juin - **Solutions**

Question 1 - Run Time (2 points)

Expliquer comment fonctionne à l'exécution le "run time" d'un programme Python. En particulier, expliquer, grâce à des diagrammes d'état, comment le programme récursif ci-dessous fonctionne.

```
def evalue(v):
    if type(v) is not list:
        res= v
    elif v[1]=='+':
        res = evalue(v[0]) + evalue(v[2])
    elif v[1]=='-':
        res = evalue(v[0]) - evalue(v[2])
    elif v[1] == '*':
        res = evalue(v[0]) * evalue(v[2])
    elif v[1]=='/':
        res = evalue(v[0]) / evalue(v[2])
    else:
        res = None # error
    return res

exemple_exp = [3, '+', [4, '*', 5]]
res= evalue(exemple_exp)
```

Solution

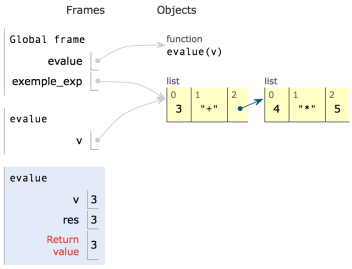
La gestion des objets et des espaces de nom en mémoire implique deux espaces mémoire :

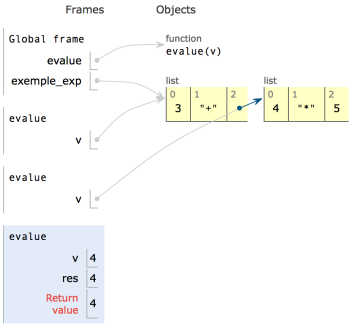
1. l'un, appelé pile d'exécution (runtime stack) qui va contenir l'espace de nom global et les espaces de noms locaux ;
2. l'autre, tas d'exécution (runtime heap), qui contient les objets.

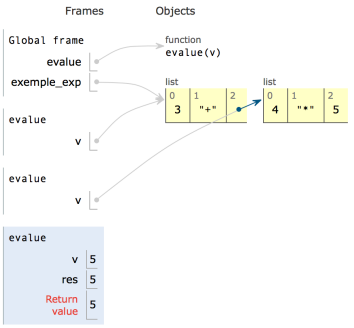
La gestion des espaces de nom locaux (ainsi que d'autres éléments en mémoire, nécessaires pour le bon fonctionnement du programme) est effectuée dans des trames (frames) dans la pile d'exécution (runtime stack) : chaque fois qu'une instance de fonction `evalue()` est appelée, une trame associée à cette instance est créée et mise comme un élément supplémentaire de la pile d'exécution. Cette trame contient en particulier, l'espace de nom local à cette instance. Cette trame sur la pile d'exécution continuera a exister jusqu'à la fin de l'exécution de cette instance de `evalue()`. Ensuite (au moment du return), la trame est enlevée de la pile système et on revient à l'instance appelante dont la trame se trouve maintenant au sommet de la pile d'exécution.

Par contre un objet créé, continue à exister jusqu'à ce qu'il ne soit plus relié à aucun nom dans un espace de noms quelconque et que le système décide de récupérer l'espace mémoire qui lui était attribué. Cette opération est nommée garbage collection et s'effectue de façon transparente pour le programmeur, si ce n'est qu'il peut observer un ralentissement ponctuel du programme pendant que le garbage collector s'exécute. C'est en raison de la gestion sans ordre de cet espace mémoire où se trouvent les objets Python, qu'il est appelé le tas d'exécution (runtime heap).

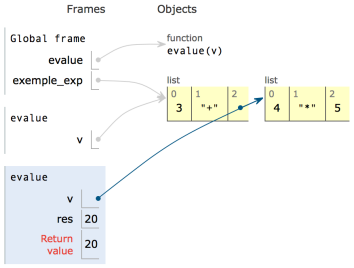
Diagrammes d'états lors de l'exécution du code donné

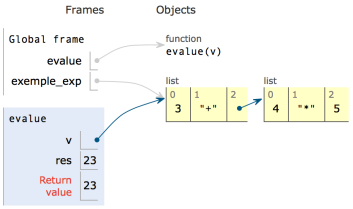






vspace-2cm





Question 2 - Fonctionnement de Python 2 (2 points)

Expliquer le mode de fonctionnement d'un compilateur et d'un interpréteur. Python de base est-il interprété ou compilé ? Expliquez quels en sont les avantages et les inconvénients pour ce langage.

Solution

Voir syllabus chapitre 14 et cours ex-cathedra.

Question 3 - Complexité (3 points)

En supposant que l'exécution se passe sans erreur, donnez la complexité moyenne et maximale de chaque instruction du code suivant (en fonction des input). **Expliquer** pourquoi nous avons ces complexités (une réponse sans explication n'a pas de valeur).

```
|                                                    # moy       max
```

```
t = list(input())                                          # O(n)      O(n) (taille de l'input)
d = dict(zip(range(len(t)),t))                             # O(n)      O(n^2)
# aide: si t = ['b','o','n','j','o','u','r']
#       d = {0: 'b', 1: 'o', 2: 'n', 3: 'j', 4: 'o', 5: 'u', 6: 'r'}
i=int(input())                                             # O(1)      O(1)
x=t[i]                                                     # O(1)      O(1)
t.append('!')                                              # O(1)      O(n)
d[len(t)]= '!'                                             # O(1)      O(n)
print(d[len(t)])                                           # O(1)      O(n)
del t[-1]                                                  # O(1)      O(1)
del d[len(t)+1]                                            # O(1)      O(n)
j = 1                                                      # O(1)      O(1)
while j < i:                                                    # O(log i)      O(log i)
   print(j)                                                # O(1)      O(1)
   j = 2*j                                                 # O(1)      O(1)
j = 2                                                      # O(1)      O(1)
while j < i:                                                    # O(log log i)    O(log log i)
   print(j)                                                # O(1)      O(1)
   j = j*j                                                 # O(1)      O(1)
```

Explications

- Tous les éléments d'une liste[1] est stockée sur le heap run time de façon séquentielle. Par ailleurs la longueur (len) d'une liste est un attribut de cette liste ; son accès prend donc un temps d'exécution en $O(1)$. Rajouter un élement d'une liste est donc normalement en $O(1)$ mais si par malchance, la liste ne peut être allongée en mémoire, il faut la recopier ailleurs, ce qui prend un temps en $O(n)$ (n étant la taille de cette liste).
- Accéder à un élément ou rajouter un élémént d'un dictionnaire prend en moyenne un temps en $O(1)$, mais en cas de collisions (ou si le dictionnaire est rempli en cas d'insertion), le pire des cas est en $O(n)$.
- la première boucle à son indice j multiplié par 2 à chance itération : on atteidra donc i quand on aura plus ou moins $2^m = i$ où m est le nombre d'étapes ; c'est-à-dire après $log_2 i$ étapes. D'où $O(logi)$, la base n'ayant pas d'importance pour le $O()$
- la seconde boucle à son indice j multiplié par j à chance itération : on atteidra donc i quand on aura plus ou moins $(...((2^2)^2)...)^2 = i = 2^(2^m))$ où m est le nombre d'étapes ; donc $m = log_2 log_2 i$. D'où $O(loglogi)$, la base n'ayant pas d'importance pour le $O()$

Question 4 - Manipulation de fichiers et de données (5 points)

Écrivez une fonction `fusion_h(fname_1, fname_2)` qui **affiche** (la fonction ne renvoie donc rien) le contenu de deux fichiers en concaténant les lignes correspondantes. Chaque ligne affichée est obtenue en mettant bout-à-bout les lignes correspondantes des deux fichiers et en les séparant par un espace.

Exemple :

Fichier `f1.txt` :

```
Ceci est un fichier
qui contient un texte
court

Le but est d'illustrer
le fonctionnement du programme
```

Fichier `f2.txt` :

```
Un fichier
quelconque et

sans importance
```

Résultat de `fusion_h("f1.txt", "f2.txt")` :

```
Ceci est un fichier Un fichier
qui contient un texte quelconque et
court
sans importance
Le but est d'illustrer
le fonctionnement du programme
```

Note : Vous pouvez utiliser la fonction `st.rstrip()` sur une chaine de caractères `st` pour retirer les "blancs" (espaces, sauts de lignes, tabulation...) qui se trouvent à la fin de cette chaine ("à droite").

Solution

1. Dans l'environnemnt Python utilisée

```
def groupAndPrint(ls1, ls2):
    lgmin = min(len(ls1), len(ls2))
    for i in range(lgmin):
        print(ls1[i], ls2[i])
    for i in range(lgmin, len(ls1)):
        print(ls1[i])
    for i in range(lgmin, len(ls2)):
        print(ls2[i])

def fusionH(filename1, filename2):
    ls1 = lines(filename1)
    ls2 = lines(filename2)
    groupAndPrint(ls1, ls2)
```

Question 5 - Tri (4 points)

On vous demande d'écrire une fonction `sorted_rands(n)` qui génère une liste *triée* de `n` nombres aléatoires dans l'intervalle $[0, 1[$. La fonction doit impérativement construire la liste *au fur et à mesure de la génération* des nouvelles valeurs et non pas en deux étapes (génération d'une liste de nombres aléatoires, puis tri de cette liste générée). Pratiquement, nous vous suggérons d'adapter la méthode de tri par insertion.

Notes :

— Utilisez la fonction `random.random()` de la librairie `random` qui génère une valeur aléatoire dans l'intervalle $[0, 1[$.

— Vous ne pouvez pas utiliser de fonction de tri existante du type `sort` ou `sorted`.

Exemple :

```
>>> sorted_rands(5)
[0.1525017040560619, 0.18414999412515853, 0.323816213456277, 0.5775189595078108, 0.9786976725577308]
```

Solution

```
import random

def posInsert(val, ls):
    """La position d'insertion de val dans ls
    ls doit être triée
    """
    k = 0
    n = len(ls)
    while k < n and val > ls[k]:
        k += 1
    return k

def sorted_rands(n):
    res = []
    for i in range(n):
        rval = random.random()
        res.insert(posInsert(rval, res), rval)
    return res
```

Solution alternative :

```
def sorted_rands(n):
    res = []
    for i in range(n):
        rval = random.random()
        res.append(rval)
        j = i
        while j > 0 and res[j - 1] > rval:
            res[j] = res[j - 1]
            j -= 1
        res[j] = rval
    return res
```

Question 6 - Récursivité (4 points)

On vous demande d'écrire une fonction `count_divisibles(llist, div)` qui renvoie le nombre d'éléments de la liste `llist` qui sont divisibles par la valeur entière `div`.

Le paramètre `llist` est une liste dont chaque élément peut être une valeur entière ou une sous-liste. Vous pouvez considérer que le format des paramètres `llist` et `div` transmis sont valides ; vous ne devez pas le vérifier.

Exemple :

```
>>> mylist = [1, [ [4, 3], 4 ], 2, [9, 3] ]
>>> count_divisibles(mylist, 2)
3
```

Solution

```
def count_divisibles(tree, div):
    res = 0
    for x in tree:
        if type(x) == list:
            res += count_divisibles(x, div)
        elif x % div == 0:
            res += 1
    return res
```