*Symbol table - documentation*

## The implementation

The symbol table is implemented using a hash table. The hash table is a data structure used to hold key-value pairs. It can perform basic operations on a set of data (addition, look-up, deletion) in Θ(1) time complexity (on average).

The main idea of the hash table is as follows: a hash table is composed of ***m*** different buckets, each having an index. The hash table uses a ***hash function*** to compute an index for a bucket (for a given key). The found bucket contains the desired element (or can be used to store one). Ideally, a hash function is injective. In reality, the hash function can map two different values to the same bucket, causing a ***collision***. Below are the descriptions for the hash function and collision resolution method:

## Hash function:

header: hash_token(token)
parameter: token (type: any)
returns: hash_value (type: integer)
Each token is converted to a string (called **str**). The hash value is then:

$value = (\sum_{index=0}^{size\ of\ str} ASCII(str[index]) \cdot p^{index}) \ modulo\ m$ , where p is a chosen prime number (31), m is the number of buckets and ASCII(v) is the ASCII code of the character v.

## Collision resolution method:

As the resolution method, I chose separate chaining. This means that all colliding elements (elements having the same computed hash value) are stored in the same bucket (which is actually a Python list). The search of an element is reduced to a search in the bucket, while the addition of an element means adding an element in the list of the bucket.

Both these operations have O(n) time complexity, but since the collisions happen rarely, the average is Θ(α), where α is the load factor (number of elements divided by the number of buckets). When the load factor gets too big (>0.8), we can increase the size of the hash table and rehash each element. This ensures the efficiency of the data structure.

## The symbol table - the position method

The method used to find a position for a token (or to create a new one if it does not exist) will search the given token in the hash table. If it finds it, it returns it's position (in the ST), else, it adds it in the hash table and returns the new position (in the ST). Details:

header: get_token_position(token)
parameter: token (type: any)
return: integer (position in the symbol table)

Class diagram: