Exercise 39: Dictionaries, Oh Lovely Dictionaries

Now I have to hurt you with another container you can use, because once you learn this container a massive world of ultra-cool will be yours. It is the most useful container ever: the dictionary.

Python calls them "dicts." Other languages call them "hashes." I tend to use both names, but it doesn't matter. What does matter is what they do when compared to lists. You see, an list lets you do this:

```
>>> things = ['a', 'b', 'c', 'd']
>>> print things[1]
b
>>> things[1] = 'z'
>>> print things[1]
z
>>> things
['a', 'z', 'c', 'd']
```

You can use numbers to "index" into a list, meaning you can use numbers to find out what's in lists. You should know this about lists by now, but make sure you understand that you can *only* use numbers to get items out of a list.

What a `dict` does is let you use *anything*, not just numbers. Yes, a dict associates one thing to another, no matter what it is. Take a look:

```
>>> stuff = {'name': 'Zed', 'age': 39, 'height': 6 * 12 + 2}
>>> print stuff['name']
Zed
>>> print stuff['age']
39
>>> print stuff['height']
74
>>> stuff['city'] = "San Francisco"
>>> print stuff['city']
San Francisco
```

You will see that instead of just numbers we're using strings to say what we want from the `stuff` dictionary. We can also put new things into the dictionary with strings. It doesn't have to be strings though. We can also do this:

```
>>> stuff[1] = "Wow"
>>> stuff[2] = "Neato"
>>> print stuff[1]
Wow
>>> print stuff[2]
Neato
>>> stuff
{'city': 'San Francisco', 2: 'Neato', 'name': 'Zed', 1: 'Wow', 'age': 39, 'height': 7
4}
```

In this code I used numbers, and then you can see there are numbers and strings as keys in the dict when I print it. I could use anything. Well, almost but just pretend you can use anything for now.

Of course, a dictionary that you can only put things in is pretty stupid, so here's how you delete things, with the `del` keyword:

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
{'name': 'Zed', 'age': 39, 'height': 74}
```

A Dictionary Example

We'll now do an exercise that you *must* study very carefully. I want you to type this code in and try to understand what's going on. Take note of when you put things in e dict, get from a hash, and all the operations you use. Notice how this example is mapping states to their abbreviations, and then the abbreviations to cities in the states. Remember, "mapping" or "associating" is the key concept in a dictionary.

```
# create a mapping of state to abbreviation
states = {
    'Oregon': 'OR',
    'Florida': 'FL',
    'California': 'CA',
    'New York': 'NY',
    'Michigan': 'MI'
}

# create a basic set of states and some cities in them
cities = {
    'CA': 'San Francisco',
    'MI': 'Detroit',
    'FL': 'Jacksonville'
}

# add some more cities
cities['NY'] = 'New York'
cities['OR'] = 'Portland'

# print out some cities
print '-' * 10
print "NY State has: ", cities['NY']
print "OR State has: ", cities['OR']

# print some states
print '-' * 10
print "Michigan's abbreviation is: ", states['Michigan']
print "Florida's abbreviation is: ", states['Florida']

# do it by using the state then cities dict
print '-' * 10
print "Michigan has: ", cities[states['Michigan']]
print "Florida has: ", cities[states['Florida']]

# print every state abbreviation
print '-' * 10
for state, abbrev in states.items():
    print "%s is abbreviated %s" % (state, abbrev)

# print every city in state
print '-' * 10
for abbrev, city in cities.items():
    print "%s has the city %s" % (abbrev, city)

# now do both at the same time
print '-' * 10
for state, abbrev in states.items():
    print "%s state is abbreviated %s and has city %s" % (
        state, abbrev, cities[abbrev])

print '-' * 10
# safely get a abbreviation by state that might not be there
```

```
state = states.get('Texas')

if not state:
    print "Sorry, no Texas."

# get a city with a default value
city = cities.get('TX', 'Does Not Exist')
print "The city for the state 'TX' is: %s" % city
```

What You Should See

```
$ python ex39.py
----------
NY State has:  New York
OR State has:  Portland
----------
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
----------
Michigan has:  Detroit
Florida has:  Jacksonville
----------
California is abbreviated CA
Michigan is abbreviated MI
New York is abbreviated NY
Florida is abbreviated FL
Oregon is abbreviated OR
----------
FL has the city Jacksonville
CA has the city San Francisco
MI has the city Detroit
OR has the city Portland
NY has the city New York
----------
California state is abbreviated CA and has city San Francisco
Michigan state is abbreviated MI and has city Detroit
New York state is abbreviated NY and has city New York
Florida state is abbreviated FL and has city Jacksonville
Oregon state is abbreviated OR and has city Portland
----------
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist
```

What Dictionaries Can Do

Dictionaries are another example of a data structure, and like lists they are one of the most commonly used data structures in programming. A dictionary is used to *map* or *associate* things you want to store to keys you need to get them. Again, programmers don't use a term like "dictionary" for something that doesn't work like an actual dictionary full of words, so let's use that as our real world example.

Let's say you want to find out what the word "Honorificabilitudinitatibus" means. Today you would simply copy-paste that word into a search engine and then find out the answer, and we could say a search engine is like a really huge super complex version of the *Oxford English Dictionary* (OED). Before search engines what you would do is this:

> 1. Go to your library and get "the dictionary". Let's say it's the OED.
> 2. You know "honorificabilitudinitatibus" starts with the letter 'H' so you look on the side of the book for the little tab that has 'H' on it.
> 3. Then you'd skim the pages until you are close to where "hon"

> started.
> 4. Then you'd skim a few more pages until you found "honorificabilitudinitatibus" or hit the beginning of the "hp" words and realize this word isn't in the OED.
> 5. Once you found the entry, you'd read the definition to figure out what it means.

This process is nearly exactly the way a dict works, and you are basically "mapping" the word "honorificabilitudinitatibus" to its definition. A dict in Python is just like a dictionary in the real world like the OED.

In fact, we can implement one like this using just lists.

Making Your Own Dictionary Module

The final piece of code in this exercise will show you how to make a dict data structure using nothing but the lists you already know how to use. This code may be a little difficult to understand, so don't worry if it takes you a while to full grasp what it's doing. There are no new concepts in this code. It is slightly more complex and has a few more things you will need to look up.

To keep things from stepping on Python's `dict` I'm going to call my data structure a `hashmap`, which is another name for a dictionary data structure since it maps one thing to another. You should type this code into a file named `hashmap.py` so we can run it in another file named `ex39_test.py` next.

```python
def new(num_buckets=256):
    """Initializes a Map with the given number of buckets."""
    aMap = []
    for i in range(0, num_buckets):
        aMap.append([])
    return aMap

def hash_key(aMap, key):
    """Given a key this will create a number and then convert it to
    an index for the aMap's buckets."""
    return hash(key) % len(aMap)

def get_bucket(aMap, key):
    """Given a key, find the bucket where it would go."""
    bucket_id = hash_key(aMap, key)
    return aMap[bucket_id]

def get_slot(aMap, key, default=None):
    """
    Returns the index, key, and value of a slot found in a bucket.
    Returns -1, key, and default (None if not set) when not found.
    """
    bucket = get_bucket(aMap, key)

    for i, kv in enumerate(bucket):
        k, v = kv
        if key == k:
            return i, k, v

    return -1, key, default

def get(aMap, key, default=None):
    """Gets the value in a bucket for the given key, or the default."""
    i, k, v = get_slot(aMap, key, default=default)
    return v

def set(aMap, key, value):
    """Sets the key to the value, replacing any existing value."""
    bucket = get_bucket(aMap, key)
    i, k, v = get_slot(aMap, key)

    if i >= 0:
        # the key exists, replace it
        bucket[i] = (key, value)
    else:
        # the key does not, append to create it
        bucket.append((key, value))

def delete(aMap, key):
    """Deletes the given key from the Map."""
    bucket = get_bucket(aMap, key)

    for i in xrange(len(bucket)):
        k, v = bucket[i]
        if key == k:
            del bucket[i]
            break

def list(aMap):
    """Prints out what's in the Map."""
    for bucket in aMap:
        if bucket:
            for k, v in bucket:
                print k, v
```

This will create a module named `hashmap` which you should be able to import and work with using this `ex39_test.py` file:

```
import hashmap

# create a mapping of state to abbreviation
states = hashmap.new()
hashmap.set(states, 'Oregon', 'OR')
hashmap.set(states, 'Florida', 'FL')
hashmap.set(states, 'California', 'CA')
hashmap.set(states, 'New York', 'NY')
hashmap.set(states, 'Michigan', 'MI')

# create a basic set of states and some cities in them
cities = hashmap.new()
hashmap.set(cities, 'CA', 'San Francisco')
hashmap.set(cities, 'MI', 'Detroit')
hashmap.set(cities, 'FL', 'Jacksonville')

# add some more cities
hashmap.set(cities, 'NY', 'New York')
hashmap.set(cities, 'OR', 'Portland')


# print out some cities
print '-' * 10
print "NY State has: %s" % hashmap.get(cities, 'NY')
print "OR State has: %s" % hashmap.get(cities, 'OR')

# print some states
print '-' * 10
print "Michigan's abbreviation is: %s" % hashmap.get(states, 'Michigan')
print "Florida's abbreviation is: %s" % hashmap.get(states, 'Florida')

# do it by using the state then cities dict
print '-' * 10
print "Michigan has: %s" % hashmap.get(cities, hashmap.get(states, 'Michigan'))
print "Florida has: %s" % hashmap.get(cities, hashmap.get(states, 'Florida'))

# print every state abbreviation
print '-' * 10
hashmap.list(states)
```

```

# print every city in state
print '-' * 10
hashmap.list(cities)

print '-' * 10
state = hashmap.get(states, 'Texas')

if not state:
  print "Sorry, no Texas."

# default values using ||= with the nil result
# can you do this on one line?
city = hashmap.get(cities, 'TX', 'Does Not Exist')
print "The city for the state 'TX' is: %s" % city
```

You should recognize that this is nearly the same code as we created at the top, except that it's using your new implementation of `hashmap` instead. Go through and confirm you understand how each line is the same operation as the same line in `ex39.py`.

The Code Description

This `hashmap` is nothing more than "a list of buckets, which are a list of slots, which have key/value pairs in them." Take a minute to break that down to see what I mean:

"a list of buckets..."
: In the `hashmap` function I create the `aMap` variable which is a list, and then I fill it with other lists...

"which are a list of slots..."
: These bucket lists are empty at first, but as we add key/value pairs to the data structure they will fill with "slots" or...

"which have key/value pairs in them."
: Meaning each slot inside a bucket contains a `(key, value)` tuple or "pair".

If this structure still doesn't make sense, take the time to draw it out on paper until you're sure you get it. In fact, doing algorithms manually on paper is a good way to make sure you understand them.

Now that you know how the data is structured, you just need to know the *algorithm* for each operation. An algorithm is the steps you take to do something to or with a data structure. It's the code that makes the data structure work. We'll go through each of these operations using the code, but a general pattern in the `hashmap` algorithms is this:

> 1. Convert a key to an integer using a hashing function: `hash_key`.
> 2. Convert this hash to a bucket number using a `%` (modulus) operator.
> 3. Get this bucket from the `aMap` list of buckets, and then traverse it to find the slot that contains the key we want.

In the case of `set` we do this, to replace duplicate keys, or we append to add new ones.

I will now walk through the code for the hashmap so you can understand what's going on, following function by function. Follow along and make sure you understand every line. Write an English comment above each line to make sure you understand what it's doing. This is so deceptively simple that I recommend you take the time to play with any lines of code mentioned in the following description either in the Python shell, or on paper until you get it.

new
: First I start by creating a function that makes a hashmap for you, also known as an initializer. What I've done is created an `aMap` variable that has a list, and then I put `num_buckets` lists inside it. These buckets will be used to hold the contents of the hashmap as I set them. Later I use `len(aMap)` in other functions to find out how many buckets there are. Be sure you understand that!

hash_key
: This deceptively simple function is the core of how a `dict` works. What it does is uses the built-in Python `hash` function to convert a string to a number. Python uses this function for its own `dict` data structure, and I'm just reusing it. You should fire up a Python console to see how it works. Once I have a number for the key, I then use the `%` (modulus) operator and the `len(aMap)` to get a bucket where this key can go. As you should know, the `%` (modulus) operator will divide any number and give me the remainder. I can also use this as a way of limiting giant numbers to a fixed smaller set of other numbers. If you don't get this then use Python to explore it.

get_bucket
: This function then uses `hash_key` to find the bucket that a key *could* be in. Since I did `% len(aMap)` in the `hash_key` function, I know whatever `bucket_id` I get will fit into the `aMap` list. Using the `bucket_id` I can then get the bucket where the key *could* be.

get_slot
: This function uses `get_bucket` to get the bucket a key could be in, and then it simply rolls through every element of that bucket until it finds a matching key. Once it does it returns the tuple of `(i, k, v)` which is the index the key was found in, the key itself, and the value set for that key. You now know enough to see how this data structure works. It takes keys, hashes and modulus them to find a bucket, then searches that bucket to find the item. This effectively cuts the amount of searching necessary to a fraction of what it would be normally.

get
: This is a "convenience function" which does what most people want a `hashmap` to do. It uses `get_slot` to get the `(i, k, v)` and then just returns the `v` (value) only. Make sure you understand how the `default` variable works, and how the `(i, k, v)` in `get_slot` is assigned to the `i, k, v` variables in `get`.

set
: To set a key/value pair all I need to do is get the bucket, and append the new `(key, value)` to it so it can be found later. However, I want my `hashmap` to allow one key at a time. To do that, first I have to find the bucket then check if this key already exists. If it does then I replace it in the bucket with the new one, and if it doesn't then I append. This is actually slower than simply appending, but more likely what a user of `hashmap` wants. If you wanted to allow multiple values for a key you could simply have `get` go through every slot in the bucket and return a list of everything it found. This is a good example of tradeoffs in design. The current version is faster on `get`, but slower on `set`.

delete
: To delete a key, I simply get the bucket and search for the key in it, and delete it from the list. However, because I chose to make `set`

only store one key/value pair I can stop when I have found one. If I had decided to allow multiple values for each key by simply appending I would have also made `delete` slower because I would have needed to go through *every* slot on delete just in case it had a key/value pair that matched.

list
: The last function is simply a little debug function that prints out what's in the `hashmap` and should be trivial for you to understand. It just gets each bucket, then goes through each slot in the bucket.

After all of these functions I just have a little bit of testing code that makes sure they work.

Three Levels of Lists

As I mentioned in the discussion, by deciding that `set` will overwrite (replace) keys with new values, I have made `set` slower but this assumption makes all of the other functions faster. What if I wanted a `hashmap` that allowed multiple values for each key but still keep everything fast?

What I need to do then is establish that every slot in a bucket is a list of values. This means that I add a third level of buckets, then slots, then values. This also matches the description of this kind of `hashmap` because I am saying, "For every key there can be *multiple* values." Another way to say that is "Every key has a list of values." Since keys go in slots, then slots have lists of values and I'm done.

If you want to take this code further, then change it to support multiple values for each key.

What You Should See (Again)

```
$ python ex39_test.py
----------
NY State has: New York
OR State has: Portland
----------
Michigan's abbreviation is: MI
Florida's abbreviation is: FL
----------
Michigan has: Detroit
Florida has: Jacksonville
----------
California CA
New York NY
Michigan MI
Oregon OR
Florida FL
----------
FL Jacksonville
NY New York
CA San Francisco
OR Portland
MI Detroit
----------
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist
```

When to Use Dictionaries or Lists

As I mentioned in Exercise 38, lists have specific properties that help you contain and organize things that need to be in a list structure. Dictionaries are the same, but the properties of a `dict` are different than lists because they work with mapping keys to values. That means you use a dictionary when:

> 1. You have to retrieve things based on some identifier, like names, addresses, or anything that can be a key.
> 2. You don't need things to be in order. Dictionaries do not normally have any notion of order, so you have to use a list for that.
> 3. You are going to be adding and removing elements and their keys.

This means that if you have to use a non-numeric key, use a `dict`. If you need things in order, use a list.

Study Drills

> 1. Do this same kind of mapping with cities and states/regions in your country or some other country.
> 2. Find the Python documentation for dictionaries and try to do even more things to them.
> 3. Find out what you *can't* do with dictionaries. A big one is that they do not have order, so try playing with that.
> 4. Read about Python's *assert* feature and then take the *hashmap* code and add assertions for each of the tests I've done instead of *print*. For example, you can assert that the first get operation returns *"New York"* instead of just printing that out.
> 5. Did you notice that the `list` function listed the items I added in a *different* order than they were added? This is an example of how dictionaries don't maintain order, and if you analyze the code you'll understand why.
> 6. Create a `dump` function that is like `list` but which dumps the full contents of every bucket so you can debug it.
> 7. Make sure you know what the `hash` function does in that code. It's a special function that converts strings to a consistent integer. Find the documentation for it online. Read about what a "hash function" is in programming.

Common Student Questions

Q: What is the difference between a list and a dictionary?

> A list is for an ordered list of items. A dictionary (or `dict`) is for matching some items (called "keys") to other items (called "values").

Q: What would I use a dictionary for?

> When you have to take one value and "look up" another value. In fact you could call dictionaries "look up tables."

Q: What would I use a list for?

> Use a list for any sequence of things that need to be in order, and you only need to look them up by a numeric index.

Q: What if I need a dictionary, but I need it to be in order?

Take a look at the `collections.OrderedDict` data structure in Python. Search for it online to find the documentation.

Video

Purchase The Videos For $29.59

For just $29.59 you can get access to all the videos for Learn Python The Hard Way, **plus** a PDF of the book and no more popups all in this one location. For $29.59 you get:

- All 52 videos, 1 per exercise, almost 2G of video.
- A PDF of the book.
- Email help from the author.
- See a list of everything you get before you buy.

When you buy the videos they will immediately show up **right here** without any hassles.

Already Paid? Reactivate Your Purchase Right Now!

Buying Is Easy

Buying is easy. Just fill out the form below and we'll get started.

Full Name
Full Name
Email Address
Email Address

[✓ Pay With Credit Card (by Stripe™)] Use your PayPal™ account.

Buy Learn Python The Hard Way, 3rd Edition