ddf_utils Documentation

Release 1.0.1

Semio Zheng

Aug 02, 2021

Contents

Contents:

CHAPTER 1

Introduction

ddf_utils is a Python library and command line tool for people working with Tabular Data Package in DDF model. It provides various functions for ETL tasks, including string formatting, data transforming, generating datapackage.json, reading data form DDF datasets, running *recipes*, a decleative DSL designed to manipulate datasets to generate new datasets, and other functions we find useful in daily works in Gapminder.

1.1 Installation

Python 3.6+ is required in order to run this module.

To install this package from pypi, run:

```
$ pip install ddf_utils
```

To install from the latest source, run:

```
$ pip3 install git+https://github.com/semio/ddf_utils.git
```

1.1.1 For Windows users

If you encounter `failed to create process` when you run the ddf command, please try updating setuptools to latest version:

```
$ pip3 install -U setuptools
```

1.2 Usage

ddf_utils can be use as a library and also a commandline utility.

1.2.1 Library

ddf_utils' helper functions are divided into a few modules based on their domain, namely:

- *chef*: Recipe cooking functions. See *Recipe Cookbook (draft)* for how to write recipes
- *i18n*: Splitting/merging translation files
- *package*: Generating/updating datapackage.json
- *model.ddf* / *model.package*: Data Models for dataset and datapackage
- *patch*: Applying patch in daff format
- *qa*: Functions for QA tasks
- *str*: Functions for string/number formatting
- *transformer*: Data transforming functions, such as column/row translation, trend bridge, etc.

see above links for documents for each module.

1.2.2 Command line helper

We provide a commandline utility `ddf` for common etl tasks. For now supported commands are:

```
$ ddf --help

Usage: ddf [OPTIONS] COMMAND [ARGS]...

Options:
  --debug / --no-debug
  --help                Show this message and exit.

Commands:
  build_recipe        create a complete recipe by expanding all...
  cleanup             clean up ddf files or translation files.
  create_datapackage  create datapackage.json
  diff                give a report on the statistical differences...
  from_csv            create ddfcsv dataset from a set of csv files
  merge_translation   merge all translation files from crowdin
  new                 create a new ddf project
  run_recipe          generate new ddf dataset with recipe
  split_translation   split ddf files for crowdin translation
  validate_recipe     validate the recipe
```

run `ddf <command> --help` for detail usage on each command.

CHAPTER 2

Recipe Cookbook (draft)

This document assumes that readers already know the DDF data model. If you are not familiar with it, please refer to DDF data model document.

2.1 What is Recipe

Recipe is a Domain-specific language(DSL) for DDF datasets, to manipulate existing datasets and create new ones. By reading and running the recipe executor(Chef), one can generate a new dataset easily with the procedures we provide. Continue with *Write Your First Recipe* or *Structure of a Recipe* to learn how to use recipes. You can also check examples here

2.2 Write Your First Recipe

In this section we will go though the usage of recipes. Suppose you are a data provider and you have access to 2 DDF datasets, ddf–gapminder–population and ddf–bp–energy. Now you want to make a new dataset, which contains **oil consumption per person** data for each country. Let's do it with Recipe!

2.2.1 0. Prologue

Before you begin, you need to create a project. With the command line tool *ddf*, we can get a well designed dataset project directory. Just run `ddf new` and answer some questions. The script will generate the directory for you. When it's done we are ready to write the recipe.

As you might notice, there is a template in project_root/recipes/, and it's in YAML format. Chef support recipes in both YAML and JSON format, but we recommend YAML because it's easier to write and read. In this document we will use YAML recipes.

2.2.2 1. Add basic info

First of all, we want to add some meta data to describe the target dataset. This kind of information can be written in the `info` section. The `info` section is optional and these metadata will be also written in the datapackage.json in the final output. You can put anything you want in this section. In our example, we add the following information:

```
info:
    id: ddf--your_company--oil_per_person
    author: your_company
    version: v1
    license: MIT
    language: en
    base:
        - ddf--gapminder--population
        - ddf--gapminder--geo_entity_domain
        - ddf--bp--energy
```

2.2.3 2. Set Options

The next thing to do is to tell Chef where to look for the source datasets. We can set this option in `config` section.

```
config:
    ddf_dir: /path/to/datasets/
```

There are other options you can set in this section, check *config section* for available options.

2.2.4 3. Define Ingredients

The basic object in recipe is ingredient. A ingredient defines a collection of data which comes form existing dataset or result of computation between several ingredients. To define ingredients form existing datasets or csv files, we append to the `ingredients` section; to define ingredients on the fly, we append to the `cooking` section.

The `ingredients` section is a list of ingredient objects. An ingredient that reads data from a data package should be defined with following parameters:

- `id`: the name of the ingredient
- `dataset`: the dataset where the ingredient is from
- `key`: the primary keys to filter from datapackage, should be comma seperated strings
- `value`: a list of concept names to filter from the result of filtering keys, or pass "*" to select all. Alternatively a mongodb-like query can be used.
- `filter`: optional, only select rows match the filter. The filter is a dictionary where keys are colunm names and values are values to filter. Alternatively a mongodb-like query can be used.

There are more parameters for ingredient definition, see the *ingredients section* document.

In our example, we need datapoints from both gapminder population dataset and oil consumption datapoints from bp dataset. Noticing the bp is using lower case short names for its geo and gapminder is using 3 letter iso for its country entities, we should align them to use one system too. So we end up with below ingredients:

```
ingredients:
    - id: oil-consumption-datapoints
      dataset: ddf--bp--energy
      key: geo, year
```

(continues on next page)

(continued from previous page)

```
      value:
          - oil_consumption_tonnes
    - id: population-datapoints
      dataset: ddf--gapminder--population
      key: country, year
      value: "*"  # note that the * symbol is reserved symbol in yaml,
                  # we should quote it if we mean a string
    - id: bp-geo-entities
      dataset: ddf--bp--energy
      key: geo
      value: "*"
    - id: gapminder-country-synonyms
      dataset: ddf--gapminder--population
      key: country, synonym
      value: "*"
```

2.2.5 4. Add Cooking Procedures

We have all ingredients we need, the next step is to cook with these ingredients. In recipe we put all cooking procedures under the `cooking` section. Because in DDF model we have 3 kinds of collections: `concepts`, `datapoints` and `entities`, we divide the cooking section into 3 corresponding sub-sections, and in each section will be a list of `procedures`. So the basic format is:

```
cooking:
    concepts:
        # procedures for concepts here
    entities:
        # procedures for entities here
    datapoints:
        # procedures for datapoints here
```

Procedures are like functions. They take ingredients as input, operate with options, and return new ingredients as result. For a complete list of supported procedures, see *Available Procedures*. With this in mind, we can start writing our cooking procedures. Suppose after some discussion, we decided our task list is:

- datapoints: oil consumption per capita, and use country/year as dimensions.
- entities: use the country entities from Gapminder
- concepts: all concepts from datapoints and entities

Firstly we look at datapoints. What we need to do to get what we need are:

1. change the dimensions to country/year for bp and gapminder datapoints
2. align bp datapoints to use gapminder's country entities
3. calculate per capita data

We can use *translate_header*, *translate_column*, *merge*, *run_op* to get these tasks done.

```
datapoints:
    # change dimension for bp
    - procedure: translate_header
      ingredients:
          - bp-datapoints
      options:
```

(continues on next page)

(continued from previous page)

```
        dictionary:
            geo: country
    result: bp-datapoints-translated

  # align bp geo to gapminder country
  - procedure: translate_column
    ingredients:
        - bp-geo-entities
    result: bp-geo-translated
    options:
        column: geo_name  # the procedure will search for values in this column
        target_column: country  # ... and put the matched value in this column
        dictionary:
            base: gapminder-country-synonyms
            # key is the columns to search for match of geo names
            key: synonym
            # value is the column to get new value
            value: country

    # align bp datapoints to new bp entities
    - procedure: translate_column
      ingredients:
          - bp-datapoints-translated
      result: bp-datapoints-translated-aligned
      options:
          column: country
          target_column: country
          dictionary:
              base: bp-geo-translated
              key: geo
              value: country

    # merge bp/gapminder data and calculate the result
    - procedure: merge
      ingredients:
          - bp-datapoints-translated-aligned
          - population-datapoints
      result: bp-population-merged-datapoints
    - procedure: run_op
      ingredients:
          - bp-population-merged-datapoints
      option:
          op:
              oil_consumption_per_capita: |
                  oil_consumption_tonnes * 1000 / population
      result: datapoints-calculated
    # only keep the indicator we need
    - procedure: filter
      ingredients:
          - datapoints-calculated
      options:
          item:
              - oil_consumption_per_capita
      result: datapoints-final
```

For entities, we will just use the country entities from gapminder, so we can skip this part. For concepts, we need to extract concepts from the ingredients:

```
concepts:
    - procedure: extract_concepts
      ingredients:
          - datapoints-final
          - gapminder-country-entities
      result: concepts-final
      options:
          overwrite:  # manually set some concept_types
              year: time
              country: entity_domain
```

2.2.6 5. Serve Dishes

After all these procedure, we have cook the dishes and it's time to serve it! In recipe we can set which ingredients are we going to serve(save to disk) in the `serving` section. Note that this section is optional, and if you don't specify then the last procedure of each sub-section of `cooking` will be served.

```
serving:
    - id: concepts-final
    - id: gapminder-country-entities
    - id: datapoints-final
```

Now we have finished the recipe. For the complete recipe, please check this gist.

2.2.7 6. Running the Recipe

To run the recipe to generate the dataset, we use the ddf command line tool. Run the following command and it will cook for you and result will be saved into `out_dir`.

```
ddf run_recipe -i example.yml -o out_dir
```

If you want to just do a dry run without saving the result, you can run with the `-d` option.

```
ddf run_recipe -i example.yml -d
```

Now you have learned the basics of Recipe. We will go though more details in Recipe in the next sections.

2.3 Structure of a Recipe

A recipe is made of following parts:

- basic info
- configuration
- includes
- ingredients
- cooking procedures
- serving section

A recipe file can be in either json or yaml format. We will explain each part of recipe in details in the next sections.

2.3.1 info section

All basic info are stored in `info` section of the recipe. an `id` field is required inside this section. Any other information about the new dataset can be store inside this section, such as `name`, `provider`, `description` and so on. Data in this section will be written into datapackage.json file of the generated dataset.

2.3.2 config section

Inside `config` section, we define the configuration of dirs. currently we can set below path:

- `ddf_dir`: the directory that contains all ddf csv repos. Must set this variable in the main recipe to run with chef, or provide as an command line option using the `ddf` utility.
- `recipes_dir`: the directory contains all recipes to include. Must set this variable if we have `include` section. If relative path is provided, the path will be related to the path of the recipe.
- `dictionary_dir`: the directory contains all translation files. Must set this variable if we have json file in the options of procedures. (translation will be discussed later). If relative path is provided, the path will be related to the path of the recipe.
- `procedures_dir`: when you want to use *custom procedures*, you should set this option to tell which dir the procedures are in.

2.3.3 include section

A recipe can include other recipes inside itself. to include a recipe, simply append the filename to the `include` section. note that it should be a absolute path or a filename inside the `recipes_dir`.

2.3.4 ingredients section

A recipe must have some ingredients for cooking. There are 2 places where we can define ingredients in recipe:

- in `ingredients` section
- in the `ingredients` parameter in procedures, which is called on-the-fly ingredients

in either case, the format of ingredient definition object is the same. An ingredient should be defined with following parameters:

- `id`: the name of the ingredient, which we can refer later in the procedures. `id` is optional when the ingredient is in a procedure object.
- `dataset` or `data`: one of them should be defined in the ingredient. Use `dataset` when we want to read data from an dataset, and use `data` when we want to read data from a csv file.
- `key`: the primary keys to filter from datapackage, should be comma seperated strings
- `value`: optional, a list of concept names to filter from the result of filtering keys, or pass "*" to select all. Mongo-like queries are also supported, see examples below. If omitted, assume "*".
- `filter`: optional, only select rows match the filter. The filter is a dictionary where keys are colunm names and values are values to filter. Mongo-like queries are also supported, see examples below and examples in `filter` procedure.

Here is an example ingredient object in recipe:

```
id: example-ingredient
dataset: ddf--example--dataset
key: "geo,time"  # key columns of ingredient
value:  # only include concepts listed here
  - concept_1
  - concept_2
filter:  # select rows by column values
  geo:  # only keep datapoint where `geo` is in [swe, usa, chn]
    - swe
    - usa
    - chn
```

`value` and `filter` can accept mongo like queries to make more complex statements, for example:

```
id: example-ingredient
dataset: ddf--example--dataset
key: geo, time
value:
    $nin:  # exclude following indicators
        - concept1
        - concept2
filter:
    geo:
        $in:
            - swe
            - usa
            - chn
    year:
        $and:
            $gt: 2000
            $lt: 2015
```

for now, value accepts `$in` and `$nin` keywords, but only one of them can be in the value option; filter supports logical keywords: `$and`, `$or`, `$not`, `$nor`, and comparision keywords: `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$in`, `$nin`.

The other way to define the ingredient data is using the `data` keyword to include external csv file, or inline the data in the ingredient definition. Example:

```
id: example-ingredient
key: concept
data: external_concepts.csv
```

You can also create On-the-fly ingredient:

```
id: example-ingredient
key: concept
data:
    - concept: concept_1
      name: concept_name_1
      concept_type: string
      description: concept_description_1
    - concept: concept_2
      name: concept_name_2
      concept_type: measure
      description: concept_description_2
```

2.3.5 cooking section

`cooking` section is a dictionary contains one or more list of procedures to build a dataset. valid keys for cooking section are *datapoints*, *entities*, *concepts*.

The basic format of a procedure is:

```
procedure: proc_name
ingredients:
  - ingredient_to_run_the_proc
options:  # options object to pass to the procedure
  foo: baz
result: id_of_new_ingredient
```

Available procedures will be shown in the below *section*.

2.3.6 serving section and serve procedure

For now there are 2 ways to tell chef which ingredients should be served, and you can choose one of them, but not both.

serve procedure

`serve` procedure should be placed in `cooking` section, with the following format:

```
procedure: serve
ingredients:
  - ingredient_to_serve
options:
  opt: val
```

multiple serve procedures are allowed in each cooking section.

serving section

`serving` section should be a top level object in the recipe, with following format:

```
serving:
  - id: ingredient_to_serve_1
    options:
      opt: val
  - id: ingredient_to_serve_2
    options:
      foo: baz
```

available options

- `digits` : *int*, controls how many decimal should be kept at most in a numeric ingredient.
- `no_keep_sets` : *bool*, by default chef will serve the entities by entity_sets, i.e. each entity set will have one file. Enabling this will make chef serve entire domain in one file, no separated files

2.4 Recipe execution

To run a recipe, you can use the `ddf run_recipe` command:

```
$ ddf run_recipe -i path_to_rsecipe.yaml -o output_dir
```

You can specify the path where your datasets are stored:

```
$ ddf run_recipe -i path_to_recipe.yaml -o output_dir --ddf_dir path_to_datasets
```

Internally, the process to generate a dataset have following steps:

- read the main recipe into Python object
- if there is include section, read each file in the include list and expand the main recipe
- if there is file name in dictionary option of each procedure, try to expand them if the option value is a filename
- check if all datasets are available
- build a procedure dependency tree, check if there are loops in it
- if there is no `serve` procedure and `serving` section, the last procedure result for each section will be served. If there is `serve` procedure or `serving` section, chef will serve the result as described
- run the procedures for each ingredient to be served and their dependencies
- save output to disk

If you want to embed the function into your script, you can write script like this:

```
import ddf_utils.chef as chef

def run_recipe(recipe_file, outdir):
    recipe = chef.build_recipe(recipe_file)  # get all sub-recipes and dictionaries
    res = chef.run_recipe(recipe)  # run the recipe, get output for serving
    chef.dishes_to_disk(res)  # save output to disk

run_recipe(path_to_recipe, outdir)
```

2.5 Available procedures

Currently supported procedures:

- *translate_header*: change ingredient data header according to a mapping dictionary
- *translate_column*: change column values of ingredient data according to a mapping dictionary
- *merge*: merge ingredients together on their keys
- *groupby*: group ingredient by columns and do aggregate/filter/transform
- *window*: run function on rolling windows
- *filter*: filter ingredient data with Mongo-like query
- *filter_row*: filter ingredient data by column values
- *filter_item*: filter ingredient data by concepts
- *run_op*: run math operations on ingredient columns
- *extract_concepts*: generate concepts ingredient from other ingredients
- *trend_bridge*: connect 2 ingredients and make custom smoothing
- *flatten*: flatten dimensions in the indicators to create new indicators

- *split_entity*: split an entity and create new entity from it
- *merge_entity*: merge some entity to create a new entity

2.5.1 translate_header

Change ingredient data header according to a mapping dictionary.

usage and options

```
procedure: translate_header
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingledient id
options:
  dictionary: str or dict  # file name or mappings dictionary
```

notes

- if `dictionary` option is a dictionary, it should be a dictionary of oldname -> newname mappings; if it's a string, the string should be a json file name that contains such dictionary.
- currently chef only support one ingredient in the `ingredients` parameter

2.5.2 translate_column

Change column values of ingredient data according to a mapping dictionary, the dictionary can be generated from an other ingredient.

usage and options

```
procedure: translate_column
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingledient id
options:
  column: str  # the column to be translated
  target_column: str  # optinoal, the target column to store the translated data
  not_found: {'drop', 'include', 'error'}  # optional, the behavior when there is␣
↪values not found in the mapping dictionary, default is 'drop'
  ambiguity: {'prompt', 'skip', 'error'}  # optional, the behavior when there is␣
↪ambiguity in the dictionary
  dictionary: str or dict  # file name or mappings dictionary
```

notes

- If `base` is provided in `dictionary`, `key` and `value` should also in `dictionary`. In this case chef will generate a mapping dictionary using the `base` ingredient. The dictionary format will be:

```
dictionary:
    base: str  # ingredient name
    key: str or list  # the columns to be the keys of the dictionary, can accept a␣
↪list
    value: str  # the column to be the values of the the dictionary, must be one␣
↪column
```

- currently chef only support one ingredient in the `ingredients` parameter

examples

here is an example when we translate the BP geo names into Gapminder's

```
procedure: translate_column
ingredients:
    - bp-geo
options:
    column: name
    target_column: geo_new
    dictionary:
        base: gw-countries
        key: ['alternative_1', 'alternative_2', 'alternative_3',
            'alternative_4_cdiac', 'pandg', 'god_id', 'alt_5', 'upper_case_name',
            'iso3166_1_alpha2', 'iso3166_1_alpha3', 'arb1', 'arb2', 'arb3', 'arb4',
            'arb5', 'arb6', 'name']
        value: country
    not_found: drop
result: geo-aligned
```

2.5.3 merge

Merge ingredients together on their keys.

usage and options

```
procedure: merge
ingredients:  # list of ingredient id
  - ingredient_id_1
  - ingredient_id_2
  - ingredient_id_3
  # ...
result: str  # new ingledient id
options:
  deep: bool  # use deep merge if true
```

notes

- The ingredients will be merged one by one in the order of how they are provided to this function. Later ones will overwrite the pervious merged results.
- **deep merge** is when we check every datapoint for existence if false, overwrite is on the file level. If key-value (e.g. geo,year-population_total) exists, whole file gets overwritten if true, overwrite is on the row level. If values (e.g. afr,2015-population_total) exists, it gets overwritten, if it doesn't it stays

2.5.4 groupby

Group ingredient by columns and do aggregate/filter/transform.

usage and options

```
procedure: groupby
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingledient id
options:
```

(continues on next page)

(continued from previous page)

```
  groupby: str or list  # columm(s) to group
  aggregate: dict  # function block
  transform: dict  # function block
  filter: dict  # function block
  insert_key: dict  # manually add columns
```

notes

- Only one of `aggregate`, `transform` or `filter` can be used in one procedure.
- Any columns not mentioned in groupby or functions are dropped.
- If you want to add back dropped columns with same values, use `insert_key` option.
- Currently chef only support one ingredient in the `ingredients` parameter

function block

Two styles of function block are supported, and they can mix in one procedure:

```
aggregate:  # or transform, filter
  col1: sum  # run sum to col1
  col2: mean
  col3:  # run foo to col3 with param1=baz
    function: foo
    param1: baz
```

also, we can use wildcard in the column names:

```
aggregate:  # or transform, filter
  "population*": sum  # run sum to all indicators starts with "population"
```

2.5.5 window

Run function on rolling windows.

usage and options

```
procedure: window
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingledient id
options:
  window:
    column: str  # column which window is created from
    size: int or 'expanding'  # if int then rolling window, if expanding then␣
↪expanding window
    min_periods: int  # as in pandas
    center: bool  # as in pandas
  aggregate: dict
```

function block

Two styles of function block are supported, and they can mix in one procedure:

```
aggregate:
  col1: sum  # run rolling sum to col1
  col2: mean  # run rolling mean to col2
```

(continues on next page)

(continued from previous page)

```
  col3:  # run foo to col3 with param1=baz
    function: foo
    param1: baz
```

notes

- currently chef only support one ingredient in the `ingredients` parameter

2.5.6 filter

Filter ingredient data with Mongo-like queries. You can filter the ingredient by item, which means indicators in datapoints or columns in other type of ingredients, and/or by row.

`item` filter accepts a list of items, or a list followed by `$in` or `$nin`. `row` filter accepts a query similar to mongo queries, supportted keywords are `$and`, `$or`, `$eq`, `$ne`, `$gt`, `$lt`. See below for an example.

usage and options:

```
- procedure: filter
  ingredients:
      - ingredient_id
  options:
      item:  # just as `value` in ingredient definition
          $in:
              - concept_1
              - concept_2
      row:  # just as `filter` in ingredient definition
          $and:
              geo:
                  $ne: usa
              year:
                  $gt: 2010
   result: output_ingredient
```

for more information, see the *ddf_utils.chef.model.ingredient.Ingredient* class and *ddf_utils.chef.procedure.filter()* function.

2.5.7 run_op

Run math operations on ingredient columns.

usage and options

```
procedure: run_op
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingledient id
options:
  op: dict  # column name -> calculation mappings
```

notes

- currently chef only support one ingredient in the `ingredients` parameter

Examples

for exmaple, if we want to add 2 columns, `col_a` and `col_b`, to create an new column, we can write

```
procedure: run_op
ingredients:
  - ingredient_to_run
result: new_ingredient_id
options:
  op:
    new_col_name: "col_a + col_b"
```

2.5.8 extract_concepts

Generate concepts ingredient from other ingredients.

usage and options

```
procedure: extract_concepts
ingredients:  # list of ingredient id
  - ingredient_id_1
  - ingredient_id_2
result: str  # new ingledient id
options:
  join:  # optional
    base: str  # base concept ingredient id
    type: {'full_outer', 'ingredients_outer'}  # default is full_outer
  include_keys: true  # if we should include the primaryKeys of the ingredients
  overwrite:  # overwirte some of the concept types
      year: time
```

notes

- all concepts in ingredients in the `ingredients` parameter will be extracted to a new concept ingredient
- `join` option is optional; if present then the `base` will merge with concepts from `ingredients`
- `full_outer` join means get the union of concepts; `ingredients_outer` means only keep concepts from `ingredients`

2.5.9 trend_bridge

Connect 2 ingredients and make custom smoothing.

usage and options

```
- procedure: trend_bridge
  ingredients:
    - data_ingredient                # optional, if not set defaults to empty␣
↪ingredient
  options:
    bridge_start:
      ingredient: old_data_ingredient # optional, if not set then assume it's the␣
↪input ingredient
      column: concept_old_data
    bridge_end:
      ingredient: new_data_ingredient # optional, if not set then assume it's the␣
↪input ingredient
      column: concept_new_data
    bridge_length: 5                 # steps in time. If year, years, if days, days.
```

(continues on next page)

(continued from previous page)

```
    bridge_on: time                    # the index column to build the bridge with
    target_column: concept_in_result   # overwrites if exists. creates if not exists.
                                       # defaults to bridge_end.column
  result: data_bridged
```

2.5.10 flatten

Flatten dimension to create new indicators.

This procedure only applies for datapoints ingredients.

usage and options

```
- procedure: flatten
  ingredients:
    - data_ingredient
  options:
    flatten_dimensions:  # a list of dimensions to be flattened
      - entity_1
      - entity_2
    dictionary:  # old name -> new name mappings, supports wildcard and template.
      "old_name_wildcard": "new_name_{entity_1}_{entity_2}"
```

example

For example, if we have datapoints for population by gender, year, country. And gender entity domain has `male` and `female` entity. And we want to create 2 seperated indicators: `population_male` and `population_female`. The procedure should be:

```
- procedure: flatten
  ingredients:
    - population_by_gender_ingredient
  options:
    flatten_dimensions:
      - gender
    dictionary:
      "population": "{concept}_{gender}"  # concept will be mapped to the concept␣
↪name being flattened
```

2.5.11 split_entity

(WIP) split an entity into several entities

2.5.12 merge_entity

(WIP) merge several entities into one new entity

2.5.13 custom procedures

You can also load your own procedures. The procedure name should be `module.function`, where `module` should be in the `procedures_dir` or other paths in `sys.path`.

The procedure should be defined as following structure:

```
from ddf_utils.chef.model.chef import Chef
from ddf_utils.chef.model.ingredient import ProcedureResult
from ddf_utils.chef.helpers import debuggable

@debuggable  # adding debug option to the procedure
def custom_procedure(chef, ingredients, result, **options):
    # you must have chef(a Chef object), ingredients (a list of string),
    # result (a string) as parameters
    #

    # procedures...

    # and finally return a ProcedureResult object
    return ProcedureResult(chef, result, primarykey, data)
```

Check our predefined procedures for examples.

2.5.14 Checking Intermediate Results

Most of the procedures supports `debug` option, which will save the result ingredient to `_debug/<ingredient_id>/` folder of your working directory. So if you want to check the intermediate results, just add `debug: true` to the `options` dictionary.

2.6 Validate the Result with ddf-validation

After generating the dataset, it would be good to check if the output dataset is valid against the DDF CSV model. There is a tool ddf-validation for that, which is written in nodejs.

to check if a dataset is valid, install ddf-validation and run:

```
cd path_to_your_dataset
validate-ddf
```

2.7 Validate Recipe with Schema

In ddf_utils we provided a command for recipe writers to check if the recipe is valid using a JSON schema for recipe. The following command check and report any errors in recipe:

```
$ ddf validate_recipe input.yaml
```

Note that if you have includes in your recipe, you may want to build a complete recipe before validating it. You can firstly build your recipe and validate it:

```
$ ddf build_recipe input.yaml > output.json

$ ddf validate_recipe output.json
```

or just run `ddf validate_recipe --build input.yaml` without creating a new file.

The validate command will output the json paths that are invalid, so that you can easily check which part of your recipe is wrong. For example,

```
$ ddf validate_recipe --build etl.yml
On .cooking.datapoints[7]
{'procedure': 'translate_header', 'ingredients': ['unpop-datapoints-pop-by-age-aligned
↪'], 'options': {'dictionary_f': {'country_code': 'country'}}, 'result': 'unpop-
↪datapoints-pop-by-age-country'} is not valid under any of the given schemas
```

For a pretty printed output of the invalid path, try using json processors like jq:

```
// $ ddf build_recipe etl.yml | jq ".cooking.datapoints[7]"
{
  "procedure": "translate_header",
  "ingredients": [
    "unpop-datapoints-pop-by-age-aligned"
  ],
  "options": {
    "dictionary_f": {
      "country_code": "country"
    }
  },
  "result": "unpop-datapoints-pop-by-age-country"
}
```

Other then the json schema, we can also valiate recipe using `dhall`, as we will talk about in next section.

2.8 Write recipe in Dhall

Sometimes there will be recurring tasks, for example, you might applying same procedures again and again to different ingredients. In this case we would benefit from Dhall language. Also, there are more advantages on using Dhall over yaml, such as type checking. We provide type definitions for the recipe in an other repo. Check examples in the repo to see how to use them.

2.9 General guidelines for writing recipes

- if you need to use `translate_header` / `translate_column` in your recipe, place them at the beginning of recipe. This can improve the performance of running the recipe.
- run recipe with `ddf --debug run_recipe` will enable debug output when running recipes. use it with the `debug` option will help you in the development of recipes.

2.10 The Hy Mode

From Hy's home page:

> Hy is a wonderful dialect of Lisp that's embedded in Python.
>
> Since Hy transforms its Lisp code into the Python Abstract Syntax Tree, you have the whole beautiful world of Python at your fingertips, in Lisp form!

Okay, if you're still with me, then let's dive into the world of Hy recipe!

We provided some macros for writing recipes. They are similar to the sections in YAML:

```
;; import all macros
(require [ddf_utils.chef.hy_mod.macros [*]])

;; you should call init macro at the beginning.
;; This will initial a global variable *chef*
(init)

;; info macro, just like the info section in YAML
(info :id "my_fancy_dataset"
      :date "2017-12-01")

;; config macro, just like the config section in YAML
(config :ddf_dir "path_to_ddf_dir"
        :dictionary_dir "path_to_dict_dir")

;; ingredient macro, each one defines an ingredient. Just like a
;; list element in ingredients section in YAML
(ingredient :id "datapoints-source"
            :dataset "source_dataset"
            :key "geo, year")

;; procedure macro, each one defines a procedure. Just like an element
;; in the cooking blocks. First 2 parameters are the result id and the
;; collection it's in.
(procedure "result-ingredient" "datapoints"
           :procedure "translate_header"
           :ingredients ["datapoints-source"]
           :options {:dictionary
                     {"geo" "country"}})  ;; it doesn't matter if you use keyword or␣
↪plain string
                                          ;; for the options dictionary's key

 ;; serve macro
 (serve :ingredients ["result-ingredient"]
        :options {"digits" 2})

 ;; run the recipe
 (run)

 ;; you can do anything to the global chef element
 (print (*chef*.to_recipe))
```

There are more examples in the example folder.

CHAPTER 3

Downloading Data from Source Providers

ddf_utils provides a few classes under `ddf_utils.factory` module to help downloading data from serval data providers. Currently we support downloading files from Clio-infra, IHME GBD, ILOStat, OECD and WorldBank. Whenever possible we will use bulk download method from the provider.

3.1 General Interface

We created a general class for these data loaders, which has a `metadata` property and `load_metadata`, `has_newer_source` and `bulk_download` methods.

`metadata` is for all kinds of metadata provided by data source, such as dimension list and available values in a dimension. `load_metadata` tries to load these metadata. `has_newer_source` tries to find out if there is newer version of source data available. And `bulk_download` download the requested data files.

3.2 IHME GBD Loader

Note: The API we use in the IHME loader was not documented anywhere in the IHME website. So there may be problems in the loader.

The IHME GBD loader works in the same way as the GBD Result Tool. Just like one would do a search in the result tool, we need to select context/country/age etc. In IHME loader, we should provide a dictionary of query parameters to the *`bulk_download`* method (See the docstring for bulk_download for the usage). And the values for them should be the numeric IDs from IHME. We can check these ID from metadata.

Example Usage:

```
In [1]: from ddf_utils.factory.ihme import IHMELoader

In [2]: l = IHMELoader()
```

(continues on next page)

(continued from previous page)

```
In [3]: md = l.load_metadata()

In [4]: md.keys()
Out[4]: dict_keys(['age', 'cause', 'groups', 'location', 'measure', 'metric', 'rei',␣
↪'sequela', 'sex', 'year', 'year_range', 'version'])

In [5]: md['age'].head()
Out[5]:
id      name short_name  sort  plot       type
1    1   Under 5         <5    22     0  aggregate
10  10  25 to 29         25    10     1   specific
11  11  30 to 34         30    11     1   specific
12  12  35 to 39         35    12     1   specific
13  13  40 to 44         40    13     1   specific

In [6]: l.bulk_download('/tmp/', context='le', version=376, year=[2017],␣
↪email='your-email@mailer.com')
working on https://s3.healthdata.org/gbd-api-2017-public/xxxx
check status as http://ghdx.healthdata.org/gbd-results-tool/result/xxxx
available downloads:
http://s3.healthdata.org/gbd-api-2017-public/xxxx_files/IHME-GBD_2017_DATA-03cf30ab-1.
↪zip
downloading http://s3.healthdata.org/gbd-api-2017-public/xxxx_files/IHME-GBD_2017_
↪DATA-03cf30ab-1.zip to /tmp/xxxx/IHME-GBD_2017_DATA-xxxx-1.zip
1.13MB [00:01, 582kB/s]
Out[6]: ['03cf30ab']
```

3.3 ILOStat Loader

The ILO data loader use the bulk download facility from ILO.

See the API doc for how to use this loader.

3.4 WorldBank Loader

The Worldbank loader can download all datasets listed in the data catalog in CSV(zip) format.

Example Usage:

```
In [1]: from ddf_utils.factory.worldbank import WorldBankLoader

In [2]: w = WorldBankLoader()

In [3]: md = w.load_metadata()

In [4]: md.head()
Out[4]:
                    accessoption acronym api                          apiaccessurl  .
↪..
0  API, Bulk download, Query tool     WDI   1  http://data.worldbank.org/developers  .
↪..
1  API, Bulk download, Query tool     ADI   1  http://data.worldbank.org/developers  .
↪..
```

(continues on next page)

(continued from previous page)

```
2  API, Bulk download, Query tool     GEM   1  http://data.worldbank.org/developers  .
↪..
3                      Query tool     NaN   0                                    NaN  .
↪..
4  API, Bulk download, Query tool    MDGs   1  http://data.worldbank.org/developers  .
↪..
...

In [5]: w.bulk_download('MDGs', '/tmp/')
Out[5]: '/tmp/'
```

3.5 OECD Loader

The OECD loader can download all datasets in OECD stats. We use the SDMX-JSON api and the downloaded dataset will be in json file. Learn more about SDMX-JSON in the OECD api doc.

Example Usage:

```
In [1]: from ddf_utils.factory.oecd import OECDLoader

In [2]: o = OECDLoader()

In [3]: md = o.load_metadata()

In [4]: # metadata contains all available datasets.

In [5]: md.head()
Out[5]:
id                                          name
0          QNA                Quarterly National Accounts
1      PAT_IND                           Patent indicators
2  SNA_TABLE11     11. Government expenditure by function (COFOG)
3    EO78_MAIN  Economic Outlook No 78 - December 2005 - Annua...
4        ANHRS    Average annual hours actually worked per worker

In [6]: o.bulk_download('/tmp/', 'EO78_MAIN')
```

3.6 Clio-infra Loader

The Clio infra loader parse the home page for clio infra and do bulk download for all datasets or all country profiles.

Example Usage:

```
In [1]: from ddf_utils.factory.clio_infra import ClioInfraLoader

In [2]: c = ClioInfraLoader()

In [3]: md = c.load_metadata()

In [4]: md.head()
Out[4]:
                  name                                        url     type
```

(continues on next page)

(continued from previous page)

```
0    Cattle per Capita    ../data/CattleperCapita_Compact.xlsx  dataset
1  Cropland per Capita  ../data/CroplandperCapita_Compact.xlsx  dataset
2     Goats per Capita     ../data/GoatsperCapita_Compact.xlsx  dataset
3   Pasture per Capita   ../data/PastureperCapita_Compact.xlsx  dataset
4      Pigs per Capita      ../data/PigsperCapita_Compact.xlsx  dataset

In [5]: md['type'].unique()
Out[5]: array(['dataset', 'country'], dtype=object)

In [6]: c.bulk_download('/tmp', data_type='dataset')
downloading https://clio-infra.eu/data/CattleperCapita_Compact.xlsx to /tmp/Cattle␣
↪per Capita.xlsx
...
```

CHAPTER 4

Use ddf_utils for ETL tasks

4.1 Create DDF dataset from non-DDF data files

If you want to learn how to compose DDF datasets, read *Recipe Cookbook (draft)*. If you are not familiar with DDF model, please refer to DDF data model document.

ddf_utils provides most of data classes and methods in terms of the DDF model: concept/entity/datapoint/synonmy (and more to come.) Together with the other utility functions, we hope to provide a tool box for users to easily create a DDF dataset. To see it in action, check this notebook for a demo.

In general, we are building scripts to transform data from one format to the other format, so guidelines for programming and data ETL applies here. You should care about the correctness of the scripts and be ware of bad data.

4.2 Create DDF dataset from CSV file

When you have clean CSV data file, you can use the `ddf from_csv` command to create DDF dataset. Currently only one format is supported: Primary Keys as well as all indicators should be in columns.

```
ddf from_csv -i input_file_or_path -o out_path
```

Where `-i` sets the input file or path and when it is a path all files in the path will be proceed; `-o` sets the path the generated DDF dataset will be put to. If `-i` is not set, it defaults to current path.

4.3 Compare 2 datasets

`ddf diff` command compares 2 datasets and return useful statistics for each indicator.

```
ddf diff -i indicator1 -i indicator2 dataset1 dataset2
```

For now this command supports following statistics:

- `rval`: the standard correlation coefficient
- `avg_pct_chg`: average percentage changes
- `max_pct_chg`: the maximum of change in percentage
- `rmse`: the root mean squared error
- `nrmse`: equals `rmse`/(max - min) where max and min are calculated with data in dataset2
- `new_datapoints`: datapoints in dataset1 but not dataset2
- `dropped_datapoints`: datapoints in dataset2 but not dataset1

If no indicator specified in the command, `rmse` and `nrmse` will be calculated.

Note: Please note that rval and avg_pct_chg assumes there is a `geo` column in datapoints, which is not very useful for now. We will improve this later.

You can also compare 2 commits for a git folder too. In this case you should run

```
cd dataset_path

ddf diff --git -o path/to/export/to -i indicator head_ref base_ref
```

Because the script needs to export different commits for the git repo, you should provide the `-o` flag to set which path you'd like to put the exported datasets into.

CHAPTER 5

API References

5.1 ddf_utils package

5.1.1 Subpackages

5.1.1.1 ddf_utils.chef package

Subpackages

ddf_utils.chef.model package

ddf_utils.chef.model.chef

The Chef object

class ddf_utils.chef.model.chef.**Chef**(*dag: ddf_utils.chef.model.dag.DAG = None*, *metadata=None*, *config=None*, *cooking=None*, *serving=None*, *recipe=None*)

Bases: object

the chef api

add_config(***config*)
add configs, all keyword args will be added/replace existing in config dictionary

add_dish(*ingredients*, *options=None*)

add_ingredient(***kwargs*)
add a new ingredient in DAG.

keyword arguments will send as a dictionary to the `dictionary` keyword of *`ddf_utils.chef.model.ingredient.ingredient_from_dict()`* method.

add_metadata(***metadata*)
add metadata, all keyword args will be added/replace existing in metadata dictionary

add_procedure(*collection*, *procedure*, *ingredients*, *result=None*, *options=None*)

config

copy()

classmethod from_recipe(*recipe_file*, ***config*)

ingredients

static register_procedure(*func*)

run(*serve=False*, *outpath=None*)

serving

to_graph(*node=None*)

to_recipe(*fp=None*)
write chef in yaml recipe format

validate()
validate if the chef is good to run.

The following will be tested:

1. check if datasets required by ingredients are available
2. check if procedures are available
3. check if the DAG is valid. i.e no dependency cycle, no missing dependency.

ddf_utils.chef.model.dag

the DAG model of chef

The DAG consists of 2 types of nodes: IngredientNode and ProcedureNode. each node will have a *evaluate()* function, which will return an ingredient on eval.

class ddf_utils.chef.model.dag.**BaseNode**(*node_id*, *chef*)
Bases: object

The base node which IngredientNode and ProcedureNode inherit from

Parameters

- **node_id** (*str*) – the name of the node
- **dag** (DAG) – the *DAG* object the node is in

add_downstream(*node*)

add_upstream(*node*)

detect_missing_dependency()
check if every upstream is available in the DAG. raise error if something is missing

downstream_list

evaluate()

get_direct_relatives(*upstream=False*)
Get the direct relatives to the current node, upstream or downstream.

upstream_list

class ddf_utils.chef.model.dag.**DAG**(*node_dict=None*)
Bases: object

The DAG model.

A dag (directed acyclic graph) is a collection of tasks with directional dependencies. DAGs essentially act as namespaces for its nodes. A node_id can only be added once to a DAG.

add_dependency(*upstream_node_id*, *downstream_node_id*)
Simple utility method to set dependency between two nodes that already have been added to the DAG using add_node()

add_node(*node*)
add a node to DAG

copy()

detect_cycles()
Detect cycles in DAG, following Tarjan's algorithm.

get_node(*node_id*)

has_node(*node_id*)

node_dict

nodes
return all nodes

roots
return the roots of the DAG

tree_view()
Shows an ascii tree representation of the DAG

class ddf_utils.chef.model.dag.**IngredientNode**(*node_id*, *ingredient*, *chef*)
Bases: *ddf_utils.chef.model.dag.BaseNode*

Node for storing dataset ingredients.

Parameters **ingredient** (Ingredient) – the ingredient in this node

evaluate() → ddf_utils.chef.model.ingredient.Ingredient
return the ingredient as is

class ddf_utils.chef.model.dag.**ProcedureNode**(*node_id*, *procedure*, *chef*)
Bases: *ddf_utils.chef.model.dag.BaseNode*

The node for storing procedure results

The evaluate() function will run a procedure according to *self.procedure*, using other nodes' data. Other nodes will be evaluated if when necessary.

Parameters **procedure** (*dict*) – the procedure dictionary

evaluate() → ddf_utils.chef.model.ingredient.Ingredient

ddf_utils.chef.model.ingredient

main ingredient class

class ddf_utils.chef.model.ingredient.**Ingredient**(*id: str, key: Union[list, str], value: Union[list, dict, str] = '*', dataset: str = None, data: dict = None, row_filter: dict = None, base_dir: str = './'*)

Bases: abc.ABC

Protocol class for all ingredients.

all ingredients should have following format:

```
id: example-ingredient
dataset: ddf--example--dataset
key: "geo,time"  # key columns of ingredient
value:  # only include concepts listed here
  - concept_1
  - concept_2
filter:  # select rows by column values
  geo:  # only keep datapoint where `geo` is in [swe, usa, chn]
    - swe
    - usa
    - chn
```

The other way to define the ingredient data is using the `data` keyword to include external csv file, or inline the data in the ingredient definition. Example:

```
id: example-ingredient
key: concept
data: external_concepts.csv
```

On-the-fly ingredient:

```
id: example-ingredient
key: concept
data:
    - concept: concept_1
      name: concept_name_1
      concept_type: string
      description: concept_description_1
    - concept: concept_2
      name: concept_name_2
      concept_type: measure
      description: concept_description_2
```

dataset_path
return the full path to ingredient's dataset if the ingredient is from local ddf dataset.

ddf

ddf_id

dtype = 'abc'

static filter_row(*data: dict*, *row_filter*)
return the rows selected by row_filter.

classmethod from_procedure_result(*id*, *key*, *data_computed: dict*)

get_data()

ingredient_type

serve(**args*, ***kwargs*)
serving data to disk

class ddf_utils.chef.model.ingredient.**ConceptIngredient**(*id: str, key: Union[list, str], value: Union[list, dict, str] = '*', dataset: str = None, data: dict = None, row_filter: dict = None, base_dir: str = './'*)

Bases: *ddf_utils.chef.model.ingredient.Ingredient*

dtype = 'concepts'

get_data() → Dict[str, <Mock name='mock.DataFrame' id='140512929027728'>]

static get_data_from_ddf_dataset(*dataset_path*, *value*, *row_filter*)

static get_data_from_external_csv(*file_path*, *key*, *row_filter*)

static get_data_from_inline_data(*data*, *key*, *row_filter*)

serve(*outpath*, ***options*)
serving data to disk

class ddf_utils.chef.model.ingredient.**EntityIngredient**(*id: str, key: Union[list, str], value: Union[list, dict, str] = '*', dataset: str = None, data: dict = None, row_filter: dict = None, base_dir: str = './'*)

Bases: *ddf_utils.chef.model.ingredient.Ingredient*

dtype = 'entities'

get_data() → Dict[str, <Mock name='mock.DataFrame' id='140512929027728'>]

static get_data_from_ddf_dataset(*dataset_path*, *key*, *value*, *row_filter*)

static get_data_from_external_csv(*file_path*, *key*, *row_filter*)

static get_data_from_inline_data(*data*, *key*, *row_filter*)

serve(*outpath*, ***options*)
serving data to disk

class ddf_utils.chef.model.ingredient.**DataPointIngredient**(*id: str, key: Union[list, str], value: Union[list, dict, str] = '*', dataset: str = None, data: dict = None, row_filter: dict = None, base_dir: str = './'*)

Bases: *ddf_utils.chef.model.ingredient.Ingredient*

compute() → Dict[str, <Mock name='mock.DataFrame' id='140512929027728'>]
return a pandas dataframe version of self.data

dtype = 'datapoints'

classmethod from_procedure_result(*id*, *key*, *data_computed: dict*)

get_data() → Dict[str, <Mock name='mock.dataframe.DataFrame' id='140512915978832'>]

static get_data_from_ddf_dataset(*id*, *dataset_path*, *key*, *value*, *row_filter*)

static get_data_from_external_csv(*file_path*, *key*, *row_filter*)

static get_data_from_inline_data(*data*, *key*, *row_filter*)

serve(*outpath*, ***options*)
serving data to disk

ddf_utils.chef.model.ingredient.**ingredient_from_dict**(*dictionary: dict*, ***chef_options*) → ddf_utils.chef.model.ingredient.Ingredient
create ingredient from recipe definition and options. Parameters for ingredient should be passed in a dictionary. See the doc for *3. Define Ingredients* or `ddf_utils.chef.model.ingredient.Ingredient` for available parameters.

ddf_utils.chef.model.ingredient.**key_to_list**(*key*)
make a list that contains primaryKey of this ingredient

ddf_utils.chef.model.ingredient.**get_ingredient_class**(*cls*)

ddf_utils.chef.procedure package

Available Procedures

extract_concepts procedure for recipes

ddf_utils.chef.procedure.extract_concepts.**extract_concepts**(*chef: ddf_utils.chef.model.chef.Chef*, *ingredients: List[ddf_utils.chef.model.ingredient.Ingredient*, *result*, *join=None*, *overwrite=None*, *include_keys=False*) → ddf_utils.chef.model.ingredient.ConceptIngred
extract concepts from other ingredients.

Procedure format:

```
procedure: extract_concepts
ingredients:  # list of ingredient id
  - ingredient_id_1
  - ingredient_id_2
result: str  # new ingredient id
options:
  join:  # optional
    base: str  # base concept ingredient id
    type: {'full_outer', 'ingredients_outer'}  # default is full_outer
  overwrite:  # overwrite some concept types
    country: entity_set
    year: time
  include_keys: true  # if we should include the primaryKeys concepts
```

Parameters **ingredients** – any numbers of ingredient that needs to extract concepts from

Keyword Arguments

- **join** (*dict, optional*) – the base ingredient to join
- **overwrite** (*dict, optional*) – overwrite concept types for some concepts

- **include_keys** (*bool, optional*) – if we shuld include the primaryKeys of the ingredients, default to false

See also:

ddf_utils.transformer.extract_concepts() : related function in transformer module

---

Note:

- all concepts in ingredients in the `ingredients` parameter will be extracted to a new concept ingredient
- `join` option is optional; if present then the `base` will merge with concepts from `ingredients`
- `full_outer` join means get the union of concepts; `ingredients_outer` means only keep concepts from `ingredients`

---

filter procedure for recipes

ddf_utils.chef.procedure.filter.**filter**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.Ingredient], result, **options*) → ddf_utils.chef.model.ingredient.Ingredient

filter items and rows just as what *value* and *filter* do in ingredient definition.

Procedure format:

```
- procedure: filter
  ingredients:
      - ingredient_id
  options:
      item:  # just as `value` in ingredient def
          $in:
              - concept_1
              - concept_2
      row:  # just as `filter` in ingredient def
          $and:
              geo:
                  $ne: usa
              year:
                  $gt: 2010

  result: output_ingredient
```

for more information, see the `ddf_utils.chef.ingredient.Ingredient` class.

Parameters

- **chef** (Chef) – the Chef instance
- **ingredients** – list of ingredient id in the DAG
- **result** (*str*) –

Keyword Arguments

- **item** (*list or dict, optional*) – The item filter
- **row** (*dict, optional*) – The row filter

flatten procedure for recipes

ddf_utils.chef.procedure.flatten.**flatten**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], result, **options*) → ddf_utils.chef.model.ingredient.DataPointIngredient

flattening some dimensions, create new indicators.

procedure format:

```
procedure: flatten
ingredients:
    - ingredient_to_run
options:
    flatten_dimensions:
        - entity_1
        - entity_2
    dictionary:
        "concept_name_wildcard": "new_concept_name_template"
    skip_totals_among_entities:
        - entity_1
        - entity_2
```

The `dictionary` can have multiple entries, for each entry the concepts that matches the key in wildcard matching will be flatten to the value, which should be a template string. The variables for the templates will be provided with a dictionary contains `concept`, and all columns from `flatten_dimensions` as keys.

Parameters

- **chef** (Chef) – the Chef instance
- **ingredients** (*list*) – a list of ingredients
- **result** (*str*) – id of result ingredient
- **skip_totals_among_entities** (*list*) – a list of total among entities, which we don't add to new indicator names

Keyword Arguments

- **flatten_dimensions** (*list*) – a list of dimension to be flattened
- **dictionary** (*dict*) – the dictionary for old name -> new name mapping

groupby procedure for recipes

ddf_utils.chef.procedure.groupby.**groupby**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], result, **options*) → ddf_utils.chef.model.ingredient.DataPointIngredient

group ingredient data by column(s) and run aggregate function

Procedure format:

```
procedure: groupby
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingredient id
options:
  groupby: str or list  # column(s) to group
  aggregate: dict  # function block
  transform: dict  # function block
  filter: dict  # function block
```

The function block should have below format:

```
aggregate:
  column1: func_name1
  column2: func_name2
```

or

```
aggrgrate:
  column1:
    function: func_name
    param1: foo
    param2: baz
```

wildcard is supported in the column names. So `aggreagte: {"*": "sum"}` will run on every indicator in the ingredient

Keyword Arguments

- **groupby** (*str* or *list*) – the column(s) to group, can be a list or a string
- **insert_key** (*dict*) – manually insert keys in to result. This is useful when we want to add back the aggregated column and set them to one value. For example `geo: global` inserts the `geo` column with all values are "global"
- **aggregate** –
- **transform** –
- **filter** (*dict*, optinoal) – the function to run. only one of *aggregate*, *transform* and *filter* should be supplied.

Note:

- Only one of `aggregate`, `transform` or `filter` can be used in one procedure.
- Any columns not mentioned in groupby or functions are dropped.

---

merge procedure for recipes

ddf_utils.chef.procedure.merge.**merge**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.Ingredient], result, deep=False*) → ddf_utils.chef.model.ingredient.Ingredient

merge a list of ingredients

The ingredients will be merged one by one in the order of how they are provided to this function. Later ones will overwrite the previous merged results.

Procedure format:

```
procedure: merge
ingredients:  # list of ingredient id
  - ingredient_id_1
  - ingredient_id_2
  - ingredient_id_3
  # ...
result: str  # new ingredient id
options:
  deep: bool  # use deep merge if true
```

Parameters

- **chef** (Chef) – a Chef instance
- **ingredients** – Any numbers of ingredients to be merged

Keyword Arguments **deep** (*bool*, optional) – if True, then do deep merging. Default is False

Notes

deep merge is when we check every datapoint for existence if false, overwrite is on the file level. If key-value (e.g. geo,year-population_total) exists, whole file gets overwritten if true, overwrite is on the row level. If values (e.g. afr,2015-population_total) exists, it gets overwritten, if it doesn't it stays

merge_entity procedure for recipes

ddf_utils.chef.procedure.merge_entity.**merge_entity**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], dictionary, target_column, result, merged='drop'*) → ddf_utils.chef.model.ingredient.DataPointIngredient

merge entities

run_op procedure for recipes

ddf_utils.chef.procedure.run_op.**run_op**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], result, op*) → ddf_utils.chef.model.ingredient.DataPointIngredient

run math operation on each row of ingredient data.

Procedure format:

```
procedure: run_op
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingredient id
options:
  op: dict  # a dictionary describing calculation for each columns.
```

Keyword Arguments **op** (*dict*) – a dictionary of concept_name -> function mapping

Examples

for exmaple, if we want to add 2 columns, col_a and col_b, to create an new column, we can write

```
procedure: run_op
ingredients:
  - ingredient_to_run
result: new_ingredient_id
options:
  op:
    new_col_name: "col_a + col_b"
```

split_entity procedure for recipes

ddf_utils.chef.procedure.split_entity.**split_entity**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], dictionary, target_column, result, splitted='drop'*) → ddf_utils.chef.model.ingredient.DataPointIngredient

split entities

translate_column procedures for recipes

ddf_utils.chef.procedure.translate_column.**translate_column**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.Ingredient, result, dictionary, column, *, target_column=None, not_found='drop', ambiguity='prompt', ignore_case=False, value_modifier=None*) → ddf_utils.chef.model.ingredient.Ingredient

Translate column values.

Procedure format:

```
procedure: translate_column
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingredient id
options:
  column: str  # the column to be translated
  target_column: str  # optional, the target column to store the translated data
  not_found: {'drop', 'include', 'error'}  # optional, the behavior when there is␣
↪values not
                                           # found in the mapping dictionary,␣
↪default is 'drop'
  ambiguity: {'prompt', 'skip', 'error'}  # optional, the behavior when there is␣
↪ambiguity
                                          # in the dictionary
  dictionary: str or dict  # file name or mappings dictionary
```

If base is provided in dictionary, key and value should also in dictionary. In this case chef will generate a mapping dictionary using the base ingredient. The dictionary format will be:

```
dictionary:
  base: str  # ingredient name
  key: str or list  # the columns to be the keys of the dictionary, can accept a␣
↪list
  value: str  # the column to be the values of the the dictionary, must be one␣
↪column
```

Parameters

- **chef** (Chef) – The Chef the procedure will run on

- **ingredients** (*list*) – A list of ingredient id in the dag to translate

Keyword Arguments

- **dictionary** (*dict*) – A dictionary of oldname -> newname mappings. If 'base' is provided in the dictionary, 'key' and 'value' should also in the dictionary. See *ddf_utils.transformer.translate_column()* for more on how this is handled.
- **column** (*str*) – the column to be translated
- **target_column** (*str*, optional) – the target column to store the translated data. If this is not set then the *column* column will be replaced
- **not_found** (*{'drop', 'include', 'error'}, optional*) – the behavior when there is values not found in the mapping dictionary, default is 'drop'
- **ambiguity** (*{'prompt', 'skip', 'error'}, optional*) – the behavior when there is ambiguity in the dictionary, default is 'prompt'
- **value_modifier** (*str*, optional) – a function to modify new column values, default is None

See also:

ddf_utils.transformer.translate_column() : related function in transformer module

translate_header procedures for recipes

ddf_utils.chef.procedure.translate_header.**translate_header**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.Ingredient result, dictionary, duplicated='error'*) → ddf_utils.chef.model.ingredient.Ingredient

Translate column headers

Procedure format:

```
procedure: translate_header
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingredient id
options:
  dictionary: str or dict  # file name or mappings dictionary
```

Parameters

- **chef** (Chef) – The Chef the procedure will run on
- **ingredients** (*list*) – A list of ingredient id in the dag to translate
- **dictionary** (dict or *str*) – A dictionary for name mapping, or filepath to the dictionary
- **duplicated** (*str*) – What to do when there are duplicated columns after renaming. Available options are *error*, *replace*
- **result** (*str*) – The result ingredient id

See also:

ddf_utils.transformer.translate_header() : Related function in transformer module

all procedures for recipes

ddf_utils.chef.procedure.trend_bridge.**trend_bridge**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], bridge_start, bridge_end, bridge_length, bridge_on, result, target_column=None*) → ddf_utils.chef.model.ingredient.DataPointIngredient

run trend bridge on ingredients

Procedure format:

```
procedure: trend_bridge
ingredients:
  - data_ingredient                 # optional, if not set defaults to empty␣
↪ingredient
result: data_bridged
options:
  bridge_start:
      ingredient: old_data_ingredient # optional, if not set then assume it's the␣
↪input ingredient
      column:
        - concept_old_data
  bridge_end:
      ingredient: new_data_ingredient # optional, if not set then assume it's the␣
↪input ingredient
      column:
        - concept_new_data
  bridge_length: 5                  # steps in time. If year, years, if days,␣
↪days.
  bridge_on: time                   # the index column to build the bridge with
  target_column:
        - concept_in_result  # overwrites if exists. creates if not exists.␣
↪default to bridge_end.column
```

Parameters

- **chef** (Chef) – A Chef instance
- **ingredients** (*list*) – The input ingredient. The bridged result will be merged in to this ingredient. If this is None, then the only the bridged result will be returned
- **bridge_start** (*dict*) – Describe the start of bridge
- **bridge_end** (*dict*) – Describe the end of bridge
- **bridge_length** (*int*) – The size of bridge
- **bridge_on** (*str*) – The column to bridge
- **result** (*str*) – The output ingredient id

Keyword Arguments **target_column** (*list*, optional) – The column name of the bridge result. default to *bridge_end.column*

See also:

ddf_utils.transformer.trend_bridge() : related function in transformer module

window procedure for recipes

ddf_utils.chef.procedure.window.**window**(*chef: ddf_utils.chef.model.chef.Chef, ingredients: List[ddf_utils.chef.model.ingredient.DataPointIngredient], result, **options*) → ddf_utils.chef.model.ingredient.DataPointIngredient

apply functions on a rolling window

Procedure format:

```
procedure: window
ingredients:  # list of ingredient id
  - ingredient_id
result: str  # new ingredient id
options:
  window:
    column: str  # column which window is created from
    size: int or 'expanding'  # if int then rolling window, if expanding then␣
↪expanding window
    min_periods: int  # as in pandas
    center: bool  # as in pandas
    aggregate: dict
```

Two styles of function block are supported, and they can mix in one procedure:

```
aggregate:
  col1: sum  # run rolling sum to col1
  col2: mean  # run rolling mean to col2
  col3:  # run foo to col3 with param1=baz
    function: foo
    param1: baz
```

Keyword Arguments

- **window** (*dict*) – window definition, see above for the dictionary format
- **aggregate** (*dict*) – aggregation functions

Examples

An example of rolling windows:

```
procedure: window
ingredients:
    - ingredient_to_roll
result: new_ingredient_id
options:
  window:
    column: year
    size: 10
    min_periods: 1
    center: false
  aggregate:
    column_to_aggregate: sum
```

Notes

Any column not mentioned in the *aggregate* block will be dropped in the returned ingredient.

Submodules

ddf_utils.chef.api module

APIs for chef

ddf_utils.chef.api.**run_recipe**(*fn*, *ddf_dir*, *out_dir*)
run the recipe file and serve result

ddf_utils.chef.exceptions module

exceptions for chef

exception ddf_utils.chef.exceptions.**ChefRuntimeError**
Bases: Exception

exception ddf_utils.chef.exceptions.**IngredientError**
Bases: Exception

exception ddf_utils.chef.exceptions.**ProcedureError**
Bases: Exception

ddf_utils.chef.helpers module

ddf_utils.chef.helpers.**build_dictionary**(*chef*, *dict_def*, *ignore_case=False*, *value_modifier=None*)
build a dictionary from a dictionary definition

ddf_utils.chef.helpers.**build_dictionary_from_dataframe**(*df*, *keys*, *value*, *ignore_case=False*)

ddf_utils.chef.helpers.**build_dictionary_from_file**(*file_path*)

ddf_utils.chef.helpers.**create_dsk**(*data*, *parts=10*)
given a dictionary of {string: pandas dataframe}, create a new dictionary with dask dataframe

ddf_utils.chef.helpers.**debuggable**(*func*)
return a function that accepts *debug* as keyword parameters.

ddf_utils.chef.helpers.**dsk_to_pandas**(*data*)
The reverse for create_dsk function

ddf_utils.chef.helpers.**gen_query**(*conds*, *scope=None*, *available_scopes=None*)
generate dataframe query from mongo-like queries

ddf_utils.chef.helpers.**gen_sym**(*key*, *others=None*, *options=None*)
generate symbol for chef ingredient/procedure result

ddf_utils.chef.helpers.**get_procedure**(*procedure*, *base_dir*)
return a procedure function from the procedure name

Parameters

- **procedure** (*str*) – the procedure to get, supported formats are 1. procedure: sub/dir/module.function 2. procedure: module.function
- **base_dir** (*str*) – the path for searching procedures

ddf_utils.chef.helpers.**make_abs_path**(*path*, *base_dir*)
return a absolute path from a relative path and base dir.

If path is absoulte path arleady, it will ignore base dir and return path as is.

ddf_utils.chef.helpers.**mkfunc**(*options*)
create function warppers base on the options provided

This function is used in procedures which have a function block. Such as *ddf_utils.chef.procedure.groupby()*. It will try to return functions from numpy or ddf_utils.ops.

Parameters **options** (*str* or dict) – if a dictionary provided, "function" should be a key in the dictionary

ddf_utils.chef.helpers.**prompt_select**(*selects*, *text_before=None*)
ask user to choose in a list of options

ddf_utils.chef.helpers.**query**(*df*, *conditions*, *available_scopes=None*)
query a dataframe with mongo-like queries

ddf_utils.chef.helpers.**read_opt**(*options*, *key*, *required=False*, *default=None*, *method='get'*)
utility to read an attribute from an options dictionary

Parameters

- **options** (*dict*) – the option dictionary to read
- **key** (*str*) – the key to read

Keyword Arguments

- **required** (*bool*) – if true, raise error if the *key* is not in the option dict
- **default** (*object*) – a default to return if *key* is not in option dict and *required* is false

ddf_utils.chef.helpers.**sort_df**(*df*, *key*, *sort_key_columns=True*, *custom_column_order=None*)
Sorting df columns and rows.

Parameters

- **df** (*pd.DataFrame*) – DataFrame to sort
- **key** (*str* or list) – columns of dataframe, to be used as sorting key(s)

Keyword Arguments

- **sort_key_columns** (*bool*) – whehter to sort index column orders. If false index columns will retain the order of *key* parameter.
- **custom_column_order** (*dict*) – column weights for columns except keys. Columns not mentioned will have 0 weight. Bigger weight means higher rank.

ddf_utils.chef.ops module

commonly used calculation methods

ddf_utils.chef.ops.**aagr**(*df: <Mock name='mock.DataFrame' id='140512929027728'>*, *window: int = 10*)
average annual growth rate

Parameters **window** (*int*) – the rolling window size

Returns **return** – The rolling apply result

Return type *DataFrame*

ddf_utils.chef.ops.**between**(*x*, *lower*, *upper*, *how='all'*, *include_upper=False*, *include_lower=False*)

ddf_utils.chef.ops.**gt**(*x*, *val*, *how='all'*, *include_eq=False*)

ddf_utils.chef.ops.**lt**(*x*, *val*, *how='all'*, *include_eq=False*)

ddf_utils.chef.ops.**zcore**(*x*)

5.1.1.2 ddf_utils.model package

Submodules

ddf_utils.model.ddf module

The DDF model

class ddf_utils.model.ddf.**Concept**(*id: str*, *concept_type: str*, *props: dict = NOTHING*)
Bases: object

to_dict()

class ddf_utils.model.ddf.**DDF**(*concepts: Dict[str, ddf_utils.model.ddf.Concept] = NOTHING*, *entities: Dict[str, ddf_utils.model.ddf.EntityDomain] = NOTHING*, *datapoints: Dict[str, Dict[str, ddf_utils.model.ddf.DataPoint]] = NOTHING*, *synonyms: Dict[str, ddf_utils.model.ddf.Synonym] = NOTHING*, *props: dict = NOTHING*)
Bases: object

get_datapoints(*i*, *by=None*)

get_entities(*domain*, *eset=None*)

get_synonyms(*concept*)
get synonym dictionary. return None if no synonyms for the concept.

indicators(*by=None*)

class ddf_utils.model.ddf.**DaskDataPoint**(*id: str, dimensions: Tuple[str], path: Union[List[str], str], concept_types: dict, read_csv_options: dict = NOTHING, store='dask'*)
Bases: *ddf_utils.model.ddf.DataPoint*

load datapoints with dask

data

class ddf_utils.model.ddf.**DataPoint**(*id: str, dimensions: Tuple[str], store: str*)
Bases: abc.ABC

A DataPoint object stores a set of datapoints which have same dimensions and which belongs to only one indicator.

data

class ddf_utils.model.ddf.**Entity**(*id: str, domain: str, sets: List[str], props: dict = NOTHING*)
Bases: object

to_dict(*pkey=None*)
create a dictionary containing name/domain/is–headers/and properties So this can be easily plug in pandas.DataFrame.from_records()

class ddf_utils.model.ddf.**EntityDomain**(*id: str, entities: List[ddf_utils.model.ddf.Entity] = NOTHING, props: dict = NOTHING*)
Bases: object

add_entity(*ent: ddf_utils.model.ddf.Entity*)

entity_ids

entity_sets

classmethod from_entity_list(*domain_id, entities, allow_duplicated=True, **kwargs*)

get_entity_set(*s*)

has_entity(*sid*)

to_dict(*eset=False*)

class ddf_utils.model.ddf.**PandasDataPoint**(*id: str, dimensions: Tuple[str], path: str, dtypes: dict, store='pandas'*)
Bases: *ddf_utils.model.ddf.DataPoint*

load datapoints with pandas

data

class ddf_utils.model.ddf.**Synonym**(*concept_id: str, synonyms: Dict[str, str]*)
Bases: object

to_dataframe()

to_dict()

ddf_utils.model.package module

datapackage model

class ddf_utils.model.package.**DDFSchema**(*primaryKey: List[str], value: str, resources: List[str]*)
Bases: object

classmethod from_dict(*d: dict*)

class ddf_utils.model.package.**DDFcsv**(*base_path: str, resources: List[ddf_utils.model.package.Resource], props: dict = NOTHING, ddfSchema: Dict[str, List[ddf_utils.model.package.DDFSchema]] = NOTHING*)
Bases: *ddf_utils.model.package.DataPackage*

DDFCSV datapackage.

static entity_domain_to_categorical(*domain: ddf_utils.model.ddf.EntityDomain*)

static entity_set_to_categorical(*domain: ddf_utils.model.ddf.EntityDomain, s: str*)

classmethod from_dict(*d_: dict, base_path='./'*)

generate_ddf_schema(*progress_bar=False*)
generate ddf schema from all resources.

Parameters **progress_bar** (*bool*) – whether progress bar should be shown when generating ddfSchema.

get_ddf_schema(*update=False*)

load_ddf()
-> DDF

to_dict()
dump the datapackage to disk

class ddf_utils.model.package.**DataPackage**(*base_path: str, resources: List[ddf_utils.model.package.Resource], props: dict = NOTHING*)

Bases: object

classmethod from_dict(*d_: dict*, *base_path='./'*)

classmethod from_json(*json_path*)

classmethod from_path(*path*)

to_dict()
dump the datapackage to disk

class ddf_utils.model.package.**Resource**(*name: str*, *path: str*, *schema: ddf_utils.model.package.TableSchema*)

Bases: object

classmethod from_dict(*d: dict*)

to_dict()

class ddf_utils.model.package.**TableSchema**(*fields: List[dict], primaryKey: Union[List[str], str]*)

Bases: object

Table Schema Object Class

common_fields

field_names

classmethod from_dict(*d: dict*)

ddf_utils.model.repo module

model for dataset repositories.

class ddf_utils.model.repo.**Repo**(*uri*, *base_path=None*)

Bases: object

local_path

name

show_versions()

to_datapackage(*ref=None*)
turn repo@ref into Datapackage

ddf_utils.model.repo.**is_url**(*r*)

ddf_utils.model.utils module

ddf_utils.model.utils.**absolute_path**(*path: str*) → str

ddf_utils.model.utils.**sort_json**(*dp*)
sort json object. dp means datapackage.json

5.1.1.3 ddf_utils.factory package

Submodules

ddf_utils.factory.common module

class ddf_utils.factory.common.**DataFactory**
Bases: abc.ABC

bulk_download(**args*, ***kwargs*)

has_newer_source(**args*, ***kwargs*)

load_metadata(**args*, ***kwargs*)

ddf_utils.factory.common.**download**(*url*, *out_file*, *session=None*, *resume=True*, *method='GET'*, *post_data=None*, *retry_times=5*, *backoff=0.5*, *progress_bar=True*, *timeout=30*)
Download a url, and optionally try to resume it.

Parameters

- **url** (*str*) – URL to be downloaded
- **out_file** (*filepath*) – output file path
- **session** (*requests session object*) – Please note that if you want to use *requests_retry_session*, you must not use resume=True
- **resume** (*bool*) – whether to resume the download
- **method** (*str*) – could be "GET" or "POST". When posting you can pass a dictionary to *post_data*
- **post_data** (*dict*) –
- **times** (*int*) –
- **backoff** (*float*) –
- **progress_bar** (*bool*) – whether to display a progress bar
- **timeout** (*int*) – maximum time to wait for connect/read server responses. (Note: not the time limit for total response)

ddf_utils.factory.common.**requests_retry_session**(*retries=5*, *backoff_factor=0.3*, *status_forcelist=(500, 502, 504)*, *session=None*)

ddf_utils.factory.common.**retry**(*times=5*, *backoff=0.5*, *exceptions=<class 'Exception'>*)
general wrapper to retry things

ddf_utils.factory.clio_infra module

functions for scraping data from clio infra website.

Source link: Clio-infra website

class ddf_utils.factory.clio_infra.**ClioInfraLoader**
Bases: *ddf_utils.factory.common.DataFactory*

bulk_download(*out_dir*, *data_type=None*)

has_newer_source(*ver*)

load_metadata()

url = 'https://clio-infra.eu/index.html'

ddf_utils.factory.ihme module

Functions for IHME

The GBD result tool at IHME contains all data for GBD results, but they don't have an open API to query the data. However the website uses a json endpoint and it doesn't need authorization. So we also make use of it.

class ddf_utils.factory.ihme.**IHMELoader**
Bases: *ddf_utils.factory.common.DataFactory*

bulk_download(*out_dir*, *version*, *context*, ***kwargs*)
download the selected contexts/queries from GBD result tools.

`context` could be a string or a list of strings. The complete query will be generated with `_make_query` method and all keywork args. When context is a list, multiple queries will be run.

download_links(*url*)

has_newer_source(*ver*)

load_metadata()
load all codes used in GBD in a dictionary.

url_data = 'http://ghdx.healthdata.org/sites/all/modules/custom/ihme_query_tool/gbd-se

url_hir = 'http://ghdx.healthdata.org/sites/all/modules/custom/ihme_query_tool/gbd-sea

url_metadata = 'http://ghdx.healthdata.org/sites/all/modules/custom/ihme_query_tool/gb

url_task = 'https://s3.healthdata.org/gbd-api-2019-public/{hash}'

url_version = 'http://ghdx.healthdata.org/sites/all/modules/custom/ihme_query_tool/gbd

ddf_utils.factory.ilo module

Functions for scraping ILO datasets

using the bulk downloader, see its doc.

class ddf_utils.factory.ilo.**ILOLoader**
Bases: *ddf_utils.factory.common.DataFactory*

bulk_download(*out_dir*, *indicators: list*, *pool_size=5*)
Download a list of indicators simultaneously.

download(*i*, *out_dir*)
Download an indicator to out_dir.

has_newer_source(*indicator*, *date*)
check if an indicator's last modified date is newer than given date.

indicator_meta_url_tmpl = 'http://www.ilo.org/ilostat-files/WEB_bulk_download/indicato

load_metadata(*table='indicator'*, *lang='en'*)
get code list for a specified table and language.

Check ILO doc for all available tables and languages.

main_url = 'http://www.ilo.org/ilostat-files/WEB_bulk_download/'

other_meta_url_tmpl = 'http://www.ilo.org/ilostat-files/WEB_bulk_download/dic/{table}_

ddf_utils.factory.oecd module

Functions for scraping OECD website using their SDMX API

source link OECD website

class ddf_utils.factory.oecd.**OECDLoader**
Bases: *ddf_utils.factory.common.DataFactory*

bulk_download(*out_dir*, *dataset*)
download the full json, including observation/dimension lists.

data_url_tmpl = 'http://stats.oecd.org/SDMX-JSON/data/{dataset}/all/all'

datastructure_url_tmpl = 'http://stats.oecd.org/restsdmx/sdmx.ashx/GetDataStructure/{d

has_newer_source(*dataset*, *version*)

load_metadata()

metadata_url = 'http://stats.oecd.org/RestSDMX/sdmx.ashx/GetKeyFamily/all'

ddf_utils.factory.worldbank module

Functions to load data from Worldbank API.

We use its bulkdownload utilities.

Source link: WorldBank website

class ddf_utils.factory.worldbank.**WorldBankLoader**
Bases: *ddf_utils.factory.common.DataFactory*

T.B.D

bulk_download(*dataset*, *out_dir*, ***kwargs*)

has_newer_source(*dataset*, *date*)

load_metadata()

url = 'http://api.worldbank.org/v2/datacatalog?format=json'

5.1.2 Submodules

5.1.3 ddf_utils.cli module

script for ddf dataset management tasks

5.1.4 ddf_utils.i18n module

i18n project management for Gapminder's datasets.

The workflow is described in this google doc. The json part comes from discussion here

`ddf_utils.i18n.`**`merge_translations_csv`**(*path*, *split_path='langsplit'*, *lang_path='lang'*, *overwrite=False*)

merge all translated csv files and update datapackage.json

`ddf_utils.i18n.`**`merge_translations_json`**(*path*, *split_path='langsplit'*, *lang_path='lang'*, *overwrite=False*)

merge all translated json files and update datapackage.json.

`ddf_utils.i18n.`**`split_translations_csv`**(*path*, *split_path='langsplit'*, *exclude_concepts=None*, *overwrite=False*)

split all string concepts and save them as csv files

`ddf_utils.i18n.`**`split_translations_json`**(*path*, *split_path='langsplit'*, *exclude_concepts=None*, *overwrite=False*)

split all string concepts and save them as json files

Note: There is an issue with dataframe.to_json() method for multiIndex files (i.e. datapoints), which cause we can't read back the split files and merge them. In this case `merge_translations_json()` will fail.

5.1.5 ddf_utils.package module

functions for handling DDF datapackage

`ddf_utils.package.`**`create_datapackage`**(*path*, *gen_schema=True*, *progress_bar=False*, ***kwargs*)

create datapackage.json base on the files in *path*.

If you want to set some attributes manually, you can pass them as keyword arguments to this function

---

Note: A DDFcsv datapackage MUST contain the fields *name* and *resources*.

if name is not provided, then the base name of *path* will be used.

Parameters

- **`path`** (*str*) – the dataset path to create datapackage.json
- **`gen_schema`** (*bool*) – whether to create DDFSchema in datapackage.json. Default is True
- **`progress_bar`** (*bool*) – whether progress bar should be shown when generating ddfSchema.

- **kwargs** (*dict*) – metadata to write into datapackage.json. According to spec, title, description, author and license SHOULD be fields in datapackage.json.

ddf_utils.package.**get_datapackage**(*path*, *use_existing=True*, *update=False*, *progress_bar=False*)
get the datapackage.json from a dataset path, create one if it's not exists

Parameters **path** (*str*) – the dataset path

Keyword Arguments

- **use_existing** (*bool*) – whether or not to use the existing datapackage
- **update** (*bool*) – if update is true, will update the resources and schema in existing datapackage.json. else just return existing datapackage.json
- **progress_bar** (*bool*) – whether progress bar should be shown when generating ddfSchema.

ddf_utils.package.**get_ddf_files**(*path*, *root=None*)
yield all csv files which are named following the DDF model standard.

Parameters

- **path** (*str*) – the path to check
- **root** (*str*, optional) – if path is relative, append the root to all files.

ddf_utils.package.**is_datapackage**(*path*)
check if a directory is a dataset directory

This function checks if ddf–index.csv and datapackage.json exists to judge if the dir is a dataset.

5.1.6 ddf_utils.io module

io functions for ddf files

ddf_utils.io.**cleanup**(*path*, *how='ddf'*, *exclude=None*, *use_default_exclude=True*)
remove all ddf files in the given path

ddf_utils.io.**csvs_to_ddf**(*files*, *out_path*)
convert raw files to ddfcsv

Parameters

- **files** (*list*) – a list of file paths to build ddf csv
- **out_path** (*str*) – the directory to put the ddf dataset

ddf_utils.io.**download_csv**(*urls*, *out_path*)
download csv files

ddf_utils.io.**dump_json**(*path*, *obj*)
convenient function to dump a dictionary object to json

ddf_utils.io.**open_google_spreadsheet**(*docid*)
read google spreadsheet into excel io object

ddf_utils.io.**serve_concept**()

ddf_utils.io.**serve_datapoint**(*df_: <Mock name='mock.DataFrame' id='140512929027728'>*, *out_dir*, *concept*, *copy=True*, *by: Iterable[T_co] = None*, *formatter: Callable = <function format_float_digits>*, ***kwargs*)
save a pandas dataframe to datapoint file. the file path of csv will be out_dir/ddf–datapoints–$concept–$by.csv

addition keyword arguments can be passed to *pd.DataFrame.to_csv()* function.

ddf_utils.io.**serve_entity**()

5.1.7 ddf_utils.patch module

functions working with patches

ddf_utils.patch.**apply_patch**(*base*, *patch*)
apply patch created with daff. more on the diff format, see: http://specs.frictionlessdata.io/tabular-diff-format/

return: the updated DataFrame.

5.1.8 ddf_utils.qa module

QA functioins.

ddf_utils.qa.**avg_pct_chg**(*comp_df*, *indicator*, *on='geo'*)
return average precentage changes between old and new data

ddf_utils.qa.**compare_with_func**(*dataset1*, *dataset2*, *fns=None*, *indicators=None*, *key=None*, ***kwargs*)
compare 2 datasets with functions

ddf_utils.qa.**dropped_datapoints**(*comp_df*, *indicator*, ***kwargs*)

ddf_utils.qa.**max_change_index**(*comp_df*, *indicator*, ***kwargs*)

ddf_utils.qa.**max_pct_chg**(*comp_df*, *indicator*, ***kwargs*)
return average precentage changes between old and new data

ddf_utils.qa.**new_datapoints**(*comp_df*, *indicator*, ***kwargs*)

ddf_utils.qa.**nrmse**(*comp_df*, *indicator*, ***kwargs*)

ddf_utils.qa.**rmse**(*comp_df*, *indicator*, ***kwargs*)

ddf_utils.qa.**rval**(*comp_df*, *indicator*, *on='geo'*)
return r-value between old and new data

5.1.9 ddf_utils.str module

string functions for ddf files

ddf_utils.str.**fix_time_range**(*s*)
change a time range to the middle of year in the range. e.g. fix_time_range('1980-90') = 1985

ddf_utils.str.**format_float_digits**(*number*, *digits=5*, *threshold=None*, *keep_decimal=False*)
format the number to string, limit the maximum amount of digits. Removing tailing zeros.

ddf_utils.str.**format_float_sigfig**(*number*, *sigfig=5*, *threshold=None*)
format the number to string, keeping some significant digits.

ddf_utils.str.**parse_time_series**(*ser*, *engine='pandas'*)
try to parse date time from a Series of string

see document https://docs.google.com/document/d/1Cd2kEH5w3SRJYaDcu-M4dU5SY8No84T3g-QlNSW6pIE/edit#heading=h.oafc7aswaafy for more details of formats

ddf_utils.str.**to_concept_id**(*s*, *sep='_'*)
convert a string to alphanumeric format.

5.1.10 ddf_utils.transformer module

functions for common tasks on ddf datasets

ddf_utils.transformer.**extract_concepts**(*dfs*, *base=None*, *join='full_outer'*)
extract concepts from a list of dataframes.

Parameters **dfs** (*list[DataFrame]*) – a list of dataframes to be extracted

Keyword Arguments

- **base** (*DataFrame*) – the base concept table to join
- **join** (*{'full_outer', 'ingredients_outer'}*) – how to join the *base* dataframe. full_outer means union of the base and extracted, ingredients_outer means only keep concepts in extracted

Returns the result concept table

Return type DataFrame

ddf_utils.transformer.**merge_keys**(*df*, *dictionary*, *target_column*, *merged='drop'*, *agg_method='sum'*)
merge keys

ddf_utils.transformer.**split_keys**(*df*, *target_column*, *dictionary*, *splited='drop'*)
split entities

ddf_utils.transformer.**translate_column**(*df*, *column*, *dictionary_type*, *dictionary*, *target_column=None*, *base_df=None*, *not_found='drop'*, *ambiguity='prompt'*, *ignore_case=False*)
change values in a column base on a mapping dictionary.

The dictionary can be provided as a python dictionary, pandas dataframe or read from file.

Note: When translating with a base DataFrame, if ambiguity is found in the data, for example, a dataset with entity-id *congo*, to align to a dataset with *cod* (Democratic Republic of the Congo) and *cog* (Republic of the Congo), the function will ask for user input to choose which one or to skip it.

Parameters

- **df** (*DataFrame*) – The dataframe to be translated
- **column** (*str*) – The column to be translated
- **dictionary_type** (*str*) – The type of dictionary, choose from *inline*, *file* and *dataframe*
- **dictionary** (*str* or *dict*) – The dictionary. Depanding on the *dictionary_type*, the value of this parameter should be: *inline*: *dict file*: the file path, *str dataframe*: *dict*, must have *key* and *value* keys. see examples in examples section.
- **target_column** (*str*, optional) – The column to store translated resluts. If this is None, then the one set with *column* will be replaced.
- **base_df** (*DataFrame*, optional) – When *dictionary_type* is *dataframe*, this option should be set
- **not_found** (*str*) – What to do if key in the dictionary is not found in the dataframe to be translated. avaliable options are *drop*, *error*, *include*

- **ambiguity** (*str*) – What to do when there is ambiguities in the dictionary. avaliable options are *prompt*, *skip*, *error*

Examples

```
>>> df = pd.DataFrame([['geo', 'Geographical places'], ['time', 'Year']],␣
↪columns=['concept', 'name'])
>>> df
  concept                name
0     geo  Geographical places
1    time                Year
>>> translate_column(df, 'concept', 'inline', {'geo': 'country', 'time': 'year'})
   concept                name
0  country  Geographical places
1     year                Year
```

```
>>> base_df = pd.DataFrame([['geo', 'country'], ['time', 'year']],␣
↪columns=['concept', 'alternative_name'])
>>> base_df
  concept alternative_name
0     geo          country
1    time             year
>>> translate_column(df, 'concept', 'dataframe',
...                  {'key': 'concept', 'value': 'alternative_name'},
...                  target_column='new_name', base_df=base_df)
  concept                name new_name
0     geo  Geographical places  country
1    time                Year     year
```

ddf_utils.transformer.**translate_header**(*df*, *dictionary*, *dictionary_type='inline'*)

change the headers of a dataframe base on a mapping dictionary.

Parameters

- **df** (*DataFrame*) – The dataframe to be translated
- **dictionary_type** (*str*, default to *inline*) – The type of dictionary, choose from *inline* or *file*
- **dictionary** (*dict* or *str*) – The mapping dictionary or path of mapping file

ddf_utils.transformer.**trend_bridge**(*old_ser: <Mock name='mock.Series' id='140512928420432'>*, *new_ser: <Mock name='mock.Series' id='140512928420432'>*, *bridge_length: int*) → <Mock name='mock.Series' id='140512928420432'>

smoothing data between series.

To avoid getting artificial stairs in the data, we smooth between to series. Sometime one source is systematically higher than another source, and if we jump from one to another in a single year, this looks like an actual change in the data.

Parameters

- **old_data** (*Series*) –
- **new_data** (*Series*) –
- **bridge_length** (*int*) – the length of bridge

Returns **bridge_data**

Return type the bridged data

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

E

F

G

H

I

K

L

M

N

U

V

W

Z