The Hash Function

This is the first building block of a hash table. Let's see how it works.

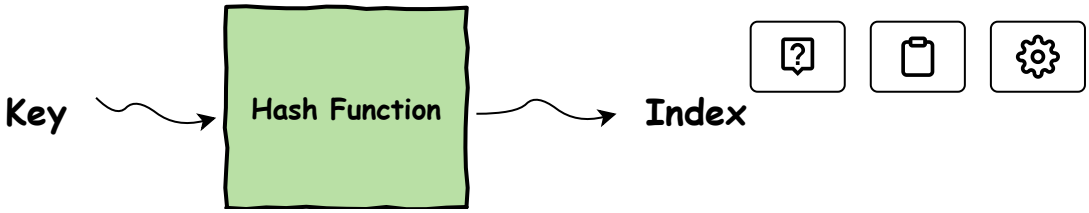
We'll cover the following

Restricting the Key Size#

In the last lesson, we learned that a list can be used to implement a hash table in Python. A **key** is used to map a value on the list and the efficiency of a hash table depends on how a key is computed. At first glance, you may observe that we can directly use the indices as keys because each index is unique.

The only problem is that the key would eventually exceed the size of the list and, at every insertion, the list would need to be resized. Syntactically, we can easily increase list size in Python, but as we learned before, the process still takes *O(n)* time at the back end.

In order to limit the range of the keys to the boundaries of the list, we need a function that converts a large key into a smaller key. This is the job of the **hash function**.



A hash function simply takes an item's key and returns the corresponding index in the list for that item. Depending on your program, the calculation of this index can be a simple arithmetic or a very complicated encryption method. However, it is very important to choose an efficient hashing function as it directly affects the performance of the hash table mechanism.

Let's have a look at some of the most common hash functions used in modern programming.

Arithmetic Modular#

In this approach, we take the modular of the key with the list size:

$$index = key\ MOD\ table\_size$$

Hence, the `index` will always stay between `0` and `tableSize - 1`.

```
def hash_modular(key, size):
    return key % size


lst = [None] * 10  # List of size 10
key = 35
index = hash_modular(key, len(lst))  # Fit the key into the list size
print("The index for key " + str(key) + " is " + str(index))
```

Truncation#

Select a part of the key as the index rather than the whole key. For instance, we can use a mod function for this operation, although it does not need to be based on the list size:

$$key = 123456 -> index = 3456$$

```
def hash_trunc(key):
    return key % 1000  # Will always give us a key of up to 3 digits


key = 123456
index = hash_trunc(key)  # Fit the key into the list size
print("The index for key " + str(key) + " is " + str(index))
```

Folding#

Divide the key into small chunks and apply a different arithmetic strategy at each chunk. For example, you add all the smaller chunks together:

$$key = 456789, \ \ chunk = 2 -> index = 45 + 67 + 89$$

```
def hash_fold(key, chunk_size):  # Define the size of each divided portion
    str_key = str(key)  # Convert integer into string for slicing
    print("Key: " + str_key)
    hash_val = 0
    print("Chunks:")
    for i in range(0, len(str_key), chunk_size):
        if(i + chunk_size < len(str_key)):
            # Slice the appropriate chunk from the string
            print(str_key[i:i+chunk_size])
            hash_val += int(str_key[i:i+chunk_size])  # convert into integer
        else:
            print(str_key[i:len(str_key)])
            hash_val += int(str_key[i:len(str_key)])
    return hash_val


key = 3456789
```

```
chunk_size = 2
print("Hash Key: " + str(hash_fold(key, chunk_size)))
```

In the next lesson, we will discuss a serious problem that can occur when dealing with hash tables.