==========
CHAPTER 4: REGEX - COMPLETE SOLUTIONS
==========

==========
MULTIPLE CHOICE QUESTIONS (MCQs) - ANSWERS
==========

1. Which function in Python checks only at the beginning of a string?
   Answer: A
   Explanation: re.match() checks for a pattern only at the beginning of the string

2. What does the regex pattern \d+ match?
   Answer: B
   Explanation: \d+ matches one or more digits

3. Which regex will match any string ending with "ing"?
   Answer: B
   Explanation: ing$ matches any string ending with "ing"

4. The output of re.findall (r" [aeiou]", "Python Programming") is:
   Answer: D
   Explanation: Returns ['o', 'o', 'a', 'i']

5. Which regex matches a valid variable name in Python (letters, numbers, underscores, not starting with digit)?
   Answer: B
   Explanation: ^[A-Za-z_]\w*$ matches valid variable names

6. The metacharacter inside brackets [^...] means:
   Answer: C
   Explanation: Negation (not these characters)

7. What will be the result of: re.split(r"\s+", "Python is easy")
   Answer: B
   Explanation: Returns ['Python', 'is', 'easy']

8. Which function is best for replacing substrings using regex?
   Answer: B
   Explanation: re.sub() is best for replacing substrings using regex

==========
TRUE/FALSE QUESTIONS - ANSWERS

==========

1. re.match() scans the entire string for a pattern.
   Answer: False
   Explanation: re.match() only checks at the beginning of the string, not the entire string. Use re.search() to scan the entire string.

2. The regex . matches any character except a newline.
   Answer: True
   Explanation: The dot (.) matches any character except a newline by default.

3. Regex \w+ matches only uppercase letters.
   Answer: False
   Explanation: \w+ matches letters, digits, and underscores, not just uppercase letters.

4. The regex \d{3} matches exactly three digits.
   Answer: True
   Explanation: {3} specifies exactly 3 occurrences.

5. re.sub() can be used for both searching and replacing.
   Answer: True
   Explanation: re.sub() searches for patterns and replaces them.

6. Regex patterns in Python are case-sensitive unless re.IGNORECASE is used.
   Answer: True
   Explanation: By default, regex is case-sensitive.

7. re.findall() returns only the first match found.
   Answer: False
   Explanation: re.findall() returns all non-overlapping matches in a list, not just the first one.

8. Python$ matches the string "I love Python".
   Answer: True
   Explanation: Python$ matches at the end of the string, which matches "I love Python".

==========
SHORT ANSWER & CONCEPTUAL QUESTIONS - SOLUTIONS
==========

Question 1: Differentiate between re.match() and re.search().

Answer:
- re.match(): Checks for a pattern only at the beginning of the string. Returns a match object if found at the start, None otherwise.

- re.search(): Scans the entire string for a pattern. Returns a match object for the first occurrence anywhere in the string.

Example:
```
import re
text = "Python is great"
print(re.match(r'is', text))     # None (not at the beginning)
print(re.search(r'is', text))    # Match object (found in the string)
```

Question 2: Explain the difference between +, *, and ? quantifiers in regex.

Answer:
- + (one or more): Matches one or more occurrences of the preceding element.
  Example: r'\d+' matches "123" but not ""

- * (zero or more): Matches zero or more occurrences of the preceding element.
  Example: r'\d*' matches "123" or ""

- ? (zero or one): Matches zero or one occurrence of the preceding element.
  Example: r'\d?' matches "1" or ""

Question 3: What is the purpose of named groups in regex? Provide an example.

Answer:
Named groups allow you to extract specific parts of a match using meaningful names instead of numeric indices. This makes code more readable and maintainable.

Example:
```
import re
pattern = r'(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})'
text = "2023-05-12"
match = re.search(pattern, text)
print(match.group('year'))   # 2023
print(match.group('month'))  # 05
print(match.group('day'))    # 12
```

Question 4: Write a regex to match a date in the format YYYY-MM-DD.

Answer:
Simple pattern: r'\d{4}-\d{2}-\d{2}'

More strict pattern with validation: r'^([0-9]{4})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$'

- \d{4} or [0-9]{4}: Exactly 4 digits for year
- \d{2} or (0[1-9]|1[0-2]): 2 digits for month (01-12)
- \d{2} or (0[1-9]|[12][0-9]|3[01]): 2 digits for day (01-31)

Question 5: What does the pattern ^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$ validate?

Answer:
This pattern validates email addresses with the following components:
- ^ : Start of string
- [A-Za-z0-9._%+-]+ : Local part (letters, numbers, dot, underscore, percent, plus, hyphen) - one or more
- @ : Literal @ symbol
- [A-Za-z0-9.-]+ : Domain name (letters, numbers, dot, hyphen) - one or more
- \. : Literal dot
- [A-Za-z]{2,} : TLD (letters only) - at least 2 characters
- $: End of string

Example valid emails: user@example.com, john.doe@company.co.uk
Example invalid: user@.com, @example.com, user.example.com

Question 6: Why might re.split(r"\s+", text) be preferred over str.split()?

Answer:
- re.split(r"\s+", text): Treats multiple consecutive whitespaces (spaces, tabs, newlines) as a single separator
- str.split(): When called without arguments, splits on any whitespace, but may behave inconsistently with multiple spaces
- re.split() provides more control and flexibility over the splitting pattern

Example:
```
import re
text = "Hello    world\t\tPython"
print(text.split())                # ['Hello', 'world', 'Python']
print(re.split(r'\s+', text))      # ['Hello', 'world', 'Python']
print(re.split(r'\s', text))       # ['Hello', '', '', '', 'world', '', '', 'Python']
```

Question 7: What is the difference between a raw string (r"pattern") and a normal string in regex?

Answer:

- Raw string (r"pattern"): Backslashes are treated literally, useful for regex where \ has special meaning
- Normal string ("pattern"): Backslashes need to be escaped (\\), making patterns harder to read

Example:
r'\d+' is equivalent to '\\d+'
r'\w+' is equivalent to '\\w+'

Raw strings prevent Python from interpreting the backslash, allowing it to be passed to the regex engine.

Question 8: Explain how re.sub() can be used for text normalization (e.g., removing multiple spaces).

Answer:
re.sub(r'\s+', ' ', text) replaces one or more whitespace characters with a single space.

Example:
```
import re
text = "Hello    world    Python"
normalized = re.sub(r'\s+', ' ', text)
print(normalized)  # "Hello world Python"
```

Other normalization examples:
```
# Remove leading/trailing spaces
normalized = text.strip()

# Remove all vowels
normalized = re.sub(r'[aeiouAEIOU]', '', text)

# Replace multiple punctuation with single
normalized = re.sub(r'[.!?]+', '.', text)
```

==========
PROGRAMMING PROBLEMS - SOLUTIONS
==========

PROBLEM 1: Validate Email Addresses
==========

Code:
```
import re
emails = ["user@example.com", "bad-email", "test@domain.org"]
```

```
pattern = re.compile(r'^[A-Za-z0-9._]+@[A-Za-z0-9.-]+\.(?:com|org|edu)$')
print([e for e in emails if pattern.match(e)])
```

Output:
['user@example.com', 'test@domain.org']

Explanation:
- Pattern starts with ^ and ends with $ for full string match
- [A-Za-z0-9._]+ matches the email local part (before @)
- @ matches the literal @ symbol
- [A-Za-z0-9.-]+ matches the domain name
- \. matches the literal dot
- (?:com|org|edu) matches one of the three allowed extensions
- Non-capturing group (?:...) prevents capture, just for grouping

---

PROBLEM 2: Extract Hashtags
==========

Code:
```
import re
text = "I love #Python and #AI"
hashtags = re.findall(r'#\w+', text)
print(hashtags)
```

Output:
['#Python', '#AI']

Explanation:
- # matches the literal hash symbol
- \w+ matches one or more word characters (letters, digits, underscore)
- re.findall() returns all non-overlapping matches as a list

PROBLEM 3: Validate Phone Numbers
==========

Code:
```
import re
phone_numbers = ["+1-555-1234", "123-456-7890", "5551234"]
pattern = re.compile(r'^(\+1-\d{3}-\d{4}|\d{3}-\d{3}-\d{4})$')
print([p for p in phone_numbers if pattern.match(p)])
```

Output:
['+1-555-1234', '123-456-7890']

Explanation:
- Pattern uses alternation (|) for two formats
- First format: \+1-\d{3}-\d{4} matches +1-555-1234 (international format)
  - \+ matches literal + symbol (escaped because + is special in regex)
  - 1- matches literal "1-"
  - \d{3} matches exactly 3 digits
  - - matches literal hyphen
  - \d{4} matches exactly 4 digits
- Second format: \d{3}-\d{3}-\d{4} matches 123-456-7890 (US format)
- ^ and $ ensure full string match

PROBLEM 4: Word Frequency (Regex Tokenizer)
==========

Code:
```
import re
from collections import Counter
text = "Python, Python! AI is great; Python AI."
words = re.findall(r'\b\w+\b', text)
word_freq = Counter(word.lower() for word in words)
print(dict(word_freq))
```

Output:
```
{'python': 3, 'ai': 2, 'is': 1, 'great': 1}
```

Explanation:
- \b matches word boundaries
- \w+ matches one or more word characters
- re.findall() extracts all words
- word.lower() converts to lowercase for case-insensitive counting
- Counter counts occurrences of each word
- dict() converts Counter to regular dictionary

PROBLEM 5: Find Duplicate Words
==========

Code:
```
import re
text = "This is is a test test"
pattern = r'\b(\w+)\s+\1\b'
matches = re.findall(pattern, text, re.IGNORECASE)
print(matches)
```

Output:

['is', 'test']

Explanation:
- (\w+) captures a word in group 1
- \s+ matches one or more whitespace characters
- \1 is a backreference to group 1 (matches the same word again)
- \b ensures word boundaries
- re.IGNORECASE makes it case-insensitive
- re.findall() returns the captured groups, not the full match

Advanced version to return full matches:
pattern = r'\b(\w+)\s+\1\b'
full_matches = re.findall(r'\b\w+\s+(?=\w+\b)', text)

PROBLEM 6: Extract Dates
==========

Code:
import re
text = "The events are on 2023-05-12 and 2024-01-01."
pattern = r'\d{4}-\d{2}-\d{2}'
dates = re.findall(pattern, text)
print(dates)

Output:
['2023-05-12', '2024-01-01']

Explanation:
- \d{4} matches exactly 4 digits (year)
- - matches literal hyphen
- \d{2} matches exactly 2 digits (month)
- - matches literal hyphen
- \d{2} matches exactly 2 digits (day)
- re.findall() returns all non-overlapping matches

More strict validation (with month/day ranges):
pattern = r'([0-9]{4})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])'

PROBLEM 7: Mask Sensitive Data
==========

Code:
import re
text = "Card: 1234-5678-9012-3456"

```
pattern = r'\d{4}-\d{4}-\d{4}-(\d{4})'
masked = re.sub(pattern, r'****-****-****-\1', text)
print(masked)
```

Output:
Card: ****-****-****-3456

Explanation:
- (\d{4}) captures the last 4 digits in group 1
- re.sub() replaces the entire pattern
- r'****-****-****-\1' replaces with asterisks and group 1 (last 4 digits)
- \1 is a backreference to the captured group

Alternative approach:
```
import re
text = "Card: 1234-5678-9012-3456"
masked = re.sub(r'\d(?=\d{4})', '*', text)
print(masked)  # Card: ****-****-****-3456
```

PROBLEM 8: Extract Programming Languages
==========

Code:
```
import re
text = "I know Python, Java, and C++ but not Ruby."
languages = ['Python', 'Java', 'C\\+\\+', 'Ruby', 'JavaScript']
pattern = r'\b(' + '|'.join(languages) + r')\b'
extracted = re.findall(pattern, text)
print(extracted)
```

Output:
['Python', 'Java', 'Ruby']

Explanation:
- languages list contains known programming languages
- 'C\\+\\+' escapes the + symbols (+ is special in regex, needs \\ in string)
- '|'.join(languages) creates alternation pattern: Python|Java|C\+\+|Ruby|JavaScript
- \b ensures word boundaries (prevents matching "JavaScripter" if searching for "JavaScript")
- re.findall() returns all matches

==========
END OF CHAPTER 4 SOLUTIONS
==========