**1-6– RegEx**

**Regular Expressions, also known as RegEx, are a sequence of characters that allows the users to create patterns that helps match, locate and manage string data. They are convenient for extracting, restructuring, scraping and formatting a specific pattern of text from codes, log files, spreadsheets and documents. Regex is universal and can be used in all main programming languages and data formatting tools such as Tableau and Alteryx. Regex is a lot more flexible as compared to string functions and you can generate complex string operations while using less tools and code.**

**Regex Tool is used to leverage the powerful pattern matching abilities of regular expression syntax to parse, match, or replace data.**

**For our demonstration, we are going to use regex to clean this csv dataset which has 3 columns, ID, street\_address and email\_ip. Drag a regex tool from the parse tool tab and connect it to our input data. Configuration for the regex tool is separated in 3 parts, first is the field or column that will be parsed, 2nd is the regular expression pattern, and 3rd is the output where you need to specify the method used for parsing. There are 4 parsing methods on the tool; Replace to swap the text with a second expression, Tokenize to slip the incoming data using the regular expression pattern, Parse to separate the expression into new columns, and Match to check if the pattern is found on the string.**

For the first example, lets remove the numeric digits on the street\_address field and retain the address strings such as “Calypso Avenue”. First, set the column to parse into the field “street\_address”. Next, we need to set the regular expression. Input the following “(\d+)\s([A-Z]+)\s([A-Z]+)\s(\d{3})-(\d{2})-(\d{4})” On this regular expression, we have created a marked group for each section of the string by putting the pattern inside parentheses, the first set of parentheses will be the 1st marked group, and so on. The first group contains **/d+** which means one or more digits. Each parentheses group was separated by **\s** or space in between. The 2nd and 3rd group has the qualifier **[A-Z]** which signifies one or more letter from A to Z and the quantifier **+** that indicates there is more than 1 letter that follows it. This follows the string pattern of our text, where in the street number precedes the name of the place that is usually two to 3 words. For the last part of the regex, **\d{3}** indicates that there are exactly 3 digits followed by a dash symbol, then **\d{2}** for another 2 digits, another dash again to separate the digits, and lastly, **\d{4}** signifies that there are exactly 4 digits at the end of the string. For those new to regex, you can use the + button to access common regular expressions that you may need while creating your expression. Back to our configuration, we will set the Output method to “Replace”, then add the replacement text. Since we used marked groups, we will type in “$2 $3” meaning, the 2nd and 3rd marked group or the address words will be the replacement. We will leave the “Copy Unmatched Text to Output” unchecked since we wanted to see which rows were empty or unmatched. Run the workflow once done. As we can see from the output, ID 1 has the replacement “Calypso Avenue”. Comparing it to the original string, the street number zero was removed, along with the contact number on the suffix.

Next, we want to Tokenize the same street address data. This method is similar to the text to columns tool except instead of matching and removing what you do not want, you match for what you want to keep. Add another regex tool to the workflow, and connect it to our input data. Set the column to parse to the field “email\_ip”. Next, set the Regular Expression. Type in “[^\s]+”. This pattern looks for groups of consecutive characters (regardless if numeric or not) and excluding space in the string. Since we are using tokenize, this means that we are using the space as delimiter. The brackets **[]** enclose a list of possible characters to match, **^** caret means to exclude a specific symbol or character, and **/s** means space, so **^/s** indicates that we will exclude space. Next, set the output method to “Tokenize”. Once selected, we can now choose how to split our field. We can either split to columns, or split to rows. For this example, we will split to columns. Then set the number of columns. Since we only have 2 information for this field, email and ip address, set the number of columns to 2. You also have the option to set the behavior if the data matched exceeds the number of columns. You can choose from Drop extra with warning, drop extra without warning, or error to stop processing the workflow. Let’s set this to Drop Extra without warning. Finally, set the root name for the output split columns. We will retain the default email\_ip for this example. Once done, run the workflow. As we can see from the output, the regex tool split the string into email\_ip1 and email\_ip2. Email\_ip1 contains the email address, while Email\_ip2 contains the IP address.

Next, we are going to parse using the regex tool. For this third example, we are going to parse the telephone number from the street\_address string. Drag a 3rd regex tool to the canvas and connect it to our input data. In its configuration, set the column to parse to “street\_address”. After that, proceed to typing the regular expression. Type “(\d{3}-\d{2}-\d{4})” Unlike how we set the pattern during the replace method, we’ve placed the parentheses on the tail ends so we can parse the telephone number to 1 field only. The telephone numeric pattern consists of first 3 digits, followed by a dash, then 2 digits, and ended by 4 digits. **\d** signifies that it is a digit, and the curly brackets with a specific number indicates that it has a specific number of digits. To configure the output, set the method to “Parse” and set the Output Columns. On the table, you can rename the output field, set its data type and size. Lets set the name to “Telephone”, retain the default datatype V\_string, and minimize the size to 11. Once the configuration is complete, run the workflow. As we can see from the output, the new field “Telephone” was appended. Its value was extracted from the street address string. If ever one of the records did not match the exact number of digits that we indicated on regex, its telephone column will be null.

For the final example, we are going to look at how the match method works on regex. Duplicate the first regex tool where we used the replace method, and connect it to the input anchor. We are going to use the same column and regex, which is the street address and its string pattern that has the street number, address strings, and telephone combined. For the output method, set it to “Match”. Below the method dropdown is a textbox where we can set the fieldname for the new column that will tag if a match was found in the field or not. Let’s just leave it to the default, and leave the “error if not matched” unchecked. Once done, run the workflow. The resulting output were mostly tagged as true, meaning the regex pattern matched the string. But there are still a few records that are tagged False such as record ID 10 and 11, because it has 3 address strings on its text.

Aside from the patterns we’ve discussed, here are other regex codes you can use:

