Flatten Tool Documentation

Release 0.0.0

Open Data Services

August 02, 2016

Contents

Caution: This documentation is a work in progress.

Flatten Tool is a Python library and command line interface for converting single or multi-sheet spreadsheets to a JSON document and back again. In Flatten Tool terminology *flattening* is the process of converting a JSON document to spreadsheet sheets, and *unflattening* is the process of converting spreadsheet sheets to a JSON document.

Flatten Tool can make use of a JSON Schema during the flattening and unflattening processes to make sure different types are handled correctly, to support more human-friendly column headings and to give hints about the spreadsheet structure you would like.

Flatten Tool's main use case is to allow people to enter data into a spreadsheet so that it can be converted to a JSON document and validated against a JSON Schema. To support this use case it is very forgiving in what it accepts and prefers to output as much of the input spreadsheet data as it can to be validated by a JSON Schema later, rather than raise errors itself.

Contents:

CHAPTER 1

Introduction

1.1 Why

Imagine a simple dataset that describes grants. Chances are if is to represent the world, it is going to need to contain some one-to-many relationships (.e.g. one grant, many categories). This is structured data.

But, consider two audiences for this dataset:

The developer wants structured data that she can iterate over, one record for each grant, and then the classifications nested inside that record.

The analyst needs flat data - tables that can be sorted, filtered and explored in a spreadsheet.

Which format should the data be published in? Flattten-Tool thinks it should be both.

By introducing a couple of simple rules, Flatten-Tool is aiming to allow data to be round-tripped between JSON and flat formats, sticking to sensible idioms in both flat-land and a structured world.

1.2 How

Flatten-Tool was designed to work along with a JSON Schema. Flatten-Tool likes JSON Schemas which:

(1) Provide an "id" at every level of the structure

So that each entity in the data structure can be referenced easily in the flat version. It turns out this is also pretty useful for JSON-LD mapping.

(2) Describes the ideal root table by rolling up properties

Often in a data structure, there are only a few properties that exist at the root level, with most properties at least one level deep in the structure. However, if Flatten-Tool hides away all the important properties in sub tables, then the spreadsheet user has to hunt all over the place for the properties that matter to them.

So, we introduce a custom 'rollUp' property to out JSON Schema. This allows the schema to specify which relationships and properties should be included in the first table of a spreadsheet.

You can even roll up fields which *could* be one-to-many, but which often will be one-to-one relationships, so that there is a good chance of a user of the flattened data being able to do all the data creation or analysis they want in a single table.

(3) Provide unique field titles

“Recipient Org: Name” is a lot friendlier to spreadsheet users than ‘receipientOrganisation/name’. So, Flatten-Tool includes support for using the titles of JSON fields instead of the field names when creating a spreadsheet template and converting data.

But - to make that use, the titles at each level of the structure do need to be unique.

(4) Don’t nest too deep

Whilst Flatten-Tool can cope with multiple laters of nesting in a data structure, the deeper the structure gets, the trickier it is for the spreadsheet user to understand what is going on. So, we try and just go a few layers deep at most in data for Flatten-Tool to work with.

CHAPTER 2

Examples

2.1 Simple example

The JSON `simple.json`:

```
{
    "main": [
        {
            "a": {
                "b": "1",
                "c": "2"
            },
            "d": "3"
        },
        {
            "a": {
                "b": "4",
                "c": "5"
            },
            "d": "6"
        }
    ]
}
```

Can be converted to/from a spreadsheet like `simple/main.csv`:

a/b	a/c	d
1	2	3
4	5	6

Using the commands:

```
flatten-tool unflatten -f csv examples/simple -o examples/simple.json
flatten-tool flatten -f csv examples/simple.json -o examples/simple
```

2.2 One to many relationships (JSON arrays)

There are multiple shapes of spreadsheet that can be used to produce the same JSON arrays. E.g. to produce `this JSON`:

```
{
    "main": [
        {
            "id": "1",
            "d": "6",
            "a": [
                {
                    "b": "2",
                    "c": "3"
                },
                {
                    "b": "4",
                    "c": "5"
                }
            ]
        },
        {
            "id": "7",
            "d": "12",
            "a": [
                {
                    "b": "8",
                    "c": "9"
                },
                {
                    "b": "10",
                    "c": "11"
                }
            ]
        }
    ]
}
```

We can use these Spreadsheets:

id	a/0/b	a/0/c
1	2	3
1	4	5
7	8	9
7	10	11

id	d
1	6
7	12

These are also the spreadsheets that flatten-tool's *flatten* (JSON to Spreadsheet) will produce.

Commands used to generate this:

```
flatten-tool unflatten -f csv examples/array_multisheet -o examples/array_multisheet.json
flatten-tool flatten -f csv examples/array.json -o examples/array_multisheet
```

However, there are other "shapes" of spreadsheet that can produce the same JSON.

New columns for each item of the array:

id	a/0/b	a/0/c	a/1/b	a/1/c	d
1	2	3	4	5	6
7	8	9	10	11	12

```
flatten-tool unflatten -f csv examples/array_pointer -o examples/array.json
```

Repeated rows:

id	a/0/b	a/0/c	d
1	2	3	6
1	4	5	6
7	8	9	12
7	10	11	12

```
flatten-tool unflatten -f csv examples/array_repeat_rows -o examples/array.json
```

2.3 Arrays within arrays

```
{
    "main": [
        {
            "id": "1",
            "d": "6",
            "a": [
                {
                    "id": "2",
                    "b": [
                        {
                            "c": "3"
                        },
                        {
                            "c": "3a"
                        }
                    ]
                },
                {
                    "id": "4",
                    "b": [
                        {
                            "c": "5"
                        },
                        {
                            "c": "5a"
                        }
                    ]
                }
            ]
        },
        {
            "id": "7",
            "d": "12",
            "a": [
                {
                    "id": "8",
                    "b": [
                        {
                            "c": "9"
                        },
                        {
                            "c": "9a"
                        }
                    ]
                },
                {
                    "id": "10",
                    "b": [
                        {
                            "c": "11"
                        },
                        {
                            "c": "11a"
                        }
                    ]
                }
            ]
        }
    ]
}
```

id	a/0/id
1	2
1	4
7	8
7	10

id	d
1	6
7	12

CHAPTER 3

Getting Started

3.1 Installation

```
git clone https://github.com/OpenDataServices/flatten-tool.git
cd flatten-tool
virtualenv -p $(which python3) .ve
source .ve/bin/activate
pip install -r requirements_dev.txt
```

3.2 Commandline Usage

```
flatten-tool -h
```

will print general help information.

```
flatten-tool {create-template,flatten,unflatten} -h
```

will print help information specific to that subcommand.

3.3 Python Library Usage

```
from flattentool import create_template, flatten, unflatten
```

3.4 Python Version Support

This code supports Python 2.7 and Python 3.3 (and later). Python 3 is strongly preferred. Only servere Python 2 specific bugs will be fixed, see the python2-wontfix label on GitHub for known minor issues.

Python 2.6 and earlier are not supported at all because our code makes use new language constructs introduced in Python 3 and 2.7. Python 3.2 (also 3.1 and 3.0) is not supported, because one of the dependencies (openpyxl) does not support it.

CHAPTER 4

Spreadsheet Designer's Guide

In this guide you'll learn the various rules Flatten Tool uses to convert one or more sheets in a spreadsheet into a JSON document. These rules are documented with examples based around a cafe theme.

Once you've understood how Flatten Tool works you should be able to design your own spreadsheet structures, debug problems in your spreadsheets and be able to make use of Flatten Tool's more advanced features.

Before we get into too much detail though, let's start by looking at the Command Line API for unflattening a spreadsheet.

4.1 Command-Line API

To demonstrate the command line API you'll start with the simplest possible example, a sheet listing Cafe names:

name
Healthy Cafe

We'd like Flatten Tool to convert it to the following JSON structure for an array of cafes, with the cafe name being the only property we want for each cafe:

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

Let's try converting the sheet to the JSON above.

```
$ flatten-tool unflatten -f=csv examples/cafe/too-simple/
```

```
{
    "main": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

That's not too far off what we wanted. You can see the array of cafes, but the key is named *main* instead of *cafe*. You can tell Flatten Tool that that the rows in the spreadsheet are cafes and should come under a *cafe* key by specifying a *root list path*, described next.

Caution: Older Python versions add a trailing space after , characters when outputting indented JSON. This means that your output might have whitespace differences compared to what is described here.

4.1.1 Root List Path

The *root list path* is the key under which Flatten Tool should add an array of objects representing each row of the main sheet.

You specify the root list path with *–root-list-path* option. If you don't specify it, *main* is used as the default as you saw in the last example.

Let's set *–root-list-path* to *cafe* so that our original input generates the JSON we were expecting:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/simple/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

That's what we expected. Great.

Note: Although *–root-list-path* sounds like it accepts a path such as *building/cafe*, it only accepts a single key.

4.1.2 Writing output to a file

By default, Flatten Tool now prints its output to stdout. If you want it to write its output to a file instead, you can use the *-o* option.

Here's the same example, this time writing its output to *unflattened.json*:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe -o=examples/cafe/simple-file/unflattened.json e
$ cat examples/cafe/simple-file/unflattened.json
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

4.1.3 Base JSON

If you want the resulting JSON to also include other keys that you know in advance, you can specify them in a separate *base JSON* file and Flatten Tool will merge the data from your spreadsheet into that file.

For example, if *base.json* looks like this:

```
{
    "country": "England"
}
```

and the data looks like this:

name
Healthy Cafe

you can run this command using the *–base-json* option to see the *base.json* data with the with the spreadsheet rows merged in:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --base-json=examples/cafe/simple-base-json/base
```

```
{
    "country": "England",
    "cafe": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

Warning: If you give the base JSON the same key as you specify in *–root-list-path* then Flatten Tool will overwrite its value.

4.1.4 All unflatten options

You can see all the options available for unflattening by running:

```
$ flatten-tool unflatten -h
```

```
usage: flatten-tool unflatten [-h] -f INPUT_FORMAT [-b BASE_JSON]
                              [-m ROOT_LIST_PATH] [-e ENCODING]
                              [-o OUTPUT_NAME] [-c CELL_SOURCE_MAP]
                              [-a HEADING_SOURCE_MAP]
                              [--timezone-name TIMEZONE_NAME] [-r ROOT_ID]
                              [-s SCHEMA] [--convert-titles]
                              input_name

positional arguments:
  input_name            Name of the input file or directory.

optional arguments:
  -h, --help            show this help message and exit
  -f INPUT_FORMAT, --input-format INPUT_FORMAT
                        File format of input file or directory.
  -b BASE_JSON, --base-json BASE_JSON
                        A base json file to populate with the unflattened
                        data.
  -m ROOT_LIST_PATH, --root-list-path ROOT_LIST_PATH
                        The path in the JSON that will contain the unflattened
                        list. Defaults to main.
  -e ENCODING, --encoding ENCODING
                        Encoding of the input file(s) (only relevant for CSV).
                        This can be any encoding recognised by Python.
                        Defaults to utf8.
  -o OUTPUT_NAME, --output-name OUTPUT_NAME
                        Name of the outputted file. Will have an extension
                        appended as appropriate. Defaults to unflattened.json
  -c CELL_SOURCE_MAP, --cell-source-map CELL_SOURCE_MAP
                        Path to write a cell source map to. Will have an
                        extension appended as appropriate.
  -a HEADING_SOURCE_MAP, --heading-source-map HEADING_SOURCE_MAP
                        Path to write a heading source map to. Will have an
                        extension appended as appropriate.
  --timezone-name TIMEZONE_NAME
                        Name of the timezone, defaults to UTC. Should be in
                        tzdata format, e.g. Europe/London
  -r ROOT_ID, --root-id ROOT_ID
                        Root ID of the data format, e.g. ocid for OCDS
  -s SCHEMA, --schema SCHEMA
                        Path to a relevant schema.
  --convert-titles      Convert titles. Requires a schema to be specified.
```

As you can see, some of the documentation is specific to two projects that use Flatten Tool:

- OCDS - http://standard.open-contracting.org/validator/
- 360Giving - http://www.threesixtygiving.org/standard/reference/

Other options such as *–cell-source-map* and *–heading-source-map* will be described in the Developer Guide once the features stabilise.

4.2 Understanding JSON Pointer and how Flatten Tool uses it

Let’s consider this data again and explore the algorithm Flatten Tool uses to make it work:

name
Healthy Cafe

Here’s a command to unflatten it and the resulting JSON:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/simple/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        }
    ]
}
```

The key to understanding how Flatten Tool represents more complex examples in a spreadsheet lies in knowing about the JSON Pointer specification. This specification describes a fairly intuitive way to reference values in a JSON document.

To breifly describe how it works, each / character after the first one drills down into a JSON strucutre. If they value after the / is a string, then a key is looked up, if it is an integer then an array index is taken.

For example, in the JSON pointer */cafe/0/name* is equivalent to taking the following value out of a JSON document named *document*:

```
>>> document['cafe'][0]['name']
```

In JSON document above, the JSON pointer */cafe/0/name* would return *Healthy Cafe*.

Note: JSON pointer array indexes start at 0, just like lists in Python, hence the first cafe is at index 0.

Whilst JSON pointer is designed as a way for getting data *out* of a JSON document, Flatten Tool uses JSON Pointer as a way of describing how to move values *into* a JSON document from a spreadsheet.

To do this, as it comes across JSON pointers, it automatically creates the objects and arrays required.

You can think of Flatten Tool doing the following as it parses a sheet:

- Load the base JSON or use an empty JSON object
- For each row:
  - Convert each column heading to a JSON pointer by removing whitespace and prepending with */cafe/*, then adding the row index and another */* to the front
  - Take the value in each column and associate it with the JSON pointer (treating any numbers as array indexes, and overwriting existing JSON pointer values for that row if necessary)
  - Write the value into the position in the JSON object being specified by the JSON pointer, creating more structures as you go

In this example there is only one sheet, and only one row, so when parsing that first row, */cafe/0/* is appended to *name* to give the JSON pointer */cafe/0/name*. Flatten Tool then writes *Healthy Cafe* in the correct position.

4.2.1 Index behaviour

There is one subtlety you need to be aware of though before you see some examples.

Although Flatten Tool always uses strings in a JSON pointer as object keys, it only takes numbers it comes across as an *indication* of the array position.

For example, if you gave it the JSON pointer */cafe/1503/name*, there is no guarantee that the *name* would be placed in an object at index position 1503.

Instead Flatten Tool uses numbers in the same sheet that are at the same parent JSON pointer path (*/cafe/* in this case), as being the sort order the child objects should appear in, but not the literal index positions.

If two objects use the same index at the same base JSON pointer path, Flatten Tool will keep both but the one it comes across first will come before the other.

This behaviour has two advantages:

- data won't be lost if for some reason the index wasn't specified correctly
- the data in the generated JSON will be in the same order as it was specified in the sheets which is likely to be what the person putting data into the spreadsheet would expect

This behaviour is also important when you learn about Lists of Objects (without IDs) later.

Tip: You'll see later in the relationships section, that special *id* values can alter the index behavior described here and allow Flatten Tool to merge rows from multiple sheets.

4.2.2 Multiple rows

Let's look at a multi-row example:

name
Healthy Cafe
Vegetarian Cafe

This time *Healthy Cafe* would be placed at */cafe/0/name* and *Vegetarian Cafe* at */cafe/1/name* producing this:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/simple-row/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        },
        {
            "name": "Vegetarian Cafe"
        }
    ]
}
```

Although both *Healthy Cafe* and *Vegetarian Cafe* are under a column that resolves to */cafe/0/name*, the rules described in the previous section explain why noth are present in the output and why *Healthy Cafe* comes before *Vegetarian Cafe*.

4.2.3 Multiple columns

Let's add the cafe address to the spreadsheet:

name	address
Healthy Cafe	123 City Street, London
Vegetarian Cafe	42 Town Road, Bristol

Note: CSV files require cells containing , characters to be escaped by wrapping them in double quotes. That's why if you look at the source CSV, the addresses are escaped with " characters.

This time *Healthy Cafe* is placed at */cafe/0/name* as before, *London* is placed at */cafe/0/address*. *Vegetarian Cafe* at */cafe/1/name* as before and *Bristol* is at */cafe/1/address*.

The result is:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/simple-col/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "address": "123 City Street, London"
        },
        {
            "name": "Vegetarian Cafe",
            "address": "42 Town Road, Bristol"
        }
    ]
}
```

4.2.4 Multiple sheets

So far, all the examples have just used one sheet. When multiple sheets are involved, the behaviour isn't much different.

In effect, all Flatten Tool does is:

- take the JSON structure produced after processing the previous sheets and use it as the base JSON for processing the next sheet
- keep track of the index numbers of existing objects and generate JSON pointers that point to the next free index at any existing locations (with the effect of having new objects appended to any existing ones at the same location)

Once all the sheets have been processed the resulting JSON is returned.

Note: The CSV specification doesn't support multiple sheets. To work around this, Flatten Tool treats a directory of CSV files as a single spreadsheet with multiple sheets - one for each file.

This is why all the CSV file examples given so far have been written to a file in an empty directory and why only the directory name was needed in the *flatten-tool* commands.

Here's a simple two-sheet example where the headings are the same in both sheets:

Table 4.1: sheet: data

name
Healthy Cafe

Table 4.2: sheet: other

name
Vegetarian Cafe

When you run the example you get this:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/multiple/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe"
        },
        {
            "name": "Vegetarian Cafe"
        }
    ]
}
```

The order is because the *data* sheet was processed before the *other* sheet.

Tip: CSV file sheets are processed in the order returned by *os.listdir()* so you should name them in the order you would like them processed.

4.3 Objects

Now you know that the column headings are really just a JSON Pointer specification, and the index values are only treated as indicators of the presence of arrays you can write some more sophisticated examples.

Rather than have the address just as string, we could represent it as an object. For example, imagine you'd like out output JSON in this structure:

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "address": {
                "street": "123 City Street",
                "city": "London"
            }
        },
        {
            "name": "Vegetarian Cafe",
            "address": {
                "street": "42 Town Road",
                "city": "Bristol"
            }
        }
    ]
}
```

You can do this by knowing that the JSON Pointer to "123 City Street" would be */cafe/0/address/street* so that we would need to name the street column *address/street*.

Here's the data:

name	address/street	address/city
Healthy Cafe	123 City Street	London
Vegetarian Cafe	42 Town Road	Bristol

Let's try it:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/object/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "address": {
                "street": "123 City Street",
                "city": "London"
            }
        },
        {
            "name": "Vegetarian Cafe",
            "address": {
                "street": "42 Town Road",
                "city": "Bristol"
            }
        }
    ]
}
```

4.4 Lists of Objects (without IDs)

The cafe's that have made up our examples so far also have tables, and the tables have a table number so that the waiters know where the food has to be taken to.

Each cafe has many tables, so this is an example of a one-to-many relationship if you are used to working with relational databases.

You can represent the table information in JSON as a array of objects, where each object represents a table, and each table has a *number* key. Let's imagine the *Healthy Cafe* has three tables numbered 1, 2 and 3. We'd like to produce this structure:

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "1"
                },
                {
                    "number": "2"
                },
                {
                    "number": "3"
                }
            ]
        }
    ]
}
```

In the relationships section later, we'll see other (often better) ways of arranging this data using *identifiers*, but for now we'll demonstrate an approach that puts all the table information in the same row as the cafe itself.

For example, consider this spreadsheet data:

name	table/0/number	table/1/number	table/2/number
Healthy Cafe	1	2	3

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/list-of-objects/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "1"
                },
                {
                    "number": "2"
                },
                {
                    "number": "3"
                }
            ]
        }
    ]
}
```

We'll use this example of tables (of the furniture variety) in subsequent examples.

4.4.1 Index behaviour

Just as in the multiple sheets example earlier, the exact numbers at the table index positions aren't too important to Flatten Tool. They just tell Flatten Tool that the value in the cell is part of an object in an array.

In this particular case though, Flatten Tool will keep columns in order implied by the indexes.

For example here the index values are such that the lowest number comes last:

name	table/30/number	table/20/number	table/10/number
Healthy Cafe	1	2	3

We'd still expect 3 tables in the output, but we expect Flatten Tool to re-order the columns so that table 3 comes first, then 2, then 1:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/tables-index/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "3"
                },
                {
                    "number": "2"
                },
                {
                    "number": "1"
                }
            ]
        }
    ]
}
```

Child objects like these tables can, of course have more than one key. Let's add a *reserved* key to table number 1 but to try to confuse Flatten Tool, we'll specify it at the end:

name	table/30/number	table/20/number	table/10/number	table/30/reserved
Healthy Cafe	1	2	3	True

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/tables-index-reserved/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "3"
                },
                {
                    "number": "2"
                },
                {
                    "number": "1",
                    "reserved": "True"
                }
            ]
        }
    ]
}
```

Notice that Flatten Tool correctly associated the *reserved* key with table 1 because of the index numbered *30*, even though the columns weren't next to each other.

For a much richer way of organising arrays of objects, see the Relationships section.

4.4.2 Plain Lists (Unsupported)

Flatten Tool doesn't support arrays of JSON values other than objects (just described in the previous section).

As a result heading names such as *tag/0* and *tag/1* would be ignored and an empty array would be put into the JSON.

Here's some example data:

name	tag/0
Healthy Cafe	health
Vegetarian Cafe	veggie

And the result:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/plain-list/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "tag": []
        },
        {
            "name": "Vegetarian Cafe",
            "tag": []
        }
    ]
}
```

4.5 Typed fields

In the table examples you've seen so far, the table numbers are produced as strings in the JSON. The JSON Pointer specification doesn't provide any way of telling you what type the value being pointed to is, so we can't get the information from the column headings.

There are two places we can get it from though:

- The spreadsheet cell (if the underlying spreadsheet type supports it, e.g. CSV doesn't but XLSX does)
- An external JSON Schema describing the data

If we can't get any type information we fall back to assuming strings.

Here is the sample data we'll use for the examples in the next two sections:

name	table/0/number	table/1/number	table/2/number
Healthy Cafe	1	2	3

4.5.1 Using spreadsheet cell formatting

CSV files only support string values, so the easiest way to get the example above to use integers would be to use a spreadsheet format such xlsx that supported integers and make sure the cell type was number. Flatten Tool would pass the cell value through to the JSON as a number in that case.

Note: Make sure you specify the correct format *-f=xlsx* on the command line if you want to use an xlsx file.

```
$ flatten-tool unflatten -f=xlsx --root-list-path=cafe examples/cafe/tables-typed-xlsx/tables.xlsx
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": 1
                },
                {
                    "number": 2
                },
                {
                    "number": 3
                }
            ]
        }
    ]
}
```

Caution: Number formats in spreadsheets are ignored in Python 2.7 so this example won't work. It does work in Python 3.4 and above though.
If you look at Flatten Tool's source code you'll see the in *test_docs.py* that the above example is skipped on older Python versions.

4.5.2 Using a JSON Schema with types

Here's an example of a JSON Schema that can provide the typing information:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "number": {
                    "type": "integer"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "type": "array"
        }
    }
}
```

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --schema=examples/cafe/tables-typed-schema/cafe
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": 1
                },
                {
                    "number": 2
                },
                {
                    "number": 3
                }
            ]
        }
    ]
}
```

Tip: Although this example is too simple to demonstrate it, Flatten Tool ignores the order of individual properties in a JSON schema when producing JSON output, and instead follows the order of the columns in the sheets.

4.6 Human-friendly headings using a JSON Schema with titles

Let's take a closer look at the array of objects example from earlier again:

name	table/0/number	table/1/number	table/2/number
Healthy Cafe	1	2	3

The column headings *table/0/number*, *table/1/number* and *table/2/number* aren't very human readable, wouldn't it be great if we could use headings like this:

name	Table: 0: Number	Table: 1: Number	Table: 2: Number
Healthy Cafe	1	2	3

Flatten Tool supports this if you do the following:

- Write a JSON Schema specifying the titles being used and specify it with the *–schema* option
- Use *:* characters instead of */* characters in the headings
- Specify the *–convert-titles* option on the command line

Caution: If you forget any of these, Flatten Tool might produce incorrect JSON rather than failing.

Here's a new JSON schema for this example:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "number": {
                    "title": "Number",
                    "type": "integer"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "title": "Table",
            "type": "array"
        }
    }
}
```

Notice that both *Table* and *Number* are specified as titles.

Here's what we get when we run it:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --schema=examples/cafe/tables-human-1/cafe.sche
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": 1
                },
                {
                    "number": 2
                },
                {
                    "number": 3
                }
            ]
        }
    ]
}
```

4.6.1 Optional array indexes

Looking at the JSON Schema from the last example again you'll see that *table* is specified as an array type:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "number": {
                    "title": "Number",
                    "type": "integer"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "title": "Table",
            "type": "array"
        }
    }
}
```

This means that Flatten Tool can work out that any names specified in that column are part of that array. If you had an example with just one column representing each level of the tree, you could miss out the index in the heading when using *–schema* and *–convert-titles*.

Here’s a similar example, but with just one rolled up column:

name	Table: Number
Healthy Cafe	1

Here’s what we get when we run this new data with this schema:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --schema=examples/cafe/tables-human-2/cafe.sche
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": 1
                }
            ]
        }
    ]
}
```

4.7 Relationships using Identifiers

So far, all the examples you’ve seen have served to demonstrate how Flatten Tool works, but probably wouldn’t be particularly useful in real life, simply because they require everything related to be on the same row.

In this section you’ll learn how identifiers work and that will allow you much more freedom in designing different spreadsheet layouts that produce the same JSON.

In Flatten Tool, any field named *id* is considered special. Flatten Tool knows that any objects with the same *id* at the same level are the same object and that their values should be merged.

4.7.1 ID-based object merge behaviour

The merge behaviour happens whether the two IDs are specified in:

- different rows in the same sheet
- two rows in two different sheets

Basically, any time Flatten Tool comes across a row with an *id* in it, it will lookup any other objects in the array to see if that *id* is already used and if it is, it will merge it. If not, it will just append a new object to the array.

Caution: It is important to make sure your *id* values really are unique. If you accidentally use the same *id* for two different objects, Flatten Tool will think they are the same and merge them.

Flatten Tool will merge an existing and new object as follows:

- Any fields in new object that are missing in the existing one are added
- Any fields in the existing object that aren't in the new one are left as they are
- If there are fields that are in both that have the same value, that value is kept
- If there are fields that are in both with different values, the existing values are kept and conflict warnings issued

This means that values in later rows do not overwrite existing conflicting values.

Let's have a look at these rules in action in the next two sections with an example from a single sheet, and one from multiple sheets.

ID-based object merge in a single sheet

Here's an example that demonstrates these rules:

id	name	address	number_of_tables
CAFE-HEALTH	Healthy Cafe		
CAFE-HEALTH	Vegetarian Cafe		3
CAFE-HEALTH		123 City Street, London	
CAFE-HEALTH			4

Let's run it:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-merge-single/
```

Notice the warnings above about values being over-written:

```
... UserWarning: Conflict when merging field "name" for id "CAFE-HEALTH" in sheet data: "Healthy Cafe
  key, id_info, debug_info.get('sheet_name'), base_value, value))
... UserWarning: Conflict when merging field "number_of_tables" for id "CAFE-HEALTH" in sheet data: '
  key, id_info, debug_info.get('sheet_name'), base_value, value))
```

The actual JSON contains a single Cafe with *id* value *CAFE-HEALTH* and all the values merged in:

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "number_of_tables": "3",
            "address": "123 City Street, London"
        }
    ]
}
```

ID-based object merge in multiple sheets

Here's an example that uses the same data as the single sheet example above, but spreads the rows over four sheets named *a*, *b*, *c* and *d*:

Table 4.3: sheet: a

id	name	address	number_of_tables
CAFE-HEALTH	Healthy Cafe		

Table 4.4: sheet: b

id	name	address	number_of_tables
CAFE-HEALTH	Incorrect value		3

Table 4.5: sheet: c

id	name	address	number_of_tables
CAFE-HEALTH		123 City Street, London	

Table 4.6: sheet: d

id	name	address	number_of_tables
CAFE-HEALTH			4

Let's run it:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-merge-multiple/
```

Notice the warnings above about values being over-written:

```
... UserWarning: Conflict when merging field "name" for id "CAFE-HEALTH" in sheet b: "Healthy Cafe"
  key, id_info, debug_info.get('sheet_name'), base_value, value))
... UserWarning: Conflict when merging field "number_of_tables" for id "CAFE-HEALTH" in sheet d: "3"
  key, id_info, debug_info.get('sheet_name'), base_value, value))
```

And the rest of the output:

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "number_of_tables": "3",
            "address": "123 City Street, London"
        }
    ]
}
```

The result is the same as before.

4.7.2 Parent-child relationships (arrays of objects)

Things get much more interesting when you start dealing with arrays of objects whose parents have an *id*. This enables you to split the parents and children up into multiple sheets rather than requiring everything sits one the same row.

As an example, let's imagine that *Vegetarian Cafe* is arranged having two tables numbered *16* and *17* because they are share tables with another restaurant next door.

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "1"
                },
                {
                    "number": "2"
                },
                {
                    "number": "3"
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "name": "Vegetarian Cafe",
            "table": [
                {
                    "number": "16"
                },
                {
                    "number": "17"
                }
            ]
        }
    ]
}
```

From the knowledge you gained when learning about arrays of objects without IDs earlier, you know that you can produce the correct structure with a CSV file like this:

Table 4.7: sheet: cafes

id	name	table/0/number	table/1/number	table/2/number
CAFE-HEALTH	Healthy Cafe	1	2	3
CAFE-VEG	Vegetarian Cafe	16	17	

This time, we'll give both the Cafe's IDs and move the tables into a separate sheet:

Table 4.8: sheet: cafes

id	name
CAFE-HEALTH	Healthy Cafe
CAFE-VEG	Vegetarian Cafe

Table 4.9: sheet: tables

id	table/0/number
CAFE-HEALTH	1
CAFE-VEG	16
CAFE-HEALTH	2
CAFE-HEALTH	3
CAFE-VEG	17

By having the tables in a separate sheet, you can now support cafe's with as many tables as you like, just by adding more rows and making sure the *id* column for the table matches the *id* value for the cafe.

Let's run this example:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-lists-of-objects/
```

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "number": "1"
                },
                {
                    "number": "2"
                },
                {
                    "number": "3"
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "name": "Vegetarian Cafe",
            "table": [
                {
                    "number": "16"
                },
                {
                    "number": "17"
                }
            ]
        }
    ]
}
```

By specifying an ID, the values in the tables sheet can be associated with the correct part of the tree created by the cafes sheet.

4.7.3 Index behaviour

Within the array of tables for each cafe, you might have noticed that each table number has a JSON Pointer that ends in with */0/number*. Since they all have the same index, they are simply ordered within each cafe in the order of the rows in the sheet.

4.7.4 Grandchild relationships

In future we might like to extend this example so that we can track the dishes ordered by each table so we can generate a bill.

Let's take the case of dishes served at tables and imagine that *Healthy Cafe* has its own health *fish and chips* dish. Now let's also imagine that the dish is ordered at tables 1 and 3.

If you are used to thinking about relational database you would probably think about having a new sheet called *dishes* with a two columns, one for an *id* and one for the *name* of the dish. You would then create a sheet to represent a join table called *table_dishes* that contained the ID of the table and of the dish.

The problem with this approach is that the output is actually a tree, and not a normalised relational model. Have a think about how you would write the *table_dishes* sheet. You'd need to write something like this:

table/0/id	dish/0/id
TABLE-1	DISH-fish-and-chips
TABLE-3	DISH-fish-and-chips

The problem is that *dish/0/id* is really a JSON Pointer to */cafe/0/dish/0/id* and so would try to create a new *dish* key under each *cafe*, not a *dish* key under each *table*.

You can't do it this way. Instead you have to design you *dish* sheet to specify both the ID of the cafe and the ID of the table as well as the name of the dish. If a dish is used in multiple tables, you will have multiple rows, each with the same name in the name column. In this each way row contains the entire path to its position in the tree.

Since nothing depends on the dishes yet, they don't have to have an ID themselves, they just need to reference their parent IDs:

Table 4.10: sheet: cafes

id	name
CAFE-HEALTH	Healthy Cafe

Table 4.11: sheet: tables

id	table/0/id	table/0/number
CAFE-HEALTH	TABLE-1	1
CAFE-HEALTH	TABLE-2	2
CAFE-HEALTH	TABLE-3	3

Table 4.12: sheet: dishes

id	table/0/id	table/0/dish/0/name
CAFE-HEALTH	TABLE-1	Fish and Chips
CAFE-HEALTH	TABLE-3	Fish and Chips

Here are the results:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-multiple/
```

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "id": "TABLE-1",
                    "dish": [
                        {
                            "name": "Fish and Chips"
                        }
                    ],
                    "number": "1"
                },
                {
                    "id": "TABLE-3",
                    "dish": [
                        {
                            "name": "Fish and Chips"
                        }
                    ],
                    "number": "3"
                },
                {
                    "id": "TABLE-2",
                    "number": "2"
                }
            ]
        }
    ]
}
```

Notice the ordering in this example. Because *dishes* is processed before *tables*, *TABLE-3* gets defined before *TABLE-2*, and *dish* gets added as a key before *tables*.

If the sheets were processed the other way around the data would be the same, but the ordering different.

Tip: Flatten Tool supports producing JSON hierarchies of arbitrary depth, not just the parent-child and parent-child-grandchild relationships you've seen in the examples so far. Just make sure that however deep an object is, it always has the IDs of *all* of its parents in the same row as it, as the tables and dishes sheets do.

Arbitrary-depth in a single table

You can also structure all the data into a single table. It is only recommended to do this if you have a very simple data structure where there is only one object at each part of the hierarchy.

In this example we'll use a JSON Schema to infer the structure, allowing us to use human-readable column titles.

Here's the data:

Table 4.13: sheet: dishes

Identifier	Name	Table: Identifier	Table: Number	Table: Dish: Name
CAFE-HEALTH	Healthy Cafe	TABLE-1	1	Fish and Chips
CAFE-HEALTH	Healthy Cafe	TABLE-2	2	
CAFE-HEALTH	Healthy Cafe	TABLE-3	3	Fish and Chips

Let's unflatten this table:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-grandchild/ -s examp
```

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "id": "TABLE-1",
                    "number": 1,
                    "dish": [
                        {
                            "name": "Fish and Chips"
                        }
                    ]
                },
                {
                    "id": "TABLE-2",
                    "number": 2
                },
                {
                    "id": "TABLE-3",
                    "number": 3,
                    "dish": [
                        {
                            "name": "Fish and Chips"
                        }
                    ]
                }
            ]
        }
    ]
}
```

If you'd like to explore this example yourself, here's the schema used in the example above:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "number": {
                    "type": "integer",
                    "title": "Number"
                },
                "dish": {
                    "items": {
                        "$ref": "#/definitions/DishObject"
                    },
                    "type": "array",
                    "title": "Dish"
                }
            }
        },
        "DishObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "name": {
                    "type": "string",
                    "title": "Name"
                },
                "cost": {
                    "type": "number",
                    "title": "Cost"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "title": "Identifier"
        },
        "name": {
            "type": "string",
            "title": "Name"
        },
        "address": {
            "type": "string",
            "title": "Address"
        },
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "type": "array",
            "title": "Table"
        }
    }
}
```

4.7.5 Missing IDs

You might be wondering what happens if IDs are accidentally missing. There are two cases where this can happen:

- The ID is missing but no child objects reference it anyway
- The ID is missing and so children can't be added

To demonstrate both of these in one example consider the following example. In particular notice that:

- *CAFE-VEG* is missing from the *cafes* sheet
- *CAFE-VEG* is missing from the last row in the *tables* sheet

Table 4.14: sheet: cafes

id	name	address
CAFE-HEALTH	Healthy Cafe	123 City Street, London
	Vegetarian Cafe	42 Town Road, Bristol

Table 4.15: sheet: tables

id	table/0/number
CAFE-HEALTH	1
CAFE-HEALTH	2
CAFE-HEALTH	3
CAFE-VEG	16
	17

Let's run this example:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/relationship-missing-ids/
```

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "address": "123 City Street, London",
            "table": [
                {
                    "number": "1"
                },
                {
                    "number": "2"
                },
                {
                    "number": "3"
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "table": [
                {
                    "number": "16"
                }
            ]
        },
        {
            "name": "Vegetarian Cafe",
            "address": "42 Town Road, Bristol"
        },
        {
            "table": [
                {
                    "number": "17"
                }
            ]
        }
    ]
}
```

You'll notice that all the data and tables for *CAFE-HEALTH* are output correctly in the first object. This is what we'd expect because all the IDs were present.

```
{
    "id": "CAFE-HEALTH",
    "name": "Healthy Cafe",
    "address": "123 City Street, London",
    "table": [
        {
            "number": "1"
        },
        {
            "number": "2"
        },
        {
            "number": "3"
        }
    ]
},
```

Next is this cafe:

```
{
    "name": "Vegetarian Cafe",
    "address": "42 Town Road, Bristol"
},
```

This is as much information as Flatten Tool can work out from the second row of the *cafes* sheet because the ID is missing. Flatten Tool just appends a new cafe with the data it has.

Next, Flatten Tool works through the *tables* sheet, it finds table 16 and knows it must be associated with a cafe called *CAFE-VEG* that is specified in the *id* column, but because this *id* is present in the *cafes* sheet, it can't merge it in. Instead it just appends data for the cafe:

```
{
    "id": "CAFE-VEG",
    "table": [
        {
            "number": "16"
        }
    ]
},
```

Finally, Flatten Tool finds table 17 in the *tables* sheet. It doesn't know which Cafe this is for, but it knows tables are part of cafes so it adds another unnamed cafe:

```
{
    "table": [
        {
            "number": "17"
        }
    ]
}
```

4.7.6 Relationships with JSON Schema

If you want to use Flatten Tool's support for JSON Schema to extend to relationships you need to amend your JSON Schema to tell it about the *id* fields:

1. Make sure that the *id* field is specified for every object in the hierarchy (although this isn't necessary for objects right at the bottom of the hierarchy)
2. Give the *id* field a title of *Identifier*

With these two things in place, Flatten Tool will correctly handle relationships.

Caution: If you forget to add the *id* field, Flatten Tool will not know anything about it when generating templates or converting titles.

4.8 Sheet Shapes

Now that you've seen some of the details of how Flatten Tool works we can look in more detail at the different shapes your data can have in the sheets.

To discuss the pros and cons of the different shapes, we'll work through a whole example.

Imagine that you Healthy Cafe and Vegetarian Cafe are both part of a chain and you have to create a receipt system for them. You need to track which dishes are ordered at which tables in which cafes.

The JSON you would like to produce from the sheets the waiters write as they take orders looks like this:

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "id": "TABLE-1",
                    "number": "1",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        },
                        {
                            "name": "Pesto Pasta Salad",
                            "cost": "6.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-2",
                    "number": "2"
                },
                {
                    "id": "TABLE-3",
                    "number": "3",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        }
                    ]
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "name": "Vegetarian Cafe",
            "table": [
                {
                    "id": "TABLE-16",
                    "number": "16",
                    "dish": [
                        {
                            "name": "Large Glass Sauvignon",
                            "cost": "5.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-17",
                    "number": "17"
                }
            ]
        }
    ]
}
```

There are many ways we could arrange this data:

- cafes, tables and dishes all separate
- cafes and tables together, dishes separate, with one row per table in the cafes and tables sheet
- cafes and tables together, dishes separate, with one row per cafe in the cafes and tables sheet
- tables and dishes together, cafes separate, with one row per table in the tables and dishes sheet
- tables and dishes together, cafes separate, with one row per dish in the tables and dishes sheet

Let's take a look at the first three cases. Combining tables and dishes into one sheet follows the same principles as combining cafes and tables, so we won't demonstrate those examples too.

4.8.1 Separate sheet for each object

Here's the first way of doing this with everything in its own sheet. This is the recommended approach unless you have a good reason to move some parts of a table into another one. It is also the default you will get when using Flatten Tool to flatten or generate a template for a JSON structure. You'll learn about that later.

Table 4.16: Sheet: 1_cafes

id	name
CAFE-HEALTH	Healthy Cafe
CAFE-VEG	Vegetarian Cafe

Table 4.17: Sheet: 2_tables

id	table/0/id	table/0/number
CAFE-HEALTH	TABLE-1	1
CAFE-HEALTH	TABLE-2	2
CAFE-HEALTH	TABLE-3	3
CAFE-VEG	TABLE-16	16
CAFE-VEG	TABLE-17	17

Table 4.18: Sheet: 3_dishes

id	table/0/id	table/0/dish/0/name	table/0/dish/0/cost
CAFE-HEALTH	TABLE-1	Fish and Chips	9.95
CAFE-HEALTH	TABLE-1	Pesto Pasta Salad	6.95
CAFE-HEALTH	TABLE-3	Fish and Chips	9.95
CAFE-VEG	TABLE-16	Large Glass Sauvignon	5.95

Note: Notice that this time the CSV sheets are prefixed with an integer to make sure they are processed in the right order. If the prefixes weren't there, the order of the tables in the resulting JSON might be different.

If you were using an XLSX file, Flatten Tool would process the sheets in the order they appeared, regardless of their names, so the prefix wouldn't be needed.

You can run the example with this:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/receipt/normalised/
```

You should see the same JSON as shown at the top of the section.

The advantage of this set up is that it allows any number of cafes, tables and dishes. The disadvantage is that it requires three sheets, making data a bit harder to find.

4.8.2 Combining objects

Now let's imagine that all your cafe's are small and they never have more than three tables. In this case we can combine tables into cafes so that we just have two sheets.

Table per row

Here's what it looks like when you want to use one row per table:

Table 4.19: Sheet: cafes and tables

id	name	table/0/id	table/0/number
CAFE-HEALTH	Healthy Cafe	TABLE-1	1
CAFE-HEALTH	Healthy Cafe	TABLE-2	2
CAFE-HEALTH	Healthy Cafe	TABLE-3	3
CAFE-VEG	Vegetarian Cafe	TABLE-16	16
CAFE-VEG	Vegetarian Cafe	TABLE-17	17

Table 4.20: Sheet: dishes

id	table/0/id	table/0/dish/0/name	table/0/dish/0/cost
CAFE-HEALTH	TABLE-1	Fish and Chips	9.95
CAFE-HEALTH	TABLE-1	Pesto Pasta Salad	6.95
CAFE-HEALTH	TABLE-3	Fish and Chips	9.95
CAFE-VEG	TABLE-16	Large Glass Sauvignon	5.95

You can run the example with this:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/receipt/combine-table-into-cafe/
```

If you do run it you'll see the JSON is exactly the same as before.

Unlike a database, Flatten Tool won't complain if different Cafe names are associated with the same Cafe ID in the same table, instead you'll just get a warning.

Combining sheets works best when:

- the child (the object being combined in to the parent) doesn't have that many properties
- you can be sure there won't be too many children for each parent
- there is a low risk of typos being made in the duplicated data

Cafe per row

There's another variant of this shape that we can use. If we just want to use one row per cafe.

Table 4.21: Sheet: cafes and tables

id	name	table/0/id	table/0/number	table/1/id	table/1/number	table/2/id	table/2/number
CAFE-HEALTH	Healthy Cafe	TABLE-1	1	TABLE-2	2	TABLE-3	3
CAFE-VEG	Vegetarian Cafe	TABLE-16	16	TABLE-17	17		

Table 4.22: Sheet: dishes

id	table/0/id	table/0/dish/0/name	table/0/dish/0/cost
CAFE-HEALTH	TABLE-1	Fish and Chips	9.95
CAFE-HEALTH	TABLE-1	Pesto Pasta Salad	6.95
CAFE-HEALTH	TABLE-3	Fish and Chips	9.95
CAFE-VEG	TABLE-16	Large Glass Sauvignon	5.95

You can run the example with this:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/receipt/combine-table-into-cafe-2/
```

The JSON is the same as before, as you would expect.

All in one table

It would also be possible to put all the data in a single table, but this would look quite complicated since there is more than one table in each cafe and more than one dish at each table.

To understand the approach, have a look at the "Arbitrary-depth in a single table" section earlier.

Tip: If you'd like to explore these examples yourself using human-readable column titles, you can use the schema in the "Arbitrary-depth in a single table" section too.

4.9 Source maps

Once you have unflattened a spreadsheet into a JSON document you will usually pass the document to a JSON Schema validator to make sure all the data is valid.

If there are any errors in the JSON, it is very useful to be able to point the user back to the corresponding place in the original spreadsheet. Flatten Tool provides *source maps* for exactly this purpose.

There are two types of source map:

- Cell source map - points from a JSON pointer path to a cell (or row) in the original spreadsheet
- Heading source map - specifies the column for each heading

Here's an example where we unflatten a normalised spreadsheet, but generate both a cell and a heading source map as we do.

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --cell-source-map examples/receipt/source-map/a
```

Here's the source data:

Table 4.23: sheet: 1_cafes.csv

id	name
CAFE-HEALTH	Healthy Cafe
CAFE-VEG	Vegetarian Cafe

Table 4.24: sheet: 2_tables.csv

id	table/0/id	table/0/number
CAFE-HEALTH	TABLE-1	1
CAFE-HEALTH	TABLE-2	2
CAFE-HEALTH	TABLE-3	3
CAFE-VEG	TABLE-16	16
CAFE-VEG	TABLE-17	17

Table 4.25: sheet: 3_dishes.csv

id	table/0/id	table/0/dish/0/name	table/0/dish/0/cost
CAFE-HEALTH	TABLE-1	Fish and Chips	9.95
CAFE-HEALTH	TABLE-1	Pesto Pasta Salad	6.95
CAFE-HEALTH	TABLE-3	Fish and Chips	9.95
CAFE-VEG	TABLE-16	Large Glass Sauvignon	5.95

Here's the resulting JSON document (the same as before):

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "id": "TABLE-1",
                    "number": "1",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        },
                        {
                            "name": "Pesto Pasta Salad",
                            "cost": "6.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-2",
                    "number": "2"
                },
                {
                    "id": "TABLE-3",
                    "number": "3",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        }
                    ]
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "name": "Vegetarian Cafe",
            "table": [
                {
                    "id": "TABLE-16",
                    "number": "16",
                    "dish": [
                        {
                            "name": "Large Glass Sauvignon",
                            "cost": "5.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-17",
                    "number": "17"
                }
            ]
        }
    ]
}
```

Let's look in detail at the cell source map and heading source map for this example.

4.9.1 Cell source map

A cell source map maps each JSON pointer in the document above back to the cells where that value is referenced.

Using the example you've just seen, let's look at the very last value in the spreadsheet for the number of *TABLE-17* in *CAFE-VEG*. The JSON pointer is *cafe/1/table/1/number* and the value itself is *17*.

Looking back at the source sheets you can see the only place this value appears is in *2_tables.csv*. It appears in column C (the third column), row 6 (row 1 is treated as the heading so the values start at row 2). The heading of this column in *table/0/number* (which happens to be a JSON pointer, but if we were using human readable headings, those headings would be used instead). We'd therefore expect the cell source map to have just one entry for *cafe/1/table/1/number* that points to cell *C2* like this:

```
"cafe/1/table/1/number": [
    [
        "2_tables",
        "C",
        6,
        "table/0/number"
    ]
],
```

Here's the actual cell source map and as you can see, the entry for *cafe/1/table/1/number* is as we expect (it is near the end):

```
{
    "cafe/0/id": [
        [
            "1_cafes",
            "A",
            2,
            "id"
        ],
        [
            "2_tables",
            "A",
            2,
            "id"
        ],
        [
            "2_tables",
            "A",
            3,
            "id"
        ],
        [
            "2_tables",
            "A",
            4,
            "id"
        ],
        [
            "3_dishes",
            "A",
            2,
            "id"
        ],
        [
            "3_dishes",
            "A",
            3,
            "id"
        ],
        [
            "3_dishes",
            "A",
            4,
            "id"
        ]
    ],
    "cafe/0/name": [
        [
            "1_cafes",
            "B",
            2,
            "name"
        ]
    ],
    "cafe/0/table/0/dish/0/cost": [
        [
            "3_dishes",
            "D",
            2,
            "table/0/dish/0/cost"
        ]
    ],
    "cafe/0/table/0/dish/0/name": [
        [
            "3_dishes",
            "C",
            2,
            "table/0/dish/0/name"
        ]
    ],
    "cafe/0/table/0/dish/1/cost": [
        [
            "3_dishes",
            "D",
            3,
            "table/0/dish/0/cost"
        ]
    ],
    "cafe/0/table/0/dish/1/name": [
        [
            "3_dishes",
            "C",
            3,
            "table/0/dish/0/name"
        ]
    ],
    "cafe/0/table/0/id": [
        [
            "2_tables",
            "B",
            2,
            "table/0/id"
        ],
        [
            "3_dishes",
            "B",
            2,
            "table/0/id"
        ],
        [
            "3_dishes",
            "B",
            3,
            "table/0/id"
        ]
    ],
    "cafe/0/table/0/number": [
        [
            "2_tables",
            "C",
            2,
            "table/0/number"
        ]
    ],
    "cafe/0/table/1/id": [
        [
            "2_tables",
            "B",
            3,
            "table/0/id"
        ]
    ],
    "cafe/0/table/1/number": [
        [
            "2_tables",
            "C",
            3,
            "table/0/number"
        ]
    ],
    "cafe/0/table/2/dish/0/cost": [
        [
            "3_dishes",
            "D",
            4,
            "table/0/dish/0/cost"
        ]
    ],
    "cafe/0/table/2/dish/0/name": [
        [
            "3_dishes",
            "C",
            4,
            "table/0/dish/0/name"
        ]
    ],
    "cafe/0/table/2/id": [
        [
            "2_tables",
            "B",
            4,
            "table/0/id"
        ],
        [
            "3_dishes",
            "B",
            4,
            "table/0/id"
        ]
    ],
    "cafe/0/table/2/number": [
        [
            "2_tables",
            "C",
            4,
            "table/0/number"
        ]
    ],
    "cafe/1/id": [
        [
            "1_cafes",
            "A",
            3,
            "id"
        ],
        [
            "2_tables",
            "A",
            5,
            "id"
        ],
        [
            "2_tables",
            "A",
            6,
            "id"
        ],
        [
            "3_dishes",
            "A",
            5,
            "id"
        ]
    ],
    "cafe/1/name": [
        [
            "1_cafes",
            "B",
            3,
            "name"
        ]
    ],
    "cafe/1/table/0/dish/0/cost": [
        [
            "3_dishes",
            "D",
            5,
            "table/0/dish/0/cost"
        ]
    ],
    "cafe/1/table/0/dish/0/name": [
        [
            "3_dishes",
            "C",
            5,
            "table/0/dish/0/name"
        ]
    ],
    "cafe/1/table/0/id": [
        [
            "2_tables",
            "B",
            5,
            "table/0/id"
        ],
        [
            "3_dishes",
            "B",
            5,
            "table/0/id"
        ]
    ],
    "cafe/1/table/0/number": [
        [
            "2_tables",
            "C",
            5,
            "table/0/number"
        ]
    ],
    "cafe/1/table/1/id": [
        [
            "2_tables",
            "B",
            6,
            "table/0/id"
        ]
    ],
    "cafe/1/table/1/number": [
        [
            "2_tables",
            "C",
            6,
            "table/0/number"
        ]
    ],
    "cafe/0": [
        [
            "1_cafes",
            2
        ],
        [
            "2_tables",
            2
        ],
        [
            "2_tables",
            3
        ],
        [
            "2_tables",
            4
        ],
        [
            "3_dishes",
            2
        ],
        [
            "3_dishes",
            3
        ],
        [
            "3_dishes",
            4
        ]
    ],
    "cafe/0/table/0/dish/0": [
        [
            "3_dishes",
            2
        ]
    ],
    "cafe/0/table/0/dish/1": [
        [
            "3_dishes",
            3
        ]
    ],
    "cafe/0/table/0": [
        [
            "2_tables",
            2
        ],
        [
            "3_dishes",
            2
        ],
        [
            "3_dishes",
            3
        ]
    ],
    "cafe/0/table/1": [
        [
            "2_tables",
            3
        ]
    ],
    "cafe/0/table/2/dish/0": [
        [
            "3_dishes",
            4
        ]
    ],
    "cafe/0/table/2": [
        [
            "2_tables",
            4
        ],
        [
            "3_dishes",
            4
        ]
    ],
    "cafe/1": [
        [
            "1_cafes",
            3
        ],
        [
            "2_tables",
            5
        ],
        [
            "2_tables",
            6
        ],
        [
            "3_dishes",
            5
        ]
    ],
    "cafe/1/table/0/dish/0": [
        [
            "3_dishes",
            5
        ]
    ],
    "cafe/1/table/0": [
        [
            "2_tables",
            5
        ],
        [
            "3_dishes",
            5
        ]
    ],
    "cafe/1/table/1": [
        [
            "2_tables",
            6
        ]
    ]
}
```

You'll notice that some JSON pointers map to multiple source cells. This happens when data appears in multiple places, such as when the cell refers to an identifier.

You'll also notice that after all the JSON pointers that point to values such as *cafe/0/id* or *cafe/1/table/1/number* there are a set of JSON pointers that point to objects rather than cells. For example *cafe/0* or *cafe/1/table/1*. These JSON pointers refer back to the rows which contain values that make up the object. For example *cafe/1/table/1* looks like this:

```
"cafe/1/table/1": [
    [
        "2_tables",
        6
    ]
]
```

This tells us that the data that makes up that table in the final JSON was all defined in the *2_tables* sheet, row 6 (remembering that rows start at 2 because the header row is row 1). Again, if data from multiple rows goes to make up the object, there may be multiple arrays in the JSON pointer result.

This second kind of entry in the cell source map is useful when a JSON schema validator gives errors to describe a missing value since it is likely that you will need to add the value on one for the rows where the other values are defined.

4.9.2 Heading source map

The heading source map maps a JSON pointer with all numbers removed, back to the column heading at the top of the columns where corresponding values have been placed.

Here's the heading source map that was generated in the example we've been using in this section:

```
{
    "cafe/id": [
        [
            "1_cafes",
            "id"
        ],
        [
            "2_tables",
            "id"
        ],
        [
            "3_dishes",
            "id"
        ]
    ],
    "cafe/name": [
        [
            "1_cafes",
            "name"
        ]
    ],
    "cafe/table/dish/cost": [
        [
            "3_dishes",
            "table/0/dish/0/cost"
        ]
    ],
    "cafe/table/dish/name": [
        [
            "3_dishes",
            "table/0/dish/0/name"
        ]
    ],
    "cafe/table/id": [
        [
            "2_tables",
            "table/0/id"
        ],
        [
            "3_dishes",
            "table/0/id"
        ]
    ],
    "cafe/table/number": [
        [
            "2_tables",
            "table/0/number"
        ]
    ]
}
```

The heading source map is generated separately from the cell source map, so headings can be found even if they have no corresponding data in the resulting JSON.

Creating Templates

So far, all the examples you've seen have been about unflattening - taking a spreadsheet and producing a JSON document.

If you already have a JSON schema, Flatten Tool can automatically create a template spreadsheet with the correct headers that you can start filling in.

Flatten Tool's sub-command for this is *flatten-tool create-template*.

5.1 Generating a spreadsheet template from a JSON Schema

Here's an example command that uses a schema from the cafe example under *Sheet shapes* and generates a spreadsheet:

```
$ flatten-tool create-template --use-titles --main-sheet-name=cafe --schema=examples/receipt/cafe.sch
```

The example uses *–use-titles* so that the generated spreadsheet has human readable titles and *–main-sheet-name=cafe* so that the generated spreadsheet have *cafe* as their first tab and not the default, *main*.

If you don't specify *-o*, Flatten Tool will choose a spreadsheet called *template* in the current working directory.

If you don't specify a format with *-f*, Flatten Tool will create a *template.xlsx* file and a set of CSV files under *template/*.

The schema is the same as the one used in the user guide and looks like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "number": {
                    "type": "integer",
                    "title": "Number"
                },
                "dish": {
                    "items": {
                        "$ref": "#/definitions/DishObject"
                    },
                    "type": "array",
                    "title": "Dish"
                }
            }
        },
        "DishObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "name": {
                    "type": "string",
                    "title": "Name"
                },
                "cost": {
                    "type": "number",
                    "title": "Cost"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "title": "Identifier"
        },
        "name": {
            "type": "string",
            "title": "Name"
        },
        "address": {
            "type": "string",
            "title": "Address"
        },
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "type": "array",
            "title": "Table"
        }
    }
}
```

If you run the example above, Flatten Tool will generate the following CSV files for you:

Table 5.1: sheet: cafe.csv

Identifier	Name	Address

Table 5.2: sheet: table.csv

Identifier	Table:Identifier	Table:Number

Table 5.3: sheet: tab_dish.csv

Identifier	Table:Identifier	Table:Dish:Identifier	Table:Dish:Name	Table:Dish:Cost

As you can see, by default Flatten Tool puts each item with an Identifier in its own sheet.

5.1.1 Rolling up

If you have a JSON schema where objects are modeled as lists of objects but actually represent one to one relationships, you can *roll up* certain properties.

This means taking the values and rather than having them as a separate sheet, have the values listed on the main sheet.

To enable roll up behaviour you have to:

- Use the *–rollup* flag
- Add the *rollUp* key to the JSON Schema to the child object with a value that is an array of the fields to roll up

Here are the changes we make to the schema:

```
--- ../examples/receipt/cafe.schema
+++ ../examples/receipt/cafe-rollup.schema
@@ -58,7 +58,8 @@
                 "$ref": "#/definitions/TableObject"
             },
             "type": "array",
-            "title": "Table"
+            "title": "Table",
+            "rollUp": ["number"]
         }
     }
 }
```

Here's the command we run:

```
$ flatten-tool create-template --use-titles --main-sheet-name=cafe --schema=examples/receipt/cafe-ro
```

Here are the resulting sheets:

Table 5.4: sheet: cafe.csv

Identifier	Name	Address	Table:Number

Table 5.5: sheet: table.csv

Identifier	Table:Identifier	Table:Number

Table 5.6: sheet: tab_dish.csv

Identifier	Table:Identifier	Table:Dish:Identifier	Table:Dish:Name	Table:Dish:Cost

Notice how *Table: Number* has now been moved into the *cafe.csv* file.

Caution: If you try to roll up multiple values you'll get a warning like this:

```
UserWarning: More than one value supplied for "table". Could not provide rollup, so adding a warnin
  warn('More than one value supplied for "{}". Could not provide rollup, so adding a warning to the
```

5.1.2 Empty objects

If you have a JSON schema where an object's only property is an array represented by another sheet, Flatten Tool will generate an empty sheet for the object so that you can still add columns at a later date.

5.1.3 All create-template options

```
$ flatten-tool create-template -h
```

```
usage: flatten-tool create-template [-h] -s SCHEMA [-f OUTPUT_FORMAT]
                                    [-m MAIN_SHEET_NAME] [-o OUTPUT_NAME]
                                    [--rollup] [-r ROOT_ID] [--use-titles]

optional arguments:
  -h, --help            show this help message and exit
  -s SCHEMA, --schema SCHEMA
                        Path to the schema file you want to use to create the
                        template
  -f OUTPUT_FORMAT, --output-format OUTPUT_FORMAT
                        Type of template you want to create. Defaults to all
                        available options
  -m MAIN_SHEET_NAME, --main-sheet-name MAIN_SHEET_NAME
                        The name of the main sheet, as seen in the first tab
                        of the spreadsheet for example. Defaults to main
  -o OUTPUT_NAME, --output-name OUTPUT_NAME
                        Name of the outputted file. Will have an extension
                        appended if format is all.
  --rollup              "Roll up" columns from subsheets into the main sheet
                        if they are specified in a rollUp attribute in the
                        schema.
  -r ROOT_ID, --root-id ROOT_ID
                        Root ID of the data format, e.g. ocid for OCDS
  --use-titles          Convert titles. Requires a schema to be specified.
```

CHAPTER 6

Flattening

Caution: This page is a work in progress. The information may not be complete, and unlike the Spreadsheet Designer's Guide, the tests backing up the documented examples are not correct. Use with caution.

So far, all the examples you've seen have been about unflattening - taking a spreadsheet and producing a JSON document.

In this section you'll learn about flattening. The main use case for wanting to flatten a JSON document is so that you can manage the data in a spreadsheet from now on.

Flatten Tool provides *flatten-tool flatten* sub-command for this purpose.

6.1 Generating a spreadsheet from a JSON document

Generating a spreadsheet from a JSON document is very similar to creating a template.

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --schema=examples/receipt/cafe.sc
```

One difference is that the default output name is *flatten*, and so the command above will generate a *flatten/* directory with CSV files and a *flatten.xlsx* file in the current working directory.

The schema is the same as the one used in the user guide and looks like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "TableObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "number": {
                    "type": "integer",
                    "title": "Number"
                },
                "dish": {
                    "items": {
                        "$ref": "#/definitions/DishObject"
                    },
                    "type": "array",
                    "title": "Dish"
                }
            }
        },
        "DishObject": {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "Identifier"
                },
                "name": {
                    "type": "string",
                    "title": "Name"
                },
                "cost": {
                    "type": "number",
                    "title": "Cost"
                }
            }
        }
    },
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "title": "Identifier"
        },
        "name": {
            "type": "string",
            "title": "Name"
        },
        "address": {
            "type": "string",
            "title": "Address"
        },
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "type": "array",
            "title": "Table"
        }
    }
}
```

The input JSON file looks like this:

```
{
    "cafe": [
        {
            "id": "CAFE-HEALTH",
            "name": "Healthy Cafe",
            "table": [
                {
                    "id": "TABLE-1",
                    "number": "1",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        },
                        {
                            "name": "Pesto Pasta Salad",
                            "cost": "6.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-2",
                    "number": "2"
                },
                {
                    "id": "TABLE-3",
                    "number": "3",
                    "dish": [
                        {
                            "name": "Fish and Chips",
                            "cost": "9.95"
                        }
                    ]
                }
            ]
        },
        {
            "id": "CAFE-VEG",
            "name": "Vegetarian Cafe",
            "table": [
                {
                    "id": "TABLE-16",
                    "number": "16",
                    "dish": [
                        {
                            "name": "Large Glass Sauvignon",
                            "cost": "5.95"
                        }
                    ]
                },
                {
                    "id": "TABLE-17",
                    "number": "17"
                }
            ]
        }
    ]
}
```

If you run the example above, Flatten Tool will generate the following CSV files for you, populated with the data from the input JSON file.

Warning: You can't use *–use-titles* with flatten.

Table 6.1: sheet: cafe.csv

id	name	address
CAFE-HEALTH	Healthy Cafe	
CAFE-VEG	Vegetarian Cafe	

Table 6.2: sheet: table.csv

id	table/0/id	table/0/number
CAFE-HEALTH	TABLE-1	1
CAFE-HEALTH	TABLE-2	2
CAFE-HEALTH	TABLE-3	3
CAFE-VEG	TABLE-16	16
CAFE-VEG	TABLE-17	17

Table 6.3: sheet: tab_dish.csv

id	table/0/id	table/0/dish/0/id	table/0/dish/0/name	table/0/dish/0/cost

Table 6.4: sheet: dish.csv

id	table/0/id	table/0/dish/0/name	table/0/dish/0/cost
CAFE-HEALTH	TABLE-1	Fish and Chips	9.95
CAFE-HEALTH	TABLE-1	Pesto Pasta Salad	6.95
CAFE-HEALTH	TABLE-3	Fish and Chips	9.95
CAFE-VEG	TABLE-16	Large Glass Sauvignon	5.95

Caution: If you forget the *–root-list-path* option and your data isn't under a top level key called *main*, Flatten Tool won't find your data and will instead generate empty a single empty sheet called *main*, which probably isn't what you want.

6.1.1 All flatten options

```
$ flatten-tool flatten -h
```

```
usage: flatten-tool flatten [-h] [-s SCHEMA] [-f OUTPUT_FORMAT]
                            [-m MAIN_SHEET_NAME] [-o OUTPUT_NAME]
                            [--root-list-path ROOT_LIST_PATH] [--rollup]
                            [-r ROOT_ID] [--use-titles]
                            input_name

positional arguments:
  input_name            Name of the input JSON file.

optional arguments:
  -h, --help            show this help message and exit
  -s SCHEMA, --schema SCHEMA
                        Path to a relevant schema.
  -f OUTPUT_FORMAT, --output-format OUTPUT_FORMAT
                        Type of template you want to create. Defaults to all
                        available options
  -m MAIN_SHEET_NAME, --main-sheet-name MAIN_SHEET_NAME
                        The name of the main sheet, as seen in the first tab
                        of the spreadsheet for example. Defaults to main
  -o OUTPUT_NAME, --output-name OUTPUT_NAME
                        Name of the outputted file. Will have an extension
                        appended if format is all.
  --root-list-path ROOT_LIST_PATH
                        Path of the root list, defaults to main
  --rollup              "Roll up" columns from subsheets into the main sheet
                        if they are specified in a rollUp attribute in the
                        schema.
  -r ROOT_ID, --root-id ROOT_ID
                        Root ID of the data format, e.g. ocid for OCDS
  --use-titles          Convert titles. Requires a schema to be specified.
```

CHAPTER 7

Developer Guide

The primary use case for Flatten Tool is to convert spreadsheets to JSON so that the data can be validated using a JSON Schema.

Flatten Tool has to be very forgiving in what it accepts so that it can deal with spreadsheets that are a work-in-progress. It tries its best to make sense of what you give it, even if you give it inconsistent, conflicting or patchy data. It leaves the work of reporting problems to the JSON Schema validator that will be run on the JSON it produces, and it only generates warnings if it is forced to ignore data from the source spreadsheet.

Flatten Tool tries its best to output as much as it can so the JSON it produces will be as good or bad as the spreadsheet input it receives. The benefit of this approach that the user can be shown all the problems in one go when the JSON Schema validator is run on that JSON.

Programming a very forgiving tool that tries to accept lots of categories of errors is a lot more complex than programming a tool where the data structures are very predictable. Understanding this intention not to raise errors is key to understanding Flatten Tool's internal design.

7.1 Helper libraries

As you'll have read in the User Guide, Flatten Tool makes use of JSON Pointer, JSON Schema and JSON Ref standards. The Python libraries that support this are *jsonpointer*, *jsonschema* and *jsonref* respectively.

7.2 Running the tests

After following the installation above, run `py.test`.

Note that the tests require the Python testsuite. This should come with python, but some distros split it out. On Ubuntu you will need to install a package like `libpython3.4-testsuite` (depending on which Python version you are using).

7.3 Testing coverage of documentation examples

```
rm -f .coverage # Remove the old coverage if it exists
python flattentool/tests/test_docs.py
coverage combine
coverage report --omit=flattentool/tests/**
```

7.4 What's coming up

7.4.1 Three layer design

The codebase will be refactored so that the unflatten part of the library comes in three parts:

Spreadsheet Loaders

> Responsible for loading data out of spreadsheets and representing it in the correct format for the unflattener - a Python structure of basic JSON types and the special *Empty* value

Unflatten function

> Takes the Python data structure described above and unflattens it, using a JSON Schema if present and keeping all state explicit.
>
> Use the JSON Schema to convert any basic JSON types to richer types that can be correctly serailised by a serialiser later (e.g. dates). Returns a cell tree.
>
> **Tip:** Take a look at the *run()* function in *flattentool/tests/test_headings.py* to see a function that behaves a little like a pure Python entry point to Flatten Tool's functionality.

Serialisers

> Take a cell tree and serialise it to either a JSON tree, a source map, or both

This pattern will make it easier to support testing the core unflatten function, as well as making it easier to support future spreadsheet and serialiser formats.

7.4.2 Explicit float support

The existing implementation makes a special effort to correctly handle decimal types such as currency.

This special effort also means that Flatten Tool treats float values as *Decimal* too.

Most of the time this is perfectly fine, since Python correctly treats a *Decimal* generated from a float as being equal to the float itself:

```
>>> from decimal import Decimal
>>> Decimal(1.3) == 1.3
True
```

Do be aware of this small quirk of Python's behaviour though. Python doesn't treat a *Decimal* obtained from *'1.3'* as being the same as one generated from *1.3*:

```
>>> Decimal('1.3') == Decimal(1.3)
False
>>> Decimal(1.3)
Decimal('1.3000000000000000444089209850062616169452667236328125')
```

7.4.3 Stdin support

The next version could support a single sheet being fed into *stdin* like this:

```
cat << EOF | flatten-tool unflatten -f=csv --root-list-path=cafe
name,
Healthy Cafe,
EOF
```

7.4.4 More documentation

- Flattening, roll up and template creation
- Timezone support
- Using Flatten Tool as a library
- Source maps

7.4.5 Naming and Versioning

The next release of Flatten Tool will likely start a version numbering schema. We could also name the command line tool *flattentool* rather than *flatten-tool* so that everything is consistent.

7.4.6 Other possible directions

It might be also be good to add a *CHANGELOG.txt* which could document changes such as:

- This documentation
- Changed stdout behaviour for unflatten and loss of the default - writing to *unflattened.json*.
- Publishing on PyPi

CHAPTER 8

Flatten-Tool for OCDS

The Open Contracting Data Standard (OCDS) has an unofficial CSV serialization that can be converted to/from the canonical JSON form using Flatten-Tool.

8.1 Templates

A comprehensive spreadsheet template for OCDS can be downloaded from https://github.com/open-contracting/sample-data/tree/master/flat-template - this is generated directly from the OCDS schema with the commands listed in *Creating spreadsheet templates* below. [link to http://flatten-tool.readthedocs.io/en/latest/create-template/ ?]

There are multiple shapes of spreadsheet that would produce the same valid OCDS data... [It is possible to custom design CSV or spreadsheet templates that can be used to provide valid OCDS data (Linking to Spreadsheet Designers Guide).]

It's also possible to add additional fields..

8.2 Web interface

Flatten-Tool is integrated into the Open Contracting Data Standard Validator, an online tool for validating and converting OCDS files.

This supports XLSX, but currently only supports uploading CSV (and only one CSV file). [clarify this]

8.3 Commandline Usage

8.3.1 Converting a JSON file to a spreadsheet

```
flatten-tool flatten input.json --root-id=ocid --main-sheet-name releases --output-name flattened --
```

This will command will create an output called flattened in all the formats we support - currently this is `flattened.xlsx` and a `flattened/` directory of csv files.

See `flatten-tool flatten --help` for details of the commandline options.

8.3.2 Converting a populated spreadsheet to JSON

```
cp base.json.example base.json
```

And populate this with the package information for your release.

Then, for a populated xlsx template in (in release_populated.xlsx):

```
flatten-tool unflatten release_populated.xlsx --root-id=ocid --base-json base.json --input-format xls
```

Or for populated CSV files (in the release_populated directory):

```
flatten-tool unflatten release_populated --root-id=ocid --base-json base.json --input-format csv --ou
```

These produce a release.json file based on the data in the spreadsheets.

See `flatten-tool unflatten --help` for details of the commandline options.

8.3.3 Creating spreadsheet templates

[link to http://flatten-tool.readthedocs.io/en/latest/create-template/]

Download https://raw.githubusercontent.com/open-contracting/standard/1.0/standard/schema/release-schema.json to the current directory.

```
flatten-tool create-template --root-id=ocid --output-format all --output-name template --schema relea
```

This will create *template.xlsx* and a *template/* directory of csv files.

See `flatten-tool create-template --help` for details of the commandline options.

8.4 Python Library Usage

```
from flattentool import create_template, flatten, unflatten
```

Flatten-Tool for 360Giving

You can also upload the file to http://cove.opendataservices.coop/360

Download https://raw.githubusercontent.com/ThreeSixtyGiving/standard/master/schema/360-giving-schema.json to the current directory.

```
flatten-tool create-template --output-format all --output-name 360giving-template --schema 360-giving
```

```
flatten-tool unflatten -o out.json -f xlsx input.xlsx --schema 360-giving-schema.json --convert-title
```

Get started by reading the Spreadsheet Designer's Guide to understand the core concepts, how to use the *flatten-tool* command and how to structure your own data as spreadsheet sheets.

The Developer Guide (work in progress) will go into more detail about how Flatten Tool works internally, how you can use it as a library and how you can generate *source maps* that locate each value in a JSON document back to the sheet and cell it came from in a source spreadsheet. Source maps are handy for notifying users where they can go in their source spreadsheet to correct any errors.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search