TECHNOLOGY NEWS

REGULAR EXPRESSION IN PYTHON

Winfred Lam, CFA

Share this article

In the previous edition of *The Analyst,* we provided a broad overview of common data types in Python. In this edition, we focus on the data type "string”. More specifically, we look at how to extract, search, and manipulate strings using Regular Expression (regex), a powerful approach to match patterns when working with strings.

A regex is a sequence of characters that describe a pattern, and is often used in web scraping, data wrangling, speech recognition and natural language processing. Effective use of regex can save analysts, data scientists and programmers a lot of time.

This article provides a high-level introduction to regex in Python and is broadly split into two sections:

1. Regex Overview and Rules

2. Examples

Although this article demonstrates the use of regex through Python, the core concepts of regex are not unique to Python. Mastering regex is extremely worthwhile as it is implemented in many other programming languages.

Regex Overview

Suppose we are reading a string (text) with only numbers and commas, and we want to isolate all the numbers. To keep it simple, let this string be '123,45'. Therefore, the answers we are looking for are '123' and '45', as we have two numbers separated by a comma ",". We can use Python's string.split() method.

```
[ ] text = "123,45"
    text.split(",")

    ['123', '45']
```

The "," argument fed to the split method is the delimiter, splitting the text whenever a comma is encountered. Unfortunately, in the real world, data is often unstructured and not laid out in such a digestible manner. As text gets more complicated, it becomes more difficult to parse the same data successfully.

Consider the new text below:

```
[ ] text = "123,,45"
    text.split(",")

    ['123', '', '45']
```

The same numbers are embedded in the text, but ***text.split(",")*** could no longer successfully isolate the two numbers. Instead, it also included an empty string because there is nothing between the two commas.

The program needs to recognize not just the comma, but a pattern, namely “one or more commas". Consider below a new attempt, which uses the **.split()** method of Python's built-in Regular Expression package, "**re**". It can be imported via "**import re**".

```
[ ]  import re

     text = "123,,45"

     regex = "[,]+"

     re.split(regex,text)

     ['123', '45']
```

The square brackets "[]" represents a set, and belongs to a class of special symbols called metacharacters, which have special meanings (more on this below). The plus sign “+” that follows is also a metacharacter and is the logical equivalent of "one or more". As such, this regex means "one or more commas", and when passed as an argument to the split method, will now be able to correctly split all numbers separated by commas.

This is the power of regex—they form patterns rather than fixed characters. We can now split the numbers no matter how many commas there are:

```
[ ]  text = "123,,,,,45,30,,,2000"

     regex = "[,]+"

     re.split(regex,text)

     ['123', '45', '30', '2000']
```

What if the numbers are no longer separated by just commas? We can generalize the regex further to handle more complex text. For now, let us go through some of the basic rules:

Metacharacters

As mentioned above, metacharacters are interpreted in specific ways by a regex. They can be thought of as the grammatical rules of regex and give a regex power. Below is a list of key metacharacters, some of which are present in our regex "[,]+" above:

- Square brackets [ ] : specifies a set of characters
- Period . : matches any single character (line break "\n" is an exception)
- Caret ^ : detects if the string starts with a certain character; denotes **complement** when used inside a bracket
- Dollar sign $: detects if the string ends with a certain character
- Plus sign + : matches one or more occurrences of the pattern **to its left**
- Asterisk * : matches zero or more occurrences of the pattern **to its left**
- Question mark ? : matches zero or **one** occurrence of the pattern **to its left**
- Curly braces {x, y} : at least x and at most y occurrences of the pattern **to its left**
- Parentheses ( ) : helps group sub-patterns
- Vertical bar | : indicates the "OR" operator
- Backslash \ : "untoggles" the above metacharacters, so they can be recognized as normal characters

Special Sequences

In addition to metacharacters, "special sequences" also take on their own meaning. The list below provides the sequences and their corresponding matches:

- \A: matches if the specified characters are at the start of a string
- \Z: matches if the specified characters are at the end of a string
- \b: matches if the specified characters are at the beginning or end of a word
- \B: matches if the specified characters are **not** at the beginning or end of a word
- \d: matches any decimal digit between 0 and 9
- \D: matches any non-digit
- \s: matches "whitespace characters" like space, tab, or line break
- \S: matches non-whitespace characters
- \w: matches any alphanumeric character (i.e., a–z, A–Z, 0–9)
- \W: matches any non-alphanumeric character

We will now demonstrate some of these metacharacters and special sequences with more examples. There are too many to cover them all in detail in this article, so the above is best referenced as a “glossary” to use when writing your own regex.

We revisit the first example of isolating numbers in a text. Consider now a string of numbers separated not just by commas, but also by exclamation marks. We will use a new regex:

```
[ ]  text = "123,,!!,45,30!2000"

     regex = "[,!]+"

     re.split(regex,text)

     ['123', '45', '30', '2000']
```

By adding the exclamation mark ! to the set, the new regex now means "one or more commas or exclamation marks". However, because we limited ourselves by specifying "comma" or "exclamation mark", those are the only two symbols it would be able to recognize. This means the regex will struggle with the below text:

```
[ ]  text = "123,,!!#        $,45,((((30!//;2000"

     regex = "[,!]+"

     re.split(regex,text)

     ['123', '#        $', '45', '((((30', '//;2000']
```

We need an even more powerful regex. We can broaden the regex more to handle "all non-numeric characters". A naive approach is to expand the above regex to include all symbols, namely "[,!.%...]+" .etc until we include all non-numeric characters. But that would be inefficient, and the regex could fail if we miss even one non-digit character. Instead, we can use the caret “^” metacharacter.

“[^A]” means the “complement of A”, or “anything that is not A”. Note that this needs to be used inside a pair of square brackets, as ^ has a different meaning when used in isolation, as described in the “Metacharacters” section above.

In this case, the regex is as simple as "[^0-9]+", which means "one or more of anything that is not between 0 and 9". Hence, splitting the text with the regex is equivalent to saying, "split all numbers by any non-numeric character".

```
[ ]  text = "123,,!!#        $,45,((((30!//;2000"

     regex = "[^0-9]+"

     re.split(regex,text)

     ['123', '45', '30', '2000']
```

The regex "[^0-9]+" would work for any alphabets too, because they are also "not in the range 0–9". That is the power of "complement". As such, it will have no problem splitting the numbers in the below text:

```
[ ]  text = "123 CFA exam candidates, pass rate - 45%! 30 candidates set to retake, total candidates worldwide: 2000"

     regex = "[^0-9]+"

     re.split(regex,text)

     ['123', '45', '30', '2000']
```

We can make this even more robust by having the regex account for decimals. For example, say we change the pass rate to 45.5%. The regex would fail because "." is a non-numeric character, and it would be used to split 45 and 5 from 45.5.

```
[ ]  text = "123 CFA exam candidates, pass rate - 45.5%! 30 candidates set to retake, total candidates worldwide: 2000"

     regex = "[^0-9]+"

     re.split(regex,text)

     ['123', '45', '5', '30', '2000']
```

Instead, we can make a small adjustment to the regex by adding a "." after “0-9”. The regex now means "one or more of anything that is not numeric and not a period", and will split the text accordingly.

```
[ ]  text = "123 candidates signed up for the CFA Exam, pass rate was 45.5%! 30 candidates are set to retake, total candidates worldwide: 2000"

     regex = "[^0-9.]+"

     re.split(regex,text)

     ['123', '45.5', '30', '2000']
```

It should be emphasized that the above logic can be written in **many** different ways (i.e., "[^\d.]+"). Regex is its own language and can match just about any pattern the user defines. However, writing a foolproof regex requires a deep understanding of the metacharacters and how they work with one another, as any changes in logic can immediately create new problems. (For example, adding the period to the regex “[^0-9.]+” solved the decimal problem but also introduced a new one. Can you spot what it is? Hint: Notice we purposely did not include a single isolated period in the text.)

More Examples

Python's built-in "re" supports various regex methods. So far we have only discussed the **re.split()** method. Some of the other common methods are:

- re.search(pattern, string): finds the **first occurrence** of a match and returns a match object. Returns none if no match is found.

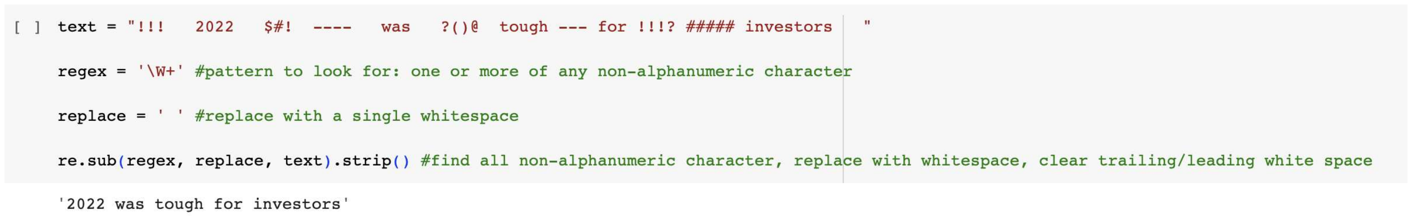
- re.match(pattern, string): matches a pattern **at the beginning** of the string.
- re.sub(pattern, replace, string): searches for a pattern and replaces it with another.
- re.findall(pattern, string): finds all matches in a string, and then returns them as a **list of strings**.

The next section demonstrates how to use **re.sub()** to do a simple find and replace, as well as **re.findall()**, which is great for extraction.

Cleaning up with Regex

Consider text where words and numbers are separated by unpredictable whitespaces and symbols. We want to replace them with single whitespaces, but we are unsure what the unwanted sequences look like. This makes it impossible to use simple “Find and Replace” functions in, say, Microsoft Office.

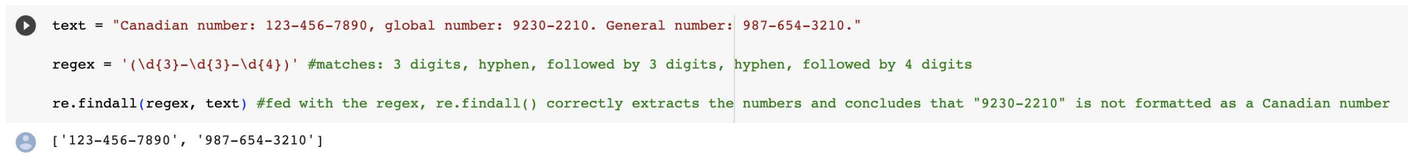
With **re.sub()**, we can "generalize" the pattern in a regex, replace it with something we want, and them finish with **string.strip()** to clear out leading/trailing whitespaces:



```
[ ] text = "!!!   2022   $#!  ----   was   ?()@  tough --- for !!!? ##### investors    "

    regex = '\W+' #pattern to look for: one or more of any non-alphanumeric character

    replace = ' ' #replace with a single whitespace

    re.sub(regex, replace, text).strip() #find all non-alphanumeric character, replace with whitespace, clear trailing/leading white space

    '2022 was tough for investors'
```

Phone Number Search

Suppose we want to search for phone numbers in a text and are only interested in phone numbers that follow the Canadian format of ddd-ddd-dddd (where “d” can be any digit from 0 to 9). We can define a regex pattern accordingly.



```
text = "Canadian number: 123-456-7890, global number: 9230-2210. General number: 987-654-3210."

regex = '(\d{3}-\d{3}-\d{4})' #matches: 3 digits, hyphen, followed by 3 digits, hyphen, followed by 4 digits

re.findall(regex, text) #fed with the regex, re.findall() correctly extracts the numbers and concludes that "9230-2210" is not formatted as a Canadian number

['123-456-7890', '987-654-3210']
```

Match Dates

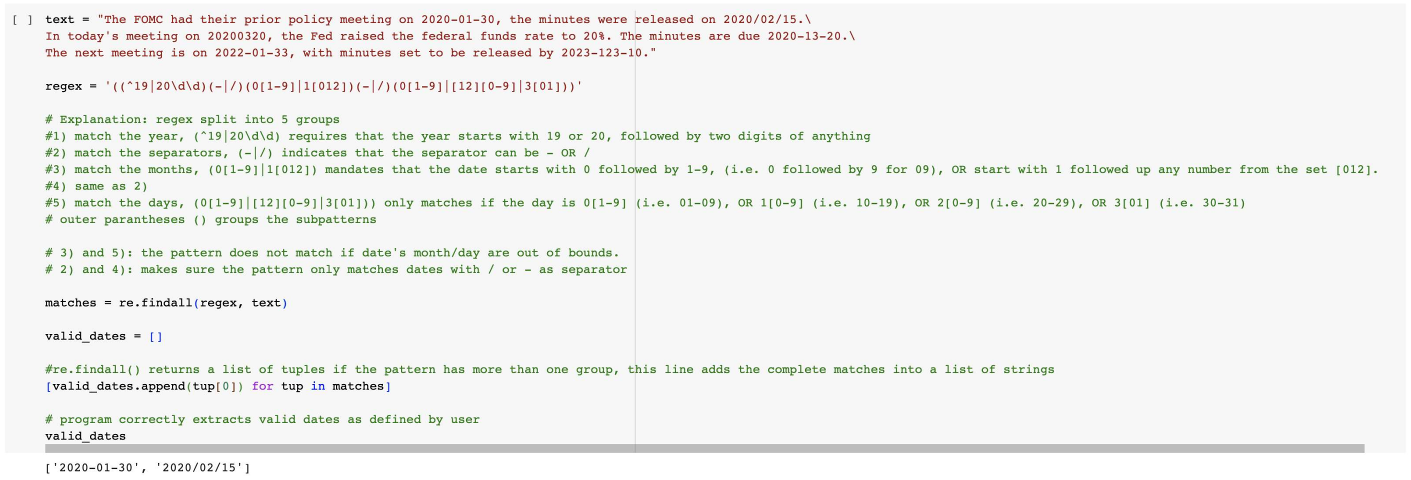
Matching dates follows a similar idea. Let's say we have more stringent requirements and only want to search for valid dates in the format yyyy-mm-dd or yyyy/mm/dd. By valid, we not only want dates that adhere to these formats, we also want to make sure that the dates are actually possible:

1. The month is two digits, not 00, and not greater than 12
2. The day is not 00 and not greater than 31
3. The year is recent: starts with 19 or 20

For example, even though the below are in the format yyyy mm dd, none of them are valid dates by our definition:

20200320, 2020-13-20, 2022-01-33, 2023-123-10

The regex can get quite complicated, as it has to account for all the above requirements.

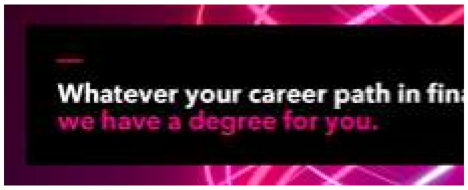


```
[ ] text = "The FOMC had their prior policy meeting on 2020-01-30, the minutes were released on 2020/02/15.\
    In today's meeting on 20200320, the Fed raised the federal funds rate to 20%. The minutes are due 2020-13-20.\
    The next meeting is on 2022-01-33, with minutes set to be released by 2023-123-10."

    regex = '((^19|20\d\d)(-|/)(0[1-9]|1[012])(-|/)(0[1-9]|[12][0-9]|3[01]))'

    # Explanation: regex split into 5 groups
    #1) match the year, (^19|20\d\d) requires that the year starts with 19 or 20, followed by two digits of anything
    #2) match the separators, (-|/) indicates that the separator can be - OR /
    #3) match the months, (0[1-9]|1[012]) mandates that the date starts with 0 followed by 1-9, (i.e. 0 followed by 9 for 09), OR start with 1 followed up any number from the set [012].
    #4) same as 2)
    #5) match the days, (0[1-9]|[12][0-9]|3[01])) only matches if the day is 0[1-9] (i.e. 01-09), OR 1[0-9] (i.e. 10-19), OR 2[0-9] (i.e. 20-29), OR 3[01] (i.e. 30-31)
    # outer parantheses () groups the subpatterns

    # 3) and 5): the pattern does not match if date's month/day are out of bounds.
    # 2) and 4): makes sure the pattern only matches dates with / or - as separator

    matches = re.findall(regex, text)

    valid_dates = []

    #re.findall() returns a list of tuples if the pattern has more than one group, this line adds the complete matches into a list of strings
    [valid_dates.append(tup[0]) for tup in matches]

    # program correctly extracts valid dates as defined by user
    valid_dates

    ['2020-01-30', '2020/02/15']
```

Conclusion

It takes substantial practice to develop foolproof regexes consistently, and there are often multiple ways to define a pattern. Fortunately, they are all products of standard and well-documented rules, and there are excellent tools, such as www.regex101.com, that provide live testing and explanation of regexes. After all, writing a regex is just orchestrating a sequence made up of 1) metacharacters/special sequences, which have special powers, and 2) everything else. Regex can narrow down or generalize textual searches and can be an invaluable addition to an analyst or programmer's toolkit.

Winfred Lam, CFA, is a Senior Product Manager at BMO Global Asset Management and a graduate student in computer science at the University of Pennsylvania. He is a volunteer member of CFA Society Toronto's Member Communication Committee.



Whatever your career path in finance, we have a degree for you.

Full-Time MBA

Morning or Evening MBA

Master of Finance

GMAT/GRE requirements are waived for CFA Level II and above

Rotman Here's where it changes.