Overview

Tuesday, May 28, 2024 4:47 AM

1. **Intro**
2. ***re* Module**
3. **Searching Operations**
4. **Meta Characters**
5. **Character Sets**
6. **Quantifiers**
7. **Grouping**
8. **String Modifications**
9. **Look-ahead and Look-behind Assertions**
10. **Flags**
11. **Practice Exercises**

Introduction to Regular Expressions

Tuesday, May 28, 2024 4:50 AM

1. What are Regular Expressions?

- Regular expressions, often abbreviated as *regex*, are sequences of characters that define text patterns.
- They can be used to represent any sort of text data as a pattern.
- Regular expressions are powerful tools in programming and text processing, and they are supported by many programming languages, including Python, JavaScript, Perl, and others.

2. Why Regular Expressions?

- Text Searching (*find* operations)
- Text Replacement (*search & find* operations)
- String Input Validation (emails, phone numbers, address, etc.)
- Parsing text from formatted data (HTML, log files, etc.)

3. What are Raw Strings?

- In python, we can denote a string as a 'raw string' by prefixing a regular string with 'r'
- A raw string treats every character literally, including the *escape characters*

re Module

Tuesday, May 28, 2024 5:10 AM

This is a built-in module in Python that supports the usage of regular expressions.

Provides various functions for defining and using regular expressions for convenient handling of textual data.

Involves working with *regex* objects for defining patterns and parsing results.

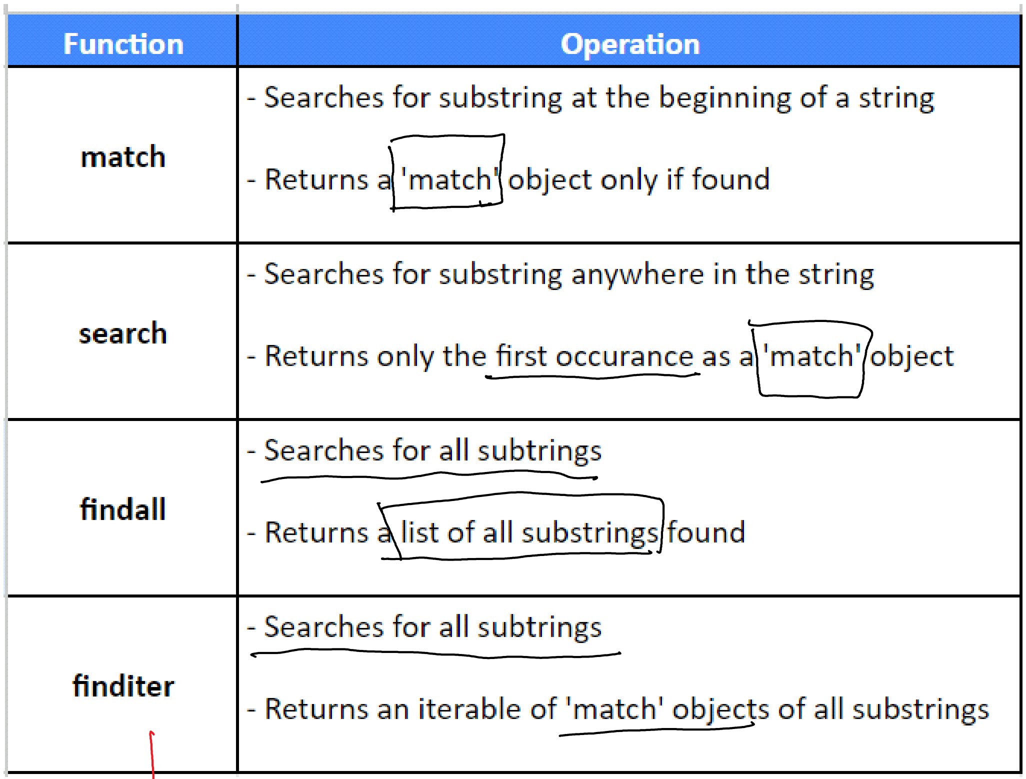
Most commonly used functions:

- **compile** - *converts a pattern into a regular expression object, memory-efficient when patterns are to be reused*
- **match**
- **search** } → Searching
- **finditer**
- **findall**
- **sub** → search & replace
- **split** → Splitting

Searching Operations

Wednesday, May 29, 2024 4:14 AM

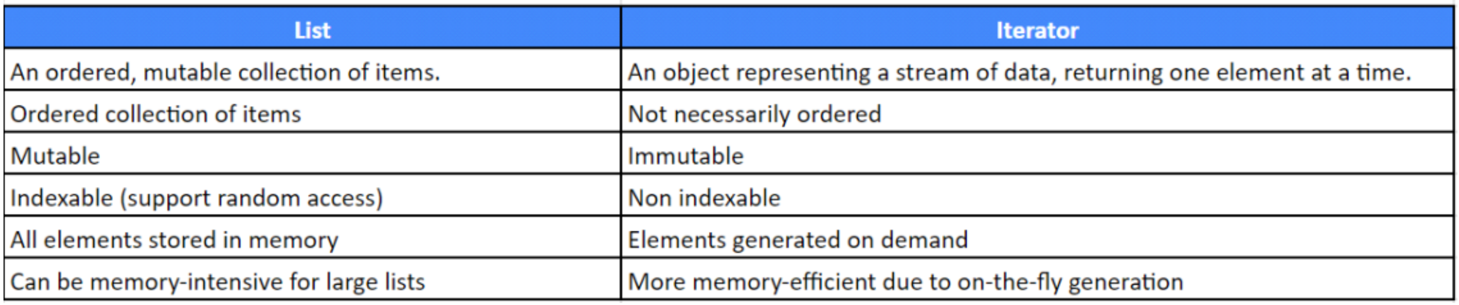
The re module provides 4 functions to perform searching operations over a string:



Function	Operation
match	- Searches for substring at the beginning of a string - Returns a 'match' object only if found
search	- Searches for substring anywhere in the string - Returns only the first occurance as a 'match' object
findall	- Searches for all subtrings - Returns a list of all substrings found
finditer	- Searches for all subtrings - Returns an iterable of 'match' objects of all substrings

→ Iterator of 'match' objects

List vs Iterator:



List	Iterator
An ordered, mutable collection of items.	An object representing a stream of data, returning one element at a time.
Ordered collection of items	Not necessarily ordered
Mutable	Immutable
Indexable (support random access)	Non indexable
All elements stored in memory	Elements generated on demand
Can be memory-intensive for large lists	More memory-efficient due to on-the-fly generation

Meta Characters

Wednesday, May 29, 2024 4:24 AM

- These are special characters, with each one having a unique meaning in the context of string matching
- Help in creating complex patterns conveniently with less code

. - Any character (except newline character)
^ - Beginning of a string
$ - Ending of a string
* - Zero or more occurrences
+ - One or more occurrences
{ } - Exactly the specified number of occurrences
[] - A set of characters
\ - Signals a special sequence (can also be used to escape special characters)
|- Either Or
( ) - Capture and group

Character Sets

Wednesday, May 29, 2024 4:41 AM

- A character set in regular expressions refers to a combination of characters that can be used complex and flexible pattern matching.
- Typically, they are defined using square brackets [] and allow you to match any one of the characters within the brackets.
- Character sets can be ***user-defined*** as well as ***pre-defined***.

1. Custom Character Sets: user-defined

- Includes a bunch of characters specified within square brackets []
- The complexity of such sets is only limited by the user's requirement
- By default, any single character from the specified set will be matched. However, this behaviour can be modified to match an arbitrary number of characters as well.

2. Pre-defined Character Sets: Character Classes (pre-defined)

- These are special characters followed by a '\'
- Each set holds a unique meaning and is used for matching a particular set of characters
- These sets provide a short-hand for matching common types of characters

\d: Matches any digit; equivalent to **[0-9]**
\D: Matches any non-digit character; equivalent to **[^0-9]**

\s: Matches any whitespace character (space, tab, newline)
\S: Matches any non-whitespace character

\w: Matches any alphanumeric character; equivalent to **[a-zA-Z0-9_]**
\W: Matches any non-alphanumeric character; equivalent to **[^a-zA-Z0-9_]**

\b: Matches any whitespace or non-alphanumeric character before or after character(s) (*useful for identifying individual words in a string*)
\B: Negation of \b

\b ⇒ white space or non-alphanum-char

\\ ho <u></u>hey ✓

✗

Quantifiers

Wednesday, May 29, 2024 4:46 AM

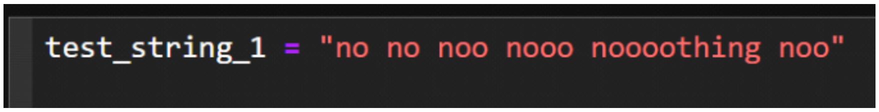
- Quantifiers in regular expressions (regex) specify the number of times that a character, group, or character class must occur to make a match.
- They are used to define the permissible number of repetitions for the preceding element.
- Can be categorized as ***greedy*** and ***non-greedy***

1. **Greedy Quantifiers:** Try to match as many characters as possible

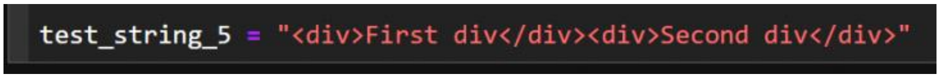
* : 0 or more occurrences of preceding element noo*
+ : 1 or more occurrences of preceding element
? : 0 or 1, used when a character can be optional
{m} : exactly 'm' characters
{m, n} : range of characters (m, n)

2. **Non-greedy (lazy) Quantifiers:** Try to match as few characters as possible

*? : 0 or more
+? : 1 or more
?? : 0 or 1, used when a character can be optional (*as few as possible*)
{m}? : exactly 'm' characters (*as few as possible*)
{m, n}? : range of characters (m, n) (*as few as possible*)



```
test_string_1 = "no no noo nooo noooothing noo"
```



```
test_string_5 = "<div>First div</div><div>Second div</div>"
```

```
test_string_5 = "<div>First div</div><div>Second div</div>"
```

Grouping

Wednesday, May 29, 2024 5:13 AM

- Used for identifying group(s) of matching substrings within a larger string
- Grouping is useful for extracting specific parts of a string which could provide useful information
- Characters to form a group are mentioned within parentheses ()
- The captured groups are stored for 'later use'
- Groups allow the usage of ***back-references***

1. **Capture Groups:**
    - By default, all groups are 'Capture Groups' until explicitly altered
    - By default, all identified groups are assigned ***integral*** names
    - **Syntax:** (pattern)

2. **Named Capture Groups:**
    - Behaves similarly like Capture Groups
    - Each captured group can be given a name explicitly
    - This improves code readability and group access
    - **Syntax:** (?P<group_name>pattern)

3. **Back-references:**
    - Used for referencing **captured groups** by short-hand notation
    - Mainly used when some parts of the pattern repeat

- **Syntax (default capture groups):** \group_index
- **Syntax (named capture groups):** (?P=group_name)

4. Non-capture Groups:

- The 'groups' aren’t captured for later use
- **Syntax:** (?:pattern)

5. Alternation:

- Allows to match any pattern from listed alternatives
- Implemented by using the **|** (pipe) symbol

Modifications

Wednesday, May 29, 2024 5:25 AM

- Involves splitting a string or replacing parts of a string
- The re module provides the ***split*** and ***sub*** functions to achieve this

1. Split:

- Allows to split a string on any matched pattern
- Works similar to Python's ***str.split*** function, with the additional flexibility of regular expressions
- **Syntax:** re.split(pattern, string, maxsplit=0)

2. Substitution:

- Allows to replace a part of a string with any matched pattern
- Useful for search and replace operations with the added flexibility of regular expressions
- **Syntax:** re.sub(pattern, replacement, string, count=1)

Lookahead and Lookbehind Assertions

Friday, May 31, 2024 5:54 AM

- Lookahead and lookbehind assertions are powerful tools in regular expressions that allow for complex pattern matching based on the context in which a pattern appears
- They are used to match a pattern only if it is followed or preceded by another specified pattern

1. Lookahead Assertion:

1.1 Positive lookahead assertion:

- Asserts if a pattern to be matched (X) is immediately followed by another specified pattern (Y)
- **Syntax**: X(?=Y)

1.2 Negative lookahead assertion:

- Asserts if a pattern to be matched (X) is not immediately followed by another specified pattern (Y)
- **Syntax**: X(?!Y)

2. Lookbehind Assertion:

2.1 Positive lookbehind assertion:

- Asserts if a pattern to be matched (X) is immediately preceded by another specified pattern (Y)
- **Syntax**: (?<=Y)X

2.2 Negative lookbehind assertion:

- Asserts if a pattern to be matched (X) is not immediately preceded by another specified pattern (Y)

- **Syntax**: (?<!Y)X

Flags

Wednesday, May 29, 2024 5:45 AM

- Flags provide additional control over pattern matching by altering the behaviour of regular expressions
- Can be used for insensitive case matching, making the dot operator match newline character, etc.
- Flags are usually passed as arguments to the functions of re module

Common Flags:

- **re.IGNORECASE (or re.I)**: Makes the pattern case-insensitive
- **re.MULTILINE (or re.M)**: Allows ^ and $ to match the start and end of each line
- **re.DOTALL (or re.S)**: Allows the . to match newline characters as well
- **re.VERBOSE (or re.X)**: Allows you to write more readable regex by ignoring whitespace and comments within the pattern
- **re.ASCII (or re.A)**: Makes \w, \b, \d, and \s match only ASCII characters
- **re.LOCALE (or re.L)**: Makes \w, \b, \d, and \s dependent on the current locale

Exercises

Saturday, June 1, 2024 3:18 PM