Documentation of CBonds Project

Summer Internship Project

Shayona Basu
Supervisors: Steve Harcourt and Kristyna Rodrigues
Dates: June to September (remote)

Abstract

Creating a process to summarise the weekly CBonds release, reconcile it with WFI data, and produce an excel file per manually selected field with tabs for Missings, Mismatched securities, and a summary of these findings called Summary.

NOTE: have to update some rules in the following fields:
- SecuredBy
- ResISIN
- DebtCurrency
- InterestPaymentFrequency (note about converting NaNs to 0's)
- Maturity Structure and Floating (Missing -> empty)

Overview of the process:

Section 1: Reading in the data

I use the pandas library method read_excel to read the **CBonds** data which are in excel format, and read_csv to read the **WFI** data. I saved pandas as ‘pd’, so when I use a pandas function, writing ‘pd’ followed by a full stop, calls the Pandas library, and retrieves the appropriate function. So, for example pd.read_excel, is calling the ‘read_excel’ function from the Pandas library or ‘pd’.

Reading CBonds data:

I set the parameters as below to read the excel CBonds file,

```
df = pd.read_excel(filename, header = 0, dtype = object, engine =
'openpyxl', na_values = [' ','N/A', 'NaN', 'nan', 'null',
'','#N\A', '#NA','-NaN','n/a','-1.#IND','-1.#QNAN'],
keep_default_na= False)
```

Manually Inputting File Location (important):

Currently, the project relies on hardcoded locations pointing towards folders that store .csv files of the CBonds and WFI dataset. While the project still has this manual component, please note the following steps to update the file paths with the most recent weeks datasets.

1. Download the latest release of CBonds data from FileZilla
2. Download EDI’s latest release of data and make sure it is saved in a folder
3. Navigate towards get_dataframes.py
4. On line 14, and 15 respectively, you will find the two following constants
   ```
   CBONDS_FOLDER_PATH = "file_path_for_CBONDS_folder"
   WFI_FOLDER_PATH = 'file_path_for_WFI_folder'
   ```
5. Replace the strings with the correct file path to the folders

Setting each data type as ‘object’:
Some parameters to note, is specifying ‘dtype’ as ‘object’. Dtype stands for ‘datatype’, which means what the format the data is going to be read as. For example, a number value of 1, will be treated as a number, or ‘int’, but it could also be treated as a string, and so accessed as ‘1’. When python finds mixed dtype values, or values with characters not in the accepted characters for a number eg. ‘/’ , which we see in dates, python can store it as an ‘object’ or as a datetime object. Usually when calling the function **read_excel**, python reads through the column values and tries to **guess** the datatype. With CBonds providing us such a large file size, this **guessing** of data types for each column by reading each row, slows down the process significantly. For this reason, I added this parameter to tell this function to treat every field as an ‘object’ dtype.

Previously, I manually hardcoded dictionaries of each field, and the dtype, as an alternative to this panoptic setting. However, I was getting a lot of errors, and this approach seemed to be very error prone, as you had to have specific dictionaries for each CBond file. Specifically, if a new field was added, or a field name in CBonds was altered slightly, the entire reading of the file would crash. For this reason, for the program to be more resistant to small changes, I settled on setting each data type to an object.

Specifying NA values:
I set this to avoid the string value 'NA' to be treated as none. 'NA' is the code value for Namibia.

Reading WFI:

Here is the line to read WFI EDI data, which are given as tab delimited text files:

```
df = pd.read_csv(pa, header = 1,sep = '\t', encoding =
"ISO-8859-1", encoding_errors = "ignore", low_memory=False)
```

Notes:
The encoding encapsulates different languages like Russian, and other characters found in these databases.
The header is set to 1, and not 0, because the first line is the title of the file. The second line in the txt file is the column names.
I could set dtype to 'object', but dtype's default of 'infer' adds another 10-15 seconds, and so I do not see a huge positive in enacted a standard dtype.

Universe Size & Timings:

The size of the universe from the databases downloaded on the 2023/09/09 and 2023/09/10:

Universe size:

Default	(8594, 16)
Emitents	(280211, 30)
Emissions	(758923, 99)
INDEF	(15992, 10)
SCMST	(827438, 41)
CONVT	(11637, 27)
ISSUR	(73598, 29)
BOND	(827438, 135)

Time it takes to load each file:

```
reading  Default
finished reading  Default  TIME:  2.721985064999899
reading  Emitents
finished reading  Emitents  TIME:  161.00709547600127
reading  Emissions
```

```
finished reading  Emissions  TIME:  1448.2572918160004
reading file
finished reading INDEF csv file, TIME:  0.23124585399637
reading file
finished reading SCMST csv file, TIME:  10.43838906098972
reading file
finished reading CONVT csv file, TIME:  0.12103742500767112
reading file
finished reading ISSUR csv file, TIME:  0.45409713001572527
reading file
finished reading BOND csv file, TIME:  76.20212729802006
```
(Time measured in seconds)

Subsection: Dealing with three different ISINS in CBonds

Our current workflow involves working with datasets that can contain up to 800,000 rows and three columns, each recording unique ISIN codes for securities. The challenge we face is ensuring that each row represents a single security with only one ISIN code, as multiple ISINs in a single row can lead to data integrity issues.

Proposal:
I propose the implementation of an optimized data cleaning and **ISIN selection method** using NumPy, which offers substantial performance improvements over our current approach that uses pandas. The code will perform the following tasks:

1. Count the number of non-null ISIN codes in each row.
2. Select the first non-null ISIN code (if present) and store it in a new column called 'Selected_ISIN'.
3. Drop the original ISIN columns to maintain a clean and efficient dataset.

Example:
To illustrate the improvement in processing speed, let's consider a recent dataset with 758,923 rows and three ISIN columns. Using the optimized NumPy approach, the data cleaning and selection process took approximately 20 seconds, a significant reduction from over 8 minutes using the previous pandas-based method.

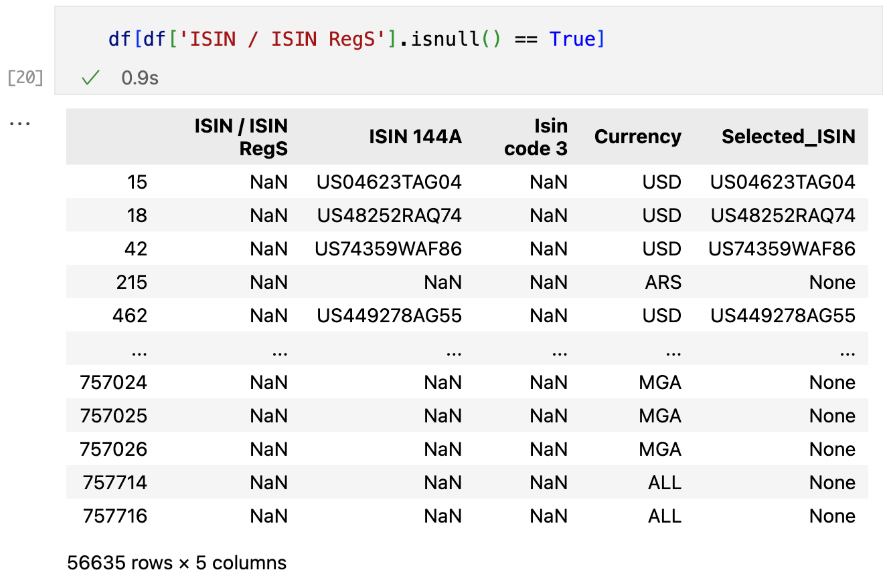
```python
isin_cols = ['ISIN / ISIN RegS', 'ISIN 144A', 'Isin code 3']

# Create a new column 'Selected_ISIN' using NumPy
df['Selected_ISIN'] = np.where(df[isin_cols].count(axis=1) > 0, df[isin_cols].bfill(axis=1).iloc[:, 0], None)

# Drop the original ISIN columns
df = df.drop(isin_cols, axis=1)
```

Benefits:
- Enhanced Efficiency: The proposed method substantially reduces the processing time for our large datasets, allowing us to work more efficiently and analyse data more quickly.
- Improved Data Integrity: Ensuring that each row contains only one ISIN code minimizes the risk of data errors and inconsistencies.
- Scalability: The optimized approach is scalable and can handle even larger datasets without a significant increase in processing time.

Display of how the program works, where it takes the first ISIN it see's from the three columns. If the first is empty, and the second is non empty, it will store the second (as below).



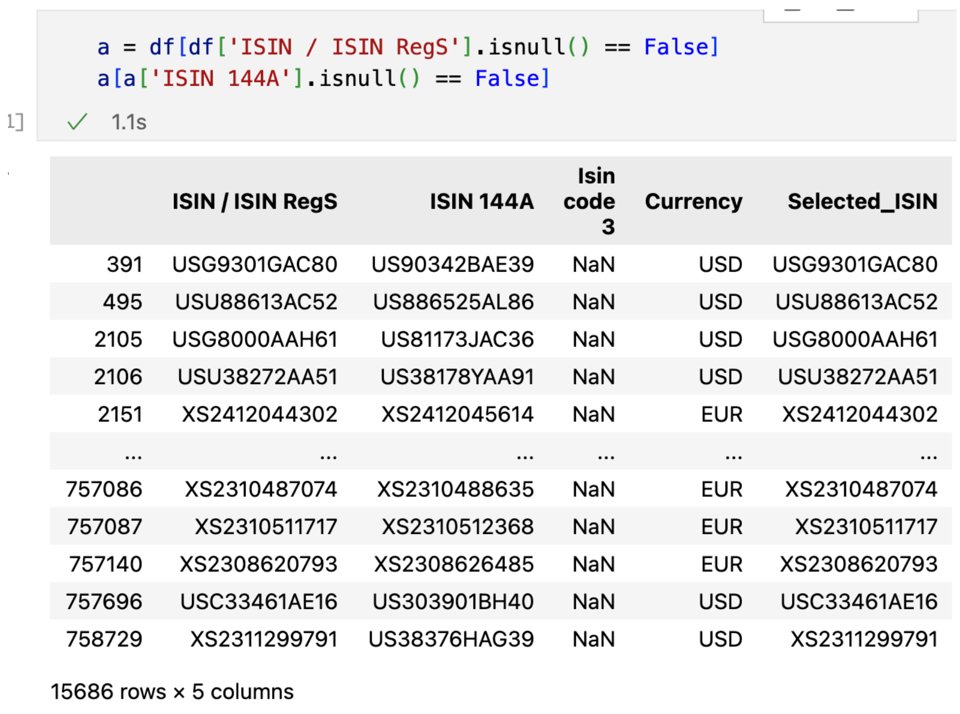
```
df[df['ISIN / ISIN RegS'].isnull() == True]
```

[20] ✓ 0.9s

	ISIN / ISIN RegS	ISIN 144A	Isin code 3	Currency	Selected_ISIN
15	NaN	US04623TAG04	NaN	USD	US04623TAG04
18	NaN	US48252RAQ74	NaN	USD	US48252RAQ74
42	NaN	US74359WAF86	NaN	USD	US74359WAF86
215	NaN	NaN	NaN	ARS	None
462	NaN	US449278AG55	NaN	USD	US449278AG55
...	...	...	...	...	...
757024	NaN	NaN	NaN	MGA	None
757025	NaN	NaN	NaN	MGA	None
757026	NaN	NaN	NaN	MGA	None
757714	NaN	NaN	NaN	ALL	None
757716	NaN	NaN	NaN	ALL	None

56635 rows × 5 columns

If more than one ISIN field is non-empty (as below), the program will store the first found ISIN in the 'Selected_ISIN' column.



```
a = df[df['ISIN / ISIN RegS'].isnull() == False]
a[a['ISIN 144A'].isnull() == False]
```

1] ✓ 1.1s

	ISIN / ISIN RegS	ISIN 144A	Isin code 3	Currency	Selected_ISIN
391	USG9301GAC80	US90342BAE39	NaN	USD	USG9301GAC80
495	USU88613AC52	US886525AL86	NaN	USD	USU88613AC52
2105	USG8000AAH61	US81173JAC36	NaN	USD	USG8000AAH61
2106	USU38272AA51	US38178YAA91	NaN	USD	USU38272AA51
2151	XS2412044302	XS2412045614	NaN	EUR	XS2412044302
...	...	...	...	...	...
757086	XS2310487074	XS2310488635	NaN	EUR	XS2310487074
757087	XS2310511717	XS2310512368	NaN	EUR	XS2310511717
757140	XS2308620793	XS2308626485	NaN	EUR	XS2308620793
757696	USC33461AE16	US303901BH40	NaN	USD	USC33461AE16
758729	XS2311299791	US38376HAG39	NaN	USD	XS2311299791

15686 rows × 5 columns

Section 2: Creating Field Object

The following code expert reads the Connecting_Field.csv, which is a table of each CBond field we want to encapsulate in our process. I save each of these fields into a dictionary, where the key is the **CBonds field name,** and the value is a **Field_Item** object I created to store all the values. This allows me to access each of the values by making it a parameter of the field. I preferred this over a list or another dictionary, due to flexibility of using a more object oriented approach. And as this object is quite simple, python handles this really well, and this object oriented design quickly became the backbone of my system, and was easy to use.

```
class Field_Item:
def __init__(self, cbonds_field = None, cbonds_file = None,
wfi_field = None, wfi_lookup = None, match_rules = None):
        self.cbonds_field = cbonds_field
        self.cbonds_file = cbonds_file
        self.wfi_field = wfi_field
        self.wfi_lookup = wfi_lookup
        self.match_rules = match_rules
```

This also allows further automation, as setting up this connection means greater automation within the code, and less hard coding connections.

Table 1: Definitions of parameters Field_Item

Parameter	Definition
cbonds_field	Name of the CBonds field. Used as the key when looping through fields

	and outputting the file name, because there is a one to one, but also one to two possibility of connecting it to a WFI field. For example, for one CBonds field, we might need to connect it to two wfi fields, and so creating two excel files for one CBonds field name. There is one exception where two CBonds fields connect to one WFI field.
cbonds_file	The file name of the CBond data. There is ‘Emitents’, ‘Emissions’ and ‘Default
wfi_field	Name of the wfi field .. Used as the name when exporting file name
wfi_lookup	The wfi lookup table. Flexible and reads whatever is in the file. These fields currently use BOND, SCMST, CONVT and INDEFF
match_rules	I did not really end up implementing **match_rules,** especially because we were still really going through them and deciding the labels. Once that was done, then I would not have to manually use my lists, where the name of the list would be the ‘match_rules’ type. So, I can use it, I just need to write the match_rules in the .csv first

Section 3: Merging CBonds and WFI tables

There were specific for some fields

Section 4: Applying rules for Mismatch and Missing

Categories of Fields:

CBonds data, from EMITENTS, DEFAULT and EMISSIONS tables, matched to WFI Fields,

*Table 2: Table of **definitions**, which would make it easier to read the code.*

Variable/Expression	**Explanation**
fe	Name of the Field Item object
mdf	Merged dataframe
!=	Not equivalent
==	equivalent
isnull()	The field is empty. Recall the definitions for empty in pd.read_excel
choices = [‘missing’, ‘missing’]	Assigning the label, depending on if the CBonds and wfi values match the conditions. For ease of the reader, I coloured the ‘mismatch’ condition as blue,

	and ‘missing’ condition as missing.
mmdf	Final dataframe that contains all the **mismatch** securities, to be conevreted to an excel tab
msdf	Final dataframe that contains all the **missing** securities, to be conevreted to an excel tab

I also am rewriting the code into pseudocode, so the logic can be read universally

Exact Matches:

If a field is labelled as an ‘exact match’, then this field undergoes the same dataframe transform. Exceptions can occur during building of the merged dataframe

```
conditions = [
            (mdf[fe.cbonds_field] != mdf[fe.wfi_field]) &
((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == False)),
            ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == True))]
        choices = ['mismatch', 'missing']
        mdf['Match?'] = np.select(conditions, choices, default=' ')
        mmdf = mdf[mdf['Match?'] == 'mismatch']
        msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch:
CBonds value **is not equal** to the WFI value, AND CBonds value and wfi value **is not empty**

Missing:
CBonds value **is not empty** and WFI value **is empty**

This rule is applied to all of the below fields,

Exact:
1. 'CFI / CFI RegS',
2. 'Currency',
3. 'CFI 144A',
4. 'ISIN of underlying asset',
5. 'CIK number'

There are a subset of fields that are also ‘exact matches’, but are manipulated a bit differently because they are not string values.

NUMBERS
1. 'Integral multiple',

2. 'Minimum settlement amount / Calculation amount',
3. 'Price at primary placement',
4. 'Margin',

Numbers

This is how we treat numbers, which are the follow fields:
'Integral multiple',
'Minimum settlement amount / Calculation amount',
'Price at primary placement',
'Margin',

```
cbondnotempty = mdf[mdf[fe.cbonds_field].isnull() == False]
msdf = cbondnotempty[cbondnotempty[fe.wfi_field].isnull() == True]
misma = cbondnotempty[cbondnotempty[fe.wfi_field].isnull() == False]

if fe.cbonds_field == 'Margin':
    #adding Coupon rate col to Margin
    tomerge = DATAFRAMES['Emissions'].loc[:,['Coupon rate (eng)','ISIN / ISIN
RegS']]
    tomerge = tomerge.sort_values('ISIN / ISIN RegS')
     mdf = mdf.reset_index().merge(tomerge, on = 'ISIN / ISIN RegS',how =
'left').set_index('SecID')
     misma[fe.cbonds_field] =  misma[fe.cbonds_field].map('{:.2f}'.format)
else:
     misma[fe.cbonds_field] = misma[fe.cbonds_field].map('{:.1f}'.format)

misma[fe.cbonds_field] = misma[fe.cbonds_field].astype('float64')
mmdf = misma[misma[fe.wfi_field]!=misma[fe.cbonds_field]]
```

Note: Exception for margin, because we add another column to the merged dataframe. In the highlighted lines of code, we merge ‘Coupon rate (eng)’ from the Emissions file to Margin’s merged dataframe

Mismatch	**Missing**
Make sure cbonds field has 2 trailing zero’s if ‘Margin’, else, 1 trailing zero. Mismatch if wfi != cbonds	Cbonds field **is not empty** and WFI **is empty**

Dates:

1. 'Maturity date',
2. 'Settlement date',
3. 'Date until which the bond can be converted'

Yes/No with Unique rules:

For every field that is labelled as Y/N, I convert CBond's 0's and 1 integer values to string 'N' and 'Y''s, using the following function.

```
def change_to_YN(mdf, fe):
    '''
    changing cbonds column from 0,1's to Y,N's
    input:  mdf: pd.DataFrame, merged data frame
            fe: FieldObject
    output: mdf: pd.DataFrame, changed inplace
    '''
    mdf[fe.cbonds_field] = mdf[fe.cbonds_field].astype(int, errors = 'ignore')
    mdf[fe.cbonds_field] = mdf[fe.cbonds_field].replace([0,1], ['N','Y'])
    return mdf
```

1. 'Subordinated debt (yes/no)',

```
conditions = [
    (mdf[fe.cbonds_field] != mdf[fe.wfi_field]) &
((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == False)),
    ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == True))]
choices = ['mismatch', 'missing']
mdf['Match?'] = np.select(conditions, choices, default=' ')
mmdf = mdf[mdf['Match?'] == 'mismatch']
msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**

Cbonds value != WFI value and CBonds and WFI value are **not empty**	Cbonds field **is not empty** and WFI **is empty**

2. 'Mortgage bonds (yes/no)',

```
conditions = [
                        ((mdf[fe.cbonds_field] == 'Y') &
(~mdf[fe.wfi_field].str.contains('M', na = False)) &
(mdf[fe.wfi_field].isna() == False)) |
                        ((mdf[fe.cbonds_field] == 'N') &
(mdf[fe.wfi_field].str.contains('M'))),
                        ((mdf[fe.cbonds_field] == 'Y') &
(mdf[fe.wfi_field].isna() == True))]
              choices = ['mismatch', 'missing']
              mdf['Match?'] = np.select(conditions, choices,
default='')
              mmdf = mdf[mdf['Match?'] == 'mismatch']
              msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
Cbonds == 'Y', and WFI string **does not contain** M and **not empty** OR CBonds == 'N' and WFI string **does contain** M	Cbonds field == 'Y' and WFI **is empty**

3. 'Structured products (yes/no)', (4)

```
conditions = [
                   ((mdf[fe.cbonds_field] == 'N') &
(mdf[fe.wfi_field] == 'SP')) |
                   ((mdf[fe.cbonds_field] == 'Y') &
(mdf[fe.wfi_field] !='SP')& (mdf[fe.wfi_field].isna() == False)),
                   ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == True))]
              choices = ['mismatch', 'missing']
              mdf['Match?'] = np.select(conditions, choices,
default=' ')
              mmdf = mdf[mdf['Match?'] == 'mismatch']
              msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**

Cbonds == ‘N’, and WFI == ‘SP’ OR CBonds == ‘N’ and WFI != ‘SP’	Cbonds field == ‘Y’ and WFI **is empty**

4. 'Floating rate (yes/no)', (5)

```
conditions = [
                ((mdf[fe.cbonds_field] == 'N') &
(mdf[fe.wfi_field].str.contains('FR'))) |
                ((mdf[fe.cbonds_field] == 'Y') &
(mdf[fe.wfi_field].str.contains('FR') == False)),
                ((mdf[fe.cbonds_field] == 'Y') &
(mdf[fe.wfi_field].isnull() == True))]
            choices = ['mismatch', 'missing']
            mdf['Match?'] = np.select(conditions, choices,
default=' ')
            mmdf = mdf[mdf['Match?'] == 'mismatch']
            msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
Cbonds == ‘N’, and WFI value CONTAINS FR OR CBonds == ‘Y’ and WF value DOES NOT CONTAIN FR	Cbonds field == ‘Y’ and WFI **is empty**

5. 'Covered debt (yes/no)', (3)

```
conditions = [ #FIX THIS
     ((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field] != 'CB')
(mdf[fe.wfi_field].isna() == False)) |
     ((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field] ==
'CB')),
   ((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field].isnull() ==
True))]
            choices = ['mismatch', 'missing'] #missing should be 0
            mdf['Match?'] = np.select(conditions, choices,
default=' ')
            mmdf = mdf[mdf['Match?'] == 'mismatch']
            msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**

Cbonds == ‘Y’, and WFI != ‘CB’ and **not empty** OR CBonds == ‘N’ and WFI == ‘CB’ CBonds == ‘Y’ and WFI != ‘R’ and is **not empty** OR	Cbonds field == ‘Y’ and WFI **is empty**

6. 'Perpetual (yes/no)', (6)

```
conditions = [
((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field] == 'P')) |
((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field] == 'U')) |
((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field] == 'I')),
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field].isnull() ==
True))]
            choices = ['mismatch', 'missing']
            mdf['Match?'] = np.select(conditions, choices,
default=' ')
            mmdf = mdf[mdf['Match?'] == 'mismatch']
            msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
Cbonds == ‘N’, and WFI != ‘P’ and **not empty** OR CBonds == ‘N’ and WFI == ‘U’ OR CBonds == ‘N’ and WFI == ‘I’	Cbonds field == ‘Y’ and WFI **is empty**

7. 'Placement type (eng)',

```
mdf.loc[(mdf['Placement type (eng)'] == 'Public') &
(mdf['PrivatePlacement'] == 'Y'),"Match?"] = 'mismatch'
mdf.loc[(mdf['Placement type (eng)'] == 'Private') &
(mdf['PrivatePlacement'] == 'N'),"Match?"] = 'mismatch'
mmdf = mdf[mdf['Match?'] == 'mismatch']
a = mdf[mdf['Placement type (eng)'] == 'Private']
msdf = a[a['PrivatePlacement'].isna() == True]
```

Mismatch	**Missing**
CBonds == ‘Public’ and WFI == ‘Y’ CBonds == ‘Private’ and WFI == ‘N	Cbonds field == ‘Private’ and WFI **is empty**

8. 'Securitisation',

```
conditions = [
((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field].isna() ==
False)),
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field].isna() ==
True))]
                choices = ['mismatch', 'missing']
                mdf['Match?'] = np.select(conditions, choices,
default=' ')
                mmdf = mdf[mdf['Match?'] == 'mismatch']
                msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
CBonds == ‘N’ and WFI **is not empty**	Cbonds field == ‘Y’ and WFI **is empty**

9. 'Pik',

```
conditions = [
                     ((mdf[fe.cbonds_field] == 'N') &
(mdf[fe.wfi_field].isna() == False)),
                     ((mdf[fe.cbonds_field] == 'Y') &
(mdf[fe.wfi_field].isna() == True))]
                choices = ['mismatch', 'missing']
                mdf['Match?'] = np.select(conditions, choices,
default=' ')
                mmdf = mdf[mdf['Match?'] == 'mismatch']
                msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
CBonds == ‘N’ and WFI **is not empty**	Cbonds field == ‘Y’ and WFI **is empty**

10. 'Convertable (yes/no)', (2)

```
conditions = [
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field] == 'R')) |
((mdf[fe.cbonds_field] == 'N') & ((mdf[fe.wfi_field] !=
'R')&(mdf[fe.wfi_field].isna() == False) )),
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field].isna() ==
True))]
```

```
            choices = ['mismatch', 'missing']
            mdf['Match?'] = np.select(conditions, choices,
default='')
            mmdf = mdf[mdf['Match?'] == 'mismatch']
            msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
Cbonds == 'Y', and WFI == 'R' OR CBonds == 'N' and WFI != 'R' and is **not empty** OR CBonds == 'Y' and WFI != 'R' and is **not empty** OR	Cbonds field == 'Y' and WFI **is empty**

11. 'SPV (yes/no)'

```
conditions = [
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field] != ‘SPV’) &
(mdf[fe.wfi_field].isna() == False)) |
((mdf[fe.cbonds_field] == 'N') & (mdf[fe.wfi_field] == 'SPV')),
((mdf[fe.cbonds_field] == 'Y') & (mdf[fe.wfi_field].isna() ==
True))]
            choices = ['mismatch', 'missing']
            mdf['Match?'] = np.select(conditions, choices,
default=' ')
            mmdf = mdf[mdf['Match?'] == 'mismatch']
            msdf = mdf[mdf['Match?'] == 'missing']
```

Mismatch	**Missing**
Cbonds == 'Y', and WFI != 'SPV' **and** WFI **is not empty** OR CBonds == 'N' and WFI == 'SPV'	Cbonds field == 'Y' and WFI **is empty**

Unique Rules:

1. 'Coupon frequency',

```
Python
mmdf, msdf = run_coupon(mdf, fe)
```

```
Python
def run_coupon(mdf, fe):
  mdf[fe.cbonds_field] = mdf[fe.cbonds_field].fillna(0).astype(int)
  conditions = [
    ((mdf[fe.cbonds_field].isnull() == False) & (mdf[fe.wfi_field].isnull() ==
True))]
  choices = ['missing']
  mdf['Match?'] = np.select(conditions, choices, default='n')
  msdf = mdf[mdf['Match?'] == 'missing']
  mmdf = mdf[mdf['Match?'] == 'n']

  def cpf(row):
    if row["InterestPaymentFrequency"] == "ITM" and row['Coupon frequency'] == 0:
      return 'match'
    if row["InterestPaymentFrequency"] == "ANL" and row['Coupon frequency'] == 1:
      return 'match'
    if row["InterestPaymentFrequency"] == "182" and row['Coupon frequency'] == 2:
      return 'match'
    if row["InterestPaymentFrequency"] == "SMA" and row['Coupon frequency'] == 2:
      return 'match'
    if row["InterestPaymentFrequency"] == "180" and row['Coupon frequency'] == 2:
      return 'match'
    if row["InterestPaymentFrequency"] == "91D"  and row['Coupon frequency'] == 4:
      return 'match'
    if row["InterestPaymentFrequency"] == "QTR" and row['Coupon frequency'] == 4:
      return 'match'
    if row["InterestPaymentFrequency"] == "BIM" and row['Coupon frequency'] == 6:
      return 'match'
    if row["InterestPaymentFrequency"] == "35D"and (row['Coupon frequency'] == 10 or
row['InterestPaymentFrequency'] == 11 ):
      return 'match'
    if row["InterestPaymentFrequency"] == "MNT" and (row['Coupon frequency'] == 11 or
row['Coupon frequency'] == 12 or row['Coupon frequency'] == 13):
      return 'match'
    if row["InterestPaymentFrequency"] == "28D" and (row['Coupon frequency'] == 12 or
row['Coupon frequency'] == 13 or row['Coupon frequency'] == 28):
      return 'match'
    if row["InterestPaymentFrequency"] == "WKY" and (row['Coupon frequency'] == 51 or
row['Coupon frequency'] == 52 ):
      return 'match'
    else:
      return 'mismatch'
  testdf = mmdf.assign(Match=mmdf.apply(cpf, axis=1))
  mismatchdf = testdf[testdf['Match'] == 'mismatch']
  mismatchdf.drop('Match', axis = 1)
  return (mismatchdf, msdf)
```

Mismatch	Missing

If not:
For each row in merged data frame (mdf), check the row's CBOND and WFI columns value to be equal to a value, if it does label as '**match'**. Goes through all conditions (ifelse). If none of the conditions match, label as **mismatch**
`    if row["InterestPaymentFrequency"] == "ITM" and row['Coupon frequency'] == 0:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "ANL" and row['Coupon frequency'] == 1:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "182" and row['Coupon frequency'] == 2:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "SMA" and row['Coupon frequency'] == 2:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "180" and row['Coupon frequency'] == 2:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "91D"  and row['Coupon frequency'] == 4:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "QTR" and row['Coupon frequency'] == 4:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "BIM" and row['Coupon frequency'] == 6:`
`      return 'match'`
`    if row["InterestPaymentFrequency"] == "35D"and (row['Coupon frequency'] == 10 or row['InterestPaymentFrequency'] == 11 ):`
	Cbonds field **is not empty** and WFI **is empty**

```
        return 'match'
    if row["InterestPaymentFrequency"] == "MNT" and
(row['Coupon frequency'] == 11 or row['Coupon frequency']
== 12 or row['Coupon frequency'] == 13):
        return 'match'
    if row["InterestPaymentFrequency"] == "28D" and
(row['Coupon frequency'] == 12 or row['Coupon frequency']
== 13 or row['Coupon frequency'] == 28):
        return 'match'
    if row["InterestPaymentFrequency"] == "WKY" and
(row['Coupon frequency'] == 51 or row['Coupon frequency']
== 52 ):
else:
        return 'mismatch'
```

2. 'Country of the issuer (eng)',

```
from rules import CountryOfIssuer
  mdf['Match?'] = [CountryOfIssuer.get(v, "n") for v in
mdf['Country of the issuer (eng)']]
mmdf = mdf[mdf['Match?'] != mdf['CntryofIncorp']]
        conditions = [
            ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field].isnull() == True))]
        choices = [True]
        mdf['Missing'] = np.select(conditions, choices,
default=False)
        msdf = mdf[mdf['Missing'] == True]
```

Mismatch	**Missing**
Connecting CBonds value with wfi value through dictionary. If not matching, label ‘mismatch’	Cbonds field **is not empty** and WFI **is empty**

Handling the Namibia Issue

The country code for Namibia is ‘NA’, and pandas reads this as None value. I tried changing this in the pd.read_excel /pd.read_csv, but we still get this error. For a a quick and robust approach to to handle the 'Namibia' and 'NA' issue in our DataFrame, I implemented the following code:

` ` `

```
nam = msdf[msdf['Country of the issuer (eng)'] == 'Namibia']
nam['CntryofIncorp'] = nam['CntryofIncorp'].astype(str)
nami = nam[nam['CntryofIncorp'] == 'nan']
condition = ~msdf.index.isin(nami.index)
msdf = msdf[condition]
```

Here's why it works:

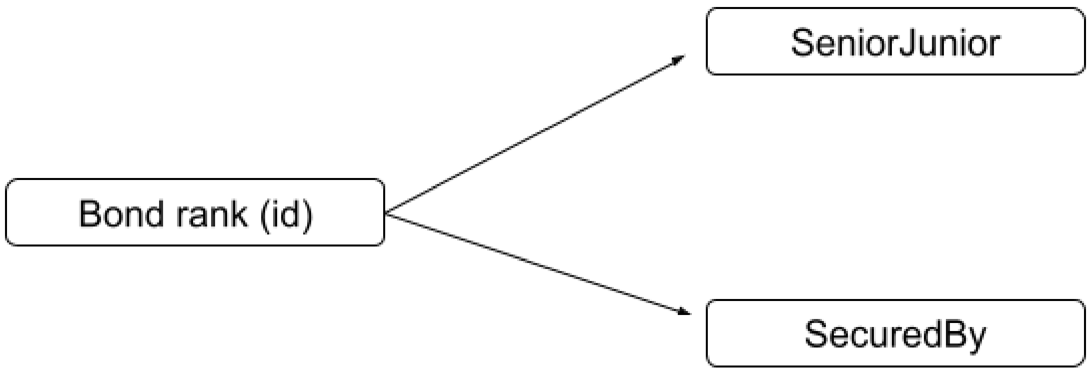
We identify rows where the 'Country of the issuer (eng)' is 'Namibia.' In these cases, 'NA' in 'CntryofIncorp' represents 'Namibia,' so we convert these values to strings to ensure clarity.

For rows where 'CntryofIncorp' equals 'nan,' we create a 'nami' subset. These are potential cases where 'NA' might be mistaken for a null value. We remove them from the DataFrame while preserving the rest of the data.

This approach safeguards 'NA' as a valid code for 'Namibia,' reducing the risk of misinterpretation.

3. 'Bond rank (id)',

Figure 1: one CBonds field to two WFI fields



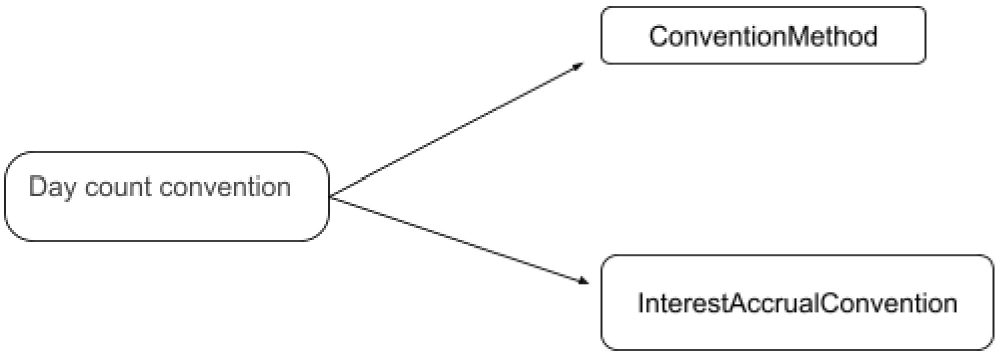
```
mdf[fe.cbonds_field] = mdf[fe.cbonds_field].astype(int)
        if fe.wfi_field == 'SeniorJunior':
            conditions = [
((mdf[fe.cbonds_field] == 1) & (mdf['SeniorJunior'] == 'S')) |
((mdf[fe.cbonds_field] == 2) & (mdf['SeniorJunior'] == 'S')) |
((mdf[fe.cbonds_field] == 3) & (mdf['SeniorJunior'] == 'S')) |
((mdf[fe.cbonds_field] == 5) & (mdf['SeniorJunior'] == 'J')) |
((mdf[fe.cbonds_field] == 7) & (mdf['SeniorJunior'] == 'N')) |
((mdf[fe.cbonds_field] == 8) & (mdf['SeniorJunior'] == 'P')) |
((mdf[fe.cbonds_field] == 0)) |((mdf[fe.cbonds_field] == 4)) |
((mdf[fe.cbonds_field] == 6)),
((mdf[fe.cbonds_field].isna() == False) &
(mdf['SeniorJunior'].isna() == True))]
```

Mismatch	**Missing**
Connecting CBonds value with wfi value through dictionary. If not matching, label ‘mismatch’	Cbonds field **is not empty** and WFI **is empty**

```
if fe.wfi_field == 'SecuredBy':
        pass
```

4. 'Day count convention',

Here again, we have **two** WFI excel outputs for one cbonds file



```
Python
if fe.wfi_field == 'ConventionMethod': #1
      conditions = [
      #have to check for cases when does not contain ISDA/ICMA and we have
ISDA/ICMA
      ((mdf[fe.cbonds_field].str.contains('ISDA',case = True)) &
(mdf['ConventionMethod'] != 'ISDA') & (mdf['ConventionMethod'].isna() ==
False))|
      ((mdf[fe.cbonds_field].str.contains('ICMA',case = True)) &
(mdf['ConventionMethod'] != 'ICMA') & (mdf['ConventionMethod'].isna() ==
False))|
      ((mdf[fe.cbonds_field].str.contains('ISDA',case = True) == False) &
(mdf['ConventionMethod'] == 'ISDA'))|
      ((mdf[fe.cbonds_field].str.contains('ICMA',case = True) == False) &
(mdf['ConventionMethod'] == 'ICMA')),
      #missing where CBONDS does not have ISDA & ICMA
      (((mdf[fe.cbonds_field].isna() == False) &
((mdf[fe.cbonds_field].str.contains('ISDA',case = True)) |
(mdf[fe.cbonds_field].str.contains('ICMA',case = True)))) &
(mdf['ConventionMethod'].isna() == True))]
      choices = ['mismatch', 'missing']
      mdf['Match?'] = np.select(conditions, choices, default=' ')
      mismatchdf = mdf[mdf['Match?'] == 'mismatch']
```

```
        msdf = mdf[mdf['Match?'] == 'missing']
```

Bond Rank and Convention Method

<table>
<tr><th>Mismatch</th><th>Missing</th></tr>
<tr><td>

CBonds	**WFI**
Contains ISDA	!= ISDA and **is not empty**
Contains ICMA	! = ICMA and **is not empty**
Does not contain ISDA	== ISDA
Does not contain ICMA	== ICMA

</td><td>

CBonds field must satisfy:
1. **Is not empty**
2. Contain ISDA OR
3. Contain ICMA

AND

WFI field **is empty**

</td></tr>
</table>

```python
Python
 if fe.wfi_field == 'InterestAccrualConvention': #0
        from rules import DCCLvl1, DCCLvl2
        conditions = [
          ((mdf[fe.cbonds_field].isnull() == False) & (mdf[fe.wfi_field].isnull()
== True))]
        choices = ['missing']
        mdf['Match?'] = np.select(conditions, choices, default='n')
        msdf = mdf[mdf['Match?'] == 'missing']
        mmdf = mdf[mdf['Match?'] == 'n']
        testdf = mmdf.assign(Match=mmdf.apply(DCCLvl1, axis=1))
        mismatchdf = testdf[testdf['Match'] == 'mismatch']
        mismatchdf = mismatchdf.drop('Match', axis = 1)
        testdf1 = mmdf.assign(Match=mmdf.apply(DCCLvl2, axis=1))
        mmdf1 = testdf1[testdf1['Match'] == 'mismatch']
        mmdf1 = mmdf1.drop('Match', axis = 1)
```

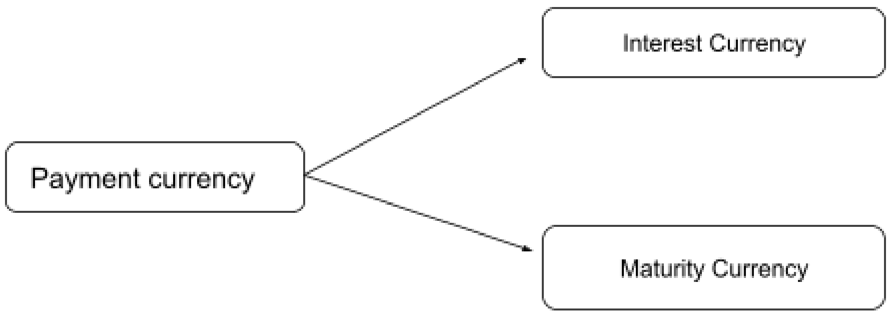
Mismatch	**Missing**

`from rules import DCCLvl1, DCCLvl2`
Importing rules from another file
 	
Level 1 comparison is in DCCLvl1
`testdf = mmdf.assign(Match=mmdf.apply(DCCLvl1, axis=1))`
	Cbonds field **is not empty** and wfi field **is empty**

Level 2 comparison is in DCCLvl2 `testdf1 = mmdf.assign(Match=mmdf.apply(DCCLvl2, axis=1))`	

5. 'Payment currency'

Unlike the others, here we are concating both WFI fields in output



Interest:

```
conditions = [
            (mdf[fe.cbonds_field] != mdf[fe.wfi_field[0]]) &
((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field[0]].isnull() == False)),
            ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field[0]].isnull() == True))]
        choices = ['mismatch', 'missing']
        mdf['Interest Match?'] = np.select(conditions, choices,
default=' ')
```

Maturity:

```
conditions = [
            (mdf[fe.cbonds_field] != mdf[fe.wfi_field[1]]) &
((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field[1]].isnull() == False)),
            ((mdf[fe.cbonds_field].isnull() == False) &
(mdf[fe.wfi_field[1]].isnull() == True))]
        choices = ['mismatch', 'missing']
        mdf['Maturity Match?'] = np.select(conditions, choices,
default=' ')
```

Missing:

```
        mmdf = mdf.loc[(mdf['Interest Match?'] == 'mismatch') &
(mdf['Maturity Match?'] == 'mismatch')]
```

Mismatch:

```
        msdf = mdf.loc[(mdf['Interest Match?'] == 'missing') &
(mdf['Maturity Match?'] == 'missing')]
```

Mismatch	**Missing**
Interest Currency:	