Data Representation 1: Tabular, relational & JSON

Sandy Brownlee

sbr@cs.stir.ac.uk

Data Representation

- Programming languages have a rich set of data representations:
  - Trees, lists, sets, arrays, dictionaries ...
  - Objects
- Storage options are often more limited
- You can serialise an object, but you need to know exactly how to read the data to get the object back into memory
- So called Object Impedence

Tabular Data

- Examples: csv, spreadsheets, relational DB tables

	Variable	Variable	Variable
Entry	Value	Value	Value
Entry	Value	Value	Value
Entry	Value	Value	Value

- Limits the variables you can record to those with a column

More Flexiblility

- Tables are fine when every entry has the same variables associated with it, for example name, address, phone number
- They become more problematic when different entries have different variables
- Or some entries are lists, or objects themselves

Relational Model

- The relational model (see database course for more) solves this problem with
  - Joins
  - Foreign keys

Example

- Lets try to store the following facts:
  - Tom lives in Bridge of Allan
  - He has three email addresses
  - He owns a house in Causewayhead

Example

PersonID	Name
1	Tom

PersonID	email
1	tom@work
1	tom@home
1	tom@gmail

PersonID	HouseID
1	1
1	2

HouseID	Line1	Line2	Postcode
1	1 High St	Bridge of Allan	FK9 4LA
2	1 Wallace St	Causewayhead	FK9 5QW

```
Select Name, email, Line1, Line2, Postcode FROM People, Houses, Emails, HousePeople
WHERE People.PersonID=Houses.PersonID AND People.PersonID=HousePeople.PersonID
AND People.PersonID=email.PersonID AND HousePeople.HouseID=Houses.HouseID
```

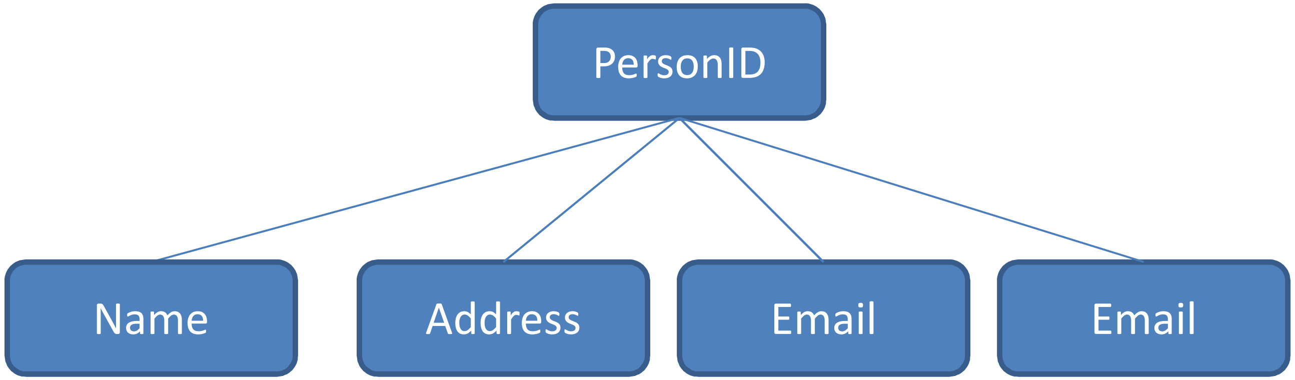
Example

- That works, but it is not too pretty
- Becomes complex with very large number of columns and tables
- How else might we store that data?

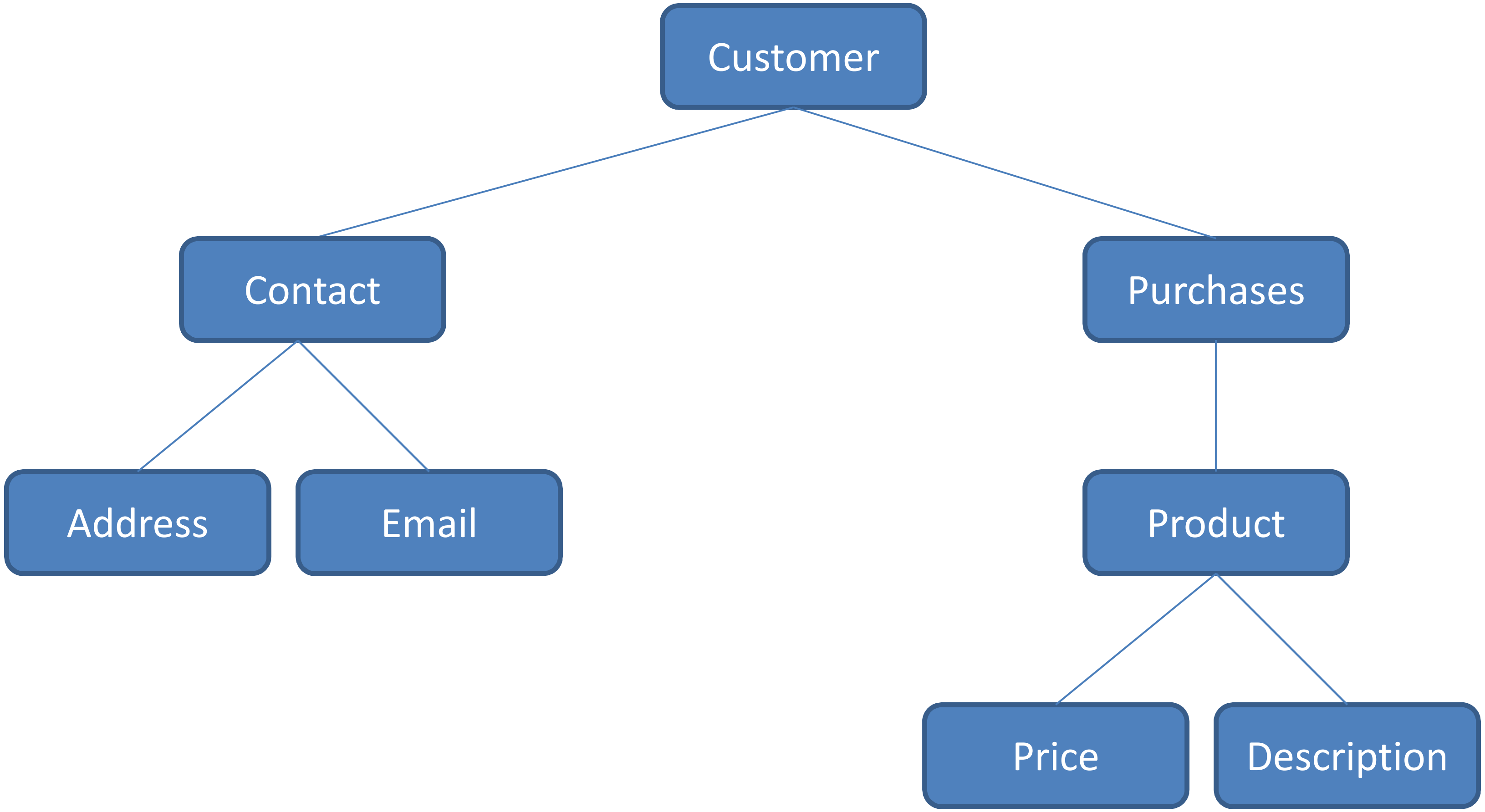
Documents - Tree Structure

- Store data related to particular objects or subjects in documents
- Data can be arranged into a tree structure
- That turns out to be pretty much anything:

Tabular Data



Customer Data



XML

- eXtensible Markup Language
- Extensible, meaning you can define your own tags e.g. **`<name>Bob</name>`**
- Markup language means that data is stored and represented as text, with the structure of the data defined within the text in a way that is very general
- Now a very commonly used standard
- We'll come back to this in a later lecture

JSON

- XML is powerful and very common
- But it is rather large and cumbersome for some uses
- JSON (JavaScript Object Notation) is gaining popularity as an alternative
- Origins in JavaScript, but language independent

JSON

- Hierarchy of name, value pairs
- Limited types
  - string, number, object, array, true/false, null
- See www.json.org for specification and documentation

JSON / XML comparison

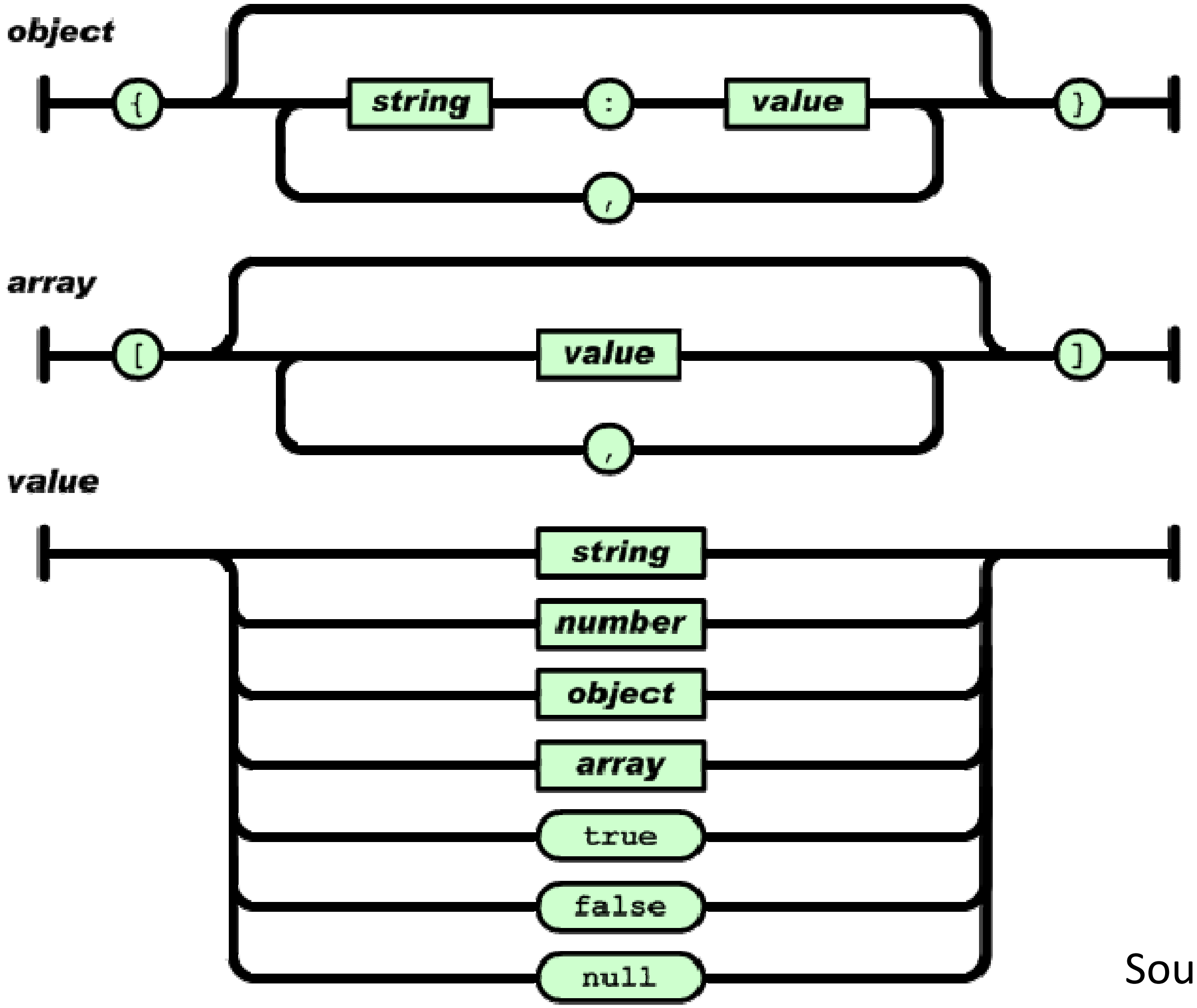
- More compact / less verbose
  - XML:

```
<person>
    <age>42</age>
    <name>Bob</name>
</person>
```

  - JSON:

```
{"age" : 42, "name" : "Bob"}
```

JSON Structure



Source: json.org

Object

- A JSON object in its simplest form is a set of name:value pairs
- An object is enclosed in { } braces
- The name part is a string, so enclosed in “”
- The colon means equals
- The value can be a single value or an array of values
- Values can be objects themselves

Array

- An array of values, including objects and arrays
- Examples

[“Fish”,2,3] Strings and numbers

[[1,2,3],[3,4,5]] Array of arrays

[{“Name”: “Sandy”},{“email”: “sbr”}] Array of objects

- Can be of mixed type – but this won't work if parsing using languages like Java

Value

- String
  - Like a Java string
  - “Enclosed in double quotes”
  - Escaped with \
- Number – No more specific types such as int, float

  - also no infinity / not-a-number
- Object – An embedded JSON object
- Array – An array of values
- true / false (must be lowercase)
- null

Example

```
{
    “Name”: “Tom”,
    “Email”: [“tom@home”,”tom@work”,”tom@gmail”],
    “Address”:
     [{“Line1”: “1 High St”,”Line2”: “Bridge of Allan”,
    “Postcode”: “FK9 4LA”},
     {“Line1”: “1 Wallace St”,”Line2”: “Causewayhead”,
    “Postcode”: “FK9 5QW”}]
}
```

JSON web services (1)

- IP + location
- http://www.telize.com/geoip

```
{"dma_code":"0","ip":"139.153.253.xxx","asn":"AS786","city":"Stirling","latitude":56.1167,"country_code":"GB","offset":"2","country":"United Kingdom","region_code":"W6","isp":"Jisc Services Limited","timezone":"Africa\/Gaborone","area_code":"0","continent_code":"EU","longitude":-3.95,"region":"Stirling","postal_code":"FK8","country_code3":"GBR"}
```

JSON web services (2)

- True random numbers
- https://qrng.anu.edu.au/API/jsonI.php?length=10&type=uint8

```
{
   "type":"uint8",
   "length":10,
   "data":[201,155,166,144,157,80,169,9,204,47],
   "success":true
}
```

JSON Schema

- Allows formal definition of the structure for JSON documents, good for interoperability
- JSON schema definition is a JSON document
- Specification currently in draft but already available for use
- More details and documentation available at http://json-schema.org
- Validator at http://jsonschemalint.com

JSON Schema (2)

This is a schema document

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Cat",
    "properties": {
        "name": {
            "type": "string"
        },
        "age": {
            "type": "number",
            "description": "Your cat's age in years.",
            "minimum": 0
        },
        "clawstrimmed": {
            "type": "boolean"
        }
    },
    "required": ["name", "age"],
    "additionalProperties": false
}
```

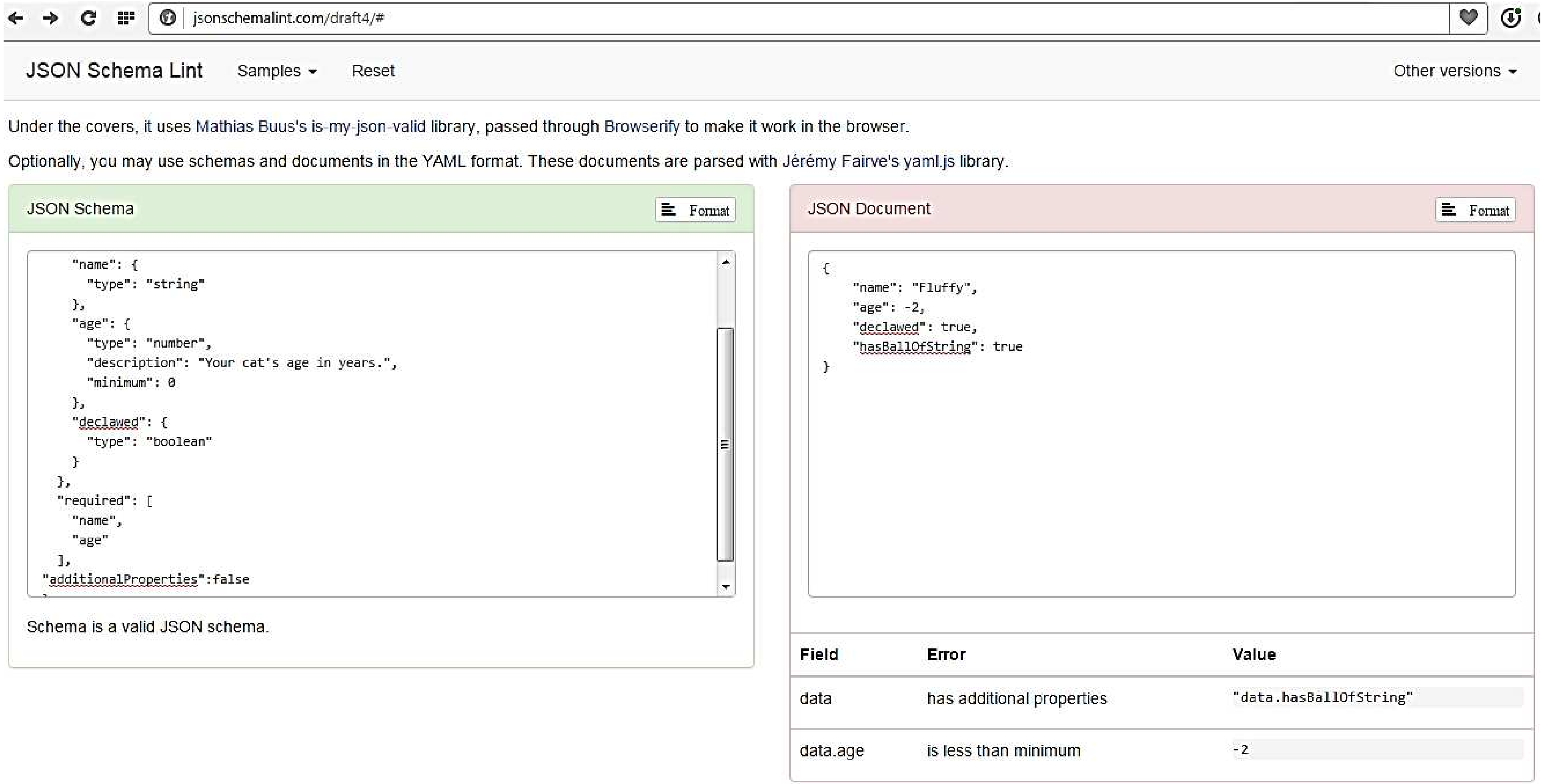
Title of the schema

An array of the allowed name-value pairs

Names are optional unless included here

Additional names are not allowed

JSON Schema (3)



Under the covers, it uses Mathias Buus's is-my-json-valid library, passed through Browserify to make it work in the browser.

Optionally, you may use schemas and documents in the YAML format. These documents are parsed with Jérémy Fairve's yaml.js library.

```
      "name": {
        "type": "string"
      },
      "age": {
        "type": "number",
        "description": "Your cat's age in years.",
        "minimum": 0
      },
      "declawed": {
        "type": "boolean"
      }
    },
    "required": [
      "name",
      "age"
    ],
  "additionalProperties":false
```

Schema is a valid JSON schema.

```
{
    "name": "Fluffy",
    "age": -2,
    "declawed": true,
    "hasBallOfString": true
}
```

Field	Error	Value
data	has additional properties	"data.hasBallOfString"
data.age	is less than minimum	-2

JSON in Python (1)

- Use JSON library
- Third party libraries exist for JSON Schema
  - Not covered here
- Library maps types like this (bold for JSON->Python):

Python	JSON
dict	object
list, tuple	array
str, **unicode**	string
int, long, float	number
True	true
False	false
None	null

JSON in Python (2)

- `json.load(f)`
  - read JSON from file f
- `json.loads(catData1)`
  - read JSON from string catData1
- `json.dump(catData3, f)`
  - write dictionary catData3 as JSON to file f
  - `json.dump(catData3, f, indent=4)` enables "pretty printing", more human-readable
- `s = json.dumps(catData3)`
  - convert catData3 to JSON string s

JSON Python Example - reading

```python
import json

with open('cat-fluffy.json') as f:
    parsedCatData2 = json.load(f)
    print(parsedCatData2)
    print("Age:" + str(parsedCatData2['age']))
    print("Stays at number: " + parsedCatData2['address']['number'])

    if (parsedCatData2['clawstrimmed']):
        print('Safe!')
    else:
        print('Get some gloves!')

    print ("Friends of " + parsedCatData2['name'] + ":")
    for friend in parsedCatData2['friends']:
        print("  " + friend)

    print ("Full address:")
    for name, value in parsedCatData2['address'].items():
        print("  " + name + " --- " + value)

print("done")
```

JSON Python Example - reading

Output:

```
{'age': 2, 'friends': ['Spot', 'Bob', 'Mr. Meow'], 'name':
'Fluffy', 'address': {'number': '4a', 'street': 'Felix
Street'}, 'clawstrimmed': True}
Age:2
Stays at number: 4a
Safe!
Friends of Fluffy:
  Spot
  Bob
  Mr. Meow
Full address:
  number --- 4a
  street --- Felix Street
done
```

cat-fluffy.json

```
{
   "name": "Fluffy",
   "age": 2,
   "clawstrimmed": true,
   "friends": ["Spot", "Bob",
"Mr. Meow"],
   "address": {
       "number": "4a",
       "street": "Felix Street"
   }
}
```

JSON Python Example - writing

```
import json
catData3 = { 'name': 'Spot', 'age':3, 'clawstrimmed':False, \
            'address':{'number':'D', 'street':'Enterprise'}, \
            'offspring':('Spot II','Spot Junior','Dot') \
           }

print (catData3)
print (json.dumps(catData3))

with open('cat-spot.json', 'w') as f:
    json.dump(catData3, f)
```

cat-fluffy.json

```
{"name": "Spot", "age": 3, "offspring": ["Spot II", "Spot
Junior", "Dot"], "address": {"street": "Enterprise",
"number": "D"}, "clawstrimmed": false}
```

Output:

```
{'name': 'Spot', 'age': 3, 'offspring': ('Spot II', 'Spot Junior', 'Dot'),
'address': {'street': 'Enterprise', 'number': 'D'}, 'clawstrimmed': False}
{"name": "Spot", "age": 3, "offspring": ["Spot II", "Spot Junior", "Dot"],
"address": {"street": "Enterprise", "number": "D"}, "clawstrimmed": false}
```

JSON - summary

- Arguably easier for humans to read
- Easy for programs to parse
- Elegant and simple

This week's Lab

- Some more practice with RegEx
- Reading, writing and manipulating JSON files