Regular Expression Basics: Takeaways

Syntax

REGULAR EXPRESSION MODULE

- Importing the regular expression module:

```
import re
```

- Searching a string for a regex pattern:

```
re.search(r"blue", "Rhythm and blues")
```

PANDAS REGEX METHODS

- Return a boolean mask if a regex pattern is found in a series:

```
s.str.contains(pattern)
```

- Extract a regex capture group from a series:

```
s.str.extract(pattern_with_capture_group)
```

ESCAPING CHARACTERS

- Treating special characters as ordinary text using backslashes:

```
\[pdf\]
```

Concepts

- Regular expressions, often referred to as regex, are a set of syntax components used for matching sequences of characters in strings.
- A pattern is described as a regular expression that we've written. We say regular expression has matched if it finds the pattern exists in the string.
- Character classes allow us to match certain classes of characters.
- A set contains two or more characters that can match in a single character's position.
- Quantifiers specify how many of the previous characters the pattern requires.
- Capture groups allow us to specify one or more groups within our match that we can access separately.
- Negative character classes are character classes that match every character except a character class.
- An anchor matches something that isn't a character, as opposed to character classes which match specific characters.
- A word boundary matches the space between a word character and a non-word character, or a word character and the start/end of a string
- Common character classes:

Character Class	Pattern	Explanation
Set	`[fud]`	Either `f`, `u`, or `d`
Range	`[a-e]`	Any of the characters `a`, `b`, `c`, `d`, or `e`
Range	`[0-3]`	Any of the characters `0`, `1`, `2`, or `3`
Range	`[A-Z]`	Any uppercase letter
Set + Range	`[A-Za-z]`	Any uppercase or lowercase character
Digit	`\d`	Any digit character (equivalent to `[0-9]`)
Word	`\w`	Any digit, uppercase, or lowercase character (equivalent to `[A-Za-z0-9]`)
Whitespace	`\s`	Any space, tab or linebreak character
Dot	`.`	Any character except newline

- Common quantifiers:

Quantifier	Pattern	Explanation
Zero or more	`a*`	The character `a` zero or more times
One or more	`a+`	The character `a` one or more times
Optional	`a?`	The character `a` zero or one times
Numeric	`a{3}`	The character a three times
Numeric	`a{3,5}`	The character `a` three, four, or five times
Numeric	`a{,3}`	The character `a` one, two, or three times
Numeric	`a{8,}`	The character `a` eight or more times

- Common negative character classes:

Character Class	Pattern	Explanation
Negative Set	`[^fud]`	Any character except `f`, `u`, or `d`
Negative Set	`[^1-3Z\s]`	Any characters except `1`, `2`, `3`, `Z`, or whitespace characters
Negative Digit	`\D`	Any character except digit characters
Negative Word	`\W`	Any character except word characters
Negative Whitespace	`\S`	Any character except whitespace characters

- Common anchors:

Anchor	Pattern	Explanation
Beginning	`^abc`	Matches `abc` only at the start of a string
End	`abc$`	Matches `abc` only at the end of a string
Word boundary	`s\b`	Matches `s` only when it's followed by a word boundary
Word boundary	`s\B`	Matches `s` only when it's not followed by a word boundary

Resources

- re module
- Building regular expressions

Advanced Regular Expressions: Takeaways

Syntax

CAPTURE GROUPS

- Extracting text using a capture group:

```
s.str.extract(pattern_with_capture_group)
```

- Extracting text using multiple capture groups:

```
s.str.extract(pattern_with_multiple_capture_groups)
```

SUBSTITUTION

- Substituting a regex match:

```
s.str.replace(pattern, replacement_text)
```

Concepts

- Capture groups allow us to specify one or more groups within our match that we can access separately.

Pattern	Explanation
`(yes)no`	Matches `yesno`, capturing `yes` in a single capture group.
`(yes)(no)`	Matches `yesno`, capturing `yes` and `no` in two capture groups.

- Backreferences allow us to repeat a capture group within our regex pattern by referring to them with an integer in the order they are captured.

Pattern	Explanation
`(yes)no\1`	Matches `yesnoyes`
`(yes)(no)\2\1`	Matches `yesnonoyes`

- Lookarounds let us define a positive or negative match before or after our string.

Pattern	Explanation
`zzz(?=abc)`	Matches `zzz` only when it is followed by `abc`
`zzz(?!abc)`	Matches `zzz` only when it is not followed by `abc`
`(?<=abc)zzz`	Matches `zzz` only when it is preceded by `abc`
`(?<!abc)zzz`	Matches `zzz` only when it is not preceded by `abc`

Resources

- re module
- RegExr Regular Expression Builder

List Comprehensions and Lambda Functions: Takeaways

Syntax

WORKING WITH JSON FILES

- Open a JSON data set from a file to Python objects:

```
f = open('filename.json')
json.load(f)
```

- Convert JSON data from a string to Python objects:

```
json.loads(json_string)
```

- Convert JSON data stored in Python objects to string form:

```
json.dumps(json_obj)
```

LIST COMPREHENSIONS

- Converting a for loop to a list comprehension:
  - Using a for loop:

```
letters=['a', 'b', 'c', 'd']
caps=[]
for l in letters:
    caps.append(l.upper())
```

  - Using a List comprehension:

```
caps = [l.upper() for l in letters]
```

- Common list comprehension patterns:
  - Transforming a list

```
ints = [25, 14, 13, 84, 43, 6, 77, 56]
doubled_ints = [i * 2 for i in ints]
```

  - Creating test data

```
tenths = [i/10 for i in range(5)]
```

  - Reducing a list

```
big_ints = [i for i in ints if i >= 50]
```

LAMBDA FUNCTIONS

- Converting a definition to a lambda function:
  - Defining a function:

```
def double(x):
    return x * 2
```

- Defining a lambda function:

```
run_function(function=lambda x: x * 2)
```

THE TERNARY OPERATOR

- Create a one-line version of an if/else statement:

```
"val_1 is bigger" if val_1 > val_2 else "val_1 is not bigger"
```

Concepts

- JSON is a language independent format for storying structured data.
    - In Python, it can be represented by a series of nested lists, dictionaries, strings, and numeric objects.
- A list comprehension provides a concise way of creating lists using a single line of code, where:
    - You start with an iterable object
    - Optionally Transform the items in the iterable object
    - Optionally reduce the items in the iterable object using an if statement
    - Create a new list
- Lambda functions can be defined in a single line, which lets you define a function at the time you need it.
- The ternary operator can be used to replace an if/else statement with a single line.

Resources

- Official JSON specification
- Python Documentation: JSON Module
- Python Documentation: List Comprehensions
- Python Documentation: Lambda Functions

Working with Missing Data: Takeaways

Syntax

- Replacing matching values with a single value:

```
s.mask(s == var, value_to_replace)
```

- Replacing matching values with corresponding values from a series:

```
s1.mask(s == var, series_to_replace)
```

- A function to create a null matrix

```
def plot_null_matrix(df, figsize=(18,15)):
    # initiate the figure
    plt.figure(figsize=figsize)
    # create a boolean dataframe based on whether values are null
    df_null = df.isnull()
    # create a heatmap of the boolean dataframe
    sns.heatmap(~df_null, cbar=False, yticklabels=False)
    plt.show()
```

- A function to create a null correlation heatmap

```
def plot_null_correlations(df):
    # create a correlation matrix only for columns with at least
    # one missing value
    cols_with_missing_vals = df.columns[df.isnull().sum() > 0]
    missing_corr = df[cols_with_missing_vals].isnull().corr()
    # create a triangular mask to avoid repeated values and make
    # the plot easier to read
    missing_corr = missing_corr.iloc[1:, :-1]
    mask = np.triu(np.ones_like(missing_corr), k=1)
    # plot a heatmap of the values
    plt.figure(figsize=(20,12))
    ax = sns.heatmap(missing_corr, vmin=-1, vmax=1,
                     cmap='RdBu', mask=mask, annot=True)
    # round the labels and hide labels for values near zero
    for text in ax.texts:
        t = float(text.get_text())
        if -0.05 < t < 0.01:
            text.set_text('')
        else:
            text.set_text(round(t, 2))
    plt.show()
```

Concepts

- Imputation is the process of replacing missing values with other values.

- Imputing can be a better option than simply dropping values because you retain more of your original data.
- You might find values for imputation by:
    - Deriving the value from related columns.
    - Using the most common non-null value from a column.
    - Using an placeholder for missing values.
    - Augmenting factual data (e.g. location data) using an external resource.
- Using plots can help identify patterns in missing values which can help with imputation.

Resources

- pandas documentation