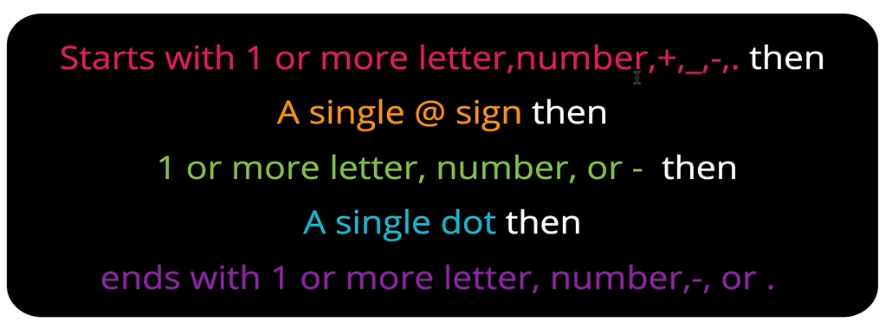
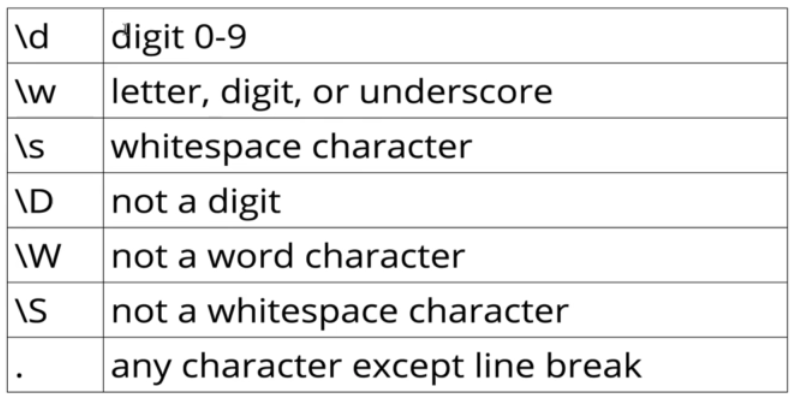
* **Regular expressions** are a way of describing patterns within search strings
  + They are NOT specific to Python, but Python does have an implementation for writing and using regular expressions
  + Basically, a fancy set of characters that you can string together to make patterns or templates
  + In Python, you can then test whether a string matches a RegEx pattern that you want to find
* Example: validating emails – let’s say we want to check that an email address has a valid format
  + General rules for email addresses
    - Emails generally take the form of [stuff@gmail.com](mailto:stuff@gmail.com)
    - There is only one @ symbol, and it cannot be at the beginning or the end of the email address
    - We can also have multiple dots (periods) within the email addresses
    - We can also have dashes and numbers for the email addresses
    - We can also have plus signs and underscores, but ONLY before the @ sign
  + Here is what it boils down to:



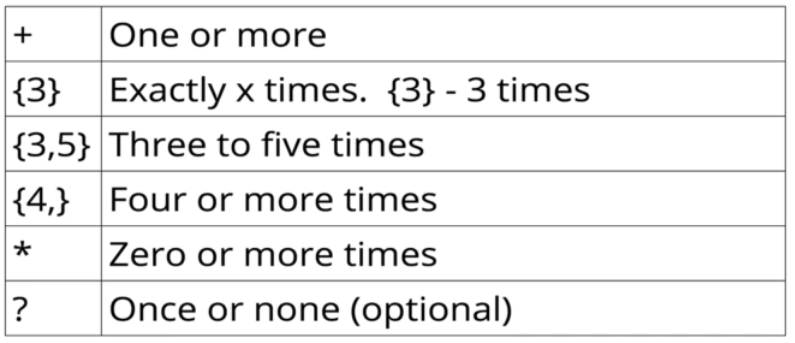
* + - To check the validity of email addresses given what we know, it would be a TON of logic to implement in Pytthon
    - Instead, we can use the logic of regular expressions
  + Here’s a regular expression that would validate emails!



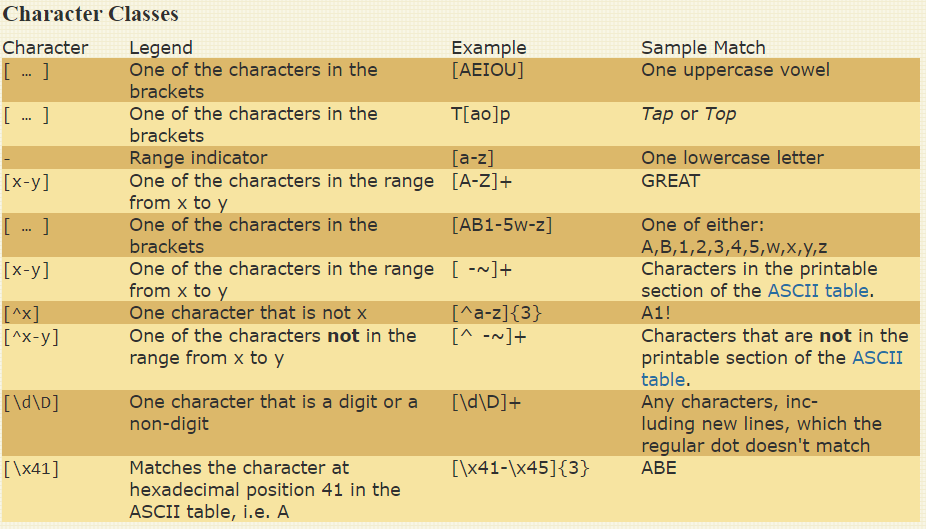
* + - The first section in pink stands are “you can have any lowercase letter, any uppercase letter, any digit 0 through 9, and underscore, period (dot), plus sign, or dash, and you can have one or more of them”
      * The carrot (^) indicates that it must start with this
    - The @ sign in yellow means that the next character needs to be an “@” sign
    - The green section indicates you can have next have “one or more letters, numbers, or dashes”
    - The blue section indicates that you must have a single dot. The “\” is an escape character to indicate that you literally want a period
      * An actual dot (.) is a special character in regular expressions, so an escape is needed
    - The final purple section indicates that you can end with one or more letters, numbers, a dash, or a period. The dollar sign means that the expression must end with this section
* Potential Use Cases for RegEx
  + Credit card number validation
  + Phone number validation
  + Advanced find/replace in text
    - If you want to find repeated words in a body of text, and potentially replace them
  + Formatting text/output
  + Syntax highlighting
    - Coding software typically highlights text based on properties, presumably by following rules grounded in regular expressions
    - Example: If a group of characters sits between quotation marks, make all those characters one color
* Regex Links
  + Rex Egg RegEx cheat sheet:<http://www.rexegg.com/regex-quickstart.html>
  + Regex Python RegEX editor: <https://pythex.org/>
  + Python re documentation: https://docs.python.org/3/howto/regex.html
* Some useful RegEx characters



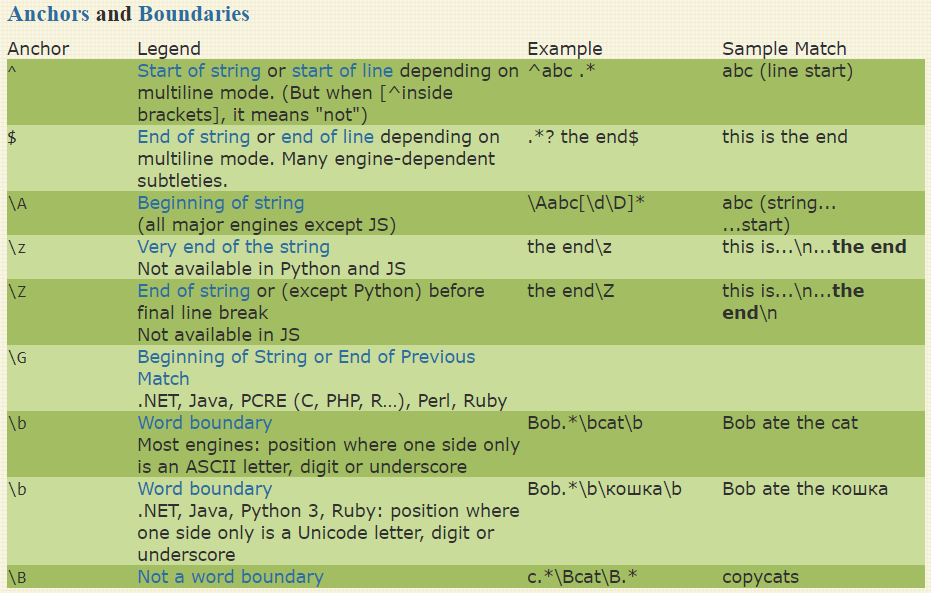
* Useful RegEx Quantifiers, which specify how many times something should occur in a match pattern



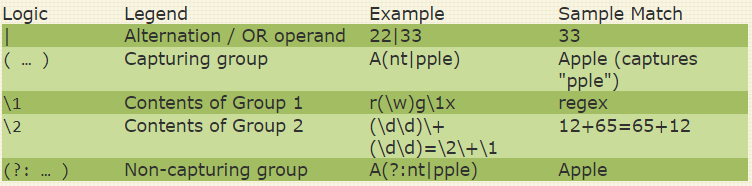
* Another very useful aspect of regular expressions is character classes
  + The brackets ([]) are especially useful, as they allow you to match any of the characters within the brackets
  + Additionally, you can use quantifiers (e.g. {3}) after the brackets to indicate how many times you
  + Within the context of brackets, the carrot character (**^**) is a **negation**
    - When placed within a bracket, the negation applies to all characters in the bracket
    - This is different from the carrot character at the beginning of a search string, where it indicates “starts with”. This is an anchor, and we’ll see more on this below.



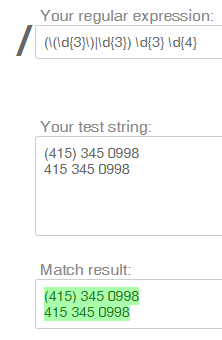
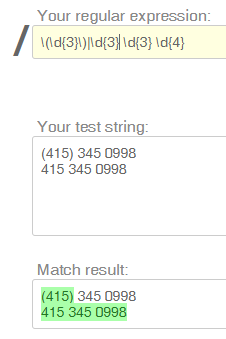
* Ranges can be placed with brackets to indicate a search for any value within that range
  + a-z: any lowercase letter
  + A-Z: any uppercase letter
  + 1-9: any digit 1 through 9
  + Ranges can also include any subset of the above
* Anchors and boundaries are useful in allowing you define how your string start and ends
  + The most useful are the **carrot** (^) and the **dollar sign** ($)
  + The word boundary character \b is also useful
  + For instance, oftentimes you want to ensure that the string you are searching for is only that string and nothing else; that is, it is not buried within other strings



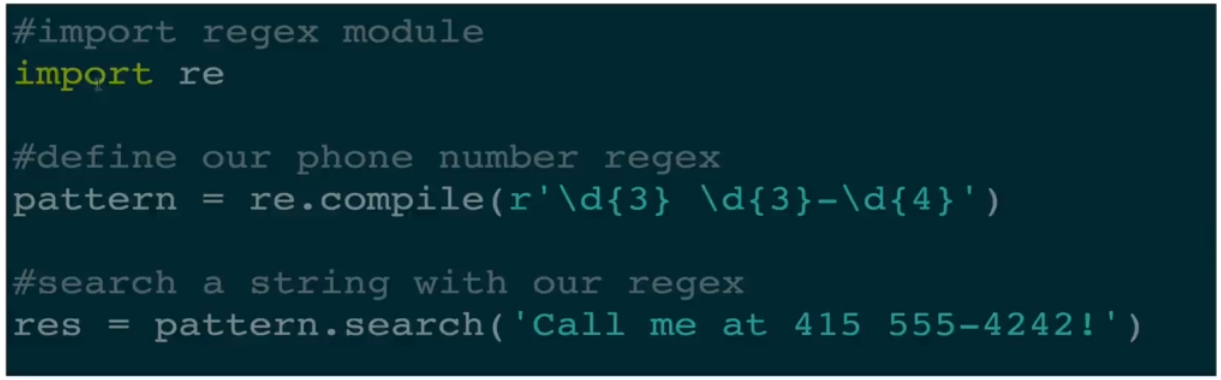
* The logical OR character, the **pipe** (|)



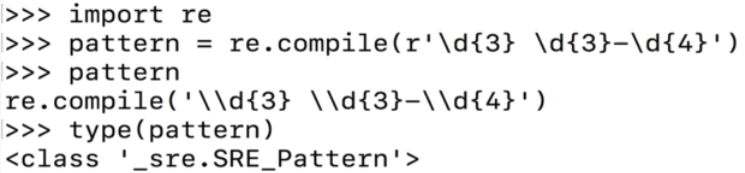
* + By default, the pipe character will match either everything on the left OR everything on the right. If you want your regex search to apply only to a subset of your expression, place standard parentheses around that subset

vs 

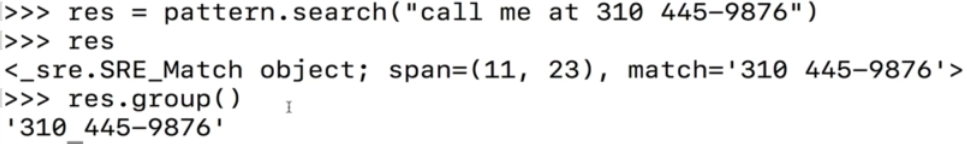
* + Generally speaking, the regular parentheses are used for grouping regex expressions. When working with regex in Python, regex search results will often be returned by this grouping
* The **re** module is a built-in python module for using regular expressions
  + <https://docs.python.org/3/library/re.html>
  + Allows you to do very useful things, like write functions that return a particular value if a regular expression is found
  + Is chock full of useful methods
  + This is a common pattern for using the **re** module



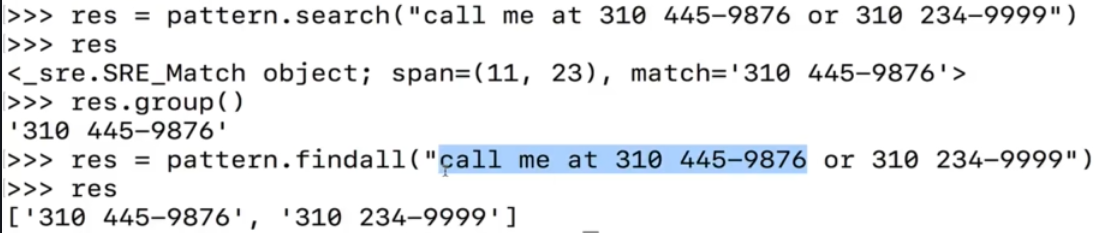
* + - Start by importing *re*
    - Then you create a regular expression object and compile it by passing it into **re.compile**, which creates the pattern object. We usually save this to a pattern variable name (“pattern” in this example)
      * The “r” stands for “raw”, which tells Python that we want to use regular backslashes and we don’t have to escape them



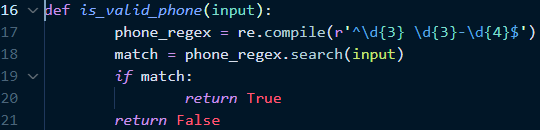
* + - Then we can call **pattern.search()** on a string that we want to search. We usually save that as a result variable (res [for result] in this example).
      * Note that this search does not return the matched string automatically. Instead, it returns a **match object**
      * Finally, we can extract the match string out of the match using **result\_variable.group()**



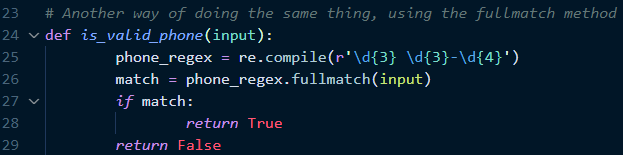
* + - Pattern.search() does not support finding more than *one match*. If you want to search for multiple matches, use **pattern.findall()**. Again we usually save this to a result variable
      * This will return a list of all matched strings. You can access the strings by accessing the list by index



* + Up until now, we have been using regular\_expression\_object.search(), which requires us to compile the regex string beforehand. However we can also use an alternative method called **re.search()**, where we must pass in the regular expression pattern, along with the string to search.
    - re.search() gives the same single match result, and we still need to use result\_variable.group() to get the actual string
    - If you plan to use a regular expression more than once, it is best to compile it first and use the regular\_expression\_object.search() approach
  + There is also another method called **regex\_object.fullmatch()**, which will return a match if and only if the entire input string is a match. This is an alternative to placing beginning of string and end of string characters in your regex pattern. Compare the two blocks of code below:
    - Option 1: Using anchors

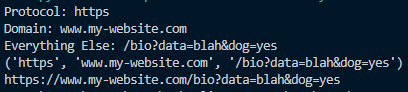


* + - Option 2: Using fullmatch()

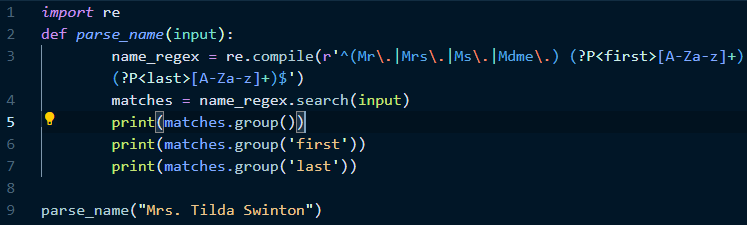


* Recall that groupings of regular expressions can be created by using parentheses around groups of regex terms within your search pattern
  + When you then use the **pattern.search()**, groups will be created based on the parentheses that were included in the regex search pattern.
  + You can access these different groups by calling **matched\_pattern.goup(group\_number)**
    - .group(0) gives you the entire match. It’s the same as calling match.group() with no number
    - .group(1) gives you the first group
    - .group(2) gives you the second group and so forth
  + Calling **matched\_pattern.groups()** (notice the **s**) will return a tuple with all of the different groups in it



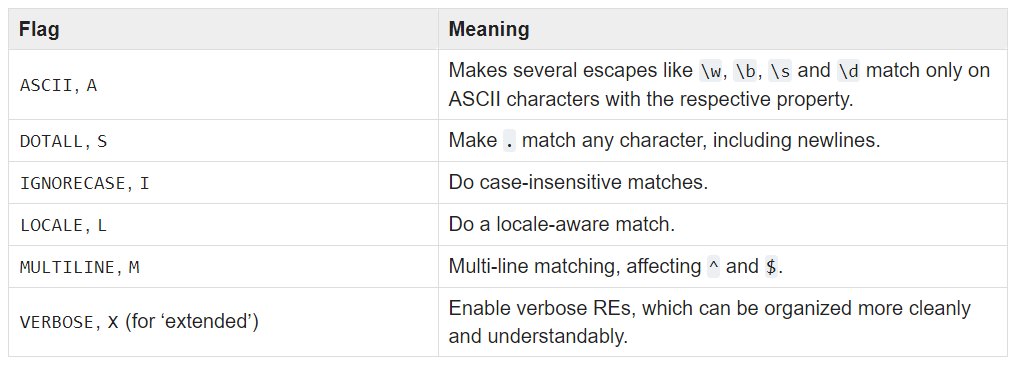


* **Symbolic grouping** of regex expressions can be accomplished using the **?P<group\_name>** syntax.
  + By grouping in this manner, you can then directly call the symbolic name of the group in **matches.group(‘name\_of\_group’)** instead of using the group number.
  + You can only use each group name once
  + This is a bit more intuitive if you can stand the weird syntax





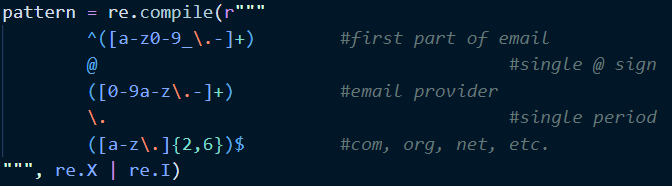
* Regular expression **compilation flags** are preferences that can be set when compiling regular expressions to impact how they work



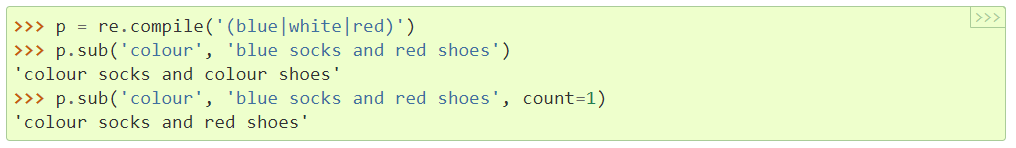
* + Example of the **verbose** flag, which ignores whitespace within the RE string except when a whitespace is in a character class as defined, or is preceded by an unescaped backslash. This allows you to organize the RE pattern to be more readable
    - Here is what a regex for an email may look like without a verbose flag



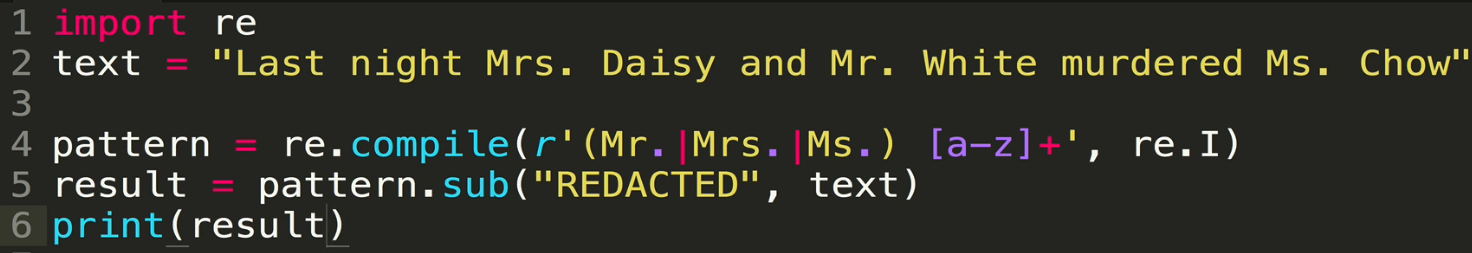
* + - Here is the same regex using the verbose flag. Note that the **re.VERBOSE** (represented here by its shorthand **re.X**) flag is passed in as an argument to the re.compile() method. What’s nice about this approach is that we can add comments to each portion!

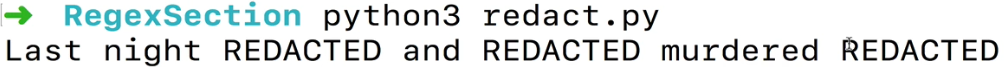


* + The example above also illustrates the **ignore case** flag. Note that in order to use multiple compilation flags, we need to use the bitwise pipe |
    - Don’t get confused since pipe | usually indicates OR. In this case it indicates AND.
* Find and Replace is one of the most powerful and common use cases of regular expression searches
  + The **.sub()** method is used to replace a match with something else. You call it on your compiled regex pattern, passing in as arguments the word you are replacing and the string you are searching for a match of your regex pattern

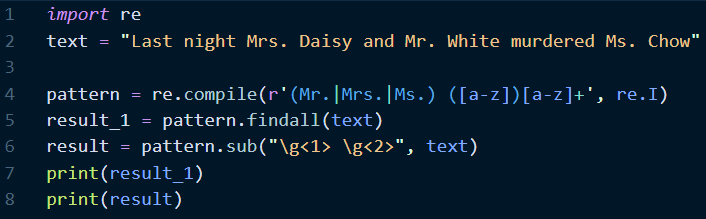


* + Another example of .sub()





* + - Another version where we keep the prefix and the first letter of each name. This utilize the group reference mechanism of the .sub() method, **\g<group\_number>**. Specifically, we will replace each match (e.g. Mr. White) with the first two groups defined by our regex pattern, which are the prefix (Mr./Mrs./Ms., referenced by \g<1> below) and the last initial (e.g. Mr. W, referenced by \g<2> below)
      * This example also illustrates how .findall() works for grouped regular expression patterns. When you create different groups for your regex pattern using parentheses, findall() will return a list of tuples, one tuple for each match that it found. Within each tuple will be the matches for the individual groups in parentheses





* The .sub() method can also be used to swap positions of text within a matched string! This can be very useful, especially if you want to reformat your strings into something that is easier to sort
  + In the example below, a list of book titles with dates is passed in. The code will capture the title and date separately, then create a new list in which the format is changed
    - Original format: Title (date)
      * We would be unable to simply use string.sort() on this because it would sort by title, whereas we want to sort chronologically by year of publication
    - New format: Date – Title
      * We can now sort this chronologically by year of publication



