**WEEK-3**

**Functional Programming and Data Parsing**

**DESCRIPTION**

This unit focuses on advanced core Python concepts, primarily revolving around functional programming and data parsing techniques using JSON, XML, and regular expressions.

Functional programming in Python emphasizes the use of functions as first-class citizens. This approach promotes code reusability, clarity, and brevity. Key concepts include:

* **Lambda Functions**: Anonymous functions defined with the lambda keyword. Useful for short operations, especially when used in conjunction with higher-order functions.
* **Map**: Applies a function to all the items in an input list (or any iterable).
* **Filter**: Filters elements of an iterable based on a function that returns either True or False.
* **Reduce**: Performs a rolling computation to sequential pairs of elements. Requires the functools module.
* **Iterators**: Objects that can be iterated upon and implement \_\_iter\_\_() and \_\_next\_\_() methods.
* **Generators**: A simpler way to create iterators using functions and the yield keyword. They are memory efficient as values are produced on the fly.
* **List Comprehensions**: A concise way to create lists using a single line of code with conditions and loops embedded.

The unit also introduces **parsing and handling data formats** commonly used in real-world applications:

* **JSON (JavaScript Object Notation)**: A lightweight data-interchange format. Python’s json module allows conversion between JSON and Python dictionaries using json.dumps(), json.loads(), etc.
* **XML (eXtensible Markup Language)**: Python’s xml.etree.ElementTree provides tools to parse and manipulate XML data.
* **Regular Expressions**: Powerful for pattern matching and text manipulation using Python’s re module. They help in tasks like validating email formats, extracting dates, phone numbers, and more.

By the end of this unit, students will be able to write concise functional code, process real-world data formats like JSON and XML, and manipulate text using regular expressions.

**Write a program for the following**

1. **To Validate Email Addresses**
2. **To Extract Dates from a Text**
3. **to Replace Multiple Spaces with a Single Space**
4. **To Extract Phone Numbers from a String**

**Find All Words Starting with a Specific Letter**

**Description:**

This experiment demonstrates the power of regular expressions (re module) in pattern matching and string processing. The tasks simulate common real-world problems such as validating input, extracting data, and cleaning text.

**Program:**

import re

# 1. Validate email addresses

email = "test@example.com"

pattern\_email = r'^[\w\.-]+@[\w\.-]+\.\w+$'

print("Valid Email:" if re.match(pattern\_email, email) else "Invalid Email")

# 2. Extract dates

text\_with\_dates = "The events are scheduled on 12-04-2024, 25-12-2025 and 01-01-2026."

dates = re.findall(r'\d{2}-\d{2}-\d{4}', text\_with\_dates)

print("Extracted Dates:", dates)

# 3. Replace multiple spaces with a single space

sentence = "This is a sentence with extra spaces."

cleaned = re.sub(r'\s+', ' ', sentence)

print("Cleaned Sentence:", cleaned)

# 4. Extract phone numbers

phones = "Contact: 9876543210, 8123456789"

phone\_numbers = re.findall(r'\b[6-9]\d{9}\b', phones)

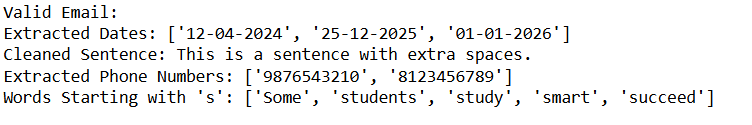
print("Extracted Phone Numbers:", phone\_numbers)

# 5. Find words starting with a specific letter (e.g., 's')

text = "Some students study smart to succeed."

words\_with\_s = re.findall(r'\bs\w\*', text, re.IGNORECASE)

print("Words Starting with 's':", words\_with\_s)

**Output:**

**Additional Programs**

**Write a program to demonstrate the use of generators and the yield keyword for producing a sequence of Fibonacci numbers.**

**Description:**

This experiment introduces Python generators, which are used to produce a stream of values using the yield keyword. The program generates Fibonacci numbers one at a time without storing the entire sequence in memory, showing the benefit of lazy evaluation.

**Program:**

def fibonacci\_generator(limit):

a, b = 0, 1

while a < limit:

yield a

a, b = b, a + b

for num in fibonacci\_generator(100):

print(num, end=" ")

**Output:**

****

**Write a program using iterators and demonstrate how to build a custom iterator class to iterate over a list in reverse.**

**Description:**

This experiment focuses on creating a custom **iterator class** in Python by implementing the \_\_iter\_\_() and \_\_next\_\_() methods. It helps students understand how iteration protocols work in Python and how to control the iteration flow manually.

**Program:**

class ReverseIterator:

def \_\_init\_\_(self, data):

self.data = data

self.index = len(data)

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.index == 0:

raise StopIteration

self.index -= 1

return self.data[self.index]

# Test the iterator

my\_list = [10, 20, 30, 40]

rev\_iter = ReverseIterator(my\_list)

for item in rev\_iter:

print(item)

**Output:**

****

**VIVA QUESTIONS**

**UNIT – 1: Viva Questions with Answers**

1. **What is the difference between iter() and next() in Python?**

iter() returns an iterator object from an iterable, while next() retrieves the next item from the iterator. When there are no more items, next() raises a StopIteration exception.

**2. What is a generator expression? How is it different from a list comprehension?**

A generator expression is similar to a list comprehension but uses () instead of []. It generates items lazily, meaning values are produced one at a time as needed, making it more memory efficient.

**3. What are the benefits of using functional programming in Python?**

Functional programming leads to concise, modular, and easier-to-test code. It emphasizes immutability and avoids side effects, making it more predictable and parallelizable.

**4. Give an example of a regular expression to extract all digits from a string**.

import re

text = "Order 12345 was placed on 2024-04-08"

digits = re.findall(r'\d+', text)

print(digits) # Output: ['12345', '2024', '04', '08']