coding temple

Python Intermediate

Module 3:

Python Regular Expressions

Introduction

Welcome, Coding Templers! 🌟

Get ready for an exciting exploration into the world of Regular Expressions (regex) in Python. Throughout this journey, you'll learn foundational regex syntax, including metacharacters, special sequences, and character sets. Python's re module will be your guide, offering functions like findall(), search(), match(), split(), and sub() for versatile text processing.

The real-world applications are diverse, from web scraping to data validation. We'll cover best practices, debugging, and performance tips. Each step in this regex adventure enhances your Python skills, making you a more proficient coder. Brace yourself for an adventure that promises to elevate your programming abilities and open up new horizons in data processing.

Happy learning! 🚀🌟🐍

Learning objectives

At the end of this lesson you should be able to:

- Identify and apply various metacharacters, special sequences, and sets in regex for pattern matching.
- Utilize the re module in Python for implementing regex functionalities.
- Apply the re.findall() function to extract all occurrences of a pattern in a given text.
- Use the re.search() function to locate the first occurrence of a pattern within a string.
- Employ the re.match() function to check for the presence of a pattern at the beginning of a string.
- Use the re.split() function to split strings.
- Use the re.sub() function for replacing patterns in strings and performing text formatting tasks

The Magic of Regular Expressions:

Why Every Python Programmer Should Learn Them

Regular expressions (regex) are a powerful tool for pattern matching and text manipulation that allows you to:

- Search Efficiently
- Validate Data
- Manipulate Text
- Save Time

Leveraging Knowledge of Strings and Loops

Regex is a natural progression that builds upon these foundations:

- **String:** Enhance your ability to analyze and manipulate strings.
- **Loops:** Help you process multiple pieces of text in sophisticated ways.

Leveraging Knowledge of Strings and Loops

To master regex, you need to know that it uses two main types of characters:

- **Literal Characters:** Exact characters you want to match.
- **Metacharacters:** Special symbols that represent broader categories or patterns

Mastering Metacharacters in Python's Regular Expressions

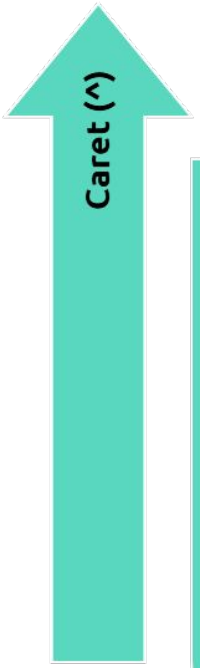
Metacharacters are the building blocks of regex patterns, each symbol holds a unique power to decipher patterns in text serving a special function. Ready to meet these magical symbols?

Dot (.)

The wildcard: Matches any single character except newline (`\n`).

Imagine it as a shape-shifter, able to become any character you need, just for a moment.

- Task: Find any three-character sequence in a text, where the middle character can be anything, the first has to be ‘c’ and the last has to be ‘t’.
- Regex Pattern: `` `c.t` ``
- Test Sentence: "I found a cat, a cot, and a cut in the room."
- Expected Matches: `` `['cat', 'cot', 'cut']` ``
- Explanation: The dot `` `.` `` matches any single character (except newline), so it finds sequences where 'c' and 't' are separated by any character.



The anchor for the start of a string.

Like a sentinel, standing guard at the beginning of your text.

- **Task**: Find strings that start with 'Py'.
- **Regex Pattern**: ^Py
- **Test Sentence**: "Python is fun"
- **Expected Matches**: ['Py'] from 'Python' at the beginning of the sentence.
- **Explanation**: The caret ^ ensures that the match must occur at the start of the string or line

Dollar ($)

The anchor for the end of a string.

The sentinel at the gates, ensuring nothing goes beyond the end of your text.

- **Task**: Identify strings that end with 'fun'.
- **Regex Pattern**: `fun$`
- **Test Sentence**: "Learning regex is fun"
- **Expected Matches**: `['fun']` from 'Learning regex is fun.'
- **Explanation**: The dollar `$` ensures that 'fun' is matched only if it's at the end of the string or line.

Simple Exercises for Understanding

Asterisk (*)

Matches zero or more occurrences of the pattern left to it.

Think of it as a multiplier, creating copies of the character before it.

- **Task**: Match a character followed by zero or more 'a's.
- **Regex Pattern**: `ba*`
- **Test Sentence**: "I saw a bat, and a ball in my bed, baaah!"
- **Expected Matches**: `['ba', 'ba', 'b', 'baaa']` from 'bat', 'ball', 'bed', and 'baaah!'.
- **Explanation**: The pattern starts with the literal character 'b'. This means it will first look for occurrences of 'b' in the text. Following the 'b', we have 'a*'. Then, the asterisk * which matches zero or more occurrences of the preceding character ('a' in this case).

Simple Exercises for Understanding

Plus (+)

Matches one or more occurrences of the pattern left to it.

Similar to the asterisk, but insists on at least one occurrence.

- **Task**: Find a character followed by one or more 'a's.
- **Regex Pattern**: `ba+`
- **Test Sentence**: "The battle of ba and baat."
- **Expected Matches**: `['ba', 'ba', 'baa']` from 'battle', 'ba', and 'baat'.
- **Explanation**: The plus `+` matches one or more occurrences of the preceding character ('a' in this case).

Simple Exercises for Understanding

coding temple

It makes the preceding character optional.

It's the symbol of uncertainty, allowing flexibility in your patterns.

- **Task**: Match 'colour' or 'color'.
- **Regex Pattern**: `colou?r`
- **Test Sentence**: "The color is nice. I like this colour."
- **Expected Matches**: `['color', 'colour']`
- **Explanation**: The question mark `?` makes the preceding character ('u' in this case) optional.

Backslash (\)

Escapes special characters or signals a special sequence.

The key to differentiating between a literal character and a magical symbol.

- **Task**: Match a period character in a sentence.
- **Regex Pattern**: `\.`
- **Test Sentence**: "End of sentence. Start of a new one."
- **Expected Matches**: The periods `[.]` at the end of 'sentence.' and before 'Start'.
- **Explanation**: In regex, the period (.) is a special character used as a wildcard. To match an actual period, the backslash `\` is used to escape the special meaning of the period, treating it as a literal character. The pattern `\.` specifically looks for the period character in the text.

Simple Exercises for Understanding

Square Brackets ([])

A set of characters. Matches any one character in the brackets.

Like choosing one tool from a toolbox, it selects one character from a set.

- **Task**: Find all vowels in a word.
- **Regex Pattern**: `[aeiou]`
- **Test Word**: "Regular"
- **Expected Matches**: `['e', 'u', 'a']`
- **Explanation**: The square brackets [] define a set of characters, any of which can be matched.

Simple Exercises for Understanding

coding temple

The OR operator. It matches either the pattern before or after it.

A fork in the road, giving you a choice between paths.

- **Task**: Match 'cat' or 'dog'.
- **Regex Pattern**: `cat|dog`
- **Test Sentence**: "I have a cat and a dog."
- **Expected Matches**: `['cat', 'dog']`
- **Explanation**: The pipe `|` acts as an OR operator, matching either the pattern before or after it.

Parentheses (())

Groups patterns together and captures them.

Think of them as binding a spell, containing its power within.

- **Task**: Find repetitions of 'woof' or 'meow'.
- **Regex Pattern**: `(woof|meow)+`
- **Test Sentence**: "The pets say woof woof and meow."
- **Expected Matches**: `['woof', 'woof', 'meow']`
- **Explanation**: Parentheses `()` group patterns, allowing the plus `+` to apply to the entire group.

Simple Exercises for Understanding

Curly Braces ({})

Curly braces are used to define the exact number of times a character or a pattern must occur for a match to be found.

It's like telling your magic spell exactly how much power to use.

- **Task**: Match a word where 'l' is followed by exactly two 'o's.
- **Regex Pattern**: `lo{2}`
- **Test Sentence**: "Look at the loom and the balloon in the room."
- **Expected Matches**: 'loo' in 'loom' and 'balloon'
- **Explanation**: The pattern `lo{2}` searches for an 'l' followed by exactly two 'o's. In our test sentence, it successfully identifies 'loo' within the words 'loom' and 'balloon', demonstrating the ability of curly braces {} to specify an exact number of occurrences.

Simple Exercises for Understanding

Deciphering the Code of Special Sequences in Regular Expressions

Each one, like `\d`, `\w`, or `\s`, among others unlocks a specific pattern in text, making our journey through the dense forest of data both exciting and efficient.

Let's unravel these mysterious codes!

Discovering the Power of Sets in Regular Expressions

Regex sets, symbolized by square brackets, function like a painter's palette for characters. They allow you to tailor your search by defining specific groups. For instance:

- `[abc]` can match 'a', 'b', or 'c'.
- `[a-z]` match any lowercase letter.

These sets act as customizable tools, enabling you to precisely mix and match characters in the vast expanse of text, akin to a painter creating the perfect shade with a palette of colors.

Exploring Ranges and Patterns

Within sets, you can define ranges. For example:

- `[a-e]` matches any letter from 'a' to 'e'.
- `[a-z]` match any alphanumeric character.

These sets act as customizable tools, enabling you to precisely mix and match characters in the vast expanse of text, akin to a painter creating the perfect shade with a palette of colors.

Integrating Sets with Python Concepts

- **With Loops:** Iterate through strings, using sets in regex to process or extract specific patterns.
- **With Conditions:** Use sets in regex within conditional statements to make decisions based on text patterns.

Exploring `re.findall()`: A Treasure Hunter in Python's Regex Toolkit

How `re.findall()` works?

This function is particularly useful in data extraction, text analysis, and processing tasks where pattern recurrence is key.

We invite you to watch a collection of videos showing examples of `re.findall()`.

Extracting Email Addresses

1. **Use Case:** Gather all email addresses from a large text.
2. **Regex Pattern:** `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}`
3. **Python Implementation:**

```
import re
text = "Contact us at support@example.com or sales@example.com."
emails = re.findall(r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}", text)
print(emails)
```

4. **Expected Output:** `['support@example.com', 'sales@example.com']`
5. **Explanation:** `re.findall()` extracts every email address in the given text, acting like a magnet pulling out metallic bits (emails) from the sand (text).

Click the following video to view how this works

Finding Hashtags in Social Media Posts

1. **Use Case:** Analyze social media content by extracting hashtags.
2. **Regex Pattern:** `#\w+`
3. **Python Implementation:**

```
import re
post = "Loving the #Python and #Regex learning journey! #coding"
hashtags = re.findall(r"#\w+", post)
print(hashtags)
```

4. **Expected Output:** `['#Python', '#Regex', '#coding']`
5. **Explanation:** Here, `re.findall()` identifies all words prefixed woth `#` - similar to picking out landmarks (hashtags) on a map (post).

Click the following video to view how this works

Finding All Instances of a Pattern

1. **Use Case:** Extract all words that start with “b” and end with “e” from a string.
2. **Regex Pattern:** `\bb\w*e\b`
3. **Test String:** “Write a program to build a bridge but beware of the beehive”
4. **Expected Matches:** ['bridge', 'beware', 'beehive']
5. **Python Implementation:**

```
import re
sentence = “Write a program to build a bridge but beware of the beehive."
words = re.findall(r"\bb\w*e\b", sentence)
Print(words)
```

6. **Explanation:** The pattern `\bb\w*e\b` looks for words that start with “b” and end with “e”. `re.findall()` then retrieves every word in a sentence that fits this criterion, like finding specific types of artifacts in an archeaeological site.

Click the following video to view how this works

Diving Into `re.search()`: The Precise Finder in Python's Regex Toolkit

Think of `re.search()` as a high-precision tool in your text exploration journey, akin to a metal detector that beeps when it finds something interesting. Unlike `re.findall()`, which retrieves all matches, `re.search()` focuses on finding the first occurrence of a pattern within a string. It's perfect for tasks where you need to locate specific information quickly and efficiently.

Validating Email Addresses

1. **Use Case:** Confirm if a string is a valid email address.
2. **Regex Pattern:** `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}`
3. **Python Implementation:**

```
import re
email = "user@example.com"
if re.search (r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}", email):
print("Valid email address.")
else:
print("invalid email address.")
```

4. **Expected Output:** "Valid email address"
5. **Explanation:** `re.search()` checks the entire string to find a match that fits the email pattern, acting like a scanner.

Click the following video to view how this works

Finding Phone Numbers in Text

1. **Use Case:** Locate a phone number in a larger body of text.
2. **Regex Pattern:** `\d{3}-\d{3}-d{4}`
3. **Python Implementation:**

```
import re
text = "Contact us at 123-456-7890."
If match:
print("Phone number found:", match.group())
```

4. **Expected Output:** `Phone number found: 123-456-7890`
5. **Explanation:** `re.search()` scans the text and, upon finding the first phone number pattern, returns the match.

Click the following video to view how this works

Extracting Specific Information from Text

1. **Use Case:** Extract the year from a date string.
2. **Regex Pattern:** `\b\d{4}\b`
3. **Test String:** "The event was held on 15/06/2021"
4. **Expected Matches:** 2021
5. **Python Implementation:**

```python
import re
sentence = "The event was held on 15/06/2021."
match = re.search(r"\b\d{4}\b", sentence)
if match:
print("Year extracted:", match.group())
```

6. **Explanation:** The pattern `\b\d{4}\b` looks for a standalone sequence of four digits (the year). `re.search()` finds first occurrence of this pattern and extracts the year from the sequence.

Click the following video to view how this works

Exploring `re.match()`: The Pattern Matcher at the Start of Strings in Python

Picture `re.match()` as a vigilant guard standing at the gates of your text, only allowing patterns that appear right at the beginning to pass through. This function is specifically designed to check if a regex pattern matches at the start of a string.

Checking for Specific Prefixes

1. **Use Case:** Verify if a string starts with specific word or character.
2. **Regex Pattern:** `^Hello`
3. **Python Implementation:**

```
import re
text = “Hello, World!"
if re.search (r"^Hello", email):
print("The string starts with 'Hello'.")
else:
print(“The string  does not start with 'Hello.")
```

4. **Expected Output:** “The string starts with Hello”
5. **Explanation:** `re.match()` checks if “Hello” is at the beginning of the string and confirms the match.

Click the following video to view how this works

Validating Format

1. **Use Case:** Ensure a string follows a specific format from the start.
2. **Regex Pattern:** `@\w+`
3. **Python Implementation:**

```
import re
text = “@user123"
if re.match (r"@\w+”, username):
print(“Valid username format.")
else:
print(“Invalid username format.")
```

4. **Expected Output:** “Valid username format”
5. **Explanation:** The pattern `@\w+` checks for a username starting with “@” followed by word characters, validating the format.

Click the following video to view how this works

Extracting Specific Information from Text

1. **Use Case:** Extract a protocol type (e.g., “http”, “https”) at the beginning of a URL.
2. **Regex Pattern:** `^(https|http)`
3. **Test String:** “https://www.example.com”
4. **Expected Matches:** “https”
5. **Python Implementation:**

```
import re
url = "https://www.example.com”
match = re.match(r"^(https|http)", url)
if match:
print(“Protocol found:", match.group())
```

6. **Explanation:** The pattern `^(https|http)` looks for “http” or “https” right at the start of the string. `re.match ()` finds “https” as it is at the beginning of the URL.

Click the following video to view how this works

Exploring re.split(): The Pattern-Based String Splitter in Python's Regex

How `re.split()` works?

This function is particularly useful for breaking down data into manageable parts or when dealing with inconsistent delimiters.

We invite you to watch a collection of videos showing examples of re.split().

Splitting a String at Multiple Delimiters

1. **Use Case:** Break a string into words, splitting at spaces, commas, semicolon, periods or hyphens.

2. **Regex Pattern:** `[,;.\s-]+`

3. **Python Implementation:**

```
import re
text = "Python,Regex;Splitting-Examples. Fun, right?"
words = re.split(r"[,;.\s-]+", text)
print(words)
```

4. **Expected Output:** `['Python', 'Regex', 'Splitting', 'Examples', 'Fun', 'right?']`

5. **Explanation:** The pattern `[,;.\s-]+` matches spaces, commas, semicolon, periods or hyphens, splitting the text at any of these characters.

Click the following video to view how this works

Separating Data in a CSV String

1. **Use Case:** Extract individual data items from a CSV (comma-separated values) string.
2. **Regex Pattern:** `,`
3. **Python Implementation:**

```
import re
csv_data = "Name,Age,Occupation"
fields = re.split(r",", csv_data)
print(fields)
```

4. **Expected Output:** `['Name', 'Age', 'Occupation']`
5. **Explanation:** The pattern `,` is used to split the CSV string at each comma, turning it into a list of data fields.

Click the following video to view how this works

Splitting at Words and Punctuation

1. **Use Case:** Separate a text into words and punctuation marks.

2. **Regex Pattern:** `\W+` (splits at any sequence of non-word characters)

3. **Python Implementation:**

```
import re
sentence = "Hello, world! Welcome to Python."
parts = re.split(r"(\W+)", sentence)
print(parts)
```

4. **Expected Output:** `['Hello', ', ', 'world', '! ', 'Welcome', ' ', 'to', ' ', 'Python', '.']`

5. **Explanation:** The pattern `\W+` is used to split the sentence at non-word characters like spaces, commas, and exclamation points. The inclusion of parentheses in the pattern captures the delimiters as well, keeping the punctuation as separate elements in the resulting list.

Click the following video to view how this works

Mastering re.sub(): The Art of Text Transformation in Python's Regex

This function is a powerful tool for modifying and cleaning up strings, allowing for both simple replacements and more complex, pattern-based transformations.

We invite you to watch a collection of videos showing examples of `re.sub()`

Text Formatting: Standardizing Phone Numbers

1. **Use Case:** Format a variety of phone number formats into a standard format.
2. **Regex Pattern:** `D`
3. **Replacement Pattern:** An empty string to remove non-digit characters.
4. **Python Implementation:**

```
import re
phone = "Phone: +1 (123) 456-7890"
standard_phone = re.sub(r"\D", "", phone)
print(standard_phone)
```

5. **Expected Output:** "11234567890"
6. **Explanation:** `re.sub()` removes all non-digit characters from the phone number, standardizing it to a simple digit sequence.

Click the following video to view how this works

Data Cleaning: Removing HTML Tags

1. **Use Case:** Clean a string by removing HTML tags.
2. **Regex Pattern:** `<.*?>`
3. **Replacement Pattern:** An empty string to remove HTML tags.
4. **Python Implementation:**

```
import re
html = "<p>This is <em>HTML</em> content!</p>"
clean_text = re.sub(r"<.*?>", "", html)
print(clean_text)
```

5. **Expected Output:** "This is HTML content!"
6. **Explanation:** The pattern `<.*?>` matches any HTML tag, and `re.sub()` replaces these tags with an empty string, effectively removing them.

Click the following video to view how this works

Formatting Text Data

1. **Use Case:** Convert dates from 'dd/mm/yyyy' to 'yyyy-mm-dd' format.
2. **Regex Pattern:** `(\d{2})/(\d{2})/(\d{4})`
3. **Replacement Pattern:** `r"\3-\2-\1"`
4. **Test String:** "Today's date is 15/04/2021."
5. **Python Implementation:**

```python
import re
date_string = "Today's date is 15/04/2021."
formatted_date = re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\3-\2-\1", date_string)
print(formatted_date)
```

6. **Expected Output:** "Today's date is 2021-04-15."
7. **Explanation:** The pattern captures the day, month, and year in groups, and `re.sub()` rearranges these groups into the desired format.

Click the following video to view how this works

Real-World Applications of Regular Expressions in Python

The function regex in Python opens up a world of possibilities from extracting valuable information to validating and transforming data.

Let’s dive into some real-world applications, complete with Python code examples, to see how regex can be your ally in various text-processing tasks.

Real-World Applications of Regular Expressions in Python

Extracting Data from Files or Web Scraping

Regex helps you find the exact pieces of information you need.

Let’s navigate through a couple of videos showing examples of this.

Let's check the next video

Real-World Applications of Regular Expressions in Python

Log File Analysis

- Think of regex as a filter that sifts through the dense jungle of log entries to find the events of interest.
- **Example**: Identifying error messages in a log file.

Click on the next video

Real-World Applications of Regular Expressions in Python

Data Validation

- Regex serves as a checkpoint, ensuring the data conforms to specific patterns.

Let’s navigate through a couple of videos showing examples of this.

Real-World Applications of Regular Expressions in Python

Password Strength Checking

- Like testing the strength of a fortress's walls, regex can ensure passwords meet certain criteria.
- **Example**: Checking password strength.

Click on the next video!

Real-World Applications of Regular Expressions in Python

Search and replace in Text Editing

- Regex acts as an automated editor, swiftly modifying the text to fit our needs.

To gain a more comprehensive insight, consider watching this video.

Real-World Applications of Regular Expressions in Python

Integrating with Functions for Dynamic Text Processing

Combine regex with Python functions for responsive text processing applications.

To gain a more comprehensive insight, consider watching this video.

Exercise 1: E-Commerce Product Data Extraction

Objective:
To develop a Python script that efficiently extracts and organizes product information from a text data source, replicating a common task in e-commerce data management.

Problem Statement:
In an e-commerce setting, you are faced with a large text file containing detailed product information. The challenge is to parse this semi-structured data to extract relevant details such as product IDs, names, categories, and prices, and then store this information for easy access and analysis.

Instructions:

1. Read a string that contains product information (simulating data from a file).
2. Implement a regex pattern to identify and extract the necessary product details.
3. Store and organize the extracted data into a structured format, like a list of dictionaries.
4. Incorporate error handling for potential data parsing issues.
5. Display the organized data in a readable format for end-users.

Hints:

- Utilize `re.findall()` for pattern-based data extraction.
- Craft a regex pattern tailored to the specific data format of the product details.
- Use dictionaries for storing individual product data, and a list to hold all products.
- Implement loops to process and store each extracted product.
- Apply try-except blocks to gracefully handle any parsing errors.

Exercise 2: Marketing Campaign Email Validator

Objective:
Develop a Python script to validate a list of email addresses for a marketing campaign, ensuring they adhere to a standard email format.

Problem Statement:
You are tasked with preparing an email list for an upcoming marketing campaign. The list contains various email addresses, some of which may not be in the correct format. Your goal is to validate these email addresses, filter out invalid ones, and compile a list of valid email addresses for the campaign.

Instructions:

1. Read a list of email addresses (simulated through a predefined list).
2. Implement a regex pattern to validate the email addresses.
3. Store valid email addresses in one list and invalid ones in another.
4. Handle any errors that might occur during validation.
5. Display both lists of valid and invalid email addresses.

Hints:

- Use `re.match()` with an appropriate regex pattern to validate each email address.
- Iterate through the list of emails, applying the validation regex to each.
- Store email addresses in separate lists based on their validity.
- Use try-except blocks to handle exceptions during the validation process.

Exercise 3: Retail Inventory Categorization

Objective:
Create a Python script to categorize products in a retail inventory based on their codes, using regex to identify different product categories.

Problem Statement:
You are managing a retail inventory system that includes a diverse range of products, each identified by a unique code. These codes follow a specific pattern, indicating the product category. Your task is to categorize these products into different groups based on their codes for better inventory management.

Instructions:

1. Work with a provided list of product codes.
2. Develop regex patterns to identify different product categories (e.g., Electronics, Clothing, Groceries).
3. Categorize each product into the appropriate group based on its code.
4. Handle any exceptions or errors in categorization.
5. Display the categorized list of products, grouped by their categories.

Hints:

- Define specific regex patterns for each category based on the code structure.
- Iterate through the list of product codes, using regex to determine each product's category.
- Use a dictionary to store categorized products, with category names as keys and lists of product codes as values.
- Implement try-except blocks for robust error handling during the categorization process.

Exercise 4: Social Media Sentiment Analysis

Objective:
Develop a Python script to perform basic sentiment analysis on social media comments, categorizing them as positive, negative, or neutral based on specific keywords using regex.

Problem Statement:
You're working on a project that involves analyzing social media comments to gauge public sentiment. The task is to automatically categorize each comment as positive, negative, or neutral based on the presence of certain keywords indicative of the sentiment.

Instructions:

1. Utilize a predefined list of social media comments for analysis.
2. Create regex patterns for positive, negative, and neutral keywords or phrases.
3. Analyze each comment and categorize it according to the identified sentiment.
4. Handle exceptions or edge cases in the categorization process.
5. Output the categorized comments, showing the distribution of sentiments.

Hints:

- Define separate regex patterns for positive, negative, and neutral sentiments.
- Use a loop to process each comment, applying regex to identify the sentiment.
- Store the categorized comments in a dictionary or separate lists.
- Implement try-except blocks for handling ambiguous or unclear comments.

Exercise 5: Customer Feedback Analysis for a Restaurant

Objective:
Create a Python script to analyze customer feedback for a restaurant, categorizing comments into feedback about food, service, or ambiance using regex.

Problem Statement:
Your restaurant has collected a series of customer feedback comments. The goal is to categorize these comments to better understand customer opinions about different aspects of the restaurant: food, service, and ambiance.

Instructions:

1. Utilize a provided list of customer feedback comments.
2. Develop regex patterns to identify mentions of food, service, and ambiance.
3. Analyze each comment and categorize it into the appropriate feedback category.
4. Implement error handling for any issues in the categorization process.
5. Display the results, showing the number of comments in each category.

Hints:

- Define distinct regex patterns for keywords related to food, service, and ambiance.
- Iterate through the feedback list, applying regex to categorize each comment.
- Use a dictionary to store the count of comments in each category.
- Employ try-except blocks to manage any exceptions during processing.

Exercise 6: Library Book Sorting System

Objective:
Develop a Python script to sort a list of library book titles into different genres based on specific keywords in their titles using regex.

Problem Statement:
You're assisting a library in organizing their books more efficiently. The library has a list of book titles, each belonging to different genres like Fiction, Non-Fiction, Science, and History. Your task is to categorize these books into their respective genres based on keywords in their titles.

Instructions:

1. Work with a predefined list of book titles.
2. Create regex patterns to identify keywords associated with each genre.
3. Sort each book into the appropriate genre category based on its title.
4. Handle any exceptions or ambiguous cases in categorization.
5. Display the sorted list of books, categorized by genre.

Hints:

- Define distinct regex patterns for each genre based on title keywords.
- Use a loop to process each book title, applying regex to determine its genre.
- Store the sorted books in a dictionary with genres as keys and lists of titles as values.
- Implement try-except blocks to handle titles that don't clearly fit into any category.

Exercise 7: Product Code Validation System

Objective:
Create a Python script to validate a list of product codes based on specific formatting rules, using the `re.match()` function in regex.

Problem Statement:
In a warehouse management system, product codes follow a strict format that combines letters and numbers. Your task is to validate a list of product codes, ensuring they adhere to the format 'ABC-1234', where 'ABC' represents a series of letters and '1234' represents a series of numbers.

Instructions:

1. Start with a provided list of product codes.
2. Implement a regex pattern using `re.match()` to validate each code's format.
3. Categorize and store codes into 'valid' and 'invalid' groups based on the validation.
4. Implement error handling for any anomalies encountered during validation.
5. Display the results, listing both valid and invalid product codes.

Hints:

- Develop a regex pattern that matches the specific format of the product codes.
- Use a loop to process each code in the list, applying the `re.match()` function for validation.
- Store validated codes in two separate lists: one for valid codes and one for invalid codes.
- Use try-except blocks to ensure robust processing of each code.

Exercise 8: Conference Badge Registration System

Objective:
Develop a Python script to validate attendee registration details for a conference, focusing on the correctness of badge IDs using the `re.match()` function.

Problem Statement:
You are helping to manage a professional conference. Each attendee has a badge ID, which follows a specific format: two uppercase letters, a dash, and three digits (e.g., 'AB-123'). Your task is to validate a list of badge IDs, ensuring they comply with this format.

Hints:

1. Start with a list of badge IDs provided by attendees.
2. Create a regex pattern using `re.match()` to validate the format of each badge ID.
3. Separate the badge IDs into 'valid' and 'invalid' categories based on their format.
4. Implement error handling for any issues encountered during the validation process.
5. Display the categorized badge IDs, showing which are valid and which are invalid.

Hints:

- Construct a regex pattern that matches the specific badge ID format.
- Iterate through the list of badge IDs, using `re.match()` to validate each ID.
- Use two separate lists to store and differentiate valid and invalid badge IDs.
- Apply try-except blocks for robust processing and error handling.

Exercise 9: Customer Inquiry Response Categorization

Objective:
Create a Python script to categorize customer inquiries from a single text file into different departments using the `re.split()` function.

Problem Statement:
You are tasked with organizing a large volume of customer inquiries that have been compiled into a single text file. Each inquiry is separated by a special delimiter, "#END#". The inquiries need to be categorized into different departments (Sales, Support, and Billing) based on specific keywords in the text.

Instructions:

1. Read a string representing the compiled customer inquiries.
2. Utilize `re.split()` to separate individual inquiries using the delimiter.
3. Categorize each inquiry based on keywords: 'purchase' for Sales, 'help' for Support, and 'invoice' for Billing.
4. Implement error handling for categorization and splitting.
5. Display the inquiries categorized under each department.

Hints:

- Create a regex pattern to split the text into separate inquiries using the "#END#" delimiter.
- Analyze each inquiry using regex to identify department-specific keywords.
- Use dictionaries or lists to store inquiries categorized by department.
- Employ try-except blocks to ensure robust processing of each inquiry.

Exercise 10: Formatting Log Files for Analysis

Objective:
Develop a Python script to reformat entries in a log file for easier analysis, using the `re.sub()` function to standardize date formats and anonymize sensitive information.

Problem Statement:
You have a log file from a web server that contains various entries with timestamps and user emails. The timestamps are in different formats, and you need to standardize them. Additionally, for privacy concerns, you need to anonymize user email addresses in the log.

Instructions:

1. Read a string that represents log file entries.
2. Use `re.sub()` to standardize timestamp formats (e.g., convert "14/03/2022" to "2022-03-14").
3. Use `re.sub()` to replace email addresses with a placeholder text like "[ANONYMIZED]".
4. Handle any exceptions that might occur during the process.
5. Display the reformatted log entries.

Hints:

- Develop regex patterns to identify different timestamp formats and email addresses.
- Use `re.sub()` to replace identified patterns with standardized formats or placeholder text.
- Employ try-except blocks to manage any anomalies in the log entries.

Wrapping Up: Embracing the Power of Regex in Python

On this journey, we've navigated through these topics:

- The purpose and intricacies of regex, diving into the depths of regex syntax, metacharacters, special sequences, and sets.
- Functionalities of the re module, our journey has been rich and enlightening.
- The functions of re.findall(), re.search(), re.match(), re.split(), and re.sub().
- Best practices for crafting and debugging regex patterns and addressing common pitfalls to avoid.
- How regex comes to life in real-world applications, from extracting and validating data to performing sophisticated search and replace operations in text editing.

As you continue your Python journey, let the knowledge and skills you've gained in regex be a foundation that not only enhances your capabilities but also inspires further exploration and discovery in the vast and exciting world of programming.

Happy coding, and may each line of regex you write bring you closer to the solutions and insights you seek! 🌟🐍🔍