Assignment 2 - Hash Tables in Python

This assignment is a part of the course "Data Structures and Algorithms in Python".

In this assignment, you will implement Python dictionaries from scratch using hash tables.

As you go through this notebook, you will find the symbol **???** in certain places. To complete this assignment, you must replace all the **???** with appropriate values, expressions or statements to ensure that the notebook runs properly end-to-end.

Guidelines

1. Make sure to run all the code cells, otherwise you may get errors like `NameError` for undefined variables.
2. Do not change variable names, delete cells or disturb other existing code. It may cause problems during evaluation.
3. In some cases, you may need to add some code cells or new statements before or after the line of code containing the **???**.
4. Since you'll be using a temporary online service for code execution, save your work by running `jovian.commit` at regular intervals.
5. Questions marked **(Optional)** will not be considered for evaluation, and can be skipped. They are for your learning.
6. If you are stuck, you can ask for help on the community forum. Post errors or ask for hints, but **please don't ask for OR share the full working answer code** on the forum.
7. There are some tests included with this notebook to help you test your implementation. However, after submission your code will be tested with some hidden test cases. Make sure to test your code exhaustively to cover all edge cases.

Important Links

- Submit your work here: https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries
- Ask questions and get help: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88
- Lesson 2 video for review: https://jovian.ai/aakashns/python-binary-search-trees
- Lesson 2 notebook for review: https://jovian.ai/aakashns/python-binary-search-trees

How to Run the Code and Save Your Work

Option 1: Running using free online resources (1-click, recommended): Click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally: To run the code on your computer locally, you'll need to set up Python & Conda, download the notebook and install the required libraries. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Saving your work: You can save a snapshot of the assignment to your Jovian profile, so that you can access it later and continue your work. Keep saving your work by running `jovian.commit` from time to time.

```
project='python-hash-tables-assignment'
```

```
!pip install jovian --upgrade --quiet
```

```
import jovian
jovian.commit(project=project, privacy='secret', environment=None)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Problem Statement - Python Dictionaries and Hash Tables

In this assignment, you will recreate Python dictionaries from scratch using data structure called *hash table*. Dictionaries in Python are used to store key-value pairs. Keys are used to store and retrieve values. For example, here's a dictionary for storing and retrieving phone numbers using people's names.

```
phone_numbers = {
  'Aakash' : '9489484949',
  'Hemanth' : '9595949494',
  'Siddhant' : '9231325312'
}
phone_numbers
```

```
{'Aakash': '9489484949', 'Hemanth': '9595949494', 'Siddhant': '9231325312'}
```

You can access a person's phone number using their name as follows:

```
phone_numbers['Aakash']
```

```
'9489484949'
```

You can store new phone numbers, or update existing ones as follows:

```
# Add a new value
phone_numbers['Vishal'] = '8787878787'
# Update existing value
phone_numbers['Aakash'] = '7878787878'
# View the updated dictionary
phone_numbers
```

```
{'Aakash': '7878787878',
 'Hemanth': '9595949494',
 'Siddhant': '9231325312',
 'Vishal': '8787878787'}
```

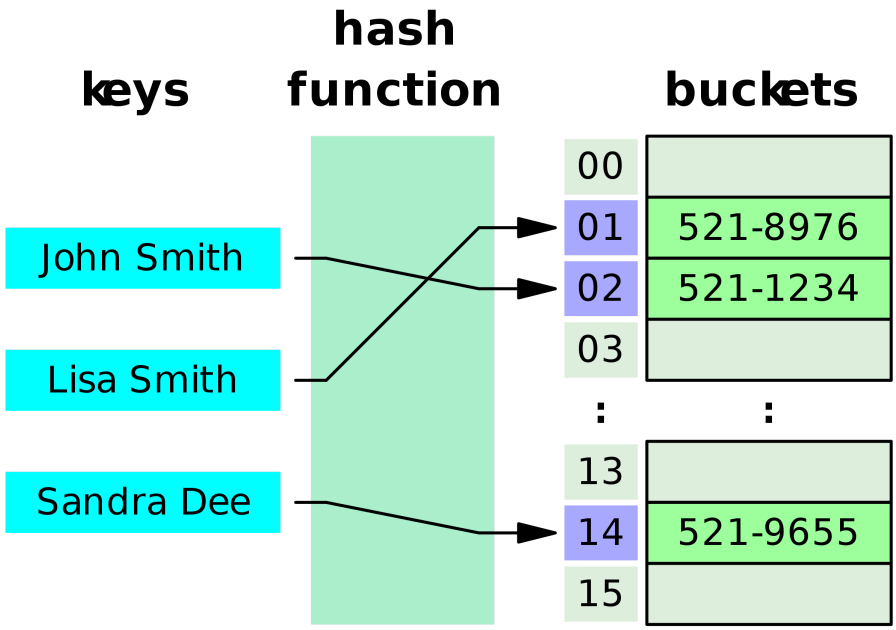
You can also view all the names and phone numbers stored in `phone_numbers` using a `for` loop.

```
for name in phone_numbers:
    print('Name:', name, ', Phone Number:', phone_numbers[name])
```

```
Name: Aakash , Phone Number: 7878787878
Name: Hemanth , Phone Number: 9595949494
Name: Siddhant , Phone Number: 9231325312
Name: Vishal , Phone Number: 8787878787
```

Dictionaries in Python are implemented using a data structure called **hash table**. A hash table uses a list/array to store the key-value pairs, and uses a *hashing function* to determine the index for storing or retrieving the data associated with a given key.

Here's a visual representation of a hash table (source):



Your objective in this assignment is to implement a `HashTable` class which supports the following operations:

1. **Insert**: Insert a new key-value pair
2. **Find**: Find the value associated with a key
3. **Update**: Update the value associated with a key
4. **List**: List all the keys stored in the hash table

The `HashTable` class will have the following structure (note the function signatures):

```
class HashTable:
    def insert(self, key, value):
        """Insert a new key-value pair"""
        pass

    def find(self, key):
        """Find the value associated with a key"""
```

```
        pass

    def update(self, key, value):
        """Change the value associated with a key"""
        pass

    def list_all(self):
        """List all the keys"""
        pass
```

Before we begin our implementation, let's save and commit our work.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Data List

We'll build the `HashTable` class step-by-step. As a first step is to create a Python list which will hold all the key-value pairs. We'll start by creating a list of a fixed size.

```
MAX_HASH_TABLE_SIZE = 4096
```

QUESTION 1: Create a Python list of size `MAX_HASH_TABLE_SIZE`, with all the values set to `None`.

Hint: Use the `*` operator.

```
# List of size MAX_HASH_TABLE_SIZE with all values None
data_list = [None] * MAX_HASH_TABLE_SIZE
```

If the list was created successfully, the following cells should output `True`.

```
len(data_list) == 4096
```

```
True
```

```
data_list[99] == None
```

```
True
```

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on
```

```
https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Hashing Function

A *hashing function* is used to convert strings and other non-numeric data types into numbers, which can then be used as list indices. For instance, if a hashing function converts the string `"Aakash"` into the number `4`, then the key-value pair `('Aakash', '7878787878')` will be stored at the position `4` within the data list.

Here's a simple algorithm for hashing, which can convert strings into numeric list indices.

1. Iterate over the string, character by character
2. Convert each character to a number using Python's built-in `ord` function.
3. Add the numbers for each character to obtain the hash for the entire string
4. Take the remainder of the result with the size of the data list

QUESTION 2: Complete the `get_index` function below which implements the hashing algorithm described above.

```
def get_index(data_list, a_string):
    # Variable to store the result (updated after each iteration)
    result = 0

    for a_character in a_string:
        # Convert the character to a number (using ord)
        a_number = ord(a_character)
        # Update result by adding the number
        result += a_number

    # Take the remainder of the result with the size of the data list
    list_index = result % len(data_list)
    return list_index
```

If the `get_index` function was defined correctly, the following cells should output `True`.

```
get_index(data_list, '') == 0
```

```
True
```

```
get_index(data_list, 'Aakash') == 585
```

```
True
```

```
get_index(data_list, 'Don O Leary') == 941
```

```
True
```

(Optional) Test the `get_index` function using the empty cells below.

```
```

```
```

Insert

To insert a key-value pair into a hash table, we can simply get the hash of the key, and store the pair at that index in the data list.

```
key, value = 'Aakash', '7878787878'
```

```
idx = get_index(data_list, key)
idx
```

```
585
```

```
data_list[idx] = (key, value)
```

Here's the same operation expressed in a single line of code.

```
data_list[get_index(data_list, 'Hemanth')] = ('Hemanth', '9595949494')
```

Find

The retrieve the value associated with a pair, we can get the hash of the key and look up that index in the data list.

```
idx = get_index(data_list, 'Aakash')
idx
```

```
585
```

```
key, value = data_list[idx]
value
```

```
'7878787878'
```

List

To get the list of keys, we can use a simple list comprehension.

```
pairs = [kv[0] for kv in data_list if kv is not None]
```

```
pairs
```

```
['Aakash', 'Hemanth']
```

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Basic Hash Table Implementation

We can now use the hashing function defined above to implement a basic hash table in Python.

QUESTION 3: Complete the hash table implementation below by following the instructions in the comments.

Hint: Insert and update can have identical implementations.

```
class BasicHashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        # 1. Create a list of size `max_size` with all values None
        self.data_list = [None] * max_size


    def insert(self, key, value):
        # 1. Find the index for the key using get_index
        idx = get_index(self.data_list, key)

        # 2. Store the key-value pair at the right index
        self.data_list[idx] = (key, value)


    def find(self, key):
        # 1. Find the index for the key using get_index
        idx = get_index(self.data_list, key)

        # 2. Retrieve the data stored at the index
        kv = self.data_list[idx]

        # 3. Return the value if found, else return None
        if kv is None:
            return None
        else:
            key, value = kv
            return value


    def update(self, key, value):
        # 1. Find the index for the key using get_index
        idx = get_index(self.data_list, key)
        # 2. Store the new key-value pair at the right index
```

```
        self.data_list[idx] = (key, value)


    def list_all(self):
        # 1. Extract the key from each key-value pair
        return [kv[0] for kv in self.data_list if kv is not None]
```

If the `BasicHashTable` class was defined correctly, the following cells should output `True`.

```
basic_table = BasicHashTable(max_size=1024)
len(basic_table.data_list) == 1024
```

```
True
```

```
# Insert some values
basic_table.insert('Aakash', '9999999999')
basic_table.insert('Hemanth', '8888888888')

# Find a value
basic_table.find('Hemanth') == '8888888888'
```

```
True
```

```
# Update a value
basic_table.update('Aakash', '7777777777')

# Check the updated value
basic_table.find('Aakash') == '7777777777'
```

```
True
```

```
# Get the list of keys
basic_table.list_all() == ['Aakash', 'Hemanth']
```

```
True
```

(Optional) Test your implementation of `BasicHashTable` with some more examples below.

```

```

```

```

```

```

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on
```

```
https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Handling Collisions with Linear Probing

As you might have wondered, multiple keys can have the same hash. For instance, the keys `"listen"` and `"silent"` have the same hash. This is referred to as *collision*. Data stored against one key may override the data stored against another, if they have the same hash.

```
basic_table.insert('listen', 99)
```

```
basic_table.insert('silent', 200)
```

```
basic_table.find('listen')
```

```
200
```

As you can see above, the value for the key `listen` was overwritten by the value for the key `silent`. Our hash table implementation is incomplete because it does not handle collisions correctly.

To handle collisions we'll use a technique called linear probing. Here's how it works:

1. While inserting a new key-value pair if the target index for a key is occupied by another key, then we try the next index, followed by the next and so on till we the closest empty location.
2. While finding a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key.
3. While updating a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key, and update its value.

We'll define a function called `get_valid_index`, which starts searching the data list from the index determined by the hashing function `get_index` and returns the first index which is either empty or contains a key-value pair matching the given key.

QUESTION 4: Complete the function `get_valid_index` below by following the instructions in the comments.

```
def get_valid_index(data_list, key):
    # Start with the index returned by get_index
    idx = get_index(data_list, key)

    while True:
        # Get the key-value pair stored at idx
        kv = data_list[idx]

        # If it is None, return the index
        if kv == None:
            return idx
```

```
        # If the stored key matches the given key, return the index
        k, v = kv
        if k == key:
            return idx

        # Move to the next index
        idx += 1

        # Go back to the start if you have reached the end of the array
        if idx == len(data_list):
            idx = 0
```

If `get_valid_index` was defined correctly, the following cells should output `True`.

```
# Create an empty hash table
data_list2 = [None] * MAX_HASH_TABLE_SIZE

# New key 'listen' should return expected index
get_valid_index(data_list2, 'listen') == 655
```

```
True
```

```
# Insert a key-value pair for the key 'listen'
data_list2[get_index(data_list2, 'listen')] = ('listen', 99)

# Colliding key 'silent' should return next index
get_valid_index(data_list2, 'silent') == 656
```

```
True
```

(Optional) Test your implementation of `get_valid_index` on some more examples using the empty cells below.

Let's save our work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Hash Table with Linear Probing

We can now implement a hash table with linear probing.

QUESTION 5: Complete the hash table (with linear probing) implementation below by following the instructions in the comments.

```
class ProbingHashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        # 1. Create a list of size `max_size` with all values None
        self.data_list = [None] * max_size


    def insert(self, key, value):
        # 1. Find the index for the key using get_valid_index
        idx = get_valid_index(self.data_list, key)

        # 2. Store the key-value pair at the right index
        self.data_list[idx] = (key, value)


    def find(self, key):
        # 1. Find the index for the key using get_valid_index
        idx = get_valid_index(self.data_list, key)

        # 2. Retrieve the data stored at the index
        kv = self.data_list[idx]

        # 3. Return the value if found, else return None
        return None if kv is None else kv[1]


    def update(self, key, value):
        # 1. Find the index for the key using get_valid_index
        idx =  get_valid_index(self.data_list, key)

        # 2. Store the new key-value pair at the right index
        self.data_list[idx] = (key, value)


    def list_all(self):
        # 1. Extract the key from each key-value pair
        return [kv[0] for kv in self.data_list if kv is not None]
```

If the `ProbingHashTable` class was defined correctly, the following cells should output `True`.

```
# Create a new hash table
probing_table = ProbingHashTable()
```

```
# Insert a value
probing_table.insert('listen', 99)

# Check the value
probing_table.find('listen') == 99
```

```
True
```

```
# Insert a colliding key
probing_table.insert('silent', 200)

# Check the new and old keys
probing_table.find('listen') == 99 and probing_table.find('silent') == 200
```

```
True
```

```
# Update a key
probing_table.insert('listen', 101)

# Check the value
probing_table.find('listen') == 101
```

```
True
```

```
probing_table.list_all() == ['listen', 'silent']
```

```
True
```

(Optional) Test your implementation of `ProbingHashTable` using the empty cells below.

```

```

```

```

```

```

Save your work before continuing.

```
jovian.commit(project=project)
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
```

```
'https://jovian.ai/megha-goyate/python-hash-tables-assignment'
```

Make a Submission

Congrats! You have now implemented hash tables from scratch. The rest of this assignment is optional.

You can make a submission on this page: https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries . Submit the link to your Jovian notebook (the output of the previous cell).

You can also make a direct submission by executing the following cell:

```
jovian.submit(assignment="pythondsa-assignment2")
```

```
[jovian] Updating notebook "megha-goyate/python-hash-tables-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/megha-goyate/python-hash-tables-assignment
[jovian] Submitting assignment..
[jovian] Verify your submission at https://jovian.ai/learn/data-structures-and-algorithms-in-python/assignment/assignment-2-hash-table-and-python-dictionaries
```

If you are stuck, you can get help on the forum: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88

(Optional) Python Dictionaries using Hash Tables

We can now implement Python dictionaries using hash tables. Also, Python provides a built-in function called `hash` which we can use instead of our custom hash function. It is likely to have far fewer collisions

(Optional) Question: Implement a python-friendly interface for the hash table.

```
MAX_HASH_TABLE_SIZE = 4096

class HashTable:
    def __init__(self, max_size=MAX_HASH_TABLE_SIZE):
        self.data_list = [None] * max_size

    def get_valid_index(self, key):
        # Use Python's in-built `hash` function and implement linear probing
         # change this
        return str(hash(self.key))

    def __getitem__(self, key):
        # Implement the logic for "find" here
        idx = get_valid_index(self, key)
        kv = self.data_list[idx]
        return kv[1]

    def __setitem__(self, key, value):
        # Implement the logic for "insert/update" here
        idx = get_valid_index(self, key)
        self.data_list[idx] = (key, value)

    def __iter__(self):
```

```
        return (x for x in self.data_list if x is not None)

    def __len__(self):
        return len([x for x in self])

    def __repr__(self):
        from textwrap import indent
        pairs = [indent("{} : {}".format(repr(kv[0]), repr(kv[1])), '  ') for kv in sel
        return "{\n" + "{}".format(',\n'.join(pairs)) + "\n}"

    def __str__(self):
        return repr(self)
```

If the `HashTable` class was defined correctly, the following cells should output `True`.

```
# Create a hash table
table = HashTable()

# Insert some key-value pairs
table['a'] = 1
table['b'] = 34

# Retrieve the inserted values
table['a'] == 1 and table['b'] == 34
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_56/3572160856.py in <module>
      3
      4 # Insert some key-value pairs
----> 5 table['a'] = 1
      6 table['b'] = 34
      7

/tmp/ipykernel_56/3848394.py in __setitem__(self, key, value)
     18     def __setitem__(self, key, value):
     19         # Implement the logic for "insert/update" here
---> 20         idx = get_valid_index(self, key)
     21         self.data_list[idx] = (key, value)
     22

/tmp/ipykernel_56/975592262.py in get_valid_index(data_list, key)
      1 def get_valid_index(data_list, key):
      2     # Start with the index returned by get_index
----> 3     idx = get_index(data_list, key)
      4
      5     while True:

/tmp/ipykernel_56/3942751503.py in get_index(data_list, a_string)
     10
     11     # Take the remainder of the result with the size of the data list
```

```
---> 12     list_index = result % len(data_list)
     13     return list_index

ZeroDivisionError: integer division or modulo by zero
```

```
# Update a value
table['a'] = 99

# Check the updated value
table['a'] == 99
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_56/2218104034.py in <module>
      1 # Update a value
----> 2 table['a'] = 99
      3
      4 # Check the updated value
      5 table['a'] == 99

/tmp/ipykernel_56/3848394.py in __setitem__(self, key, value)
     18     def __setitem__(self, key, value):
     19         # Implement the logic for "insert/update" here
---> 20         idx = get_valid_index(self, key)
     21         self.data_list[idx] = (key, value)
     22

/tmp/ipykernel_56/975592262.py in get_valid_index(data_list, key)
      1 def get_valid_index(data_list, key):
      2     # Start with the index returned by get_index
----> 3     idx = get_index(data_list, key)
      4
      5     while True:

/tmp/ipykernel_56/3942751503.py in get_index(data_list, a_string)
     10
     11     # Take the remainder of the result with the size of the data list
---> 12     list_index = result % len(data_list)
     13     return list_index

ZeroDivisionError: integer division or modulo by zero
```

```
# Get a list of key-value pairs
list(table) == [('a', 99), ('b', 34)]
```

```
False
```

Since we have also implemented the `__repr__` and `__str__` functions, the output of the next cell should be:

```
{
  'a' : 99,
```

```
    'b' : 34
}
```

```
table
```

```
{

}
```

Let's save our work before continuing.

```
import jovian
```

```
jovian.commit(project=project)
```

(Optional) Hash Table Improvements

Here are some more improvements/changes you can make to your hash table implementation:

- **Track the size of the hash table** i.e. number of key-value pairs so that `len(table)` has complexity O(1).
- **Implement deletion with tombstones** as described here: https://research.cs.vt.edu/AVresearch/hashing/deletion.php
- **Implement dynamic resizing** to automatically grow/shrink the data list: https://charlesreid1.com/wiki/Hash_Maps/Dynamic_Resizing
- **Implement separate chaining**, an alternative to linear probing for collision resolution: https://www.youtube.com/watch/T9gct6Dx-jo

(Optional) Complexity Analysis

With choice of a good hashing function and other improvements like dynamic resizing, you can

Operation	Average-case time complexity	Worst-case time complexity
Insert/Update	O(1)	O(n)
Find	O(1)	O(n)
Delete	O(1)	O(n)
List	O(n)	O(n)

Here are some questions to ponder upon?

- What is average case complexity? How does it differ from worst-case complexity?
- Do you see why insert/find/update have average-case complexity of O(1) and worst-case complexity of O(n) ?
- How is the complexity of hash tables different from binary search trees?
- When should you prefer using hash table over binary trees or vice versa?

Discuss your answers on the forum: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/assignment-2/88