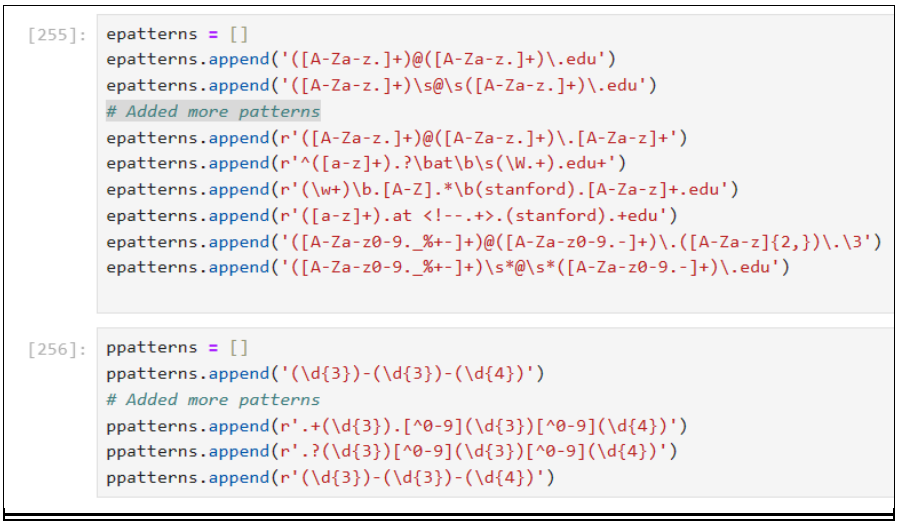
Program:



```
[255]: epatterns = []
       epatterns.append('([A-Za-z.]+)@([A-Za-z.]+)\.edu')
       epatterns.append('([A-Za-z.]+)\s@\s([A-Za-z.]+)\.edu')
       # Added more patterns
       epatterns.append(r'([A-Za-z.]+)@([A-Za-z.]+)\.[A-Za-z]+')
       epatterns.append(r'^([a-z]+).?\bat\b\s(\W.+).edu+')
       epatterns.append(r'(\w+)\b.[A-Z].*\b(stanford).[A-Za-z]+.edu')
       epatterns.append(r'([a-z]+).at <!--.+>.(stanford).+edu')
       epatterns.append('([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+)\.([A-Za-z]{2,})\.\3')
       epatterns.append('([A-Za-z0-9._%+-]+)\s*@\s*([A-Za-z0-9.-]+)\.edu')

[256]: ppatterns = []
       ppatterns.append('(\d{3})-(\d{3})-(\d{4})')
       # Added more patterns
       ppatterns.append(r'.+(\d{3}).[^0-9](\d{3})[^0-9](\d{4})')
       ppatterns.append(r'.?(\d{3})[^0-9](\d{3})[^0-9](\d{4})')
       ppatterns.append(r'(\d{3})-(\d{3})-(\d{4})')
```

I have appended some regex patterns to match email IDs and Phone numbers. Please refer to the above screenshot.

Added code snippets are adding a series of regular expression (regex) patterns to two lists, epatterns and ppatterns, which are likely used for extracting email addresses and phone numbers from text, respectively. Here's a brief explanation of each pattern:

Email Patterns (epatterns):

Matches email addresses with any domain extension (not just .edu), capturing the username and domain.

Targets emails embedded in text, possibly with obfuscation using "at" for "@" and non-standard characters before ".edu".

Looks for email addresses associated with Stanford, allowing for various obfuscations and capitalization before mentioning "stanford" and ending with ".edu".

Matches emails obfuscated with ".at" within HTML comments, specifically targeting "stanford.edu".

Attempts to match email addresses ending with a repeated top-level domain (TLD), though the pattern seems to be incorrect due to the misuse of \3 without a corresponding third capturing group in the same pattern.

Captures email addresses ending with .edu, allowing for optional spaces around the "@" symbol.

Phone Patterns (ppatterns):

Matches phone numbers with various separators, capturing area code, exchange, and line number, allowing for leading characters.

Similar to the first, but with an optional leading character, capturing phone numbers with non-digit separators.

Directly matches phone numbers formatted as XXX-YYY-ZZZZ with dashes as separators.

The use of r before each pattern string denotes a raw string in Python, which tells the interpreter not to treat backslashes as escape characters, making it easier to work with regex patterns.

Patterns use capturing groups (()) to extract specific parts of the matched text, such as the username and domain of an email, or the segments of a phone number.

Some patterns include special regex constructs like \b for word boundaries, \W for non-word characters, and character classes like \d for digits.

The attempt to reference a backreference with \3 in the fifth email pattern is misplaced, as there's no third capturing group within that pattern, indicating a potential error or oversight in the regex construction.

These patterns collectively aim to match and extract email addresses and phone numbers from text, dealing with a range of common formats and some attempts to obfuscate or vary the formatting.

Observations:

The shared program, adapted from a Stanford NLP class assignment, is designed to extract obscured email addresses and phone numbers from HTML text files. It utilizes regular expressions (regex) to identify patterns in the text that correspond to these contact details, aiming to standardize them into a recognizable format. Here's a breakdown of its key components and functionality:

Pattern Lists: The program defines two lists, epatterns for email patterns and ppatterns for phone patterns. Each regex pattern within these lists includes groups (denoted by parentheses) to capture different parts of emails or phone numbers. For emails, the groups capture the user ID and domain parts; for phone numbers, they capture the area code, exchange, and number parts.

File Processing: The process_file function reads through each line of a given file, applying the regex patterns to find matches. When a match is found, it constructs the standard form of the email or phone number from the captured groups and adds this information to a results list, alongside the type of contact ('e' for email, 'p' for phone) and the file name.

Directory Processing: The process_dir function iterates over all files in a specified directory, applying the process_file function to each and compiling a complete list of guessed contacts.

Scoring and Evaluation: The program compares the extracted contacts against a "gold standard" file containing the correct answers. It categorizes matches as true positives (correctly identified contacts), false positives (incorrectly identified or formatted contacts), and false negatives (missed contacts). The output is designed to help debug and refine the regex patterns by showing where they succeed or fail.

Goal: The main objective is to enhance the regex patterns to improve the program's accuracy in identifying and standardizing obscured contact information. This involves minimizing false positives and negatives while maximizing true positives.

Optional Extension: The program allows for further refinement and expansion, such as adding new lists of patterns for different types of obscured contacts or

adjusting the process_file function to handle additional pattern matching and standardization scenarios.

This program illustrates the practical application of regex for data extraction and the iterative process of pattern refinement for improved NLP accuracy.

Lessons:

Throughout the tasks involving regular expressions (regex) for parsing and extracting email addresses and phone numbers from text data, several key lessons can be learned:

Importance of Flexibility: Regular expressions need to be flexible enough to handle variations in the data, such as spaces around symbols or obfuscated characters. This flexibility ensures broader applicability and higher accuracy.

Precision vs. Recall Trade-off: There's a balance between capturing as many valid cases as possible (recall) and ensuring the matches are correct (precision). Adjusting regex patterns can affect this balance, highlighting the need for iterative refinement to optimize both.

Understanding Regex Components: Mastery of regex syntax and components (like character sets, quantifiers, and groups) is crucial for creating effective patterns. Each component plays a role in how effectively the pattern matches the intended text.

Debugging and Testing: Regular testing and debugging are essential, especially when dealing with complex patterns. Evaluating the patterns against a known dataset helps identify and correct false positives and negatives.

Simplifying Complex Patterns: Breaking down complex patterns into simpler components can make the regex more understandable and maintainable. It also helps in isolating issues for easier debugging.

Domain-Specific Patterns: The effectiveness of regex patterns can be highly domain-specific. Patterns that work well for one dataset (like academic emails)

might not be as effective for another (like commercial emails), underscoring the importance of customizing regex to the task.

Regular Expressions Are Not a Panacea: While powerful, regex has limitations, especially with highly obfuscated or irregular data formats. Sometimes, a combination of regex and other text processing techniques might be necessary.

Learning from False Positives/Negatives: Analyzing instances where the regex fails to match correctly or matches incorrectly can provide insights into both the data and how to refine the patterns.

Ethical Considerations: When extracting personal information like emails and phone numbers, it's important to consider privacy and ethical implications, ensuring compliance with data protection laws and guidelines.

Continuous Improvement: The process of defining and refining regex patterns is iterative. Patterns can always be improved as new examples are encountered or as the data evolves.

These lessons highlight the nuanced, powerful nature of regex in text processing and the careful consideration needed to effectively extract and analyze data.
