Flatten Tool Documentation

Release 0.0.0

Open Data Services

Jul 21, 2021

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? Flatten 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 'recipientOrganization/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 layers 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"
```

(continues on next page)

(continued from previous page)


```
                        },
                        {
                            "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 Prerequisites

You will need Python 3.6 or later, including the venv module.

Generally the venv module should come with your default Python install, but not on Ubuntu. On Ubuntu run:

```
sudo apt-get install python3 python3-venv
```

3.2 Installation

```
git clone https://github.com/OpenDataServices/flatten-tool.git
cd flatten-tool
python3 -m venv .ve
source .ve/bin/activate
pip install -r requirements_dev.txt
```

3.3 Usage

```
flatten-tool -h
```

will print general help information.

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

will print help information specific to that sub-command.

3.4 Python Version Support

This code supports Python 3.6 (and later).

Python 3.5 and earlier (including Python 2) are not supported, because they are end of life, and some of the dependencies do not support them.

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/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 examples/cafe/simple/ --root-list-path=cafe
```

```
{
    "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 The root is a list

You can also specify the data outputted is just a list, using the `--root-is-list` option.

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

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

4.1.3 Writing output to a file

By default, Flatten Tool prints its output to standard output (your terminal). If you want it to write its output to a file instead, you can use the `--output-name` option (or `-o` for short).

```
$ flatten-tool unflatten -f=csv examples/cafe/simple/ -o=examples/cafe/simple-file/
↪unflattened.json
$ cat examples/cafe/simple-file/unflattened.json
```

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

4.1.4 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 spreadsheet rows merged in:

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

```
{
    "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.

4.1.5 All unflatten options

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

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

```
usage: flatten-tool unflatten [-h] -f {csv,ods,xlsx} [--xml] [--id-name ID_NAME]
                              [-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]
                              [--vertical-orientation]
                              [--metatab-name METATAB_NAME]
                              [--metatab-schema METATAB_SCHEMA]
                              [--metatab-only]
                              [--metatab-vertical-orientation]
                              [--xml-schema [XML_SCHEMA [XML_SCHEMA ...]]]
                              [--default-configuration DEFAULT_CONFIGURATION]
                              [--root-is-list] [--disable-local-refs]
                              [--xml-comment XML_COMMENT]
                              input_name

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

optional arguments:
  -h, --help            show this help message and exit
  -f {csv,ods,xlsx}, --input-format {csv,ods,xlsx}
                        File format of input file or directory.
  --xml                 Use XML as the output format
  --id-name ID_NAME     String to use for the identifier key, defaults to 'id'
  -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.
  -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
```

(continues on next page)

(continued from previous page)

```
                        Path to a relevant schema.
  --convert-titles      Convert titles. Requires a schema to be specified.
  --vertical-orientation
                        Read spreadsheet so that headings are in the first
                        column and data is read vertically. Only for XLSX not
                        CSV
  --metatab-name METATAB_NAME
                        If supplied will assume there is a metadata tab with
                        the given name
  --metatab-schema METATAB_SCHEMA
                        The jsonschema of the metadata tab
  --metatab-only        Parse the metatab and nothing else
  --metatab-vertical-orientation
                        Read metatab so that headings are in the first column
                        and data is read vertically. Only for XLSX not CSV
  --xml-schema [XML_SCHEMA [XML_SCHEMA ...]]
                        Path to one or more XML schemas (used for sorting)
  --default-configuration DEFAULT_CONFIGURATION
                        Comma separated list of default parsing commands for
                        all sheets. Only for XLSX not CSV
  --root-is-list        The root element is a list. --root-list-path and meta
                        data will be ignored.
  --disable-local-refs  Disable local refs when parsing JSON Schema.
  --xml-comment XML_COMMENT
                        String comment of what generates the xml file
```

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 examples/cafe/simple/ --root-list-path=cafe
```

```
{
    "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 briefly describe how it works, each `/` character after the first one drills down into a JSON structure. If the 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, 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 the 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 both 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 1: sheet: data

name
Healthy Cafe

Table 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"
                },
```

(continues on next page)

(continued from previous page)

```
                {
                    "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 recognise arrays of JSON values other than objects (just described in the previous section) unless a schema is used.

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": []
        }
    ]
}
```

However, an array of tags in the following format (semi-colon separated) can be handled if a schema is passed to Flatten Tool specifying the array type of the field.

name	tags
Healthy Cafe	health;low-cost;locally sourced food;take-out
Vegetarian Cafe	veggie

The schema we'll pass is:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "tags": {
            "items": {
                "type": "string"
            },
            "type": "array"
        },
        "name": {
                  "type": "string"
        }
    }
}
```

And the result:

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe --schema=examples/cafe/plain-
↪list-schema/tagsArraySchema.json examples/cafe/plain-list-schema/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "tags": [
                "health",
                "low-cost",
                "locally sourced food",
                "take-out"
```

(continues on next page)

(continued from previous page)

```
            ]
        },
        {
            "name": "Vegetarian Cafe",
            "tags": [
                "veggie"
            ]
        }
    ]
}
```

Read on for more about typed fields and use of schemas.

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 as 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
                },
```

(continues on next page)

(continued from previous page)

```
                {
                    "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.schema examples/cafe/tables-typed-schema/
```

```
{
    "cafe": [
        {
            "name": "Healthy Cafe",
            "table": [
                {
```

(continues on next page)

(continued from previous page)

```
                    "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",
```

(continues on next page)

(continued from previous page)

```
                    "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.schema --convert-titles examples/cafe/tables-human-1/
```

```
{
    "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",
```

(continues on next page)

(continued from previous page)

```
                    "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.schema --convert-titles examples/cafe/tables-human-2/
```

```
{
    "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.

4.7.1.1 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:

```
You may have a duplicate Identifier: We couldn't merge these rows with the id "CAFE-
↪HEALTH": field "name" in sheet "data": one cell has the value: "Healthy Cafe", the␣
↪other cell has the value: "Vegetarian Cafe"
You may have a duplicate Identifier: We couldn't merge these rows with the id "CAFE-
↪HEALTH": field "number_of_tables" in sheet "data": one cell has the value: "3", the␣
↪other cell has the value: "4"
```

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"
        }
    ]
}
```

4.7.1.2 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 3: sheet: a

id	name	address	number_of_tables
CAFE-HEALTH	Healthy Cafe		

Table 4: sheet: b

id	name	address	number_of_tables
CAFE-HEALTH	Incorrect value		3

Table 5: sheet: c

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

Table 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:

```
You may have a duplicate Identifier: We couldn't merge these rows with the id "CAFE-
↪HEALTH": field "name" in sheet "b": one cell has the value: "Healthy Cafe", the␣
↪other cell has the value: "Incorrect value"
You may have a duplicate Identifier: We couldn't merge these rows with the id "CAFE-
↪HEALTH": field "number_of_tables" in sheet "d": one cell has the value: "3", the␣
↪other cell has the value: "4"
```

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 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 8: sheet: cafes

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

Table 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"
                },
```

(continues on next page)

(continued from previous page)

```
                {
                    "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 10: sheet: cafes

id	name
CAFE-HEALTH	Healthy Cafe

Table 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 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.

4.7.4.1 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 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 examples/receipt/cafe.schema --convert-titles
```


```
{
    "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"
                        }
                    ]
                }
            ]
```


(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

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"
        },
```

(continues on next page)

(continued from previous page)

```
        "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 14: sheet: cafes

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

Table 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": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "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:

```
{
```

(continues on next page)

(continued from previous page)

```
    "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"
```


(continues on next page)

(continued from previous page)

```
                    }
                ]
            }
        ]
}
```

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 16: Sheet: 1_cafes

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

Table 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 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.

4.8.2.1 Table per row

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

Table 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 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

4.8.2.2 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 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 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.

4.8.2.3 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 Metadata Tab

Flatten Tool supports naming of a special sheet (or Tab) in a spreadsheet to add data to the top level of the returned data structure. Currently it only supports output format JSON and the input format has to be XLSX.

4.9.1 Example Usage

You have a spreadsheet named "mydata.xlsx with" 2 sheets. The first sheet named "Cafe":

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

A second sheet you would like to add some metadata to this list of rows to a sheet named "Meta":

dataLicense	CC
rowCount	2
publishedDate	2001-01-01

As you can see it is also possible to choose to have the metadata headings on the first column (not the first row) with metadata vertical.

The command for doing this:

```
$ flatten-tool unflatten --input-format=xlsx --metatab-name Meta --metatab-vertical-
↪orientation examples/cafe/meta-tab/meta-tab.xlsx
```

```
{
    "dataLicense": "CC",
    "rowCount": "2",
    "publishedDate": "2001-01-01",
    "main": [
        {
            "name": "Healthy Cafe",
            "address": "123 City Street, London"
        },
        {
            "name": "Vegetarian Cafe",
            "address": "42 Town Road, Bristol"
        }
    ]
}
```

4.9.2 Options

`--metatab-name`

This is the name of the sheet with the metadata on. It is case sensitive. It is the only mandatory option if you want to parse a metatab, without it no metatab will be parsed

`--metatab-schema`

The JSON schema of the metatab. This schema will be used to determine the types and/or titles of the data in the metatab. It works in the same way as the –schema option but just for the metatab. The schema used with the –schema option has no effect on the metatab parsing, so this has to be specified if you need title handling or want to specify types.

`--metatab-only`

Just return the metatab information and not the rest of the doc. Using the example above:

```
$ flatten-tool unflatten --input-format=xlsx --metatab-only --metatab-name Meta --
↪metatab-vertical-orientation examples/cafe/meta-tab-only/meta-tab.xlsx
```

```
{
    "dataLicense": "CC",
    "rowCount": "2",
    "publishedDate": "2001-01-01"
}
```

`--metatab-vertical-orientation`

Say that the metatab data runs vertically rather that horizontally see example above.

4.10 Configuration properties: skip and header rows

Flatten Tool supports directives in the first row of a file to tell it to:

- **skiprows** - start processing data from n rows down
- **headerrows** - the total number of header rows. Note that the first header row will be treated as field paths.

4.10.1 Example usage

You have a CSV file named "mydata.csv" that contains:

- Two rows of general provenance information or notes;
- The field paths;
- Two rows that explain the meaning of the fields

This pattern may occur, for example, when you export from a spreadsheet that includes formatted header rows that explain the data.

By adding a row containing a cell with '#', and then a set of configuration directives, you can instruct Flatten Tool to skip rows at the top of the file, and to recognise that the field paths are followed by a set of additional header lines.

#	skiprows 2	headerrows 3
This row is skipped	It could contain some provenance data	
		This row is skipped too
name	address/street	address/city
Name	Street Address	City address
Use name from the sign	Don't include the postcode	
Healthy Cafe	123 City Street	London
Vegetarian Cafe	42 Town Road	Bristol

Flatten tool will interpret the '#' configuration row, and generate the appropriate output with no additional parameters needed at the command line.

```
$ flatten-tool unflatten -f=csv --root-list-path=cafe examples/cafe/skip-and-headers/
```

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

(continues on next page)

(continued from previous page)

```
                "city": "Bristol"
            }
        }
    ]
}
```

4.10.2 List of configuration features

There is also the 'ignore' command, which can be used to ignore sheets in a multi-tab workbook.

When configuration options are set in metatab, they apply to all sheets unless they are overridden.

Further configuration options can be seen at https://github.com/OpenDataServices/flatten-tool/blob/7fa96933b8fc3ba07a3d44fe07dccf2791165686/flattentool/lib.py

4.11 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/actual/cell_source_map.json --heading-source-map examples/
↪receipt/source-map/actual/heading_source_map.json examples/receipt/source-map/input/
```

Here's the source data:

Table 23: sheet: 1_cafes.csv

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

Table 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 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"
```


(continues on next page)

(continued from previous page)

```
                        }
                    ]
                },
                {
                    "id": "TABLE-17",
                    "number": "17"
                }
            ]
        }
    ]
}
```

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

4.11.1 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",
```

(continues on next page)

(continued from previous page)


```
            "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",
```


(continues on next page)

(continued from previous page)


```
            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": [
```


(continues on next page)

(continued from previous page)

```
        [
            "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"
        ],
        [
```

(continues on next page)

(continued from previous page)


```
            "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,
```


(continues on next page)

(continued from previous page)


```
            "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
```


(continues on next page)

(continued from previous page)

```
        ],
        [
            "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": [
```

(continues on next page)

(continued from previous page)

```
        [
            "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.

4.11.2 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",
```

(continues on next page)

(continued from previous page)

```
            "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.

4.11.3 XML Comment

When a file is unflatten to an xml file, there is a default comment that says:

XML generated by flatten-tool

This comment can be edited by using the unflatten argument xml_comment or cli command –xml-comment.

CHAPTER 5

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.schema -f csv -o examples/create-template/simple/actual
```

The example uses `--use-titles` so that the generated spreadsheet has human readable titles and `--main-sheet-name=cafe` so that the generated spreadsheet has `cafe` as its 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",
```

(continues on next page)

(continued from previous page)

```
                    "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 1: sheet: cafe.csv

Identifier	Name	Address

Table 2: sheet: Table.csv

Identifier	Table:Identifier	Table:Number

Table 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, creating column headings for them 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:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/flatten-tool/checkouts/latest/
↪examples/receipt/cafe.schema
+++ /home/docs/checkouts/readthedocs.org/user_builds/flatten-tool/checkouts/latest/
↪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-rollup.schema --rollup -f csv -o examples/create-template/rollup/actual
```

Here are the resulting sheets:

Table 4: sheet: cafe.csv

Identifier	Name	Address	Table:Number

Table 5: sheet: Table.csv

Identifier	Table:Identifier	Table:Number

Table 6: sheet: Tab_Dish.csv

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

Notice how `Table: Number` now appears in both the `Cafe.csv` and `Table.csv` files.

`rollup` can also be used to rollup actual data with `flatten`, with or without a schema: *Flattening*

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 Disable local refs

You can pass a `--disable-local-refs` flag for a special mode that will disable local refs in JSON Schema files.

```
$ flatten-tool create-template --disable-local-refs --schema=examples/create-template/
↪refs-disable-local-refs/main.json -f csv -o examples/create-template/refs-disable-
↪local-refs/actual
```

You may want to do this if running the command against JSON Schema files you don't trust.

5.1.4 Deprecated Fields

Fields which are deprecated can be marked in the JSON schema by setting the "deprecated" key to a value.

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

(continues on next page)

(continued from previous page)

```
        },
        "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"
        },
        "formalname": {
            "type": "string",
            "title": "Formal Name",
            "deprecated": {
                "description": "We found people prefer to be addressed by their nick␣
↪names",
                "deprecatedVersion": "v17"
            }
        },
        "address": {
            "type": "string",
            "title": "Address"
        },
        "table": {
            "items": {
                "$ref": "#/definitions/TableObject"
            },
            "type": "array",
            "title": "Table"
        },
        "coats": {
            "deprecated": true,
            "items": {
                "type": "object",
                "properties": {
                    "description": {
                        "type": "string",
                        "title": "Description"
```

(continues on next page)

(continued from previous page)

```
                    }
                }
            },
            "type": "array",
            "title": "Coats"
        }
    }
}
```

You can then choose whether deprecated fields appear in the output by passing the `--no-deprecated-fields` option. This is off by default.

```
$ flatten-tool create-template --use-titles --main-sheet-name=cafe --schema=examples/
↪create-template/deprecated.schema -f csv -o examples/create-template/deprecated-yes/
↪actual
```

```
$ flatten-tool create-template --no-deprecated-fields --use-titles --main-sheet-
↪name=cafe --schema=examples/create-template/deprecated.schema -f csv -o examples/
↪create-template/deprecated-no/actual
```

5.1.5 All create-template options

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

```
usage: flatten-tool create-template [-h] -s SCHEMA [-f {csv,ods,xlsx,all}]
                                    [-m MAIN_SHEET_NAME] [-o OUTPUT_NAME]
                                    [--rollup] [-r ROOT_ID] [--use-titles]
                                    [--disable-local-refs]
                                    [--no-deprecated-fields]
                                    [--truncation-length TRUNCATION_LENGTH]

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 {csv,ods,xlsx,all}, --output-format {csv,ods,xlsx,all}
                        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.
  --disable-local-refs  Disable local refs when parsing JSON Schema.
  --no-deprecated-fields
                        Exclude Fields marked as deprecated in the JSON
```

(continues on next page)

(continued from previous page)

```
                        Schema.
  --truncation-length TRUNCATION_LENGTH
                        The length of components of sub-sheet names (default
                        3).
```

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.

The *Spreadsheet Designer's Guide* was 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.

Flatten Tool provides the `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.schema examples/receipt/normalised/expected.json -o examples/flatten/
↪simple/actual
```

One difference is that the default output name is `flattened`, and so the command above will generate a `flattened/` directory with CSV files and a `flattened.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",
```

(continues on next page)

(continued from previous page)

```
                    "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"
                }
            ]
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

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.

Table 1: sheet: cafe.csv

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

Table 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 3: sheet: tab_dish.csv

id	table/0/id	table/0/dish/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 a single empty sheet called `main`, which probably isn't what you want.

If your data has a list as a root, use the `--root-is-list` option.

```
[
  {
    "id": "shop",
    "title": "Shop"
  },
  {
    "id": "pub",
    "title": "Pub"
  }
]
```

```
$ flatten-tool flatten --root-is-list  examples/flatten/root-is-list/data.json -o␣
↪examples/flatten/root-is-list/actual
```

Table 4: sheet: main.csv

id	title
shop	Shop
pub	Pub

6.1.1 Sheet Prefix

You can pass a string that will be added at the start of all CSV file names, or all Excel sheet names, using the `--sheet-prefix` option.

```
$ flatten-tool flatten --sheet-prefix=test- --root-list-path=cafe --main-sheet-
↪name=cafe --schema=examples/receipt/cafe.schema examples/receipt/normalised/
↪expected.json -o examples/flatten/sheet-prefix/actual
```

Will produce:

Table 5: sheet: test-cafe.csv

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

Table 6: sheet: test-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 7: sheet: test-tab_dish.csv

id	table/0/id	table/0/dish/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

6.1.2 Filter

When flattening, you can optionally choose to only process some of the data.

Currently, only simple filters can be specified using the `--filter-field` and `--filter-value` option.

```
{
  "main": [
    {
      "id": "1",
      "type": "cafe",
      "title": "Cafe Open",
```

(continues on next page)

(continued from previous page)

```
      "dishes": [
        { "title": "All Day Breakfast" }
      ],
      "coffee": "passable"
    },
    {
      "id": "2",
      "type": "pub",
      "title": "Pints R Us",
      "pints": [
        { "title": "Beer" },
        { "title": "Larger" }
      ],
      "coke": "pepsi"
    }
  ]
}
```

```
$ flatten-tool flatten --filter-field=type --filter-value=pub examples/flatten/filter/
↪input.json -o examples/flatten/filter/actual
```

Table 8: sheet: main.csv

id	type	title	coke
2	pub	Pints R Us	pepsi

Table 9: sheet: pints.csv

id	pints/0/title
2	Beer
2	Larger

No `dishes` sheet is produced, and the main sheet does not have a `coffee` column.

The field specified must be a field directly on the data object - it's not possible to filter on fields like `pints/0/title` .

6.1.3 Remove Empty Schema Columns

By default, when using this with *schema* specified columns that are empty (have no data) will be kept in the output.

But you can pass the *remove-empty-schema-columns* flag to have these removed.

If all columns are removed from a sheet and it is empty, the whole sheet will be removed too.

This shows without and with the flag:

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --schema=examples/
↪receipt/cafe.schema examples/receipt/normalised/expected.json -o examples/flatten/
↪simple/actual
```

Table 10: sheet: cafe.csv

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

```
$ flatten-tool flatten --remove-empty-schema-columns --root-list-path=cafe --main-
↪sheet-name=cafe --schema=examples/receipt/cafe.schema examples/flatten/remove-empty-
↪schema-columns/input.json -o examples/flatten/remove-empty-schema-columns/actual
```

Table 11: sheet: cafe.csv

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

(Using this without the *schema* option does nothing.)

6.1.4 Preserve Fields

By default, all fields in the input JSON are turned into columns in the CSV output. If you wish to keep only a subset of fields, you can pass the fields you want as a file using the `preserve-fields` option.

```
{
  "main": [
    {
      "id": "1",
      "type": "cafe",
      "title": "Cafe Open",
      "dishes": [
        {
          "title": "All Day Breakfast",
          "allergens": {
            "freefrom": ["dairy", "eggs"],
            "contains": ["gluten", "soy"],
            "label": ["vegan", "vegetarian"]
          }
        },
        {
          "title": "Jack's Soup",
          "allergens": {
            "freefrom": ["onion", "garlic"],
            "contains": ["dairy"],
            "label": ["vegetarian"]
          }
        }
      ],
      "coffee": "passable"
    },
    {
      "id": "2",
      "type": "pub",
      "title": "Pints R Us",
      "pints": [
        { "title": "Beer", "price": "2gbp" },
        { "title": "Larger" }
      ],
      "coke": "pepsi"
    }
  ]
}
```

```
id
type
title
pints
coke
dishes/title
dishes/price
dishes/allergens/label
```

```
$ flatten-tool flatten --preserve-fields examples/flatten/preserve-fields/fields_to_
↪preserve.csv examples/flatten/preserve-fields/input.json -o examples/flatten/
↪preserve-fields/actual
```

Table 12: sheet: main.csv

id	type	title	coke
1	cafe	Cafe Open	
2	pub	Pints R Us	pepsi

The input file should contain the full JSON paths of the fields you want to preserve, one per line. If any of the fields passed contain objects, all child fields will be preserved. Eg. if you pass *title*, the top level *title* fields will be preserved, but *dishes/title* will not; if you pass *dishes*, the *dishes/title* and any other children of *dishes* will automatically be preserved. If you pass *dishes and dishes/title*, *only dishes/title* will be preserved, other children of *dishes* will be excluded. The order of the fields in the input is not significant.

6.1.5 Rollup

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

And do one of:

- Via schema: 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
- Direct input: Pass one or more field at the command line
- File input: Pass a file with a line-separated list of fields

If you pass direct input or file input, *and* a schema which contains a `rollUp` attribute, the schema is used and the direct input or file input are ignored.

However, if you pass direct or file input, *and* and a schema which *does not* contain a `rollUp` attribute, the direct or file input *will* be used.

For the following examples, we use this input data:

```
{
  "cafe": [
    {
      "id": "1",
      "type": "cafe",
```

(continues on next page)

(continued from previous page)


```
      "title": "Cafe Open",
      "owners": [
        {
          "firstname": "Clark",
          "lastname": "Kent",
          "email": "contact@cafeopen.example"
        }
      ],
      "dishes": [
        {
          "id": "bfast",
          "title": "All Day Breakfast",
          "allergens": {
            "freefrom": ["dairy", "eggs"],
            "contains": ["gluten", "soy"],
            "label": ["vegan", "vegetarian"]
          }
        },
        {
          "id": "soup",
          "title": "Jack's Soup",
          "allergens": {
            "freefrom": ["onion", "garlic"],
            "contains": ["dairy"],
            "label": ["vegetarian"]
          }
        }
      ],
      "coffee": "passable"
    },
    {
      "id": "2",
      "type": "pub",
      "title": "Pints R Us",
      "owners": [
        {
          "firstname": "Jim",
          "lastname": "Kirk",
          "email": "pintsrus@protonmail.com"
        }
      ],
      "coke": "pepsi"
    }
  ]
}
```


Here, the `owners` property contains a list with a single object. The `dishes` property is also a list of objects, but cannot be rolled up, as the cafe has more than one dish.

6.1.5.1 Rollup via schema

Here are the changes we make to the schema:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/flatten-tool/checkouts/latest/
↪examples/flatten/rollup/schema/cafe.schema
+++ /home/docs/checkouts/readthedocs.org/user_builds/flatten-tool/checkouts/latest/
↪examples/flatten/rollup/schema/cafe-rollup.schema
@@ -82,7 +82,8 @@
                 "$ref": "#/definitions/OwnerObject"
             },
             "type": "array",
-            "title": "Owner"
+            "title": "Owner",
+            "rollUp": ["firstname", "lastname", "email"]
         },
         "dishes": {
             "items": {
```

Here's the command we run:

```
$ flatten-tool flatten --root-list-path=cafe --use-titles --main-sheet-name=cafe --
↪rollup --schema=examples/flatten/rollup/schema/cafe-rollup.schema examples/flatten/
↪rollup/input.json -o examples/flatten/rollup/schema/actual
```

Here are the resulting sheets:

Table 13: sheet: cafe.csv

Identifier	Type	Name	Address	Owner:First name	Owner:Last name	Owner:Email	Coffee quality	Type of coke
1	cafe	Cafe Open		Clark	Kent	contact@cafeopen.example	passable	
2	pub	Pints R Us		Jim	Kirk	pintsrus@protonmail.com		pepsi

Table 14: sheet: Owners.csv

Identifier	Owner:First name	Owner:Last name	Owner:Email
1	Clark	Kent	contact@cafeopen.example
2	Jim	Kirk	pintsrus@protonmail.com

Notice how `Owner: First name`, `Owner: Last name` and `Owner: Email` now appear in both the `cafe.csv` and `Owner.csv` files.

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

```
UserWarning: More than one value supplied for "dishes". Could not provide rollup,␣
↪so adding a warning to the relevant cell(s) in the spreadsheet.
  warn('More than one value supplied for "{}". Could not provide rollup, so adding␣
↪a warning to the relevant cell(s) in the spreadsheet.'.format(parent_name+key))
```

6.1.5.2 Rollup via direct input

Run this command to rollup all properties of the `owners` object:

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --rollup=owners␣
↪examples/flatten/rollup/input.json -o examples/flatten/rollup/direct/actual
```

You can include multiple fields to rollup by passing `--rollup` multiple times:

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --rollup=owners --
↪rollup=dishes examples/flatten/rollup/input.json -o examples/flatten/rollup/direct/
↪actual
```

For the result:

Table 15: sheet: cafe.csv

id	type	title	owners/0/firstname	owners/0/lastname	owners/0/email	coffee	coke
1	cafe	Cafe Open	Clark	Kent	contact@cafeopen.example	passable	
2	pub	Pints R Us	Jim	Kirk	pintsrus@protonmail.com		pepsi

6.1.5.3 Rollup via file input

Run this command to rollup all properties of the `owners` object:

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --rollup=examples/
↪flatten/rollup/file/fields_to_rollup.txt examples/flatten/rollup/input.json -o␣
↪examples/flatten/rollup/file/actual
```

Where the file contains:

```
owners
```

You can include multiple fields to rollup by passing `--rollup` multiple times:

For the result:

Table 16: sheet: cafe.csv

id	type	title	owners/0/firstname	owners/0/lastname	owners/0/email	coffee	coke
1	cafe	Cafe Open	Clark	Kent	contact@cafeopen.example	passable	
2	pub	Pints R Us	Jim	Kirk	pintsrus@protonmail.com		pepsi

6.1.5.4 Selective rollup

If you don't want to include all of the properties of a rolled up object in the main sheet, you can use `rollup` in combination with `preserve-fields`, eg.

```
$ flatten-tool flatten --root-list-path=cafe --main-sheet-name=cafe --rollup=owners --
↪preserve-fields fields-to-preserve.txt examples/flatten/rollup/input.json -o␣
↪examples/flatten/rollup/direct/actual
```

Where `fields-to-preserve.txt` contains:

```
id
type
title
owners/email
```

This excludes `owners/firstname` and `owners/lastname` from *both* the main sheet and the owners sheet.

6.1.6 All flatten options

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

```
usage: flatten-tool flatten [-h] [-s SCHEMA] [-f {csv,ods,xlsx,all}] [--xml]
                            [--id-name ID_NAME] [-m MAIN_SHEET_NAME]
                            [-o OUTPUT_NAME] [--root-list-path ROOT_LIST_PATH]
                            [--rollup [ROLLUP]] [-r ROOT_ID] [--use-titles]
                            [--truncation-length TRUNCATION_LENGTH]
                            [--root-is-list] [--sheet-prefix SHEET_PREFIX]
                            [--filter-field FILTER_FIELD]
                            [--filter-value FILTER_VALUE]
                            [--preserve-fields PRESERVE_FIELDS]
                            [--disable-local-refs]
                            [--remove-empty-schema-columns]
                            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 {csv,ods,xlsx,all}, --output-format {csv,ods,xlsx,all}
                        Type of template you want to create. Defaults to all
                        available options
  --xml                 Use XML as the input format
  --id-name ID_NAME     String to use for the identifier key, defaults to 'id'
  -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 [ROLLUP]     "Roll up" columns from subsheets into the main sheet.
                        Pass one or more JSON paths directly, or a file with
                        one JSON path per line, or no value and use a schema
                        containing (a) rollUp attribute(s). Schema takes
                        precedence if both direct input and schema with
                        rollUps are present.
  -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.
  --truncation-length TRUNCATION_LENGTH
                        The length of components of sub-sheet names (default
```

(continues on next page)

(continued from previous page)

```
                      3).
  --root-is-list      The root element is a list. --root-list-path and meta
                      data will be ignored.
  --sheet-prefix SHEET_PREFIX
                      A string to prefix to the start of every sheet (or
                      file) name.
  --filter-field FILTER_FIELD
                      Data Filter - only data with this will be processed.
                      Use with --filter-value
  --filter-value FILTER_VALUE
                      Data Filter - only data with this will be processed.
                      Use with --filter-field
  --preserve-fields PRESERVE_FIELDS
                      Only these fields will be processed. Pass a file with
                      JSON paths to be preserved one per line.
  --disable-local-refs  Disable local refs when parsing JSON Schema.
  --remove-empty-schema-columns
                      When using flatten with a schema, remove columns and
                      sheets from the output that contain no data.
```

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 test suite. This should come with Python, but some distributions split it out. On Ubuntu you will need to install a package like `libpython3.5-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 Versioning and CHANGELOG

This library is versioned and a changelog is kept in the CHANGELOG.md file. See that file for more.

Any pull request to this library should include updating the CHANGELOG.md file.

7.5 PyPi

This is published on pypi.org.

7.6 What's coming up

7.6.1 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 serialised 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.

7.6.2 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')
```

7.6.3 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
```

7.6.4 More documentation

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

7.6.5 Naming

We could name the command line tool `flattentool` rather than `flatten-tool` so that everything is consistent.

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

Spreadsheet templates for OCDS can be downloaded from https://github.com/open-contracting/sample-data/tree/master/flat-template

These are generated with the commands listed in *Creating spreadsheet templates* below.

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).

8.3 Command Line Usage

8.3.1 Converting a JSON file to a spreadsheet

```
flatten-tool flatten input.json --root-id=ocid --main-sheet-name releases --root-list-
↪path=releases
```

This 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 command line 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 release_populated.xlsx):

```
flatten-tool unflatten release_populated.xlsx --root-id=ocid --base-json base.json --
↪input-format xlsx --output-name release.json --root-list-path=releases
```

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 --output-name release.json --root-list-path=releases
```

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

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

8.3.3 Creating spreadsheet templates

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 --schema release-schema.json --main-sheet-
↪name releases
```

This will create `template.xlsx` and a `template/` directory of CSV files.

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

CHAPTER 9

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-schema.json --main-sheet-name grants --rollup --use-titles
```

```
flatten-tool unflatten -o out.json -f xlsx input.xlsx --schema 360-giving-schema.json␣
↪--convert-titles --root-list-path='grants'
```

CHAPTER 10

Flatten Tool for IATI

Currently flatten-tool only supports Spreadsheet->XML for IATI (unflatten), not conversion in the other direction, or automated template creation.

10.1 Convert a spreadsheet to XML

For an XLSX file called `filename.xlsx`:

```
flatten-tool unflatten --xml --id-name iati-identifier --root-list-path iati-activity␣
↪-o iati.xml -f xlsx filename.xlsx
```

For a directory of CSV files called `csv_directory`:

```
flatten-tool unflatten --xml --id-name iati-identifier --root-list-path iati-activity␣
↪-o iati.xml -f csv csv_directory
```

(Outputs a file called `iati.xml`).

10.2 Example

Given these two sheets:

Table 1: sheet: main.csv

iati-identifier	reporting-org/@ref	reporting-org/@type	reporting-org/narrative	participating-org/@role	participating-org/@ref	participating-org/@type	activity-date/@type	activity-date/@iso-date	recipient-country/@code	recipient-country/@percentage	recipient-region/@code	recipient-region/@percentage	title/narrative	description/narrative	@last-updated-datetime
AA-AAA-123456789-ABC123	AA-AAA-123456789	40	Organisation name	1	AA-AAA-123456789	2	1	2011-10-01	AF	40	XK	60	A title	A description	2011-10-01T00:00:00+00:00
AA-AAA-123456789-ABC124	AA-AAA-123456789	40	Organisation name	1	AA-AAA-123456789	3	2	2016-01-01	AG	30	XK	70	Another title	Another description	2016-01-01T00:00:00+00:00

Table 2: sheet: transactions.csv

iati-identifier	transaction/0/transaction-type/@code	transaction/0/transaction-date/@iso-date	transaction/0/value/@value-date	transaction/0/value
AA-AAA-123456789-ABC123	2	2012-01-01	2012-01-01	10
AA-AAA-123456789-ABC123	3	2012-03-03	2012-03-03	20
AA-AAA-123456789-ABC124	2	2013-04-04	2013-04-04	30
AA-AAA-123456789-ABC124	3	2013-05-05	2013-05-05	40

Running this command:

```
$ flatten-tool unflatten --xml --id-name iati-identifier --root-list-path iati-
↪activity --xml-schema examples/iati/iati-activities-schema.xsd examples/iati/iati-
↪common.xsd -f csv examples/iati
```

Produces this XML:

```
<?xml version='1.0' encoding='utf-8'?>
<iati-activities>
  <!--XML generated by flatten-tool-->
  <iati-activity last-updated-datetime="2011-10-01T00:00:00+00:00">
    <iati-identifier>AA-AAA-123456789-ABC123</iati-identifier>
    <reporting-org ref="AA-AAA-123456789" type="40">
      <narrative>Organisation name</narrative>
    </reporting-org>
    <title>
      <narrative>A title</narrative>
    </title>
    <description>
      <narrative>A description</narrative>
```

(continues on next page)

(continued from previous page)

```
    </description>
    <participating-org ref="AA-AAA-123456789" role="1"/>
    <activity-status code="2"/>
    <activity-date iso-date="2011-10-01" type="1"/>
    <recipient-country code="AF" percentage="40"/>
    <recipient-country code="XK" percentage="60"/>
    <transaction>
      <transaction-type code="2"/>
      <transaction-date iso-date="2012-01-01"/>
      <value value-date="2012-01-01">10</value>
    </transaction>
    <transaction>
      <transaction-type code="3"/>
      <transaction-date iso-date="2012-03-03"/>
      <value value-date="2012-03-03">20</value>
    </transaction>
  </iati-activity>
  <iati-activity last-updated-datetime="2016-01-01T00:00:00+00:00">
    <iati-identifier>AA-AAA-123456789-ABC124</iati-identifier>
    <reporting-org ref="AA-AAA-123456789" type="40">
      <narrative>Organisation name</narrative>
    </reporting-org>
    <title>
      <narrative>Another title</narrative>
    </title>
    <description>
      <narrative>Another description</narrative>
    </description>
    <participating-org ref="AA-AAA-123456789" role="1"/>
    <activity-status code="3"/>
    <activity-date iso-date="2016-01-01" type="2"/>
    <recipient-country code="AG" percentage="30"/>
    <recipient-country code="XK" percentage="70"/>
    <transaction>
      <transaction-type code="2"/>
      <transaction-date iso-date="2013-04-04"/>
      <value value-date="2013-04-04">30</value>
    </transaction>
    <transaction>
      <transaction-type code="3"/>
      <transaction-date iso-date="2013-05-05"/>
      <value value-date="2013-05-05">40</value>
    </transaction>
  </iati-activity>
</iati-activities>
```

CHAPTER 11

Flatten Tool for BODS

11.1 flatten and unflatten

This data standard has a list as the root element, as opposed to other standards where the root element is a dict with meta data and a list of data. When flattening and unflattening, use the `--root-is-list` option.

The id element is `statementID`, so also use the `--id-name` option.

```
flatten-tool flatten -f csv --root-is-list --id-name=statementID -o examples/bods-one-
↪flatten examples/bods-one.json
flatten-tool unflatten -f csv --root-is-list --id-name=statementID -o examples/bods-
↪one-unflattened.json examples/bods-one-flatten
```

11.2 flatten

This data standard has three types of statement - `entityStatement`, `personStatement` or `ownershipOrControlStatement`. When using flatten, the spreadsheets produced can become very mixed up.

The main.csv spreadsheet will have data of all three types in it. What's worse is that it's unclear what columns apply to what types - for instance, `foundingDate` applies to entities but `birthDate` applies to people. However both columns appear in main.csv!

It's also unclear what subsheets apply to which type - for instance there might be a subsheet called `identifiers` but it's not clear what type this applies to! (The answer is entities)

It would be better to have separate sheets for each type. That way, only the relevant columns will appear in each sheet and it will be clear which subsheet applies to which sheet.

You can solve this with a combination of the filter and sheet prefix options. To flatten a set of data, run 3 commands:

```
flatten-tool flatten --sheet-prefix=1_person_ --filter-field=statementType --filter-
↪value=personStatement -f csv -o example1/ example1.json --root-is-list --id-
↪name=statementID
flatten-tool flatten --sheet-prefix=2_entity_ --filter-field=statementType --filter-
↪value=entityStatement -f csv -o example1/ example1.json --root-is-list --id-
↪name=statementID
flatten-tool flatten --sheet-prefix=3_ownership_ --filter-field=statementType --
↪filter-value=ownershipOrControlStatement -f csv -o example1/ example1.json --root-
↪is-list --id-name=statementID
```

You will have a set of sheets:

- 1_person_addresses.csv
- 1_person_main.csv
- 1_person_names.csv
- 1_person_nationalities.csv
- 2_entity_identifiers.csv
- 2_entity_main.csv
- 3_ownership_interests.csv
- 3_ownership_main.csv

`birthDate` only appears in `1_person_main.csv` and `foundingDate` only appears in `2_entity_main.csv`, so it is clear which column is for which type.

Note this works in CSV mode. If you want to use Excel mode, you'll need to specify 3 separate output files and then combine the sheets in them into one file afterwards by hand.

```
flatten-tool flatten --sheet-prefix=1_person_ --filter-field=statementType --filter-
↪value=personStatement -f xlsx -o example1/part1.xlsx example1.json --root-is-list --
↪id-name=statementID
flatten-tool flatten --sheet-prefix=2_entity_ --filter-field=statementType --filter-
↪value=entityStatement -f xlsx -o example1/part2.xlsx example1.json --root-is-list --
↪id-name=statementID
flatten-tool flatten --sheet-prefix=3_ownership_ --filter-field=statementType --
↪filter-value=ownershipOrControlStatement -f xlsx -o example1/part3.xlsx example1.
↪json --root-is-list --id-name=statementID
```

11.3 unflatten

11.3.1 Schema

As well as the options above, also pass the `--schema` option so that types are set correctly. Note the boolean and the integer in the output.

```
$ flatten-tool unflatten -f csv examples/bods/unflatten/input  -o examples/bods/
↪unflatten/actual/out.json --root-is-list  --id-name=statementID --schema examples/
↪bods/schema/bods-package.json
```

```
[
    {
        "statementID": "fbfd0547-d0c6-4a00-b559-5c5e91c34f5c",
        "interests": [
            {
                "type": "shareholding",
                "interestLevel": "direct",
                "beneficialOwnershipOrControl": true,
                "startDate": "2016-04-06",
                "share": {
                    "exact": 100
                }
            }
        ],
        "statementType": "ownershipOrControlStatement",
        "statementDate": "2017-11-18",
        "subject": {
            "describedByEntityStatement": "1dc0e987-5c57-4a1c-b3ad-61353b66a9b7"
        },
        "interestedParty": {
            "describedByPersonStatement": "019a93f1-e470-42e9-957b-03559861b2e2"
        }
    }
]
```


11.3.2 Order is important

In the BODS schema, statements must appear in a certain order. Each of the `entityStatements` or `personStatements` referenced by a particular `ownershipOrControlStatement` must appear before that particular statement in the ordered array.

If you have only one main table, you must make sure the statements appear in the correct order.

For example, this is good:

```
statementID,statementType, ...
1dc0e987-5c57-4a1c-b3ad-61353b66a9b7,entityStatement,
019a93f1-e470-42e9-957b-03559861b2e2,personStatement,
fbfd0547-d0c6-4a00-b559-5c5e91c34f5c,ownershipOrControlStatement,
```

Pay attention to other sheets to. If a subsheet is loaded before main.csv, the order might still be wrong.

Alternatively, you may have several main tables, one for each type of statement:

- main-control-own.csv
- main-entity.csv
- main-person.csv

In this case, you may get data in the wrong order.

To fix this, put numbers in front of the file names so that you can be sure what order they will appear in. For instance:

- 1identifiers.csv
- 1main-entity.csv
- 2addresses.csv
- 2main-person.csv

- 2names.csv
- 2nationalities.csv
- 3interests.csv
- 3main-control-own.csv

11.4 create-template

You can run this directly on `bods-package.json`:

```
flatten-tool create-template -f csv -s bods-package.json -o template --root-
↪id=statementID
```

However, this will produce spreadsheets where the several types are all mixed up. As explained above, this creates problems because columns appear in the main sheet that are not relevant to all types, and it's not clear which subsheet applies to which type.

Instead, this process can be followed to obtain clearer templates:

1) Create a new blank directory and change into it.

2) Produce the person sheets only by running:

```
flatten-tool  create-template -f csv -s /path/to/person-statement.json -o . --root-
↪id=statementID
```

3) Rename all the files in the directory to have `1_person_` at the start.

If your on a bash shell, you can do this by running:

```
for FILENAME in *; do mv $FILENAME 1_person_$FILENAME; done
```

4) Produce the entity sheets only by running:

```
flatten-tool  create-template -f csv -s /path/to/entity-statement.json -o . --root-
↪id=statementID
```

5) Rename all the new files in the directory to have `2_entity_` at the start.

If your on a bash shell, you can do this by running:

```
for FILENAME in *; do if [[ $FILENAME != 1_* ]] ; then mv $FILENAME 2_entity_
↪$FILENAME; fi; done
```

6) Produce the ownership or control sheets only by running:

```
flatten-tool  create-template -f csv -s /path/to/ownership-or-control-statement.json -
↪o . --root-id=statementID
```

7) Rename all the new files in the directory to have `3_ownership_control_` at the start.

If your on a bash shell, you can do this by running:

```
for FILENAME in *; do if [[ $FILENAME != 1_* ]] && [[ $FILENAME != 2_* ]] ; then mv
↪$FILENAME 3_ownership_control_$FILENAME; fi; done
```

You will now have a directory of files that look like this:

- 1_person_addresses.csv
- 1_person_annotations.csv
- 1_person_identifiers.csv
- 1_person_main.csv
- 1_person_names.csv
- 1_person_nationalities.csv
- 1_person_pepStatus.csv
- 1_person_sou_assertedBy.csv
- 2_entity_addresses.csv
- 2_entity_annotations.csv
- 2_entity_identifiers.csv
- 2_entity_main.csv
- 2_entity_sou_assertedBy.csv
- 3_ownership_control_annotations.csv
- 3_ownership_control_interests.csv
- 3_ownership_control_main.csv
- 3_ownership_control_sou_assertedBy.csv

The advantages are:

- Separate for each type, so it's clear what sheet applies to each type.
- Each sheet only has the relevant columns in it, so there is no confusion about whether they apply or not.
- The sheets have numbers at the start, so that when `unflatten` is used the statements will appear in the right order in the output.

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 12

Indices and tables

- genindex
- modindex
- search