Carleton University
Department of Systems and Computer Engineering
SYSC 2100 - Algorithms and Data Structures - Winter 2021

Assignment 2 - Implementing a Map Using a Hash Table

A *map* (also known as a dictionary or an associative array) is a collection of (key, value) pairs. Keys are unique; that is, a map doesn't store duplicate keys. Duplicate values are permitted; that is, two or more keys can be associated with the same value.

In this assignment, you'll implement ADT Map using a hash table as the underlying data structure. The hash table will use chains to resolve collisions. When you're done, you'll have an ADT that supports many of the operations provided by Python's built-in `dict` (dictionary) type.

ADT Map - Implementation Overview

Download hashmap.py from the *Lab Materials* section of the main cuLearn course page.

Class `HashMap` has two instance variables:

- `_slots` refers to an instance of Python's built-in `list` type. Each element in the list is a slot in the hash table, and stores a pointer (a reference) to a chain of (key, value) pairs. In other words, `self._slots[i]` will point to the chain that contains all the (key, value) pairs for which the keys hash to integer `i`.
- `_size` keeps track of the number of items in the map; that is, the number of (key, value) pairs.

Each chain is implemented as a singly-linked list of nodes. Each node is an instance of class `Entry`, and stores:

- a key.
- the value associated with that key.
- the link to the next `Entry` object in the chain.

Class `HashMap` has a complete implementation of `__init__`. You can create a new, empty map this way:

```
map = HashMap()
```

After creating a map, we can insert (key, value) pairs, one at a time:

```
supplies = HashMap()
supplies.put('pencils', 1)
supplies.put('erasers', 2)
supplies.put('pens', 5)
```

`__init__` takes an optional argument, which must be an *iterable*. (An iterable is an object that is capable of returning its members one at a time. Instances of Python's `list` and `tuple` types and `range` objects, are three examples of iterables.) The iterable must contain a sequence of (key, value) pairs.

The statement:

```
supplies = HashMap([('pencils', 1), ('erasers', 2), ('pens', 5)])
```

passes a list of 2-tuples to `HashMap`'s `__init__` method. The first element in each tuple is a key, and the second element is the value associated with the key. The `HashMap` assigned to `supplies` will contain three (key, value) pairs: `('pencils', 1)`, `('erasers', 2)`, and `('pens', 5)`.

Note: `__init__` calls `put`, which you have to write, so you won't be able to create a map from an iterable until that task is finished.

`HashMap` also has a complete implementation of `__str__`. Here's an example of the strings produced by this method:

```
>>> supplies = HashMap()
>>> supplies.put('pencils', 1)
>>> supplies.put('erasers', 2)
>>> supplies.put('pens', 5)
>>> str(supplies)
"{'erasers': 2, 'pens': 5, 'pencils': 1}"
```

If the strings look familiar, it's because string representations produced by Python's `dict` type use the same format.

`HashMap` has a complete implementation of `__repr__`, which returns the same strings as `__str__`.

"Stub" implementations have been provided for methods `_hash_function`, `__len__`, `__contains__`, `put`, `get`, and `pop`. If you call any of these methods on a `HashMap` object, Python will throw a `NotImplementedError` exception.

For each method, read the docstring and replace the `raise` statement with a correct implementation of the method.

I recommend that you use an incremental, iterative approach: as you write each method, test it

before you move on to the next one. Use the shell or write a short script to test the methods. If you write a script, please put it in a separate module (file). We don't want test scripts in the hashmap.py file you submit to cuLearn.

I suggest you implement the methods in this order.

- `_hash_function`. This method is trivial: use Python's built-in `hash` function to hash the key (let Python do the hard work!), take the absolute value of the hash value, then use the `%` operator to calculate the slot in the hash table. (This is discussed in the lecture slides on hashing.)
- `__len__`. This method should be *O*(1); in other words, it shouldn't have loops that iterate over the chains.
- `put`. Test this method thoroughly. After each call to `put`, does `__len__` return the correct number of (key, value) pairs in the map? After testing `put`, verify that you can create a map by passing an iterable to `__init__`.
- `get`. Remember to test this method with an empty map and a non-empty map, and with keys that are in the map and keys that aren't in the map. Does your method work when you provide a value for the `default` argument? When you don't provide a value for the `default` argument?
- `__contains__`. Remember to test this method with an empty map and a non-empty map, and with keys that are in the map and keys that aren't in the map.
- `pop`. Remember to test this method with an empty map and a non-empty map, and with keys that are in the map and keys that aren't in the map. Does your method work when you provide a value for the `default` argument? When you don't provide a value for the `default` argument?

Programming Style - Code Formatting

Before submitting your code, please run it through Wing 101's source reformatter, as described in following paragraphs.

Wing 101 can easily be reconfigured to reformat your code so that it adheres to some of the formatting conventions described in the "official" Python coding conventions document, PEP 8 - *Style Guide for Python Code*.[1]

To configure reformatting, follow the instructions in Section 3.2 of *Installing Python 3.9.1 and Wing 101 7.2.7*, which is posted on cuLearn. (The instructions apply to both the Windows 10 and

[1] https://www.python.org/dev/peps/pep-0008/

macOS versions of Wing 101.)

If you prefer, you manually reformat your files even if you've disabled automatic reformatting. This is described in Section 3.3. Note that you still have to open the Auto-formatting window to configure which PEP 8 conventions are followed, as described in Section 3.2.

Manually reformatting an open file is easy:

- From the menu bar, select Source > Reformatting
- From the pop-up menu, select Reformat File for PEP 8

Wing 101 7.2.8 fixes a bug in release 7.2.7: the Reformat Selection for PEP 8 command now works.

Programming Style - Writing Readable Code

Focus on writing code that is readable and maintainable, and not just code "that works". Using a code reformatter can improve the layout of your code, but it can't enforce every coding convention. For example, a reformatter won't change your variable names if they aren't descriptive or follow the "official" Python convention (that is, variables names are lowercase, with words separated by underscores).

Here are some things to consider:[2]

"The ease by which other *people* can read and understand a program (often called "readability" in software engineering) is perhaps the most important quality of a program. Readable programs are used and extended by others, sometimes for decades. For this reason, we often say that programs are written to be read by humans, and only incidentally to be interpreted by computers.

A program is composed well if it is concise, well-named, understandable, and easy to follow.

Excellent composition does not mean adhering strictly to prescribed style conventions. There are many ways to program well, just as there are many styles of effective communication. However, the following guiding principles universally lead to better composition of programs:

- **Names**. To a computer, names are arbitrary symbols: "xegyawebpi" and "foo" are just as meaningful as "tally" and "denominator". To humans, comprehensible names aid immensely in comprehending programs. Choose names for your functions and variables that indicate their use, purpose, and meaning.
- **Functions**. Functions are our primary mechanism for abstraction, and so each function should ideally have a single job that can be used throughout a program. When given the

[2] Adapted from a style guide used by students in an introductory programming course at the University of California, Berkeley.

choice between calling a function or copying and pasting its body, strive to call the function and maintain abstraction in your program.

- **Purpose**. Each line of code in a program should have a purpose. Statements should be removed if they no longer have any effect (perhaps because they were useful for a previous version of the program, but are no longer needed). Large blocks of unused code, even when turned into comments, are confusing to readers. Feel free to keep your old implementations in a separate file for your own use, but don't turn them in as your finished product.
- **Brevity**. An idea expressed in four lines of code is often clearer than the same idea expressed in forty. You do not need to try to minimize the length of your program, but look for opportunities to reduce the size of your program substantially by reusing functions you have already defined."

Wrap Up

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline.

The submission deadline for this assignment is Friday, March 26, 23:55 (Ottawa time), for all lab sections.

To submit your lab work, go to the cuLearn page **for your lab section** (not the main course page). Submit hashmap.py. Ensure you submit the version of the file that contains your solutions, and not the unmodified file you downloaded from cuLearn! You are permitted to make changes to your solutions and resubmit the file as many times as you want, up to the deadline. Only the most recent submission is saved by cuLearn.

Last edited: March 15, 2021