It's best practice to define regex patterns using **raw strings** by prefixing the string with an `r` (e.g., `r"\d+"`) to prevent backslashes from being misinterpreted.

1. **Finding the First Match with `re.search()`:** This function returns a match object if the pattern is found, and `None` otherwise.

Copy Code

```
import re

text = "The price is $25.99 today."
pattern = r"\\$\\d+\\.\\d{2}" # Pattern for a dollar amount

match = re.search(pattern, text)

if match:
    print(f"Found: {match.group(0)}")
# Outputs: Found: $25.99
```

2. **Finding All Matches with `re.findall()`:** This function returns a list of all strings that match the pattern.

Copy Code

```
import re

text = "User IDs are user1, user2, and user3."
pattern = r"user\\d" # Pattern for "user" followed by a digit

matches = re.findall(pattern, text)
print(matches)
# Outputs: ['user1', 'user2', 'user3']
```

Remember, `re.match()` is different from `re.search()` as it only looks for a match at the very **beginning** of the string.

How to Use Python Regex

The built-in `re` module helps you work with regular expressions. You can use various functions from this module to work with text patterns, such as searching, matching, or replacing string parts.

To use regular expressions in Python, you need to import the `re` module. A regex pattern is a string prefixed by `r` to indicate a raw string. This prefix prevents the wrong interpretation of special characters like `\\`, , which is an example of a backslash with special meaning.

Copy Code

```
import re


pattern = r"\\d+"  # A pattern to match one or more digits
text = "There are 123 apples"
result = re.search(pattern, text)


print(result.group())  # Outputs: '123'
```

The result variable is a match object, which contains information about the match, including the start and end positions.

Python Regex Cheat Sheet

Python regex provides several vital built-in functions. Each function has a specific use case, such as looking for the first match, looking for all matches, or modifying the string.

- `re.match()` : Attempts to match a pattern at the beginning of a string.
- `re.search()` : Scans through the entire string and returns the first occurrence of the pattern.

- `re.findall()` : Returns all occurrences of the pattern in the string as a list.
- `re.sub()` : Replaces parts of the string that match the pattern with another string.

Each function has specific use cases depending on whether you're looking for the first match, all matches, or if you need to modify the string.

In Python regex, patterns comprise special characters that define how to match text. These special characters help you create flexible and powerful search patterns that match various text structures. Some frequently used components include:

- `\\d` : Matches any digit.
- `\\w` : Matches a "word" character (a letter, digit, or underscore character).
- `\\s` : Matches any whitespace characters, including spaces, newline characters, and tabs
- `.` : Matches any character except for the newline character.
- `*` : Matches zero or more occurrences of the preceding character.
- `+` : Matches one or more occurrences of the preceding character.
- `?` : Matches zero or one occurrence of the preceding character.
- [...]: Called square brackets, used to specify a set of characters (character classes).

When to Use Python Regex

Regular expressions can solve many text-related problems, from validation to extraction and manipulation. Some everyday use cases include:

Python Regex Match

You can create a regular expression pattern and use the built-in `re.match()` function to verify that user input meets specific criteria. For instance, you can ensure that an email address, phone number, or password adheres to specific rules.

Copy Code

```
pattern = r"^\\w+@\\w+\\.\\w+$"  # Email validation pattern
email = "example@test.com"
```

```
if re.match(pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")
```

Python Regex Replace

You can substitute string parts based on a pattern using the `re.sub()` function. In Python, regex substitution is commonplace for cleaning up or making data anonymous.

Copy Code

```
text = "My phone number is 123-456-7890"
updated_text = re.sub(r"\\d{3}-\\d{3}-\\d{4}", "[REDACTED]", text)
print(updated_text)  # Outputs: 'My phone number is [REDACTED]'
```

In this example, the pattern matches a phone number format, and `re.sub()` replaces it with `[REDACTED]`.

Regex Split in Python

The `re.split()` function allows you to split a string at each point where a pattern matches. This is useful for tokenizing text or separating data based on multiple delimiters.

Copy Code

```
text = "apple,banana;orange|grape"
fruits = re.split(r"[,;|]", text)
print(fruits)  # Outputs: ['apple', 'banana', 'orange', 'grape']
```

In this example, the text is split into individual fruits using multiple delimiters (commas, semicolons, and pipes).

Python Regex Search

The `re.search()` function looks for the first occurrence of a pattern in a string. This makes `re.search()` ideal for searching for specific keywords within large bodies of text, such as articles or logs.

Copy Code

```
import re

text = "Welcome to Mimo, the best platform for learning to code."
keyword = r"learn"

match = re.search(keyword, text)
if match:
    print(f"Keyword found: {match.group()}")
else:
    print("Keyword not found")
```

In this example, `re.search()` checks if the word `"learn"` exists in the text and returns the first match.

Examples of Using Python Regex

Regex is commonly used in Python 3 for working with unicode strings.

Extracting Information from Server Logs

In many server applications, logs are generated in structured text formats. Regular expressions extract useful data, such as error codes, timestamps, and user activity. For instance, a system administrator may need to extract error timestamps from server logs for troubleshooting.

Copy Code

```
log_entry = "ERROR 2023-01-01: User not found"
pattern = r"\bERROR\b (\d{4}-\d{2}-\d{2})"
match = re.search(pattern, log_entry)
```

```
if match:
    print(f"Error occurred on {match.group(1)}")
```

This example extracts the date from an error log message using a regex pattern that matches the string `"ERROR"` followed by a date.

Reach your coding goals faster

Get started

by scraping product pages.

Copy Code

```
html = "<div>Price: $29.99</div>"
pattern = r"\$\d+\.\d{2}"
price = re.search(pattern, html).group()
print(price)  # Outputs: '$29.99'
```

The regex pattern in this example matches the price in the format of a dollar sign followed by digits and two decimal places.

Filtering Data in a Contact Form

Web applications that process user-submitted data often use regex to filter out unwanted or potentially harmful inputs. For example, a contact form on a website might use regex to validate phone numbers. Using regex, the website can require a particular format for the phone number.

Copy Code

```
contact_info = "Call me at 555-123-4567"
pattern = r"\d{3}-\d{3}-\d{4}"
phone_number = re.search(pattern, contact_info)
```

```
if phone_number:
    print(f"Phone number found: {phone_number.group()}")
```

This example searches for phone numbers formatted as `XXX-XXX-XXXX`.

Learn More About Python Regex

You can iterate through matches using an iterator returned by re.finditer().

Findall Regex in Python

While `re.search()` finds only the first match, `re.findall()` returns all occurrences of the pattern in the string as a list.

Copy Code

```
text = "I have 2 apples and 3 oranges"
matches = re.findall(r"\\d+", text)
print(matches)  # Outputs: ['2', '3']
```

This example finds all digits representing the quantities of apples and oranges in the text.

Case-Insensitive Matching

For a regex pattern to be case-insensitive, you can use the `re.IGNORECASE` flag (also written as `re.I`) as an additional argument. Case-insensitive matching is ideal when text input contains different capitalizations, such as in user-submitted forms.

Copy Code

```
import re

text = "My favorite colors are Blue, RED, and green."
pattern = r"red|blue|green"  # Looking for any color (red, blue, or
matches = re.findall(pattern, text, re.IGNORECASE)


print(matches)  # Outputs: ['Blue', 'RED', 'green']
```

In this case, the regex matches all three strings even though the pattern is lowercase.

Tuple-Based Results

Some functions like `re.findall()` with capturing groups return matches as a tuple.

Handling Multiline Strings with Regex

The `re.MULTILINE` flag allows you to work with multiline strings, treating each line separately for matching purposes.

Copy Code

```
text = """First line
Second line
Third line"""
matches = re.findall(r"^\\w+", text, re.MULTILINE)
print(matches)  # Outputs: ['First', 'Second', 'Third']
```

In this example, the pattern matches the first word on each line.

Using Compiled Regex for Performance

Compiling the pattern with `re.compile()` can improve performance for frequent or more complex regex operations. Compiled patterns can be beneficial for using the same pattern multiple times.

Copy Code

```
pattern = re.compile(r"\\d+")
result = pattern.findall("Numbers: 123, 456")
print(result)  # Outputs: ['123', '456'
```

In this example, compiling the regex pattern reduces overhead and increases efficiency in repetitive operations.

Python Regex for Data Processing and Validation

Regex is invaluable for validating fields such as email addresses, phone numbers, or file formats. It also helps identify and extract data from large datasets, such as parsing CSV files or processing text-based logs.

Copy Code

```
import re

def validate_email(email):
    # Regular expression for validating an email address
    email_pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.
    if re.match(email_pattern, email):
        return "Valid email"
    else:
        return "Invalid email"

def validate_phone(phone):
    # Regular expression for validating a phone number (e.g., 123-45
    phone_pattern = r"^\d{3}-\d{3}-\d{4}$"
    if re.match(phone_pattern, phone):
        return "Valid phone number"
    else:
        return "Invalid phone number"
```

```
# Testing the validation functions
email = "example@example.com"
phone = "123-456-7890"

print(validate_email(email))  # Outputs: "Valid email"
print(validate_phone(phone))  # Outputs: "Valid phone number"
```

This example validates an email address and a phone number to ensure they meet the correct format.

Regular Expressions and Other Programming Languages

Regex in Python code is similar to regex in other languages like JavaScript, Perl, and Java. However, Python provides additional flexibility, such as using raw strings (`r"..."`) to simplify pattern creation.

Although the syntax varies slightly between languages, the core principles of regex remain consistent.

Key Takeaways for Python Regex

- **Import the `re` Module:** All regular expression operations in Python begin with `import re`.
- **Use Raw Strings for Patterns:** Always define your regex patterns with a `r` prefix (e.g., `r"\d+"`). This prevents backslashes from being interpreted as escape sequences by Python.
- **`search()` vs. `match()`:** `re.search()` finds a match **anywhere** in the string, while `re.match()` only finds a match if it occurs at the very **beginning** of the string. `re.search()` is generally more useful.
- **`findall()` Returns a List:** If you need to find every occurrence of a pattern, `re.findall()` is the ideal function. It returns a list of all non-overlapping matches as strings.
- **Get the Match with `.group()`:** When `re.search()` or `re.match()` are successful, they return a match object. You must call the `.group()` method on this object to retrieve the actual matched string.