Contents

Tackling the creation of a functionality that a “numpy.where” function provides is a very interesting task, but it has some limitations.

Acknowledging the Data Types

In order to provide the user any option regarding the query choices that he has, firstly it must be aware of the type of the data that were loaded.

Thankfully, we can check some of the formation with the help of pandas:

- pandas.DataFrame.dtypes will return the type of the values in the dataframe columns.

```
[26]: df = pd.DataFrame({'float': [1.0],
                         'int': [1],
                         'datetime': [pd.Timestamp('20180310')],
                         'string': ['foo']})
      df.dtypes
```

```
[26]: float              float64
      int                  int64
      datetime    datetime64[ns]
      string              object
      dtype: object
```

In case we need a transformation based on the specific types above, to_numeric function can be used:

- pd.to_numeric() : can be handled by raising errors, transforming strings to NaN, or returning a boolean if the specific columns or rows can be transformed to numbers or not.

```
:  df = pd.DataFrame({'float': [1.0],
                      'int': [1],
                      'datetime': [pd.Timestamp('20180310')],
                      'string': ['foo']})

   df.apply(lambda s: pd.to_numeric(s, errors='coerce').notnull().all())
```

```
:  float        True
   int          True
   datetime     True
   string      False
   dtype: bool
```

Proposition of the Choices

Once the pre-processing of the table has been made, it suggests the available queries to the user based on the type of variables that exist. For instance:

```
type_list = df.apply(lambda s: pd.to_numeric(s, errors='coerce').notnull().all())

display(df)
print("Length of the column is: ", len(df.columns))
print()

for i in range(len(df.columns)):
    if type_list[i] == True:
        print("Perform a numerical operation")
```

	float	int	datetime	string
0	1.0	1	2018-03-10	foo

```
Length of the column is:  4

Perform a numerical operation
Perform a numerical operation
Perform a numerical operation
```

If not, it does not propose this option.

Otherwise, it can provide all possible options available and handle every operation

- { if the query succeeds, everything is ok }
- { if the query fails , raise an error }

<u>Advantages:</u>

It does not require any pre-processing regarding the types or proposition to the user.

User does not have to be aware of the dataset formation

<u>Disadvantages:</u>

In case it fails, it is inconvenient for the user, even if it provides the purpose of the error.

User must be aware of the exact formation and type of the dataset

Label Suggestion (Dropdown)

Again, with the help of the dataframe's functionality it obtains the name of the rows and columns, for instance:

```
display(df)
print("\n-- Column name for the dropdown --\n")
display(df.columns.values)
print("\n\n-- Row name for the dropdown --\n")
display(df.index.values)
```

	float	int	datetime	string
0	1.0	1	2018-03-10	foo

```
-- Column name for the dropdown --

array(['float', 'int', 'datetime', 'string'], dtype=object)


-- Row name for the dropdown --

array([0], dtype=int64)
```

Also, it should provide the functionality, on what to do with these labels with the help of signs.

"=", equal

">", bigger than

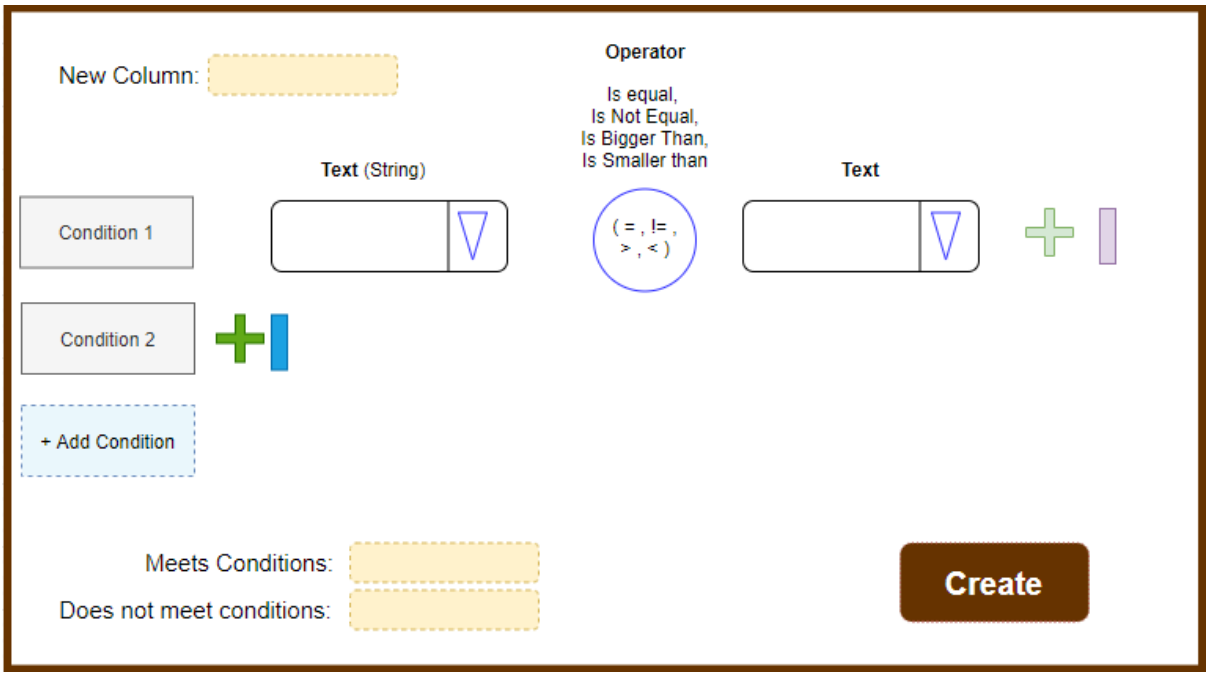
"<", smaller than

The "bigger than" and "smaller than" sign should suggest a numeric input from the user, if first it has been checked at the pre-processing stage for its type (confirmed the label consists of numbers)

The equality sign can be used in pretty much every type. Therefore, the input can be both a text and a number.

Interface

After the dataframe has been loaded, the user query bar should look something like this:



1. The user has the option to name the column that he wants to generate
2. Then he should select the first condition that he wants to apply

    - He can specify a new string or dropdown one of the existing “suggested labels” that were mentioned before.
    - Then the operator he wants to use.
    - And the text or label or number it should be (equal, not equal, bigger than, smaller than).

3. After the first selection, he can add another state on the existing condition or create a new one.
4. By clicking the light green cross or the purple vertical line  (AND, OR) he can specify if both or one of the two statements should be true to meet the condition.
5. The user can also create a different condition, but firstly as on point 4. he should also specify if both or one of the two statements should be active.
6. After determining the conditions, he will finally specify what happens in case it meets or fails to meet the conditions.
7. Finally, he will create the new column.

Test Driving the Example

```
df['points'] = np.where(
    ( (df['gender'] == 'male') & (df['pet1'] == df['pet2'] ) ) | (
(df['gender'] == 'female') & (df['pet1'].isin(['cat','dog'] ) ) )
    , 5, 0)
```

1. The user specifies the column name to 'points'.
2. At the first condition he dropdowns and finds the gender column name and sets it equal to 'male' as a string.
3. Then he adds another statement at the same condition, by clicking the  (AND) and he dropdowns the column names as sets the operator to "is equal".
4. After that he creates a new statement and specifies if both this one and the first argument should be true in order to meet the condition  (AND, OR).
5. Same as before he chooses the necessary inputs for use.
6. In the case of an element be inside of a list can follow the same principles, by adjusting the list creation on the third input.
7. Next, he writes what the output in case the condition meets of fails to meet should be.
8. Finally, he creates the table.

Handling the Input

The input can be handled with event listeners using the react.js.

When the user specifies each box, the specific value can be stored in a LIST.

This provides the option to examine step by step what the conditions of the user were.

Again, in a protype level a list of the above example could be something like:

["points", "gender_l", "=", "male", "and_sc", "pet1_l", "=", "pet2_l", "or_dc", "gender_l", "=", "female", "and_sc", "pet1_l", "exists in", ["cat", "dog"], 5, 0]

So, all the values needed to create the numpy.where function are available.

The Python Functionality

There are many approaches after the list is generated.

There can be a function that will take this list and recreate the “numpy.where” functionality.

1. Firstly, the first element and the last two elements of the list can be saved somewhere else so only the where clause is available on the list.
2. [“gender_l”, “=”, “male”, “and_sc”, “pet1_l”, “=”, “pet2_l”, “or_dc”, “gender_l”, “=”, “female”, “and_sc”, “pet1_l”, “exists_in”, [“cat”, “dog”]]
3. Every input selected or is a label, is transformed into df[‘name’] for the query.
4. The rest of the input are specified as they are e.g. “male”, “cat”.
5. The “and_sc” will have the meaning of “and same condition” and the “and_dc” the meaning of the “and different condition”. This way it can handle the parentheses of the np.where statement.
6. Other list inputs like the “exists_in” can be handled appropriately.

This way the query does not have to exist before-hand which provides flexibility to the application

Deployment

For deployment, working with AWS lambda provides an easier setup of the application, but it shows a disadvantage regarding the safety from outside attacks, while multiple functions can be harder to be monitored in comparison with the Amazon EC2. Handling the time out that may occur can make lambda a good choice over the errors when dealing with insecure networks of EC2. Since the application will not probably require too many packages the disadvantage of AWS lambda with the storage restriction will not affect the application. Also, both perform well when it comes to scalability, which is automated for the AWS and manual for the EC2. EC2 giving more freedom to the user, so it requires greater effort for setting up the infrastructure to setting up the security layers.

Overall, lambda is optimal with low code complexity. If the functionality gets out of hand thought, it may present some problems. Both have trade-offs regarding the security, but probably AWS lambda is the better choice here.