

Maps, Key-Value Stores, JSON

Theory

A lot of distributed computations you see in this class take place on objects often referred to as Maps or collections of Key-Value pairs or Key-Value stores.

Maps. In our conversations, a **map** is a partial finite function between two domains. That is:

Let $\mathcal{K} = \{f_\infty, \dots, f_\setminus, \dots\}$ be a set of objects called *keys*¹. Let \mathcal{V} be another set of objects (possibly infinite, possibly uncountable).

Let $K = \{k_1, \dots, k_N\} \subseteq \mathcal{K}$ be a finite set of keys.

A **map** is any function $M : K \rightarrow \mathcal{V}$.

Dictionary. Another name for a **map** defined as above that has been traditionally used in programming languages is **dictionary**.

We use the terms **map** and **dictionary** as synonyms.

Key-Value pairs. Given a **map** M , consider some key $k \in K$. Let $v = M(k)$. The pair $\langle k, v \rangle$ is known as a **key-value pair** in M .

Key-Value stores. Another way of looking at **maps** is to think of them as *sets of key-value pairs*. Indeed, we can describe a **map** M both as a function:

$$M : K \rightarrow \mathcal{V}$$

as well as a set:

$$M = \{\langle k, v \rangle \mid k \in K, v \in \mathcal{V}, \sqsubseteq = \mathcal{M}(\|\|\}\}.$$

¹In this definition, this set is made countable. This is not a strict requirement, but under most circumstances it suffices.

or

$$M = \{(k, M(k)) | k \in K\}$$

These two views of a **map** (as a function or as a set) are equivalent.

When viewed as a set of key-value pairs, a **map** is often referred to as a **Key-Value Store**.

Key-Value Store as Abstract Data Type

Maps/dictionaries are often implemented as an **Abstract Data Type**. The **Map** ADT comes with the following set of operations:

Operation	Parameters	Result	Action
put	key, value	none	add the <key, value> pair to the map
get	key	value	retrieve the value given a key
exists	key	True/False	return True if map contains a key
size	none	<i>integer</i>	return the number of key-value pairs in map
remove	key	none	remove the key-value pair with given key from map
update	key, value	none	replace the existing key-value pair for given key with the new <key, value> pair
clear	none	none	remove all key-value pairs from map

Note: The minimally viable **Map** ADT really just needs to implement **put** and **get** operations. Truly mutable maps will also require **remove** operation. All other operations are there for convenience.

Key-Value Store Implementations

Many programming languages have **Key-Value** stores as implementations of the **Map** ADT.

Python. Python implements maps as dictionary objects.

Java. Java has a representation of the map ADT: the **Map <K,V>** interface. Its implementations are **HashMap**, **TreeMap** and **SortedMap**. The **Map** interface essentially implements the entire set of **map** operations, plus adds a few more operations for convenient manipulation of data.

JSON. A single JSON object can be easily viewed as a dictionary mapping the attribute/field names to their values.

JSON

JSON, short for JavaScript Object Notation is a human- and machine-readable serialization mechanism for representing collections of key-value pairs².

Properties. JSON has the following nice properties.

- **JSON is plain text.** JSON objects are plain text objects that can be viewed and read by humans.
- **JSON is lightweight.** JSON specification is very simple.
- **JSON is structured.** JSON objects can contain other JSON objects in them allowing for structured data representation.
- **JSON is schemaless.** JSON does not require a schema to operate. This means JSON objects can be used to conveniently represent semi-structured data.

JSON Specification

JSON objects can be specified formally (in Backus-Naur notation) as follows:

```
<JSON Object> ::= '{' '}' |  
                '{' <string> ':' <value> (',' <string> ':' <value>)* '}'  
  
<JSON Array> ::= '[' ']' |  
                '[' <value> (',' <value>)* ']'  
  
<value> ::= <string> | <number> |  
           <JSON object> | <JSON array> |  
           true | false | null
```

Here:

- Identifiers in angle brackets (e.g., <JSON Object> or <string>) otherwise called *non-terminals* are specific parts of the described syntax that are being defined.
- The ::= symbol is the "is defined as" notation.
- Items in single quotes (e.g., '{' or ']') are *terminals* or the actual symbols used in the JSON syntax.

²<http://www.json.org>

- The `||` is the "or" symbol stating that a specific notion can be defined in more than one way.
- The `(...)*` notation means *one or more copy of what is inside the parentheses*.

With this in mind, here is a translation:

- A **JSON Object** is either an empty object `{ }` or a collection of comma-separated key-value pairs, inside curly braces, where the key is a **string** object, and the value is a **value** object.
- A **JSON Array** is either an empty array `[]` or a sequence of comma-separated **value** objects inside angle brackets.
- A **value** object in JSON is either a single **string** object, or a single **number** object, or a single **JSON object** or a single **JSON array**. In addition, three trivial **value** objects exist: **true**, **false** and **null**.
- **string** objects are sequences of characters in quotes. **number** objects follow the standard syntax for numeric notation for either *integer* or *floating point* numbers. This includes scientific notation.

Examples. Here are some sample JSON objects.

```
{ "name": "Bob",
  "class": "senior",
  "grades": ["A", "A", "B"]
}

{ "id": 103424,
  "product": {"name": "widget",
              "description": [{ "language": "English",
                                "text": "this is a widget"},
                              {"language": "Welsh",
                                "text": "Mae hon yn widget"}
                                ]
              },
  "price": 5.99,
  "stock": 73
}

{ "array1" : [1,2,3,4,5],
  "array2" : ["a", "b", "c"],
  "array3" : [{"a":1}, 2, "c"]
}
```

Handling of JSON Objects in Python

Both Python 2 and Python 3 have standard library support for JSON Objects.


```
>>> file=open('json','r')
>>> d1 = json.load(file)
>>> d1
{'name': {'first': 'Mary', 'las': 'Young'}, 'id': 75, 'magicDigits': [1, 2, 4, 5, 'nothing'],
'hometown': {'state': 'CA', 'town': 'Santa Cruz'}}
>>> d1['first']
>>> d1['name']
{'first': 'Mary', 'las': 'Young'}
>>> d1['name']['first']
'Mary'
```

Creating JSON objects.

```
>>> kv = {"a":3, "b":17, "g": [1,2,3]}
>>> jsonKV = json.dumps(kv)
>>> jsonKV
'{"b": 17, "a": 3, "g": [1, 2, 3]}'
>>> fileToWrite = open('myJson', "w")
>>> json.dump(kv, fileToWrite)
>>> fileToWrite.close()
>>> fl = open('myJson', "r")
>>> text = fl.read()
>>> text
'{"b": 17, "a": 3, "g": [1, 2, 3]}'
```