Module 9 - Regular Expressions

Welcome to Module 9! In this module, we'll dive into the world of **Regular Expressions (Regex)**, a powerful and concise language for pattern matching in strings. Whether you're validating user input, parsing log files, or extracting specific information from text, regex will become an indispensable tool.

Chapter 1: Introduction to Regular Expressions

1.1 What are Regular Expressions?

- A **Regular Expression (Regex or Regexp)** is a sequence of characters that defines a **search pattern**.
- It's a mini-programming language embedded within Python (and many other languages) that allows you to:
  - **Search:** Find specific patterns within text.
  - **Validate:** Check if a string conforms to a certain format (e.g., email address, phone number).
  - **Extract:** Pull out specific pieces of information from a larger text.
  - **Replace:** Substitute parts of a string that match a pattern.
- **Why use them?**
  - **Flexibility:** Much more powerful than simple string methods (`startswith()`, `endswith()`, `find()`).
  - **Conciseness:** Complex pattern matching can be expressed in a single, albeit sometimes cryptic, line.
  - **Efficiency:** Regex engines are highly optimized for pattern matching.
- **The `re` Module in Python:** Python's built-in `re` module provides full support for regular expressions. You'll need to import it to use regex functionalities.

  Python

  ```
  import re
  ```

1.2 Raw Strings (`r""`)

- **The Problem:** In Python, the backslash `\` is used to escape special characters within strings (e.g., `\n` for newline, `\t` for tab). Regular expressions also use backslashes to escape their own special characters (e.g., `\d` for digit, `\.` for a literal dot). This can lead to a lot of confusing double backslashes (`\\d` to match a digit if not using raw strings).
- **The Solution:** Python's **raw strings** (prefixed with `r` or `R`) treat backslashes as literal characters, ignoring Python's usual escape sequence rules. This makes writing regular expressions much cleaner and less error-prone.

  Python

```
# Without raw string (incorrect regex for a literal backslash
followed by 'n'):
# pattern = "\\n" # This means Python's newline character.
# To match a literal '\' followed by 'n', you'd need:
# pattern = "\\\\n"

# With raw string (correct and clear):
pattern = r"\n" # Matches a literal backslash followed by 'n' in the
regex engine.
                # If you want Python's newline, just use '\n' without
raw string,
                # or make sure regex is clear, e.g., r'\n' for
newline when appropriate.

# Example: Matching a backslash in text
text = "This contains a \\ backslash."

# Using a regular string (needs double backslash for literal \)
match1 = re.search("\\\\", text) # Matches a single literal backslash

# Using a raw string (preferred for regex)
match2 = re.search(r"\\", text) # Matches a single literal backslash

print(f"Match 1: {match1.group() if match1 else 'No Match'}")
print(f"Match 2: {match2.group() if match2 else 'No Match'}")
```

Always use raw strings for regular expression patterns in Python. It avoids ambiguity and makes your regex patterns much more readable.

Chapter 2: `re.match()` and `re.findall()`

The `re` module provides several functions to perform pattern matching. We'll start with two common ones: `match()` and `findall()`.

2.1 `re.match()`

- **Purpose:** The `re.match()` function attempts to match a pattern **only at the beginning** of the string.
- It checks if the regex pattern exists from the very first character of the string. If the pattern is found anywhere else in the string (not at the beginning), `re.match()` will return `None`.

Syntax:

Python

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

- `pattern`: The regular expression pattern (preferably a raw string).
- `string`: The string to search within.
- `flags` (optional): Modifies the regex behavior (e.g., `re.IGNORECASE`, `re.DOTALL`).

Return Value:

- If a match is found at the beginning of the string, `re.match()` returns a **match object**.
- If no match is found, it returns `None`.

Match Object Methods:

A match object has several useful methods:

- `.group()`: Returns the string matched by the regex.
- `.start()`: Returns the starting index of the match.
- `.end()`: Returns the ending index (exclusive) of the match.
- `.span()`: Returns a tuple `(start, end)` of the match.

Example:

Python

```
import re

text = "Hello World"
pattern1 = r"Hello" # Matches at the beginning
pattern2 = r"World" # Does not match at the beginning

# Using re.match()
match_obj1 = re.match(pattern1, text)
match_obj2 = re.match(pattern2, text)

print(f"Text: '{text}'")

if match_obj1:
    print(f"\nPattern '{pattern1}' matched:")
    print(f"  Match: '{match_obj1.group()}'")
    print(f"  Start Index: {match_obj1.start()}")
    print(f"  End Index: {match_obj1.end()}")
    print(f"  Span: {match_obj1.span()}")
else:
    print(f"\nPattern '{pattern1}' did not match.")

if match_obj2:
    print(f"\nPattern '{pattern2}' matched:")
    print(f"  Match: '{match_obj2.group()}'")
else:
    print(f"\nPattern '{pattern2}' did not match.") # This will be executed
```

Output:

```
Text: 'Hello World'

Pattern 'Hello' matched:
  Match: 'Hello'
  Start Index: 0
  End Index: 5
  Span: (0, 5)

Pattern 'World' did not match.
```

2.2 `re.findall()`

- **Purpose:** The `re.findall()` function finds **all non-overlapping matches** of a pattern in the string and returns them as a list.
- Unlike `re.match()` or `re.search()` (which stops after the first match), `re.findall()` iterates through the entire string to find every instance of the pattern.

Syntax:

Python

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

Return Value:

- A list of strings containing all non-overlapping matches.
- If the pattern contains capturing groups (parts enclosed in parentheses), it returns a list of tuples, where each tuple represents a match and contains the strings captured by each group.

Example:

Python

```
import re

text = "The quick brown fox jumps over the lazy dog. Fox and Dog."
pattern1 = r"fox|dog" # Matches "fox" or "dog" (case-sensitive)
pattern2 = r"Fox|Dog" # Matches "Fox" or "Dog" (case-sensitive)
pattern3 = r"(fox|dog)" # With capturing group

# Using re.findall()
matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)
matches3 = re.findall(pattern3, text, re.IGNORECASE) # Ignore case for
combined match

print(f"Text: '{text}'")
print(f"Pattern '{pattern1}': {matches1}") # Output: ['fox', 'dog']
print(f"Pattern '{pattern2}': {matches2}") # Output: ['Fox', 'Dog']
print(f"Pattern '{pattern3}' with IGNORECASE: {matches3}") # Output:
['fox', 'dog', 'Fox', 'Dog']
```

2.3 Comparison of `re.match()` and `re.findall()`

Feature	`re.match()`	`re.findall()`
Search Scope	Only checks from the **beginning** of the string.	Scans the **entire string** for all non-overlapping matches.
Return Value	A single **match object** or `None`.	A **list of strings** (or tuples for groups).
Usage	Best for validating if a string *starts with* a pattern.	Best for extracting *all occurrences* of a pattern.

Python

```
import re

text = "apple banana apple cherry"
pattern = r"apple"
```

```
# Using re.match()
match_at_start = re.match(pattern, text)
print(f"re.match(): {match_at_start.group() if match_at_start else 'No
match at beginning'}")
# Output: re.match(): apple

# Using re.findall()
all_matches = re.findall(pattern, text)
print(f"re.findall(): {all_matches}")
# Output: re.findall(): ['apple', 'apple']

text_no_start = "banana apple cherry"
match_no_start = re.match(pattern, text_no_start)
print(f"re.match() on '{text_no_start}': {match_no_start.group() if
match_no_start else 'No match at beginning'}")
# Output: re.match() on 'banana apple cherry': No match at beginning
```

Chapter 3: `re.search()`

3.1 `re.search()`

- **Purpose:** The `re.search()` function scans through the string looking for the **first location** where the pattern produces a match.
- Unlike `re.match()`, `re.search()` does not restrict the match to the beginning of the string. It will find the pattern anywhere in the string.
- Once it finds the first match, it stops searching.

Syntax:

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

- `pattern`: The regular expression pattern.
- `string`: The string to search within.
- `flags` (optional): Modifies the regex behavior.

Return Value:

- If a match is found anywhere in the string, `re.search()` returns a **match object**.
- If no match is found, it returns `None`.

Key Difference from `re.match()`:

`re.search()` is a more general-purpose search function compared to `re.match()`.

- `re.match()` checks for a match *only at the beginning*.
- `re.search()` checks for a match *anywhere* in the string, but only returns the *first* one.

3.2 Examples and Common Usage

Python

```
import re

text = "This is a test string. This is another test."
pattern = r"test"

# Using re.search()
match_obj = re.search(pattern, text)

print(f"Text: '{text}'")

if match_obj:
    print(f"\nPattern '{pattern}' found:")
    print(f"  Match: '{match_obj.group()}'")
    print(f"  Start Index: {match_obj.start()}")
    print(f"  End Index: {match_obj.end()}")
    print(f"  Span: {match_obj.span()}")
else:
    print(f"\nPattern '{pattern}' not found.")

# Example demonstrating difference with re.match()
text2 = "Some text with a keyword here."
pattern2 = r"keyword"

match_search = re.search(pattern2, text2)
match_match = re.match(pattern2, text2)

print(f"\nText 2: '{text2}'")
print(f"re.search('{pattern2}'): {match_search.group() if match_search else
'No match'}")
print(f"re.match('{pattern2}'): {match_match.group() if match_match else
'No match'}")

# Example: Finding a specific word (case-insensitive)
text3 = "Python is powerful. python is versatile."
pattern3 = r"python"

search_case_sensitive = re.search(pattern3, text3)
search_case_insensitive = re.search(pattern3, text3, re.IGNORECASE)

print(f"\nText 3: '{text3}'")
print(f"Case-sensitive search: {search_case_sensitive.group() if
search_case_sensitive else 'Not found'}")
print(f"Case-insensitive search: {search_case_insensitive.group() if
search_case_insensitive else 'Not found'}")
```

Output:

```
Text: 'This is a test string. This is another test.'

Pattern 'test' found:
  Match: 'test'
  Start Index: 10
  End Index: 14
  Span: (10, 14)

Text 2: 'Some text with a keyword here.'
re.search('keyword'): keyword
re.match('keyword'): No match
```

```
Text 3: 'Python is powerful. python is versatile.'
Case-sensitive search: python
Case-insensitive search: Python
```

`re.search()` is often the most generally useful function when you just need to find the first occurrence of a pattern anywhere within a string.

Chapter 4: `re.sub()`

4.1 `re.sub()`

- **Purpose:** The `re.sub()` function is used to **substitute (replace)** occurrences of a regular expression pattern in a string with a replacement string or the result of a function call.
- It's extremely powerful for text manipulation and data cleaning.

Syntax:

Python

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

- `pattern`: The regular expression pattern to search for.
- `repl`: The replacement string or a function to be called for each match.
  - If `repl` is a string, backreferences like `\1`, `\2` can be used to refer to captured groups in the pattern. `\g<name>` can be used for named groups.
- `string`: The input string where substitutions will be made.
- `count` (optional): The maximum number of pattern occurrences to be replaced. Default is 0, which means all occurrences.
- `flags` (optional): Modifies the regex behavior.

Return Value:

- The new string with the substitutions applied.

4.2 Examples

1. Simple String Replacement:

Python

```
import re

text = "Hello, Python World! Python is great."
pattern = r"Python"
replacement = "Java"

new_text = re.sub(pattern, replacement, text)
print(f"Original: {text}")
print(f"Modified: {new_text}")
# Output: Original: Hello, Python World! Python is great.
#         Modified: Hello, Java World! Java is great.
```

2. Using `count` to Limit Replacements:

Python
```
text = "one two one three one four"
pattern = r"one"
replacement = "X"

new_text_all = re.sub(pattern, replacement, text)
new_text_limit = re.sub(pattern, replacement, text, count=1)

print(f"Original: {text}")
print(f"All replaced: {new_text_all}") # Output: X two X three X four
print(f"Limited to 1: {new_text_limit}") # Output: X two one three one four
```

3. Using Backreferences in Replacement String:

Backreferences allow you to re-use parts of the matched pattern in the replacement string. `\1` refers to the content of the first capturing group, `\2` to the second, and so on.

Python
```
text = "Name: John Doe, Age: 30. Name: Jane Smith, Age: 25."
# Pattern to find "Name: [First Name] [Last Name]"
# Groups: (John) (Doe)
pattern = r"Name: (\w+) (\w+)"
# Replacement: "Last Name, First Name"
replacement = r"\2, \1"

new_text = re.sub(pattern, replacement, text)
print(f"Original: {text}")
print(f"Modified: {new_text}")
# Output: Original: Name: John Doe, Age: 30. Name: Jane Smith, Age: 25.
#         Modified: Doe, John, Age: 30. Smith, Jane, Age: 25.
```

4. Using a Function as `repl`:

You can pass a function as the `repl` argument. This function will be called for each match, and its return value will be used as the replacement string. The function receives the match object as its argument.

Python
```
def double_number(match):
    num = int(match.group(0)) # Get the matched number string, convert to
int
    return str(num * 2)      # Double it and convert back to string

text = "The numbers are 10, 25, and 50."
pattern = r"\d+" # Matches one or more digits

new_text = re.sub(pattern, double_number, text)
print(f"Original: {text}")
print(f"Modified: {new_text}")
# Output: Original: The numbers are 10, 25, and 50.
#         Modified: The numbers are 20, 50, and 100.
```

`re.sub()` is incredibly versatile for cleaning, formatting, and transforming text data.

Chapter 5: Characters and Character Sequences (Part 1: Literals, Escapes, Wildcards, Anchors)

This chapter begins our deep dive into the syntax of regular expression patterns.

5.1 Literal Characters

- Most characters in a regex pattern match themselves literally.
- **Example:** The pattern `abc` will match the exact sequence "abc".

  Python

  ```
  import re
  text = "The quick brown fox."
  print(re.search(r"quick", text).group()) # Output: quick
  ```

5.2 Escaping Special Characters (`\`)

Some characters have special meanings in regex. To match them literally, you must **escape** them with a backslash (`\`).

- **Common Special Characters:** `. * + ? | ( ) [ ] { } ^ $ \`

  Python

  ```
  import re
  text = "This price is $10.50."

  # To match a literal '$' and '.'
  # Incorrect: pattern = r"$10.50" (would have special meaning for $
  and .)
  pattern = r"\$10\.50"

  match = re.search(pattern, text)
  print(match.group() if match else "No match") # Output: $10.50

  # To match a literal backslash
  text2 = "Path: C:\\Users\\User"
  pattern2 = r"C:\\Users" # Matches "C:\Users"
  match2 = re.search(pattern2, text2)
  print(match2.group() if match2 else "No match") # Output: C:\Users
  ```

5.3 The Dot (`.`) - Any Character

- The dot `.` is a wildcard character. It matches **any single character** except for a newline character (`\n`) by default.
- **Example:** `a.b` will match "aab", "axb", "a-b", etc., but not "ab" or "a\nb".

  Python

```
import re
text = "cat, cot, cut, c.t"
pattern = r"c.t" # Matches 'c', any character, then 't'

matches = re.findall(pattern, text)
print(matches) # Output: ['cat', 'cot', 'cut', 'c.t']
```

Note: You can make `.` match newlines by using the `re.DOTALL` flag: `re.search(pattern, text, re.DOTALL)`.

5.4 Anchors (`^` and `$`):

Anchors don't match characters; they match positions within the string.

- **Caret (`^`): Start of the String**
  - `^pattern` matches `pattern` only if it appears at the very beginning of the string.
- **Dollar Sign (`$`): End of the String**
  - `pattern$` matches `pattern` only if it appears at the very end of the string.
- **Combined (`^pattern$`): Exact String Match**
  - `^pattern$` matches `pattern` only if the entire string is an exact match for the pattern.

Python

```
import re

text1 = "apple pie"
text2 = "I like apple"
text3 = "apple"

# ^ (Start of string)
print(f"'{text1}' matches '^apple': {re.search(r'^apple', text1) is
not None}") # True
print(f"'{text2}' matches '^apple': {re.search(r'^apple', text2) is
not None}") # False

# $ (End of string)
print(f"'{text1}' matches 'pie$': {re.search(r'pie$', text1) is not
None}")     # True
print(f"'{text2}' matches 'apple$': {re.search(r'apple$', text2) is
not None}") # True

# ^...$ (Exact match for entire string)
print(f"'{text1}' matches '^apple pie$': {re.search(r'^apple pie$',
text1) is not None}") # True
print(f"'{text3}' matches '^apple$': {re.search(r'^apple$', text3) is
not None}") # True
print(f"'{text1}' matches '^apple$': {re.search(r'^apple$', text1) is
not None}") # False
```

Chapter 6: Characters and Character Sequences (Part 2: Quantifiers and OR)

This chapter focuses on controlling how many times a part of your pattern can repeat and how to match one of several alternatives.

6.1 Quantifiers

Quantifiers specify how many occurrences of the preceding character, group, or character set must be present for a match.

- **`*` (Zero or more occurrences):**
  - Matches the preceding element zero or more times.
  - `a*` will match "", "a", "aa", "aaa", and so on.

  Python

  ```
  import re
  print(re.findall(r"ab*c", "ac abc abbc abbbc")) # Output: ['ac',
  'abc', 'abbc', 'abbbc']
  ```

- **+ (One or more occurrences):**
  - Matches the preceding element one or more times.
  - `a+` will match "a", "aa", "aaa", but not "".

  Python

  ```
  print(re.findall(r"ab+c", "ac abc abbc abbbc")) # Output: ['abc',
  'abbc', 'abbbc']
  ```

- **`?` (Zero or one occurrence):**
  - Matches the preceding element zero or one time (makes it optional).
  - `colou?r` will match "color" or "colour".

  Python

  ```
  print(re.findall(r"colou?r", "color colour coleur")) # Output:
  ['color', 'colour']
  ```

- **`{n}` (Exactly `n` times):**
  - Matches the preceding element exactly `n` times.
  - `a{3}` matches "aaa" but not "aa" or "aaaa".

  Python

  ```
  print(re.findall(r"a{3}", "baaab aaaab aaaaab")) # Output: ['aaa',
  'aaa']
  ```

- **`{n,}` (At least `n` times):**
  - Matches the preceding element `n` or more times.
  - `a{2,}` matches "aa", "aaa", "aaaa", etc.

  Python

```
print(re.findall(r"a{2,}", "baaab aaaab aaaaab")) # Output: ['aaa',
'aaaa', 'aaaaa']
```

- **`{n,m}` (Between `n` and `m` times):**
  - Matches the preceding element at least `n` times but no more than `m` times.
  - `a{2,4}` matches "aa", "aaa", "aaaa".

  Python

  ```
  print(re.findall(r"a{2,4}", "baab aaaab aaaaab aaaaab")) # Output:
  ['aa', 'aaa', 'aaaa', 'aaaa']
  ```

6.2 Greedy vs. Non-Greedy Quantifiers (`?` after quantifier)

By default, quantifiers (`*`, `+`, `?`, `{}`) are **greedy**. They try to match the *longest possible* string. You can make them **non-greedy** (or *lazy*) by adding a `?` immediately after the quantifier. Non-greedy quantifiers match the *shortest possible* string.

- `*?` (Zero or more, non-greedy)
- `+?` (One or more, non-greedy)
- `??` (Zero or one, non-greedy)
- `{n,}?` (At least n times, non-greedy)
- `{n,m}?` (Between n and m times, non-greedy)

Python
```
import re

html_tag = "<h1>Title</h1><p>Paragraph</p>"

# Greedy match: Matches from the first < to the last >
greedy_pattern = r"<.*>"
print(f"Greedy: {re.findall(greedy_pattern, html_tag)}")
# Output: ['<h1>Title</h1><p>Paragraph</p>']

# Non-greedy match: Matches the shortest possible <...> tag
non_greedy_pattern = r"<.*?>"
print(f"Non-Greedy: {re.findall(non_greedy_pattern, html_tag)}")
# Output: ['<h1>', '</h1>', '<p>', '</p>']
```

This is a common pitfall when parsing structured text; non-greedy quantifiers are often what you need.

6.3 The OR Operator (`|`)

- The pipe symbol `|` acts as an "OR" operator. It matches either the pattern before the `|` or the pattern after it.

  Python

  ```
  import re
  text = "apple, banana, cherry, orange"

  # Match 'apple' OR 'orange'
  ```

```
pattern = r"apple|orange"
print(re.findall(pattern, text)) # Output: ['apple', 'orange']

# Match 'cat' OR 'dog' OR 'mouse'
text2 = "I have a cat and a dog. Also a small mouse."
pattern2 = r"cat|dog|mouse"
print(re.findall(pattern2, text2)) # Output: ['cat', 'dog', 'mouse']
```

Note: When using `|` inside a group `()`, it applies to the content of the group only: `(cat|dog)food` would match "catfood" or "dogfood".

Chapter 7: Characters and Character Sequences (Part 3: Character Sets and Classes, Boundaries)

This chapter concludes our dive into regex syntax, covering ways to define groups of characters and specific positions.

7.1 Character Sets (`[]`)

- Square brackets `[]` define a **character set**. They match **any single character** *within* the set.
- **Examples:**
  - `[abc]`: Matches 'a', 'b', or 'c'.
  - `[aeiou]`: Matches any lowercase vowel.
  - `[0-9]`: Matches any digit from 0 to 9 (equivalent to `\d`).
  - `[a-z]`: Matches any lowercase letter.
  - `[A-Z]`: Matches any uppercase letter.
  - `[A-Za-z0-9_]`: Matches any alphanumeric character or underscore (equivalent to `\w`).

  Python

  ```
  import re
  text = "The color is grey or gray."

  # Match 'grey' or 'gray'
  print(re.findall(r"gr[ae]y", text)) # Output: ['grey', 'gray']

  # Match any digit
  print(re.findall(r"[0-9]", "Product ID: P123_abc_456")) # Output:
  ['1', '2', '3', '4', '5', '6']

  # Match any character that is NOT a digit (using negation)
  print(re.findall(r"[^0-9]", "Product ID: P123_abc_456")) # Output:
  ['P', 'r', 'o', 'd', 'u', 'c', 't', ' ', 'I', 'D', ':', ' ', 'P',
  '_', 'a', 'b', 'c', '_']
  ```

- **Negated Character Sets (`[^...]`)**:
  - If the first character inside `[]` is a caret `^`, the set becomes negated. It matches any single character *not* in the set.

    - `[^0-9]` matches any character that is *not* a digit.

7.2 Predefined Character Classes

Python's `re` module provides shorthand character classes for common character sets. These are more concise and readable than using `[]`.

- **`\d`:** Matches any digit (0-9). Equivalent to `[0-9]`.
- **`\D`:** Matches any non-digit character. Equivalent to `[^0-9]`.

  Python

```
print(re.findall(r"\d", "Phone: 123-456-7890"))  # Output: ['1', '2',
'3', '4', '5', '6', '7', '8', '9', '0']
print(re.findall(r"\D", "Phone: 123-456-7890")) # Output: ['P', 'h',
'o', 'n', 'e', ':', ' ', '-', '-']
```

- **`\w`:** Matches any "word" character (alphanumeric characters `a-z`, `A-Z`, `0-9`, and underscore _). Equivalent to `[a-zA-Z0-9_]`.
- **`\W`:** Matches any non-"word" character. Equivalent to `[^a-zA-Z0-9_]`.

  Python

```
print(re.findall(r"\w", "Hello, World!_123")) # Output: ['H', 'e',
'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd', '_', '1', '2', '3']
print(re.findall(r"\W", "Hello, World!_123")) # Output: [',', ' ',
'!']
```

- **`\s`:** Matches any whitespace character (space, tab `\t`, newline `\n`, carriage return `\r`, form feed `\f`, vertical tab `\v`).
- **`\S`:** Matches any non-whitespace character.

  Python

```
print(re.findall(r"\s", "Hello\tWorld\n")) # Output: ['\t', '\n']
print(re.findall(r"\S", "Hello\tWorld\n")) # Output: ['H', 'e', 'l',
'l', 'o', 'W', 'o', 'r', 'l', 'd']
```

7.3 Word Boundaries (`\b` and `\B`)

These are zero-width assertions; they match positions, not characters.

- **`\b` (Word Boundary):** Matches the empty string at the beginning or end of a word. A "word" is defined as a sequence of word characters (`\w`).
    - It matches the position between a word character and a non-word character (`\W`), or at the beginning/end of the string if it's followed/preceded by a word character.

  Python

```
text = "cat catcher concat"
# Matches the whole word "cat"
```

```
print(re.findall(r"\bcat\b", text)) # Output: ['cat']

text2 = "cat."
print(re.findall(r"\bcat\b", text2)) # Output: ['cat'] (matches
before '.')
```

- **`\B` (Non-Word Boundary):** Matches the empty string where `\b` does not. It matches positions that are *not* at the beginning or end of a word.

  Python

  ```
  text = "cat catcher concat"
  # Matches "cat" where it's part of a larger word
  print(re.findall(r"\Bcat\B", text)) # Output: ['cat'] (from 'concat')

  # Example: "cat" in "catcher" is not a word boundary on the right
  print(re.findall(r"cat\B", text)) # Output: ['cat'] (from 'catcher')
  ```

7.4 Groups (`()`)

Parentheses `()` are used to **group** parts of a regular expression together. This serves two main purposes:

1. **Capturing Groups:**
   - The matched text within a group can be extracted separately.
   - In `re.findall()`, if groups are present, it returns tuples of captured groups.
   - In `re.search()`/`re.match()`, you can access captured groups using `match_obj.group(1)`, `match_obj.group(2)`, etc., or `match_obj.groups()`.

   Python

   ```
   import re
   text = "Date: 2023-10-26"
   # Capture year, month, and day separately
   pattern = r"(\d{4})-(\d{2})-(\d{2})"

   match = re.search(pattern, text)
   if match:
       print(f"Full match: {match.group(0)}") # Or match.group()
       print(f"Year: {match.group(1)}")
       print(f"Month: {match.group(2)}")
       print(f"Day: {match.group(3)}")
       print(f"All groups as tuple: {match.groups()}")

   # With findall, it returns a list of tuples for captured groups
   text_multi = "Dates: 2023-01-15, 2024-03-20"
   all_dates = re.findall(pattern, text_multi)
   print(f"All captured dates: {all_dates}")
   # Output: [('2023', '01', '15'), ('2024', '03', '20')]
   ```

2. **Non-Capturing Groups (`(?:...)`):**
   - Sometimes you need to group parts of a regex for applying quantifiers or the `|` operator, but you don't want to capture the content.
   - Use `(?:pattern)` for a non-capturing group.

Python

```
# Match "abc" or "def", followed by "xyz"
# Capturing group: (abc|def)xyz will capture "abc" or "def"
print(re.findall(r"(abc|def)xyz", "abcxyz defxyz")) # Output: ['abc',
'def']

# Non-capturing group: (?:abc|def)xyz will just match, but not
capture the alternatives
print(re.findall(r"(?:abc|def)xyz", "abcxyz defxyz")) # Output:
['abcxyz', 'defxyz']
```

Chapter 8: Coding Challenges for Regular Expressions

Let's apply these regex patterns and functions to solve some common text processing problems!

Challenge 1: Basic Email Validation

Goal: Write a Python script to check if a given string is a valid (simple) email address format.

Concepts Covered: `re.match()` or `re.search()`, `^`, `$`, `\w`, `\.`, `@`, quantifiers (`+`).

Requirements:

1. Define a function `is_valid_email(email_string)` that takes a string.
2. Use a regular expression to validate the email format. A simple valid format is:
   - Starts with one or more word characters (`\w+`).
   - Followed by an `@` symbol.
   - Followed by one or more word characters (`\w+`).
   - Followed by a literal dot (`\.`).
   - Followed by 2 or 3 word characters for the top-level domain (`\w{2,3}`).
   - The pattern should match the **entire string** (`^...$`).
3. The function should return `True` if it matches, `False` otherwise.
4. Test with a few valid and invalid examples.

Example Test Cases:

- Valid: `"user@example.com"`, `"john.doe123@sub.domain.co"`
- Invalid: `"invalid-email"`, `"user@.com"`, `"@domain.com"`, `"user@domain."`, `"user@domain.comm"`

Challenge 2: Extracting Phone Numbers

Goal: Extract all phone numbers from a given text.

Concepts Covered: `re.findall()`, `\d`, `\-`, `\(`, `\)`, quantifiers (`{}`, `+`), optional characters (`?`).

Requirements:

1. Define a function `extract_phone_numbers(text)` that takes a string.
2. Use `re.findall()` to find all phone numbers in the text.
3. A phone number can be in these simple formats:
   - `XXX-XXX-XXXX` (e.g., "123-456-7890")
   - `(XXX) XXX-XXXX` (e.g., "(123) 456-7890")
   - `XXXXXXXXXX` (e.g., "1234567890")
4. Return a list of all found phone numbers.

Example Test Case:

```
text = "Call me at 123-456-7890 or (987) 654-3210. My old number was
5551234567. Contact support at (111)222-3333."
```

Challenge 3: Finding Dates

Goal: Find all occurrences of dates in a specific format within a string.

Concepts Covered: `re.findall()`, `\d`, grouping (`()`), fixed repetitions (`{}`), literal characters.

Requirements:

1. Define a function `find_dates(text)` that takes a string.
2. Use `re.findall()` to locate all dates in the `YYYY-MM-DD` format.
   - Capture Year (4 digits), Month (2 digits), and Day (2 digits) as separate groups.
3. The function should return a list of tuples, where each tuple contains (Year, Month, Day) for each found date.

Example Test Case:

```
text = "Meeting on 2023-11-05. Project due by 2024-01-31. Another date:
2022-07-10."
```

Challenge 4: Text Cleanup (`re.sub()`)

Goal: Clean up a messy string by replacing multiple spaces with a single space, and removing leading/trailing spaces.

Concepts Covered: `re.sub()`, `\s`, quantifiers (`+`, `*`), anchors (`^`, `$`).

Requirements:

1. Define a function `clean_text(text)` that takes a string.
2. Use `re.sub()` to perform two operations:
   - Replace any sequence of two or more whitespace characters (`\s+`) with a single space (`" "`).

   - Remove any leading or trailing whitespace. (You might need `re.sub()` twice or string `.strip()` after the first `re.sub()`).
3. Return the cleaned string.

Example Test Case:

```
messy_text = "   This  is  a   messy   string   with   extra    spaces.   "
```