TP8

Partie 1 — Récursivité

Ex. 1 — Factorielle

Consigne pour les tests automatiques

Dans la partie Récursivité du TP, un certain nombre de tests automatiques vous aideront à réaliser vos fonctions en vous donnant des indices si elles ne paraissent pas justes. Si vous voulez que ces tests fonctionnent le mieux possible et puissent vous indiquer la voie à suivre, vous devez nommer vos fonctions et leurs paramètres EXACTEMENT de la manière demandée dans l'énoncé.

Récursivité

En informatique, on dit qu'un algorithme est **récursif** quand il fait référence à lui-même.

Par exemple, la fonction suivante est récursive :

```
def print_n_hello(n):
    if n > 0:
        print("Hello")
    print_n_hello(n-1)
```

En effet, on trouve à l'intérieur de la définition de la fonction `print_n_hello(n)` un appel à cette même fonction. On appelle cela un **appel récursif**, c'est-à-dire l'utilisation d'une fonction dans sa définition.

Notre fonction permet donc d'afficher `n` fois le mot `"Hello"` en remarquant qu'afficher `n` fois ce mot revient à l'afficher une fois puis à recommencer pour `n-1`.

Quand on écrit une fonction récursive, il faut toujours penser à mettre un **cas de base**. Cela signifie qu'il faut savoir quand la fonction va arrêter de faire des appels récursifs, sinon, elle ne s'arrêtera jamais !

Par exemple, notre algorithme s'arrête car on rappelle la fonction à chaque fois avec un entier plus petit que lors de l'appel précédent (`n-1` au lieu de `n`) et quand `n` devient nul, on ne fait rien. On est donc certains qu'au bout d'un nombre d'appels récursifs fini, la fonction ne fera plus rien et s'arrêtera.

Questions

Certaines fonctions se définissent naturellement de manière récursive. Considérons la fonction factorielle : `n! = n x (n-1) x (n-2) x ... x 1`. On peut

remarquer que `n! = n x (n-1)!` pour `n>1` et `1! = 1` ce qui constitue notre appel récursif et notre cas de base.

Question 1 : Définissez une fonction **récursive** `factorial(n)` qui renvoie le résultat du calcul `n!`.

Question 2 : Affichez le résultat de votre fonction pour `n=1`.

Question 3 : Affichez le résultat de votre fonction pour `n=10`.

Question 4 : Écrivez une nouvelle fonction `factorial_print_step(n)` qui calcule le résultat du calcul `n!` mais en plus affiche la valeur du paramètre `n` sur laquelle elle est appelée.

Question 5 : Affichez le résultat de votre fonction pour `n=10` et observez tous les appels récursifs à votre fonction.

Correction

```
# Q1
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)


# Q2
print(factorial(1))

# Q3
print(factorial(10))


# Q4
def factorial_print_step(n):
    print(n)
    if n == 1:
        return 1
    else:
        return n * factorial_print_step(n - 1)


# Q5
print(factorial_print_step(10))
```

Ex. 2 — Exponentiation

Quand une fonction ne fait pas un appel à elle-même, on dit qu'elle est **itérative**. Toutes les fonctions que vous avez écrites jusqu'à maintenant étaient donc itératives.

Exponentiation "normale"

Question 1 : Écrivez une fonction **itérative** `exponentiation(x, n)` pour calculer la valeur de x^n.

Question 2 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Question 3 : Écrivez une fonction **récursive** `recursive_exponentiation(x, n)` pour calculer la valeur de x^n. On pourra remarquer que $x^n = x * x^{n-1}$.

Question 4 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Exponentiation rapide

Lors d'un TP précédent, vous avez écrit une fonction d'exponentiation rapide. Pour cela, vous avez utilisé la remarque suivante pour calculer x^n :

- si x est pair : $x^n = (x * x)^{n/2}$;
- si x est impair : $x^n = x * (x * x)^{\frac{n-1}{2}}$.

Vous ne connaissiez pas la notion de fonction récursive à ce moment-là et vous avez donc écrit une fonction **itérative**.

Question 5 : Écrivez une fonction **récursive** `fast_exponentiation(x,n)` qui calcule x^n en utilisant la méthode de l'exponentiation rapide. Attention, n'oubliez pas les cas de base.

Question 6 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Correction

```
# Q1
def exponentiation(x, n):
    result = 1
    for i in range(n):
        result = result * x
    return result
```

```
# Q2
print(exponentiation(2, 4))


# Q3
def recursive_exponentiation(x, n):
    if n == 0:
        return 1
    else:
        return x * recursive_exponentiation(x, n - 1)


# Q4
print(recursive_exponentiation(2, 4))


# Q5
def fast_exponentiation(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        if n % 2 == 0:
            return fast_exponentiation(x * x, n // 2)
        else:
            return x * fast_exponentiation(x * x, (n - 1) // 2)


# Q6
print(fast_exponentiation(2, 4))
```

Ex. 3 — Tri fusion

Le tri fusion est une méthode récursive de tri d'une liste. L'algorithme appelé pour trier une liste `l` se résume ainsi :

- On sépare la liste `l` en deux listes `l1` et `l2` de tailles égales.
- On trie `l1` et `l2` par la méthode du tri fusion.
- On fusionne les deux listes ainsi triées.

On vous donne la fonction `merge(l1, l2)` qui fusionne deux listes triées (troisième étape de l'algorithme).

Rappel : à partir d'une liste `l`, on peut créer une nouvelle liste `l1` contenant les `i` premiers éléments de `l` en utilisant `l1 = l[:i]`. On peut de même récupérer tous les éléments situés après l'indice `i` avec `l2 = l[i:]`.

Question 1 : Écrivez la fonction **récursive** `merge_sort(l)` qui utilise la fonction `merge` pour réaliser le tri fusion. N'oubliez pas **les** cas de base.

Question 2 : Affichez le résultat de votre fonction pour `[4, 7, 1, 0, 3, 6]`.

Correction

```
def merge(l1, l2):
    i1 = 0
    i2 = 0
    merged = []
    while i1 < len(l1) and i2 < len(l2):
        if l1[i1] < l2[i2]:
            merged.append(l1[i1])
            i1 += 1
        else:
            merged.append(l2[i2])
            i2 += 1

    while i1 < len(l1):
        merged.append(l1[i1])
        i1 += 1

    while i2 < len(l2):
        merged.append(l2[i2])
        i2 += 1

    return merged


# Q1
def merge_sort(l):
    if len(l) == 0 or len(l) == 1:
        return l
    else:
        n = len(l)
        l1 = l[:n // 2]
        l2 = l[n // 2:]
        l1_sorted = merge_sort(l1)
        l2_sorted = merge_sort(l2)
```

```
        return merge(l1_sorted, l2_sorted)


# Q2
print(merge_sort([4, 7, 1, 0, 3, 6]))
```

Ex. 4 — Fibonacci (Optionnel) : les possibles soucis du récursif

Vous avez vu lors de TPs précédents la définition de la suite de Fibonacci : $F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$. Cette définition est naturellement récursive.

Question 1 : Écrivez une fonction **récursive** `recursive_fibo(n)` qui calcule le n^{ime} terme de la suite de Fibonacci et affiche lors de son appel la valeur de n sur laquelle elle a été appelée.

Question 2 : Affichez le résultat de votre fonction pour `n=7`. Que remarquez-vous ?

Cet exemple permet d'illustrer que même si les fonctions récursives sont souvent plus simples que leur version itérative, elles ne sont pas toujours plus efficaces. Ici, en utilisant une fonction récursive qui implémente directement la définition de la suite, on va faire plusieurs fois exactement le même calcul et donc ne pas être efficace du tout.

Question 3 : Affichez le résultat de votre fonction pour `n=15`. Observez le nombre d'appels qui sont faits alors qu'on pourrait retourner le résultat en faisant seulement une quinzaine d'additions.

Correction

```
# Q1
def recursive_fibo(n):
    print(n)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return recursive_fibo(n - 1) + recursive_fibo(n - 2)


# Q2
```

```
print(recursive_fibo(7))
# Q3
print(recursive_fibo(15))
```

Ex. 5 — Tours de Hanoi (Optionnel)

Le problème des tours de Hanoi est le suivant :

- On dipose de trois piquets et de n palets de tailles $1, 2, 3, ..., n$.
- Au départ, tous les palets sont empilés de manière décroissante sur le piquet de gauche : le palet de taille n est tout en bas et celui de taille 1 tout en haut.
- On ne peut déplacer qu'un palet à la fois.
- On ne peut déplacer que le palet le plus haut d'une pile.
- On ne peut mettre un palet sur un autre que s'il est de taille plus petite.
- On veut faire passer tous les palets sur le piquet de droite.

Une résolution récursive du problème pour plus de 1 palet est la suivante : - on bouge $n-1$ palets du piquet de départ au piquet intermédiaire; - on bouge le grand palet du piquet de départ au piquet d'arrivée; - on bouge $n-1$ palets du piquet intermédiaire au piquet d'arrivée.

Question 1 : Écrivez la fonction `hanoi(n, begin, end, temporary)` permettant de bouger n palets depuis le piquet `begin` vers le piquer `end` en utilisant le piquet `temporary` comme piquet intermédiaire et qui renvoie les piquets begin, end, temporary modifiés. On représentera chaque piquet par une liste représentant les tailles des palets présents sur le piquet, de bas en haut.

Question 2 : Affichez le résultat de votre fonction appliquée au déplacement de 6 palets.

Correction

```
# Q1
def hanoi(n, begin, end, temporary):
    if n > 0:
        hanoi(n - 1, begin, temporary, end)
        element_to_move = begin.pop()
        end.append(element_to_move)
        hanoi(n - 1, temporary, end, begin)
    return begin, end, temporary
```

```
# Q2
print(hanoi(6, [6, 5, 4, 3, 2, 1], [], []))
```

Ex.6 — Fractales

Correction automatique

Attention, cet exercice vous demandant de faire des dessins, il n'y aura pas de tests automatiques. Lancez votre code et regardez si le dessin vous convient !

Turtle

Le module turtle de python permet la production facile de figures géométriques, à l'aide d'un ensemble d'instructions de dessins élémentaires.

On importe tout ce que contient un module par la ligne `from turtle import *`. Cela permet de dire à python de chercher le module `turtle` et de récupérer toutes (c'est le sens de *) les fonctions qui y sont définies.

Grâce à cette ligne, on a accès dans le fichier à toutes les fonctions de turtle :

- `reset()` : efface le dessin
- `up()` : relève le crayon pour ne plus dessiner
- `down()` : abaisse le crayon pour dessiner
- `forward(distance)` : avance d'une distance donnée
- `backward(distance)`: recule d'une distance donnée
- `left(angle)` : tourne à gauche d'un angle exprimé en degrés
- `right(angle)` : tourne à droite d'un angle exprimé en degrés.

Question 1 : Écrivez une fonction `draw_square(length)` qui dessine un carré de côté donné.

Question 2 : Appelez votre fonction pour dessiner un carré de côté 100.

Si vous souhaitez que votre dessin ne se ferme pas dès qu'il est terminé, vous pouvez utiliser le module `time` de python et sa fonction `sleep(time)` qui permet de faire une pause de `time` secondes, juste après le dessin. Il faut alors ajouter l'import de la fonction `sleep` depuis le module `time` : `from time import sleep` tout en haut du fichier, qui dit à python d'aller chercher la fonction qui nous intéresse dans le bon module.

Flocon de Von Koch

Le flocon de Von Koch est une fractale c'est-à-dire une construction géométrique récursive, invariante par changement d'échelle. Pour construire le flocon de Von Koch, on part d'un triangle équilatéral, puis, à chaque itération, on divise chaque côté du polygone en trois segments égaux. À partir de ces trois segments, on

trace un triangle équilatéral orienté vers l'extérieur, de base le segment du milieu puis on supprime le segment du milieu.

On commencera par écrire une fonction qui permet d'appliquer un certain nombre de fois la transformation à un segment :

- l'appliquer 0 fois trace le segment;
- l'appliquer n fois revient à l'appliquer $n-1$ fois sur un segment de taille divisée par 3, tourner à gauche de 60

degrés, l'appliquer à nouveau $n-1$ fois sur un segment de taille divisée par 3, tourner à droite de 120 degrés, l'appliquer encore $n-1$ fois sur un segment de taille divisée par 3, tourner à gauche de 60 degrés et enfin l'appliquer $n-1$ fois sur un segment de taille divisée par 3.

Question 3 : Écrire une fonction `snowflake_side(length, steps)` qui permet d'appliquer `steps` fois la transformation à un segment de taille `length`.

Question 4 : Appelez votre fonction pour `2` étapes sur un segment de taille `100`.

Question 5 : Écrire la fonction `snowflake(length, step)` qui fabrique le flocon de Von Koch en appliquant `steps` fois la transformation pour un triangle équilatéral de départ de côté `length`.

Question 6 : Appelez votre fonction `snowflake` pour `1` étapes sur un segment de taille `250`.

Question 7 : Appelez votre fonction `snowflake` pour `2` étapes sur un segment de taille `250`.

Question 8 : Appelez votre fonction `snowflake` pour `5` étapes sur un segment de taille `250`. Vous pouvez augmenter la vitesse de dessin en utilisant `speed(speed)` avec en paramètre la vitesse que vous souhaitez, par exemple 1000.

Correction

```
from turtle import *
from time import sleep


# Q1
def draw_square(length):
    for i in range(4):
        forward(length)
        right(90)
```

```
# Q2
draw_square(100)


# Q3
def snowflake_side(distance, step):
    if step == 0:
        forward(distance)
    else:
        snowflake_side(distance / 3, step - 1)
        left(60)
        snowflake_side(distance / 3, step - 1)
        right(120)
        snowflake_side(distance / 3, step - 1)
        left(60)
        snowflake_side(distance / 3, step - 1)


# Q4
reset()
snowflake_side(100, 2)


# Q5
def snowflake(distance, etape):
    for i in range(3):
        snowflake_side(distance, etape)
        right(120)


# Q6
reset()
speed(10)
snowflake(250, 1)
sleep(5)

# Q7
reset()
speed(10)
snowflake(250, 2)
sleep(5)

# Q8
reset()
speed(1000)
```

```
snowflake(250, 5)
sleep(5)
```

Partie 2 — Arbres

Ex. 7 — Expressions

Cet exercice se fera dans la console.

Types

Jusqu'à present nous avons utilisé des types simples, comme par exemple les entiers et les chaines de caractères.

Question 1 : Evaluer dans la console :

```
>>> type(2)
>>> type("2")
>>> type(int("2"))
>>> type(str(2))
```

Dans le premier cas la console renvoie `<class 'int'>` *pour signifier que* `2` *est du type simple* `int`*. Savez-vous pourquoi le second cas renvoie* `str`*? Comment expliquer les deux derniers cas?*

Nous avons aussi utilisé des types composés, comme par exemple les listes:

Question 2 : Evaluer dans la console :

```
>>> type([1,2,3])
```

Nous n'avions pas de type pour représenter par exemple le symbole d'une inconnue x, dans une expression mathématique comme $2 + x$.

Question 3 : Evaluer dans la console :

```
>>> 2+x
>>> 2+"x"
```

Notez que dans un premier temps l'interpréteur se plaint de ne pas connaître la variable x, ce qui cause une erreur. En effet: une variable au sens informatique du terme, ce n'est pas la même chose qu'une inconnue au sens mathématique du terme. Une variable, en informatique, c'est un emplacement mémoire, qui contient une valeur concrète. Une inconnue, en mathémétique, est un symbole qui n'a pas de valeur concrète associée.

Dans une second temps nous avons tenté de représenter l'inconnue x par une simple chaîne de caractère, ce est une bonne idée. Le problème, c'est que python

ne comprends pas que l'on veuille sommer des chaînes de caractères et des entiers. Cela cause une erreur de type.

Nous allons utiliser le module `sympy` pour avoir un type symbole inconnu.

Question 4 : Evaluer dans la console :

```
>>> from sympy import *
>>> S("x")
>>> type(S("x"))
>>> 2+S("x")
```

Le module `sympy` nous a donné une fonction `S(.)` qui crée des symboles. Ces symboles sont d'un type particulier, définit par `sympy`. On peut y songer comme à des chaînes de caractères, mais qui cette fois admettent d'être additionnées à des entiers. Ces symboles vont nous permettre de representer les inconnues en mathématiques.

Calcul symbolique

Nous savions déjà que nous pouvions utiliser la console python comme un calculatrice. Maintenant, grâce à `sympy`, nous pouvons nous en servir pour faire de l'algèbre.

Tout d'abord ameillorons l'affichage de nos expressions.

Question 5 : Evaluer dans la console :

```
>>> x=S("x")
>>> (sqrt(2)+x**2)/x
>>> init_printing()
>>> (sqrt(2)+x**2)/x
```

Notez que la première ligne met le symbole x dans une variable que l'on a appelé `x`, mais que l'on aurait pu appeler autrement. Cela nous permet juste d'écrire `x` au lieu de `S("x")`. Notez comme la procédure `init_printing()` met en place un affichage plus mathématique de nos expressions.

Puis, découvrons quelques fonctionnalités de `sympy`.

Question 6 : Evaluer dans la console :

```
>>> diff(sin(x)+x**2)
>>> integrate(sin(x)+x**2)
>>> solve(Eq(x**2+2,0), x)
>>> simplify(cos(x)**2+sin(x)**2)
>>> expand((x+1)**10)
>>> factor(x**2+100+20*x)
```

Ces fonctionnalités sont présentes aussi dans tout Computer Algebra System (CAS) qui se réspecte, comme Mathematica, Maple, Sage, Matlab… Elles cou-

vrent probablement la totalité du programme de Licence 1 de Mathématiques, et au-delà. Cela signifie, donc, que derrière chacune des notions qui vous sont enseignées actuellement en mathématiques, se cache un algorithme, que l'on peut expliciter et coder notamment en python.

Correction

```
# TODO: Cet exercice est à faire dans la console python.
```

Ex. 8 — Arbres

Cet exercice se fera dans la console.

Pour cet exercice on suppose que vous avez déjà évalué:

```
>>> from sympy import *
>>> x=S("x")
>>> init_printing()
```

Au cours de l'exercice précédent nous avons vu quel était le type de `x` ou de `2` dans l'expression `2+x`, en évaluant:

```
>>> type(x)
>>> type(2)
```

Mais, quel est le type d'une expression comme `2+x`?

Question 1 : Evaluer dans la console :

```
>>> type(2+x)
>>> type(2*x)
```

Notez que `2+x` est de type `Add`, alors que `2*x` est de type `Mul`.

Sympy a donc définit un type `Add` pour représenter les sommes, un type `Mul` pour représenter les produits, etc. On devine que type `Add` est un type composé, tout comme l'était le type `List`, puisqu'une expression de type somme doit contenir la liste des éléments sommés.

Question 2 : Evaluer dans la console :

```
>>> expr=2+x
>>> expr
>>> expr.func
>>> expr.args
>>> expr.args[0]
>>> expr.args[1]
```

Notez que `(2+x).args` renvoie la paire `(2,x)` des éléments sommés.

De même qu'une liste peut contenir des sous listes, une expression peut contenir des sous-expressions.

Question 3 : Evaluer dans la console :

```
>>> expr=x*(2+x)
>>> expr
>>> expr.func
>>> expr.args
>>> expr.args[1]
>>> expr.args[1].func
>>> srepr(expr)
```

Notez que `(x*(2+x)).args[1]` n'est autre que l'expression `2+x`.

Les types composés sont aussi appelés "structures de données" par les informaticiens. Ici, la structure de donnée utilisée pour par `sympy` pour représenter les expressions peut être comparée à un arbre. En effet, concernant l'expression `expr=x*(2+x)`:

- La racine de l'arbre est une multiplication. C'est que renvoie `expr.func`.
- L'arbre contient deux sous arbres. C'est ce que renvoie `expr.args`.
- Le premier sous-arbre est le symbole x. C'est ce que renvoie `expr.args[0]`. Il ne contient aucun sous-arbre, on dit c'est une feuille.
- Le deuxième sous-arbre est l'expression $2 + x$. C'est ce que renvoie `expr.args[1]`.
- La racine du sous-arbre est une somme. Le sous-arbre contient deux sous-sous-arbres. C'est ce que renvoie `expr.args[1].args`.
- Les deux sous arbres sont des feuilles: l'une est le symbole x, l'autre l'entier 2.

D'un manière générale, une structure de donnée de type arbre est un type défini récursivement, comme étant:

- soit une feuille (cas de base);
- soit un arbre ayant une racine est des sous-arbres (cas récursif).

Ces structures de données sont extrêmement utilisées en informatique, par exemple pour représenter des programmes, des hiérarchies, et bien sûr l'arborescence de vos fichiers (répertoires contenant des sous-répertoires, etc.).

Correction

```
# TODO: Cet exercice est à faire dans la console python.
```

Ex. 9 — Parcours

Cet exercice se fait dans `task.py`.

Dans la première partie de ce TP nous avons étudié les algorithmes récursifs. Dans l'exercice précédent nous avons étudié les structures de données définies récursivement : les arbres. Ici, nous allons appliquer un algorithme récursif, à une structure de donnée récursive: nous allons faire un parcours d'arbre.

Aplatissement d'abre

La fonction `flatten` a pour but de parcourir et afficher un arbre d'expression `sympy`. Par exemple `flatten(2+x)` affiche:

```
addition de :
    2
    x
```

Question 1 : Compléter `flatten` afin qu'elle puisse gérer la multiplication.

Question 2 : Appliquer `flatten` sur l'expression $x(2+x)$.

Evaluation d'une expression

La fonction `calculate` prend en données une expression, et une valeur pour x, et a pour but de calculer numériquement la valeur de l'expression en ce point. Par exemple `calculate(2+x,8)` renvoie `10` en sortie.

Question 3 : Compléter `calculate` afin qu'elle puisse gérer la multiplication.

Question 4 : Appliquer `calculate` sur l'expression $x(2+x)$, pour $x = 7$ et afficher la valeur de retour.

Correction

```
from sympy import *

init_printing()
x = S("x")
# Ignore the next four lines if you want, they are just a trick
# to recover the names of the different sympy types once and for all
typeofsymbol = x.func
typeofinteger = S("2").func
typeofaddition = (2 + x).func
typeofmultiplication = (2 * x).func
```

```
# Q1
def flatten(expr):
    iflatten("", expr)


def iflatten(indent, expr):
    # This is the base case
    if expr.func == typeofsymbol or expr.func == typeofinteger:
        print(indent, expr)
    # Those are the recursive case
    elif expr.func == typeofaddition:
        print(indent, "addition de :")
        iflatten(indent + "    ", expr.args[0])
        iflatten(indent + "    ", expr.args[1])
    elif expr.func == typeofmultiplication:
        print(indent, "multiplication de :")
        iflatten(indent + "    ", expr.args[0])
        iflatten(indent + "    ", expr.args[1])
    # We are only handling a sublanguage of sympy
    else:
        print(indent, "Expression inconnue par flatten.")


# Q2
flatten(x * (2 + x))


# Q3
def calculate(expr, val):
    # This is the base case
    if expr.func == typeofsymbol:
        return val
    if expr.func == typeofinteger:
        return int(expr)
    # Those are the recursive case
    elif expr.func == typeofaddition:
        return calculate(expr.args[0], val) + calculate(expr.args[1], val)
    elif expr.func == typeofmultiplication:
        return calculate(expr.args[0], val) * calculate(expr.args[1], val)
    # We are only handling a sublanguage of sympy
    else:
        print("Expression inconnue par calculate.")
        return float("NaN")
```

```
# Q4
print(calculate(x * (2 + x), 7))
```