Python Regular Expressions: Functions and Objects

Introduction

Now that you understand metacharacters and special sequence characters in regular expressions, let's explore how to use Python's `re` module to work with regular expressions. The `re` module provides a variety of functions and objects that allow you to search, match, and manipulate text using regular expressions.

Regular Expression Functions

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

The `compile()` function converts a regular expression pattern into a regular expression object that can be used for matching. It's more efficient when you plan to use the same pattern multiple times.

```python
import re

# Compile a pattern
pattern = re.compile(r'\d+')

# Use the compiled pattern
result = pattern.search("I have 42 apples and 31 oranges")
print(result.group())  # Output: 42
```

Benefits of using `compile()`:

- Improved performance when reusing patterns
- Cleaner code by separating pattern definition from usage
- Access to additional pattern methods

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

The `match()` function attempts to match the pattern at the **beginning** of the string. It returns a Match object if successful, or `None` if no match is found.

python

```
import re

# Match at the beginning of the string
result = re.match(r'Hello', 'Hello, world!')
print(result.group())  # Output: Hello

# No match since "world" is not at the beginning
result = re.match(r'world', 'Hello, world!')
print(result)  # Output: None
```

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

The `search()` function scans through the string looking for the **first** occurrence of the pattern. Unlike `match()`, it doesn't require the pattern to be at the beginning.

python

```
import re

# Search anywhere in the string
result = re.search(r'world', 'Hello, world!')
print(result.group())  # Output: world

# Both match and search work here
result1 = re.match(r'Hello', 'Hello, world!')
result2 = re.search(r'Hello', 'Hello, world!')
print(result1.group(), result2.group())  # Output: Hello Hello
```

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

The `findall()` function returns **all** non-overlapping matches of the pattern in the string, as a list of strings.

python

```
import re

# Find all occurrences of digits
result = re.findall(r'\d+', 'I have 42 apples and 31 oranges')
print(result)  # Output: ['42', '31']

# Find all words
words = re.findall(r'\w+', 'Hello, world! Python is amazing.')
print(words)  # Output: ['Hello', 'world', 'Python', 'is', 'amazing']
```

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

The `finditer()` function is similar to `findall()`, but returns an iterator yielding Match objects instead of strings.

```python
import re

# Find all occurrences of digits and get match objects
matches = re.finditer(r'\d+', 'I have 42 apples and 31 oranges')
for match in matches:
    print(f"Found '{match.group()}' at position {match.start()}-{match.end()}")
# Output:
# Found '42' at position 7-9
# Found '31' at position 20-22
```

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

The `split()` function splits the string by the occurrences of the pattern.

```python
import re

# Split by commas or spaces
result = re.split(r'[,\s]+', 'apple, banana  orange, grape')
print(result)  # Output: ['apple', 'banana', 'orange', 'grape']

# Limit the number of splits
result = re.split(r'[,\s]+', 'apple, banana  orange, grape', maxsplit=2)
print(result)  # Output: ['apple', 'banana', 'orange, grape']
```

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

The `sub()` function replaces all occurrences of the pattern in the string with `repl`. If `count` is provided, only the first `count` occurrences are replaced.

python

```python
import re

# Replace digits with 'X'
result = re.sub(r'\d', 'X', 'Phone: 123-456-7890')
print(result)  # Output: Phone: XXX-XXX-XXXX

# Limit the number of replacements
result = re.sub(r'\d', 'X', 'Phone: 123-456-7890', count=4)
print(result)  # Output: Phone: XXXX-456-7890
```

The replacement can also be a function that receives the match object and returns a string:

python

```python
import re

def double_digits(match):
    return str(int(match.group()) * 2)

result = re.sub(r'\d+', double_digits, 'I have 42 apples')
print(result)  # Output: I have 84 apples
```

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

The `subn()` function is similar to `sub()`, but returns a tuple containing the new string and the number of replacements made.

python

```python
import re

# Replace and count
result = re.subn(r'\d', 'X', 'Phone: 123-456-7890')
print(result)  # Output: ('Phone: XXX-XXX-XXXX', 10)

# Limit the number of replacements
result = re.subn(r'\d', 'X', 'Phone: 123-456-7890', count=4)
print(result)  # Output: ('Phone: XXXX-456-7890', 4)
```

Match Object Methods

When using functions like `match()`, `search()`, or `finditer()`, you get Match objects that provide information about the matches. Here are the most important methods:

`group([group1, ...])`

The `group()` method returns the substring matched by the RE. You can also specify capturing groups by number or name.

```python
import re

# Basic group usage
match = re.search(r'(\d+)-(\d+)', 'Product ID: 123-456')
print(match.group())   # Output: 123-456
print(match.group(0))  # Output: 123-456 (same as above)
print(match.group(1))  # Output: 123
print(match.group(2))  # Output: 456

# Named groups
match = re.search(r'(?P<first>\d+)-(?P<second>\d+)', 'Product ID: 123-456')
print(match.group('first'))   # Output: 123
print(match.group('second'))  # Output: 456
```

`groups(default=None)`

The `groups()` method returns a tuple containing all the subgroups of the match.

```python
import re

match = re.search(r'(\d+)-(\d+)', 'Product ID: 123-456')
print(match.groups())  # Output: ('123', '456')

# With default value for groups that didn't match
match = re.search(r'(\d+)(-(\d+))?', 'Product ID: 123')
print(match.groups())        # Output: ('123', None, None)
print(match.groups(default=0))  # Output: ('123', None, 0)
```

`groupdict(default=None)`

The `groupdict()` method returns a dictionary containing all the named subgroups of the match.

```python
import re

match = re.search(r'(?P<product>[\w\s]+): \$(?P<price>\d+(\.\d+)?)', 'Item: Apple Juice: $5.99'
print(match.groupdict())  # Output: {'product': 'Apple Juice', 'price': '5.99'}
```

`start([group])` and `end([group])`

These methods return the indices of the start and end of the substring matched by the group.

```python
import re

match = re.search(r'world', 'Hello, world!')
print(match.start())  # Output: 7
print(match.end())    # Output: 12

# With groups
match = re.search(r'(\w+), (\w+)', 'Hello, world!')
print(match.start(1))  # Output: 0
print(match.end(1))    # Output: 5
print(match.start(2))  # Output: 7
print(match.end(2))    # Output: 12
```

`span([group])`

The `span()` method returns a tuple containing the (start, end) positions of the match.

```python
import re

match = re.search(r'world', 'Hello, world!')
print(match.span())  # Output: (7, 12)

# With groups
match = re.search(r'(\w+), (\w+)', 'Hello, world!')
print(match.span(1))  # Output: (0, 5)
print(match.span(2))  # Output: (7, 12)
```

`expand(template)`

The `expand()` method returns the string obtained by doing backslash substitution on the template string, as done by the `sub()` method.

```python
import re

match = re.search(r'(\w+), (\w+)', 'Hello, world!')
print(match.expand(r'\2 \1'))  # Output: world Hello
```

Module-Level Operations

`re.escape(pattern)`

The `escape()` function escapes all the characters in the pattern that might have special meaning in a regular expression.

```python
import re

# Escape special characters
pattern = re.escape('www.example.com/search?q=python+regex')
print(pattern)  # Output: www\.example\.com/search\?q=python\+regex

# Use it in a search
text = "Visit www.example.com/search?q=python+regex for more info"
match = re.search(pattern, text)
print(match.group())  # Output: www.example.com/search?q=python+regex
```

`re.purge()`

The `purge()` function clears the regular expression cache.

```python
import re

# Clear the cache
re.purge()
```

Flags

You can modify how regular expressions behave using flags. The most common flags are:

- `re.IGNORECASE` or `re.I`: Ignore case
- `re.MULTILINE` or `re.M`: Make `^` and `$` match the start/end of each line
- `re.DOTALL` or `re.S`: Make `.` match any character including newlines

- `re.VERBOSE` or `re.X`: Allow patterns to be more readable and include comments

```python
import re

# Case-insensitive matching
result = re.search(r'python', 'PYTHON is amazing', re.IGNORECASE)
print(result.group())  # Output: PYTHON

# Multiline mode
text = "Line 1\nLine 2\nLine 3"
results = re.findall(r'^Line \d', text, re.MULTILINE)
print(results)  # Output: ['Line 1', 'Line 2', 'Line 3']

# Verbose mode
pattern = re.compile(r"""
    \d{3}  # Area code
    -      # Separator
    \d{3}  # Exchange code
    -      # Separator
    \d{4}  # Subscriber number
""", re.VERBOSE)
match = pattern.search("Phone: 123-456-7890")
print(match.group())  # Output: 123-456-7890
```

Practical Examples

Example 1: Extracting Email Addresses

```python
import re

text = """
Contact us at info@example.com or support@company.org.
For sales inquiries, reach out to sales@example.com.
"""

# Find all email addresses
emails = re.findall(r'[\w\.-]+@[\w\.-]+', text)
print(emails)  # Output: ['info@example.com', 'support@company.org', 'sales@example.com']
```

Example 2: Parsing Log Files

python

```python
import re

log_line = '192.168.1.1 - - [25/Mar/2021:10:15:32 +0000] "GET /index.html HTTP/1.1" 200 1234'

# Parse log line
pattern = r'(\d+\.\d+\.\d+\.\d+).*?\[(\d+/\w+/\d+:\d+:\d+:\d+).*?\] "(\w+) (.*?) HTTP/.*?" (\d+
match = re.search(pattern, log_line)

if match:
    ip, date, method, path, status, size = match.groups()
    print(f"IP: {ip}")
    print(f"Date: {date}")
    print(f"Method: {method}")
    print(f"Path: {path}")
    print(f"Status: {status}")
    print(f"Size: {size}")
```

Example 3: Form Validation

python

```python
import re

def validate_email(email):
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    return bool(re.match(pattern, email))

def validate_password(password):
    # At least 8 chars, with digits, lowercase and uppercase letters
    pattern = r'^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$'
    return bool(re.match(pattern, password))

# Test validation
emails = ['user@example.com', 'invalid-email', 'another@example']
for email in emails:
    print(f"{email}: {'Valid' if validate_email(email) else 'Invalid'}")

passwords = ['Passw0rd', 'password', 'PASSWORD123', 'Pass']
for password in passwords:
    print(f"{password}: {'Valid' if validate_password(password) else 'Invalid'}")
```

Conclusion

Python's `re` module provides powerful tools for working with regular expressions. By understanding these functions and objects, you can efficiently search, match, extract, and manipulate text based on patterns. Practice with these examples and gradually build up to more complex regular expressions to solve real-world problems.

Remember that regular expressions can become complex, so it's often helpful to break them down into smaller parts and test each part separately. Tools like regex101.com can be invaluable for testing and debugging your regular expressions.