Lecture 18: Dictionaries

CS 51P

November 7, 2022

Tom Yeh
he/him/his

What are Dictionaries?

- Data structure which associates a key with a value (key:value pairs)
  - Key is a unique identifier
  - Value is something we associate with the key

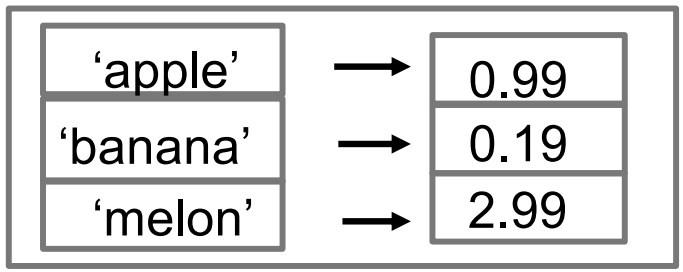
- Real world examples:
  - Phonebook - Keys: names Values: phone numbers
  - Dictionary
  - Pricelist

Dictionaries in Python

- Creating dictionaries
  - Dictionary start/end with braces
  - Key:Value pairs separated by colon
  - Each pair is separated by a comma

  - `price = {'apple': .99, 'banana': .19, 'melon': 2.99}`
  - phone = { 'Sage' : '555-1111', 'Hen' : '666-2222' }
  - empty_dict = { }

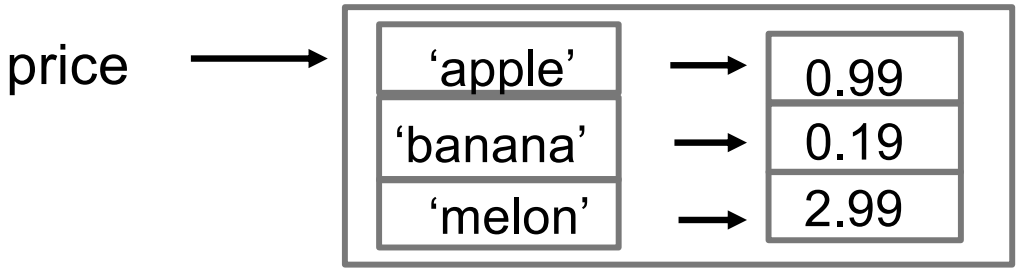
  - price



Accessing Elements of Dictionary

- Given this dictionary:
- `price = {'apple': .99, 'banana': .19, ‘melon': 2.99}`

- Like a set of variables that are indexed by keys

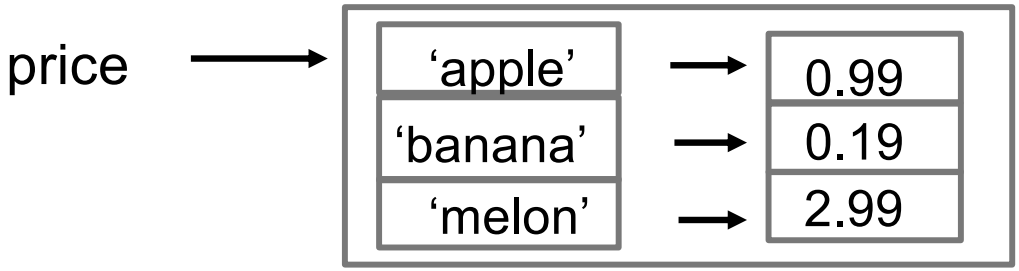


- Use key to access associated value:
  - price[‘apple’] is 0.99
  - price[‘melon’] is 2.99

Accessing Elements of Dictionary

- Given this dictionary:
- `price = {'apple': .99, 'banana': .19, 'melon': 2.99}`

- Like a set of variables that are indexed by keys

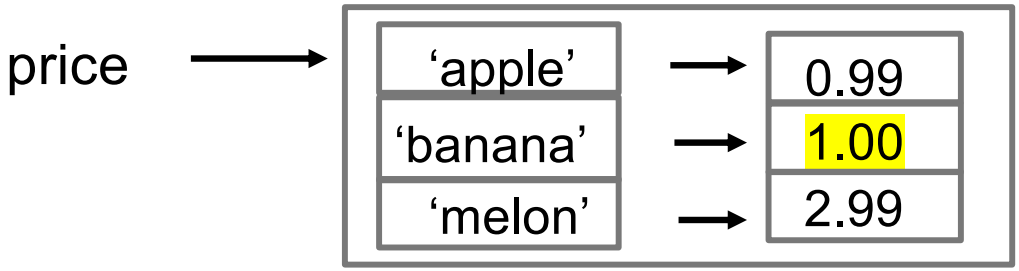


- Use key to access associated value:
  - price['apple'] is 0.99
  - price['melon'] is 2.99
- Can set values like regular variables
  - price['banana'] = 1.00

Accessing Elements of Dictionary

- Given this dictionary:
- `price = {'apple': .99, 'banana': .19, ‘melon': 2.99}`

- Like a set of variables that are indexed by keys

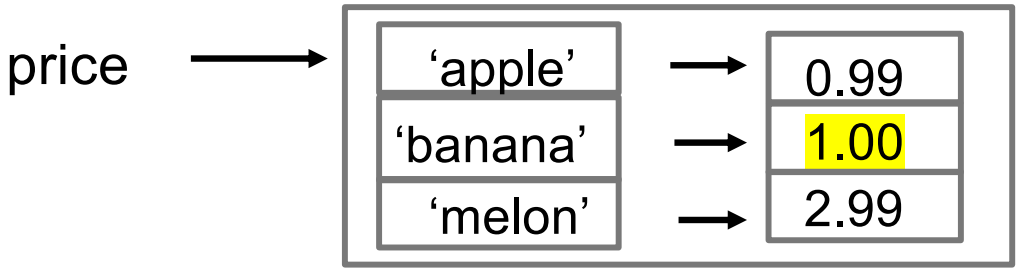


- Accessing pairs:
  - banana_price = price[‘banana’]
  - banana_price is 1.00
  - grape_price = price[‘grape’] # What does this line do?

Accessing Elements of Dictionary

- Given this dictionary:
- `price = {'apple': .99, 'banana': .19, ‘melon': 2.99}`

- Like a set of variables that are indexed by keys

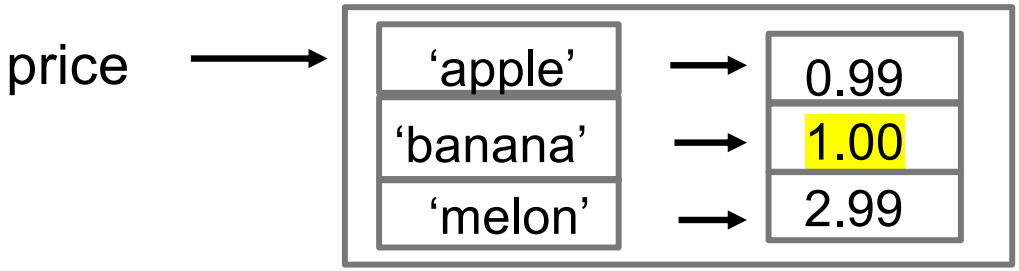


- Accessing pairs:
  - banana_price = price[‘banana’]
  - banana_price is 1.00
  - grape_price = price[‘grape’] # What does this line do?
  - KeyError: ‘grape’

Accessing Elements of Dictionary

- Given this dictionary:
- `price = {'apple': .99, 'banana': .19, 'melon': 2.99}`

- Like a set of variables that are indexed by keys



- Checking membership
  - >>> 'apple' in price
  - True
  - >>> 'grape' in price
  - False

Adding Elements to Dictionary

- Adding pairs to a dictionary:
  - price = { }

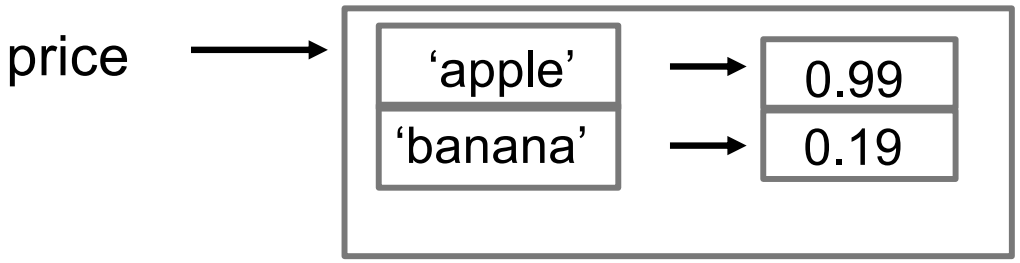
Adding Elements to Dictionary

- Adding pairs to a dictionary:
  - price = { }

  - price[‘apple’] = 0.99

Adding Elements to Dictionary

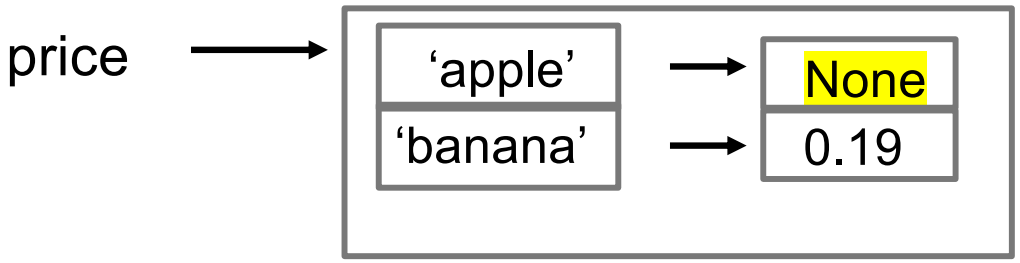
- Adding pairs to a dictionary:
  - price = { }



  - price[‘apple’] = 0.99
  - price[‘banana’] = 0.19

Adding Elements to Dictionary

- Adding pairs to a dictionary:
  - price = { }



  - price['apple'] = 0.99
  - price['banana'] = 0.19
  - price['apple'] = None

Key/Values

- Keys must be immutable types
  - E.g. int, float, string
  - Keys cannot be changed in place
- Values can be mutable or immutable types
  - E.g. int, float, string, lists, dictionaries
  - Values can be changed in place
- Dictionaries are mutable
  - Changes made to a dictionary in a function persist after function completes

Changing Dictionary in a Function

- def inflation(dict, item):
  - dict[item] *= 5

- def main():
  - `price = {'apple': .99, 'banana': .19, 'melon': 2.99}`
  - `inflation(price,'apple')`
  - `print(price)`

- `Terminal:`
  - `{'apple' : 4.95. 'banana': .19, 'melon': 2.99}`

Dictionary Operations

- `get: dict.get(key)`
  - `Returns value associated with key`
  - `Returns None if key doesn't exist`
- `get: dict.get(key, default)`
  - `Returns default if key doesn't exist`
- `keys: dict.keys()`
  - `Returns a "range" of keys`
  - `Can loop over all keys`
  - `for key in dict.keys():`
    - `print(dict[key])`

More Dictionary Operations

- Can also loop over dictionary like this:
  - for key in dict:
    - print(key)
- Values: dict.values()
  - Returns a “range” of the values in dictionary
  - Can use to loop over all values in the dictionary
    - for value in dict.values():
      - print(value)
- Can turn keys() or values() into a list
  - list(dict.keys())
  - list(dict.values())

Dictionary Operations

- `add: d["apple"] = .99`
- `update: d["apple"] = 1.05`
- `delete: del(d["apple"])`
- `pop: x = d.pop("apple")`
  - removes entry with that key, returns value
- `d.keys(), d.values(), d.items()`
  - returns list of all the keys, values, (key, value) tuples
- clear: d.clear()
  - Removes all key/value pairs in the dictionary
- len(dict) – returns number of key/value pairs

Dictionary Operations

adding to a dictionary

- a_dict[key] = value
- a_dict.update(*b_dict*)

removing from a dictionary

- del (a_dict[*key*])
- a_dict.pop(*key*)
  - returns a_dict[key]

other

- len(a_dict)
- a_dict.get(key)
  - returns associated value
- a_dict.keys()
  - returns list
- a_dict.values()
  - returns list
- a_dict.items()
  - returns list of tuples
- b_dict = a_dict.copy()
  - shallow copy!

Lists, dictionaries

- Both data structures.
- Why would you choose one over the other?
  - a data structure is something that holds a collection of data and that supports certain operations for working with that data

- Lists: sequential access
  - Access with index
- Dictionaries: fast lookup
  - Access with key

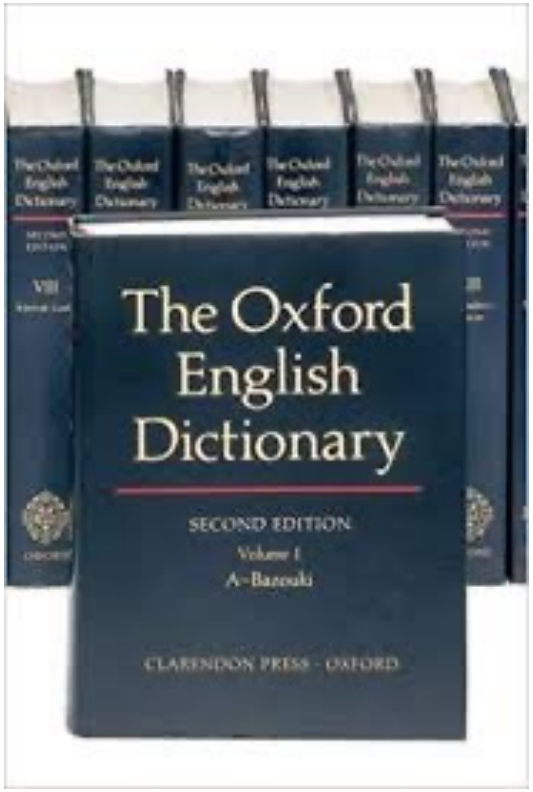
Example

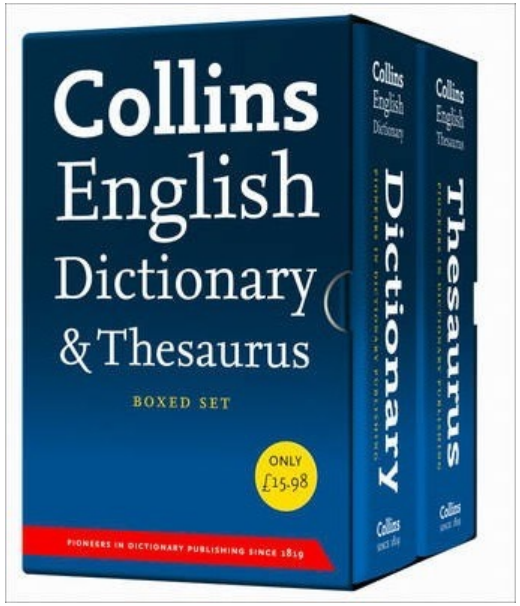
```
>>> d_cost = {}
>>> d_cost["apple"] = .99
>>> d_cost["banana"] = .19
>>> d_cost[”melon"] = 2.99
>>> d_cost["durian"] = 5.99
>>> v = d_cost.pop("banana")
>>> print(d_cost.values())
>>> print(d_cost.items())
```

- Write a function compute_cost(prices, shopping_list) which returns the total price of buying the items from shopping_list (a list of (string, int) pairs).
- (name, number)

Exercise: Merge

- Write a function merge(d1,d2) that merges two dictionaries and returns a single dictionary containing all the key:value pairs from both input dictionaries.



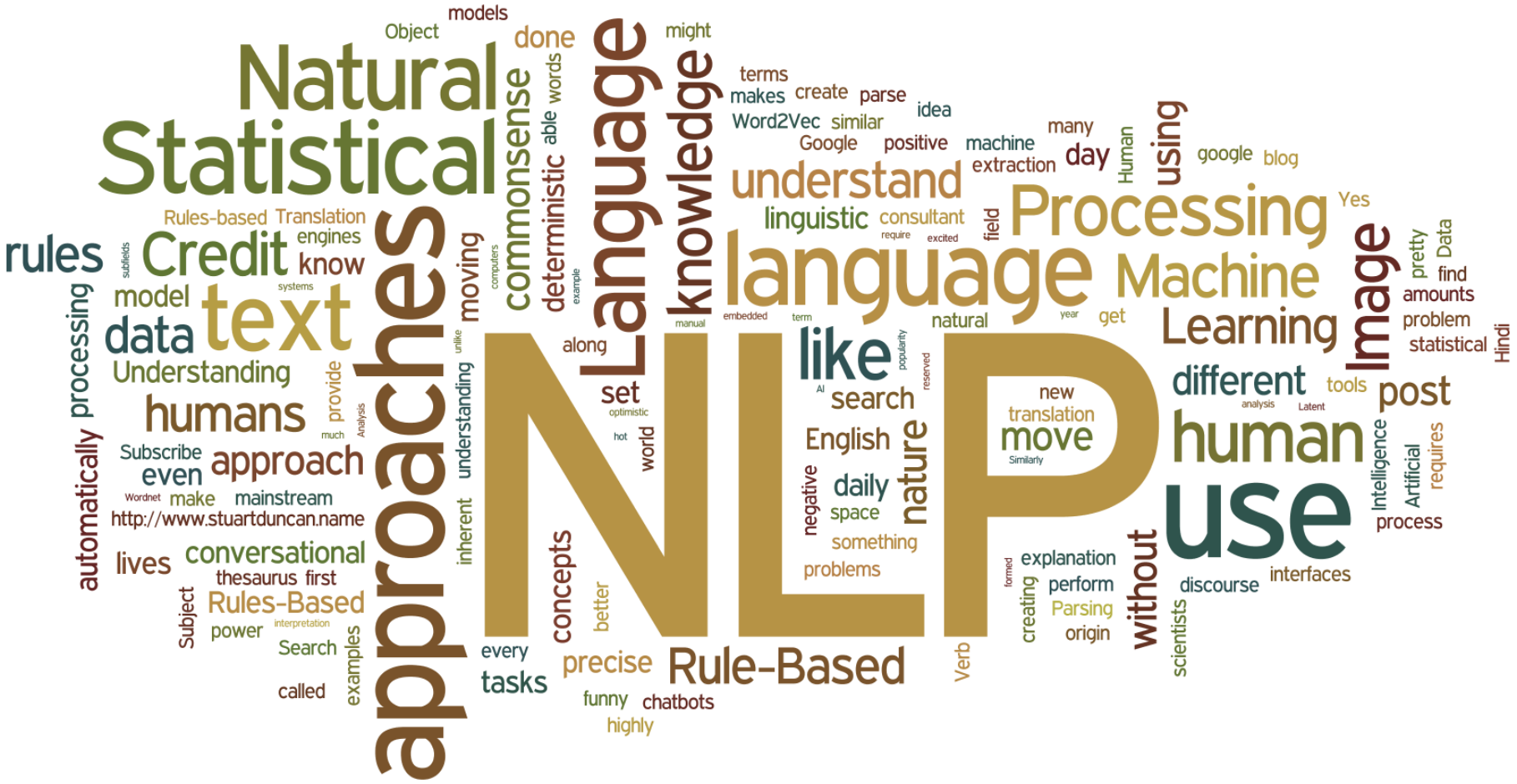


Loops and Dictionaries

- Write a function first_last(gs) that returns a pair containing the (alphabetically) first and last items available for sale in a grocery store gs (a dictionary).

- Write a function under_price(gs, price) that takes a dictionary as input returns the number of unique items available for less than (or equal to) price p

Example: Word Count



- Write a function that processes a file and returns a dictionary for handling repeated queries of the form "How many times does the word _____ appear?"

Dictionary Operations

File Functions

- f = fopen(fname, mode)
  - open file named fname in mode
- f.close
  - close f
- f.readlines()
  - returns list

String Functions

- s.split()
- s.strip(string.punctuation)
  - removes punctuation

Dictionary Functions

- a_dict[key] = value
  - set, update
- len(a_dict)
- a_dict.keys()
  - returns list
- a_dict.values()
  - returns list
- a_dict.items()
  - returns list of tuples
- b_dict = a_dict.copy()
  - shallow copy!

Example: Word Count

- Write a function that processes a file and returns a dictionary for handling repeated queries of the form "How many times does the word _____ appear?"

```
import string
word_counts = {}
f = fopen("file.txt", "r")
text = f.readlines()
for line in text:
      words = line.split()
      for w in words:
            w2 = w.strip(string.punctuation)
            if w2 in word_counts:
                  word_counts[w2] = word_counts[w2] + 1
            else:
                  word_counts[w2] = 1
```

```
def mystery(my_dict):
    d = {}
    for i in my_dict.keys():
        if my_dict[i] in d:
            d[my_dict[i]].append(i)
        else:
            d[my_dict[i]] = [i]
    return d

def main():
    d = {"a":1, "b":2, "c":1, "d":0, "e":2}
    print(mystery(d))

main()
```

Tuple – another built-in data type

- a tuple is a way to keep track of an *ordered collection* of items
  - Similar to a list, but **immutable** (can not be changed in place)
  - **Ordered**: can refer to elements by their position (start with 0)
  - **Collection**: tuple can contain multiple items

```
num_tuple = (1, 2, 3)
```

- Often used to track data that are related:
  - Coordinates for a point: (x, y)
  - RGB values for a color: (red, green, blue)

- Can be used to return multiple values from a function
  - More on this later

Creating Tuples

- Creating tuples
  - Tuples start/end with parenthesis with elements separated by commas.

```
random_tuple = (3, 6, 2, 1)
point = (5.1, 6.2)
addr = ('333 N College Way', 'Claremont', 'CA 91711')
empty_tuple = ()
```

- Tuple with 1 element **is** the same as the element
  - >>> tuple_one = (51)
  - >>> one = 51
  - >>> tuple_one == one
  - True

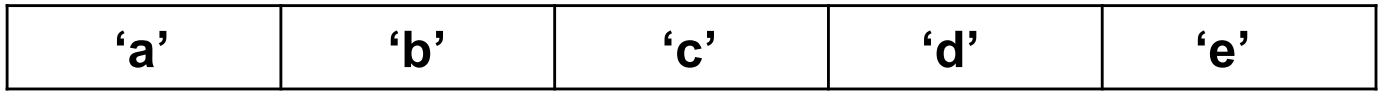
Accessing Elements of a Tuple

- Consider this tuple:

```
a_tuple = ('a', 'b', 'c', 'd', 'e')
```

- Access elements of tuple just like the list
  - Index starts from 0

- a_tuple



- Can **not** assign to individual elements:
  - Tuples are immutable
  - No append/pop functions

- To change a tuple, we need to create new tuple and overwrite variable
  - a_tuple = a_tuple[0:2]

Similar to lists

- Same for
  - Indexing
  - slicing
  - checking for empty tuple
  - checking if tuple contains an element
  - same ways with for loop to iterate through tuples
- Few functions
  - Min, max, sum

Assignment with tuples

- Can use tuples to assign multiple variables at the same time
  - Number of variables on left hand side needs to be the same as the right hand side

  - >>> (x, y) = (5, 1)
  - >>> x
  - 5
  - >>> y
  - 1

Tuples and List

- Can create tuple from list
- >>> num_tuple = (1, 2, 3, 4, 5)
- >>> num_list = list(num_tuple)
- >>> num_list
- [1, 2, 3, 4, 5]

- Can create list from tuple
- >>> a_list = [’Red’ , ‘Green’ , ‘Blue’]
- >>> a_tuple = tuple(a_list)
- >>> a_tuple
- (‘Red’ , ‘Green’ , ‘Blue’)

Why Tuples?

- More restrictive because it is immutable
- Tuples are more memory efficient than lists
- Execution speed of using tuples is faster than using lists