Implémentation de quelques algorithmes classiques avec Python

Ensimag

Stage LIESSE, juin 2014

1 Manipulations et parcours de listes

Répertoire du squelette : `lists/`

Notions abordées :

- « sequence unpacking » ou affectation de tuples : `http://docs.python.org/2/tutorial/datastructures.html#tuples-and-sequences`
- Fonction lambda : `http://docs.python.org/2/tutorial/controlflow.html#lambda-forms`
- Compréhension de listes : `http://docs.python.org/2/whatsnew/2.0.html#list-comprehensions`
- Fonctions `map`, `reduce` et `filter` : `http://docs.python.org/2/library/functions.html`

Pour commencer cette partie, un exercice sans difficulté algorithmique : le fichier `lists.py` contient une déclaration de liste `grades_list` qui représente un listing de notes. Chaque élément de la liste est un triplet (nom, note au TP, et note à l'examen).

Exercice 1.1 (Calcul de la moyenne) *Parcourir la liste, et pour chaque ligne, écrire le nom de l'élève suivi de sa moyenne, calculée par* $\frac{tp+2*exam}{3}$*. On affichera le résultat avec* `print`*.*

☞ *On peut utiliser l'affectation de tuple directement dans la boucle for, par exemple :* « `for name, lab, exam in grades_list:` ».

Exercice 1.2 (Calcul de la moyenne avec `map`) *En utilisant la construction* `list(map(`*`fonction`*`,` *`liste`*`))` *de Python, calculer la liste des moyennes. La liste des notes est déjà extraite pour vous dans* `grades_list`*.*

Exercice 1.3 (Calcul de la somme avec `reduce`) *Calculer la somme des notes avec la fonction* `reduce(fonction, iterable)` *de Python. La fonction* `reduce` *applique* `fonction` *aux éléments de* `iterable` *deux à deux.*

☞ *Pour définir une fonction somme, on peut utiliser une fonction* `lambda`*, ou bien* `operator.__add__` *qui est prédéfini en Python.*

Exercice 1.4 (Calcul du maximum avec `reduce`) *Calculer la plus haute note, toujours avec* `reduce` *et* `list_exam`*.*

☞ *Une utilisation judicieuse de* `lambda` *et de* *`valeur`* `if` *`condition`* `else` *`valeur`* *permet de faire cela en une ligne.*

☞ *Python permet en fait de faire les deux exercices précédents avec respectivement* `sum(read_exam)` *et* `max(read_exam)`*.*

Nous allons maintenant utiliser une compréhension de liste, qui permet avec une construction Python de parcourir une liste, sélectionner certains éléments, leur appliquer une transformation, et construire une liste avec le résultat. Une compréhension de liste ressemble à : `[`*`expression-utilisant-x`* `for` *`x`* `in` *`iterable`* `if` *`condition-sur-x`*`]`. On peut ici aussi utiliser les affectations de tuples pour écrire par exemple : `[x for (x, _) in list]`, qui va extraire le premier élément de chaque doublet d'une liste.

Le squelette de code propose une extraction de la liste des étudiants (couples (nom, exam)) ayant une note inférieure à 10 en utilisant les fonctions `map(fonction, iterable)` et `filter(fonction, iterable)` et des fonctions `lambda`. `map` applique `fonction` à chaque élément de la liste, et renvoie la liste des résultats, et `filter` renvoie la liste des éléments de la liste pour laquelle la fonction a renvoyé `True`.

Exercice 1.5 (Extraction d'une sous-liste par compréhension de listes) *Écrire une autre version du même exercice en utilisant une compréhension de liste (1 ligne). Le résultat est identique sur le plan algorithmique, mais beaucoup plus lisible...*

2 Tris de listes

Notions abordées :

- Algorithmes de tri (QuickSort, tri par sélection)
- Révisions sur les constructions de bases de Python.

2.1 Tri par sélection du minimum

Répertoire du squelette : `sort/`

Exercice 2.1 *Après avoir regardé l'implémentation du tri par insertion (fichier* `insertion.py`*), implémentez un tri par sélection du minimum (fichier* `selection.py`*). Vous aurez probablement besoin de deux boucles* `for` *imbriquées.*

☞ *Pour permuter deux cases du tableau, on peut faire «* `l[i], l[index_min] = l[index_min], l[i]` *».*

2.2 Tri par segmentation (QuickSort)

Répertoire du squelette : `qsort/`

Nous allons voir dans cette partie différentes implémentations d'un tri par segmentation [1] : choix d'une valeur pivot, partition de la liste en une sous-liste des éléments plus petits que le pivot, et une contenant les éléments plus grands que le pivot, puis appel récursif de la fonction de tri sur ces sous-listes. On ne cherche pas à faire le tri en place : on construit une nouvelle liste, triée, sans modifier la liste originale.

Votre squelette contient un fichier `test_sort.py`, qui sera notre programme principal et testera les différentes versions de la fonction de tri.

Exercice 2.2 *Exécutez le fichier* `test_sort.py`*. Pour l'instant, aucun tri sauf « native » (qui appelle la fonction de la bibliothèque standard de Python) et « imperative » (que nous verrons dans la section suivante) n'est implémenté. Essayez également de modifier les valeurs des constantes* `long_lists` *et* `fatal` *en tête du fichier :* `long_lists = True` *lance plus de tests, et* `fatal = True` *s'arrête sur la première erreur rencontrée (pratique pour débugger).*

Le fichier `test_sort.py` est complet, les plus curieux peuvent regarder son contenu mais ce n'est pas nécessaire pour la suite.

2.3 Programmation impérative « classique »

Exercice 2.3 *Regardez l'implémentation de la version impérative du QuickSort, dans le fichier* `imperative.py`*.*

1. Exercices inspirés de la page `http://en.literateprograms.org/Quicksort_%28Python%29`

2.4 En utilisant des compréhensions de listes : `comprehension.py`

Ouvrir le fichier `comprehension.py`. Ce fichier contient une fonction (incomplète) `qsort` (qui sera donc `comprehension.qsort` depuis l'extérieur du module `comprehension`).

Exercice 2.4 *Implémenter un tri par segmentation :*

- *Traiter le cas de base (pour une liste vide, ou une liste à un élément, on peut renvoyer une copie de la liste passée en paramètre).*
- *Choisir le pivot (par exemple,* `list[0]`*).*
- *Extraire trois sous-listes,* `lesser`*,* `greater` *et* `equal` *contenant respectivement les éléments strictement plus petits, strictement plus grands, et égaux au pivot. Les listes* `lesser` *et* `greater` *doivent être triées avec deux appels récursifs à* `qsort`*. Utiliser des compréhensions de listes avec comme condition* `if x < pivot`*,* `if x >= pivot` *et* `if x == pivot`*.*
- *Renvoyer la concaténation de* `lesser`*, du pivot et de* `equals`*, et de* `greater`*.*

☞ *Pour extraire tous les éléments d'une liste sauf le premier, on peut écrire* `list[1:]`

☞ *L'opérateur de concaténation de liste est* `+`

⚠ *L'opérateur* `+` *concatène deux listes, ou additionne deux entiers, mais écrire* `lesser + pivot` *n'a pas de sens. En revanche, on peut concaténer* `lesser` *et* `[pivot]` *(avec les crochets autour : liste à un élément contenant* `pivot`*).*

2.5 En une seule ligne : `oneline.py`

Une implémentation propre de `comprehension.py` prend un peu moins de 10 lignes. Nous allons maintenant transformer cette fonction pour la faire tenir sur une ligne (un peu longue, on aurait pu la découper en deux : le corrigé fait 117 caractères).

Exercice 2.5 *Dans* `oneline.py`*, implémenter un Quicksort en une ligne. Par rapport à la version précédente :*

- *On remplacera «* `if c: return i else: return e` *» par une expression conditionnelle : «* `return i if c else e` *».*
- *Les variables* `lesser` *et* `greater` *n'existent plus, on construit la concaténation en une expression.*

Le résultat n'est pas très lisible et donc peu recommandable, mais il illustre l'expressivité du langage.

2.6 (BONUS) Choix du pivot avec médiane de 3 éléments : `median.py`

(Ces sections ne sont gardées que pour occuper les plus rapides. Les autres peuvent passer à la comparaison de performance)

Un problème avec l'implémentation précédente du tri est qu'il repose sur le choix d'un pivot. Si le pivot est proche de la médiane des éléments de la liste, alors `lesser` et `greater` auront une taille comparable, et il suffira d'une profondeur de $log(N)$ appels récursif pour arriver au cas de base : le tri sera en $O(N \log(N))$.

Par contre, si le pivot est le plus petit ou le plus grand élément de la liste, alors `lesser` ou `greater` sera vide, et on fera un appel récursif sur une liste de taille $N-1$. Il faudra N appels récursifs imbriqués et le tri sera en N^2. La taille de pile étant limité, si N est grand, on aura une erreur à l'exécution. Malheureusement, ce cas se produit sur des listes déjà triées et est donc assez courant en pratique.

Nous allons ici améliorer le choix du pivot en prenant la médiane de 3 éléments : le premier, le dernier, et l'élément du milieu de la liste. Dans le cas d'une liste triée, on prendra toujours l'élément du milieu et on reviendra au cas $N \log(N)$, et dans les autres cas, prendre la médiane de 3 éléments augmente les chances de prendre un pivot proche de la médiane.

Exercice 2.6 *Dans* `median.py`*, implémenter la fonction* `median3`*. Elle prend en argument une liste de 3 couples* `[(`x_1`, `n_1`), (`x_2`, `n_2`), (`x_3`, `n_3`)]`*, ou les* x_i *sont les valeurs, et les* n_i *correspondent à la position des* x_i *dans la liste d'origine.* `median3` *renvoie l'indice* n_i *de l'élément médian.*

☞ *On peut s'autoriser à utiliser* `sorted(list3)` *pour trier* `list3`*, il reste alors à pendre l'élément de droite du doublet du milieu.*

Exercice 2.7 *Implémenter* `median.qsort` *en utilisant* `median3` *pour le choix du pivot.*

2.7 (BONUS) Écriture d'une fonction de partitionnement récursive : `partition.py`

Exercice 2.8 *Complétez l'implémentation de la fonction* `partition`*. Cette fonction s'appelle récursivement, en enlevant le premier élément* `head` *de* `list`*, et en ajoutant* `head` *au bon argument (*`lesser`*,* `equal` *ou* `greater`*) selon les valeurs de* `head` *et du* `pivot`*.*

2.8 Comparaison des performances

En utilisant `test_sort.py`, on peut vérifier que `comprehension.qsort()` et `oneline.qsort()` ont des temps d'exécutions comparables. `median.qsort()` est comparable sur des listes non-triées, et sans surprise bien meilleur sur des listes triées.

On peut remarquer que `partition.qsort()` est très peu performant. En effet Python n'implémente pas l'optimisation des appels récursifs terminaux [2], donc `partition` doit faire N appels récursifs imbriqués, même dans le meilleur cas.

L'écart entre nos implémentations et `native.qsort()` est énorme (un facteur 100 environ), ce qui est logique puisque `native.qsort()` a pu être optimisé finement en langage compilé (C), une implémentation dans un langage interprété comme Python est forcément beaucoup plus lente. En pratique, les gros logiciels écrits en Python utilisent souvent des bibliothèques en C pour des questions de performances.

3 Dessins de fractales

Répertoire du squelette : `fract/`

Notions abordées :

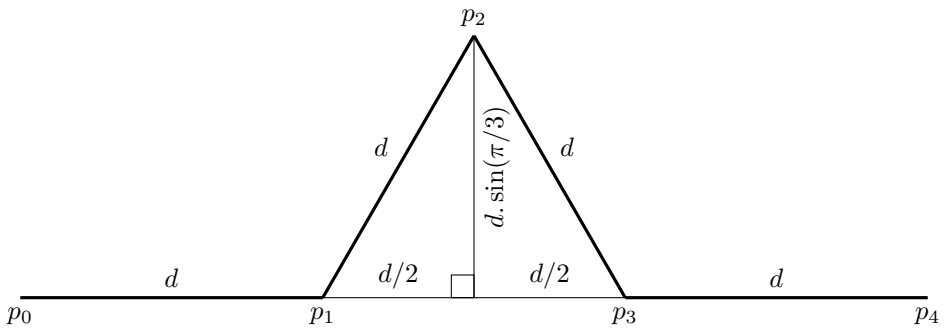
- Interfaces graphiques simples avec tkinter : `http://docs.python.org/2/library/tkinter.html`
- Dessins en utilisant différents systèmes de coordonnées (relatives à une "tortue", absolues cartésiennes) : `http://docs.python.org/2/library/turtle.html`
- Utilisation des nombres complexes en Python : `http://docs.python.org/2/library/cmath.html`

3.1 Flocon de Koch

Le flocon de Koch est une fractale facile à construire récursivement, en partant d'un triangle :

- Pour chaque segment, on le divise en trois
- On trace le premier et le dernier tiers ($[p_0p_1]$ et $[p_3p_4]$)
- On trace un triangle équilatéral sur le tiers du milieu ($[p_1p_2]$ et $[p_2p_3]$)

On ne dessinera qu'un côté du flocon (i.e. on part d'un segment au lieu de partir d'un triangle) pour simplifier l'exercice. Une itération découpe chaque segments comme ceci :



2. à la différence d'OCaml par exemple

3.1.1 En utilisant la bibliothèque `turtle` : `koch_simple_turtle.py`

La bibliothèque « turtle » est inspirée du langage de programmation « LOGO », qui a été utilisé dans les années 1980/1990 pour l'initiation à la programmation. Le principe est de dessiner des figures géométriques avec un stylo, ou « tortue », qui se déplace en laissant un trait derrière elle. Les commandes de base sont avancer, et tourner à gauche ou à droite. Les déplacement se font donc de manière relative à la position et à la direction courante.

Exercice 3.1 *Exécutez le script* `square.py`*, qui est un exemple simple de programme utilisant turtle. Regardez rapidement son code source.*

Les points importants sont :
- `from turtle import *` pour importer pouvoir utiliser les fonctions de la bibliothèque.
- Les commandes de base : `forward(`*`distance`*`)`, `left(`*`angle`*`)`, `right(`*`angle`*`)`. Le premier appel à une fonction de turtle ouvre une fenêtre graphique.
- Pour déplacer la tortue sans tracer de trait, on lève le stylo avec `penup()`, on de déplace, et on repose le stylo avec `pendown()`.
- Pour un résultat plus esthétique, on peut changer la vitesse de déplacement avec `speed` et la couleur avec `color`.

Exercice 3.2 *Dans le fichier* `koch_simple_turtle.py`*, complétez l'implémentation de la fonction* `koch(length)` *: si la longueur du segment est plus petite que 2, alors on trace le segment avec* `forward`*. Sinon, on trace les 4 segments de longueur* `length/3`*.*

☞ *En utilisant turtle, il n'y a aucun calcul à faire sur les coordonnées.*

Jusqu'ici, on n'utilisait qu'une seule tortue, et c'est implicitement à cette tortue que les fonctions comme `forward` s'appliquaient. Nous allons maintenant utiliser turtle avec plusieurs tortues : chaque tortue sera un objet (instance de la classe `RawTurtle`), et on utilisera les méthodes de cet objet (e.g. `t.forward()`) au lieu des fonctions (e.g. `forward()`).

Exercice 3.3 *Dans le fichier* `koch_oo_turtle.py`*, complétez l'implémentation de la fonction* `koch(t, length)`*. Une première tortue* `t1` *est déjà instanciée : faites de même pour instancier une deuxième tortue* `t2`*, puis faites dessiner des flocons de Koch aux deux tortues.*

⚠ *Cette fois, on a utilisé* `import turtle` *et non* `from turtle import *` *pour ne pas polluer notre espace de nom.*

3.1.2 Dessin dans un canevas en coordonnées carthésiennes : `koch_coord.py`

Le dessin avec turtle nous a évité des calculs de coordonnées, mais on peut faire le même dessin en coordonnées cartésiennes.

Exercice 3.4 *Complétez* `koch_coord.py`*. Cette fois-ci, la fonction* `koch` *prend en argument les coordonnées* `x1, x2` *du point de départ, et* `y1, y2` *du point d'arrivée. Pour aider au débogage, on a aussi un paramètre* `depth` *qui permet d'arrêter le tracer après un nombre fixe d'itérations (on passera* `depth - 1` *en paramètre aux appels récursifs).*

3.1.3 Utilisation des nombres complexes pour simplifier les calculs : `koch_complex.py`

L'utilisation de nombres complexes pour représenter les coordonnées peut simplifier le code (une variable complexe par point au lieu de deux coordonnées) et les calculs (une rotation de $\pi/3$ peut se faire en multipliant par $e^{i\pi/3}$, noté « `cmath.rect(1, math.pi/3)` » en Python).

Exercice 3.5 *Dans* `koch_complex.py`*, complétez l'implémentation de la fonction* `koch`*.*

3.2 Dessin d'image pixel par pixel avec la fractale de Mandebrodt : `mandelbrot.py`

Jusqu'ici, nos avons tracé des figures constituées de segments. Nous nous intéressons maintenant au dessin pixel par pixel (bitmap), en utilisant la classe `PhotoImage` de Tkinter.

Exercice 3.6 *Complétez les fonctions* `mandel` *et* `draw` *du fichier* `mandelbrot.py`*.*

☞ *L'opérateur puissance est* `**` *en Python (i.e.* `x ** y` *calcule* x^y*).*

On peut expérimenter les clics des boutons 1 et 2 de la souris pour zoomer et raffiner le dessin.

Exercice 3.7 *Si le temps le permet, implémentez une fonction* `unzoom` *qui fait le contraire de* `zoom`*, appelée sur un clic du bouton 3.*

4 Recherche de zéro d'une fonction continue

Répertoire du squelette : `root/`

Notions abordées :

- Méthode numérique simple de résolution d'équation
- Ordre supérieur : passage de fonction en argument d'une autre fonction
- Fonctions imbriquées

On cherche à résoudre une équation du type $f(x) = 0$ (i.e. trouver une racine de f) sur un intervalle $[a, b]$, avec $f(a) \leq 0$ et $f(b) \geq 0$ numériquement. Pour cela, on teste la valeur $f(\frac{a+b}{2})$:

- Si elle est nulle, on peut arrêter la recherche : on vient de trouver la racine,
- Si elle est négative, alors la solution est dans $[a, \frac{a+b}{2}[$,
- Si elle est positive, alors la solution est dans $]\frac{a+b}{2}, b]$.

Dans les deux derniers cas, on fait un appel récursif sur l'intervalle contenant la racine. On arrête la récursion quand l'intervalle est suffisamment petit.

4.1 Recherche dichotomique : `find_root.py`

Dans `find_root.py`, on fournit une fonction `find_root_binsearch` qui s'occupe de tester la précondition ($f(a) \leq 0$ et $f(b) \geq 0$) compléter la fonction, et appelle `find_root_binsearch_internal`. Cette deuxième fonction est récursive, et ne refera pas les tests de précondition.

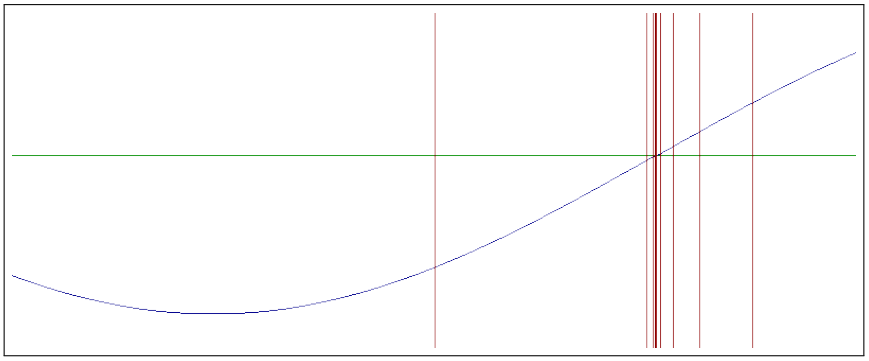
Exercice 4.1 *Compléter la fonction* `find_root_binsearch_internal` *: condition d'arrêt, calcul de* $f(\frac{a+b}{2})$*, et appel récursif sur le bon intervalle.*

☞ *La fonction* `find_root_binsearch` *a un argument* `precision` *qui ne changera pas pendant le calcul. La fonction* `find_root_binsearch_internal` *ne prend pas d'argument* `precision`*; elle utilise directement l'argument de* `find_root_binsearch`*. Vu que c'est une fonction imbriquée, les variables et paramètres de la fonction englobante restent visibles.*

☞ *L'argument* `f` *de la fonction* `find_root_binsearch` *est lui-même une fonction. Ça ne pose pas de problème en Python : on peut manipuler* `f` *soit comme une variable (e.g.* `g = f`*), soit comme une fonction (e.g.* `f(x)`*).*

4.2 Représentation graphique d'une trace d'exécution : `find_root_graphical.py`

Pour voir ce qu'il se passe pendant la résolution, `find_root_graphical.py` est une variante de `find_root.py` qui trace la courbe de la fonction, et va ajouter un trait vertical sur la courbe pour chaque valeur de f évaluée :



Si le temps a passé trop vite, regardez rapidement le corrigé et passez aux questions suivantes.

Exercice 4.2 *Complétez la fonction* `find_root_binsearch_internal` *de* `find_root_graphical.py`*. L'algorithme est le même que précédemment, mais il faut ajouter une instruction* `c.create_line(`x_1`,` y_1`,` x_2`,` y_2`, fill='red')` *où* (x_1, y_1) *et* (x_2, y_2) *sont les extrémités de la ligne à tracer.*

⚠ (x_1, y_1) *et* (x_2, y_2) *sont exprimés en pixels, en partant du coin supérieur gauche de la fenêtre. Il faut utiliser* `math_to_canvas()` *pour transformer une abscisse mathématique en abscisse dans la fenêtre de dessin.*

5 Manipulation d'arbres d'expression

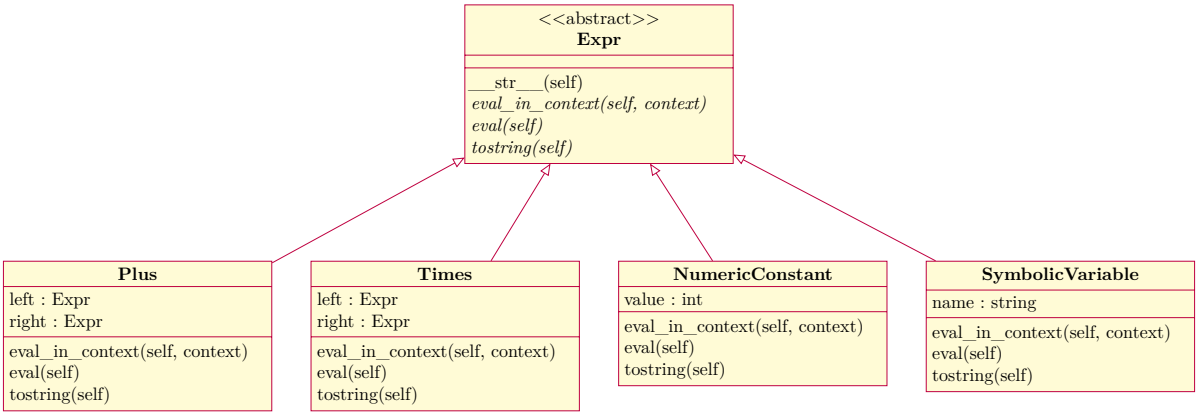
Répertoire du squelette : `expr/`

Notions abordées :

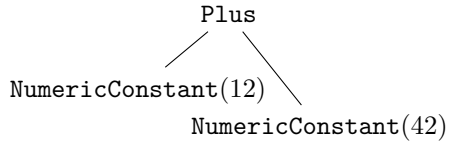
- Classes : `http://docs.python.org/2/tutorial/classes.html`
- Surcharge d'opérateurs : `http://docs.python.org/2/reference/datamodel.html#special-method-names`
- Le patron de conception (design pattern) interprète : `http://fr.wikipedia.org/wiki/Interpr%C3%A9teur_%28patron_de_conception%29`

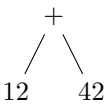
Nous allons maintenant écrire un mini programme de calcul symbolique, en utilisant la programmation objet. Les concepts manipulés ici sont également proches de ceux utilisés dans un compilateur.

Votre squelette contient un fichier `expr.py` qui implémente la structure de données et les calculs, et `test_expr.py` qui permet de tester l'implémentation. Pour chaque construction mathématique (opérateurs `+` et `*`, variable, constante numérique), `expr.py` définit une classe. Une classe de base commune à toutes ces classe est `Expr`. On peut voir la relation d'héritage comme la relation « est un » : `NumericConstant` dérive de `Expr`, donc une constante numérique *est une* expression. On doit pouvoir utiliser une instance de `NumericConstant` partout où le type `Expr` est accepté. On parle aussi de sous-typage : `NumericConstant` est un sous-type de `Expr`. La relation d'héritage entre les classes est illustrée ci-dessous (c'est un diagramme de classes UML) :

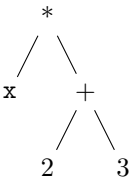


Par exemple, l'expression « 12 + 42 » est représentée par l'arbre



que l'on peut également représenter par 

.

L'expression « x * (2 + 3) » est représentée par l'arbre 

. On peut noter que les arbres d'expressions n'étant pas ambigus, il n'est pas nécessaire de représenter les parenthèses.

Nous utilisons le patron de conception « interprète » pour parcourir les arbres d'expression : par exemple, pour afficher l'expression sous forme de chaîne, on implémente une méthode `tostring` pour chaque classe. Si la classe contient elle-même des sous-expressions (par exemple, les sous-expressions `left` et `right` de la classe `Plus`), alors elle fera des appels récursifs à la même méthode sur ces sous-expressions. La résolution de l'appel de méthode sur un objet appellera la méthode du bon objet. Pour l'exemple de la classe `Plus`, la méthode est donc :

```
def tostring(self):
    return '(' + self.left.tostring() + ' + ' + self.right.tostring() + ')'
```

Il n'est pas nécessaire de connaître le type de `left` et `right` pour écrire ce code : Python fera ce qu'il faut pour appeler les méthodes `tostring` des bonnes classes.

Dans des langages comme Java ou C++, il serait nécessaire de déclarer toutes les méthodes dans `Expr`, mais en Python, la résolution de la méthode est faite dynamiquement, et ce n'est pas nécessaire.

Exercice 5.1 *Observer l'implémentation de la classe* `Plus`*, et adaptez-la à la classe* `Times`*. Tester avec* `test_expr.py`*.*

☞ *Pour faire les choses plus proprement, on aurait pu introduire une classe* `BinaryOperator`*, dérivant de* `Expr` *et dont* `Plus` *et* `Times` *hériteraient. Ainsi, on aurait pu factoriser l'essentiel du code dans la classe* `BinaryOperator`*.*

Nos expressions peuvent contenir des variables symboliques (par exemple, `x + 42`). On peut les évaluer dans un *contexte*, qui donne une valeur à chaque variable. Pour nous, un contexte est un dictionnaire Python qui associe à chaque nom de variable une valeur numérique (par exemple, `{'x' : 12}`).

Exercice 5.2 *Implémenter la fonction* `eval_in_contexte` *de la classe* `SymbolicVariable`*.*

Devoir écrire `Plus(x, NumericConstant(42))` pour construire l'expression `x + 42` est un peu long à écrire. On va remplacer l'appel explicite au constructeur de `Plus` par un opérateur `+` pour permettre d'écrire `x + NumericConstant(42)`. Python traduira cette expression en `x.__add__(NumericConstant(42))`, donc il nous suffit d'implémenter une méthode `__add__` dans la classe `Expr`.

Exercice 5.3 *Implémenter les méthodes* `__add__` *et* `__mul__` *dans la classe* `Expr`*.*

Cerise sur le gâteau : maintenant que nous avons une représentation symbolique des expressions, nous pouvons faire des manipulation symboliques dessus.

Exercice 5.4 *Ajouter une méthode* `derivative(variable)` *à chaque classe dérivant de* `Expr`*, qui calcule la dérivée symboliquement par rapport à la variable* `variable`*, en appliquant les règles :*

- $(f + g)' = f' + g'$,
- $(f.g)' = f'.g + f.g'$,
- *La dérivée d'une constante est nulle (*`NumericConstant(0)`*),*
- *La dérivée d'une variable par rapport à elle-même est 1, la dérivée d'une variable par rapport à une autre est 0.*

☞ *Les expressions obtenues sont souvent plus compliquées que nécessaires (par exemple, la dérivée de* `x * x` *est* `((1 * x) + (x * 1))` *alors qu'on aurait aimé obtenir directement* `2 * x`*) : si on voulait faire mieux, on pourrait ajouter une passe de simplification des expressions.*