Module 24 - Regular Expressions Revisited

This module will delve into the powerful world of Regular Expressions (often shortened to "regex" or "regexp"). Regular expressions provide a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. They are essential for tasks like data validation, parsing log files, extracting information, and find-and-replace operations.

Chapter 1: Introduction to Regular Expressions & Raw Strings

1.1 Introduction to Regular Expressions & Raw Strings

- **What are Regular Expressions (Regex)?**
  - Regular expressions are a sequence of characters that define a search pattern.
  - They are a mini-language within Python (and many other languages like Java, JavaScript, Perl, PHP) specifically designed for pattern matching in strings.
  - You can use them to:
    - **Search:** Find specific patterns within a larger 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 string.
    - **Replace:** Find and replace parts of a string based on a pattern.
- **Python's `re` Module:**
  - Python's built-in `re` module provides full support for regular expressions.
  - To use regex in Python, you'll almost always start by importing this module: `import re`.
- **Raw Strings (`r""` or `R""`)**:
  - This is CRUCIAL when working with regular expressions in Python.
  - Regular expressions often use backslashes (`\`) for special sequences (e.g., `\d` for a digit, `\s` for whitespace).
  - In standard Python strings, the backslash is also used as an escape character (e.g., `\n` for newline, `\t` for tab).
  - This dual meaning can lead to confusion and errors. For example, `\n` in a normal string means newline, but in regex, you might intend `\n` to literally match the characters `\` followed by `n`.
  - **Raw strings** treat backslashes as literal characters, ignoring their special meaning as escape sequences. By prefixing your string literal with `r` or `R`, you tell Python not to process backslashes as escape characters.

Python

```
import re

# Example without raw string (problematic)
```

```
# The \n here will be interpreted as a newline character by Python
string literal
# before re.compile even sees it.
# pattern_problem = "\nexample"
# print(pattern_problem) # This will print a newline then "example"

# Example with raw string (correct for regex)
# The \n here is treated as the literal characters '\' and 'n'
pattern_correct = r"\nexample"
print(pattern_correct) # This will print "\nexample" literally

# Let's see a practical example: matching a literal backslash
text = r"This has a backslash: \"
# To match a literal backslash, you need to escape it in regex
# In a normal string, you'd need "\\\\" to represent one literal
backslash in regex
# In a raw string, you only need "\\" to represent one literal
backslash in regex
match = re.search(r"\\", text)
if match:
    print(f"Matched literal backslash: {match.group()}")
```

Rule of Thumb: Always use raw strings (`r"your_regex_pattern"`) for your regular expression patterns in Python.

Chapter 2: Search & Match Methods

The `re` module provides several functions to work with regular expressions. The two most fundamental for finding patterns are `re.search()` and `re.match()`.

2.1 Search & Match Methods

- **`re.search(pattern, string, flags=0)`**:
  - This function scans through the *entire* `string` from beginning to end to find the *first* location where the `pattern` produces a match.
  - If a match is found, it returns a **Match Object**.
  - If no match is found anywhere in the string, it returns `None`.
- **`re.match(pattern, string, flags=0)`**:
  - This function checks for a match *only at the beginning* of the `string`.
  - If the `pattern` matches at the start of the string, it returns a **Match Object**.
  - If the pattern does not match at the very beginning, it returns `None` (even if the pattern exists later in the string).
- **Key Difference:**
  - `re.match()` is like asking: "Does this string *start* with this pattern?"
  - `re.search()` is like asking: "Does this pattern exist *anywhere* in this string?"
- **Match Object:** When `re.search()` or `re.match()` finds a match, they return a **Match Object**. This object contains information about the match. Common methods of a Match Object:
  - **`match_object.group(0)` or `match_object.group()`**: Returns the entire matched string.
  - **`match_object.start()`**: Returns the starting index of the match.

    - **`match_object.end()`**: Returns the ending index (exclusive) of the match.
    - **`match_object.span()`**: Returns a tuple `(start, end)` of the match.
- **Example:**

  Python

```
import re

text = "The quick brown fox jumps over the lazy dog."
pattern = r"fox"

# --- Using re.search() ---
search_match = re.search(pattern, text)

if search_match:
    print("--- re.search() found a match! ---")
    print(f"Matched string: '{search_match.group()}'")
    print(f"Start index: {search_match.start()}")
    print(f"End index: {search_match.end()}")
    print(f"Span: {search_match.span()}")
else:
    print("--- re.search() did NOT find a match. ---")

print("\n" + "="*30 + "\n")

# --- Using re.match() ---
text2 = "The quick brown fox jumps over the lazy dog."
pattern2 = r"quick" # This pattern is not at the beginning

match_match = re.match(pattern2, text2)

if match_match:
    print("--- re.match() found a match! ---")
    print(f"Matched string: '{match_match.group()}'")
else:
    print("--- re.match() did NOT find a match at the beginning. ---
")

text3 = "Python is powerful."
pattern3 = r"Python"

match_match2 = re.match(pattern3, text3)

if match_match2:
    print("\n--- re.match() found a match at the beginning (text3)! -
--")
    print(f"Matched string: '{match_match2.group()}'")
```

Chapter 3: Introduction To Metacharacters

3.1 Introduction To Metacharacters

- **What are Metacharacters?**
    - Metacharacters are special characters in regular expressions that have a specific meaning rather than representing themselves literally.
    - They are the building blocks that allow you to define complex patterns.

    - If you need to match a metacharacter literally (e.g., match a literal `.` or `*`), you usually need to "escape" it by preceding it with a backslash (`\`). For example, `\.` matches a literal dot, and `\*` matches a literal asterisk.
- **Common Metacharacters (we will cover these in detail):**
    - `.` (Dot): Matches any single character (except newline).
    - `*` (Asterisk / Star): Matches zero or more occurrences.
    - `+` (Plus): Matches one or more occurrences.
    - `?` (Question Mark): Matches zero or one occurrence (makes preceding optional).
    - `{n,m}` (Curly Braces): Quantifier for specific number of occurrences.
    - `^` (Caret): Matches the start of the string/line.
    - `$` (Dollar): Matches the end of the string/line.
    - `[]` (Square Brackets): Character set (matches any one character inside the set).
    - `|` (Pipe): OR operator.
    - `()` (Parentheses): Grouping and capturing.
    - `\` (Backslash): Escapes special characters, or defines special sequences (`\d`, `\s`, `\w`).

Chapter 4: Star Meta Character

4.1 Star Meta Character (*)

- **Meaning:** The `*` metacharacter (also called the "star" or "asterisk") matches the preceding character (or group) **zero or more** times.
- It's a quantifier, meaning it specifies how many times the element before it can repeat.
- **Zero or More:** This is important. It means the element doesn't even have to be present for a match to occur.
- **Example:**

  Python

```
import re

print("--- Using '*' (zero or more) ---")

# Pattern: 'a' followed by zero or more 'b's
pattern = r"ab*"

text1 = "ac" # 'a' followed by zero 'b's
match1 = re.search(pattern, text1)
if match1:
    print(f"'{text1}' matched '{match1.group()}'") # Output: 'a'

text2 = "abc" # 'a' followed by one 'b'
match2 = re.search(pattern, text2)
if match2:
    print(f"'{text2}' matched '{match2.group()}'") # Output: 'ab'

text3 = "abbbbc" # 'a' followed by multiple 'b's
match3 = re.search(pattern, text3)
```

```
if match3:
    print(f"'{text3}' matched '{match3.group()}'") # Output: 'abbbb'

text4 = "cba" # No 'a' at the beginning, but 'ab*' can be found later
match4 = re.search(pattern, text4)
if match4:
    print(f"'{text4}' matched '{match4.group()}'") # Output: 'a'
(finds the 'a' only)

text5 = "Apple"
# Matches 'A' followed by zero or more 'p's. Finds 'App'.
match5 = re.search(r"Ap*", text5)
if match5:
    print(f"'{text5}' matched '{match5.group()}'")

text6 = "xyz"
# This pattern 'Ap*' will NOT find 'xyz' as there's no 'A'
match6 = re.search(r"Ap*", text6)
if match6:
    print(f"'{text6}' matched '{match6.group()}'")
else:
    print(f"'{text6}' did not match '{pattern}'")

print("\n--- Matching patterns at the beginning ---")
text_begin1 = "000abc"
# Matches zero or more '0's at the beginning, followed by 'a'
match_begin1 = re.match(r"0*a", text_begin1)
if match_begin1:
    print(f"'{text_begin1}' matched '{match_begin1.group()}'") #
Output: '000a'

text_begin2 = "abc"
# Matches zero or more '0's (which is zero here), followed by 'a'
match_begin2 = re.match(r"0*a", text_begin2)
if match_begin2:
    print(f"'{text_begin2}' matched '{match_begin2.group()}'") #
Output: 'a'
```

Chapter 5: Introduction To Plus Meta Character

5.1 Introduction To Plus Meta Character (+)

- **Meaning:** The + metacharacter (the "plus") matches the preceding character (or group) **one or more** times.
- Like *, it's a quantifier.
- **One or More:** This is the key difference from *. The element *must* be present at least once for a match to occur.

Chapter 6: Plus Meta Character Example

6.1 Plus Meta Character Example

- **Example:**

  Python

```
import re

print("--- Using '+' (one or more) ---")

# Pattern: 'a' followed by one or more 'b's
pattern = r"ab+"

text1 = "ac" # 'a' followed by zero 'b's - NO MATCH
match1 = re.search(pattern, text1)
if match1:
    print(f"'{text1}' matched '{match1.group()}'")
else:
    print(f"'{text1}' did not match '{pattern}' (no 'b' present).") #
Output: No match

text2 = "abc" # 'a' followed by one 'b'
match2 = re.search(pattern, text2)
if match2:
    print(f"'{text2}' matched '{match2.group()}'") # Output: 'ab'

text3 = "abbbbc" # 'a' followed by multiple 'b's
match3 = re.search(pattern, text3)
if match3:
    print(f"'{text3}' matched '{match3.group()}'") # Output: 'abbbb'

text4 = "cba" # 'a' is not followed by 'b'
match4 = re.search(pattern, text4)
if match4:
    print(f"'{text4}' matched '{match4.group()}'")
else:
    print(f"'{text4}' did not match '{pattern}'.") # Output: No match

text5 = "Aapple"
# Matches 'A' followed by one or more 'p's. Finds 'App'.
match5 = re.search(r"Ap+", text5)
if match5:
    print(f"'{text5}' matched '{match5.group()}'")

print("\n--- Contrasting '*' vs. '+' ---")
text_contrast = "colr"
pattern_star = r"colo*r" # Matches 'color' or 'colr' or 'colooooor'
pattern_plus = r"colo+r" # Matches 'color' or 'colooooor', but NOT
'colr' (needs at least one 'o')

match_star = re.search(pattern_star, text_contrast)
if match_star:
    print(f"'{text_contrast}' matched '{pattern_star}':
{match_star.group()}") # Output: 'colr'

match_plus = re.search(pattern_plus, text_contrast)
if match_plus:
    print(f"'{text_contrast}' matched '{pattern_plus}':
{match_plus.group()}")
else:
    print(f"'{text_contrast}' did not match '{pattern_plus}' (needs
at least one 'o').") # Output: No match
```

Chapter 7: Introduction To Curly Braces

7.1 Introduction To Curly Braces (`{ }`)

- **Meaning:** Curly braces are also quantifiers, providing a more precise way to specify the number of occurrences of the preceding character or group.
- They allow you to define:
  - **Exact number:** Exactly `n` times.
  - **Minimum number:** At least `n` times.
  - **Range:** Between `n` and `m` times (inclusive).
- **Syntax:**
  - **`{n}`**: Matches exactly `n` occurrences of the preceding element.
  - **`{n,}`**: Matches `n` or more occurrences of the preceding element.
  - **`{n,m}`**: Matches between `n` and `m` (inclusive) occurrences of the preceding element.

Chapter 8: Curly Braces Example

8.1 Curly Braces Example

- **Example:**

  Python

```
import re

print("--- Using Curly Braces Quantifiers ---")

# {n}: Exactly n times
pattern_exact = r"a{3}b" # Matches 'aaab'
text1 = "aaab"
match1 = re.search(pattern_exact, text1)
if match1:
    print(f"'{text1}' matched '{pattern_exact}': {match1.group()}") #
Output: 'aaab'
text2 = "aab" # Too few 'a's
match2 = re.search(pattern_exact, text2)
if not match2:
    print(f"'{text2}' did not match '{pattern_exact}' (needs exactly
3 'a's).")

# {n,}: At least n times
pattern_at_least = r"a{2,}b" # Matches 'aab', 'aaab', 'aaaab', etc.
text3 = "aab"
match3 = re.search(pattern_at_least, text3)
if match3:
    print(f"'{text3}' matched '{pattern_at_least}':
{match3.group()}") # Output: 'aab'
text4 = "aaaaab"
```

```
match4 = re.search(pattern_at_least, text4)
if match4:
    print(f"'{text4}' matched '{pattern_at_least}':
{match4.group()}") # Output: 'aaaaab'
text5 = "ab" # Too few 'a's
match5 = re.search(pattern_at_least, text5)
if not match5:
    print(f"'{text5}' did not match '{pattern_at_least}' (needs at
least 2 'a's).")

# {n,m}: Between n and m times (inclusive)
pattern_range = r"a{2,4}b" # Matches 'aab', 'aaab', 'aaaab'
text6 = "aab"
match6 = re.search(pattern_range, text6)
if match6:
    print(f"'{text6}' matched '{pattern_range}': {match6.group()}") #
Output: 'aab'
text7 = "aaaab"
match7 = re.search(pattern_range, text7)
if match7:
    print(f"'{text7}' matched '{pattern_range}': {match7.group()}") #
Output: 'aaaab'
text8 = "aaaaaab" # Too many 'a's
match8 = re.search(pattern_range, text8)
if not match8:
    print(f"'{text8}' did not match '{pattern_range}' (needs 2-4
'a's).")
text9 = "ab" # Too few 'a's
match9 = re.search(pattern_range, text9)
if not match9:
    print(f"'{text9}' did not match '{pattern_range}' (needs 2-4
'a's).")

# Practical example: phone number (simple format like ddd-ddd-dddd)
# Note: this is a very simplified example, real phone numbers are
complex
phone_pattern = r"\d{3}-\d{3}-\d{4}"
phone_number1 = "123-456-7890"
match_phone1 = re.search(phone_pattern, phone_number1)
if match_phone1:
    print(f"\nPhone: '{phone_number1}' matched '{phone_pattern}':
{match_phone1.group()}")
phone_number2 = "123-4567-890" # Incorrect format
match_phone2 = re.search(phone_pattern, phone_number2)
if not match_phone2:
    print(f"Phone: '{phone_number2}' did not match
'{phone_pattern}'.")
```

Chapter 9: Introduction To Wildcard

9.1 Introduction To Wildcard (.)

- **Meaning:** The . (dot) metacharacter is often called the "wildcard" character.
- It matches **any single character** except for a newline character (`\n`).
- It's incredibly useful when you want to match patterns where one specific character can vary.

Chapter 10: Wildcard Example

10.1 Wildcard Example

- **Example:**

  Python

```
import re

print("--- Using '.' (Wildcard) ---")

# Pattern: 'c' followed by any single character, followed by 't'
pattern = r"c.t"

text1 = "cat"
match1 = re.search(pattern, text1)
if match1:
    print(f"'{text1}' matched '{pattern}': {match1.group()}") #
Output: 'cat'

text2 = "cot"
match2 = re.search(pattern, text2)
if match2:
    print(f"'{text2}' matched '{pattern}': {match2.group()}") #
Output: 'cot'

text3 = "cut"
match3 = re.search(pattern, text3)
if match3:
    print(f"'{text3}' matched '{pattern}': {match3.group()}") #
Output: 'cut'

text4 = "c t" # Matches space
match4 = re.search(pattern, text4)
if match4:
    print(f"'{text4}' matched '{pattern}': {match4.group()}'") #
Output: 'c t'

text5 = "caat" # Does not match 'aa'
match5 = re.search(pattern, text5)
if not match5:
    print(f"'{text5}' did not match '{pattern}' (needs exactly one
character between c and t).")

text6 = "ct" # Too few characters
match6 = re.search(pattern, text6)
if not match6:
    print(f"'{text6}' did not match '{pattern}' (needs exactly one
character between c and t).")

text7 = "The red car parked."
# Matches 'car'
match7 = re.search(r"c.r", text7)
if match7:
    print(f"'{text7}' matched '{match7.group()}'")
```

```
text8 = "Hello\nWorld"
# The dot normally does NOT match newline characters.
match8 = re.search(r"Hello.World", text8)
if not match8:
    print(f"'{text8}' did not match 'Hello.World' (because of
newline).")

# To make '.' match newline characters as well, use re.DOTALL flag
match9 = re.search(r"Hello.World", text8, re.DOTALL)
if match9:
    print(f"'{text8}' matched 'Hello.World' with re.DOTALL:
{match9.group()}")
```

Chapter 11: Optional Meta Character

11.1 Optional Meta Character (?)

- **Meaning:** The ? (question mark) metacharacter makes the preceding character (or group) **optional**.
- It matches the preceding element **zero or one** time.
- This is often used to match variations of a word where a letter might be present or absent.
- **Example:**

  Python

```
import re

print("--- Using '?' (Optional) ---")

# Pattern: 'colou?r' - matches 'color' or 'colour'
pattern = r"colou?r"

text1 = "color" # 'u' is present zero times
match1 = re.search(pattern, text1)
if match1:
    print(f"'{text1}' matched '{pattern}': {match1.group()}") #
Output: 'color'

text2 = "colour" # 'u' is present one time
match2 = re.search(pattern, text2)
if match2:
    print(f"'{text2}' matched '{pattern}': {match2.group()}") #
Output: 'colour'

text3 = "coloor" # Too many 'o's, or 'u' is not in position
match3 = re.search(pattern, text3)
if not match3:
    print(f"'{text3}' did not match '{pattern}'.")

text4 = "Is it grey or gray?"
# Matches 'grey' or 'gray'
match4 = re.search(r"gr(e|a)y", text4) # More advanced using OR, or
r"gra?ey" if only 'e' optional
# A better example for just '?' might be:
```

```
match5 = re.search(r"gra?y", "gray")
if match5:
    print(f"'gray' matched 'gra?y': {match5.group()}")
match6 = re.search(r"gra?y", "gry")
if not match6:
    print(f"'gry' did not match 'gra?y'.")

# '?' also has a second meaning: non-greedy matching.
# When placed after a quantifier (*, +, {n,m}), it makes the
quantifier match
# as few characters as possible (non-greedy) instead of as many as
possible (greedy).
# This will be covered in more advanced regex topics.
```

Chapter 12: Caret Meta Character

12.1 Caret Meta Character (^)

The ^ (caret) metacharacter has two primary meanings depending on its context:

1. **Matches the Start of a String/Line:**
    - When ^ is used at the beginning of a regular expression pattern (or at the beginning of a line if the `re.MULTILINE` flag is set), it asserts the position at the start of the string (or line).
2. **Negation within a Character Class:**
    - When ^ is used immediately after an opening square bracket ([) within a character class, it negates the class. This means it matches any character *not* in the specified set.

- **Example (Start of String/Line):**

  Python

```
import re

print("--- Using '^' (Start of String/Line) ---")

# Matches 'Hello' only if it's at the very beginning of the string
pattern_start = r"^Hello"

text1 = "Hello World"
match1 = re.search(pattern_start, text1)
if match1:
    print(f"'{text1}' matched '{pattern_start}': {match1.group()}") #
Output: 'Hello'

text2 = "World Hello" # 'Hello' is not at the start
match2 = re.search(pattern_start, text2)
if not match2:
    print(f"'{text2}' did not match '{pattern_start}'.")

text3 = "Line1\nHello Line2"
match3 = re.search(pattern_start, text3)
if not match3:
```

```
    print(f"'{text3}' did not match '{pattern_start}' (without
MULTILINE flag, ^ means start of string).")

# Using re.MULTILINE flag: '^' matches start of each line
match4 = re.search(pattern_start, text3, re.MULTILINE)
if match4:
    print(f"'{text3}' matched '{pattern_start}' with MULTILINE:
{match4.group()}") # Output: 'Hello' (from second line)
```

- **Example (Negation in Character Class):**

  Python

```
import re

print("\n--- Using '^' (Negation in Character Class) ---")

# Pattern: Match any character that is NOT 'a', 'b', or 'c'
pattern_negation = r"[^abc]"

text5 = "xyz" # All characters are not 'a', 'b', or 'c'
match5 = re.search(pattern_negation, text5)
if match5:
    print(f"'{text5}' matched '{pattern_negation}':
{match5.group()}") # Output: 'x' (first char not a/b/c)

text6 = "abcde" # 'a', 'b', 'c' are skipped, then 'd' matches
match6 = re.search(pattern_negation, text6)
if match6:
    print(f"'{text6}' matched '{pattern_negation}':
{match6.group()}") # Output: 'd'

text7 = "aaaaa" # All characters are 'a'
match7 = re.search(pattern_negation, text7)
if not match7:
    print(f"'{text7}' did not match '{pattern_negation}'.")

# Match any character that is NOT a digit
pattern_no_digit = r"[^0-9]"
text8 = "Product123"
match8 = re.search(pattern_no_digit, text8)
if match8:
    print(f"'{text8}' matched '{pattern_no_digit}':
{match8.group()}") # Output: 'P'
text9 = "98765"
match9 = re.search(pattern_no_digit, text9)
if not match9:
    print(f"'{text9}' did not match '{pattern_no_digit}'.")
```

Chapter 13: Character Classes Part 1

13.1 Character Classes Part 1 (`[]`)

- **Meaning:** Square brackets `[]` define a **character class** (or character set).
- A character class matches **any one** of the characters enclosed within the brackets.

- It's a way to specify a set of characters, any one of which will satisfy the match at that position.
- **Syntax and Usage:**
  - **`[abc]`**: Matches 'a', 'b', or 'c'.
  - **`[0123456789]`**: Matches any single digit from 0 to 9.
  - **`[aeiou]`**: Matches any single lowercase vowel.
  - **Ranges with Hyphen (-):** You can specify a range of characters using a hyphen.
    - **`[0-9]`**: Matches any single digit (0 through 9).
    - **`[a-z]`**: Matches any single lowercase letter (a through z).
    - **`[A-Z]`**: Matches any single uppercase letter (A through Z).
    - **`[a-zA-Z]`**: Matches any single uppercase or lowercase letter.
    - **`[a-zA-Z0-9]`**: Matches any single alphanumeric character.
- **Example:**

  Python

```
import re

print("--- Using Character Classes ([]) ---")

# Match 'c' followed by 'a' or 'o' or 'u', followed by 't'
pattern = r"c[aou]t"

text1 = "cat"
match1 = re.search(pattern, text1)
if match1:
    print(f"'{text1}' matched '{pattern}': {match1.group()}") #
Output: 'cat'

text2 = "cot"
match2 = re.search(pattern, text2)
if match2:
    print(f"'{text2}' matched '{pattern}': {match2.group()}") #
Output: 'cot'

text3 = "cut"
match3 = re.search(pattern, text3)
if match3:
    print(f"'{text3}' matched '{pattern}': {match3.group()}") #
Output: 'cut'

text4 = "cit" # 'i' is not in the set
match4 = re.search(pattern, text4)
if not match4:
    print(f"'{text4}' did not match '{pattern}'.")

# Using ranges
print("\n--- Using Ranges in Character Classes ([0-9], [a-z], etc.) -
--")
pattern_digit = r"Number: [0-9]" # Matches "Number: " followed by a
single digit
text5 = "Number: 5"
match5 = re.search(pattern_digit, text5)
if match5:
    print(f"'{text5}' matched '{pattern_digit}': {match5.group()}")
```

```
pattern_vowel = r"[aeiouAEIOU]" # Matches any single vowel (case-
insensitive)
text6 = "Python"
match6 = re.search(pattern_vowel, text6)
if match6:
    print(f"First vowel in '{text6}' is: {match6.group()}") # Output:
'o'

# Match a hexadecimal digit (0-9, A-F, a-f)
pattern_hex = r"[0-9a-fA-F]"
text7 = "Hex: C3"
match7 = re.search(pattern_hex, text7)
if match7:
    print(f"First hex digit in '{text7}' is: {match7.group()}") #
Output: 'C'

text8 = "color or colour"
# Matches 'color' or 'colour' (more complex, but uses character
classes implicitly)
# The '?' quantifier could also be used here as seen before
match8 = re.search(r"colo[u]?r", text8) # 'u' is optional
if match8:
    print(f"'{text8}' first match for 'colo[u]?r': {match8.group()}")
```

Chapter 14: Character Classes Part 2

14.1 Character Classes Part 2 (Shorthand Character Sets)

Regular expressions provide convenient shorthand character sets (escape sequences) for commonly used character classes. These are very useful and make patterns more concise.

- **Shorthand Character Sets:**
  - **`\d`**: Matches any **digit** (0-9). Equivalent to `[0-9]`.
  - **`\D`**: Matches any **non-digit** character. Equivalent to `[^0-9]`. (Capital letter is usually the negation).
  - **`\w`**: Matches any **word character**. This includes alphanumeric characters (letters a-z, A-Z, 0-9) and the underscore (_). Equivalent to `[a-zA-Z0-9_]`.
  - **`\W`**: Matches any **non-word character**. This includes anything that is not a letter, digit, or underscore (e.g., spaces, punctuation, symbols).
  - **`\s`**: Matches any **whitespace character**. This includes space, tab (`\t`), newline (`\n`), carriage return (`\r`), form feed (`\f`), and vertical tab (`\v`).
  - **`\S`**: Matches any **non-whitespace character**.
- **Example:**

  Python

```
import re

print("--- Using Shorthand Character Sets ---")

# \d: Digit
pattern_digit = r"\d{3}" # Matches three consecutive digits
text1 = "My number is 123-456-7890."
match1 = re.search(pattern_digit, text1)
```

```
if match1:
    print(f"First 3 digits: {match1.group()}") # Output: '123'

# \D: Non-digit
pattern_non_digit = r"\D+" # Matches one or more non-digit characters
text2 = "abc123xyz"
match2 = re.search(pattern_non_digit, text2)
if match2:
    print(f"First non-digit sequence: {match2.group()}") # Output:
'abc'

# \w: Word character (alphanumeric + underscore)
pattern_word = r"\w+" # Matches one or more word characters
text3 = "Hello_World 123!"
match3 = re.search(pattern_word, text3)
if match3:
    print(f"First word sequence: {match3.group()}") # Output:
'Hello_World'

# \W: Non-word character
pattern_non_word = r"\W" # Matches any single non-word character
text4 = "Hello_World 123!"
match4 = re.search(pattern_non_word, text4)
if match4:
    print(f"First non-word character: '{match4.group()}'") # Output:
' ' (space)

# \s: Whitespace character
pattern_space = r"\s" # Matches any single whitespace character
text5 = "Hello World\nHow are you?"
match5 = re.search(pattern_space, text5)
if match5:
    print(f"First whitespace character: '{match5.group()}' (at index
{match5.start()})") # Output: ' '

# \S: Non-whitespace character
pattern_non_space = r"\S+" # Matches one or more non-whitespace
characters
text6 = " Hello World "
match6 = re.search(pattern_non_space, text6)
if match6:
    print(f"First non-whitespace sequence: '{match6.group()}'") #
Output: 'Hello'

# Combining shorthands
# Simple email pattern (very simplified for demonstration)
email_pattern = r"\w+@\w+\.\w+" # word@word.word
text7 = "My email is test@example.com and another is
user_name@domain.co.uk"
match7 = re.search(email_pattern, text7)
if match7:
    print(f"Found email: {match7.group()}") # Output:
test@example.com

# Date pattern (DD-MM-YYYY)
date_pattern = r"\d{2}-\d{2}-\d{4}"
text8 = "Today's date is 07-07-2024."
match8 = re.search(date_pattern, text8)
if match8:
    print(f"Found date: {match8.group()}") # Output: 07-07-2024
```