This note is a simple data wrangling example worked using both the Python data\_algebra package and the R cdata package. Both of these packages make data wrangling easy through he use of coordinatized data concepts (relying heavily on Codd’s “rule of access”).

The advantages of data\_algebra and cdata are:

* The user specifies their desired transform declaratively *by example* and *in data*. What one does is: work an example, and then write down what you want.
* The transform systems can print what a transform is going to do. This makes reasoning about data transforms *much* easier.
* The transforms, as they themselves are written as data, can be easily shared between systems (such as R and Python).

Let’s re-work a small R cdata example, using the Python package data\_algebra.

**An Example**

First we import some modules and packages, and import some notional data.

In [1]:

# <https://pandas.pydata.org>

import pandas

# PyYAML from <https://pyyaml.org>

import yaml

# pip install <https://github.com/WinVector/data_algebra/raw/master/dist/data_algebra-0.1.3.tar.gz>

import data\_algebra.cdata

import data\_algebra.cdata\_impl

import data\_algebra.data\_ops

import data\_algebra.yaml

import data\_algebra.SQLite

# ask YAML to write simpler structures

data\_algebra.yaml.fix\_ordered\_dict\_yaml\_rep()

# read our example data

iris = pandas.read\_csv('iris\_small.csv')

# print it out

iris

Out[1]:

|  | **Sepal.Length** | **Sepal.Width** | **Petal.Length** | **Petal.Width** | **Species** | **id** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | setosa | 0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | setosa | 1 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | setosa | 2 |

Our goal is to move from this de-normalized or wide-form (or “model matrix”/”data matrix” form, where each record is exactly one row) into a tall form where records may span more than one row.

Or, more concretely, we want our data to look like the following.

In [2]:

answer = pandas.read\_csv("answer.csv")

answer

Out[2]:

|  | **id** | **Species** | **Part** | **Measure** | **Value** |
| --- | --- | --- | --- | --- | --- |
| **0** | 0 | setosa | Petal | Length | 1.4 |
| **1** | 0 | setosa | Petal | Width | 0.2 |
| **2** | 0 | setosa | Sepal | Length | 5.1 |
| **3** | 0 | setosa | Sepal | Width | 3.5 |
| **4** | 1 | setosa | Petal | Length | 1.4 |
| **5** | 1 | setosa | Petal | Width | 0.2 |
| **6** | 1 | setosa | Sepal | Length | 4.9 |
| **7** | 1 | setosa | Sepal | Width | 3.0 |
| **8** | 2 | setosa | Petal | Length | 1.3 |
| **9** | 2 | setosa | Petal | Width | 0.2 |
| **10** | 2 | setosa | Sepal | Length | 4.7 |
| **11** | 2 | setosa | Sepal | Width | 3.2 |

Notice each row of the original data set is now four rows of the derived one. This “tall form” is often useful for plotting.

Back to our example.

To specify our desired transform, we build a structure describing what a data record looks like. We can crib this specification from the answer as follows.

In [3]:

control\_table = answer.loc[answer.id == 0, ['Part', 'Measure']]

control\_table = control\_table.reset\_index(inplace=False, drop=True)

control\_table["Value"] = [control\_table['Part'][i] + '.' + control\_table['Measure'][i] for

i in range(control\_table.shape[0])]

control\_table

Out[3]:

|  | **Part** | **Measure** | **Value** |
| --- | --- | --- | --- |
| **0** | Petal | Length | Petal.Length |
| **1** | Petal | Width | Petal.Width |
| **2** | Sepal | Length | Sepal.Length |
| **3** | Sepal | Width | Sepal.Width |

We can derive the control table from the answer, as we did here, or just type one in directly. The idea is: we can use any method we want to derive the prototype record shape, we are not limited to a sub-query language or methodology from any one package.

For each record we take care to identify what keys identify records (the record\_keys) and want parts identify rows within the record (the control\_table\_keys). We suppress the record\_key when writing the control table, as these are exactly the values that do not vary inside each record.

We combine all of these specification into a RecordSpecification as follows:

In [4]:

record\_spec = data\_algebra.cdata.RecordSpecification(

control\_table,

control\_table\_keys = ['Part', 'Measure'],

record\_keys = ['id', 'Species']

)

record\_spec

Out[4]:

RecordSpecification

record\_keys: ['id', 'Species']

control\_table\_keys: ['Part', 'Measure']

control\_table:

Part Measure Value

0 Petal Length Petal.Length

1 Petal Width Petal.Width

2 Sepal Length Sepal.Length

3 Sepal Width Sepal.Width

In this notation any cells of the control table from columns that are not control\_table\_keys are “value stand ins”. During data transfrom these cells will be replaced by values coming from the columns named by these cells.

Now we can transform our original row-record oriented data into general block records. To do this we specify a RecordMap using our record specification to describe the outgoing record structure. The incoming record structure is implicitly assumed to be single-row records, unless we specify otherwise (using the blocks\_in argument).

In [5]:

mp\_to\_blocks = data\_algebra.cdata\_impl.RecordMap(blocks\_out=record\_spec)

print(str(mp\_to\_blocks))

Transform row records of the form:

record\_keys: ['id', 'Species']

['id', 'Species', 'Petal.Length', 'Petal.Width', 'Sepal.Length', 'Sepal.Width']

to block records of structure:

RecordSpecification

record\_keys: ['id', 'Species']

control\_table\_keys: ['Part', 'Measure']

control\_table:

Part Measure Value

0 Petal Length Petal.Length

1 Petal Width Petal.Width

2 Sepal Length Sepal.Length

3 Sepal Width Sepal.Width

And we are ready to apply our specified transform.

In [6]:

arranged\_blocks = mp\_to\_blocks.transform(iris)

arranged\_blocks

Out[6]:

|  | **id** | **Species** | **Part** | **Measure** | **Value** |
| --- | --- | --- | --- | --- | --- |
| **0** | 0 | setosa | Petal | Length | 1.4 |
| **1** | 0 | setosa | Petal | Width | 0.2 |
| **2** | 0 | setosa | Sepal | Length | 5.1 |
| **3** | 0 | setosa | Sepal | Width | 3.5 |
| **4** | 1 | setosa | Petal | Length | 1.4 |
| **5** | 1 | setosa | Petal | Width | 0.2 |
| **6** | 1 | setosa | Sepal | Length | 4.9 |
| **7** | 1 | setosa | Sepal | Width | 3.0 |
| **8** | 2 | setosa | Petal | Length | 1.3 |
| **9** | 2 | setosa | Petal | Width | 0.2 |
| **10** | 2 | setosa | Sepal | Length | 4.7 |
| **11** | 2 | setosa | Sepal | Width | 3.2 |

We see the operation has been performed for us. Notice we specify the transform *declaratively* with data structures carrying deceptions of what we want, instead of having to build a sequence of verbs that realize the transformation.

**Inverting the transform**

An inverse transform is simply expressed by reversing the roles of the blocks\_out and blocks\_in arguments. In this case the output is row-records, as we didn’t specify an outgoing block structure with blocks\_out.

In [7]:

mp\_to\_rows = data\_algebra.cdata\_impl.RecordMap(blocks\_in=record\_spec)

print(str(mp\_to\_rows))

Transform block records of structure:

RecordSpecification

record\_keys: ['id', 'Species']

control\_table\_keys: ['Part', 'Measure']

control\_table:

Part Measure Value

0 Petal Length Petal.Length

1 Petal Width Petal.Width

2 Sepal Length Sepal.Length

3 Sepal Width Sepal.Width

to row records of the form:

record\_keys: ['id', 'Species']

['id', 'Species', 'Petal.Length', 'Petal.Width', 'Sepal.Length', 'Sepal.Width']

In [8]:

arranged\_rows = mp\_to\_rows.transform(arranged\_blocks)

arranged\_rows

Out[8]:

|  | **id** | **Species** | **Petal.Length** | **Petal.Width** | **Sepal.Length** | **Sepal.Width** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 0 | setosa | 1.4 | 0.2 | 5.1 | 3.5 |
| **1** | 1 | setosa | 1.4 | 0.2 | 4.9 | 3.0 |
| **2** | 2 | setosa | 1.3 | 0.2 | 4.7 | 3.2 |

**Arbitrary transforms**

Arbitrary record to record transforms can be specified by setting both blocks\_in (to describe incoming structure) and blocks\_out (to describe outgoing structure) at the same time.

**Transforms in databases**

data\_algebra also implements all the transform steps in databases using [SQL](https://en.wikipedia.org/wiki/SQL) (via row\_recs\_to\_blocks\_query() and blocks\_to\_row\_recs\_query()).

These queries can be seen below.

In [9]:

db\_model = data\_algebra.SQLite.SQLiteModel()

In [10]:

print(db\_model.row\_recs\_to\_blocks\_query(

source\_view=data\_algebra.data\_ops.describe\_pandas\_table(

iris, 'iris'),

record\_spec=record\_spec,

record\_view=data\_algebra.data\_ops.describe\_pandas\_table(

record\_spec.control\_table, "control\_table")

))

SELECT

a."id" AS "id",

a."Species" AS "Species",

b."Part" AS "Part",

b."Measure" AS "Measure",

CASE

WHEN b."Value" = 'Petal.Length' THEN a."Petal.Length"

WHEN b."Value" = 'Petal.Width' THEN a."Petal.Width"

WHEN b."Value" = 'Sepal.Length' THEN a."Sepal.Length"

WHEN b."Value" = 'Sepal.Width' THEN a."Sepal.Width"

ELSE NULL END AS "Value"

FROM (

"iris" ) a

CROSS JOIN (

"control\_table" ) b

ORDER BY a."id", a."Species", b."Part", b."Measure"

In [11]:

print(db\_model.blocks\_to\_row\_recs\_query(

source\_view=data\_algebra.data\_ops.describe\_pandas\_table(

iris, 'iris'),

record\_spec=record\_spec

))

SELECT

"id" AS "id",

"Species" AS "Species",

MAX(CASE WHEN ( "Part" = 'Petal' ) AND ( "Measure" = 'Length' ) THEN "Value" ELSE NULL END) AS "Petal.Length",

MAX(CASE WHEN ( "Part" = 'Petal' ) AND ( "Measure" = 'Width' ) THEN "Value" ELSE NULL END) AS "Petal.Width",

MAX(CASE WHEN ( "Part" = 'Sepal' ) AND ( "Measure" = 'Length' ) THEN "Value" ELSE NULL END) AS "Sepal.Length",

MAX(CASE WHEN ( "Part" = 'Sepal' ) AND ( "Measure" = 'Width' ) THEN "Value" ELSE NULL END) AS "Sepal.Width"

FROM (

"iris"

)

GROUP BY "id", "Species"

ORDER BY "id", "Species"

The use case for SQL queries is: the queries can be used to with “CREATE TABLE table\_name AS” to materialize transform results in a database, without round-tripping the data in and out of the database.

As complicated as the queries look, they actually expose some deep truths:

* The row\_recs\_to\_blocks\_query() is essentially a cross-join of the data to the record description. Each combination of data row and record description row builds a new result row.
* The blocks\_to\_row\_recs\_query() is an aggregation. Each set of rows corresponding to a given data record is aggregated into a single result row.
* Just about any arbitrary record shape to arbitrary record shape can be written as a transform from the first record shape to row-records (record sets that have exactly one row per record), followed by a transform from the row-records to the new format. This transform can preserve column types as in the intermediate form each different record entry has its own column. This leads us to believe that transforming to and from single-row records are in fact fundemental operations, and not just implementation details.

**The R cdata version**

The data\_algebra had been previously implemented in  
R in the cdata, rquery, and rqdatatable packages.  
This packages support both in-memory operations and translation of the transforms to SQL.

We would perform the above transforms in R as follows.

First we write out a copy of the transform.

In [12]:

# convert objects to a YAML string

xform\_rep = yaml.dump(mp\_to\_blocks.to\_simple\_obj())

# write to file

with open("xform\_yaml.txt", "wt") as f:

print(xform\_rep, file=f)

Now we we start up our R environment and read in the transform specification and data.

In [13]:

%load\_ext rpy2.ipython

In [14]:

%%R

# install.packages("cdata")

library(cdata)

library(yaml)

# <https://github.com/WinVector/data_algebra/blob/master/Examples/cdata/cdata_yaml.R>

source("cdata\_yaml.R")

iris <- read.csv('iris\_small.csv')

print(iris)

Sepal.Length Sepal.Width Petal.Length Petal.Width Species id

1 5.1 3.5 1.4 0.2 setosa 0

2 4.9 3.0 1.4 0.2 setosa 1

3 4.7 3.2 1.3 0.2 setosa 2

In [15]:

%%R

r\_yaml <- yaml.load\_file("xform\_yaml.txt")

transform <- convert\_yaml\_to\_cdata\_spec(r\_yaml)

print(transform)

{

row\_record <- wrapr::qchar\_frame(

"id" , "Species", "Petal.Length", "Petal.Width", "Sepal.Length", "Sepal.Width" |

. , . , Petal.Length , Petal.Width , Sepal.Length , Sepal.Width )

row\_keys <- c('id', 'Species')

# becomes

block\_record <- wrapr::qchar\_frame(

"id" , "Species", "Part" , "Measure", "Value" |

. , . , "Petal", "Length" , Petal.Length |

. , . , "Petal", "Width" , Petal.Width |

. , . , "Sepal", "Length" , Sepal.Length |

. , . , "Sepal", "Width" , Sepal.Width )

block\_keys <- c('id', 'Species', 'Part', 'Measure')

# args: c(checkNames = TRUE, checkKeys = FALSE, strict = FALSE, allow\_rqdatatable = TRUE)

}

Now that we have recovered the transform, we can use it in R.

In [16]:

%%R

iris %.>% transform

id Species Part Measure Value

1 0 setosa Petal Length 1.4

2 0 setosa Petal Width 0.2

3 0 setosa Sepal Length 5.1

4 0 setosa Sepal Width 3.5

5 1 setosa Petal Length 1.4

6 1 setosa Petal Width 0.2

7 1 setosa Sepal Length 4.9

8 1 setosa Sepal Width 3.0

9 2 setosa Petal Length 1.3

10 2 setosa Petal Width 0.2

11 2 setosa Sepal Length 4.7

12 2 setosa Sepal Width 3.2

**Cross-language work**

As the record transform specifications, both in Python data\_algebra and R cata are simple data structures (just the control table, and a few lists of key column names), they can be moved from one language to another by YAML.

data\_algebra supplies a write method, so cross-language interoperation is just a matter of adding additional read/write methods.

In [17]:

print(yaml.dump(mp\_to\_blocks.to\_simple\_obj()))

type: data\_algebra.cdata\_impl.RecordMap

blocks\_out:

type: data\_algebra.cdata.RecordSpecification

record\_keys:

- id

- Species

control\_table\_keys:

- Part

- Measure

control\_table:

Part:

- Petal

- Petal

- Sepal

- Sepal

Measure:

- Length

- Width

- Length

- Width

Value:

- Petal.Length

- Petal.Width

- Sepal.Length

- Sepal.Width

**Conclusion**

The cdata and data\_algebra systems yield powerful implementations, and deep understanding of the nature of record transformations. They allow one to reshape data quickly and conveniently either in R. Python/Pandas, or even SQL.