# Part 1: Regular Expressions

Part 1 of this assignment consisted of developing regular expressions to fit the most patterns in the various raw files. Regular expressions to find emails will first be explored, followed by regular expressions to match phone numbers. Each attempt to find the most matches will have the regular expression tried, along with the results of the regular expression, and the thought process behind the regular expression. Additionally, all regular expressions will be listed in chronological order, and will include all of the regular expression patterns shown before the current regular expression pattern.

## Email Patterns

### Email Regular Expression Pattern #1

**Pattern**: '([A-Za-z]+)@([A-Za-z]+)\.edu'

**Results**: TP: 4 FP: 1 FN: 113

**TP Matches**: 'balaji@stanford.edu', 'nass@stanford.edu'

**Obfuscated Matches**: "balaji@stanford.edu", "nass@stanford.edu"

**New FP Mismatches**:

**Found**: 'young@stanford.edu' **Actual**: 'patrick.young@stanford.edu'

The first regular expression attempted was a basic regular expression that found any emails that were not obfuscated in the raw files, and only contained a string of only upper and lowercase letters before the "@ " and a string of only upper and lowercase letters after the "@", and before the ".edu". The regular expression needed the "\" before the ".edu", because the "." is a special character in regular expressions that indicate any character. This regular Expression excluded any emails that contained any punctuation in the email, or any numbers.

### Email Regular Expression Pattern #2

**Pattern**: '([A-Za-z0-9.]+)@([A-Za-z0-9.]+)\.edu'

**Results**: TP: 19 FP: 0 FN: 98

**TP Matches**: 'kunle@ogun.stanford.edu', 'patrick.young@stanford.edu'

**Obfuscated Matches**: 'kunle@ogun.stanford.edu', 'kunle@ogun.stanford.edu'

**New FP Mismatches**:

None

The next regular expression pattern added onto the initial pattern any numbers potentially used in an email, and accounted for any number of "." used either in the username grouping, or the domain name grouping. This was a more basic pattern that did not account for any obfuscation of email addresses, but searched for the literal email.

### Email Regular Expression Pattern #3

**Pattern**: '([A-Za-z0-9.]+)\s{,3}@\s{,3}([A-Za-z0-9.]+)\.edu'

**Results**: TP: 23 FP: 0 FN: 95

**TP Matches**: 'ashishg@stanford.edu', 'ullman@cs.stanford.edu'

**Obfuscated Matches**: 'ashishg @ stanford.edu', 'ullman @ cs.stanford.edu',

**New FP Mismatches**:

None

The third regular expression created added a repetition qualifier of spaces between the username group and the "@" and another repetition qualifier for the spacing between the "@" and the domain name. The repetition qualifier allows for a minimum of 0 spaces, but can repeat up to 3 spaces. The first grouping once again allowed for any combination of uppercase letters, lowercase letters, numbers, and periods. After this grouping, the pattern could optionally be followed any kind of whitespace (tab, newline, space, etc.) up to three times with a required "@" symbol in the pattern. The "@" symbol is then optionally followed by a whitespace pattern up to three times, and requires a combination of letters, numbers, or a literal ".". Finally, a ".edu" will complete the pattern.

### Email Regular Expression Pattern #4

**Pattern**: '([A-Za-z0-9.]+)\s?<[A-Za-z ]+>@?\s?([A-Za-z0-9.]+)\.edu'

**Results**: TP: 28 FP: 0 FN: 89

**TP Matches**: 'latombe<del>@cs.stanford.edu', , 'manning@cs.stanford.edu'

**Obfuscated Matches**: 'latombe<del>@cs.stanford.edu', 'manning <at symbol> cs.stanford.edu'

**New FP Mismatches**:

None

The fourth regular expression was created to catch emails that attempted to obfuscate the email address some sort of fake html tag. Once again, the regular expression begins by capturing any string that contains only letters, numbers, and periods, this will be the username part of the email address. Next, the email address has an optional space, and then captures the beginning of a tag ("<") and any number of letters or spaces contained within the tag, and then captures the close of the tag (">"). After the tag has occurred, the email address has an optional "@" symbol followed by an optional space. The pattern than picks up any string of letters, numbers, and periods, and finishes with the ".edu" closing.

### Email Regular Expression Pattern #5

**Pattern**: '([A-Za-z.]+)\s[^A-Za-z0-9@\.].\*?@([A-Za-z.]+)\.edu'

**Results**: TP: 30 FP: 1 FN: 87

**TP Matches**: ouster@cs.stanford.edu'

**Obfuscated Matches**: 'ouster (followed by &ldquo;@cs.stanford.edu&rdquo;)',

**New FP Mismatches**:

"first name + first letter of last name"@cs.stanford.edu" : "name@cs.stanford.edu"

The final email regular expression created was built to capture email address that attempted to obfuscate the email address with one irregular token, not typically found in an email address, and searches until it finds the domain of the email address. First the regular expression looks for a string that contains letters or periods, which will act as the username for the email address. After the username, the pattern requires a space, followed by any character that is not a letter, number, "@", or period, which are common in an email address. After this uncommon character, any token can follow, until the "@" symbol is found. After the "@" symbol, the regular expression captures a group that has upper and lowercase letters and periods, which will be the domain name. Finally, the regular expression captures the ".edu” at the end of the domain name.

The false positive that this regular expression caught was an obfuscation that would be very difficult to capture properly. The reason it was caught was because the irregular token captured was a "+", which was acting as a concatenation of the parts of the email address that were not explicitly stated. This is why the 'first letter of last name"' was not included in the captured false positive.

Other patterns could be created to capture more specific patterns, but these would be more appropriate to capture using other regular expression substitutions, and text modification. These will be considered in Part 3.

## Phone Patterns

### Phone Regular Expression Pattern #1

**Pattern**: '(\d{3})-(\d{3})-(\d{4})'

**Results**: TP: 49 FP: 1 FN: 68

**TP Matches**: ' 650-723-1131', '650-725-3726'

**Obfuscated Matches**: ' 650-723-1131', '650-725-3726'

**New FP Mismatches**:

None

The first pattern attempted had three different groups separated by '-'. The first group must be three digits, the second group must be three digits, and the last group must be 4 digits. This regular expression pattern does not catch any obfuscated phone numbers or irregular patterns.

### Phone Regular Expression Pattern #2

**Pattern**: '(\d{3})[ -](\d{3})[- ](\d{4})'

**Results**: TP: 53 FP: 1 FN: 64

**TP Matches**: ' 650-725-9046', '650-723-3432'

**Obfuscated Matches**: '650 725 9046', '650 723-3432’

**New FP Mismatches**:

None

The second pattern created first begins with a group of digits, and there must be exactly three digits in the group. Next, the pattern requires either a space or a hyphen to separate the area code from the exchange part of the phone number. After the space or hyphen, another group of three digits is required. After the exchange, there is a required spacing of some sort, either a '-' or a space, before the phone number is finished with the final four digits of the phone number.

### Phone Regular Expression Pattern #3

**Pattern**: '[\(\[](\d{3})[\)\]][ -](\d{3})[- ](\d{4})'

**Results**: TP: 94 FP: 1 FN: 23

**TP Matches**: '650-725-3713', 650-725-2472'

**Obfuscated Matches**: '(650) 725-3713', '[650] 725-2472’

**New FP Mismatches**:

None

The next pattern created accounted for some sort of grouping punctuation around the area code. The pattern begins with a character class that contains '(' and '['. These were escaped because they both can indicate special cases. Next, the grouped of three digits representing the area code will be required. After the area code, a character class is required with the symbol being either a closed bracket or a closed parentheses, both of which are also escaped with a backslash to avoid the special cases of the symbols. The closed grouping punctuation is then followed by either a '-' or a space, and then a grouping of three digits which represent the exchange. A space or '-'­ then follows the exchange, and a final group of four digits completed the phone number.

### Phone Regular Expression Pattern #4

**Pattern**: '[\(\[](\d{3})[\)\]][ -]?(\d{3})[- ](\d{4})'

**Results**: TP: 102 FP: 1 FN: 15

**TP Matches**: '650-724-9147', '410-516-5521'

**Obfuscated Matches**: '(650)723-1614’, '(410)516-5521'

**New FP Mismatches**:

None

The final regular expression pattern was created for phone numbers was nearly the same as the previous regular expression. The only difference between the final regular expression pattern and the third regular expression pattern was that the character class that contained either a hyphen or a space became optional. This class did require brackets around the area code to ensure that 10 random digits in a row would not be caught by the pattern. This final regular expression matched all phone numbers in the

## (Option) Part 3:

After creating regular expression patterns that captured all phone number obfuscations and most email obfuscations, some changes to the code needed to be made. My process consisted of me looking at the false negatives within the original file, and then trying to find the most general solutions for capturing these parts of the email. After the more general patterns were captured by the code, I focused on the more complicated obfuscations to capture, but tried to not make the solutions too specific to that email.

One of the first steps I took was creating a new email pattern that would capture three different parts of the email: username, domain, and the domain closing (com, org, edu, etc.). This pattern allows for more flexibility when trying to capture email patterns.

There were also some changes made to the part of the code that parses through each line to make the regular expressions easier to capture. For example, all of the lines passed into the program were converted into lowercase letters. This generalizing allows for some tokens that are unique to email obfuscation be easier to identify in the program.

Some of the email patterns were fairly unique and would be quite difficult to match simply by using capturing groups in the regular expression; however regular expressions can be used to target specific obfuscations and replace with easier to capture tokens. This includes replacing obfuscations for "dot" instead of "." and the use of "at" instead of "@". The use of regular expression substitutions were generally the most helpful in capturing unique groups.

There was also a third regular expression pattern created to capture emails obfuscated by JavaScript functions. The example in the data files had the domain of the email as the first parameter and the username of the email as the second parameter. The new pattern created only captured two groups; however, instead of unpacking the groups in order (as had been done in part 1), the groups that were input into the final pattern with the index 1 group first and the index 0 group second.

The final output of this program resulted in capturing all of the true positive emails and all of the true positive phone numbers (117 total). The program did have 4 false positive matches in the data. Two of the false positive matches were two different email servers that were formatted in a manner similar to an email address. Another false positive was captured after making all of the lines lowercase. This false positive was obtained due to the generality of a rule created in Part 1, which collects every group after a special character until it reaches the "@" character. The fourth false positive was collected in part 1.

The new email patterns and regular expressions are listed below, as well as some of the changes made to the code when parsing through each of the lines. A summary to the process\_file function are as follows:

First the function initializes five different regular expressions that it will look through on each of the lines. The first regular expression looks for obfuscations of places where the "." is spelled out (e.g dot, dt, dom). The second regular expression looks characters that may be html character references. These character references generally begin with a "&" and contain letters and numbers. The third regular expression is created to catch html comments. These comments follow the pattern of beginning with "<!--“, the pattern then can have any number of characters except for another "<" character. These html comments generally look like

"<!--comment here-->". The fourth regular expression was create to look for patterns that repeat letters followed by a dash a minimum of two times (e.g e-x-a-m-p-l-e). The last regular expression created is designed to find email patterns that are spelled out and without periods, but only captures everything after the "at".

After the regular expressions are created, the program runs through a for loop, going through each line in the file. For each line, the function will make the line all lowercase to standardize the patterns used in the obfuscation. It will then look to see if the line has the repeated letter then dash pattern. If it does, the function will replace all dashes in that line with an empty string, essentially deleting the dashes. Then, the program automatically substitutes any html comments in the line with an empty string, removing the comments. After the html comments have been removed, the same process occurs for any obfuscated ".". After the dots have been replaced in the line, the function looks for any html character references. If there are any character references, the program *unescapes* the line, converting the html character reference into the character it was referencing. The function then looks to see if the line has the pattern of the domain being separated by spaces. If it does, the captured, spaced out domain is replaced with the captured domain that had the spaces replaces with periods. Finally the function replaces any semi-colons in the line with periods.

The function then goes through all the matches found from Part 1’s email and phone regular expressions as it had before any changes. Then the function sorts through any matches to the first newly created pattern that contains three captured groups (%s@%s%s) and adds them to the results list. Finally is goes through the other newly created pattern list created that has two capture groups (%s@%s); however this pattern places the index 1 captured group into the first "%s", and then places the index 0 capture group in the second "%s".

### New Email Pattern #1

**Regex Email Pattern**: '([A-Za-z\.]+)@([A-Za-z\.]+)(\.EDU)'

**New Matching Pattern:** '%s@%s%s' % m

**Results**: TP: 103 FP: 1 FN: 14

**TP Matches**: 'uma@cs.stanford.edu'

**Obfuscated Matches**: 'uma@cs.stanford.EDU'

**FP Mismatches**:

None

The first new regular expression created with three groups was meant to capture emails with capitalized end to the domain name. The regular expression first looks for a string that contains letters and periods, which will be saved as the username for the email. The email will then expect the "@" symbol followed by a string of letters and periods. Finally, the regular expression will find emails that have a capitalized ".EDU". This could have been captured in Part 1 if the ".EDU" was provided after the group for the domain name.

### Code Change #1

**Code Insert**: line = line.lower()

**Results**: TP: 103 FP: 2 FN: 14

**TP Matches**: 'uma@cs.stanford.edu'

**Obfuscated Matches**: 'uma@cs.stanford.EDU'

**FP Mismatches**:

Actual: ' Uma Mulukutla \_ uma at cs.Stanford.EDU

Captured: 'mulukutla@cs.stanford.edu'

The first new change to the code in the program makes all of the letters in the line lowercase. This change standardizes the line, so capitalization does not affect what the line looks for. Initially, this does not add any new obfuscations, but it would capture ".EDU" pattern captured with the first new regular expression added. This change to the code is primarily meant to narrow allow less specific regular expressions to capture more complicated obfuscations. This regular expression did add a new false positive because of a regular expression created in part 1 where alphabetical characters before a special character are captured and the rest of the email is captured after the "@" symbol. This change to the code will be more helpful in capturing emails later in the program.

### New Email Pattern #2

**Email Regex Pattern:** ([a-z.]+) [a-z]+ ([a-z.]+)(\.edu)

**Code Inserts**:

patDot = re.compile(" do?[mt] ")

…

for line in f:

line = re.sub(patDot, ".", line)

**Results**: TP: 109 FP: 4 FN: 8

**TP Matches**: 'hager@cs.jhu.edu', 'engler@stanford.edu'

**Obfuscated Matches**: 'hager at cs dot jhu dot edu', 'engler WHERE stanford DOM edu'

**FP Mismatches**:

Actual: 'Server at infolab.stanford.edu', 'Server at cs.stanford.edu'

Captured: 'server@infolab.stanford.edu', ' server@cs.stanford.edu'

The next change to the code was created to capture obfuscated emails that contained the words meant to obfuscate required or common characters, such as "dot" and "at", but was created to be more general to capture other obfuscations, such as "where" and "dom". The regular expression added to the code looks for a space followed by a "d" and an optional "o". The regular expression then expects a "t" or an "m", and eds with another space. This will capture such obfuscations as " dom ", " dot ", and " dt ". After finding these patterns surrounded by spaces, the pattern and spaces around the patterns will be replaced with a "." and the surrounding spaces will be removed. The regular expression to capture the contents of the email address first looks for a group of strings and periods, followed by a space and any length string of characters and another space. This string of characters will be where the "@" symbol is obfuscated. After the "@", the regular expression looks for another string of characters and periods and expects a ".edu" at the end.

The key to this regular expression is having the obfuscated "." condensed from the substitution. This will create two unique strings that look would be part of an email. Theses email looking patterns surrounding a word would likely be a email address, especially if ".edu" is on the very end of the string. The false positives captured by this regular expression are strings that look like they would be an email address, however they are just the server address.

### Code Change #2

**Code Inserts**:

patHtml = re.compile('&[a-z0-9#]')

…

if re.search(patHtml, line) is not None:

#if there is, make it literal in utf8

line = html.unescape(line)

**Results**: TP: 111 FP: 4 FN: 6

**TP Matches**: melissa@graphics.stanford.edu ', 'ada@graphics.stanford.edu'

**Obfuscated Matches**: ' melissa&#x40;graphics.stanford.edu', ' ada&#x40;graphics.stanford.edu'

**New FP Mismatches**: None

This line of code is an attempt to convert html numeric character references and convert them into the actual character it is representing. This sort of obfuscation is fairly common in html documents, as it will automatically render as the actual character when seen in the browser. Two email addresses are picked up with this line of code as "&#x40;" is a character reference to "@". With the html.unescape function, the character references are converted into the literal string if there is a pattern of an ampersand, followed by letters, numbers, and pound symbols. Even if there is some other character reference that is not the “@” symbol, this function will simplify the string for capturing.

### Code Change #3

**Code Inserts**:

for line in f:

…

line = line.replace(";", ".")

**Results**: TP: 112 FP: 4 FN: 5

**TP Matches**: 'jks@robotics.stanford.edu'

**Obfuscated Matches**: 'jks at robotics;stanford;edu'

**New FP Mismatches**: None

This line of code is added to the program when the program starts going through each line of the file. This line was added after the html.unescape() function because the html character references often end with a ";", and this would make the program not work as expected. The replace function replaces all instances of ";" with a ".". This will convert any email obfuscated with a semi-colon to a pattern that was created earlier in the program.

### New Email Pattern #3

**Email Regex Pattern**: '([a-z]+) [a-z]+ ([a-z.]+)(\.com)'

**Results**: TP: 113 FP: 4 FN: 4

**TP Matches**: 'support@gradiance.com'

**Obfuscated Matches**: 'support at gradiance dt com'

**New FP Mismatches**: None

The regular expression added to this section depends on the regular expression substitute created earlier that changed period obfuscations into the actual period. The email pattern regular expression looks for a string of any number of letters, and is the followed by a space, a word of any length, and then followed by a space. This section acts as the "@" symbol, and will not be included in the final output. The regular expression then looks for a string of any length letters, and captures that in the second group. Finally, the pattern looks for ".com" and captures that in the final group.

### New Email Pattern #4

**Email Regex Pattern**: '\'([a-z.]+.edu)\', ?\'([a-z.]+)\''

**New Matching Pattern:** '%s@%s' % (m[1], m[0])

**Results**: TP: 114 FP: 4 FN: 3

**TP Matches**: jurafsky@stanford.edu'

**Obfuscated Matches**: obfuscate('stanford.edu','jurafsky')

**New FP Mismatches**: None

To capture the jurafsky email address using regular expressions, a new list had to be created that only collected two groups, the username and the entire domain of the email address. The obfuscation in this case is created by using a JavaScript function to write the correct email address in the html document. The regular expression works by first looking for a quotation mark, and then it captures a group of lowercase letters, followed by a ".edu". This group will be the domain of the email address. After the domain has been captured, the regular expression searches for another quotation mark followed by a comma, and an optional space. This expression is essentially trying to find if a function is being used, and if one is, capture the first argument. After the optional space, the regular expression searches for another quotation mark, and then tries to capture a group to be used as the username. The group looks for a string of lowercase letters, and captures that as the username. Finally, the regular expression looks for the closing quotation mark of the second string.

The new matching pattern works by creating a tuple of the matched patterns and rearranging the items in the list to appear with index 1 first (username) and index 0 (domain) second.

### Code Change #4

**Code Inserts:**

patComment = re.compile('<!--[^<]+--> ')

…

for line in f:

…

line = re.sub(patComment, "", line)

**Results**: TP: 115 FP: 4 FN: 2

**TP Matches**: vladlen@stanford.edu'

**Obfuscated Matches**: vladlen at <!-- die!--> stanford <!-- spam pigs!--> dot <!-- die!--> edu</div>

**New FP Mismatches**: None

To capture the jurafsky email address using regular expressions, a new list had to be created that only collected two groups, the username and the entire domain of the email address. The obfuscation in this case is created by using a JavaScript function to write the correct email address in the html document. The regular expression works by first looking for a quotation mark, and then it captures a group of lowercase letters, followed by a ".edu". This group will be the domain of the email address. After the domain has been captured, the regular expression searches for another quotation mark followed by a comma, and an optional space. This expression is essentially trying to find if a function is being used, and if one is, capture the first argument. After the optional space, the regular expression searches for another quotation mark, and then tries to capture a group to be used as the username. The group looks for a string of lowercase letters, and captures that as the username. Finally, the regular expression looks for the closing quotation mark of the second string.

The new matching pattern works by creating a tuple of the matched patterns and rearranging the items in the list to appear with index 1 first (username) and index 0 (domain) second.

### Code Change #5

**Code Inserts:**

patDash = re.compile('([a-z.@]-){2,}')

…

if re.search(pattern=patDash, string= line) is not None:

line = line.replace("-", "")

**Results**: TP: 116 FP: 4 FN: 1

**TP Matches**: 'dlwh@stanford.edu'

**Obfuscated Matches**: 'd-l-w-h-@-s-t-a-n-f-o-r-d-.-e-d-u'

**New FP Mismatches**: None

Another regular expression was inserted into the code to capture email obfuscations that had dashes between every character. The regular expression captured any instance that had a letter, "@" symbol, or a period followed by a dash. However, this pattern needed to occur at least twice for the regular expression to capture it. The regular expression function re.search() was used to find any occurrences in each of the lines passed into the code. If there was an occurrence of this pattern in the line, the code would replace each of the dashes with a an empty string, essentially deleting them. After this replacement occurred, the email then appears as a normal email in the program.

### Code Change #6

**Code Inserts:**

patSpace = re.compile('at ([a-z ]+ edu)')

…

spaceMatch = re.findall(patSpace, line)

if len(spaceMatch) != 0:

line = line.replace(spaceMatch[0], spaceMatch[0].replace(" ", "."))

**Results**: TP: 117 FP: 4 FN: 0

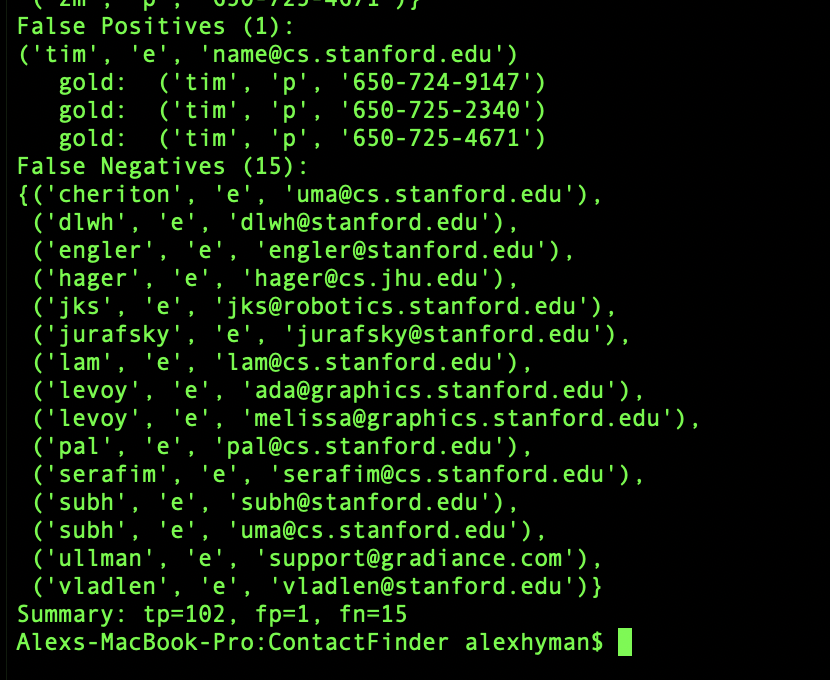
**TP Matches**: vladlen@stanford.edu'

**Obfuscated Matches**: 'email: pal at cs stanford edu'

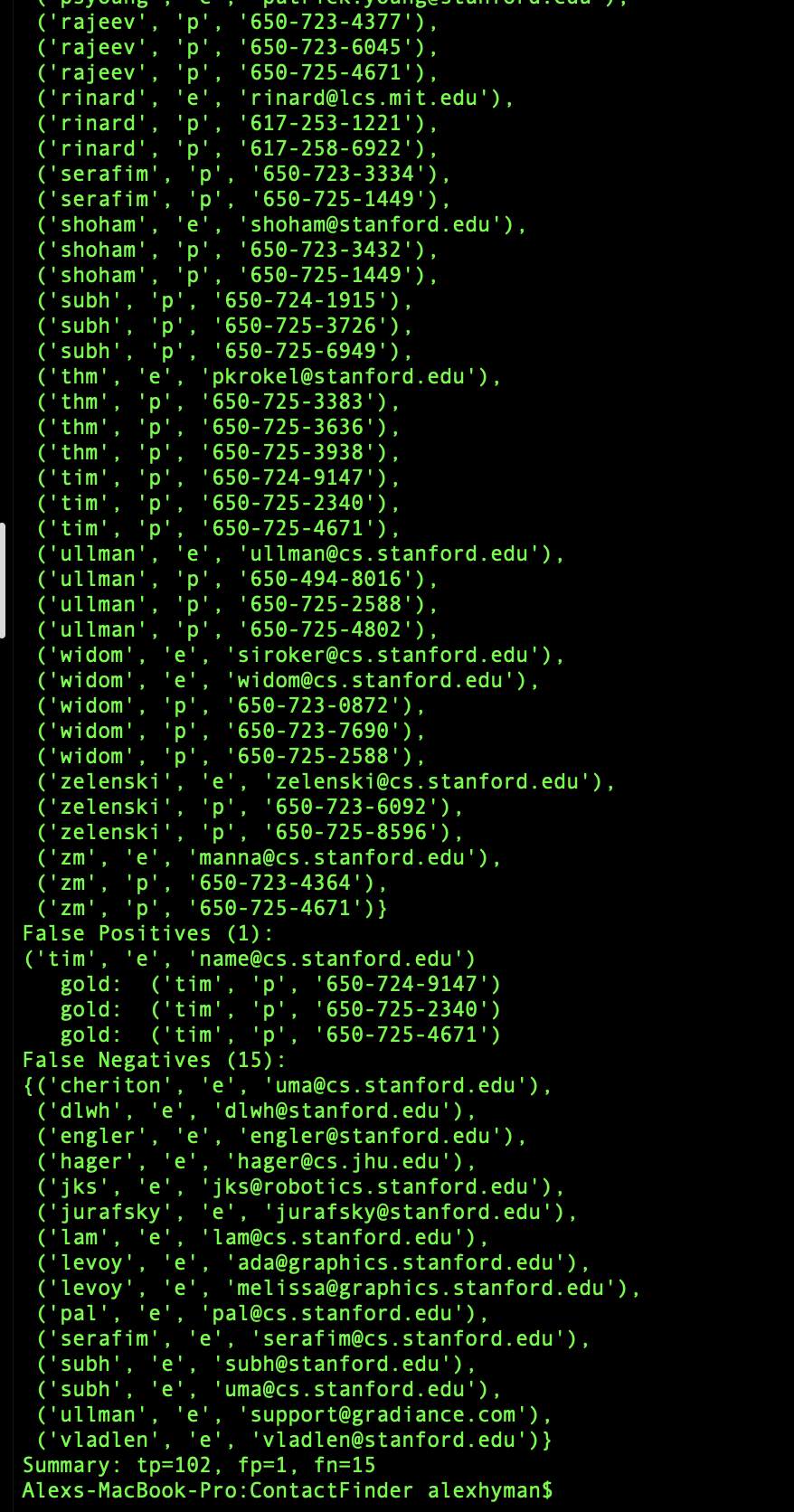
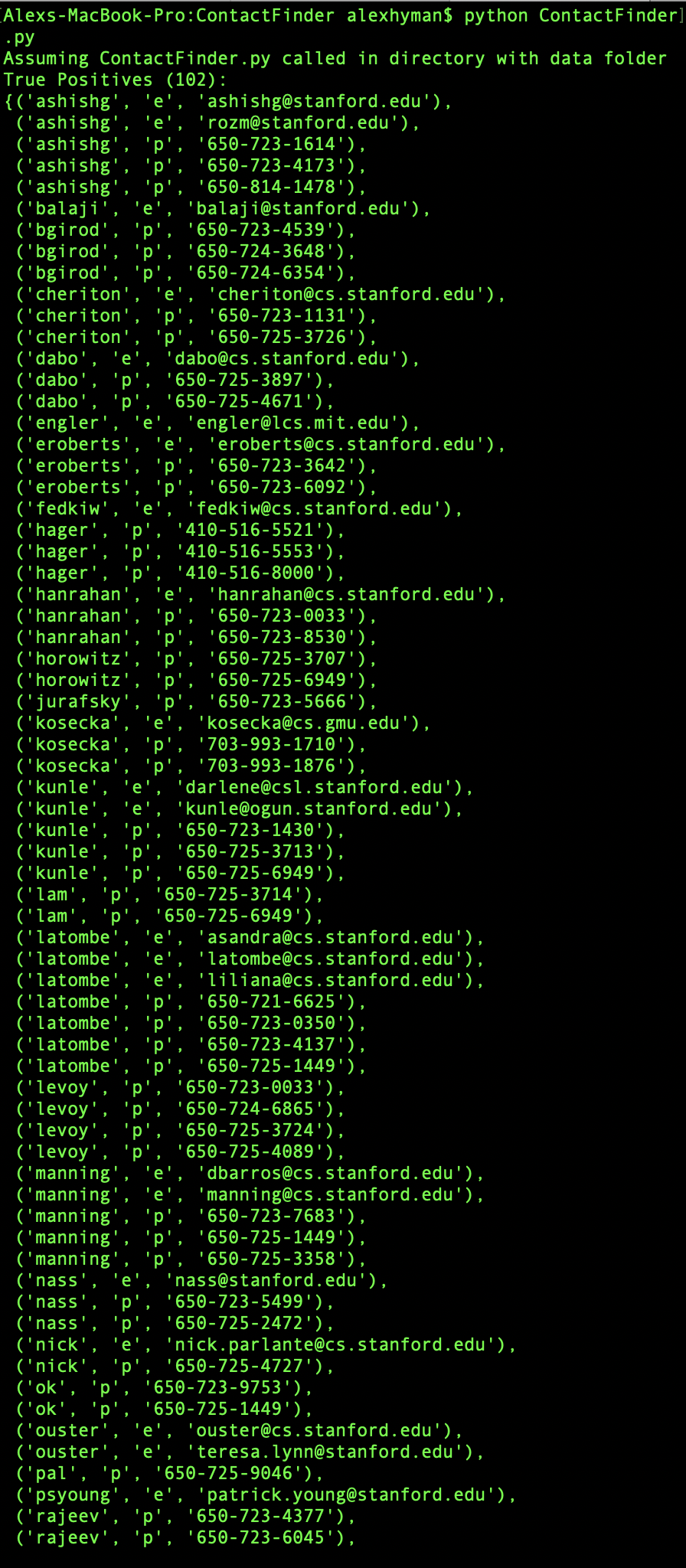
**New FP Mismatches**: None.

The final change to the program was able to capture the last obfuscated email in the dataset. A new regular expression was added to the code to look for a string that begins with the word "at" and then has a capturing group to find any words and spaces that end with a final space + "edu". The regular expression function re.findall() is used on every line passed through the program, and an if statement checks to see if the function has captured anything from the line. If the function did find anything from the line, then it will be saved in a list called spaceMatch, and will have a length greater than zero. Because the regular expression only captures the part of the line from after the "at" all the way to the "edu", each part of the domain will be separated by a space in the zero index of the spaceMatch list. If the list does have a length greater than zero, the program will replace the captured string (domain) with the captured string that has had its spaces replaced with periods. This will result in a single string with the part of the domain combined with periods.

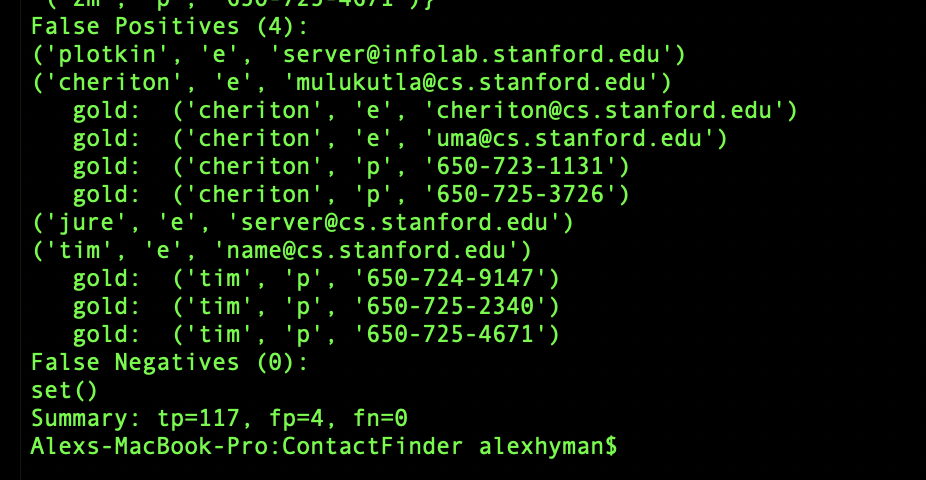
**Output Part 1 (Condensed):**

****

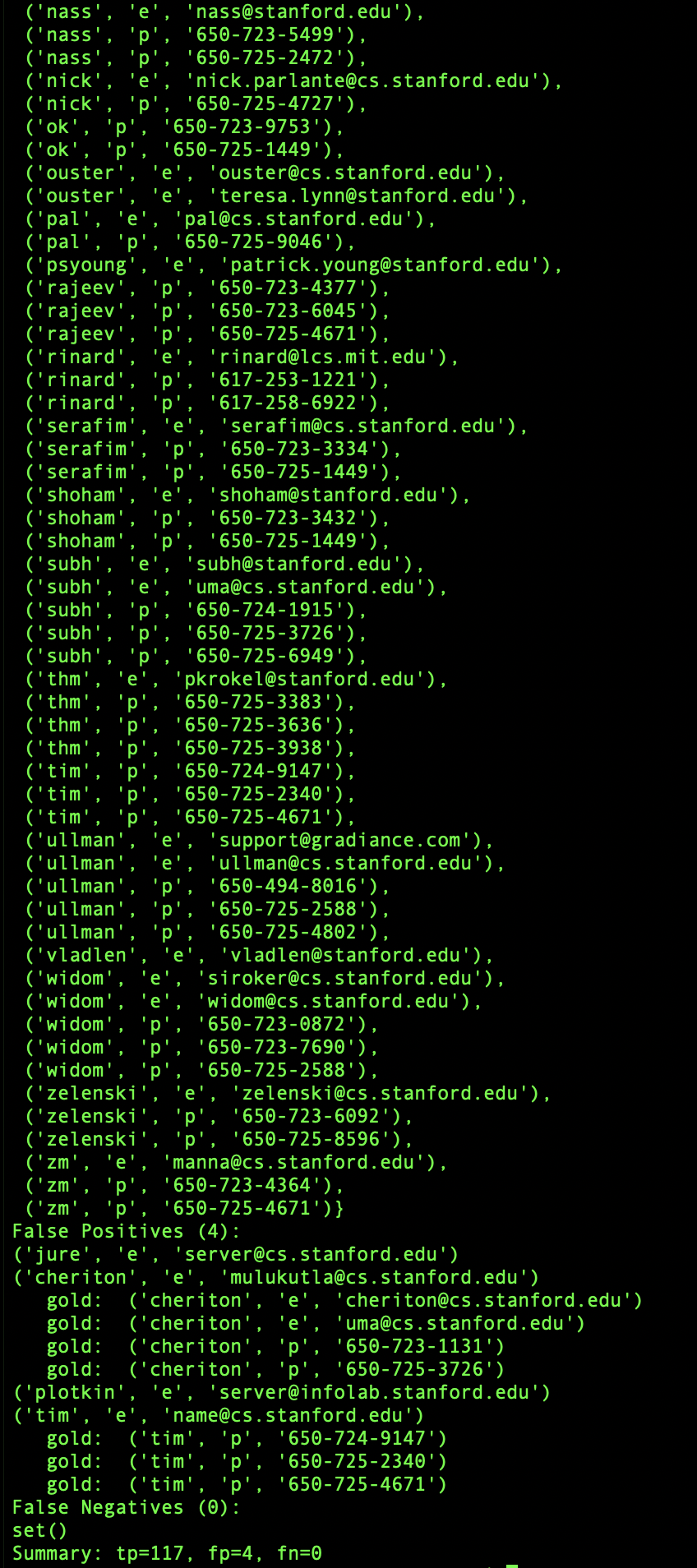
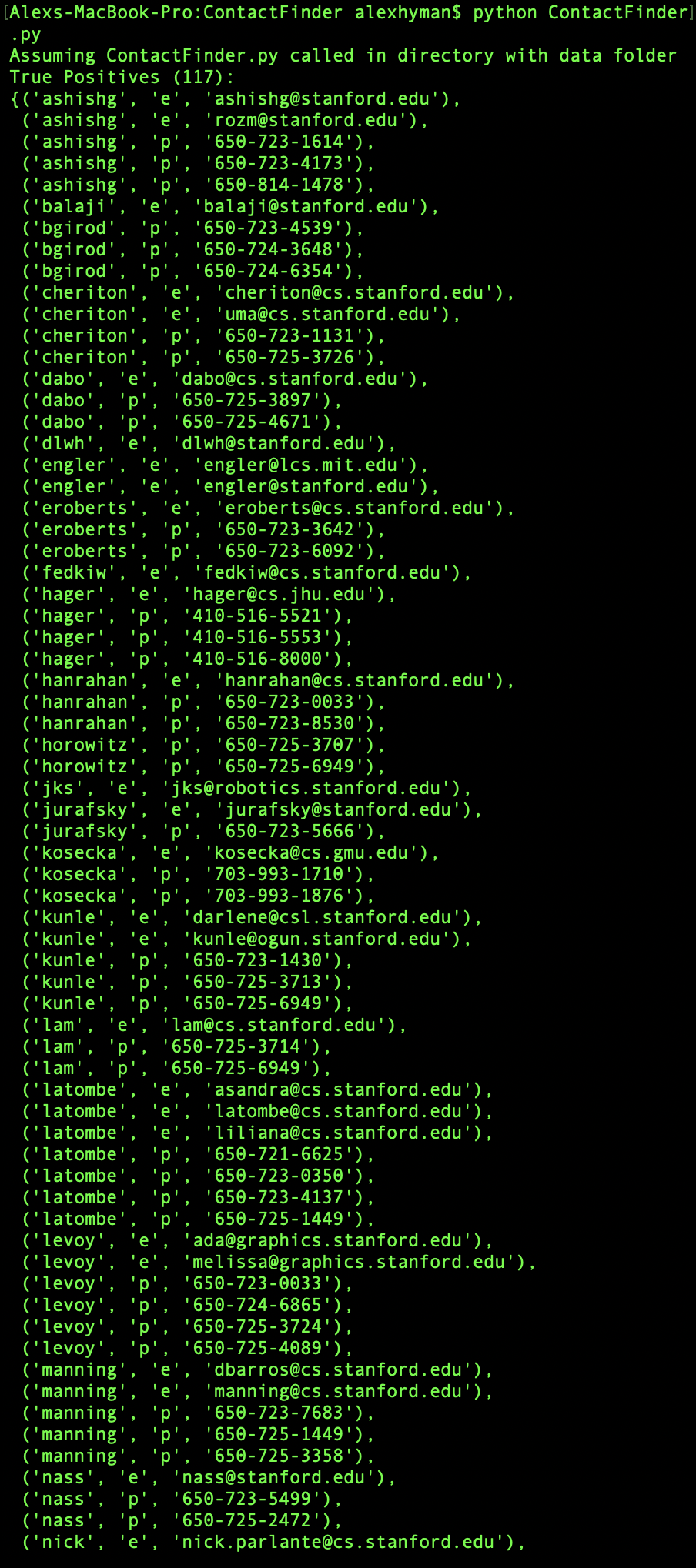
**Output Part 1 (All):**

****

**Final Part 3 (Condensed):**



**Output Part 3 (All):**

****