MP4 – Part 1: Matching

In order to tackle Part 1 of MP4, I knew at first that I wanted to draw on methods used in Part 2 of MP3, specifically pertaining to defining a function whose single argument would accept a file name in the form of a string. I knew this wasn't exactly necessary, but I feel as if there is utility to encapsulating my code in a function, if only so that I can easily apply its operations to other text documents without rewriting the logic of the code itself. Moving on, I chose to define the header for my function as **match_doubles(path: str)**, inside of which I first imported the **re** module and then assigned an empty list to the variable titled **collections**, knowing I would have to append my results to that list toward the tail-end of my function. Drawing again on methodology from MP3, I decided to utilize a *with-statement* in order to simplify the management of my file stream for **wordlist.txt**, which I assigned to the variable named **source**. Using a *for-statement* to pass code to content within **source**, I stripped trailing white space from discrete lines in the file and assigned those objects to the variable **entry**, which I then set to the syntax of **re.search** so that it would fit the argument for re patterns, written accordingly: **'(\w)1+'**. I came to this particular pattern after trying a few other ones on for size inside of a separate module, where I had reproduced a portion of **wordlist.txt** as a string for the sake of a simplified, trial-and-error approach. Compartmentalizing my code in this way has been instrumental to me in the past, so once I knew I had the correct regular expression, I went ahead and included in my function, assigning it to the variable **match**. Following that step, I ran into a problem when appending **match** to **collections**, which meant that when I later called upon the function it wouldn't print words with doubled letters but rather metadata about the desired *re* pattern (e.g. "[<re.Match object; span=(2, 4), match='tt'>"). I resolved this issue by formatting an *if-statement* that pulled the desired results I had intended upon from the start, stipulating that if **match** was found, then its concordant **entry** should be appended to **collections**. I returned **collection**, assigned a variable to the act of calling the function, then conducted my assertion tests by ensuring that the list was 197 entries long, and began and end with 'bottler' and 'volunteers' for good measure.

MP4 – Part 2: Substitution

As with Part 1 of MP4, I decided to work with a function for Part 2, writing its parameters so that re would already be imported in the first line. I then assigned the variable **pattern** to an re method entitled **re.compile**, which I found via online documentation for the module, employed in particular because it would render my code more transparent than if I had instead written the entire pattern in the first argument position for my latter **re.sub** method. Likewise, I decided to structure this pattern inside of triple quotations and flag **re.VERBOSE** so I could ignore whitespace/line-breaks in my code and return my pattern multiple times to account for each group intended to match instances of PII. I also decided to use a *negative lookbehind* in order to stipulate that a white-space character precede each phone number matched in the file, as well as a *lookahead* for every matching instance in order to stipulate a whitespace character followed the end of each phone number. I did not, however, format an either/or operator to account for the end of a string (**Z**) because I knew ahead of time that no phone number would fit this specific condition. As for the actual matching properties, I decided to use the pattern **(-*\d*-)** with asterisks on either side in order to enable a flexible enough regular expression to account for digits adjacent to a hyphen; then quantified it with desired character lengths; and finally wrote **[\d]+** to set the scope of my quantifiers. What's more, I utilized the *either/or* operator to indicate the desire for one of four different patterns, compartmentalized according to each new line following the *negative lookbehind*. After testing these patterns in another module, I assigned a variable to the substitution of **\d**, revalued at **0** by calling the **lambda m** function. Once that step was set, I implemented my second **re.sub** method in the form of **pattern.sub(nullify, self)** and assigned it to the variable **nullify**, which I returned at the end of the function. I subsequently assigned the variable **test** to the act of calling the function, printed it, then conducted my assertion tests, including one for each number instance and its relative substitution, along with "911" in order to ensure that my function had not deleted irrelevant or undesired integers.