Forward School

## Program Code: J620-002-4:2020

## Program Name: FRONT-END SOFTWARE DEVELOPMENT

## Title : Data Structures II (Dictionary)

**Name: Justin Chong**

**IC Number: 960327-07-5097** ¶

**Date : 20/6/2023**

**Introduction : Learning about Python dictionary data structure.**

**Conclusion : Able to understand dictionary and how to use it.**

# Module P04: Data Structures II (Dictionary)

In [1]:
```
def find(num, li):
    for val in li:
        if num == val:
            return True
    return False
```

In [4]:
```
%timeit find('ab', list(range(10000000)))
```

```
932 ms ± 46.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

### Intuition: Hash Tables

The reason `find(element, li)` was slow is that we were doing this for loop by going through all the elements in order, and we're checking if they match the keyword. To determine that a keyword not in the list is not there, we had to go through the whole index.

That's slow. What might be a better way to go through a list?

You can sort it - it'll be like an index? But that takes time and effort too and isn't necessary!

What we want is something that will allow us, given a keyword, to have some function that tells us where it belongs. That's called a hash function.

The **Hash function** tells us where in the entry to look. Instead of looking through the whole index, you can just look at where it belongs.

One easy way is using the first letter of a word. But there are problems with this:

- It works best if you have roughly equal buckets of words.
- If you have a million keywords, it only makes things 26 times faster at best

Let's use some function on the whole 'key' or 'keyword' that will be able to tell us where it belongs.

A hash function is a function that takes in a keyword, produces a number. It outputs the position in the hash table which is the bucket where that keyword would appear. If we assume that there are b buckets, k keywords, which properties should a hash function have?

- Output a unique number between 0 and k-1
- Output a number between 0 and b-1
- Map approximately k/b keywords to bucket 0
- Map approximately k/b keywords to bucker b-1
- Map more keywords to bucket 0 than to bucket 1

## Dictionary

Fortunately, Python has a built in hash table structure.

Python's efficient key/value hash table structure is called a "dict".

A dictionary is similar to a list, but you access values by looking up a key instead of an index. A key can be any string or number.

The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }.

The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. dict['foo'] looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys.

Dictionaries are mutable!

In [5]:
```python
# create an empty dictionary
newDict ={}
```

In [7]:
```python
myDict = {}

# key-value pairs: e.g. 'a' (key), 'gold' (value)
myDict['a'] = 'gold'
myDict['b'] = 'silver'
myDict['c'] = 'bronze'
myDict['d'] = 'platinum'

print(myDict)
```

```
{'a': 'gold', 'b': 'silver', 'c': 'bronze', 'd': 'platinum'}
```

In [9]:
```
# Looping through all entries in the dictionary. easy!
for v in myDict.items():
    print(v)
```
```
('a', 'gold')
('b', 'silver')
('c', 'bronze')
('d', 'platinum')
```

In [10]:
```
myDict = {}

myDict['a'] = 'gold'
myDict['b'] = 'silver'
myDict['c'] = 'bronze'
myDict['d'] = 'platinum'

print(myDict)
print((myDict['b']))

# checking if a value exists in a dictionary involves searching thru the keys, not values
print(('a' in myDict))
print(('gold' in myDict))
```
```
{'a': 'gold', 'b': 'silver', 'c': 'bronze', 'd': 'platinum'}
silver
True
False
```

In [16]:
```
# to list all keys
s = list(myDict.keys())

# to list all values
print(myDict.values())
print(s)
print((type(s)))

# .items() returns a list of key-value tuples
print(('\nitems:', list(myDict.items()), '\n'))

# print key-value tuples using a loop
for key in myDict:
    print((key, myDict[key]))
```
```
dict_values(['gold', 'silver', 'bronze', 'platinum'])
['a', 'b', 'c', 'd']
<class 'list'>
('\nitems:', [('a', 'gold'), ('b', 'silver'), ('c', 'bronze'), ('d', 'platinum')],
'\n')
('a', 'gold')
('b', 'silver')
('c', 'bronze')
('d', 'platinum')
```

In [10]:
```
# Assigning a dictionary with three key-value pairs, two entries have the same key...
residents = {'Puffin' : 104, 'Puffin' : 105, 'Burmese Python' : 106}

print(residents.keys())
print(residents['Puffin'])      # what's the value?
print(len(residents))           # how many entries?
```

```
dict_keys(['Puffin', 'Burmese Python'])
105
2
```

In [11]:
```
print('key, value pairs using keys')

for i in myDict.keys():
    print((i, myDict[i]))    # myDict[i] <- isn't this how we access values in dictionar

print('\nkey, value pairs using items()')

# a list of tuples can be assigned respectively to two loop iterators k and v
for k, v in list(myDict.items()):
    print((k, v))

print()
```

```
key, value pairs using keys
('a', 'gold')
('b', 'silver')
('c', 'bronze')
('d', 'platinum')

key, value pairs using items()
('a', 'gold')
('b', 'silver')
('c', 'bronze')
('d', 'platinum')
```

In [12]:
```
# if we want to do an update on the prev dictionary based on the new dictionary.....
another_d = {'e':'looser', 'a':'SUPER_GOLD'}
myDict.update(another_d)
```

In [13]:
```
print(another_d)
myDict
```

```
{'e': 'looser', 'a': 'SUPER_GOLD'}
```

Out[13]:
```
{'a': 'SUPER_GOLD',
 'b': 'silver',
 'c': 'bronze',
 'd': 'platinum',
 'e': 'looser'}
```

**Quick Exercise 1** Construct a new dictionary consisting of the following data and then proceed to update their allowances by giving an increment of 8%. Add some code to print the contents of the dictionary just to be sure it is correct

*(Expected answer - Debbie 972.00, Ishak 918.00, Rajah 1242.00, Siti 999.00)*

| | |
|---|---|
| Debbie | 900 |
| Ishak | 850 |
| Rajah | 1150 |

In [9]:
```
# write your code here

ledger = {'Debbie': '900', 'Ishak': '850', 'Rajah': '1150', 'Siti': '925'}
newLedger = {}
for name, allowance in ledger.items():
    updatedAllowance = round(int(allowance) * 1.08, 2)
    newLedger[name] = updatedAllowance

print(newLedger)
```

```
{'Debbie': 972.0, 'Ishak': 918.0, 'Rajah': 1242.0, 'Siti': 999.0}
```

**Quick Exercise 2** Write the allowance updating again using **dictionary comprehension**, which is basically done in a single line.

Note: Dictionary comprehension is similar to list comprehension, except for 2 big differences: We replace the list's square brackets with dict's curly braces, and the key and value pairs should be operated on together, separated by a colon.

In [10]:
```
# Write your code here

updatedLedger = {name: round(int(allowance) * 1.08, 2) for name, allowance in list(ledge
print(updatedLedger)
```

```
{'Debbie': 972.0, 'Ishak': 918.0, 'Rajah': 1242.0, 'Siti': 999.0}
```

## Dictionary Methods

Here's a whole bunch of dictionary methods that could be useful to you!

- len(dict) -- Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
- str(dict) -- Produces a printable string representation of a dictionary
- type(variable) -- Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.
- dict.clear() -- Removes all elements of dictionary dict
- dict.copy() -- Returns a shallow copy of dictionary dict
- dict.fromkeys() -- Create a new dictionary with keys from seq and values set to value.
- dict.get(key, default=None) -- For key key, returns value or default if key not in dictionary
- dict.items() -- Returns a iterator of dict's (key, value) tuple pairs
- dict.keys() -- Returns iterator of dictionary dict's keys
- dict.setdefault(key, default=None) -- Similar to get(), but will set dict[key]=default if key is not already in dict
- dict.update(dict2) -- Adds dictionary dict2's key-values pairs to dict
- dict.values() -- Returns list of dictionary dict's values

What about sorting? Can we sort values or keys in a dictionary?

In [12]:
```
for v in sorted(myDict.values()):   # sorted. what does it do?
    print(v)

print()

for key in sorted(myDict.keys()):
    print((key, myDict[key]))
```

```
bronze
gold
platinum
silver

('a', 'gold')
('b', 'silver')
('c', 'bronze')
('d', 'platinum')
```

In Python, we can import packages when coding to give us additional functionalities. For instance, the `time` and `datetime` packages can give us the current time reading:

In [11]:
```
import time
time.time()     # what does this number represent?
```

Out[11]:
```
1687748560.8085773
```

In [12]:
```
import datetime
datetime.datetime.now()     # this looks more familiar :)
```

Out[12]:
```
datetime.datetime(2023, 6, 26, 11, 2, 41, 798564)
```

Let's do a bit of speed benchmarking here, comparing the performance of a BIG list and dictionary. We first create 30 million integers and put them into both:

In [15]:
```
biglist = []
bigdict = {}
for j in range(30000000):
    biglist.append(j)     # recall: this is how you add to a list
    bigdict[j]=j          # this is how you add to a dictionary
```

Let's go by the worst-case scenario, and search for the "last" number in the data structure. "Last" here meaning, we assume it's going to be something we expect the data structure to find at the "end" of the container. To benchmark this, note down the start and end times and find the duration.

In [16]:
```
t1 = time.time()
list_tf = 29999999 in biglist
t2 = time.time()
dict_tf = 29999999 in bigdict
t3 = time.time()

print(("Time for LIST lookup: ", t2-t1))     # in most decent machines, this would be ~0
print(("Time for DICT lookup: ", t3-t2))
```

```
('Time for LIST lookup: ', 0.5270066261291504)
('Time for DICT lookup: ', 0.0)
```

In [17]:
```
(t2-t1)/(t3-t2)       # ugh....

# expected output is 205.18954528838665 but...
# because dictionary computation time is close to 0, so anything divided by zero is infi
# meaning a zero division error
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[17], line 1
----> 1 (t2-t1)/(t3-t2)

ZeroDivisionError: float division by zero
```

Due to the problem above, we can't possibly find out how many times are dictionaries faster than lists....

So let's use the Jupyter magic function `%timeit` which runs an operation in N number of loops for T number of times, then takes the best of the T rounds. It reports back how much time was consumed per loop.

In [24]:
```
%timeit 9999999 in biglist
```

```
141 ms ± 12.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [25]:
```
%timeit 9999999 in bigdict
```

```
99.6 ns ± 6.02 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

In [26]:
```
# do a bit of calculation here to determine how many times faster are dictionaries...
# replace the values accordingly
dict_faster_than_list = (130*10**-6) / (63.7*10**-9)
print(dict_faster_than_list)
```

```
2040.8163265306118
```

Did you get a speed-up of ~2000 times ?

Another example below:

In [23]:
```
emails = {'bob': 'bob@example.com',
          'Jane': 'jane@example.com',
          'Stou': 'stou@example.net'}

newdict = {}
for name, email in emails.items():
    if '.com' in email:
        newdict[name]=email
```

In [24]:
```
# new dictionary contains only those with emails ending with .com
newdict
```

Out[24]:
```
{'bob': 'bob@example.com', 'Jane': 'jane@example.com'}
```

In [25]:
```
# dict comprehension to check if the email contains .com domain
email_at_dotcom = {name :'.com' in email for name, email in emails.items()}
```

In [26]:
```
email_at_dotcom
```

Out[26]:
```
{'bob': True, 'Jane': True, 'Stou': False}
```

In [27]:
```
print(len(emails))    # length of dictionary
print(str(emails))    # conversion of dict to string...
print(type(emails))

# making a copy of a dictionary
new_emails = emails.copy()
print("NEW_EMAILS", new_emails)

# clear the contents of the dictionary
new_emails.clear()
print(new_emails)
```

```
3
{'bob': 'bob@example.com', 'Jane': 'jane@example.com', 'Stou': 'stou@example.net'}
<class 'dict'>
NEW_EMAILS {'bob': 'bob@example.com', 'Jane': 'jane@example.com', 'Stou': 'stou@exampl
e.net'}
{}
```

Some other interesting functionalities:

In [28]:
```
new_emails = {'jane': 'jane@example.com',
              'Julie': 'julie@example.com',
              'Stam': 'stam@example.net'}
seq = ('Jane', 'Julie')
print((type(seq)))

# create a new dictionary using keys specified in a container, fixed to a value
other = new_emails.fromkeys(seq, 'info@example.com')
print(other)

# get: retrieve the value given a key
print((emails.get('bob')))
print((emails.get('Micheal')))
```

```
<class 'tuple'>
{'Jane': 'info@example.com', 'Julie': 'info@example.com'}
bob@example.com
None
```

**TASK** Compute the price of this cart. By default everything cost RM 1, except when prices are written

In [18]:

```
# By default everything is RM 1

cart = {'apple':12,
       'banana': 1,
       'orange': 2,
       'pear':10}

prices = {'mango':12,
         'apple':2.5,
         'orange':5.5,
         'banana':None}

# write your code from here on
# TIP: Use the keys from 'cart' to access the price in 'prices'
# HINT: put dict.get() method to good use

total_price = 0
for item, quantity in cart.items():
    if (prices.get(item) == None):
        prices[item] = 1
    total_price += (prices.get(item) * quantity)
print(f'RM {total_price}')
```

```
RM 52.0
```