EXPLORE AI ACADEMY

Advanced string manipulation

Regular expressions

Introduction

A **regular expression**, commonly referred to as a "**regex**", is a sequence of characters used to **define a search pattern**. It's a powerful tool in Python for handling **strings**.

Why learn regex?

Regex in Python is essential for **finding** or **matching patterns in text**, a common requirement in data processing.

It enables **complex** text transformations and processing with **minimal code**, making tasks more **efficient** and **less error-prone**.

Regex is widely used for **parsing text** data and **extracting information** like emails, phone numbers, or specific formats.

Python regex can handle **large** volumes of data, making it suitable for **big data processing** and **automation** tasks.

Literal characters

> Literal characters are the **simplest form of pattern matching**. In regex, most characters, including all **letters** and **numbers**, are literal characters.

Basics

When we use a literal character in a regex pattern, the engine looks for an **exact match** of that character in the search string.

For example, the regex **a** will match the character `'a'` in the string "**cat**", but not `'c'` or `'t'`.

Case sensitivity

Regular expressions are **case sensitive by default**. This means that a and A are considered different characters.

For example, the regex **cat** will match "**cat**" but not "**Cat**" or "**CAT**".

Non-meta characters

Apart from a few special characters (like **. * ?** etc.), **most characters** are considered literal in regular expressions.

For example, characters like **b**, **X**, **5**, or **=** are literal characters.

Combining with meta characters

Literal characters can be combined with **meta characters** to form **more complex patterns**.

For example, **ca+t** will match `'cat'`, `'caat'`, `'caaat'`, etc., where `'a'` is a literal character and **+** is a meta character.

Meta characters and character classes

> Meta characters are characters that have a **special meaning**, **distinct from their literal interpretation** while character classes are used to create patterns that can match **a wider range of characters than a single character alone**.

Core meta characters

- **`. (Dot)`**: Represents **any character** (except newline characters).
  For example, **`a.b`** can match "**acb**", "**aab**", "**apb**", etc.
- **`^ (Caret)`**: Matches the **start of a string**. If used within a character class, it **negates the class**.
  For instance, **`^a`** matches "**a**" at the beginning of a string, while **`[^a]`** matches any character that is not "**a**".
- **`$ (Dollar)`**: Matches the **end of a string**.
  For example, a$ will match the **`"a"`** at the end of "**`lava`**".
- **`\ (Backslash)`**: Used to **escape meta characters** or to **signify a special sequence**.
  For example, **`\.`** matches a literal dot, and **`\d`** matches any digit.

Core character classes

- **`\d`**: Matches any decimal digit; equivalent to `[0-9]`.
- **`\D`**: Matches any non-digit character; equivalent to `[^0-9]`.
- **`\w`** Matches any word character (including digits and underscores); equivalent to `[a-zA-Z0-9_]`.
- **`\W`**: Matches any non-word character; opposite of **`\w`**.
- **`\s`**: Matches any whitespace character (including space, tab, newline).
- **`\S`**: Matches any non-whitespace character.

Quantifiers and repetitions

> Quantifiers and repetitions are fundamental components of regex that allow us to specify **how many times** a particular pattern **should be matched**.

Basic quantifiers

- **`*` (Asterisk)**: Matches the preceding element zero or more times.
  For example, ab* will match `'a'`, `'ab'`, `'abb'`, `'abbb'`, etc.
- **`+` (Plus)**: Matches the preceding element one or more times.
  For instance, ab+ will match `'ab'`, `'abb'`, `'abbb'`, etc., but not `'a'`.
- **`?` (Question mark)**: Makes the preceding element optional, matching either once or not at all.
  For example, ab? will match `'a'` or `'ab'`.

Specific quantifiers

`{}` (Curley): Used to specify an exact number or range of repetitions:

- **`{n}` (Exact number)**: The preceding element is matched exactly **`n`** times.
  For instance, a{3} will match `'aaa'` only.
- **`{n,m}` (Range)**: Matches the preceding element at least **`n`** times, but not more than **`m`** times.
  For example, a{2,4} will match `'aa'`, `'aaa'`, and `'aaaa'`.
- **`{n,}` (At least n)**: Matches the preceding element at least **`n`** times, but not more than **`m`** times.
  For example, a{2,} will match `'aa'`, `'aaa'`, `'aaaa'`, etc.

Quantifiers and repetitions

> Quantifiers and repetitions are fundamental components of regex that allow us to specify **how many times** a particular pattern **should be matched**.

Greedy vs. lazy matching

- Regular expressions are by default **greedy**, meaning they **match as many occurrences as possible**.
  - Example: In **aabbbb**, **ab*** will match **'abbbb'**.
- To make quantifiers **lazy** (non-greedy), use **?** after the quantifier. This will **match as few occurrences as possible**.
  - Example: In **aabbbb**, **ab*?** will match **'a'**.

- Overuse of broad quantifiers like * and + can lead to unexpected matches ("greedy" behavior).
- It's important to test regex patterns on diverse data samples to ensure accuracy.

Try it yourself

```
import re

text = "The quick brown fox jumps over 12 lazy dogs."

# Find any word of at least 4 characters
# {4,} is a greedy match
print(re.findall(r"\b\w{4,}\b", text))

# Find 'o' followed by one or more 'g's
print(re.findall(r"og+", text))
```

Custom character classes and negation

> A custom character class allows us to **specify a set of characters** from which the regex engine should match any single character while negation in character classes allows us to **match any character that is not in the specified set**.

Custom character class

Syntax: Enclosed in square brackets **`[ ]`**.

Usage examples:

- **`[abc]`**: Matches any one of 'a', 'b', or 'c'.
- **`[a-z]`**: Matches any lowercase letter.
- **`[0-9]`**: Matches any digit.
- **`[A-Za-z]`**: Matches any letter, regardless of case.
- **`[0-9a-fA-F]`**: Matches any hexadecimal digit.

Combining ranges and individual characters: We can combine ranges and individual characters.
For example, **`[abcx-z]`** matches **`'a'`**, **`'b'`**, **`'c'`**, **`'x'`**, **`'y'`**, or **`'z'`**.

Negation in character classes

Syntax: A **caret `^`** at the start of a character class signifies negation.

Usage example: **`[^abc]`** matches any character that is not **`'a'`**, **`'b'`**, or **`'c'`**.

- Make caret (`^`) the first character in negation else it's a literal.
- Place a literal hyphen at the start, end, or after a range in a class, e.g., `[a-z-]`, `[-a-z]`.
- Special characters (`.`, `*`, `+`, ?, etc.) inside a class are matched literally, not as special symbols.

Grouping and capturing

Grouping is achieved by **enclosing a part of the regex pattern in parentheses `()`**.

Syntax: `(pattern)`

Example: In the regex `'(ab)+'`, the group `(ab)` is treated as a single unit followed by a **+** quantifier. This matches one or more repetitions of "**ab**", like "**ab**", "**abab**", "**abababab**", etc.

Capturing groups

Purpose: Capturing groups store the part of the string matched by the group.

Accessing captured groups: In Python, captured groups can be accessed using the `group()` method of match objects.

Example: If a string "**abc123**" is matched against the regex `'(\\d+)'`, the part "**123**" is captured by the group and can be retrieved later. `group(0)` returns the entire match, while `group(1)` returns the first captured group.

Try it yourself

```
import re

# The string to be searched
text = "abc123"

# Regex pattern with a capturing group for one or more
digits
pattern = r"(\d+)"

# Searching the text for the pattern
match = re.search(pattern, text)

# Checking if a match is found
if match:
    # Retrieving the first captured group
    captured_group = match.group(1)
    print("Captured Group:", captured_group)
else:
    print("No match found.")
```

Grouping and capturing

Non-capturing groups

Purpose: Sometimes, we need the grouping functionality without capturing the matched substring.

Syntax: `(?:pattern)`

Example: The regex `(?:ab)+` will match one or more repetitions of "**ab**" but will not store the match.

Backreferences

Definition: Backreferences in a regex pattern allow us to **reuse** the part of the string matched by a capturing group.

Syntax: `\\n`, where **n** is the group number.

Example: The regex `(\\d+)\\s\\1` matches two identical numbers separated by a space, like "**42 42**".

Named groups

Purpose: Improve the readability and manageability of complex regular expressions.

Syntax: `(?P<name>pattern)`

Example: `(?P<digits>\\d+)` creates a group named digits that matches one or more digits.

- When you don't need the captured data, use non-capturing groups to optimise performance.
- For complex patterns with multiple groups, use named groups for better readability.
- If possible, avoid deeply nested groups as they can make the regex pattern difficult to read and maintain.

Advanced anchors and boundaries

> Anchors and boundaries are used to match positions within a string rather than specific characters.

Word boundary (`\b`):

Definition: Matches the position between a word character (**`\w`**) and a non-word character (**`\W`**).

Use cases: To find words in text, ensuring that the match is not just part of a longer word.

Example pattern: **`\bword\b`** matches **`'word'`** in "**`word is a word`**", but not in "**`swordfish`**".

Non-word boundary (`\B`)

Definition: The opposite of **`\b`**. Matches a position where there's **no word boundary**.

Use case: To match patterns that are part of a larger word.

Example pattern: **`\Bion\B`** matches **`'ion'`** in "**`ions are charged`**", but not in "**`ion is an atom`**".

Advanced anchors and boundaries

> Anchors and boundaries are used to match positions within a string rather than specific characters.

Line anchor: ^

Definition: Matches the **start of a string,** or the **start of a line** if multiline mode is enabled.

Use Cases: To ensure that a pattern appears at the beginning of a text or a line.

Example pattern: **^The** matches **`'The'`** in "**`The start`**" but not in "**`At the start`**".

Line anchor: $

Definition: Matches the **end of a string**, or the **end of a line** if multiline mode is enabled.

Use Cases: To ensure that a pattern appears at the end of a text or a line. Example Pattern: end$ matches 'end' in "the end" but not in "endless".

Example pattern: **end$** will match **`'end'`** at the end of "**`It's the end`**" but not in "**`end of the story`**".

Advanced string manipulation

Python ‘re’ module

The **re** module is dedicated to working with regex. It's a tool for **processing strings**, offering functionalities for **searching**, **modifying**, and **splitting** text based on defined patterns.

The **re** module must first be imported into a script using the statement “**import re**”. This grants access to all the functions and classes needed for regex operations.

The module offers a **set of functions** that allow for powerful and flexible **string searching** and **manipulation**.

Foundations of regex in Python

There are **foundational concepts that underpin the use of `re` methods**. These include the use of **flags**, the **syntax of regex patterns**, and the nature of the **match object**. These elements are **common** across most **`re`** methods and grasping them will make it easier to understand how each method operates.

Flags

Flags are **optional** parameters that **alter** how regex are **interpreted** and **matched**.

Common flags include:

- **`re.IGNORECASE`** (or **`re.I`**): Makes the matching process case-insensitive.
- **`re.MULTILINE`** (or **`re.M`**): Treats each line in the text as a separate string, affecting **`^`** and **`$`** anchors.
- **`re.DOTALL`** (or **`re.S`**): Makes the **`.`** character match every character, including newline characters.

Pattern syntax

The pattern syntax is the **language used to write regex.**

Key elements include:

- **Literal characters**: These match exactly in the text.
- **Meta characters**: Special characters like **`.`** (any character), **`*`** (zero or more), **`?`**, etc, that have specific meanings in regex.
- **Constructs**: Includes grouping **`()`**, quantifiers **`{}`**, and more.

The match object

When a regex method finds a match, it often **returns a match object**.

Attributes and methods of a match object include:

- **`.group()`**: Returns the string matched by the entire expression or specific subgroups.
- **`.start()`** and **`.end()`**: Return the start and end positions of the match in the input string.
- **`.span()`:** Provides a tuple with start and end positions of the match.

`re.search()` function

```
re.search(pattern, string, flags=0)
```

The **`re.search()`** function is used to scan through a given string (**`string`**), looking for the first location where the regular expression pattern (**`pattern`**) matches.

Return value:

- If a **matching object** is found a **match object** containing information about the match is returned.
- If there's no match in the string, the function returns **None**.

How it works

1. **Pattern matching**: The function searches the **`string`** from the beginning and stops as soon as it finds the first match for the **`pattern`**.
2. **Flags**: The **`flags`** argument is optional and can be used to modify certain aspects of the regex matching.

Try it yourself

```
import re

text = "Python is an amazing programming language."
pattern = "amazing"

match = re.search(pattern, text)

if match:
    print("Match found:", match.group())
    print("Match starts at position:", match.start())
else:
    print("No match found.")
```

Advanced string manipulation

`re.match()` function

```
re.match(pattern, string, flags=0)
```

The **`re.match()`** function is used for finding matches at the **beginning of a `string`**, unlike **`re.search()`**, which scans for the first occurrence of the **`pattern`** across the whole string.

Return value:

- If a match is found a **match object** containing information about the match is returned.
- If there's no match at the start of the string, the function returns **None**.

How it works

1. **Pattern matching**: The function searches for a **`pattern`** match at the **first character** of the **`string`**.
2. **Flags**: The **`flags`** argument is optional. For example, **`re.MULTILINE`** or **`re.M`** makes the **`^`** and **`$`** anchors match the start and end of each line, instead of just the start and end of the whole string.

Try it yourself

```
import re

pattern = "Hello"
text = "Hello, world!"

# Using re.match to find pattern at the start of the text
match = re.match(pattern, text)

if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

`re.findall()` function

```
re.findall(pattern, string, flags=0)
```

The **`re.findall()`** function searches a **`string`** for **all non-overlapping occurrences** of a regex **`pattern`**. It's efficient for **scanning large texts** and **extracting** pieces of info that match a particular **`pattern`**.

Return value:

- **A list of strings**, where each **`string`** is a match of the **`pattern`**.
- If the **`pattern`** includes one or more capturing groups, it returns **a list of groups** (or **tuples of groups**).

How it works

1. **Pattern matching**: It will scan the **`string`** from left to right, and for each **`pattern`** match, the matching strings will be added to a list.
2. **Non-overlapping**: Once a portion of the **`string`** is part of a match, it **can't** be part of another match.
3. **Capturing groups**: If the **`pattern`** includes capturing groups (parentheses), **`findall`** will return a list of groups. Multiple groups return a list of tuples.

Try it yourself

```
import re

text = "The dates are 2023-01-01, 2024-02-02"
# Single capturing group
pattern = r"(\d{4}-\d{2}-\d{2})"
# Multiple capturing groups
pattern_multiple = r"(\d{4})-(\d{2})-(\d{2})"

dates = re.findall(pattern, text)
date_parts = re.findall(pattern_multiple, text)

print(dates)  # Output: ['2023-01-01', '2024-02-02']
print(date_parts)
```

`re.finditer()` function

```
re.finditer(pattern, string, flags=0)
```

`re.finditer` is used to **find all non-overlapping occurrences** of the pattern in the string, just like **`re.findall`**. However, instead of returning a list of all the matches, it **returns an iterator** that yields match objects.

Return value:

Each **match object** returned contains methods and attributes that provide additional information about the match which include **`.group()`**, **`.start()`**, **`.end()`**:, and **`.span()`**.

How it works

1. **Pattern matching:** Scans the **`string`** from left to right; for each match, it yields a match object instead of the matching strings.
2. **Iterative results:** Returns an iterator yielding match objects, allowing for iteration over each match in the string.
3. **Non-overlapping:** Similar to **`findall`**, once a portion of the **`string`** is part of a match, it won't be included in subsequent matches.

Try it yourself

```
import re

text = "The rain in Spain stays mainly in the plain."
pattern = r"\bS\w+"

matches = re.finditer(pattern, text)

for match in matches:
    print(f"Match: {match.group()} at Position:
{match.span()}")
```

`re.sub()` function

```
re.sub(pattern,repl,string,count=0,flags=0)
```

`re.sub()` is used to **replace** occurrences of a regex **`pattern`** in a **`string`** with a **replacement string** (**`repl`**). It can modify a string in various ways, from simple replacements to complex pattern-based transformations.

Return value:

The method returns a **new string** with the specified substitutions made. The original string **remains unchanged**.

How it works

1. **Pattern identification**: **`string`** is scanned for any match.
2. **Replacement**: Each match is replaced with **`repl`** string.
3. **Count**: If **`count`** is specified and greater than **`0`**, only the first **`count`** occurrences are replaced. If **`count`** is **`0`** (default), all occurrences are replaced.

Try it yourself

```
import re

text = "I love Python. Python is great for scripting.
Python can be used for data analysis."
pattern = "Python"
replacement = "Java"

# Replace all occurrences of 'Python' with 'Java'
result = re.sub(pattern, replacement, text)

print(result)
```

Advanced string manipulation

`re.split()` function

```
re.split(pattern,string,maxsplit=0,flags=0)
```

The **`re.split`** method is used to **split a `string` into a list** using a specified regular expression **`pattern`** as the delimiter.

Return value:

The method returns a **list of strings** obtained by splitting the input **`string`** at every match of the **pattern**.

How it works

1. **Pattern identification**: The **`re.split`** method scans the **`string`** and identifies all occurrences of the provided **`pattern`**.
2. **Splitting**: Whenever the **`pattern`** is found, the string is split at that location.
3. **Maximum number of splits**: If specified, the list will have at most **`maxsplit+1`** elements. The default, **`0`**, means "no limit".

Try it yourself

```
import re

text = "Words, separated, by, commas (and some spaces)"
pattern = ",\s*"  # Comma followed by zero or more spaces

result = re.split(pattern, text)

print(result)
```

`re.compile()` function

```
re.compile(pattern, flags=0)
```

`re.compile` is used to compile a **regex `pattern` into a regex object**. The primary reason for using this method is **performance optimisation**, especially when the same regex pattern is to be applied multiple times.

Return value:

The method returns a **regex object** which can be used to perform various regex operations like **`match`**, **`search`**, **`findall`**, etc. This object stores the compiled version of the pattern, **reducing the overhead** of recompiling the regex each time it's used.

How it works

1. **Pattern compilation**: The given regex **`pattern`** is compiled into an object that can be used to perform matches. This step involves parsing the **`pattern`** and converting it into an internal format optimised for matching.
2. **Pattern identification**: The regex engine uses this compiled **`pattern`** to efficiently identify matches in a string.

Try it yourself

```
import re

# Compile the regex pattern for a simple word match
pattern = re.compile(r'hello')

# Sample text
text = "Hello, world!"

# Using the compiled pattern to search in the text
match = pattern.search(text)
if match:
    print("Match found")
else:
    print("No match found")
```