+1/1/60+

Programmation impérative en Python — SPUF21

Année 2022-2023 — Examen terminal

Nom : .....

Prénom : .....

Numéro d'étudiant : .....

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9

Durée : 2 heures.

Aucun document n'est autorisé. L'usage de la calculatrice ou de tout autre appareil électronique est interdit.

Les exercices sont indépendants. Au sein d'un même exercice, vous pouvez utiliser les variables et fonctions des questions précédentes, même si vous n'avez pas su les faire ; chaque question est donc indépendante.

À part les méthodes et fonctions de base, vous n'avez pas le droit d'utiliser les fonctions et les méthodes « avancées », sauf si l'énoncé vous conseille l'utilisation de certaines d'entre elles.

```python
# Fonctions autorisées
range(…) len(…)
print(…) int(…)

# Méthodes et mots-clés autorisés
L.append(x)
x in L
```

```python
# Par exemple les méthodes et fonctions suivantes sont entre autres interdites
max(…) min(…) sum(…) abs(…) eval(…)
s.split(…) s.index(…) L.extend(…)

# Vous n'avez pas le droit d'utiliser des compréhensions ou des slices
# À la place vous devez utiliser des boucles.
[ x for x in range(L) ]
chaine[début:fin:pas]
```

L'objectif du sujet est d'évaluer des expressions mathématiques exprimées sous forme de chaîne. On ne s'autorisera dans la formule que des valeurs entières (le résultat lui pourra être flottant) et les 5 opérations +, -,*, / et ^ (puissance). Par exemple : `"1+(2*3)"` ou même `"5*(-1+ (3*4))^7-8/5"`.

Exercice 1 Exercices simples d'extraction de chaîne . . . . . 3 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3

1. Écrire une fonction `chaîne(ch,début,fin)` qui renvoie une sous-chaîne de `ch` dont le premier caractère a pour indice `début` et le dernier caractère a pour indice `fin`. On suppose que $0 \leq$`début` $\leq$ `fin` $<$ `len(ch)`. On n'aura pas le droit d'utiliser les *slices* (c'est-à-dire une notation de la forme `ch[0:10:2]`)

```
>>> chaîne("0123456789", 4, 8)
'45678'
```

```
def chaîne(ch,début,fin):
    résultat = ""
    for i in range(début, fin+1):
       résultat = résultat + ch[i]
    return résultat
```

2. En utilisant la fonction `chaîne`, écrire les trois fonctions ci-dessous. Chaque fonction ne devra être constituée que d'une ou deux lignes.

```
>>> intérieur("(1+2)")
'1+2'
>>> suivant("-(3+2)")
'(3+2)'
>>> découpage("(1+2)*(3+5)",5)
('(1+2)', '*', '(3+5)')
```

(a) `intérieur(ch)` renvoyant la chaîne `ch` sans le premier ni le dernier caractère ;

(b) `suivant(ch)` qui renvoie la sous-chaîne de `ch` sauf le premier élément ;

(c) `découpage(ch,i)` renvoyant un triplet de sous-chaînes : l'une contenant le début de `ch` jusqu'à l'indice `i` (exclu), le caractère d'indice `i` et la fin de la chaîne (à partir de `i` exclu).

```
def intérieur(ch):
    return chaîne(ch,1,len(ch)-2)

def suivant(ch):
    return chaîne(ch,1,len(ch)-1)

def découpage(ch,i)
   return ( chaîne(ch,0,i-1), ch[i], chaîne(ch,i+1,len(ch)-1))
```

Exercice 2 Pré-traitement . . . . . 3 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3

1. Écrire une fonction `suppression_espaces(ch)` qui renvoie la chaîne obtenue en supprimant tous les espaces de `ch`.

```
>>> suppression_espaces("( 1+ 2) * 3 ")
'(1+2)*3'
```

```
def suppression_espaces(ch):
    r=''
    for char in ch:
        if char != " ":
            r = r+char
    return r
```

2. En mathématique, le symbole `'-'` peut avoir deux sens différents : la soustraction (comme dans `"3-2"`) ou la négation (comme dans `"3 + -2"`). L'objectif de cette question est de remplacer toutes les négations par le symbole `"m"` (m comme « moins »).
Écrire une fonction `suppression_négatif(ch)` qui renvoie une nouvelle chaîne construite à partir de `ch` en appliquant la règle suivante : *« Si un caractère `"-"` suit un des caractères de la chaîne `"0123456789)"`, alors on le laisse intact ; sinon, on le remplace par le symbole `"m"` »*. On supposera que la chaîne `ch` ne contient aucun espace.

```
>>> suppression_négatif("1+-3")
'1+m3'
>>> suppression_négatif("1-3")
'1-3'
```

```
>>> suppression_négatif("-1-3")
'm1-3'
>>> suppression_négatif("-(1+2)--5")
'm(1+2)-m5'
```

```
def suppression_négatif(ch):
    r = ""
    for i in range(0,len(ch)):
        if ch[i] == "-" and i==0:
            r = r + "m"
        elif ch[i] == "-" and not (ch[i-1] in "0123456789)"):
            r = r + "m"     # ch[i-1] not in "0123456789)":  # correct
        else:
            r = r + ch[i]
    return r
```

Exercice 3 Création du dictionnaire . . . . . 5,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4 ☐4,5 ☐5 ☐5,5

Le problème de l'écriture usuelle est la gestion des priorités. L'expression `"1+2*3"` s'interprète comme `"1+(2*3)"` quand `"1*2+3"` doit s'interpréter comme `"(1*2)+3"`. On souhaite ajouter des parenthèses supplémentaires afin de ne plus avoir à tenir compte des priorités implicites. L'objectif est de remplacer tous les `'+'` par `')))+((('`, tous les `'*'` par `'))*(('`, tous les `'^'` par `')^('` : plus une opération est prioritaire, moins on ajoute de parenthèses. L'expression obtenue sera alors correctement parenthésée et permettra de ne plus avoir à gérer les règles de priorité. Dans cet exercice, nous allons créer le dictionnaire indiquant les règles de substitution.

1. Écrire une fonction `répétition(n,c)` qui construit avec une boucle la chaîne de caractères obtenue en répétant `n` fois la chaîne `c`. Question supplémentaire : comment pourrait-on le faire en un seul calcul sans créer de nouvelle fonction ?

```
>>> répétition(5,'!')
'!!!!!'
>>> répétition(3,'(')
'((('
```

```
def répétition(n,c):
    chaîne = ""
    for i in range(n):
        chaîne = chaîne + c
    return chaîne
# ou simplement « n*c »
```

2. On se donne une liste `L` de couples de la forme `(c,n)` où `c` est un caractère représentant une opération et `n` un entier correspondant à la priorité (plus `n` est grand, moins `c` est prioritaire). Écrire une fonction `max_priorité(L)` qui renvoie la plus grande priorité de la liste `L`. Si `L` est vide, on renverra l'entier `0`.

```
>>> max_priorité([ ('+',3), ('*',2), ('^',1) ])
3
>>> max_priorité([('*',1), ('+',2), ('-',2) ])
2
```

```
def max_priorité(L):
    maximum = 0
    for i in range(len(L)):
        (c,n) = L[i]
        if n>maximum:
            maximum = n
    return maximum
```

3. On souhaite transformer la liste `L` de couples en un dictionnaire. À chaque tuple `(c,n)` de `L` le dictionnaire fera correspondre une paire clé/valeur où la clé sera le symbole `c` associé à la valeur `')…)c(…('` (concrètement : `n` parenthèses fermantes suivies du symbole `c` suivi de `n` parenthèses ouvrantes).
Par exemple `('+',3)` sera associé au couple clé/valeur : clé : `'+'` ⟶ valeur : `')))+((('`.
De même `('/',2)` sera associé au couple clé/valeur : clé : `'/'` ⟶ valeur : `'))/(('`.
Écrire une fonction `dictionnaire(L)` qui transforme une liste `L` de couples en un dictionnaire dont les clés et les valeurs correspondent à l'explication ci-dessus.

```
>>> L = [ ('+',3), ('*',2), ('^',1) ]
>>> dictionnaire(L)
{'+': ')))+(((', '*': '))*((', '^': ')^('}
```

```
def dictionnaire(L):
    dico = dict() # ou {}
    for i in range(len(L)):
      (c,n) = L[i]
      dico[c] = répétition(n,")") + c + répétition(n,"(")
    return dico
```

4. Notre dictionnaire n'est pas complet. On doit y ajouter la clé `'('` associée à la valeur `'((…('` où le symbole `'('` est répété une fois de plus que la priorité maximale calculée à la question 1 de l'exercice. De même avec la clé `')'` et la valeur `'))…)'`. Écrire une fonction `dictionnaire_complet(L)` qui fait appel à la fonction précédente et complète le dictionnaire avec les deux clés `"("` et `")"` avant de le renvoyer.

```
>>> dictionnaire_complet([ ('+',3), ('*',2), ('^',1) ]) # max_priorité -> 3
{'+': ')))+(((', '*': '))*((', '^': ')^(', '(': '((((', ')': '))))'}
>>> dictionnaire_complet([ ('*',1), ('+',2)]) # max_priorité -> 2
{'*': ')*(', '+': '))+((', '(': '(((', ')': ')))'}
```

```
def dictionnaire_complet(L):
    dico = dictionnaire(L)
    m = max_priorité(L)
    dico['('] = (m+1) * '('
    dico[')'] = (m+1) * ')'
    return dico
```

Exercice 4 Parenthésage . . . . . 2 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2

Parmi les opérations, le `"+"` et le `"-"` sont les moins prioritaires, suivis du `"*"` et du `"/"`, suivis de la puissance `"^"`. On fixe cet ordre de priorité en utilisant une liste `L = [ ('+',3), ('-',3), ('*',2), ('/',2), ('^',1) ]` définie de manière globale ; vous pouvez utiliser la liste `L` sans la redéfinir.

Écrire une fonction `parenthèsage(formule)` dans laquelle **(a)** on crée le dictionnaire complet correspondant à la liste `L` en utilisant les fonctions de l'exercice précédent et **(b)** on transforme la chaîne `formule` en appliquant ces trois étapes :

1. on ajoute un symbole `'('` au début de la formule et `')'` à la fin de la formule ;
2. on supprime les espaces et on remplace les négations par des symboles `'m'` (*cf.* Exercice 2) ;
3. on remplace chaque symbole qui correspond à une clé du dictionnaire par la valeur correspondante.

Prenons comme exemple la chaîne `"-1+2 * 3"`

- après l'étape 1 : `"(-1+2 * 3)"`
- après l'étape 2 : `"(m1+2*3)"`
- après l'étape 3 : `"((((m1)))+(((2))*((3))))"`
  - on a remplacé `"("` par `"((((" et `")"` par `"))))"`
  - on a remplacé `"+"` par `")))+((("` et `"*"` par `"))*(("`

```
>>> parenthèsage("-1+2 *3")
'((((m1)))+(((2))*((3))))'
```

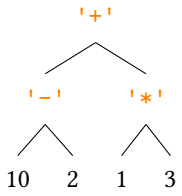
```
def parenthèsage(formule):
    dico = dictionnaire_complet(L)
    formule = '(' + formule + ')'
    formule = suppression_espace(formule)
    formule = suppression_négatif(formule)
    résultat = ""
    for c in formule:
        if c in dico:
            résultat = résultat + dico[c]
        else:
            résultat = résultat + c
    return résultat
```

La formule finale est affreuse pour l'œil humain mais a l'avantage de ne nous permettre d'ignorer les règles implicites de priorité entre opérations. Grâce aux nouvelles parenthèses, l'ordre des calculs est parfaitement défini (même si de nombreuses parenthèses sont clairement inutiles).

Exercice 5 Trouvons la racine . . . . . 2,5 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5

Lorsque plusieurs opérations de même priorité s'enchaînent, l'opération principale (celle correspondant à la racine de l'arbre) est la dernière. En effet la formule $10-2+3$ doit s'interpréter $(10-2)+3$ et non $10-(2+3)$. Malheureusement, on ne peut pas se contenter de renvoyer la dernière opération de la chaîne : il faut tenir compte des parenthèses. Par exemple, dans la formule $10-2+(1*3)$, l'opération principale est l'addition et l'arbre correspondant sera celui ci-dessous.



1. Pour trouver la racine, on va parcourir la chaîne de droite à gauche (à l'envers donc). On se donne une variable `niveau`, initialisée à `0`, correspondant au niveau de profondeur dans les parenthèses. Si le caractère lu est `")"`, on incrémente `niveau` et si le caractère lu est `"("` on décrémente `niveau`.
Si en parcourant la chaîne on tombe sur une opération (un des symboles de la chaîne `"+-*/^"`) pendant que la variable `niveau` vaut `0`, on renverra l'indice correspondant. Cela signifie que l'opération est la racine de l'arbre. Si une telle opération n'est pas trouvée on renverra `None`.

Écrire une fonction `indice_racine(formule)` qui renvoie l'indice de l'opération correspondant à la racine de l'arbre en utilisant cet algorithme.

```
>>> indice_racine("1+2-(3*5)") # l'indice de - vaut 3
3
```

```
def indice_racine(formule):
  niveau=0
  for i in range(len(formule)-1, -1, -1):
    if formule[i]==')':
      niveau = niveau+1
    elif formule[i]=='(':
      niveau = niveau -1
    elif niveau==0 and formule[i] in "+-*/^":
      return i
  return None # Inutile, par défaut None est renvoyé
```

2. En déduire une fonction `découpage_formule(chaîne)` qui renvoie un triplet de trois chaînes correspondant à la partie gauche du calcul, à l'opération principale et à la partie droite du calcul. On pourra utiliser l'indice renvoyé par la fonction précédente ainsi que la fonction `découpage` du premier exercice. Si l'indice de la racine est `None`, on renverra alors `None`

```
>>> découpage_formule("1-5+2-(3*5)")
('1-5+2', '-', '(3*5)')
```

```
def découpage_formule(formule):
    i = indice_racine(formule)
    if i!=None:
        return découpage(formule,i)
    else:
        return None
```

Exercice 6 Arborisons le calcul . . . . . 4 points

☐0 ☐0,5 ☐1 ☐1,5 ☐2 ☐2,5 ☐3 ☐3,5 ☐4

Dans l'exercice suivant nous allons travailler avec des arbres. Pour manipuler ces derniers, vous ne pouvez utiliser que les fonctions suivantes :
— `arbre(r,Ag,Ad)` renvoie un arbre de racine `r` et de fils `Ag` (gauche) et `Ad` (droit) ;
— `est_feuille(A)` renvoie `True` si `A` est une feuille, et `False` sinon ;
— `racine(A)` renvoie la racine de l'arbre `A` : `'+'`, `'-'`, `'*'`, `'/'`, ou `'^'` ;
— `fg(A)` renvoie le fils gauche de `A` ;
— `fd(A)` renvoie le fils droit de `A`.

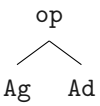
Nous avons grâce à l'exercice 4 une formule correctement parenthésée. Il reste maintenant à la transformer en un arbre. L'objectif est d'écrire une fonction récursive qui à partir d'une chaîne `formule` renvoie un tel arbre.

L'algorithme est récursif et crée un arbre. Mais attention, ce n'est pas une récurrence sur un arbre mais sur une chaîne. Prenons par exemple la chaîne `"1-5+2-(3*5)"`. On la découpe en trois parties : `"1-5+2"` correspondant au calcul du sous-arbre gauche, `'-'` à la racine et `"(3*5)"` au calcul du sous-arbre droit. L'arbre obtenu sera :

où `Ag` et `Ad` sont les sous-arbres, calculés à partir de `"1-5+2"` (pour `Ag`) et `"(3*5)"` (pour `Ad`).

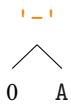
On se donne comme argument une chaîne de caractères `formule`. L'algorithme commence par y appliquer la fonction `découpage_formule` de l'exercice précédent. Selon le résultat, deux cas se présentent :

Premier cas, le découpage renvoie un triplet `(gauche, op, droite)`. On transforme récursivement `gauche` et `droite` en deux sous arbres `Ag` et `Ad` et on renvoie l'arbre :



Second cas, le découpage renvoie `None` Dans ce cas on regarde le premier caractère de la formule.

- Si le premier caractère de `formule` est un chiffre, cela signifie que la chaîne représente un entier et dans ce cas, il suffit de la convertir en feuille avec la fonction de conversion `int`.
- Si le premier caractère de `formule` est une parenthèse ouvrante `"("`, cela signifie que la formule est entourée de parenthèses inutiles, dans ce cas, on recommence récursivement avec la chaîne `intérieur(formule)` (*cf.* Exercice 1).
- Si le premier caractère est un `"m"`, dans ce cas, en notant « `A` » l'arbre obtenu en arborisant récursivement la chaîne `suivant(formule)` (*cf.* Exercice 1), on renvoie l'arbre de la forme :



- Sinon, on lance une `ValueError` avec un petit message explicatif.

1. Écrire une fonction `arboriser(formule)` qui implémente l'algorithme récursif précédent en Python.

```
def arboriser(formule):
    d = découpage_formule(formule)
    if d != None:
        (g,op,d) = d
        return arbre(op,arboriser(g),arboriser(d))
    else:
        if formule[0] == "(":
            return arboriser(intérieur(formule))
        elif formule[0] == "m":
            return arbre('-', 0, arboriser(suite(formule)))
        elif formule[0] in "0123456789":
            return int(formule)
        else:
            raise ValueError("Mais c'est quoi cette formule ?")
```

2. Enfin, écrire une fonction `calculer(formule)` qui transforme la formule en arbre et évalue le calcul correspondant. On pourra créer des fonctions supplémentaires si nécessaire.

```
def calcul_arbre(A):
    if est_feuille(A):
        return A
    else:
        op = racine(A)
        g = calcul_arbre(fg(A))
        d = calcul_arbre(fd(A))
        if op=="+":
            return g + d
        elif op=="*":
            return g * d
        elif op=="-":
            return g - d
        elif op=="/":
            return g / d
        elif op=="^":
            return g ** d



def calculer(formule):
   return calcul_arbre(arboriser(formule))
```

3. C'est génial, maintenant vous pouvez faire plein de calculs avec Python ! En vrai, ça existe déjà et c'est la fonction `eval`, mais c'est quand même super chouette de savoir le faire soi-même !

```
>>> eval("1+1")
2
```