Introducción

El propósito de este laboratorio es familiarizarse con la programación de Python y diagnosticar sus habilidades con este lenguaje.

Recursos:

https://docs.python.org/es/3/tutorial/ - Manual oficial de python

Python

Existen una serie de versiones de python actualmente disponibles que van desde la 2.7x hasta las 3.xx. En este laboratorio utilizaremos la versión 3.xx la cual se puede descargar desde el siguiente link https://www.python.org/downloads/

Respuesta de Preguntas

El archivo principal de este laboratorio es el lab0.py donde tendrán que completar los procedimientos que se encuentran en el código. Para esto tendrán que abrir el archivo con el IDLE de su preferencia. Los métodos incompletos se encuentran detallados en la primera sección. Para los métodos que se necesitan resolver en la segunda sección se deben rellenar los métodos que se encuentran en el archivo algebra.py

Probar el código

El laboratorio contiene un archivo llamado tester.py. Este archivo comprueba las respuestas del laboratorio. Para los problemas en el cual ustedes deben proveer una función, el tester comprobará la función con distintas entradas y revisará si la salida es la correcta. Para ejecutar el tester simplemente hay que ejecutar el código en el IDLE de su preferencia.

Primera Sección

Calentamiento

- cube(n) - Toma un número y retorna el cubo de este. Ejemplo: cube(3) = 27.
- factorial(n) - Toma un número entero no negativo n y retornar n!. El cual es el total de los productos de los números enteros desde 1 hasta n. (considere que por definición el factorial de 0 es 1).
  Se sugiere que se utilicen mensajes de error cuando la entrada es inválida. Por ejemplo, cuando se rechace el factorial de un número negativo se muestre un mensaje del error como esto:
  `raise Exception("Factorial: el número no puede ser negativo")`

- count_pattenr(pattern, lst) - Cuenta cuantas veces se repite un patrón de símbolos en una lista, incluyendo los patrones que se sobreponen. Por ejemplo, `count_pattern( ('a', 'b'), ('a', 'b', 'c', 'e', 'b', 'a', 'b', 'f'))` retorna 2, y `count_pattern( ('a', 'b'), ('a', 'b', 'c', 'e', 'b', 'a', 'b', 'f'))` retorna 3.

Profundidad de expresión

Una forma de medir la complejidad matemática de una expresión es a través de la profundidad de la expresión descrita en una lista entregada por python. Escriba un programa que encuentre la profundidad de una expresión.

- `depth('x') => 0`
- `depth(('expt', 'x', 2)) => 1`
- `depth(('+', ('expt', 'x', 2), ('expt', 'y', 2))) => 2`
- `depth(('/', ('expt', 'x', 5), ('expt', ('-', ('expt', 'x', 2), 1), ('/', 5, 2)))) => 4`

Nota: Se puede utilizar la función “isinstance()”. Es una función donde le pasamos el objeto y el nombre de la clase y nos indicará si ésta es o no instancia de la clase indicada, devolviendo un booleano.

```
>>> x = [1, 2, 3]
>>> y = "hi!"
>>> isinstance(x, (list, tuple))
True
>>> isinstance(y, (list, tuple))
False
```

Arbol de referencia

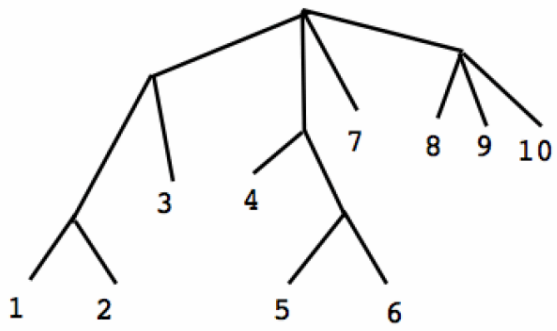

Figure 1: Example Tree

En esta sección tendrán que escribir una función que recibe como entrada un árbol (tree) y un índice, y retorna una parte del árbol (una hoja o un subárbol) de ese índice. Para árboles, los índices deberán ser listas de números enteros. Considere el árbol de la Figura 1, representado por esta tupla de Python: (((1, 2), 3), (4, (5, 6)), 7, (8, 9, 10))

Ejemplo, si quisiéramos seleccionar el elemento 9 del árbol se utiliza la función `tree_ref(tree, (3, 1))` (Notar que el índice comienza desde el número 0, y los indices estan ordenados vienen desde arriba hacia abajo; entonces los índices (3,1) significa que deberíamos tomar la cuarta raíz del árbol principal, y luego tomar la segunda rama del subárbol). Otro ejemplo, si queremos retornar el elemento 6 la llamada debería ser de la siguiente manera `tree_ref(tree, (1, 1, 1))`
Notar que está bien que el resultado retornado puede ser tanto una hoja como un subárbol. Entonces `tree_ref(tree, (0,))` retornaría ((1, 2), 3).

Segunda Sección

Álgebra Simbólica

En este problema tendrán que completar un sistema de computación algebraica simple que que reduce expresiones anidadas de sumas y productos en una sola suma de productos. Por ejemplo, convierte la expresión (2 * (x + 1) * (y + 3)) en ((2 * x * y) + (2 * x * 3) + (2 * 1 * y) + (2 * 1 * 3)). Esta expresión se puede volver a simplificar en ((2 * x * y) + (6 * x) + (2 * y) + 6)), pero esto no es necesario.

Para esto es útil recordar la ley asociativa y la ley distributiva.

Associative law

((a + b) + c) = (a + (b + c)) = (a + b + c)

((a * b) * c) = (a * (b * c)) = (a * b * c)

Distributive law

((a + b) * (c + d)) = ((a * c) + (a * d) + (b * c) + (b * d))

El código para esta parte del laboratorio se realiza en algebra.py. Esta define una clase abstracta llamada expresión. Sum y Product expresiones, y un método llamado Expression.simplify(). Este método comienza por aplicar la ley de asociatividad implementada, pero no cumplirá la ley distributiva. Para eso, se delega en una función llamada do_multiply() que debe escribir. Lea la documentación en el código para obtener más detalles.

Reconocimiento

Este laboratorio está basado en el curso de Artificial Intelligence de Mit Open Courseware[1]

Patrick Winston. *6.034 Artificial Intelligence.* Fall 2010. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.