HSS 611 - Week 5: Sets, Dictionaries, JSON

2023-09-25

Agenda

- Brief intro to time complexity
  - Understand why we need sets and dictionaries
- Sets
- Dictionaries
- JSON

Time Complexity

- **Time complexity** of a program
- Think in terms of size of input
- As the size of the input grows, how does time behave?

Time Complexity

- Need to understand how object represented in memory
- List:
  - Ordered objects stored one after another
  - `0 -> 1 -> 2 -> 3 -> 4`
  - Start from `0`, go one-by-one

Time Complexity

- Check whether an item is in the list (with `in` operator)

```
L = ['apple', 'orange', 'cherry', 'banana']

'cherry' in L
```

```
True
```

- Looks through all items one by one
  - 'apple' is not 'cherry'
  - 'orange' is not 'cherry'
  - 'cherry' is 'cherry', stop, return True
- Could have gone until end of the list

Time Complexity

- By default we think of the **worst case** scenario (thus upper limit)
- Assume the length of the list is `n`
- In the worst-case scenario, the item being searched for is not present

Time Complexity

- This is denoted O(n), Big O notation
- Big O notation describes the behavior of an algorithm as input grows
- O(n) means when input size doubles, triples, etc., time will grow linearly too
- O(1), O(n^2), O(2^n), etc.

Sets

- Sets store **unique** elements—no duplicates
- Sets do **not** order items
- Use hashing to efficiently store and retrieve elements
- See here for more on hashing
- Great for lookups and other operations (e.g., deletion)

Sets

- Looking up an element that is not in the list/set.
- Time spent on searching for 10^n +1 from a list/set whose length is 10^n

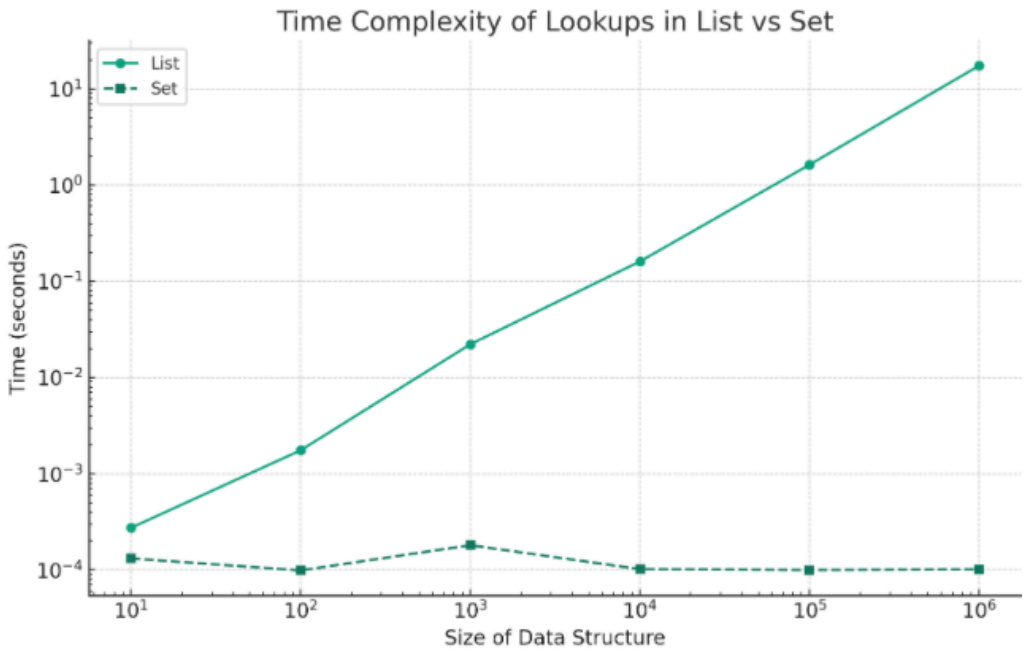

Figure 1: Lookup: list vs. set

Sets

- Checking whether an item is in a set:
  - Always takes the same time
  - Does **not** depend on set size
  - O(1) (constant time in Big O notation)

Sets

- Created with curly braces

```
my_set = {15, 'a', 4, 'k', 'k'} # duplication ignored
print(my_set)
```

```
{'a', 4, 'k', 15}
```

- Empty set is created with `set()`

```
my_empty_set = set()
my_empty_dict = {} # note that { } creates an empty diction
print(type(my_empty_dict))
```

```
<class 'dict'>
```

Sets

- Create a set of individuals banned from flight

```
flight_banned = {'Jane', 'Josh', 'John', 'Jess'}
```

Sets

- John tries to buy a ticket
- We can actually check the hash values

```
print('John' in flight_banned)
for element in flight_banned: # note that sets are iterable
    hash_value = hash(element)
    print(element + ': ' + str(hash_value))
```

```
True
Josh: -9116679342382290344
Jess: -1785976513801132799
Jane: -4961119687784961 97
John: 7426867144113784480
```

- Much faster than looking up in a list

Sets

- What will it return?

```
print(flight_banned)
flight_banned[0:2]
```

Sets

Set:

- Will **only** check the memory location

List:

- It will check the list **one by one**, till the end if necessary

Set Operations

- Let's create a set
- The order in which we supply the items **does not** matter for internal representation

```
s = {'Beth', 'Ali'}
print(s)

s = {'Ali', 'Beth'}
print(s)
```

```
{'Beth', 'Ali'}
{'Beth', 'Ali'}
```

Add an element to a set

- `.add()` an element to a set, no need for re-assignment

```
s.add('Cole')
print(s)
```

```
{'Beth', 'Ali', 'Cole'}
```

Set Operations

- If we add again, it won't change anything (duplicates not allowed)

```
s.add('Cole')
print(s)
```

```
{'Beth', 'Ali', 'Cole'}
```

Union

- `union()` will **return** the union of the two sets

```
s_new = {'Jane', 'John', 'Ali'}
s.union(s_new)
```

```
{'Ali', 'Beth', 'Cole', 'Jane', 'John'}
```

- But does not update the original set

```
print(s)
s_union = s.union(s_new) # needs to be assigned
print(s_union)
```

```
{'Beth', 'Ali', 'Cole'}
{'John', 'Beth', 'Ali', 'Cole', 'Jane'}
```

Union

- The `update()` method would both take the union **and** update original

```
s.update(s_new)
print(s)
```

```
{'John', 'Beth', 'Ali', 'Cole', 'Jane'}
```

Difference

- `difference()` will **return** the difference

```
s.difference({'Ali'})
```

```
{'Beth', 'Cole', 'Jane', 'John'}
```

- Original set still includes ‘Ali’

```
print(s)
```

```
{'John', 'Beth', 'Ali', 'Cole', 'Jane'}
```

Difference

- `difference_update()` will actually modify the original set

```
s.difference_update({'Ali'})
print(s)
```

```
{'John', 'Beth', 'Cole', 'Jane'}
```

Intersection

- `intersection()` and `intersection_update()` work similarly
- Check out this link for more

Set comprehension

- Similar to list comprehension
- use curly braces `{}`
- **absence** of `key:value` will make it a set
- unique letters in a string:

```
city = 'the tomato'
letters = {letter.lower() for letter in city if letter != ' '}
print(letters)
```

```
{'o', 't', 'h', 'a', 'e', 'm'}
```

Set comprehension

- Another example

```
states = ['mi', 'ma', 'pa', 'mi', 'pa', 'ca']

{state.upper() for state in states if state.startswith('m')}
```

```
{'MA', 'MI'}
```

Dictionaries

- Dictionaries are made up of **`key: value`** pairs
- Also created with curly braces:

```
salary = {'Jane': 100,
          'Jess': 150,
          'Janet' : 200}
```

Dictionaries

- Initialize an empty dictionary with

```
my_empty_dict = {} # this does not produce an empty set (us
```

- Or

```
my_empty_dict = dict()
```

Dictionaries

- Access the value using the `key`

```
salary = {'Jane': 100,
          'Jess': 100,
          'Janet' : 200}
print(salary)
print(salary['Jane'])
```

```
{'Jane': 100, 'Jess': 100, 'Janet': 200}
100
```

- Assign new values to existing key:

```
salary['Jess'] = 175
print(salary['Jess'])
```

```
175
```

Dictionaries

- Create new entry

```
salary['Allison'] = 130
```

- Check that new entry is there

```
print(salary)
```

```
{'Jane': 100, 'Jess': 175, 'Janet': 200, 'Allison': 130}
```

Dictionaries

- We could achieve same functionality with two lists
- As long as those are stored in correct order

Person	Grade
Ali	A+
Bella	A+
Rose	A
Sam	B+

```
person = ['Ali', 'Bella', 'Rose', 'Sam']
grade = ['A+', 'A+', 'A', 'B+']
```

Dictionaries

- What's Rose's grade?

```
# get Rose's index
ind = person.index('Rose')
print(ind)
```

2

```
# get the grade at that index
grade[ind]
```

'A'

Dictionaries

- This is, again, time inefficient
- Takes O(`n`) time
- Dictionary takes constant time, O(1)
- Dictionary has **unique, unordrered** keys (just like set)

Dictionary Methods

```
grades = dict(zip(person, grade))
print(grades)
```

```
{'Ali': 'A+', 'Bella': 'A+', 'Rose': 'A', 'Sam': 'B+'}
```

- Get all keys in the dictionary

```
grades.keys()
```

```
dict_keys(['Ali', 'Bella', 'Rose', 'Sam'])
```

Dictionary Methods

- Get all the values

```
grades.values()
```

```
dict_values(['A+', 'A+', 'A', 'B+'])
```

Dictionary Methods

- Get all the pairs

```
grades.items()
```

```
dict_items([('Ali', 'A+'), ('Bella', 'A+'), ('Rose', 'A'),
```

Iterating over a dictionary

- Will automatically iterate over **keys** if not specified (e.g., using grades.values())

```
for person in grades.keys():
  print(person + "'s grade is " + grades[person] + '.')
```

```
Ali's grade is A+.
Bella's grade is A+.
Rose's grade is A.
Sam's grade is B+.
```

Iterating over a dictionary

- Or

```
for (person, grade) in grades.items():
  print(person + "'s grade is " + grade + '.')
```

```
Ali's grade is A+.
Bella's grade is A+.
Rose's grade is A.
Sam's grade is B+.
```

Dictionaries

- Test if **key** in dictionary with **in** operator

```
'Ali' in grades
```

```
True
```

```
'Alice' in grades
```

```
False
```

- Remove item from dictionary

```
del(grades['Bella'])
```

- Check **grades**

```
grades
```

```
{'Ali': 'A+', 'Rose': 'A', 'Sam': 'B+'}
```

- Or

Keys and Values in Dictionary

- Keys
  - Must be **unique** (like we don't have the same words in actual dictionaries)
  - Must be **hashable** (`int`, `float`, `string`, `bool`, `tuple`, etc.)
- Values
  - Can be duplicates
  - Can be lists, other dictionaries, any type
- No order to keys (and thus values), just like there is no order in a set
- Hashing and order are in tension

Example

- Count the number of times a word occurs in a sentence

```
def word_freq(sentence):
    words_list = sentence.split()
    freq = {}
    for word in words_list:
        if word in freq:
            freq[word] += 1
        else:
            freq[word] = 1
    return freq
```

Example

- Let's try it

```
quote = "Float like a butterfly. Sting like a bee."
```

```
print(word_freq(quote))
```

```
{'Float': 1, 'like': 2, 'a': 2, 'butterfly.': 1, 'Sting': 1, 'bee.': 1}
```

Nested dictionaries

```
sci_fi_books = {
  "Dune": {
    "Author": "Frank Herbert",
    "Published Year": 1965,
    "Main Themes": ["Politics", "Religion", "Ecology"],
    "Character Protagonist": "Paul Atreides"
    },
  "Neuromancer": {
    "Author": "William Gibson",
    "Published Year": 1984,
    "Main Themes": ["Cybernetics", "Artificial Intelligence", "Hacker Culture"]
    "Character Protagonist": "Case"
    },
  "Foundation": {
    "Author": "Isaac Asimov",
    "Published Year": 1951,
    "Main Themes": ["Collapse and Renewal", "Psychohistory", "Politics"],
    "Character Protagonist": "Hari Seldon"
    }
  }
```

Nested dictionaries

```
print(sci_fi_books)
```

```
{'Dune': {'Author': 'Frank Herbert', 'Published Year': 1965, 'Main Themes': ['P
```

Nested dictionaries

- Use `PrettyPrinter` function in `pprint` module
- For more on `pprint` see here

```
import pprint
pp = pprint.PrettyPrinter()
pp.pprint(sci_fi_books)
```

```
{'Dune': {'Author': 'Frank Herbert',
          'Character Protagonist': 'Paul Atreides',
          'Main Themes': ['Politics', 'Religion', 'Ecology'],
          'Published Year': 1965},
 'Foundation': {'Author': 'Isaac Asimov',
                'Character Protagonist': 'Hari Seldon',
                'Main Themes': ['Collapse and Renewal',
                                'Psychohistory',
                                'Politics'],
                'Published Year': 1951},
 'Neuromancer': {'Author': 'William Gibson',
                 'Character Protagonist': 'Case',
                 'Main Themes': ['Cybernetics',
                                 'Artificial Intelligence',
                                 'Hacker Culture'],
                 'Published Year': 1984}}
```

Nested dictionaries

- Indexing/slicing like this

```
print(sci_fi_books['Dune']['Main Themes'][-1])
```

Dictionary Methods

- Check out the full list of dictionary methods here

Dictionary comprehension

- Dictionary comprehension: same thing, except the first part is a `key:value` pair

Dictionary comprehension

- Combine two lists into a dictionary **without** dictionary comprehension

```
countries = ['USA', 'Canada', 'Mexico', 'Japan']
capitals = ['Washington, D.C.', 'Ottawa', 'Mexico City', 'ToKyo']

capital_dict = {}
for i in range(len(countries)):
    capital_dict[countries[i]] = capitals[i]

capital_dict
```

```
{'USA': 'Washington, D.C.',
 'Canada': 'Ottawa',
 'Mexico': 'Mexico City',
 'Japan': 'ToKyo'}
```

Dictionary comprehension

- Another way, but still without dictionary comprehension
- use `zip()` function

```
capital_dict = dict(zip(countries, capitals))
print(capital_dict)
```

```
{'USA': 'Washington, D.C.', 'Canada': 'Ottawa', 'Mexico': 'Mexico City', 'Japan
```

Dictionary comprehension

- With dictionary comprehension

```
capital_dict = {key:value for key, value in zip(countries, capitals)}
print(capital_dict)
```

```
{'USA': 'Washington, D.C.', 'Canada': 'Ottawa', 'Mexico': 'Mexico City', 'Japan
```

Dictionary comprehension

- As with list comprehension, we can add conditions with `if` / `else`
- Condition on key:

```
{key:value for key, value in zip(countries, capitals) if key == 'USA'}
```

```
{'USA': 'Washington, D.C.'}
```

- Condition on value:

```
{key:value for key, value in zip(countries, capitals) if value == 'Ottawa'}
```

```
{'Canada': 'Ottawa'}
```

Dictionary comprehension

- Another example

```
odd = {key:(key % 2 == 1) for key in range(6)}
print(odd)
```

```
{0: False, 1: True, 2: False, 3: True, 4: False, 5: True}
```

```
odd[3]
```

```
True
```

JSON

- JSON (JavaScript Object Notation)
- A file format that stores structured data in a simple and readable way
- Very popular for exchanging data with web servers
- Let's see an example of Twitter here

JSON

- JSON is built on two main data structures
  - A collection of key/value pairs
  - An ordered sequence of items
- Makes it a very natural way of storing
  - Dictionaries
  - Lists

JSON

- Python has built-in `json` module
- `json.load()` creates a dictionary or list by from a JSON file

```
import json
with open('json_example.json', 'r') as file:
    data = json.load(file)
```

JSON

- Python reads this in as a dictionary

```
print(type(data))
```

```
<class 'dict'>
```

JSON

- Python reads this as a dictionary

```
pp.pprint(data)
print(data['quiz']['maths']['q1']['question'])
print(data['quiz']['maths']['q1']['options'][:3])
```

```
{'quiz': {'maths': {'q1': {'answer': '12',
                           'options': ['10', '11', '12', '13'],
                           'question': '5 + 7 = ?'},
                    'q2': {'answer': '4',
                           'options': ['1', '2', '3', '4'],
                           'question': '12 - 8 = ?'}},
          'sport': {'q1': {'answer': 'Huston Rocket',
                           'options': ['New York Bulls',
                                       'Los Angeles Kings',
                                       'Golden State Warriros',
                                       'Huston Rocket'],
                           'question': 'Which one is correct team name in '
                                       'NBA?'}}}}
5 + 7 = ?
['10', '11', '12']
```

JSON

- `json.loads()` creates a dict or a list from a string (not from a file)
- From a string (thus the `s`)

```
json_string = '''{"quiz": {"sport": {"q1": {"question": "Which one is correct t
```

JSON

- `json.loads()` creates a dictionary or list from a string (not from a file)
- From a string (thus the `s`)

```
json_object = json.loads(json_string)
print(type(json_object))
pp.pprint(json_object)
```

```
<class 'dict'>
{'quiz': {'maths': {'q1': {'answer': '12',
                           'options': ['10', '11', '12', '13'],
                           'question': '5 + 7 = ?'},
                    'q2': {'answer': '4',
                           'options': ['1', '2', '3', '4'],
                           'question': '12 - 8 = ?'}},
          'sport': {'q1': {'answer': 'Huston Rocket',
                           'options': ['New York Bulls',
                                       'Los Angeles Kings',
                                       'Golden State Warriros',
                                       'Huston Rocket'],
                           'question': 'Which one is correct team name in '
                                       'NBA?'}}}}
```

JSON

- Whether JSON will be read as `dict` or `list` depends on content
- If it is structured in `{}`, it'll be a dictionary
- If it is structured in `[]`, it'll be a list
- Of course, it can be any combination of things:
  - A dictionary where values are a list
  - A list of dictionaries
  - Any different combinations or nesting of these
  - This is what makes it very flexible

JSON

- `json.dump()` writes a dictionary/list to a JSON file

```
kaist = {
    'full_name': 'Korea Advanced Institute of Science and T
    'location': 'Daejeon',
    'year_founded': 1971
    }
type(kaist)
```

```
dict
```

```
with open('json_data_kaist.json', 'w') as file:
    json.dump(kaist, file, indent = 4)
```

JSON

- `json.dumps()` outputs json as a string (thus the `s`)

```
json_string = json.dumps(data)
print(type(json_string))
print(json_string)
```

```
<class 'str'>
{"quiz": {"sport": {"q1": {"question": "Which one is correct team name in NBA?"
```