*Start a new Python project folder … do not reuse the previous workspace … As with any keyboard-driven console-like environment, developing muscle -memory for the common commands is also part of the learning curve.*

***Regular expression***, or regex for short, is a “text string used for describing a certain search pattern.” This is a supercharged version of wildcards for searching.

For example, to search for the presence of ‘nlp’ in a string “I love nlp programming,” a regular expression can be used.

‘[j-q]’ this regular expression means to search for all single characters between ‘j’ and ‘q’ in whatever text we’re looking at. But this will search all characters between ‘j’ and ‘q’, and not just ‘n’, ‘l’, and ‘p’. Also, this expression will only return single character at a time.

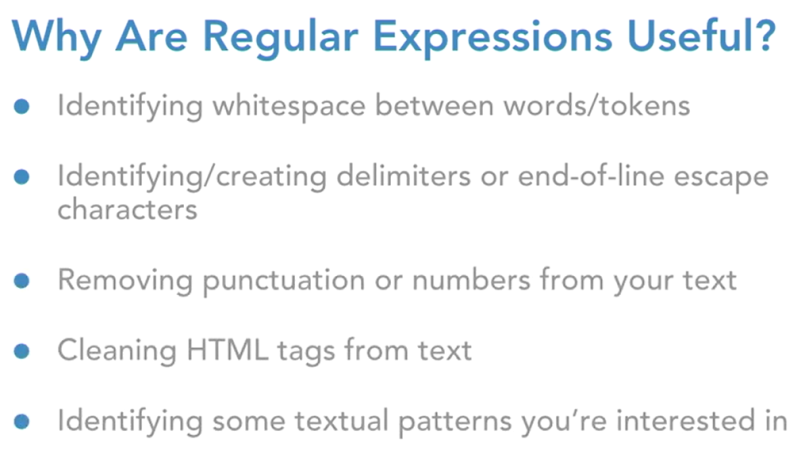
‘[j-q]+’ by adding a plus sign at the end of the expression, we are telling Python to search for strings longer than one character.

‘[0-9]+’ this one will return all numbers with the flexibility of returning sequences of more than one number. So, if there’s a year, like 2019, it will return the full year, rather than each number individually. And, combining these two concepts ...

‘[j-q0-9]+’ this will search for sequences of characters between ‘j’ and ‘q’, or numbers between 0 and 9. So, if you have a course name that was “nlp2019” without any spaces, then it would return that full string. But if you had “nlp 2019” with a space in between them, then that would return them as two separate sequences.

There is literally an infinite number of patterns that you could come up with. And regex’s give you the power and flexibility to search for almost any kind of pattern you could imagine.

Why are regular expression useful? They are particular useful when dealing with text data because a lot of the data is unstructured, where you need to be able to use these patterns to try to create some structure within the document. Here are some of its usage ...

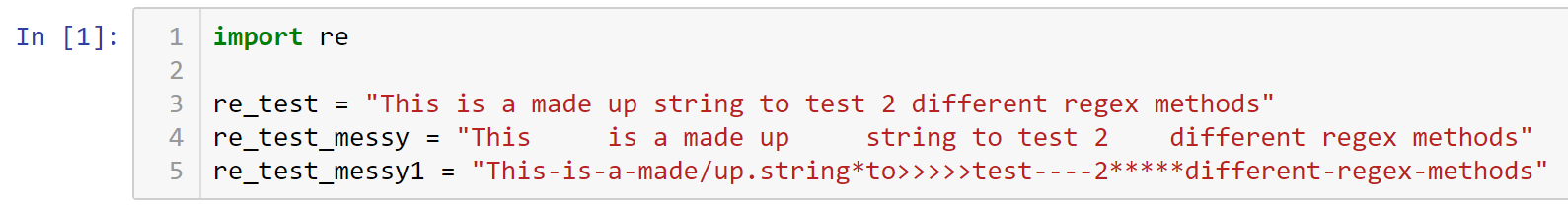


Here are some concrete examples …

* Confirming passwords meet criteria
* Searching URL for some substring
* Searching for files on your computer
* Document scraping

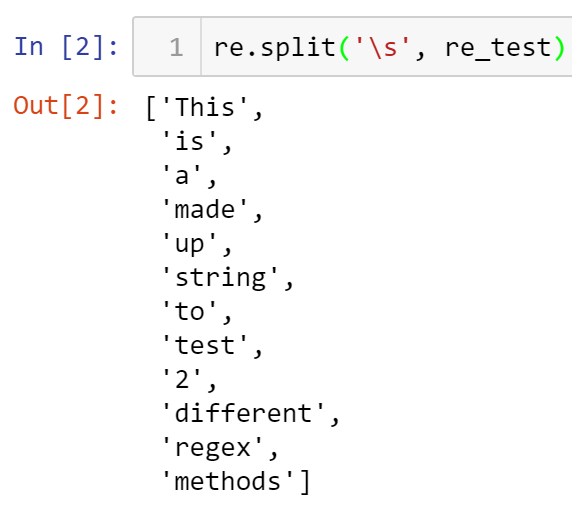
Let’s put regex to use, the low-hanging fruit here and the primary reason that we’re talking about regexes is in order to tokenize sentences or split a sentence into a list of words so that Python can understand what it needs to be looking at.

Loading regular expression, re, and creating real-world text scenarios that you will encounter in the real-world. The first is a clean text, the second is messy, and the third is more messy.

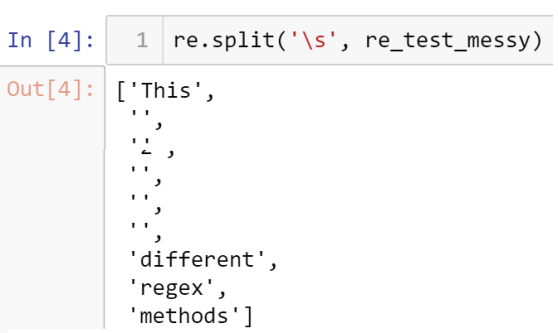


Splitting a sentence into a list of words …

Because the first test example is clean and the words are only separated by one space, a regular **split()** function can be used. ‘\s’ parameter tells Python to look for space as the delimiter between words.

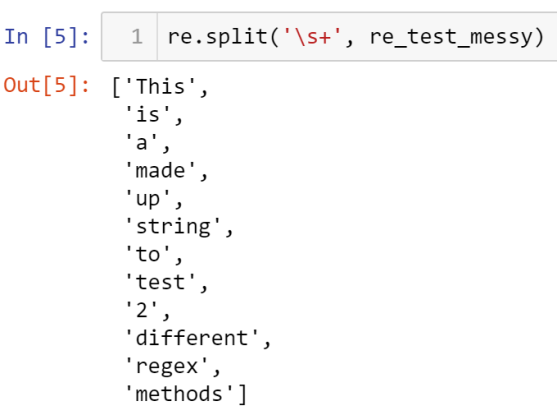


Using the same technique on the messy string …



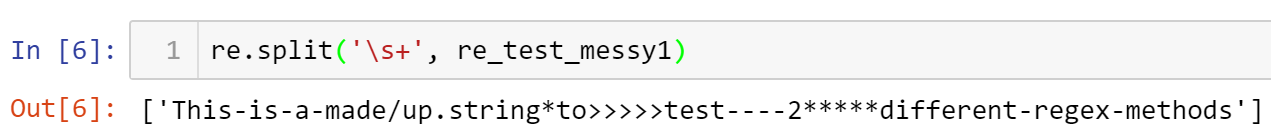
You can see that it is not effective because the white spaces is inconsistent.

To remedy this problem, add a plus sign to the delimiter to tell Python that the space delimiter maybe made up of more than one space.



You can see that the new expression handle the messy string pretty well.

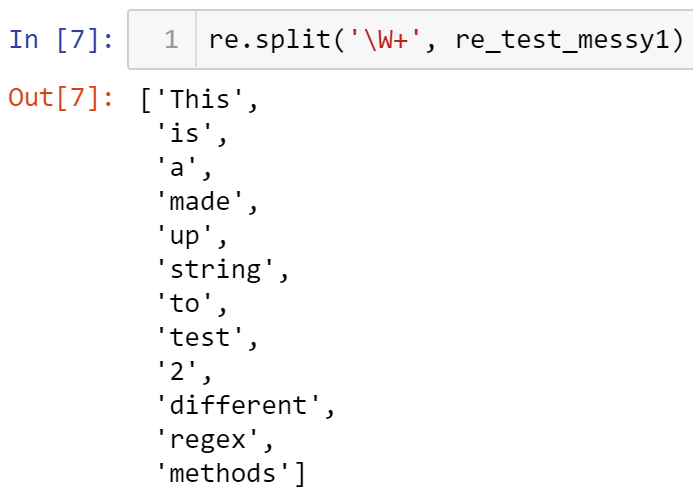
The real test will be on the last string where too many noises are added as delimiters.



You can see that it doesn’t split it at all. It just chop at the first sight of space.

The simple regex ‘\s+’ is not robust enough to handle very complex string like messy1 scenario.

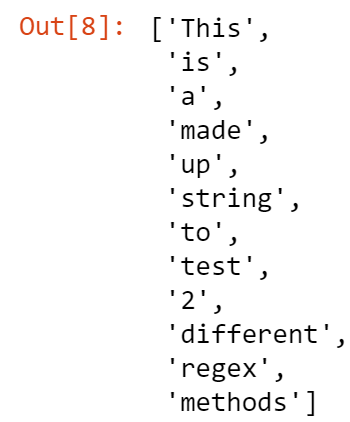
To remedy the problem, use the non-word character ‘\W+’. Also, add the plus sign because we know that some non-word characters were repeated. In other words, the plus symbol will allow multiple special characters in a row



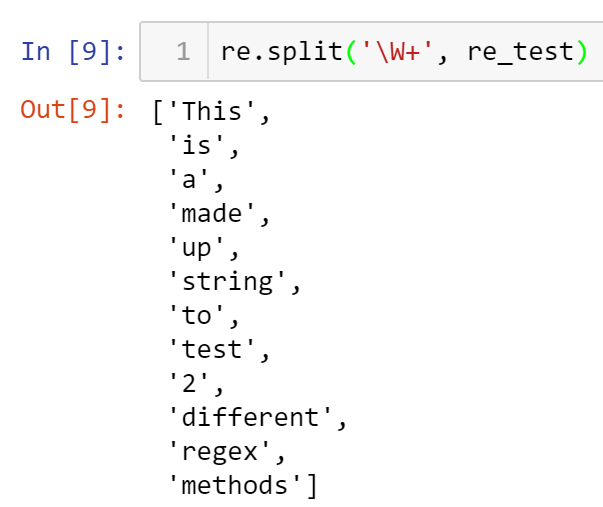
Now you see that the new non-word expression works well with this most complicated text version. Let’s verify that non-word works with the other string examples.

With re\_test\_messy string ...



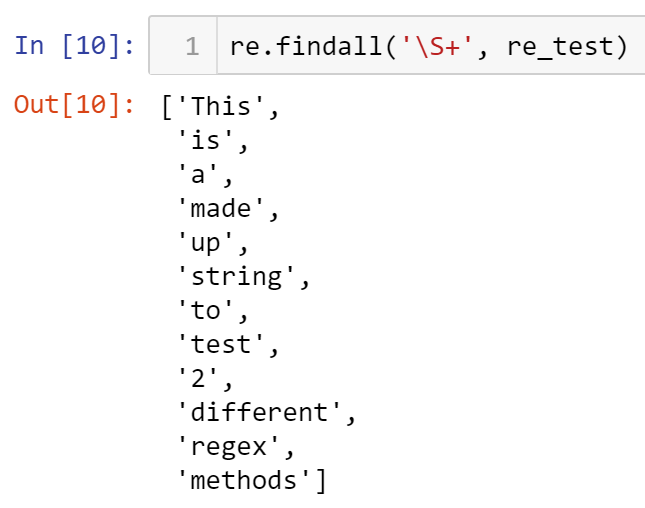


With re\_test string ...



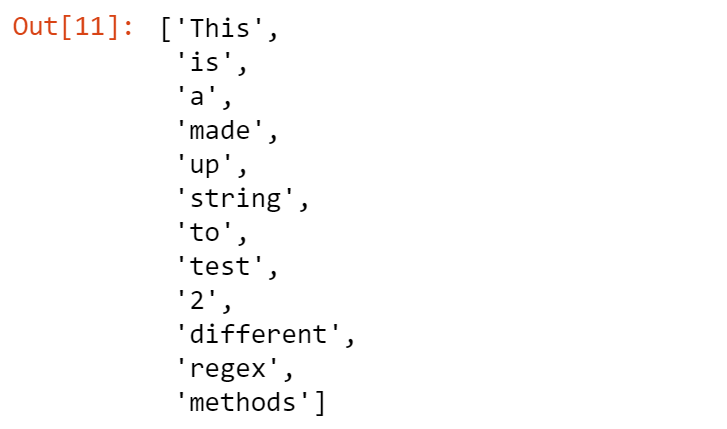
As you can see it works for simple, complex, and more complex strings.

You can also use the **findall()** method to search for a word. It has the same syntax as the split() method. The capital S parameter will find for non-whitespace character, and adding a plus sign means one or more non-whitespace characters. Starting with the most simple string ...

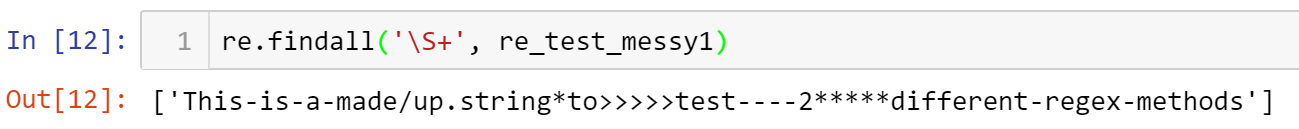


Trying it on messy string …

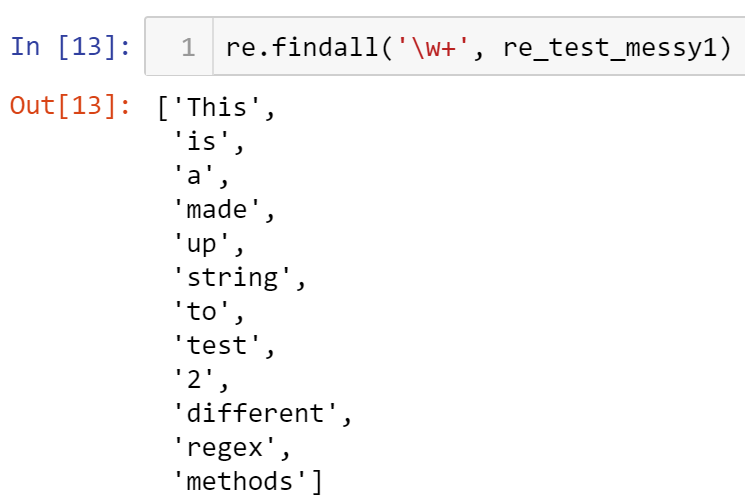




And, then to the most complex string …



Oops, it is not handling well with the most complex example because the expression is just looking for non-whitespace character. Therefore, we need a more flexible regex. By using the opposite of capital W (which looks for non-word), small w will look for a word character, and adding a plus sign means one or more word characters.



Now, you can see the new regex works. It search for a word and ignores special characters.

**Takeaways:**  Two useful methods for tokenizing

findall() - will search for the actual words while ignoring the things that separate the words

split() - will search the characters that split the words while ignoring the actual words themselves

‘\W’ & ‘\w’ - are based on words

‘\S’ & ‘\s’ - are based on whitespaces

* All submissions should be separate from other exercises and quests. Please do not lump all your answers into one document and re-using that same workspace to gain multiple points. Thanks.
* Place your name at the bottom of your code, download your Python program in html format, and submit your work in Canvas.