# Introduction to Regular Expressions in Mp3Tag

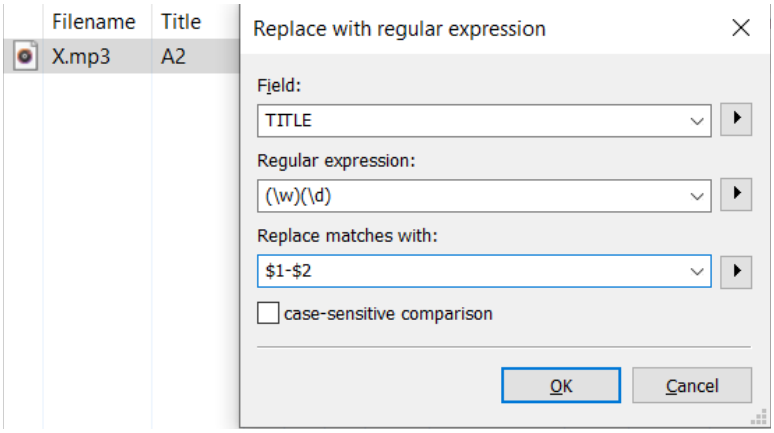
## Regular Expressions

Regular expressions are text written in a formal language that finds strings/part of strings that match a specified pattern. They're most often used in the **Replace with Regular Expression** action. Every character save these: **. | * ? + () {} [] ^ $** are *literal* characters in a regular expression, or characters that have no special meaning. The indicated characters above each perform some kind of operation, but can be made into literals by putting a \ slash in front of them.

The community forum compiles useful regular expressions here. Regular expressions are not unique to Mp3tag, but exist across programming languages. Any beginner's guide online can teach you more about them. I will go through the syntax that makes up a regular expression and offer an example for each one.

In regular expressions, parentheses have two roles: grouping items into a subexpression, same as in mathematical expressions, and marking what generated the match. Using a **$** before an expression in the "Replace matches with" field of **Replace with Regular Expression** will expand to the text that matched the sub-expression.

This means that unless marked as a non-capture group with **(?:)**, each set of parentheses will create a "capture group." A capture group is just a variable, stored in the number of the order that it was created. You can then use **$D**, with **D** being a digit, to summon which capture group you want. See the example below.




The regular expression **(\w)(\d)** looks for any word character and any digit directly next to each other, in that order. It then puts the word character it finds into the first capture group, and the digit into the second capture group. The parentheses are what define the capture group—

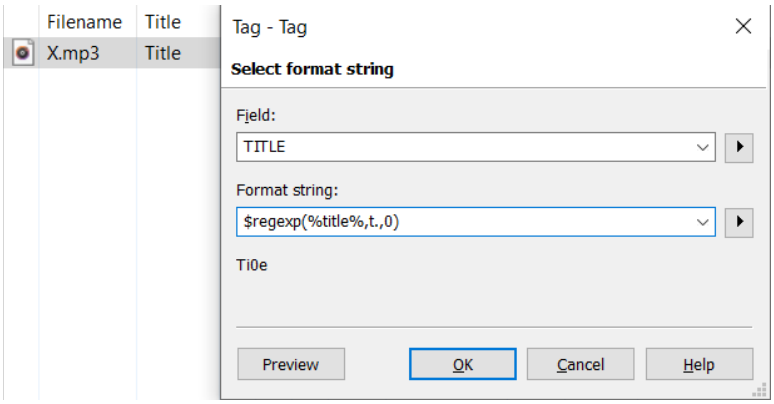
without them, no groups would be created. Then it replaces the sequence with the first capture group's value (A), a dash, and the second capture group's value (2). The result changes **A2** into **A-2**.

A non-capture group is used only to help the regular expression match its desired pattern. The result that it captures will be ignored, and not added to the numerical sequence (**$1**, **$2**, **$3**, etc.).

## Repeats

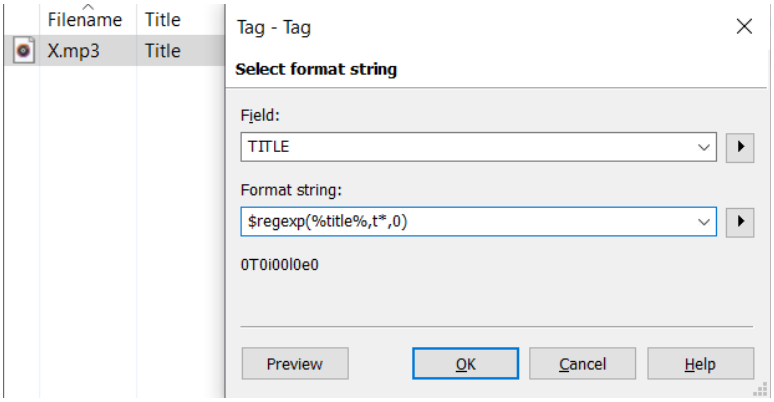
Repeats are expressions that include some repetition of a character.

The dot repeat (**.**, the period character) is a wildcard that can match any single character.



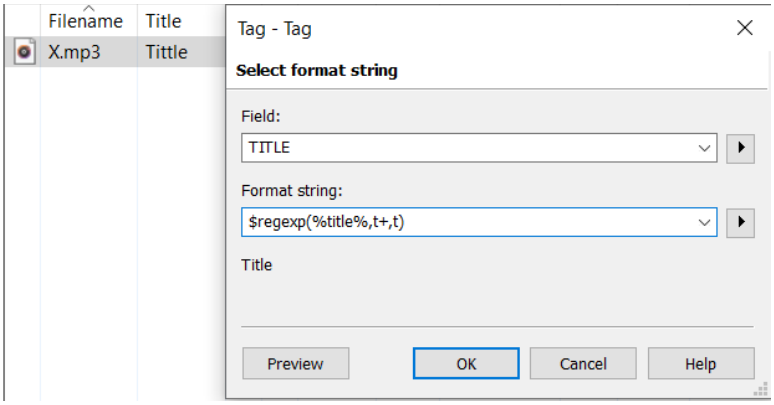

The period after "t" means that the pattern matches a lowercase "t" and any one character beyond it.

The asterisk repeat (*) searches for a character repeated any number of times, including zero.



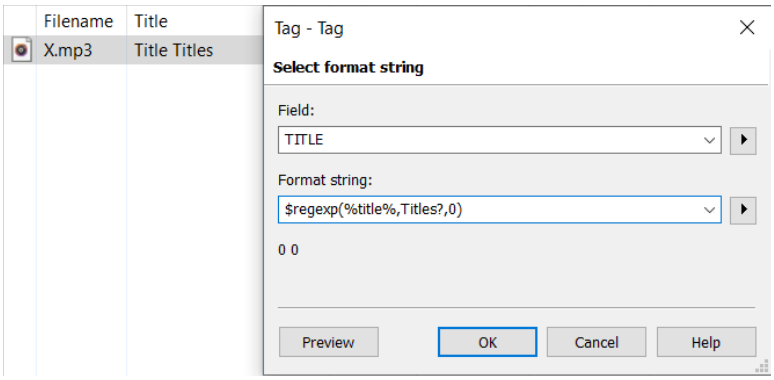

The pattern searches for any instance in which the character“t” is repeated… even if it’s repeated zero times. Thus, it inserts a zero in every possible area and replaces the one existing “t” with a zero as well.

The plus sign repeat (**+**) searches for a character repeated any number of, but at least once.




The pattern searches for instances in which “t” is repeated one or more times. When it finds the double “t” typo in the title, it replaces it with a single “t” (the last variable in the format string).

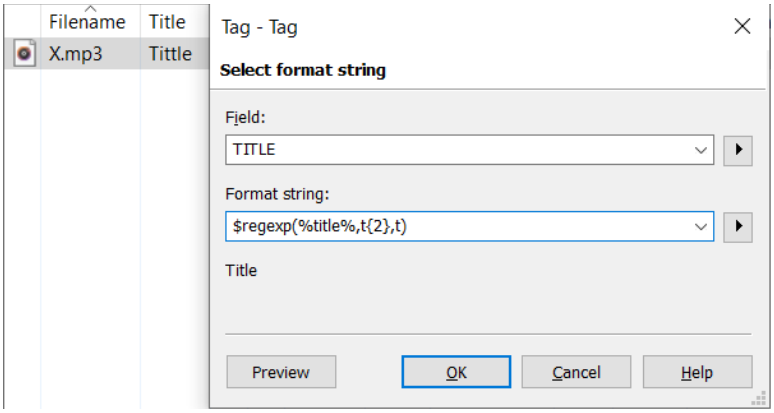
The question mark repeat (**?**) searches for a character repeated once or not at all.



Because the question mark is placed next to the “s,” the “s” becomes optional. Therefore, the pattern matches both “Title” and “Titles.”

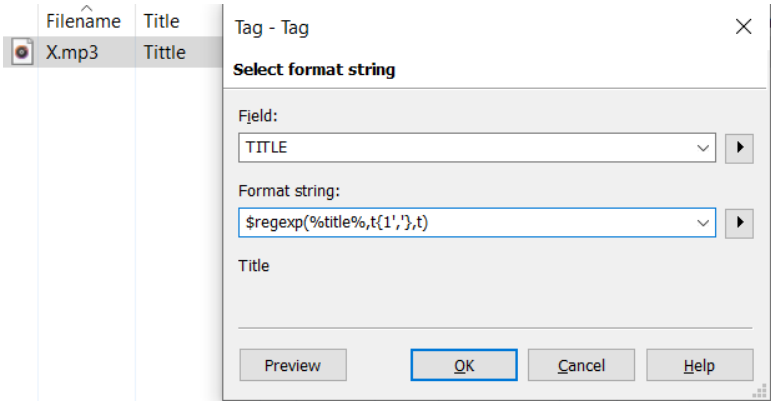
Included in repeats are **sequences**, which are repeats that specify how much a character should be repeated.

The sequence **a{n}** represents the letter **a** repeated exactly **n** times.



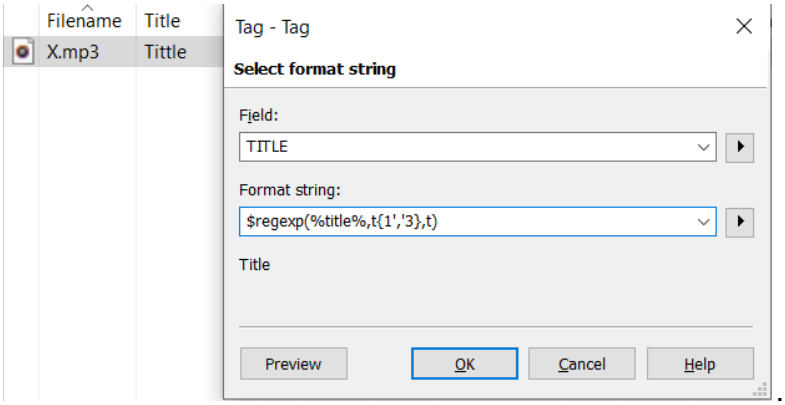

The pattern replaces the “tt” typo with one “t.”

The sequence **a{n, }** represents **a** repeated at least **n** times with no upper limit.



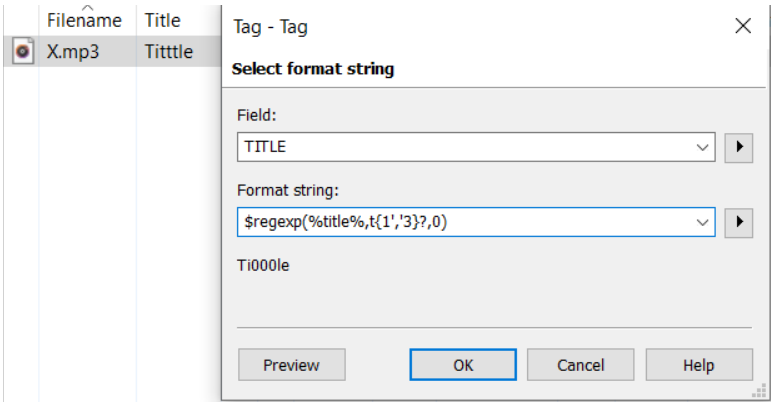

The pattern searches for the “t” character repeated at least once, but includes any greater number of repeats. Note that the comma is surrounded by single quotes to escape commas.

The sequence **a{n, m}** represents **a** repeated between **n** and **m** times.



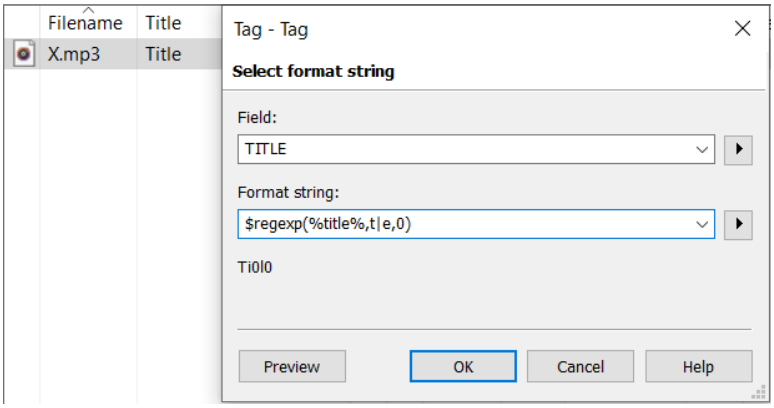

The pattern searches for the “t” character repeated between one and three times, including one and three.

Non-greedy (also called lazy) repeats, which are indicated by a **?** after using one of the repeats above, will match only the shortest possible string. Repeats are greedy by default.




By default, this example intends to replace any sequence of 1-3 “t” characters in a row with a zero. Because the question mark makes it non-greedy (looking for the shortest sequence), it replaces each of the three “t” characters in the middle of the title with a zero instead of replacing all three.

Another thing to note is **Alternatives**. Alternatives are the two or more options that occur when an expression can match one option OR another. They’re marked with a vertical bar (**|**) character in between them, which is located on the same key as the backslash.

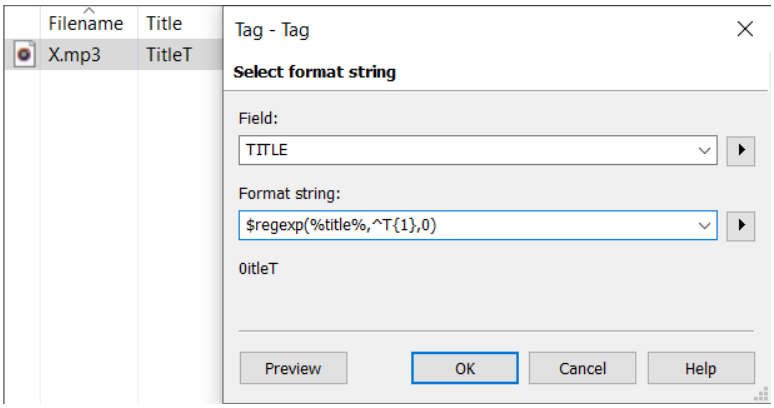



With an alternative bar between the “t” and “e” characters, the regular expression will replace both of them with a zero if they exist.

## Line Anchors

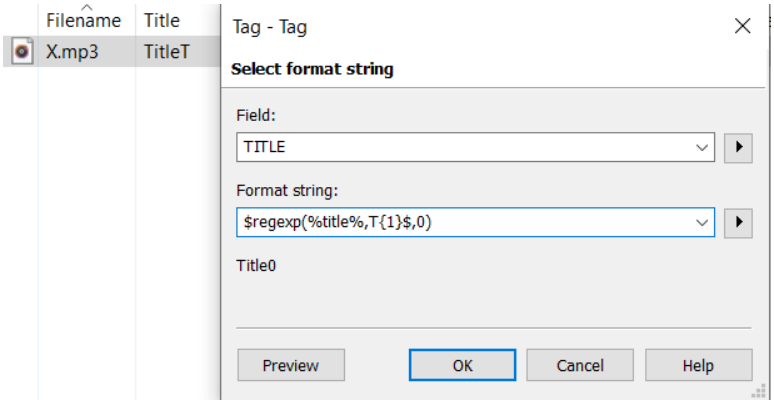
Line Anchors are indicators that tell the engine to look at either the beginning or the end of the line that the regular expression is searching in. Note that the caret should come before the regular expression and the dollar sign after it.

The caret(**^**) character matches the start of a line.




The caret symbol will replace the capital “T” at the beginning of the line, but not at the end.

The dollar sign (**$**) character matches the end of a line.

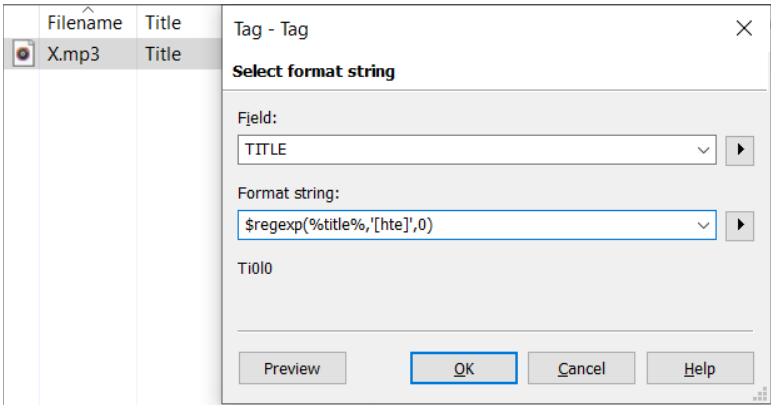



The dollar symbol will replace the capital “T” at the end of the line, but not at the beginning.

## Sets

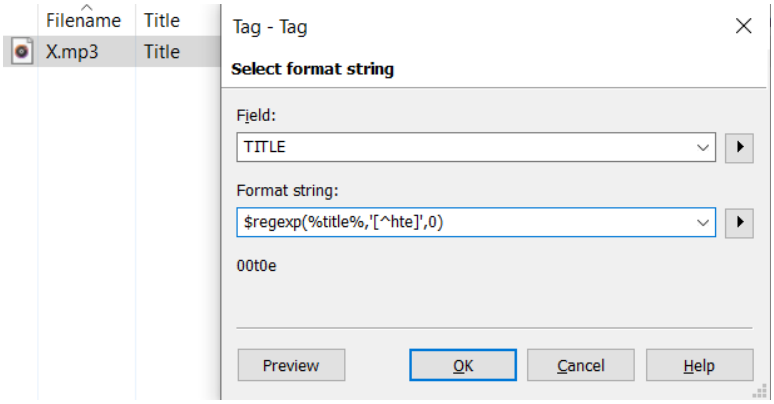
Sets are expressions that search for a set of characters. Please note that you must “escape” (indicate that they are not part of the greater syntax) square and curly brackets by putting single quotes around the expression.

The **[abc]** set matches either **a**, **b**, or **c**. Case-sensitive.



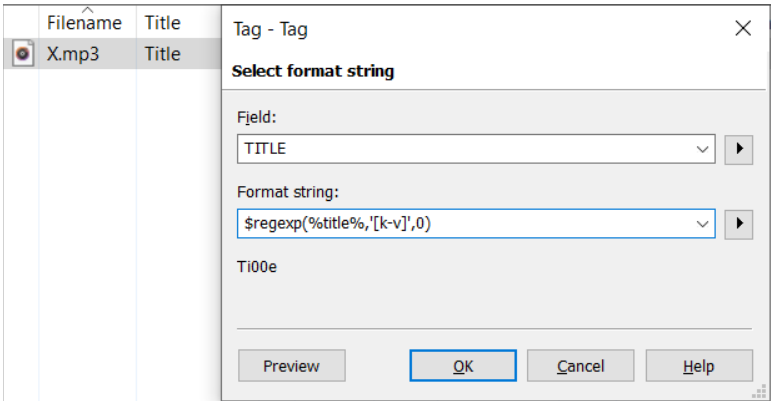

The pattern matches the letters “hte” and replaces them with zeroes. Because there is no “h,” there is nothing to replace. Because the function is case sensitive (which can be changed with the optional fourth parameter **c**), the capital “T” is not replaced.

The **[^abc]** set matches every character except **a**, **b**, or **c**. Because it’s also case-sensitive, it will still match the opposite case of the provided letter. (e.g. **[Aab]** will not match the capital and lowercase “a,” but will match the uppercase “B.”)



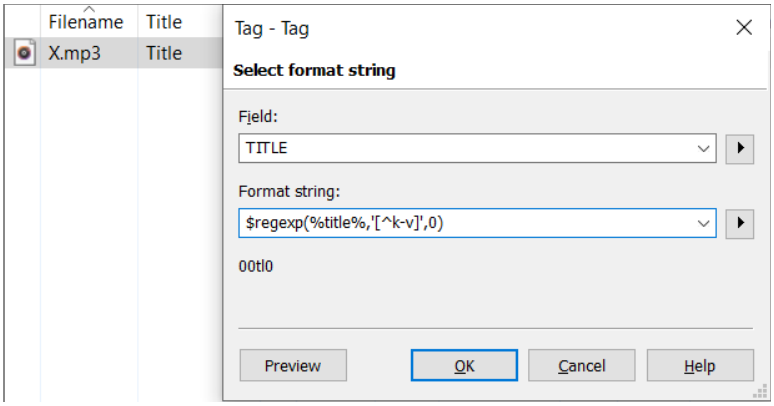

The pattern matches every character aside from “hte,” including the capital “T.”

The **[a-z]** set matches every character in the **a** - **z** range. Capitalizing the parameters (e.g. **[A-Z]**) will match capital letters only. Capitalizing one of the parameters but not the other (e.g. **[A-z]**) will match both capital and lowercase letters.



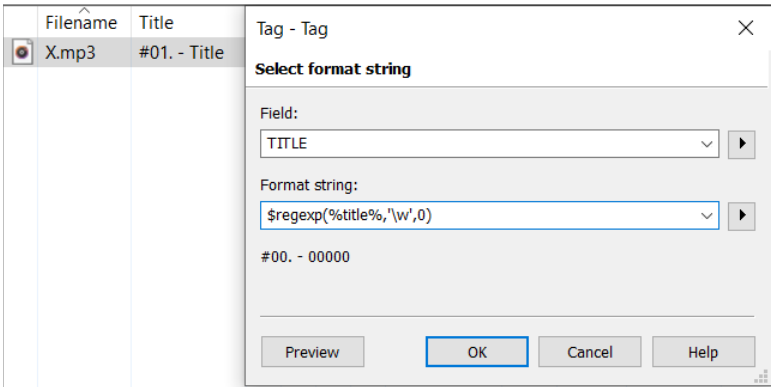

The pattern replaces “t” and “l” with zeroes, but not the capital “T.”

The **[^a-z]** set matches every character outside the **a** - **z** range. It will match all capital letters, as every capital letter is outside of a lowercase **a** - **z** range. If the range is **A** - **Z**, it will match all lowercase letters instead.



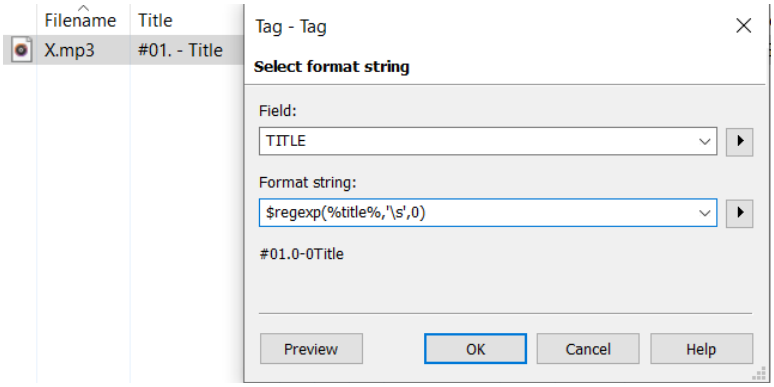

The pattern replaces all characters except for “t” and “l” with zeroes, including the capital “T.”

The **\w** set matches any word character, which is any alphanumeric character plus the underscore.



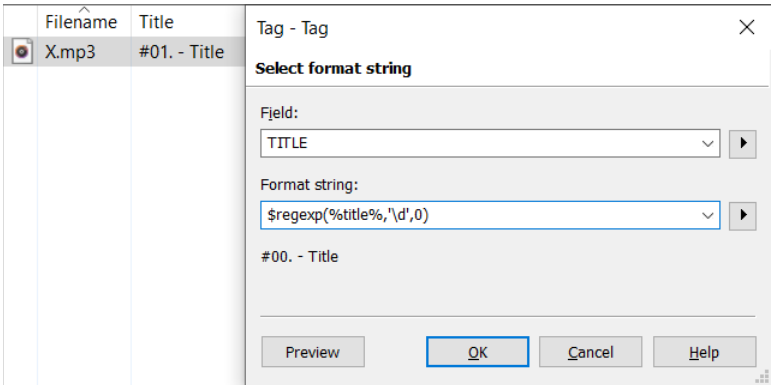

The pattern replaces all word characters with a zero, but excludes non-word characters.

The **\s** set matches any whitespace character, which is generally just the space. However, it also includes line breaks and other rare Unicode characters such as the medium mathematical space and the ideographic space.



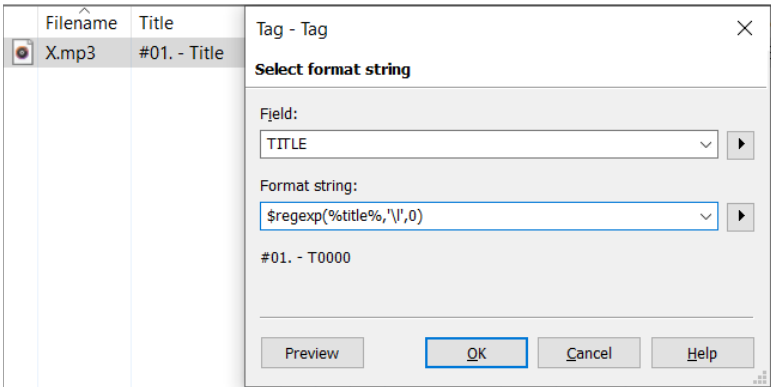

The pattern matches each space in the line.

The **\d** set matches any digit.



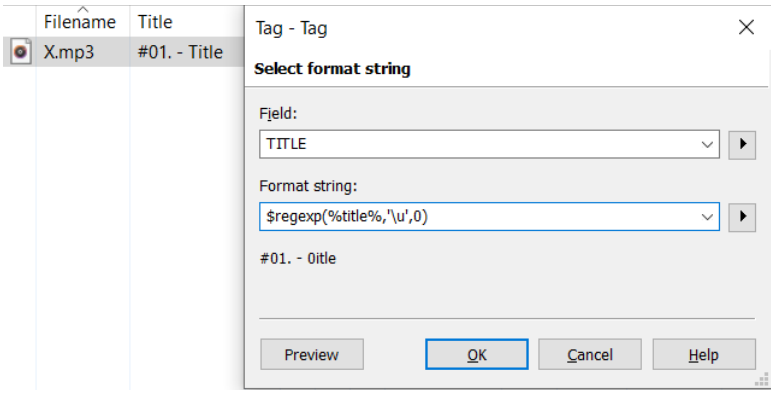

The pattern replaces the “0” and “1” digits at the beginning with zeroes.

The **\l** set matches any lowercase character.



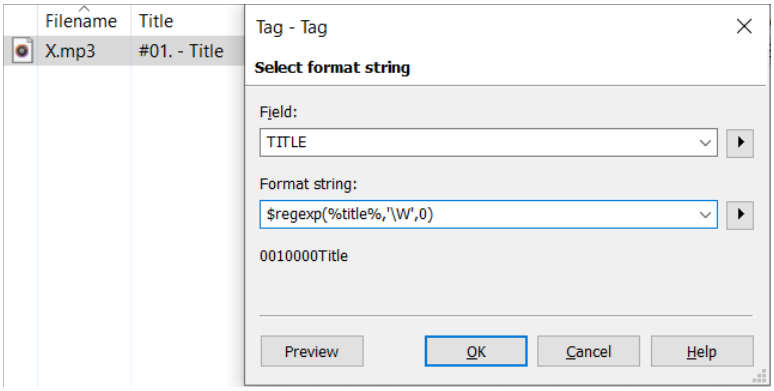

The pattern replaces all lowercase characters with zeroes.

The **\u** set matches any uppercase character.



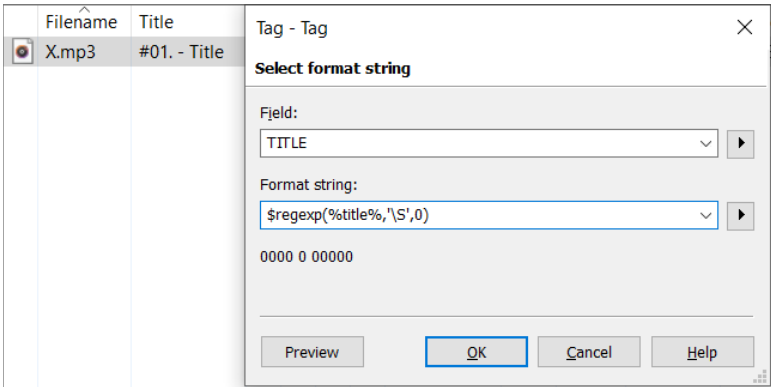

The pattern replaces all uppercase characters (in this case, only the capital “T”) with zeroes.

The **\W** set matches any non-word character, which is any non-alphanumeric character minus the underscore.



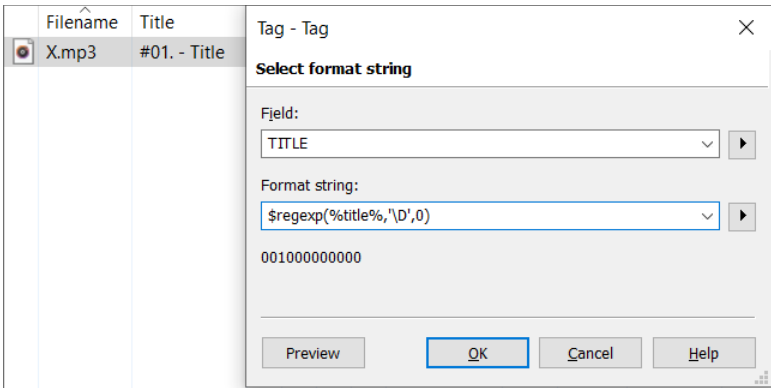

The pattern replaces all non-word characters with a zero, but excludes word characters.

The **\S** set matches any non-white space character.



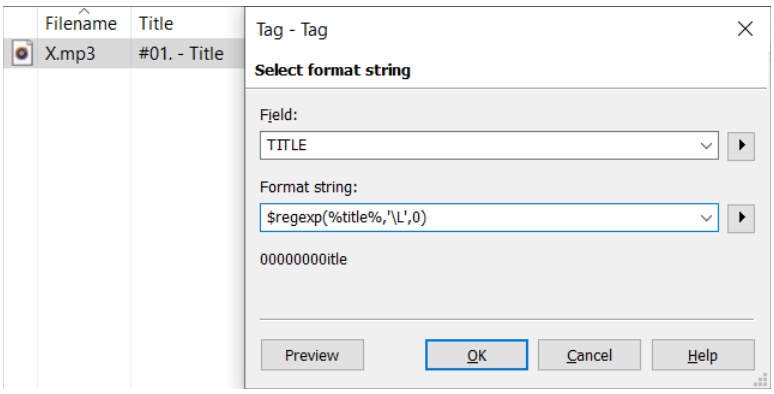

The pattern replaces all spaces with a zero.

The **\D** set matches any non-digit character.



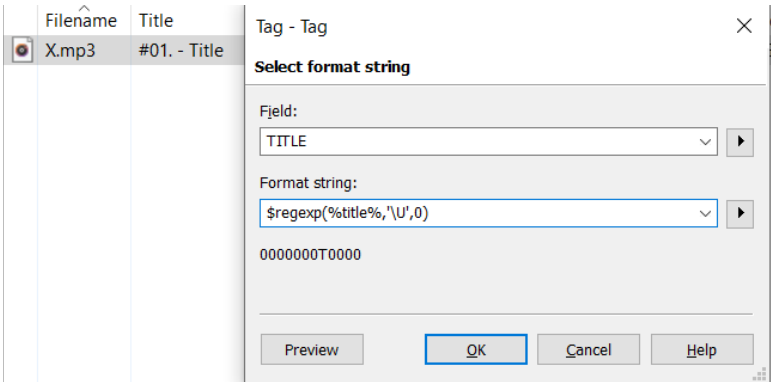
The pattern replaces all non-digit characters, including spaces, with a zero. Note that the zero already present remains the same.

The **\L** set matches any non-lowercase character.



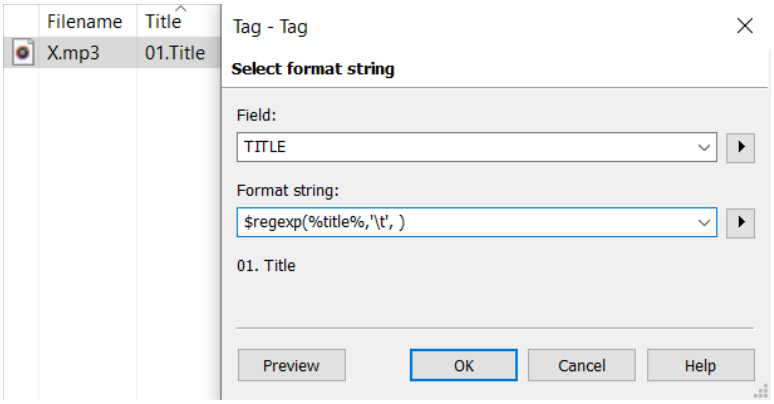

The pattern replaces all non-lowercase characters with a zero. This includes numbers and symbols, as they are not lowercase.

The **\U** set matches any non-uppercase character.




The pattern replaces all non-uppercase characters with a zero. This includes numbers and symbols, as they are not uppercase.

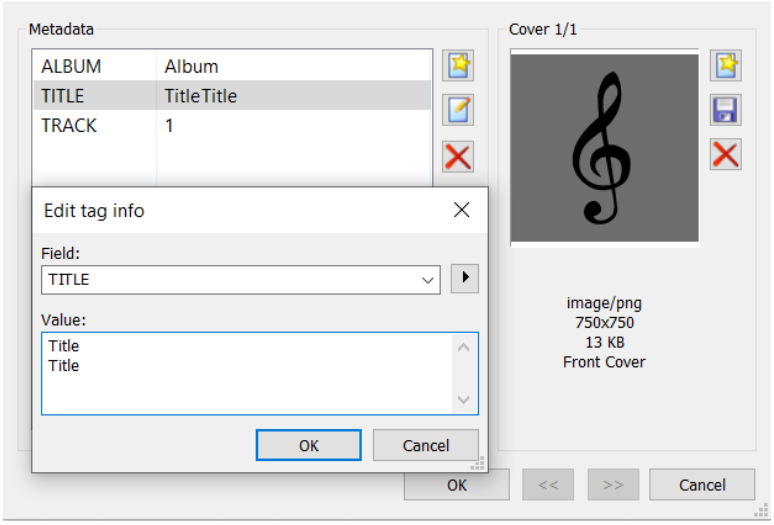
The **\t** set matches the tab character.




Though the field doesn't display it unless highlighted, there is a tab between the period and the first character. The pattern replaces the tab with a space.

A note about the following three expressions: they refer to three different control characters that perform almost the exact same operation, which is moving text onto a new line. While not typically visible or used in most MP3 tags, new lines frequently appear in text files to tag conversions. Differentiating between them is not often required, so in most cases you can use either one with no issues.

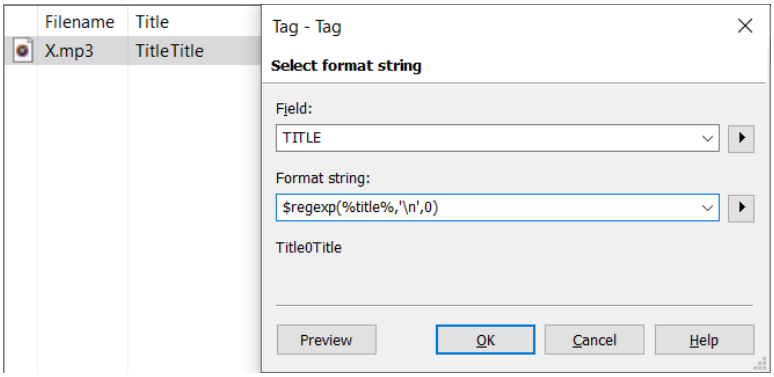
Newline/carriage return/line break characters are not added properly unless set via the "Edit tag info" dialog box, available through the "Extended Tags…" option. While they don't display in the field, they are still present.




The "Edit tag info" dialog box in the "Extended Tags…" option. Note that the two words are on different lines, meaning that there is a break between them.

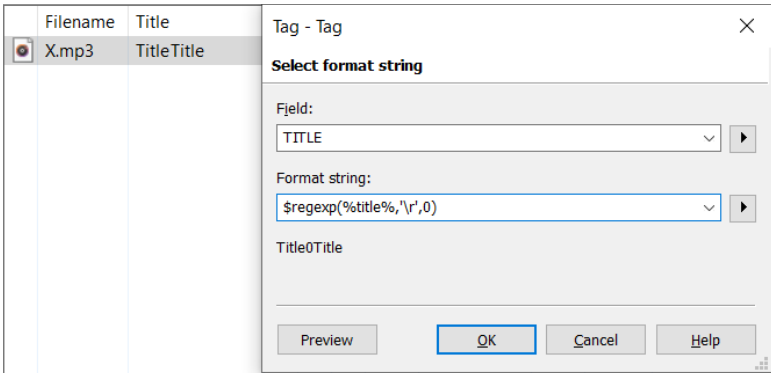
So while they aren't always visible, each of these three characters can be entered and manipulated via regular expressions. Any of the following sets will usually match with the basic line break character that you create with the **enter** key.

The **\n** set matches the newline character.



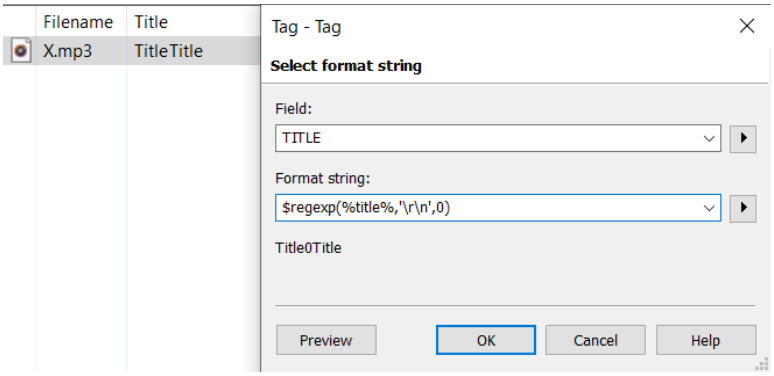

The newline character between the two "title" words is replaced with a zero.

The **\r** set matches the carriage return character.



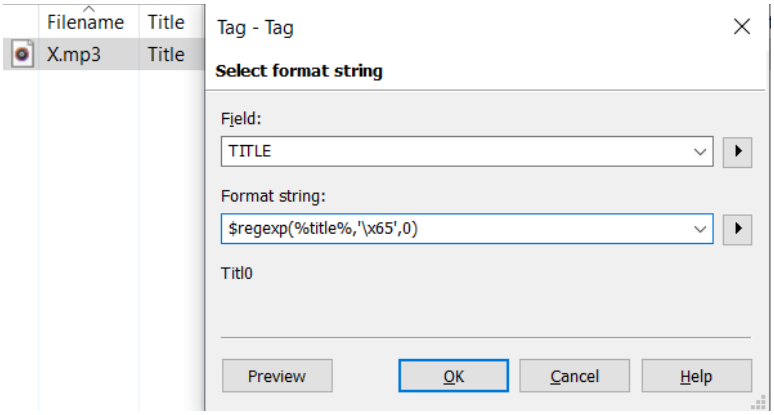

The carriage return character between the two "title" words is replaced with a zero.

The **\r\n** set matches a Windows style line break.



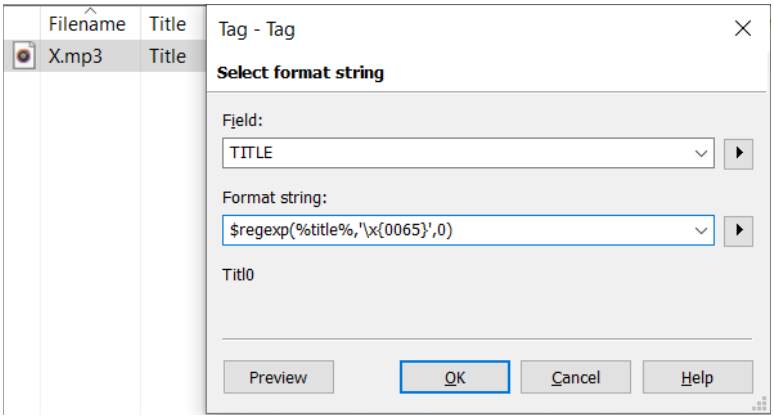

The Windows style line break character between the two “title” words is replaced with a zero.

The **\xnn** set matches the character with the Unicode hex value **nn**. For a list of Unicode hex values, see this list.




The pattern matches the Unicode hex value **65**, which corresponds to lowercase “e.” Thus, the lowercase “e” is replaced with a zero.

The **\x{nnnn}** set matches the character with the Unicode hex value **nnnn**. The difference between this expression and the previous set is that the two-character expression above uses UTF-8 values and the four-character expression uses Unicode code points. The list contains both options for each character.

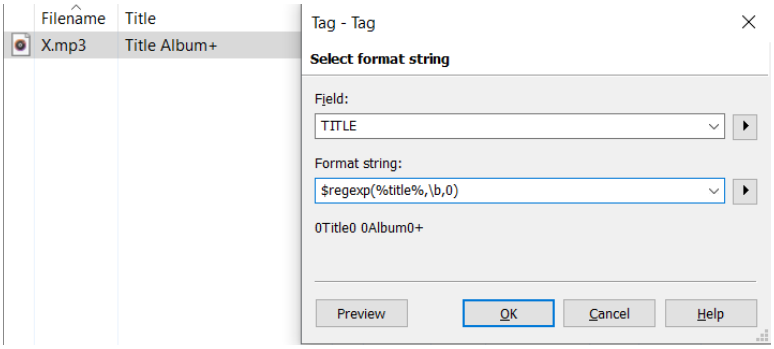



Same as before, the pattern matches the hex value of lowercase “e.” In this case, we use the Unicode code point version of the hex value, which is **0065**.

Word Boundaries

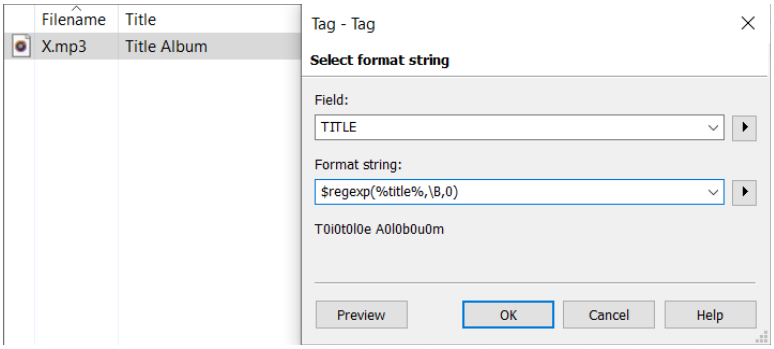
A word boundary occurs at the beginning/end of a string (assuming that the string starts with/ends with a word character) and between word and non-word characters. A word character is defined as any letter or number, plus underscores.

A lowercase b with a backslash (**\b**) matches a word boundary.




The zero appears at each word boundary: the beginning of the string, surrounding each space (as a space is a non-word character), and in front of the plus. Because the plus is a non-word character, the zero does not appear at the end of the string.

An uppercase B with a backslash (**\B**) matches only when not at a word boundary.




The zero appears at every place that is not a word boundary: between each word character, and not at the beginning or end.