JSON and JSON Schema

JSON

- JSON: JavaScript Object Notation
- At its heart, JSON is built on the following data structures:
  - object: `{ "key1": "value1", "key2": "value2" }`
  - array: `["first", "second", "third"]`
  - number: `42`, `3.1.415926`
  - string: `"this is a string"`
  - boolean: `true`, `false`
  - null: `null`

The following table maps the Javascript types to Python types:

Javascript	Python
string	string
integer	int
number	int/float
object	dict
array	list
boolean	bool
null	None

What is a Schema?

Consider the JSON representation of a person. The following are two different representations of the same person:

First representation of a person. `e1-1.json`

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

Second representation of a person. `e1-2.json`

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

- While developing applications, you may wish to have a uniform format for persons.
- Maybe the second one which breaks down the values into smaller components.
- JSON Schema is a specification that allows us to "describe" the structure of the JSON representations.
- The syntax of JSON Schema is JSON itself!

Here is a JSON schema that describes the second representation of a person: `e1.schema`

```
{
  "$schema": "http://json-schema.org/schema#",
  "$id": "http://yourdomain.com/schemas/myschema.json",
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string", "format": "date-time" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type" : "string" }
      }
    }
  }
}
```

The Python implementation of JSON Schema is available at:

```
https://github.com/Julian/jsonschema
```

Many online validators!

https://www.jsonschemavalidator.net/

https://extendsclass.com/json-schema-validator.html

The following is a terminal session that shows how to validate with `jsonschema`:

```
macbook-pro:json-schema raj$ jsonschema -i e1-1.json e1.schema
Mount Vernon, Virginia, United States: 'Mount Vernon, Virginia, United St
ates' is not of type 'object'

macbook-pro:json-schema raj$ jsonschema -i e1-2.json e1.schema
macbook-pro:json-schema raj$
```

Learn JSON Schema using examples

Trivial schemas

Example 1:

```
{ }
```

or

```
true
```

matches ANY well-formed JSON document!

Example 2:

```
false
```

matches NO JSON document!

Basic Data Types: Strings, Numbers, Integers, Boolean

Example 3:

```
{ "type": "string" }
```

matches any string

Example 4:

```
{ "type": "number" }
```

matches any number (includes integers and floats)

Example 5:

```
{ "type": ["number", "string"] }
```

matches any number OR string

Example 6:

```
{
  "type": "string",
  "minLength": 2,
  "maxLength": 3
}
```

matches "ABC" but not "ABCD"

Example 7:

```
{
  "type": "string",
  "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

matches "555-1212" or "(888)555-1212" but not "(888)555-1212 ext. 532"

String Patterns

- A single unicode character (other than the special characters below) matches itself.
- ^: Matches only at the beginning of the string.
- $: Matches only at the end of the string.
- (...): Group a series of regular expressions into a single regular expression.
- |: Matches either the regular expression preceding or following the | symbol.
- [abc]: Matches any of the characters inside the square brackets.
- [a-z]: Matches the range of characters.
- [^abc]: Matches any character not listed.
- [^a-z]: Matches any character outside of the range.
- +: Matches one or more repetitions of the preceding regular expression.
- *: Matches zero or more repetitions of the preceding regular expression.
- ?: Matches zero or one repetitions of the preceding regular expression.
- +?, *?, ??: The *, +, and ? qualifiers are all greedy; they match as much text as possible. Sometimes this behavior isn't desired and you want to match as few characters as possible.
- {x}: Match exactly x occurrences of the preceding regular expression.
- {x,y}: Match at least x and at most y occurrences of the preceding regular expression.
- {x,}: Match x occurrences or more of the preceding regular expression.
- {x}?, {x,y}?, {x,}?: Lazy versions of the above expressions.

Example 8:

```
{ "type": "integer" }
```

matches any integer

Example 9:

```
{
  "type" : "number",
  "multipleOf" : 10
}
```

matches `10` , `20` , `20.0` , etc

Example 10:

```
{
  "type": "number",
  "minimum": 0,
  "exclusiveMaximum": 100
}
```

matches numbers between `0` and `100` (`0` included and `100` excluded)

Boolean Type:

```
{
  "type": "boolean"
}
```

matches `true` and `false`

Object Type

Example 11:

```
{ "type": "object" }
```

matches any object!

Example 12:

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    }
  }
}
```

matches

```
{
  "number": 1600,
  "street_name": "Pennsylvania",
  "street_type": "Avenue"
}
```

or

```
{
  "number": 1600,
  "street_name": "Pennsylvania"
}
```

or

```
{
  "number": 1600,
  "street_name": "Pennsylvania",
  "street_type": "Avenue",
  "direction": "NW"
}
```

leaving out properties is OK!

additional properties are OK!

Example 13:

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    }
  },
  "additionalProperties": false
}
```

will not match JSON with additional properties.

Example 14:

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    }
  },
  "additionalProperties": {"type": "string"}
}
```

matches with additional properties only if they are strings.

Example 15:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "email": { "type": "string" },
    "address": { "type": "string" },
    "telephone": { "type": "string" }
  },
  "required": ["name", "email"]
}
```

matches objects with "required" fields (`name` and `email` are now required).

Restrictions on property names and number of properties:

All property names are strings. We can restrict these strings in certain ways.

Example 16:

```
{
  "type": "object",
  "propertyNames": {
    "pattern": "^[A-Za-z_][A-Za-z0-9_]*$"
  }
}
```

matches objects whose property names begin with upper or lower case letter or underscore and followed by 0 or more of upper or lower case letters or digits.

Example 17:

```
{
  "type": "object",
  "minProperties": 2,
  "maxProperties": 3
}
```

matches objects with 2 or 3 properties.

Dependencies (ADVANCED Feature!):

- The `dependencies` keyword allows the schema of the object to change based on the presence of certain special properties.

- There are two forms of dependencies in JSON Schema:
  - Property dependencies declare that certain other properties must be present if a given property is present.
  - Schema dependencies declare that the schema changes when a given property is present.

Property Dependency

In the following example, whenever a credit*card property is provided, a billing*address property must also be present:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },
  "required": ["name"],
  "dependencies": {
    "credit_card": ["billing_address"]
  }
}
```

The above schema will match the following three instances:

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

```
{
  "name": "John Doe"
}
```

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

but will not match the following:

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

If you require both or none, you can include the bidirectional dependency as follows:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },
  "required": ["name"],
  "dependencies": {
    "credit_card": ["billing_address"],
    "billing_address": ["credit_card"]
  }
}
```

Schema dependencies

Schema dependencies work like property dependencies, but instead of just specifying other required properties, they can extend the schema to have other constraints.

For example, here is another way to write the above:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" }
  },
  "required": ["name"],
  "dependencies": {
    "credit_card": {
      "properties": {
        "billing_address": { "type": "string" }
      },
      "required": ["billing_address"]
    }
  }
}
```

matches

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

and

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

but not

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

Pattern Properties

The following schema indicates that all "string" valued property names must begin with S_ and

all "integer" valued property names must begin with I_

```
{
  "type": "object",
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  },
  "additionalProperties": false
}
```

matches

```
{ "S_25": "This is a string" }

{ "I_0": 42 }
```

but not

```
{ "S_0": 42 }
```

Another Example:

```
{
  "type": "object",
  "properties": {
    "builtin": { "type": "number" }
  },
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  },
  "additionalProperties": { "type": "string" }
}
```

matches

```
{ "builtin": 42 }

{ "keyword": "value" }
```

but not

```
{ "keyword": 42 }
```

Arrays

```
{
  "type": "array"
}
```

matches

```
[1, 2, 3, 4, 5]

[3, "different", { "types" : "of values" }]
```

but not

```
{
  "Not": "an array"
}
```

Array Items:

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

matches

```
[1, 2, 3, 4, 5]
```

but not

```
[1, 2, "3", 4, 5]
```

contains:

```
{
  "type": "array",
  "contains": {
    "type": "number"
  }
}
```

matches

```
["life", "universe", "everything", 42]
```

but not

```
["life", "universe", "everything", "forty-two"]
```

Tuple validation:

```
{
  "type": "array",
  "items": [
    {
      "type": "number"
    },
    {
      "type": "string"
    },
    {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    },
    {
      "type": "string",
      "enum": ["NW", "NE", "SW", "SE"]
    }
  ]
}
```

matches

```
[1600, "Pennsylvania", "Avenue", "NW"]
[10, "Downing", "Street"]
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

We can add

```
  "additionalItems": false
```

at the end to disallow `"Washington"`

or

```
  "additionalItems": { "type": "string" }
```

to allow additional string items in the array

array lengths:

```
{
  "type": "array",
  "minItems": 2,
  "maxItems": 3
}
```

matches arrays with 2 or 3 items.

```
{
  "type": "array",
  "uniqueItems": true
}
```

matches arrays with no duplicates allowed

Metadata:

- title: title of schema
- description: description of schema
- default: default value for missing required items
- examples: examples of json that should validate

```
{
  "title" : "Match anything",
  "description" : "This is a schema that matches anything.",
  "default" : "Default value",
  "examples" : [ "Anything", 4035 ]
}
```

enumerated type:

```
{
  "type": "string",
  "enum": ["red", "amber", "green"]
}
```

matches `"red"`, `"amber"`, `"green"` and not anything else.

We can have `enum` with mixed types!

```
{
  "enum": ["red", "amber", "green", null, 42]
}
```

Combining Schemas:

`anyOf`

```
{
  "anyOf": [
    { "type": "string", "maxLength": 5 },
    { "type": "number", "minimum": 0 }
  ]
}
```

matches `"short"` and `12`, but not `"too long"`

`allOf`

```
{
  "allOf": [
    { "type": "string" },
    { "maxLength": 5 }
  ]
}
```

matches `"hello"` but not `"longer string"`

```
{
  "allOf": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

same as `false`

`oneOf`

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}
```

exactly one! matches `10`, `9`, but not `2` or `15`

`not`

```
{
  "not": { "type": "string" }
}
```

seems obvious what this would match (-:

Reuse (definitions)

Consider the following schema specification:

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

You can place the above in the "definitions" section:

```
{
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" }
      },
    "required": ["street_address", "city", "state"]
    }
  }
}
```

and refer to it as:

```
{
  "$ref": "#/definitions/address"
}
```

or if definitions is in a separate file:

```
{
  "$ref": "definitions.json#/address"
}
```

More interesting example:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  },
  "type": "object",
  "properties": {
    "billing_address":  { "$ref": "#/definitions/address" },
    "shipping_address": { "$ref": "#/definitions/address" }
  }
}
```

matches

```
{
  "shipping_address": {
    "street_address": "1600 Pennsylvania Avenue NW",
    "city": "Washington",
    "state": "DC"
  },
  "billing_address": {
    "street_address": "1st Street SE",
    "city": "Washington",
    "state": "DC"
  }
}
```

Recursion:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {
    "person": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "children": {
          "type": "array",
          "items": { "$ref": "#/definitions/person" },
          "default": []
        }
      }
    }
  },
  "type": "object",
  "properties": {
    "person": { "$ref": "#/definitions/person" }
  }
}
```

matches:

```
{
  "person": {
    "name": "Elizabeth",
    "children": [
      {
        "name": "Charles",
        "children": [
          {
            "name": "William",
            "children": [
              { "name": "George" },
              { "name": "Charlotte" }
            ]
          },
          {
            "name": "Harry"
          }
        ]
      }
    ]
  }
}
```

Following would cause a problem (loop??)

```
{
  "definitions": {
    "alice": {
      "anyOf": [
        { "$ref": "#/definitions/bob" }
      ]
    },
    "bob": {
      "anyOf": [
        { "$ref": "#/definitions/alice" }
      ]
    }
  }
}
```