Data Refinery Documentation

Release 0.1.66

BBVA-Labs

Feb 06, 2018

Contents

CHAPTER 1

Introduction

1.1 Motivation

Data Analytics is now very popular on modern companies. Nowadays nobody deny the data insights value regarding business development. Also technology allow us to manage huge amount of data, even when look imposible to process. In fact, scientific libraries allow us to perform advaced analytics.

> But i can't stop thinking about move local work to the cluster, and for me looks like a lonely valley.

This work is a must every time that a hypothesis looks promising and must hit the real world. Doing this job it's like walk along a lonely valley plenty of obstacles. The scientific libraries usually are not desingned to scale beyond machine memory (usually this is a very hard task) because modern machines are big enought. But a lot of times I spent time translating from R o Python to Java or Scala; just because clusters software is tipically programed on JVM languages.

I think that smaller the transformation code, the better is translation to the cluster software. And also Python nowadays live between the two worlds.

1.2 Solutions

I usually talk about this translation problem with my colleagues on Big Data events. Some of then put cluster tools in hands of their data scientist, like Spark. But this comes with a cost, every test boot up a complete cluster on the local machine, slow down the development process. But not every test must hit the real work, so comes with an unnecesary adaptation to the cluster, instead a quick local test. Also there is an slightly outdated AI algorithms on cluster libraries. Some times people try to do the work directly on real cluster, with real data. Thats awesome but await to all the data be processed, usually slow down development process.

There is no silver bullet for this problem, every solution that you try has its own tradeoffs. Like "i must develop on cluster, but every test run for days" or like "i have my program finished but i must rewrite it in java"

On my opinion the only way to go is more automation and be iterative, so the scientific can use their own tools and the engineer can work less. When we say iterative it's not only about development, it's also about use more data if our hypotesis is still alive. Automation allow us to be more productive and secure.

> Automation reveals the importance of a good data pipeline.

1.3 Help not substitution

If we want to be successful we need steal the better of other tools, not try to reinvent the wheel. Big Data framework move data from one place to another very well, and also can distribute computation easilly.

> But when we want add a field, or split a date field, it's completely on our hands.

Scientific libraries, also, help us loading data and show information, but if we want do a transformation, again it's in our hands.

Transformations are the more tedious work. And are written directly on the language of the cluster or notebook. The data refinery library comes to help here, giving you code that runs on cluster or notebook. This library it's not a complete ETL solution, but proves to be a huge help when things going real.

This can be done thanks it's minimalist interface and contained scope. Library only do one thing very well, transform data. It build a function that you can use anywhere. So if you need run a small script, you can use it, or if you need to scale up the process, you can use Spark.

1.4 Focus on streaming

Real world is not static, never stop, so we focus on streaming solutions. Sometimes people ask me "where i config the data file?" or "how i obtain all the values of the column?. On streaming environment you only have two things, a function and the current row. If you have a file, it's your work to split into rows and pass it to the transform function. Streamings are endless so you can't know all the values of a column.

As we say, this library does not do work that other frameworks do better. Pandas and Spark can load files easily and send results elsewhere. I strongly recommend you use a framework where you find confortable, and use this library as help.

Data scientist are usually used to work with Pandas. Its interface is based on columns, very efficient on batch, but not on streaming. If you are used to Pandas, maybe this approach will twist your mind.

> Work thinking about row not column.

This approach it's resource friendly, and you can use on any machine, you only need one row in memory.

CHAPTER 2

Installation

2.1 Requirements

- Python 3.5 or 3.6

2.2 Pip

To install datarefinery using pip:

```
pip install datarefinery
```

CHAPTER 3

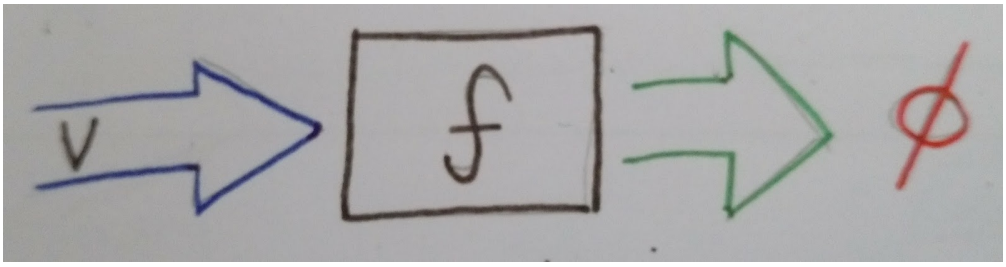
The Basics

3.1 Field operations

TODO: explain how we build the operation from field to a complete process

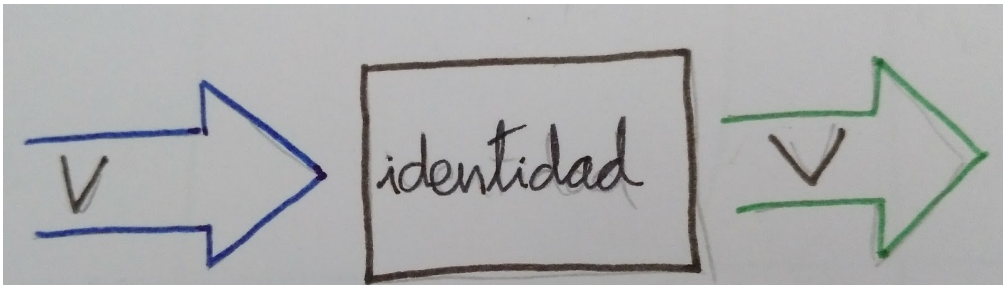
An ETL process is composed by a group of performed operations on a row and rows are composed by groups of columns. So the minimal possible operation is performed on a field.

Suppose a function is useless on a field V:



As the process is composed by operations on fields, if we execute this function to a row with one field V, there won't be output.

Thus we need an usable function, as example a basic function that keeps value of a field. This function returns the input and its name is identity:



3.1.1 How process errors in data

One of the most important situations we have to process is errors on data. A typical issue is when we expect some field but row doesn't have it (unstructured data source).

If any transformation fails we must report precisely without stop the process. All reasons for failure will aggregate by row, so the datascientist will have all the issues regarding a row together, and the process will continue with the next row. Because of this, field error will not stop the row process, and this one will not stop the dataset process. On production environment unexpected data is a common issue, so this data must be logged along their reasons for further research and process improvement.

To fix this problem, Data Refinery transformation functions returns the transformed value and the possible error. This situation takes us to three scenes: - Function returns a value and no error. - Function returns an error and provokes the value doesn't matter. - Function execution is OK, but there is not value.

In the first case we can use the transformed value. In the second one, we don't care about the value because there is an insuperable error. Finally it depends on the current operation, the process rejects the field or fails because there is not value.

3.1.2 List of functions

On data transformation processes, it's common to transform types or values if a condition is triggered, decompose data,.. Data Refinery has defined this kind of functions. Furthermore, if it's necessary we can create our own functions.

More complex situations can be solved chaining together two or more simple functions. This atomic design allows changes on application behaviour with very little user effort.

The majority of ETL's functions require some context parameters in order to produce another, more specialized function. This specialized function can then be used to transform the field. This style of operation improves simple composition given that all functions will have the same input and output instead of a variable number of parameters.

type_enforcer

Expected function in order to cast the value to a type that can rise a cast exception.

Sometimes, data source doesn't contain type or this type is wrong (numbers as string, etc). This function use as context a type changer function. If this function raises an error, type enforcer will return None and the error message raised.

For example, if you want to change the data type to integer:

```
from datarefinery.FieldOperations import type_enforcer

int_enforcer = type_enforcer(lambda x: int(x))
(res, err) = int_enforcer("6")
print(res) # 6
```

min_max_normalization

Typical operation on machine learning. It consists of interpolate a value between 0 (as minimum) and 1 (as maximum). So, this function need a context with the maximum and minimum to run. For example:

```
from datarefinery.FieldOperations import min_max_normalization

normalizator = min_max_normalization(1, 10)
(res, err) = normalizator(5)
print(res) # 0.5
```

std_score_normalization

Returns the distance of a particular datum . . .

This operation represents how far it's the value from mean, using deviation as scale.

The function requires two input parameters, the column average and the standard deviation. Usage example:

```
from datarefinery.FieldOperations import std_score_normalization

normalizator = std_score_normalization(79, 8)
(res, err) = normalizator(85)
print(res) # 0.75
```

buckets_grouping

Transform a lineal numeric value into a categorical one. For instance it can be used to group users by age.

A minimum of one input value is mandatory. This will produce two groups, the first one from negative infinity to the given value, and the second one from the given value to infinity.

For example, in order to categorize users into three groups (children, adults and elderly) the values 18 and 70 can be passed to the function. This will produce the following groups:

1. From negative infinity to 18.
2. From 18 to 70.
3. From 70 to infinity.

```
from datarefinery.FieldOperations import buckets_grouping

group = buckets_grouping(18, 70)
(res, err) = group(10)
print(res) # 1
(res, err) = group(20)
print(res) # 2
(res, err) = group(73)
print(res) # 3
```

linear_category

Translates the textual value of a field into a numeric value given a list of possible values.

The input value is a list of categories. Keep in mind that this list must always be in the same order to consistently translate the values.

The translated value will be the category index of the list.

As an example, the age can be categorized again, but this time the input a text value instead of a numeric one.

```
from datarefinery.FieldOperations import linear_category

categorizer = linear_category(["niño", "adulto", "jubilado"])
(res, err) = categorizer("adulto")
print(res) # 2
```

column_category

Translates the textual value of a field into a set of columns given a list of possible values. A column will be produced by each one of the members of the input list. This columns will have a value of *0* by default except for the corresponding category that will have a value of *1*. This is known as *one hot vector*.

Example:

```
from datarefinery.FieldOperations import column_category

categorizer = column_category(["niño", "adulto", "jubilado"])
(res, err) = categorizer("niño")
print(res) # {"niño": "1", "adulto": "0", "jubilado": "0"}
```

This operation adds new columns, so is usually used along with an event operation of type [append](##Change it).

add_column_prefix

Adds a prefix to the column name. This is useful in a scenario when other function generates a new column with the same name of another already existing.

```
from datarefinery.FieldOperations import add_column_prefix

prefix = add_column_prefix("good")
(res, err) = prefix({"one": "me"})
print(res) # {"good_one": "me"}
```

explode

Flattens a nested data structure even when is made up by a list of objects.

In the case of just one inner object, only the original name prefix will be added.

When multiple objects are present the same prefix will be added and in addition, a numerical suffix (starting on 1) fo the second position.

In this example we exploded the field *name*:

```
from datarefinery.FieldOperations import explode

explode_name = explode("name")
(res, err) = explode_name({"name": {"first": "Bob", "last": "Dylan"}})
print(res) # {"name_first": "Bob", "name_last": "Dylan"}
```

replace_if

Replaces a value when some condition fulfilled.

Two functions are expected, the former should return a boolean value and the latter should produce a new value in case of the former function returns *True*. Both function will receive the field value.

As an example, if we want to replace by zero all negative values:

```
from datarefinery.FieldOperations import replace_if

change = replace_if(lambda x: x<0, lambda x: 0)
```

```
(res, err) = change(-3)
print(res) # 0
```

date_parser - time_parser

Tries to parse a date with the given list of date formats. If none of the formats successfully parses the date then the function returns an error.

The expected formats are Python standard time formats.

```
from datarefinery.FieldOperations import date_parser

parser = date_parser(["%Y-%m-%d"])
(res, err) = parser("2017-03-22")
print(res) # <datetime class>
```

There is a similar function to format hours, minutes and seconds.

explode_date - explode_time

Transforms a datetime object to a series of columns with numeric values.

```
import datetime
from datarefinery.FieldOperations import explode_date

(res, err) = explode_date(datetime(2017,3,22))
print(res) # {"year": 2017, "month": 3, "day": 22, "hour":0, "minute": 0, "second": 0}
```

If multiple date exists on the event, please consider using the function [add_prefix](###Prefijo de columna). If no all fields are needed the function [remove column](###Quitando columnas) can be used.

This function is typically used along with *date_parser*.

remove_columns

Removes one or more columns from a set.

This function is usually used along with other functions which generate multiple columns.

In case of not require a column, is better just not to operate it. This non-operated column will be removed automatically.

```
import datetime
from datarefinery.tuple.TupleDSL import compose
from datarefinery.FieldOperations import explode_date, remove_columns

only_year_month = compose(explode_date, remove_columns("day", "hour", "minute",
↪"sencond"))
(res, err) = only_year_month(datetime(2017,3,22))
print(res) # {"year": 2017, "month": 3}
```

match_dict

Translates values from a table.

```
from datarefinery.FieldOperations import match_dict

d = {"Spain": "ES", "United States of America": "US"}
iso_decoder = match_dict(d)
(res, err) = iso_decoder("Spain")
print(res) # "ES"
```

3.1.3 Composing field operations

This powerful concept from functional programing allow us to build complex applications using simple blocks (functions).

Composition is similar to programming in the sense that a small set of operations can be combined to solve a very large set of problems.

All functions on the library can be combined together using the function *combine*.

This concept is better shown by example.

Normalize Numeric Data

Convert a numeric string to a numeric format and then normalize using min max approximation.

```
from datarefinery.tuple.TupleDSL import compose
from datarefinery.FieldOperations import type_enforcer, min_max_normalization

str_2_min_max = compose(
    type_enforcer(lambda x: int(x)),
    min_max_normalization(0, 100)
)
(res, err) = str_2_min_max("50")
print(res) # 0.5
```

Date data

Explode a date is a typical operation too. It keeps year, month and day as data. Furthermore, it adds a prefix to avoid problems with other fields.

```
from datarefinery.tuple.TupleDSL import compose
from datarefinery.FieldOperations import date_parser, explode_date, remove_columns,␣
↪add_column_prefix

complete_date = compose(
    date_parser(["%Y-%m-%d"]),
    explode_date,
    remove_columns("hour", "minute", "second"),
    add_column_prefix("x")
)
(res, err) = complete_date("2017-03-22")
print(res) # {"x_year": 2017, "x_month": 3, "x_day": 22}
```

Day to one hot vector

This example returns one hot vector using a date string and week days.

```
from datarefinery.tuple.TupleDSL import compose
from datarefinery.tuple.TupleOperations import wrap
from datarefinery.FieldOperations import date_parser, match_dict, column_category

week_days={
    0: "Mo", 1: "Tu", 2: "We", 3: "Th", 4: "Fr", 5: "Sa", 6: "Su"
}

def day_of_week(dat):
  return dat.weekday()

day_hot = compose(
    date_parser(["%Y-%m-%d"]),
    wrap(day_of_week),
    match_dict(week_days),
    column_category(week_days.values())
)

(res, err) = day_hot("2017-10-19")
print(res) # {"Mo": 0, "Tu": 0, "We": 0, "Th": 1, "Fr": 0, "Sa": 0, "Su": 0}
```

3.2 Event operations

Field functions has no affect on the row, so we need Event functions; maybe we need to change the value of a field; or maybe create a new field. Field functions has different interface. They receives the input, the accumulated output until this point and the error. Every Field function also returns the same, because of this we can compose all functions together in one bigger function.

The Field function has total control over the transformation step, and can affect to the full row, even fields already changed by other field functions. Because of this responsibility it's better using the supplied functions, but you can build your own.

3.2.1 List of functions

keep - Keep files

Keep is the simplest operation, no need of any field function. In essence take the value of a field from the input and put it on the output without change neither the value nor the name of field.

```
from datarefinery.tuple.TupleOperations import keep

operation = keep(["greet"])
(inp, res, err) = operation({"greet": "hello", "who": "world"}, {}, {})
print(res) # {"greet": "hello"}
```

If you need keep several similar fields you can use keep_regexp.

substitution - Value Substitution

The next operation change the value of a field with the supplied field function. This function will not change the name of the field. By example, given a to_float function, you can do this:

```
from datarefinery.tuple.TupleOperations import wrap, substitution

operation = substitution(["greet"], wrap(lambda x: len(x)))
(inp, res, err) = operation({"greet": "hello", "who": "world"}, {}, {})
print(res) # {"greet": 5}
```

append - Append new fields

Usually we need add new field or change the name of the field. We can use append to do this, but it expects a field function that return a python dictionary, where every key will be a new field. By example, given a len_cap function that will return the len of a string and the first letter in uppercase:

```
from datarefinery.tuple.TupleOperations import wrap, append

operation = append(["greet"], wrap(lambda x: {x: "you", "y": "None"}))
(inp, res, err) = operation({"greet": "hello", "who": "world"}, {}, {})
print(res) # {'hello': 'you', 'y': 'None'}
```

Notice that the field "greet" it's not in the output. Append only add the result of the function, and the function has no "greet" in their output.

fusion - several fields one output

We already define functions to change the values and add more than one field to the output. But also we can create one field from several inputs.

This function it's complex, and we have several scenarios. Also need the name of the new field. By example we can sum several numeric fields into total field.

```
from datarefinery.tuple.TupleOperations import wrap, fusion

operation = fusion(["a", "b", "c"], "sum_abc", wrap(lambda x: sum(x)))
(inp, res, err) = operation({"a": 1, "b": 2, "c": 3}, {}, {})
print(res) # {'sum_abc': 6}
```

But we can perform some decision with the input fields. Suppose that you need to change the value of money amount field by currency code field. But functions receive only one parameter input, so fusion will put all the values in a list as the input, in the same order that you request in fields field. Knowing that, we can decompose the list into the fields that we need.

By example, given a function to_eur that convert any currency into EUR:

```
def to_eur_wrapped(x):
  [currency, value] = x
  return to_eur(currency, value)
```

We can use this way with fusion:

```
from datarefinery.tuple.TupleOperations import wrap, fusion

```

```
val_eur_op = fusion(["currency", "value"], "val_eur", wrap(to_eur_wrapped))
(inp, res, err) = val_eur_op({"currency": "USD", "value": 1})
print(res) # {"val_eur": 0.8459}
```

fusion_append - Multiple values in, multiple values out

In fact it's the same that a fusion, but expect that the field function returns a python dict, in the same way than append function.

We can perform the same example that fusion but we can generate new fields with the currency as name of field and the money value as value:

```
from datarefinery.tuple.TupleOperations import wrap, fusion_append

def to_eur_cols(x):
  [currency, value] = x
  return {"EUR": to_eur(currency, value), currency: value}

val_eur_op = fusion_append(["currency", "value"], "val_eur", wrap(to_eur_cols))
(inp, res, err) = val_eur_op({"currency": "USD", "value": 1})
print(res) # {"EUR": 0.8459, "USD": 1}
```

filter_tuple - Discard irrelevant

When we don't need all data, the best approach it's discard as soon as possible the rows that we don't need.

With no_none function and filter field operation the genearted function will return None as result if the event it's discarded:

```
from datarefinery.tuple.TupleOperations import wrap, filter_tuple

no_none = filter_tuple(["value"], wrap(lambda x: x is not None))

(inp, res, err) = no_none({"value": None})
print(res) # None
```

It's up to you not to fail when result it's None, if no error means that event is discarded.

alternative - Plan B

Some times we have several ways to transform the event. If the first approach fail, alternative will try the next, until success or the last fail.

Suppose that you want to multiply by two, but if this operation fail, you want to append value 0.

```
from datarefinery.tuple.TupleOperations import wrap, alternative, substitution, append

need_value = alternative(
    substitution(["value"], wrap(lambda x: x*2)),
    append(["name"], wrap(lambda x: {"value": 0}))
)
(inp, res, err) = need_value({"name": "John"})
print(res) # {"value": 0}
```

Fail gracefully

When error occurs, at any point, we don't stop the process; keep in mind this principle when you develop your own functions. Instead exceptions we try to explain what it's going on, the deeper the better. This allow us to perform a recovery from fail as one of last steps. The recovery function search the field in error, write to the output with the provided function and if it's succsesful remove the error.

Remember that params to the no_error function are input, ouput and error.

```
from datarefinery.tuple.TupleOperations import wrap, recover

no_error = recover(["value"], wrap(lambda x: 0))
(inp, res, err) = no_error({},{},{"value": "not found"})
print(res) # {"value": 0}
print(err) # {}
```

3.2.2 Combine event operations

Normally, a transformation is a group of different type of functions, not only one type. For example, you want to keep some fields and change the value of a field using a function.

So that's why we need an interface for doing this kind of transformations. Data Refinery has an object that wraps event operations and exposes the methods. This object is *Tr*.

The most important methods are *then* and *apply*. *then* returns a new *Tr* object which contains last operations plus the operation was passed as argument with *then*. When we have all needed operations, we need to have a function to transform data. For this case, we use *apply*. This function returns a function that contains all operations and has the same interface that a row operation.

For example, if we want to keep a field and replace another field value with a x2 function (multiply by two):

```
from datarefinery.tuple.TupleOperations import wrap, keep, substitution
from datarefinery.Tr import Tr

x2 = wrap(lambda x: x*2)

tr = Tr(keep(["name"])).then(substitution(["value"], x2))
operation = tr.apply()
(inp, res, err) = operation({"name": "John", "value": 10})
print(res) # {"name": "John", "value": 20}
```

There are common mistakes as: - Add arguments to *apply*. This function only returns the complete function to use at the event.

```
from datarefinery.tuple.TupleOperations import substitution

tr = Tr(keep(["name"])).apply(substitution(["value"], x2)) # WRONG!!!
```

- Use directly the function with parenthesis as argument (the argument is the function result without parameters):

```
from datarefinery.tuple.TupleOperations import substitution
from datarefinery.Tr import Tr

tr = Tr(keep(["name"])).then(substitution(["value"], x2())) # WRONG!!!
```

In this case, we need a function as argument (the reference to the function).

This should be carefully used, in Data Refinery some functions receives data parameters (as min_max_normalization) and returns a function as result and other (as explode_date) use functions as parameters.

A world of possibilities

We can save our set of transformations every time we want using *Tr* object. This is very useful when you have training and production data.

Training data is often as production data, but it contains an extra field called "label". This field indicates what machine learning model must learn.

In next example, data transformation is a specified module of your application. You can get it with *etl* function. Then we will add the label logic:

```
from datarefinery.tuple.TupleOperations import keep

tr = etl()
if training == True:
    tr = tr.then(keep("label"))
operation = tr.apply()
```

By this way, the output will contain the label without knowing other transformations over the data during training phase.

Then - Adding new transformations

Data Refinery only works with Python dictionaries. So when data is not a dictionary, it's useful to use a data format operation before the event. *Tr* interface has a function to do this: *init*. This function takes the specified function and puts it at first on the group of event operations.

At datarefinery.tuple.Formats module you can find several functions to transform different formats to Python dictionaries. In addition, there is a function *reader* to read the data. It's an alias for *init*.

Note: if you want to use *init* with different *Tr* objects, be aware if this objects have the same origin, you will change both.

TODO: draw origin transformations

For example:

```
from datarefinery.tuple.TupleOperations import keep
from datarefinery.tuple.Formats import from_json

step1 = etl()
step2 = step1.then(keep("label"))
final = step2.init(from_json)
```

In this case, step1 and step2 have *from_json* operation at beginning. If you want different starts for every step, then you can use the next code as example:

```
from datarefinery.tuple.TupleOperations import keep
from datarefinery.tuple.Formats import from_json

step1 = etl()
step2 = etl().then(keep("label"))
final = step2.init(from_json)
```

Peek - Take a look at data

peek function works to read and handle data without any modification. You can use it when its needed to save data in a intermediate step without any transformation.

Keep in mind *peek* is invoked when *apply* is. Furthermore, the execution of the complete process is synchronous. In other words, the process doesn't finish until *peek* doesn't finish (process continues even when *peek* fails).

Note: *writer* method is an alias for *peek*.

Sequentiality

When you use *then* to create a group of operations, these are executed in one step. In other words, all operations use the same input and write in the same output. So, if we want to modify the value of a field which have been modified yet, even using *then*, all operations take place at the same time and we only see the last transformation.

For example:

```
from datarefinery.tuple.TupleOperations import wrap, substitution
from datarefinery.Tr import Tr

x2 = wrap(lambda x: x*2)

tr = Tr(substitution("value", x2)).then(substitution("value", x2))
operation = tr.apply()
(inp, res, err) = operation({"value": 2})
print(res) # {"value": 4}
```

In this example, you expect output value will be 8, but it's not true. When operations run in parallel, the execution is something like this:

input	value (1st time)	value(2nd time)
2	4	4

The input is the same for both operations and the output of the first operation was overwritten by the second one.

If you want to do different operations on the same value field, then you have to use *compose*. This function lets operations run sequentially, as the next example:

```
from datarefinery.tuple.TupleOperations import wrap, substitution, compose
from datarefinery.Tr import Tr

x2 = wrap(lambda x: x*2)

tr = Tr(substitution("value", compose(x2,x2)))
operation = tr.apply()
(inp, res, err) = operation({"value": 2})
print(res) # {"value": 8}
```

There is an alternative option for doing this. *change* is a function which uses the value of the output and substitutes the output with the result.

```
from datarefinery.tuple.TupleOperations import wrap, substitution, change
from datarefinery.Tr import Tr

x2 = wrap(lambda x: x*2)
```

```
tr = Tr(substitution("value", x2)).then(change("value", x2))
operation = tr.apply()
(inp, res, err) = operation({"value": 2})
print(res) # {"value": 8}
```

You can create your own operation using our DSL.

Other option is use *chain* function. This ends all operations before its execution and copy the output to the input. In other words, the original input disappears but you can propagate the error.

```
from datarefinery.tuple.TupleOperations import wrap, substitution, chain
from datarefinery.Tr import Tr

x2 = wrap(lambda x: x*2)

tr = Tr(substitution("value", x2)).then(chain).then(substitution("value", x2))
operation = tr.apply()
(inp, res, err) = operation({"value": 2})
print(res) # {"value": 8}
```

Please, use *chain* as last option. It's a **very dangerous** operation, because it **loses the original input**. This means if you have to do operations with original input, you can't do it after *chain* operation.

3.3 Review exercises

If you want or need to do basic exercises for review all your knowledge, you can execute the next notebook on your Jupyter instance: thebasics.ipynb.

CHAPTER 4

Tuple DSL documentation

4.1 DSL

DSL let us create any operation over a row, but you know... with great power comes great responsibility.

When you create an operation, you have to spread the input, the modified output and the error if proceeds.

Its usage is very simple. For example, *keep* function is shown below:

```
def keep(fields) -> Callable[[dict, dict, dict], Tuple[dict, dict, dict]]:
    operations = [compose(use_input(), read_field(f), write_field(f)) for f in fields]
    return reduce(compose, map(apply_over_output, operations))
```

In this example you can see a composition that takes different functions for every field. *keep* returns an unique function that contains all of them applied on output.

4.1.1 List of functions

- use_input
- use_output
- read_field
- read_match
- read_fields
- write_field
- write_error_field
- dict_enforcer
- apply_over_output
- compose

You can create your own function too. It has to return a function that takes 3 dictionaries (input, output and error) and returns them modified if it's necessary.

CHAPTER 5

Python modules

CHAPTER 6

Glossary

ETL

Extract, Transform and Load.

Field

Element of an event which was identified by name. For example, the first column of an excel sheet is identified as A field.

Row

Minimal unit of transformation at ETL.

DSL

Domain Specific Language

Event

Operations over a tuple or row.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search