Converting Units in Databases Using Additional Files and Flexible Coding

Mihran Simonian - 12386294
mihran.simonian@gmail.com
University of Amsterdam

ABSTRACT

Unit conversion has been a source of conflicts and errors. The current systems fail to provide simplicity in usage, allowing miscommunication to occur with all consequential effects; even NASA managed to loose a Mars climate orbiter due to using the wrong unit. This paper analyzes the existing methods and proposes a new system that uses simplicity as it's key solution for error prevention whilst providing flexibility in applications.

KEYWORDS

Supply chain, units, convert, databases, csv

1 INTRODUCTION

The manufacturing industry is currently going through a major development, described as 'Smart Industry 4.0'[5]. By using computers, big data and the implementation of various sensors, manufacturers attempt to optimize their production flows and lower costs, whilst increasing reliability and optimize production speeds.
Production optimization is welcome but production halts are costly and create significant delivery delays to the end-users. OEM producers (manufacturers whom combine loose parts to assemble one product) attempt to optimize their supply chain, by combining the data streams of various suppliers. This will enable OEM produces to draw a quick overview on the availability of a certain item inside the whole supply chain network. This overview can be used during a disruption in the supply chain by manual labour (supply chain engineers) or by automatic rescheduling and ordering.
As measurement units are different across countries, unit conversion is required in order to be able to implement these innovations.

Unit Conversion Applications

Unit conversion can further improve supply chain management but converting units is also an important aspect when doing other activities such as data analysis of production parameters. The field of application for unit conversion is broad yet, despite many efforts, the various current methods all lack in one aspect or another, as elaborated on in the appendix to this paper.

Global Organization

Various research and innovation programs working on this challenge have been initiated but not yet finalized [4]. Unit conversion however, is not part of these projects, yet it remains a key issue when implementing the various supply chain systems.

NASA's Crashing Planetary Orbiter

Differences in measurement units used in manufacturing industry are visible on all type of fields and industries. To illustrate the magnitude of the issue; in 1999 NASA lost one of it's satellites orbiting the planet Mars, due to a programming error in which the wrong unit was applied [1], indicative to the potential consequences and highlighting the importance of unit conversion.

2 RELATED WORK

Unit Representation in Information Systems

Units have been ignored frequently within the design of information systems, despite their vital role within the physical world. As no unification system has been agreed upon, various reinventions have been initiated over time, further complicating the application of unit conversion in information systems [3]. The specific issue, is that there is no computer code standard in order to represent the unit 'Celsius' for instance, complicating unit conversion systems to be applied worldwide.

Sensors, Unit Conversion and IT

Using multiple sensors and vastly storing these in computer systems potentially creates sources of conflict [6]. The paper of Waltz originates from 1990, however in 2013 there were still reasons enough for concern, as [3] described in his paper.

Integrating ERP Systems

An enterprise resource planning (ERP[1]) system is used by manufacturers to manage production. The Dutch research institute TNO is currently developing (test version live since January 2020) a communication protocol, labelled SCSN.

[1] https://en.wikipedia.org/wiki/Enterprise_resource_planning

SCSN is developed in cooperation with various manufacturers[2][3] and it allows ERP systems to be combined. Being early in development, live connections are not yet implemented, instead backlog CSV files are used and imported on scheduled moments.

Python Libraries

Unit conversion is not a new thing, as [3] indicated already programmers have been forced to create various initiatives. For Python[4] multiple libraries exist, such as PintPy[5] or the 'unit-converter' from PyPi[6]. In section 4, section 5 and section 6 a deeper overview is given, an extensive research on these libraries is added to this paper in appendix A.

3 RESEARCH

Relation to Big Data: Variety

The angle of approach for the current paper is designed around the *variety* aspect of the Big Data set of V's. The question how to reduce variety of units, which can potentially lead to all sorts of practical problems (losing a satellite in space) is key.

Relation to Data Science Methods

Data scientists often use statistical methods in order to test certain hypothesis, such as calculating the mean or median. Especially in the field of data science, variations in numbers can push the research into the wrong direction. By removing a fundamental reason why numbers can differ (different units) we can further improve the field of data science and improve future researches.

Aim

By discovering the limitations or challenges of current systems, it becomes apparent how the proposed system can be supporting the current research on the field of unit conversion.

Research Question

- How do we combine various units and translate them into a single representative unit system?

The following sub questions are part of the research:

- What is the current level of the Python libraries?
- How is unit conversion done?
- How do we influence our code using external CSV files?
- Which pre-cleaning steps are required?

In section 7 a further investigation is made on applying the proposed system to large or multiple datasets.

4 METHOD

Data Sources

For this research two self-generated dataset are used to initially develop and later test the proposed method. The dataset is generated using Microsoft Excel[7] as it enables quick generation of random numbers and datasets, with easy conversion to a set of datatypes. Python libraries have also been tested using the same dataset (where possible).

Dataset and Personal Experience. The dataset will be generated based upon author's personal experience from working with ASML products. ASML products use a specific part number system labelled '12NC' as described in [2]. The part number system allows interlinking and easy comparison of the same item in a dataset. This is not uncommon in industries, such as the ISBN number used within the book industry.

Databases and CSV Filetype

The ERP systems are hypothesized to be seperate database files which will be combined, similar to what the SCSN network is attempting to achieve as described in section 2. In line with the SCSN network, the popular CSV format[8] is used.

Unit Types

For the current research focus is made on the application of the following units:

- Distance.
- Temperatures.

There is sufficient differences in these units in order to draft up a complex study, highlighting the differences of unit systems and complexities involved in converting them.

One Unit System

The proposed solution will align all units to be represented in one unit system; this is the international recognized SI system[9]. The system will allow the user to adjust the conversion system, so that another measurement system can be applied.

[2] https://smartindustry.nl/fieldlabs/8-smart-connected-supplier-network
[3] https://www.brainportindustries.com/nl/berichten/maakindustrie-aan-de-slag-met-digitalisering
[4] https://www.python.org/
[5] https://pint.readthedocs.io/en/0.11
[6] https://pypi.org/project/unit-converter
[7] https://products.office.com/en-us/excel
[8] https://en.wikipedia.org/wiki/Comma-separated_values
[9] https://en.wikipedia.org/wiki/International_System_of_Units

Python and Jupyter

The proposed solution is designed using the Python language[10], using the popular Jupyter Notebooks[11] format.

Libraries. Python utilizes libraries which expand capabilities of the code quickly. The proposed solution will use the popular library Pandas [12]. An important consideration to use Pandas is it's high efficiency and easy to interpret code, furthermore it allows import from CSV, but also other popular data sources. This functionality is much needed when upscaling to big data, as many varieties of data sources exist.

Testing Existing Libraries

The following Python libraries that offer unit conversion capabilities are tested to discover whether these can be used for a proposed solution:

- PintPy[13]
- PiPy: Unit-convert[14]
- PiPy: Unit-converter[15]

Design Own Method

The currently available Python libraries are limited, each in their own way (please refer to section 5 and appendix A for more specific information). This paper proposes a new method to tackle the issue of unit conversion.

New System Goal. This paper proposes a new system, which allows unit conversion to take place quickly, correctly and allows users to easily adjust unit conversions. The main philosophy within the proposed system is that no specific coding is required, as the unit conversion will be described in a separate file. The code itself will remain flexible in order to cope with the various unit systems. This should allow the system to be easily adjusted, allowing batches of databases to be converted automatically, for instance during the night.

Requirements of New System

The proposed solution needs to:

(1) Convert units.
(2) Import CSV files.
(3) Be easily adjustable without touching the code.
(4) Export to other formats.

Scope and Limitations

This research attempts to describe and propose an alternative method to unit conversions in a general matter.

Limitation: Multiple Unit Systems. As many unit systems still exist today, small differences are not considered relevant, such as the (small) differences between UK (Imperial) and US (Customary) units[16]. This paper will not consider these differences as these are 'just another unit system to be converted'.

Limitation: Calculation Optimization. This research and consequential design will not try to optimize and reduce the computational cycles required in order to transform units as the highly efficient Pandas library takes care of this. Discussing calculation optimization would transform the research more into reviewing this unit issue from the big data perspective of 'velocity' oppose to the intended 'variety'.

5 IMPLEMENTATION

Testing Existing Libraries: Findings

Each library will be evaluated on the following points:

- How does it work?
- Did any error occur?
- Solutions of error prevention and consequences.
- Conclusion.

Overview of the System: How Does It Work?

A generalized layout of the system workings should be as following:

(1) Import, convert and store a new supplier:
    (a) Fixed parameters are set.
    (b) Read unit conversion CSV file.
    (c) Import a new supplier:
        (i) Import CSV database to Pandas *DataFrame* [17].
        (ii) Remove unnecessary and clean remaining data.
        (iii) Read 'metadata' from database.
        (iv) Apply conversion steps.
    (d) Store cleaned database to storage type of choice.
(2) Add converted database of supplier to main database.
    (a) Fixed parameters are set.
    (b) Import main database and new supplier.
    (c) Merge two databases.
    (d) Store new main database to storage type of choice.

In the following paragraphs specific parts are discussed, which also reflect the research questions of this report.

Principle of Unit Conversion

Converting a unit can be done through two ways:

[10] https://www.python.org/
[11] https://jupyter.org/
[12] https://pandas.pydata.org/
[13] https://pint.readthedocs.io/en/0.11/
[14] https://pypi.org/project/unit-convert/
[15] https://pypi.org/project/unit-converter/
[16] https://en.wikipedia.org/wiki/Comparison_of_the_imperial_and_US_customary_measurement_systems
[17] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

(1) Convert using a static number (multiply or divide), such as inches to centimeters:

$$D_{centimeter} = D_{inch} * 2.54$$

(2) Convert using a formula, such as Fahrenheit to Celsius:

$$T_{Celsius} = (T_{Fahrenheit} - 32) * (5/9)$$

The two ways are not complex and unit conversion formulas are abundantly available [18]. The challenges within automated unit conversions are:

- Hardcoding unit conversion
- Applying a formula when needed
- Determining what unit is used

Hard-coding Unit Conversion: Using a Dictionary

By using a dictionary a 'dynamic' work space is created, in which unit conversions can be set whilst not interfering with the actual code. This separation allows non-programmers (such as mechanical engineers) to adjust units quickly. This system also allows the creation of batches for automatic import purposes.

Numbers and Formulas. Formulas need to be hardcoded into Python in order for Python to understand the formulas. When importing a CSV into the computer, each individual variable is appointed a memory location by Python. Formulas are generally stored as a *string* datatype in the memory, which is not a formula. By re-iterating over the dictionary values after import, an evaluation of the contents in the memory location can confirm whether it is a function or not. The system should hereafter update the values in the dictionary by creating a pointer that 'points' to the memory location in which the formula has been stored.

Algorithm 1: Evaluate Formula Update to Pointer

```
for keys in dictionary do
    value = dictionary[key] if value is string: then
        dictionary[key] = evaluate if formula(value)
    end
end
```

Applying a Formula When Needed

As described previously not all conversions are simple multiplications or divisions but require the application of formulas. Please refer below to the dictionary algorithm, which allows both formulas and 'simple' conversions.

Algorithm 2: Convert: Formula or Multiplication?

```
if x is formula then
    apply formula x with data
else
    do x * data
end
```

Determining What Unit Is Used

This imported data is not useful as long as it is unclear what it represents. This is recorded in the *metadata*, which describes for instance 'this column has temperatures in Celsius'. In order to determine the contents of the database, the following two files will become required for each supplier separately:

- CSV unit conversion list.
  Describes the unit conversion factor or formula.
- CSV database.
  The actual database.

The CSV unit conversion list describes the conversion to be followed and allows quick adjustments to the system. This allows users to adjust the conversion to their own specific wishes (be it SI or USCS for instance). The unit conversion file will use a name for a certain conversion, which the system loads as a so called dictionary key. The conversion itself is then loaded as the dictionary value.

This dictionary key needs to be matched in the actual database, allowing the system to know which conversion needs to be applied for which column. Multiple columns can use a single conversion as the name (dictionary key) of the conversion needs to be unique in the conversion list, but not in the database.

Pre-Cleaning Data

Numbers displayed on the screen are not always numbers for a computer, which can also interpret a number as a piece of text (a *string* datatype). Data type differences can create errors during operations. To prevent errors and allow numeric calculations, data cleaning and potentially data type conversion is required. A function will need to be created which will clean up the input data:

(1) Checks input type.
(2) Assure decimal separators.
(3) Strings get converted to floating points.
(4) Returns float type or error.

Below a simplified algorithm is given which executes all the above items.

Assure Decimal Separators. Dots and commas are common in numbers and can become confusing when comparing international number formatting. Some dots or commas are not related to the number itself, such as thousand separators.

[18] https://en.wikipedia.org/wiki/Conversion_of_units

Algorithm 3: Pre-Clean Data

```
if x is string then
    x = x.replace( , to . )
    if . in x > 1 then
        remove all . except 1
    end
    return x to float datatype
end
```

Decimal separators [19], are part of a number and therefore important to preserve (2,01 is something else than 201).

Error Handling

Errors can (and generally will) occur during external data import. Converting large datasets can take time, in case of an error it is important to know the error, but stopping the program will be time inefficient as the whole operation needs to be redone. By returning a textual error and storing this error in the cleaned database, analysis can be performed after import, for instance by verifying which datatype is stored in a column. If they are all numeric, no error, if a string than the user knows an error has occurred. The proposed system should allow for the following error handling solutions:

- Stop the program (default option).
  A runtime error is generated which will stop execution of the process.
- Return a error text in string (optional).
  Returns an error, specific to the function. Can be used to be stored in database.

The program can use *Boolean* values (True or False) which can be given to the separate functions, in order to determine which approach is applied.

Store Output

The proposed system imports CSV files, however it allows the cleaned up database to be exported to the following file formats; CSV [20], SQL [21], HDF [22], Parquet [23] and Feather [24].

These file formats can compress data much better than CSV files, with varying compression levels, a fundamental requirement when upscaling to larger datasets.

6 EVALUATION

Link to Code, Data and Presentation

The proposed system, used databases and tools are available at location: https://github.com/mihransimonian/Uva-Block3-Big_Data-Assignment. This location also contains this paper, together with a presentation.
In the appendix B core elements of the code have been included.

Testing Existing Libraries: Findings

A general conclusion of the research conducted on the existing Python libraries in appendix A is that:

- Too simple:
  Libraries are not able to convert all measurement units.
- Data needs to be inputted in specific ways:
  Libraries can request information to be offered to them very specifically, if not errors occur.
- Strange error messages:
  Similar to the point above, upon hitting an error the programmer needs to have specific knowledge about measurement systems.

Please refer to the appendix A for an elaborate analysis of the various libraries, how they work and what limitations there are. This includes actual Python code, in case the reader wants to implement these libraries themselves.

Hard-coding Unit Conversion: Using a Dictionary

The *eval()* function in Python has been applied, which can cover the required evaluation of strings. It can be used in order to update dictionary values, please refer to appendix B for the specific code implementation.

Applying a Formula When Needed

By checking whether a formula is, using the function *callable()*, an evaluation can be made on whether the unit conversion can be applied or must be multiplied. Please refer to appendix B for the specific code implementation.

Determining What Unit Is Used

As described in section 5 the system uses an additional file that contains metadata, describing the database content.

Proposed Naming System. By the definitions used in the dictionary, mistakes are reduced as the user has to write down each individual conversion as:

```
[subject] [unit_system] [unit]
```

Users can adjust this system to what they prefer, as long as the two files match.

[19] https://en.wikipedia.org/wiki/Decimal_separator
[20] https://en.wikipedia.org/wiki/Comma-separated_values
[21] https://en.wikipedia.org/wiki/SQL
[22] https://en.wikipedia.org/wiki/Hierarchical_Data_Format
[23] https://en.wikipedia.org/wiki/Apache_Parquet
[24] https://cran.r-project.org/web/packages/feather/feather.pdf

Pre-Cleaning Data

Pre cleaning is an important aspect of data integration, as is data type verification. The current system only verifies *booleans*, *strings*, *floats*. More datatypes was deemed not required, as these are mere iterations of the same proposed system. Please refer to appendix B for the actual code.

Error Handling

The program uses boolean values (True or False) which can be given to the separate functions, in order to determine which approach is applied. Please refer to appendix B for an example of the code implementation, however it is best to look at the complete code in order to understand the various locations where the error handling has been implemented. It is advised to use the return error text option as it allows the returned error messages to be stored in the database, for future tracking purposes.

Store Output

The proposed method uses pandas library, which supports exporting to: CSV, SQL, HDF, Parquet and Feather. By using a *string* indicator, a user can specify the specific file format to be outputted. A default option has been implemented as well (CSV). For file compression results the parquet file format has been used as an example; the tested database was 11 MB in CSV, whilst 3 MB in parquet with more compression possibilities remaining (gzip, compression was set to 5, range goes up to 9). Non-CSV file formats do not only compress data better than CSV, they can also be used to perform multi-node operations on a larger cluster of networks. Refer to appendix B for the code implementation.

7 DISCUSSION

Versatility Through Simplicity

The real power of the proposed solution is the simplicity. By allowing separate files to dictate the conversions of units, the systems solution allows users the freedom to adjust which conversions are really required. As the dictionary keys are required to align with the database headings of columns, this creates a double verification whether the correct unit is in place.

Coding Comments

Not really part of the assignment, but the code has been accompanied with extensive descriptive comments, which allow any user to understand the relations between the code.

Human Factor

By using one unit system, we still don't prevent people from entering the wrong unit conversion. By using CSV files (accessible through various office software packages) we do limit input locations, but it still allows the human factor to create potential hazards. The current solution allows non experienced users to for instance format the harddrive by accident, as the system sees formulas as functions. A safety net has been created (*string.startswith*), but this is still important to highlight.

Upscaling Data

Import Batching. In case multiple imports are required to perform over time, the proposed solution can accomodate this option for perform automatic importing. The code has been built in such a way that *parameters* can be given about file locations. This allows importing of multiple batches of datasets, this significantly improves the applicability in real life.

Self-containing Functions. In order to distribute this functionality easily over a set of nodes, it is required for the functions to be self-containing. The code has been written in such a way that functions are self containing, which also allow error messages (for instance) to be specific to the function.

8 FUTURE WORK

Large Datasets

The current method allows for importing large datasets, (using parameter *lowmemory=True*), however the size depends on the available RAM of the computer. Various solutions can be applied, such as chunking up data into blocks is a possibility (using parameter *dfchunk* [25]).

Computer Imprecision

Not part of the assignment, but relative to the course, in appendix C an interesting inherent imprecision is discussed.

REFERENCES

[1] MCO Mishap Investigation Board. 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[2] N Niki de Kadt, TJG Peeters, F Langerak, and AA Alblas. 2015. Accelerating the Learning Curve at ASML. https://pure.tue.nl/ws/portalfiles/portal/46916020/839652-1.pdf

[3] Marcus P Foster. 2013. Quantities, units and computing. *Computer Standards & Interfaces* 35, 5 (2013), 529–535. https://doi.org/10.1016/j.csi.2013.02.001

[4] Boudewijn R Haverkort and Armin Zimmermann. 2017. Smart industry: How ICT will change the game! *IEEE internet computing* 21, 1 (2017), 8–10. https://doi.org/10.1109/MIC.2017.22

[5] Jay Lee, Hung-An Kao, Shanhu Yang, et al. 2014. Service innovation and smart analytics for industry 4.0 and big data environment. *Procedia Cirp* 16, 1 (2014), 3–8. https://doi.org/10.1016/j.procir.2014.02.001

[6] Edward Waltz, James Llinas, et al. 1990. *Multisensor data fusion*. Vol. 685. Artech house Boston, Norwood, United States.

[25] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

A TESTING EXISTING METHODS

Each library is evaluated on the following points:

- How does it work?
- Did any error occur?
- Solutions of error prevention and consequences
- Conclusion

PintPy

How does it work? Pintpy appears to be a very powerful, rich library. It can verify whether the intended output unit actually represents the same measurement as the output unit (temperature unit in means the output unit also needs represent a temperature).

```
import pint
ureg = pint.UnitRegistry()

# PintPy Input:
(2 * ureg.meter + 2 * ureg.ft)
# Output:
<Quantity(2.6095999999999999, 'meter')>
```

PintPy is mainly designed to sum two unit systems and immediately convert them to one unit. The library can be tricked by summing a ’0’ amount of the desired output unit system to the actual input unit amount.

```
# Fool PintPy with this input:
(0 * ureg.meter + 2 * ureg.ft)
# Output:
<Quantity(0.6095999999999999, 'meter')>
```

Did any error occur? Despite it’s vast set of unit systems, multiple errors occurred. The library misinterpreted some units, resulting in confusing error messages. Furthermore the system requires you to specify the units inside the code, which requires programmers to understand which units are being used.

Solutions of error prevention and consequences. This is where this library really shows it’s negative side. The syntax requires hardcode programming the unit of a variable (such as a column in a tabulation). This means that we cannot dynamically change the unit, thus it is not very suitable unless we expect our personnel to all understand programming, and are comfortable with everybody being able to change the code!

A solution to this could be to write special functions that retrieve the correct attributes for PintPy or add dictionary values to supply the correct corresponding attributes into PintPy. This is certainly feasible but would result in multiple translations, as we first have to translate our input unit to a unit that is understood by the package, and vice versa. It is definitely a possibility and is not to be ruled out from future work, however preference was given to write an alternative solution as PintPy does not suit our ’freedom to choose input and output conversions easily’ desire.

Conclusion. The programming library requires the programmer to understand which units he is converting, as the syntax demands unit coding. An alternative would be to integrate multiple functions to translate units to the correct unit for the PintPy program, or retrieve the correct attributes. As these alternative solutions would not yield a clean code experience, I consider it the main issue with this (otherwise) suitable package.

PiPy: Unit-convert

PiPy is a very simple to understand library which seems to imitate what PintPy does. It can combine two items and converts them into another unit, so it can take multiple units at the same time. This is a promising library as this potentially allows us to do complex conversions at the same time.

How does it work? The syntax is similar to PintPy, yet slightly more natural to interpret, as the desired output is on the last part of the code line.

```
from unit_convert import UnitConvert
# yards and kilometers are inputs, converted to miles
UnitConvert(yards=136.23, kilometres=60).miles
# Output
37.3597678005
```

Did any error occur? Yes, my first test of the temperature variable was not recognized as a measurement type. This surpsrised me a lot and I discovered that this library actually only converts data (computerstorage), time, distance and mass.

Solutions of error prevention and consequences. This would require adjusting the library itself, which in essence does not provide a ’off-the-shelf’ solution. Furthermore it suffers from the same problem as PintPy, where it requires the programmer to hardcode the desired units inside the code (loosing versatility).

Conclusion. This is a limited library and not suitable for any complex implementation.

PiPy: Unit-converter

How does it work? Unit-converter allows us to specify exactly the specific scientific notation of units. This is really suitable to be applied in scientific situations, where often data is accompanied by the scientific notation.

Did any error occur? Yes, this library actually exposed many issues with unit notation. As the example code shows, the library expects temperatures to be accompanied by a special character: °C for Celsius. Unfortunately the program does not accept any other notation for temperature scales (Fahrenheit also succumbs this annotation requirement).

```python
from unit_converter.converter import convert, converts
# The special character ° is required
converts('52°C', '°F')
# Output, but which unit?
'125.6'
```

Solutions of error prevention and consequences. The issue with the character requirement originated from the CSV file exported from Excel. When using the standard CSV format in Excel, it is not encoded in 'UTF-8', required in order to add this special character. After implementing this additional character, errors occurred in other parts of the code but this could be overcome. The question becomes how versatile this software is in order to use it in multiple solutions.

Conclusion. This library is not useful for mass implementation due to the requirements for special characters, which can suddenly lead to errors.

String format: Additionally I would like to highlight that the library requires parameters in the string format. String formats are very versatile as they can represent any type of value and thus also are heavy data containers from a memory perspective. The aim for this research is not about memory space or computational speed but as this case is so excessive it is worth pointing out.

B CODE OF THE PROPOSED METHOD

The code sections have been divided into the important items:

- Hard-coding Unit Conversion: Using a Dictionary, section 5 and 6
- Applying a Formula When Needed, section 5 and 6.
- Pre-Cleaning Data, section 5 and 6.
- Error Handling, section 7 and 6.
- Output Storage, section 7 and 6.

All code is also available at [26].

Hard-coding Unit Conversion: Using a Dictionary

As described in section 5 and 6.

```python
def
cleanup_dictionary_values_convert_numbers_to_float_and_indicated_strin
string_indicator):

    for item in list(dict.keys()):
        try:
            # cleanup the style of numbering and decimal points
            dict[item] = cleanup_data_values_return_float(dict[item])
        except:
            try:
                if type(dict[item]) is str and dict[item].startswith(string_indicator):
                    # updates value to a 'pointer' of the actual formula
                    dict[item] = eval(dict[item])
            except:
                pass
            pass
    return
```

Applying a Formula When Needed

As described in section 5 and 6.

```python
def convert_units_from_dict(dict_to_use, unit_subject, unit_system, unit_specs, data_in, on_error_return_runtimerror=True, on_wrong_datatype_return_string=False):

    message_error_string = ' is the datatype value supplied, but it must be a floating point or integer datatype'

    retrieval_value = unit_subject + '_' + unit_system + '_' + unit_specs

    if on_error_return_runtimerror:
        if retrieval_value not in dict_to_use:
            # this will stop the program!
            raise RuntimeError("Unit "  + retrieval_value + " not found in dictionary. Please update data or dictionary.")

    try:
    x = dict_to_use[retrieval_value]
        if callable(x):
            # applies the unit transform function as stated in dictionary
            return x(data_in)
```

[26] https://github.com/mihransimonian/Uva-Block3-Big_Data-Assignment

```python
        else:
            # multiple with unit transform amount (e.g. conversion rate)
            return data_in * x
    except:
        # errorhandling
        if on_wrong_datatype_return_string:
            # return a string that can be used for storage in database or identification purposes
            return str(type(data_in)) + message_error_string
        else:
            pass
    return
```

Pre-Cleaning Data

As described in section 5 and 6.

```python
def cleanup_data_values_return_float(data_in, on_error_return_runtimerror=True, on_wrong_datatype_return_errormessage_string=False):

    if type(data_in) is str:
        # assure all commas become dots
        data_in = data_in.replace(',','.')
        # assure we only remain with the most right dot
        if data_in.count('.') > 1:
            data_in = data_in.replace('.','', data_in.count('.') - 1)

    try:
        # convert to float
        data_in = float(data_in)
    except:
        # errorhandling, see next part of report

    return data_in
```

Error Handling

As described in section 7 and 6.

```python
def cleanup_data_values_return_float(data_in, on_error_return_runtimerror=True, on_wrong_datatype_return_errormessage_string=False):

    message_error_string = ' is the datatype value supplied, but it must be a string datatype'

    try:
        # convert to float
        data_in = float(data_in)
    except:
        # errorhandling
        if on_error_return_runtimerror:
            if type(data_in) is bool:
                # this will stop the program!
                raise RuntimeError("Boolean " + message_error_string)
            if type(data_in) is str:
                # this will stop the program!
                raise RuntimeError("String " + message_error_string)
        elif on_wrong_datatype_return_errormessage_string:
            # return a string that can be used for storage in database or identification purposes
            return str(type(data_in)) + message_error_string
```

Output Storage

As described in section 7 and 6.

```python
# store file to database
filepath_for_storing_the_cleaned_database = (filepath_for_storing_the_cleaned_database + '_' + str(supplier_name))
storing_filesystem = str(storing_filesystem.lower())

if storing_filesystem == 'csv':
    # csv
    # options for compression: csv, gzip, zip, bz2, xz.
    # compression = infer, takes from filename
    df_import.to_csv(
        (filepath_for_storing_the_cleaned_database + '.csv'),
        index=False,
        header=True,
        sep=';',
        mode='w',
        compression='infer',
        encoding='UTF-8',
        decimal='.'
    )

elif storing_filesystem == 'sql':
    # sql
    from sqlalchemy import create_engine
    engine = create_engine('sqlite://', echo=False)
    filepath_for_storing_the_cleaned_database = filepath_for_storing_the_cleaned_database.replace('.', '')
    filepath_for_storing_the_cleaned_database =
    ↪  filepath_for_storing_the_cleaned_database.replace('\\',
    ↪  '')
    df_import.to_sql(
        (filepath_for_storing_the_cleaned_database),
        con=engine,
        if_exists='replace',
        index_label='id',
    )

elif storing_filesystem == 'hdf':
    # hdf
    df_import.to_hdf(
        (filepath_for_storing_the_cleaned_database + '.h5'),
        key=system_name_for_product_id,
        mode='w',
        complevel=5,
        complib='zlib',
        append=False,
        format='table',
        errors='strict',
        encoding='UTF-8',
    )

elif storing_filesystem == 'parquet':
    # parquet
    df_import.to_parquet(
        (filepath_for_storing_the_cleaned_database),
        engine='auto',
        compression='snappy',
        index=None,
    )

elif storing_filesystem == 'feather':
    # parquet
    df_import.to_parquet(
        (filepath_for_storing_the_cleaned_database +
        ↪  '.feather'),
    )

else:
    # csv is the main goto storage system for this project
    df_import.to_csv(
       (filepath_for_storing_the_cleaned_database + '.csv'),
        index=False,
        header=True,
        sep=';',
        mode = 'w',
        compression = 'infer',
        encoding='UTF-8',
        decimal = '.'
    )
```

C COMPUTER IMPRECISION

In theory and nature, there are an unlimited amount of numbers possible; a number can have thousands (or many more) of digits behind the comma and end with a '1'. Computer bits determine the amount of digits we can have in a number, the same rules do not apply as in nature. Due to computer bit precision, numbers can only be represented in computer memory up to a certain precision, resulting in a slightly inaccurate result, or stored variable.

Please refer to the below to see the effect:

```
# Input
x = 0.1 + 0.2
print(x)
# Output, notice the difference
0.30000000000000004
```

Influence on databases

This imprecision has distorts the content of the database. Especially when converting small and large numbers with one another we can discover issues. Realistically though, this will only be an issue in extreme cases, such as performing high precision calculations for NASA.

Solving Computer Number Imprecision

By implementing additional code one can verify whether there is a neglect able difference, compared to the number calculated. In other words, a ratio comparison can be made. The consequence is that this will require more computational counts, and thus reduces the system's processing speeds.