Tuples, Dictionaries and Sets

HORT 530

Lecture 11

Instructor: Kranthi Varala

Core data types

- Numbers
- Strings
- Lists
- Tuples
- Dictionaries
- Files
- Sets

Tuples

- Tuples are immutable general sequence objects that allows the individual items to be of different types.
- Equivalent to lists, except that they can't be changed.

```
>>> myTuple=('0','a','bob',4.89)
>>> myTuple
('0', 'a', 'bob', 4.889999999999997)
>>> myTuple[1]
'a'
>>> myTuple[1:3]
('a', 'bob')
```

```
>>> geneT=('Chr1','TAIR10','gene',3631,5899,'.','+','.','ID=AT1G01010')
>>> geneT
('Chr1', 'TAIR10', 'gene', 3631, 5899, '.', '+', '.', 'ID=AT1G01010')
```

Tuples

- Tuple.count(value) : Returns number of occurrences of value.
- Tuple.index(value,[start,stop]) : Returns first index of value.
- Typically used to maintain data integrity within the program.

```
>>> myTuple= (0,)
>>> myTuple
(0,)
>>> myTuple= 0,
>>> myTuple
(0,)
```

Even single elements need a comma

Parentheses () are optional

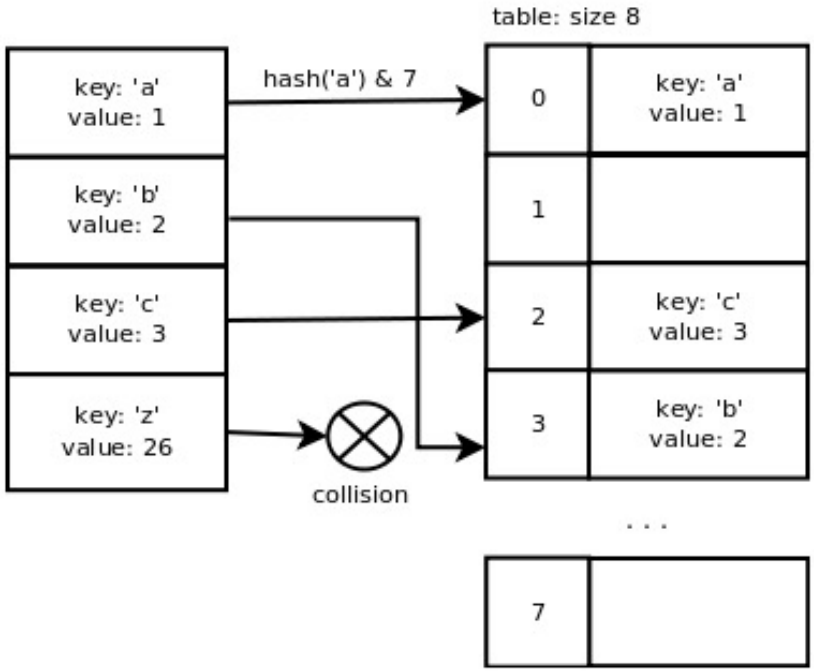
Dictionaries

- Dictionaries are unordered collections of objects, optimized for quick searching.
- Instead of an index, objects are identified by their ‘key’.
- Each item within a dictionary is a ‘key’:’value’ pair.
- Equivalent to hashes or associative arrays in other languages.
- Like lists and tuples, they can be variable-length, heterogeneous and of arbitrary depth.
- ’Keys’ are mapped to memory locations by a hash function

Hash function

- A hash function converts a given key value into a ‘slot’ where its value will be stored.
- A hash function always takes a fixed amount of time and always returns the same slot for the same ‘key’.
- When program searches for a ‘key’ in the dictionary, the slot it should be in is calculated and the value in it, if any, is returned.
- Creating hashes is expensive, searching is cheap.

Hash collisions



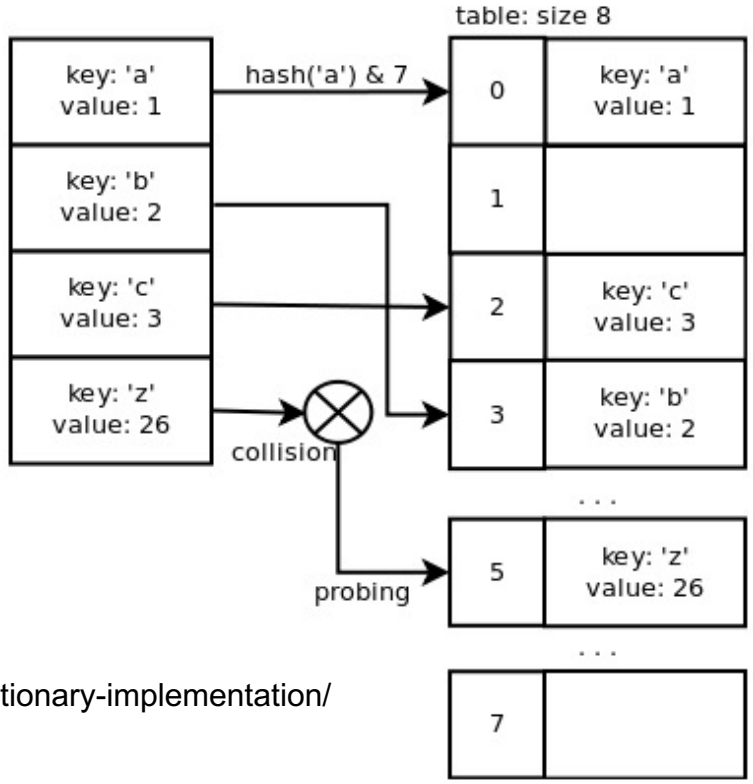


Image credit: http://www.laurentluce.com/posts/python-dictionary-implementation/

- When a new key "collides" with an existing key slot, Python uses a probing sequence to look for another close slot that is empty.
- As number of collisions increase this becomes slower and slower.
- When hash table is 2/3 full python creates a new hash table that is n*4.

Dictionaries

```
>>> myDict={}
```

```
>>> myDict={'name':'Bob','surname':'Smith','age':40}
>>> myDict
{'age': 40, 'surname': 'Smith', 'name': 'Bob'}
>>> myDict['name']
'Bob'
```

```
>>> myDict['YOB'] = 1970
>>> myDict
{'age': 40, 'surname': 'Smith', 'name': 'Bob', 'YOB': 1970}
>>> myDict['name'] = 'Robert'
>>> myDict
{'age': 40, 'surname': 'Smith', 'name': 'Robert', 'YOB': 1970}
```

Dictionaries: Common Methods

- D.keys() : List of keys
- D.values() : List of values
- D.clear() : remove all items
- D.update(D2) : Merge key values from D2 into D. NOTE: Overwrites any matching keys in D.
- D.pop(key) : returns the value of given key and removes this key:value pair from dictionary.

Looping over Tuples and Dictionaries

```
>>> T = (1,'a',45,23.45,'String')
>>> for x in T: print x
...
1
a
45
23.45
String
```

```
>>> myDict = {'a':'1','b':'2','c':'3'}
>>> for key in myDict:
...     print(key,' ==> ', myDict[key])
...
('a', ' ==> ', '1')
('c', ' ==> ', '3')
('b', ' ==> ', '2')
```

```
>>> for (key,value) in myDict.items():
...     print(key,' ==> ',value)
...
('a', ' ==> ', '1')
('c', ' ==> ', '3')
('b', ' ==> ', '2')
```

Sets

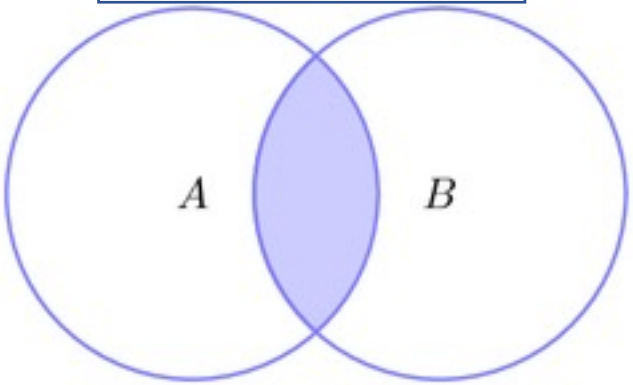
- Unordered collection of ONLY immutable objects. Set itself is mutable.
- Support operations from set theory e.g., union, intersection etc.

```
>>> mySet=set('Example string')
>>> mySet
set(['a', ' ', 'E', 'g', 'i', 'm', 'l', 'n', 'p', 's', 'r', 't', 'x', 'e'])
>>> mySet=set(1,2,3,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set expected at most 1 arguments, got 4
>>> mySet=set([1,2,3,4])
>>> mySet
set([1, 2, 3, 4])
>>> mySet={1,2,3,4}
>>> mySet
set([1, 2, 3, 4])
```

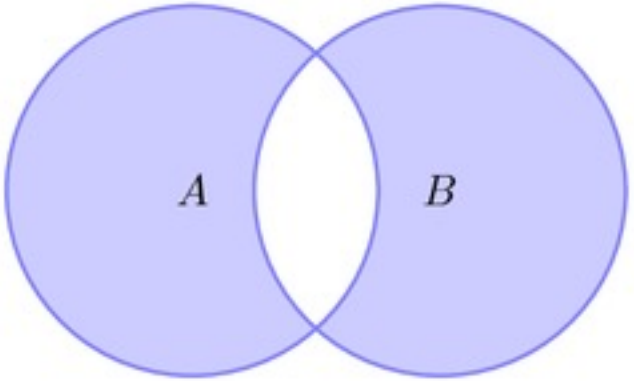
Set operations

- mySet.add(element) #Add element to set if absent
- mySet.update(newSet) #Add new elements from newSet
- mySet.remove(element) #Remove element if present
- mySet.clear() #Empty the set
- len(mySet) #returns size of set
- X in mySet #returns True if x is in mySet
- X not in mySet #returns True if x is not in mySet
- mySet.union(t) #returns new set with elements present in one or both sets
- mySet.intersection(t) #new set with elements in both sets

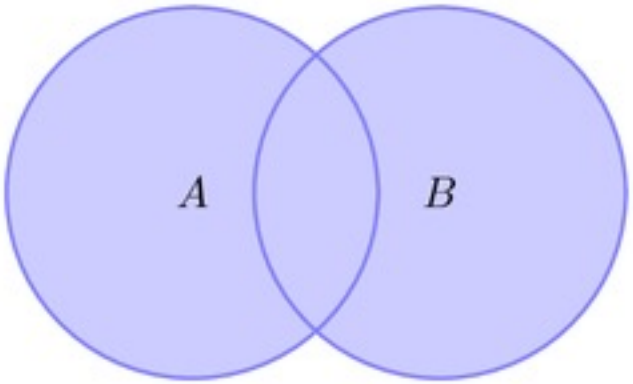
Intersection A & B



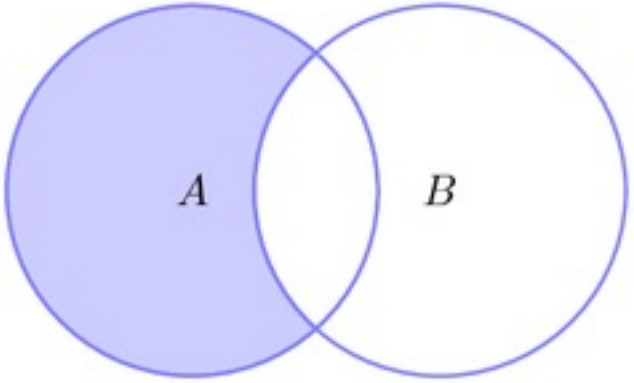
Symmetric difference A ^ B



Union A | B



Difference A - B



Summary: Tuples vs. Dictionaries vs. Sets

- All three data types store a collection of items.
- Tuples and Dictionaries allow nesting, heterogeneity and arbitrary depth.
- Choice of data type depends on intended use:
  - Tuples: Best suited for maintaining a copy of the collection that will not be accidentally changed during the program.
  - Dictionary : Best suited for storing labeled items, especially in collections where frequent searching is required.
  - Sets : Best suited for unordered collections of items that benefit from set theory operations.

Data structures

- Lists, Dictionaries, Tuples and Sets are all types of data structures.
- A data structure is a defined way to handle a collection of items.
- There are two types of operations one can do on a data structure: 1. Queries and 2. Updates
- Different data structures are optimized for different kinds of data as well as for the expected kind of operation on that data.

Lists

- Lists are useful for types of data that can be numerically indexed.
- Think of them as queues, items in the queue are processed in the order they were received.
- Lists can be changed or extended, items within it do not have to be unique.
- Separate lists can be linked with the index, but you are responsible for maintaining data integrity when making changes to one or more list.

Dictionaries

- Dictionaries are the most efficient data structures for quick lookups.
- Need a logical association between key and pair.
- BUT, as the size of the dictionary increases its lookup speed deteriorates (Hash collisions).
- They are easy to update since order is not important.

Sets

- Sets are best suited for storing unique items when the count of each item does not matter.
- Great for removing duplicates from a list, finding overlaps between two lists etc.
- Think of sets as valueless dictionaries, they have the same advantages and disadvantages as dictionaries.

Tuples

- Tuples are used when data integrity needs to be maintained.
- Tuples are also more memory efficient than lists because their exact size is known before hand.
- Tuples are most useful when a few linked characteristics of an object are read in and never changed. E.g., Geographic or genome coordinates.

How to pick a data structure

- Pick the data structure that is most reliable for your data.
- Pick the data structure most optimized for the operation you perform most often.
- Pick the data structure that performs well when the size of your data increases.
- Often, the best data structure is a combination of one or more types e.g., list of dictionaries, or dictionary of lists,

How would you store this data?

```
{
    "glossary": {
        "title": "example glossary",
                "GlossDiv": {
            "title": "S",
                        "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                                        "SortAs": "SGML",
                                        "GlossTerm": "Standard Generalized Markup Language",
                                        "Acronym": "SGML",
                                        "Abbrev": "ISO 8879:1986",
                                        "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                                                "GlossSeeAlso": ["GML", "XML"]
                    },
                                        "GlossSee": "markup"
                }
            }
        }
    }
}
```

How would you store this data?


```
{"NP_414543.1":["GO:0005737","GO:0019202","GO:0016616","GO:0009092","GO:0004412",
"GO:0044424","GO:0009089","GO:0046451","GO:0009085","GO:0016774","GO:0009070","GO
:0004072","GO:0009090"],
"NP_414544.1":["GO:0019202","GO:0006566","GO:0016773","GO:0005829","GO:0004413","
GO:0044444","GO:0009067","GO:0009088","GO:0009069","GO:0009092"],
"NP_414545.1":["GO:0006566","GO:0016838","GO:0005829","GO:0044444","GO:0009067","
GO:0009088","GO:0004795"],
"NP_414547.1":["GO:1901700","GO:0005829","GO:0044444","GO:0033194","GO:0006979"],
"NP_414548.1":["GO:0006810","GO:0016020","GO:0044464","GO:0071944","GO:0005886","
GO:0055085"]
}
```


Option 1: Optimized for finding gene IDs
Option 2: Optimized for finding genes that match GO term
Option 3: Most memory-efficient