
# MASTER OF COMPUTER APPLICATIONS

## SEMESTER 1

# PYTHON PROGRAMMING

# Unit 9

# Regular Expressions and Text Processing

## Table of Contents

# 1. INTRODUCTION

In Unit 8, you embarked on an enriching journey through the world of Object-Oriented Programming (OOP) in Python, where you learned about classes, objects, encapsulation, inheritance, and polymorphism. These concepts are foundational in crafting well-structured and modular code, allowing you to model real-world problems effectively within your programs. By mastering OOP, you've gained the skills to create complex, reusable, and easy-to-maintain software components, setting a solid foundation for tackling more advanced programming challenges.

Now, as we transition into Unit 9, we delve into the crucial topic of File Input/Output (I/O) in Python. In the digital world, data is omnipresent, and knowing how to interact with files is essential for any programmer. This unit will equip you with the knowledge and skills to read from and write to files, manage file resources, and manipulate file data—key abilities for developing applications that persist data, configure settings, or process external datasets. You'll learn about different file operations, handling various data formats, and the importance of safely interacting with the filesystem to prevent data loss or corruption.

To make the most out of this unit, approach the learning materials with a hands-on mindset. Dive into the provided examples and exercises to apply the concepts in real-world scenarios. Experiment with reading and writing different file formats, understand the nuances of file modes, and practice error handling to ensure your file operations are robust. Engaging actively with the content will not only solidify your understanding of File I/O but also enhance your ability to build more complex and data-driven Python applications. Remember, the skills you acquire in this unit are not just theoretical; they are practical tools that will serve you in a wide range of programming tasks and projects.

## 1.1 Learning Objectives

*After studying this unit, you will be able to:*

- *Apply the appropriate file modes for reading, writing, and appending data to files in Python.*
- *Demonstrate the ability to handle exceptions and errors that occur during file operations to ensure robust file I/O processes.*

- *Construct scripts that utilize Python's file I/O capabilities to parse, process, and store data from various file formats.*
- *Evaluate the effectiveness of different file handling techniques, including context managers, for resource management in Python applications.*

## 2. INTRODUCTION TO REGULAR EXPRESSIONS

Regular expressions, often abbreviated as "regex" or "regexp", are sequences of characters used as a search pattern. They are utilized for string searching and manipulation and are incredibly powerful for text processing tasks. In programming and data processing, understanding how to use regular expressions can significantly enhance your ability to work with text data efficiently.

### 2.1 Syntax

The syntax of regular expressions involves a combination of characters and symbols that form a search pattern. Key elements include:

- **Literals**: These are the simplest form of patterns and match the exact character(s) in the text. For example, the regex hello will match the substring "hello" in "hello world".

```
import re

text = "hello world"

match = re.search(r'hello', text)

if match:

    print("Match found:", match.group())
```

- Metacharacters: Special characters that have a unique meaning and transform the search. For example, . matches any single character, while * matches zero or more occurrences of the preceding element.
  - . (dot): Matches any single character except newline \n.
  - ^: Asserts the start of a string.
  - $: Asserts the end of a string.

```
text = "hello world"

# Matches any three characters at the start of the string

match = re.search(r'^...', text)

if match:

    print("Three characters from start:", match.group())
```

```
# Matches any three characters at the end of the string

match = re.search(r'...$', text)

if match:

    print("Three characters from end:", match.group())
```

- Character Classes: Denoted by square brackets [], they match any one of the enclosed characters. For example, [abc] matches "a", "b", or "c".

```
text = "a bird"

# Matches any vowel

match = re.search(r'[aeiou]', text)

if match:

    print("First vowel found:", match.group())
```

- Quantifiers: Specify how many instances of a character, group, or character class must be present in the input for a match to be found. For example, a{2,3} matches "aa" or "aaa".
  - *: Matches 0 or more repetitions.
  - +: Matches 1 or more repetitions.
  - ?: Matches 0 or 1 repetition.
  - {n}: Matches exactly n repetitions.

```
text = "wooooow, that's amazing!"

# Matches the 'o' character appearing one or more times

match = re.search(r'o+', text)

if match:

    print("Sequence of 'o's found:", match.group())
```

## 2.2 Common Patterns

Understanding and recognizing common patterns can greatly enhance your ability to use regular expressions effectively:

- Digits: \d matches any digit, equivalent to [0-9].\

```
text = "The year is 2021"

# Finds the first sequence of digits

match = re.search(r'\d+', text)

if match:

  print("Year found:", match.group())
```

- Word Characters: \w matches any alphanumeric character plus underscore, equivalent to [A-Za-z0-9_].

```
text = "Variable_1 = 10"

# Matches the first word character sequence

match = re.search(r'\w+', text)

if match:

  print("First word found:", match.group())
```

- Whitespace: \s matches any space, tab, or newline character.

```
text = "Hello, world!"

# Splits the text at whitespace

words = re.split(r'\s', text)

print("Words split:", words)
```

## 2.3 Examples

- Matching an email address: \b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b
- Finding a URL: http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+

## 2.4 Advanced Topics

### 2.4.1 Lookahead and Lookbehind

Lookahead and lookbehind assertions enable you to create patterns that assert something to be true ahead or behind a certain point in the text, without including those conditions in the match.

Positive Lookahead (?=): Asserts that the specified pattern exists after the current position.

```
text = "John Smith (CEO), Jane Doe (CTO)"
# Matches names only if followed by "(CEO)"
matches = re.findall(r'\w+ \w+(?=\s\(CEO\))', text)
print("CEO Names:", matches)
```

Negative Lookahead (?!): Asserts that the specified pattern does not exist after the current position.

```
# Matches names not followed by "(CTO)"
matches = re.findall(r'\w+ \w+(?!\s\(CTO\))', text)
print("Non-CTO Names:", matches)
```

Positive Lookbehind (?<=): Asserts that the specified pattern exists before the current position.

```
prices = "USD 100, EUR 150"
# Matches amounts only if preceded by "USD"
matches = re.findall(r'(?<=USD\s)\d+', prices)
print("USD Amounts:", matches)
```

Negative Lookbehind (?<!): Asserts that the specified pattern does not exist before the current position.

```
# Matches amounts not preceded by "EUR"
matches = re.findall(r'(?<!EUR\s)\d+', prices)
```

*print("Non-EUR Amounts:", matches)*

### 2.4.2 Performance Optimization in Regex Operations

Optimizing regex can significantly improve the performance of your text processing tasks, especially when dealing with large datasets or complex patterns.

- Avoiding Greedy Quantifiers: Greedy quantifiers (*, +) can slow down your regex by consuming as much text as possible. Use non-greedy versions (*?, +?) when you want to match the smallest possible string.

  *# Non-greedy match for content within quotes*

  *text = '"First" and "Second"'*

  *matches = re.findall(r'"(.*?)"', text)*

  *print("Non-Greedy Matches:", matches)*

- Precompiling Patterns: If you're using the same pattern multiple times, precompile it with re.compile() to save time.

  *pattern = re.compile(r'\d+')*

  *# Use the precompiled pattern multiple times*

  *match1 = pattern.search('Item 123')*

  *match2 = pattern.search('Code 456')*

Character Classes vs. Alternation: Prefer character classes ([abc]) over alternation with single characters (a|b|c) for faster matches.

Specificity: Start your pattern with the most specific part to reduce the search space. For instance, if you know your pattern always starts with a letter, start with \w instead of ..

## 2.5 Best Practices

- Debugging: Tools like regex101.com can help test and debug your regular expressions with explanations.
- Readability: Use verbose mode (re.VERBOSE in Python) to add comments and whitespace to your regex for better readability.

- Security: Be cautious of ReDoS (Regular Expression Denial of Service) when crafting complex patterns, especially those used in user input validation.

## 2.6 Exercises

1. Validate an Email: Write a regex to validate an email address, ensuring it follows the standard email format (e.g., username@domain.com).

```
def validate_email(email):
    pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
    if re.match(pattern, email):
        return True
    return False
# Test the function
emails = ["test@example.com", "invalid-email", "hello@world.net"]
for email in emails:
    print(f"{email}: {'Valid' if validate_email(email) else 'Invalid'}")
```

2. Extract Dates: Create a regex to find dates in the format DD-MM-YYYY or DD/MM/YYYY in a given text.

```
text = "Important dates are 12-05-2021 and 23-11-2020."
dates = re.findall(r'\b\d{2}-\d{2}-\d{4}\b', text)
print("Dates found:", dates)
```

3. Phone Number Pattern: Develop a regex to match phone numbers in the format +1-234-567-8900 or (234) 567-8900.

```
text = "Contact us at +1-234-567-8900 or +1-321-543-6789."
phones = re.findall(r'\+\d-\d{3}-\d{3}-\d{4}', text)
print("Phone numbers found:", phones)
```

4. Extracting Conditional Data: Use lookahead assertions to extract all usernames from a text that are followed by the domain "@company.com".

```
emails = "john.doe@company.com, jane.smith@personal.com,
mike.brown@company.com"

matches = re.findall(r'\b\w+\.\w+(?=@company\.com)', emails)

print("Company Usernames:", matches)
```

5. Filtering Data with Lookbehinds: Use negative lookbehind assertions to find all dates in a document that are not preceded by the word "Deadline:".

```
text = "Deadline: 01-01-2021, Meeting: 05-02-2021, Deadline: 10-03-2021"

matches = re.findall(r'(?<!Deadline:\s)\d{2}-\d{2}-\d{4}', text)

print("Meeting Dates:", matches)
```

6. Optimizing Email Matching: Write an optimized regex to match email addresses within a large document, ensuring it's both efficient and accurate.

```
large_text = "..."  # Imagine a large document containing many email addresses

pattern = re.compile(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b')

matches = pattern.findall(large_text)

print("Emails found:", len(matches))
```

7. Non-Greedy HTML Tag Extraction: Use non-greedy quantifiers to extract content from HTML tags without including nested tags.

```
html = "<div>Hello <b>World</b>!</div>"

content = re.findall(r'<[^>]+>(.*?)</[^>]+>', html)

print("Tag Contents:", content)
```

# 3. TEXT PROCESSING IN PYTHON

## 3.1 Parsing and Extracting Data

Parsing involves systematically breaking down text into components to extract meaningful information. Python's built-in libraries and third-party modules offer robust tools for parsing various data formats, from simple CSV files to complex HTML pages.

### 3.1.1 Using Regular Expressions

Regular expressions are invaluable for parsing text. They enable pattern matching, searching, and data extraction.

**Example: Extracting Phone Numbers**

Suppose you have a block of text containing phone numbers mixed with other information. The goal is to extract all phone numbers formatted like XXX-XXX-XXXX.

```
import re

text = """
Call us at 123-456-7890 or 987-654-3210.
Visit us at 111-222-3333 for more details.
"""

pattern = r'\b\d{3}-\d{3}-\d{4}\b'

phone_numbers = re.findall(pattern, text)

print("Phone Numbers:", phone_numbers)
```

### 3.1.2 Parsing HTML with BeautifulSoup

For web data, BeautifulSoup from the bs4 package is a powerful tool for parsing HTML and extracting data.

**Example: Extracting Links from a Web Page**

```
from bs4 import BeautifulSoup
```

```
html = """
<html>
  <body>
    <a href="http://example.com/one">Link One</a>
    <a href="http://example.com/two">Link Two</a>
  </body>
</html>
"""

soup = BeautifulSoup(html, "html.parser")
links = [a['href'] for a in soup.find_all('a')]
print("Links:", links)
```

## 3.2 Automation Scripts

Python scripts can automate the mundane parts of text processing, such as data cleaning, formatting, and file conversion.

### 3.2.1 Data Cleaning Script

Imagine needing to clean a text file by removing unwanted characters and empty lines.

```
def clean_text_file(input_file, output_file):
  with open(input_file, "r") as infile, open(output_file, "w") as outfile:
    for line in infile:
      cleaned_line = re.sub(r'[^\w\s]', '', line)  # Remove punctuation
      if cleaned_line.strip():  # Skip empty lines
        outfile.write(cleaned_line)
```

```
clean_text_file("raw_data.txt", "cleaned_data.txt")
```

### 3.2.2 Format Conversion Script

Converting a CSV file to JSON format can be streamlined with a script.

```
import csv
import json
def csv_to_json(csv_file, json_file):
  with open(csv_file, 'r') as infile, open(json_file, 'w') as outfile:
    reader = csv.DictReader(infile)
    data = [row for row in reader]
    json.dump(data, outfile, indent=4)
csv_to_json("data.csv", "data.json")
```

## 3.3 Advanced Parsing Techniques

### 3.3.1 Using Parser Combinators

Parser combinators are higher-order functions that combine simpler parsers to build more complex and versatile parsers. They are particularly useful in functional programming languages but can also be implemented in Python.

**Example: Parsing Nested Structures**

Imagine parsing a simple, nested data format like (data (inner data)).

```
from parsy import string, seq, regex
lparen = string('(')
rparen = string(')')
data = regex(r'[^\s()]+')
def nested_parser():
  return (lparen >> parser().many() << rparen).map(tuple) | data
```

```
parser = nested_parser

result = parser.parse('(data (inner data))')

print(result)
```

This example uses the parsy library, which provides combinators like seq for sequencing and | for alternatives.

### 3.3.2 Lexing and Parsing with PLY

When dealing with programming languages or complex data formats, a two-step process of lexing (tokenizing) and parsing can be more effective. The Python Lex-Yacc (PLY) library is a popular choice for this approach.

**Example: Evaluating Mathematical Expressions**

```
import ply.lex as lex

import ply.yacc as yacc

tokens = ('NUMBER', 'PLUS', 'TIMES', 'LPAREN', 'RPAREN')

t_PLUS = r'\+'

t_TIMES = r'\*'

t_LPAREN = r'\('

t_RPAREN = r'\)'

t_NUMBER = r'\d+'

def t_error(t):
   print("Illegal character '%s'" % t.value[0])
   t.lexer.skip(1)

lexer = lex.lex()

def p_expression(p):
   '''expression : expression PLUS term
                 | term'''
```

```
    if len(p) == 4:
        p[0] = p[1] + p[3]
    else:
        p[0] = p[1]
def p_term(p):
    '''term : term TIMES factor
            | factor'''
    if len(p) == 4:
        p[0] = p[1] * p[3]
    else:
        p[0] = p[1]
def p_factor(p):
    '''factor : NUMBER
              | LPAREN expression RPAREN'''
    if len(p) == 2:
        p[0] = int(p[1])
    else:
        p[0] = p[2]
def p_error(p):
    print("Syntax error at '%s'" % p.value)
parser = yacc.yacc()
result = parser.parse('(2 + 3) * 4')
print(result)
```

This PLY example defines a lexer to tokenize mathematical expressions and a parser to evaluate them.

### 3.3.3 Parsing JSON with Recursive Descent

For formats like JSON, a recursive descent parser can be hand-written to parse the nested structure effectively.

**Example: Simple JSON Parser**

```
def parse_json(text):
  if text[0] == '{':
    obj, _ = parse_object(text, 1)
    return obj
  # Implement array, number, string, boolean, and null parsers
def parse_object(text, idx):
  obj = {}
  while text[idx] != '}':
    key, idx = parse_string(text, idx)
    idx = text.find(':', idx) + 1
    value, idx = parse_value(text, idx)
    obj[key] = value
    idx += text[idx] in ',}'
  return obj, idx
# Implement parse_array, parse_string, parse_number, parse_boolean, and parse_null
json_text = '{"name": "John", "age": 30, "is_student": false}'
parsed = parse_json(json_text)
print(parsed)
```

This simplified example illustrates the basic structure of a recursive descent parser for JSON. Actual implementations should handle all JSON features and edge cases.

## 3.4 Further Exploration

- Natural Language Processing (NLP): For text involving natural language, consider using NLP libraries like nltk or spaCy. These libraries provide advanced parsing capabilities tailored to human languages, including sentence segmentation, part-of-speech tagging, and named entity recognition.
- Machine Learning for Parsing: Machine learning models, especially those based on neural networks, have shown great promise in parsing and understanding complex text structures. Libraries like TensorFlow and PyTorch, coupled with models from transformers, can be used for tasks like dependency parsing and syntactic analysis.

## 3.5 Hands-on Exercises

1. Log File Parsing: Given a server log file, write a script to extract and count the occurrences of HTTP status codes.

```
import re
from collections import Counter
# Initialize a Counter to store status code occurrences
status_code_counter = Counter()
# Open and read the log file
with open('server_log.txt', 'r') as file:
    for line in file:
        # Extract the status code using regex
        match = re.search(r'\s(\d{3})\s', line)
        if match:
            # Increment the count for this status code
            status_code = match.group(1)
            status_code_counter[status_code] += 1
# Print the count of each status code
```

```
for code, count in status_code_counter.items():
    print(f"Status Code {code}: {count} occurrences")
```

2. Email Extraction from Documents: Create a script that scans through a document and extracts all email addresses into a list.

```
import re
# Regex pattern for matching email addresses
email_pattern = re.compile(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b')
# List to store found email addresses
emails = []
# Read the document and find email addresses
with open('document.txt', 'r') as file:
    for line in file:
        emails.extend(email_pattern.findall(line))
# Write the extracted emails to a new file
with open('emails.txt', 'w') as outfile:
    for email in emails:
        outfile.write(email + '\n')
print(f"Extracted {len(emails)} email addresses.")
```

3. Automating Data Cleanup: Automate the cleanup process for a dataset containing product reviews, where you need to remove URLs, special characters, and numbers, keeping only text.

```
import re
def clean_review(text):
    # Remove URLs
```

```
    text = re.sub(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+', '', text)

    # Remove special characters and numbers

    text = re.sub(r'[^a-zA-Z\s]', '', text)

    return text

# Read the raw reviews, clean them, and write to a new file

with open('reviews_raw.txt', 'r') as infile, open('reviews_cleaned.txt', 'w') as outfile:

    for line in infile:

        cleaned_line = clean_review(line)

        outfile.write(cleaned_line + '\n')

print("Data cleanup complete.")
```

4. Build a Parser for Custom Data Format

For this exercise, we'll build a simple parser for a custom XML-like data format using PLY. The custom format is simplified and will not include attributes or nested elements for brevity.

Custom Data Format Example:

```
<data>

    <name>John Doe</name>

    <age>30</age>

    <city>New York</city>

</data>
```

PLY Parser Implementation:

First, install PLY if you haven't already:

```
pip install ply
```

Now, let's define the lexer and parser:

```
import ply.lex as lex
import ply.yacc as yacc
# Define tokens
tokens = ('TAG_OPEN', 'TAG_CLOSE', 'DATA', 'TEXT')
# Token rules
t_TAG_OPEN = r'<\w+>'
t_TAG_CLOSE = r'</\w+>'
t_DATA = r'<\w+>|</\w+>'
t_TEXT = r'[^<>]+
# Ignored characters (spaces and newlines)
t_ignore = ' \t\n'
def t_error(t):
   print(f"Illegal character {t.value[0]}")
   t.lexer.skip(1)
lexer = lex.lex()
# Parsing rules
def p_document(p):
   '''document : element
            | document element'''
   p[0] = {p[1]['tag']: p[1]['text']} if len(p) == 2 else {**p[1], **p[2]}
def p_element(p):
   'element : TAG_OPEN TEXT TAG_CLOSE'
   p[0] = {'tag': p[1][1:-1], 'text': p[2]}
def p_error(p):
```

```
    print(f"Syntax error at {p.value}")

parser = yacc.yacc()

# Parsing the custom XML-like data

data = """<name>John Doe</name>

<age>30</age>

<city>New York</city>"""

result = parser.parse(data)

print(result)
```

This parser will convert the custom XML-like data into a Python dictionary.

5. Recursive Descent Parser for Arithmetic Expressions

We'll extend the previous PLY example to support subtraction (-) and division (/), with correct handling of operator precedence.

```
    tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN')

t_PLUS = r'\+'

t_MINUS = r'-'

t_TIMES = r'\*'

t_DIVIDE = r'/'

t_LPAREN = r'\('

t_RPAREN = r'\)'

t_NUMBER = r'\d+'

def t_error(t):

    print("Illegal character '%s'" % t.value[0])

    t.lexer.skip(1)

lexer = lex.lex()
```

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]
def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
def p_expression_term(p):
    'expression : term'
    p[0] = p[1]
def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]
def p_term_divide(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]
def p_term_factor(p):
    'term : factor'
    p[0] = p[1]
def p_factor_num(p):
    'factor : NUMBER'
    p[0] = int(p[1])
def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]
```

```
def p_error(p):
    print("Syntax error at '%s'" % p.value)

parser = yacc.yacc()

result = parser.parse("7 + 3 * (10 / (12 / (3 + 1) - 1))")

print(result)
```

This parser correctly handles arithmetic expressions with precedence for multiplication (*), division (/), addition (+), and subtraction (-).

6. NLP Named Entity Extraction

For this exercise, we'll use spaCy to extract named entities from a piece of text. Ensure you have spaCy installed and have downloaded the necessary language model:

```
pip install spacy

python -m spacy download en_core_web_sm
```

Now, let's extract named entities:

```
import spacy

# Load the spaCy model

nlp = spacy.load("en_core_web_sm")

# Sample text

text = "Apple is looking at buying a U.K. startup for $1 billion. John Smith and Jane Doe will be visiting New York."

# Process the text

doc = nlp(text)

# Extract named entities

for ent in doc.ents:
    print(f"{ent.text} ({ent.label_})")
```

This script will print out named entities along with their categories, such as organizations, monetary values, persons, and geopolitical entities.

# 4. STRING FORMATTING IN PYTHON

## 4.1 Old-Style Formatting (% Operator)

Old-style formatting uses the % operator, reminiscent of the sprintf function in C. It's less flexible and considered outdated, but still appears in legacy code.

Example:

```
name = "Alice"
age = 30
print("Hello, %s. You are %d years old." % (name, age))
```

Use Cases:

- Maintaining legacy code.
- Simple formatting needs where newer methods aren't available.

## 4.2 New-Style Formatting (str.format() Method)

Introduced in Python 2.6, str.format() offers enhanced flexibility and readability over the old-style formatting.

Example:

```
print("Hello, {}. You are {} years old.".format(name, age))
# With positional arguments
print("Hello, {1}. {0} is waiting.".format("Bob", name))
# With keyword arguments
print("Hello, {name}. You are {age} years old.".format(name=name, age=age))
```

Use Cases:

- Complex formatting needs with positional and keyword arguments.
- Where template-like formatting is required.

## 4.3 f-Strings (Formatted String Literals)

f-Strings, introduced in Python 3.6, provide a concise and readable way to include expressions inside string literals.

Example:

```
print(f"Hello, {name}. You are {age} years old.")
```

Use Cases:

- Readability and conciseness are priorities.
- Injecting expressions and calling functions directly within placeholders.

## 4.4 Comparisons and Performance

Readability: f-Strings are the most readable, especially with embedded expressions or function calls. str.format() is more verbose but clearer than old-style formatting for complex cases.

Performance: f-Strings generally offer the best performance as they are evaluated at runtime without the overhead of function calls or parsing format strings.

Versatility: str.format() provides a high level of control over formatting, with features like padding, alignment, numbering formats, and more, which can be more cumbersome with old-style formatting.

## 4.5 Practical Exercises

1. Formatting a Date: Use all three methods to format a date (e.g., year, month, day) into a human-readable string.

```
year, month, day = 2023, 2, 15

# Old-style

print("Old-style: %d-%02d-%02d" % (year, month, day))

# str.format()

print("str.format(): {:d}-{:02d}-{:02d}".format(year, month, day))

# f-String
```

*print(f"f-String: {year}-{month:02d}-{day:02d}")*

2. Complex Nested Formatting: Create a complex nested structure and format it using str.format() and f-strings, comparing readability.

*person = {"name": "Alice", "job": {"title": "Engineer", "department": "Development"}}*

*# str.format()*

*print("str.format(): {p[name]} is a {p[job][title]} in {p[job][department]}.".format(p=person))*

*# f-String*

*print(f"f-String: {person['name']} is a {person['job']['title']} in {person['job']['department']}.")*

3. Performance Comparison: Write a simple benchmark to compare the performance of the three formatting methods using the timeit module.

*import timeit*

*# Old-style*

*old_style = "Hello, %s. You are %d." % ("Alice", 30)*

*# str.format()*

*new_style = "Hello, {}. You are {}.".format("Alice", 30)*

*# f-String*

*f_string = f"Hello, {'Alice'}. You are {30}."*

*print("Old-style:", timeit.timeit(lambda: old_style, number=1000000))*

*print("str.format():", timeit.timeit(lambda: new_style, number=1000000))*

*print("f-String:", timeit.timeit(lambda: f_string, number=1000000))*

## 5. SUMMARY

Throughout this unit on Regular Expressions and Text Processing, we've embarked on a comprehensive exploration of how to effectively manage and manipulate text data in Python. Starting with the basics of regular expressions, we delved into their syntax and patterns, unlocking the power of pattern matching to search, replace, and parse complex text data efficiently. We examined various metacharacters, quantifiers, and character classes, equipping you with the tools to craft intricate regex patterns for a wide array of text processing tasks.

Progressing into text processing, we explored parsing techniques and data extraction methods, emphasizing the practical application of Python's string manipulation capabilities and regex in automating common text-related tasks. From parsing log files and extracting useful information like HTTP status codes to automating data cleanup in datasets containing product reviews, we covered a spectrum of real-world scenarios where text processing skills are invaluable.

Furthermore, we ventured into advanced topics, such as parser combinators and the use of libraries like PLY for lexing and parsing more structured data formats, demonstrating the versatility of Python in handling a variety of text processing needs. We also touched on the significance of named entity recognition using spaCy, showcasing how natural language processing (NLP) techniques can be applied to extract meaningful information from text.

The unit was rich with practical exercises, encouraging hands-on practice and reinforcing learning through real-world examples. From building parsers for custom data formats to exploring advanced regex features like lookahead and lookbehind assertions, the exercises were designed to challenge and enhance your text processing skills.

In summary, this unit has provided a solid foundation in regular expressions and text processing, arming you with the knowledge and skills to tackle text-related challenges in Python programming confidently. Whether it's data parsing, validation, transformation, or analysis, the concepts covered in this unit are fundamental to becoming proficient in text processing and manipulation, opening the door to more advanced data handling and analysis tasks in your programming endeavors.

## 6. GLOSSARY

- ***Regular Expressions (Regex):*** A sequence of characters that forms a search pattern, used for string searching and manipulation.
- ***Metacharacters:*** Special characters in a regular expression that carry out specific functions, such as . (any character), ^ (start of string), $ (end of string), etc.
- ***Quantifiers:*** Symbols or characters in a regular expression that specify how many instances of a preceding element are required for a match, including * (zero or more), + (one or more), ? (zero or one), and {n} (exactly n times).
- ***Character Classes:*** Denoted by square brackets [], character classes match any one character from a set of characters defined inside the brackets, e.g., [a-z] matches any lowercase letter.
- ***Grouping:*** The use of parentheses () in a regular expression to group parts of the pattern together.
- ***Greedy Matching:*** The default behavior of quantifiers in regular expressions, where they match as many instances of the pattern as possible.
- ***Non-Greedy (Lazy) Matching:*** Modified behavior of quantifiers, making them match as few instances as possible, typically denoted by adding a ? after the quantifier, e.g., *?.
- ***Lookahead and Lookbehind:*** Zero-length assertions in regular expressions that check for the presence or absence of a pattern before or after the current position without including it in the match.
- ***Parser Combinators:*** High-order functions that combine simpler parsers into more complex ones, used for parsing structured data.
- ***Lexing (Tokenizing):*** The process of breaking down a string into a sequence of tokens, which are meaningful sequences of characters, such as words or numbers.
- ***Parsing:*** The process of analyzing a string or sequence of tokens to determine its grammatical structure with respect to a given set of rules.

- ***PLY (Python Lex-Yacc):*** A Python library for lexing and parsing, allowing the creation of complex parsers by defining tokens and grammatical rules.
- ***BeautifulSoup:*** A Python library for parsing HTML and XML documents, commonly used for web scraping.
- ***spaCy:*** An advanced natural language processing (NLP) library in Python, used for tasks like tokenization, part-of-speech tagging, and named entity recognition.
- ***String Formatting:*** The process of inserting values or variables into placeholders within a string, including old-style formatting (% operator), str.format() method, and f-strings (formatted string literals).
- ***Context Managers:*** Python constructs that provide a convenient way to allocate and release resources properly, commonly used with file operations to ensure that files are closed after their use is complete.

## 7. SELF-ASSESSMENT QUESTIONS

1. What does the regex pattern \d{2,4} match?
   A) Two to four digits
   B) Exactly 2 or exactly 4 digits
   C) Any string of two to four characters
   D) A string containing "d{2,4}"
2. Which metacharacter is used to match the start of a string?
   A) $
   B) ^
   C) *
   D) +
3. What is the function of the ? in regex patterns?
   A) Match exactly one character
   B) Match zero or one occurrence of the preceding element
   C) Make the search case-insensitive
   D) Escape special characters
4. What Python library is commonly used for parsing HTML documents in web scraping?
   A) re
   B) numpy
   C) beautifulsoup
   D) requests
5. In Python, which method is used to read an entire text file into a single string?
   A) read()
   B) readline()
   C) readlines()
   D) open()
6. Which of the following is a common use case for text processing in Python?
   A) Compiling programs
   B) Creating GUI applications
   C) Data analysis and manipulation
   D) Game development

7. Which string formatting method in Python allows the use of named placeholders?
   A) Old-style formatting (% operator)
   B) str.format() method
   C) f-strings
   D) Both B and C
8. What is the main advantage of using f-strings in Python?
   A) They are faster to write
   B) They offer better performance
   C) They support only numeric data types
   D) They are backward compatible with Python 2
9. In the str.format() method, how do you specify a variable to have a minimum width of 10 characters?
   A) {variable:10}
   B) {variable:w10}
   C) {variable:10w}
   D) {variable,width=10}
10. What is lookahead in regular expressions used for?
   A) To reverse the direction of the search
   B) To check for patterns ahead of the current match without including it
   C) To repeat the search multiple times
   D) To look for patterns behind the current match
11. In parsing, what does "lexing" refer to?
   A) Parsing based on lexical rules
   B) Breaking the text into a list of sentences
   C) Breaking the text into tokens
   D) Correcting syntax errors in the source text
12. Which tool is used in Python for lexing and parsing tasks?
   A) BeautifulSoup
   B) spaCy
   C) PLY
   D) NumPy

13. When processing log files, which Python function is most suitable for handling multiple files in a directory?
    A) open()
    B) read()
    C) glob.glob()
    D) write()
14. Which approach is recommended for safely opening and working with files in Python to ensure they are properly closed?
    A) Using the open() function with manual close()
    B) Using context managers with the with statement
    C) Opening files in 'r+' mode only
    D) Handling files exclusively in binary mode
15. How can you improve the performance of a regular expression pattern in Python?
    A) By using non-greedy quantifiers
    B) By writing longer patterns
    C) By avoiding the use of groups
    D) By using the re.IGNORECASE flag

## 8. TERMINAL QUESTIONS

1. How do you write a regex to match any word character followed by an exclamation mark?
2. Explain the difference between * and + quantifiers in regex.
3. Write a regex pattern to validate a simple email address format.
4. Describe how you would use a character class to match only vowels in a string.
5. Explain the use of the ^ metacharacter in regex patterns.
6. Demonstrate how to use Python's re module to split a string by multiple delimiters.
7. Outline a method to read a large text file in Python and search for a specific word or phrase.
8. Explain how to use BeautifulSoup to extract all hyperlinks from an HTML page.
9. Describe the process of using Python to convert a CSV file to a JSON format.
10. Discuss the approach to remove all HTML tags from a given string using regular expressions.
11. Compare and contrast the use of old-style formatting (% operator) with the str.format() method.
12. Provide an example of how to use f-strings to embed expressions within string literals.
13. Discuss the implications of using f-strings for dynamic expression evaluation and its impact on readability and performance.
14. How would you format a floating-point number to two decimal places using f-strings?
15. Explain how to use the str.format() method to align text within a fixed width.
16. Describe what lookahead and lookbehind assertions are in regex and provide an example of each.
17. Explain how parser combinators can be used to build a parser for a custom mini-language or data format.
18. Outline the steps to create a lexer and parser using PLY for a simple arithmetic expression evaluator.
19. Discuss the use of spaCy for named entity recognition and provide an example of extracting person names from text.

20. How can you improve regex performance, especially when working with large datasets or complex patterns?

## 9. ANSWERS

**Self-Assessment Questions**

1. Answer: A) Input/Output
2. Answer: C) open()
3. Answer: C) A file object
4. Answer: C) 'r'
5. Answer: B) Binary
6. Answer: C) 'a'
7. Answer: B) It automatically closes the file
8. Answer: A) with open('file.txt', 'r') as file: content = file.read()
9. Answer: C) By ensuring resources are released properly even if an error occurs
10. Answer: A) 'r+'
11. Answer: B) Python will throw a FileNotFoundError
12. Answer: B) read()
13. Answer: B) Using a try-except block
14. Answer: C) The file is opened for both reading and writing, and existing content is erased
15. Answer: C) It ensures that the file is automatically closed after the block of code is executed

**Terminal Questions**

1. Refer to the introduction to Unit 6
2. See "Text Files in Python" and "Binary Files in Python"
3. Refer to "Introduction to File I/O"
4. See "Modes for Text Files" and "Modes for Binary Files"
5. to "Modes for Binary Files"
6. Refer to "Modes for Text Files" and "Modes for Binary Files"
7. See "The with Statement in File I/O"
8. Refer to "Advantages of Using with"
9. See "Real-world Application Scenarios"
10. See practical examples in "Introduction to File I/O"
11. Refer to practical examples in "File Modes"

12. Refer practical examples in "The with Statement"
13. See practical examples in "The with Statement"
14. Refer to the summary of Unit 6
15. to "Error Handling and Best Practices"
16. Refer to "Introduction to File I/O"
17. See "Overview of File Modes"
18. Refer to practical examples in "The with Statement"
19. See practical examples in "Introduction to File I/O" and "File Modes"
20. Refer to "Text Files in Python"