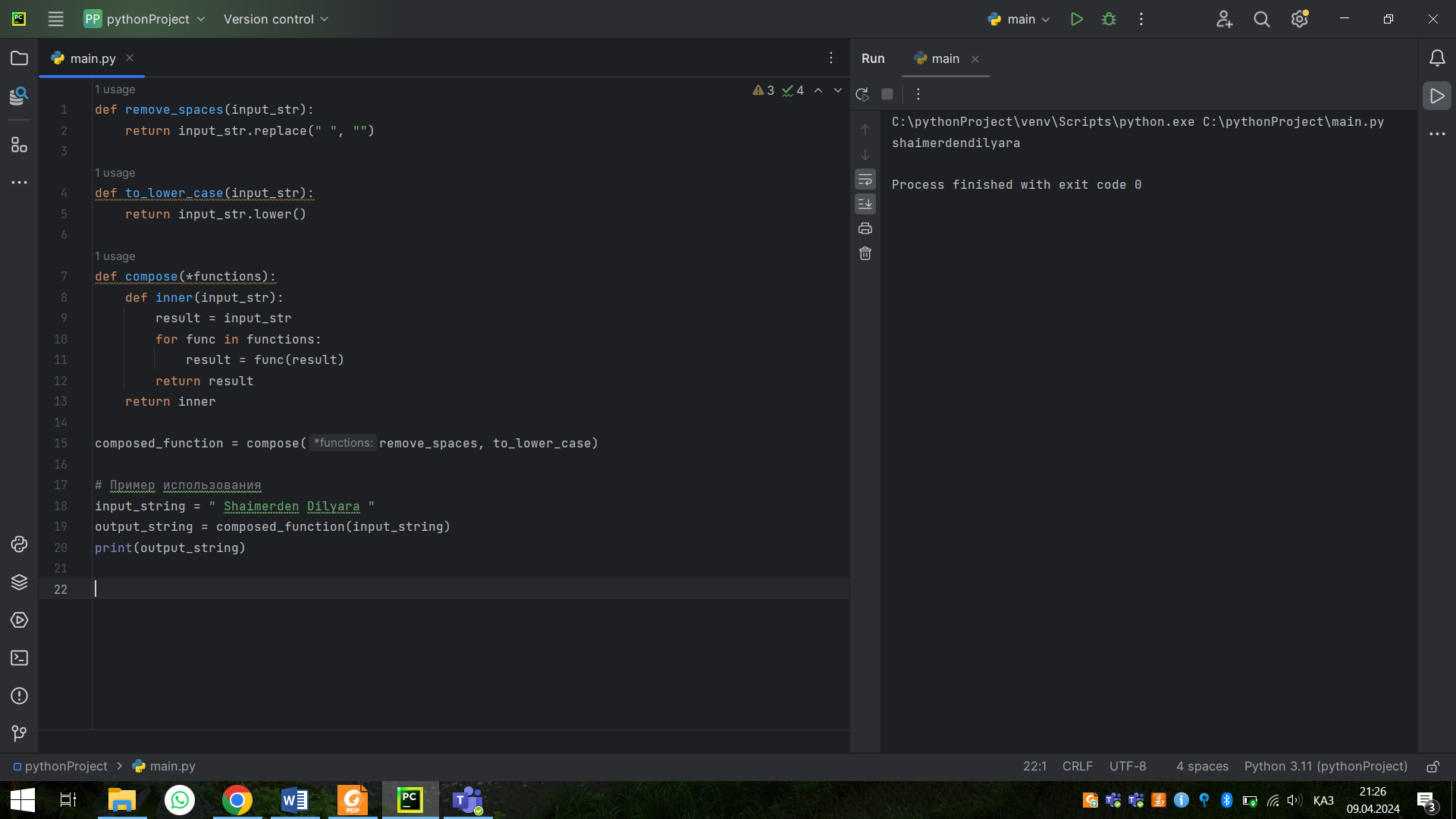
Лабораторная Работа 7: Создание Модульного Кода с ИспользованиемФункциональных Принципов

2. Композиция Функций для Обработки Строк- Реализовать функции для преобразования строк (например, убрать пробелы,привести к нижнему регистру) и создать композицию этих функций.

def remove\_spaces(input\_str):  
 return input\_str.replace(" ", "")  
  
def to\_lower\_case(input\_str):  
 return input\_str.lower()  
  
def compose(\*functions):  
 def inner(input\_str):  
 result = input\_str  
 for func in functions:  
 result = func(result)  
 return result  
 return inner  
  
composed\_function = compose(remove\_spaces, to\_lower\_case)  
  
# Пример использования  
input\_string = " Shaimerden Dilyara "  
output\_string = composed\_function(input\_string)  
print(output\_string)



1. \*Модульность в программировании\* подразумевает разбиение большой системы на отдельные компоненты (модули), каждый из которых отвечает за конкретную функциональность. Преимущества модульности включают повышение читаемости кода, упрощение поддержки и разработки, уменьшение сложности системы за счет ее разделения на более мелкие части, а также повышение переиспользуемости кода.

2. Для реализации \*\*композиции функций в Python\*\* можно использовать высокоуровневые функции или декораторы. Это подходит для решения задач, связанных с обработкой данных или выполнением последовательности действий над ними, когда требуется создать новую функцию, которая объединяет функциональность нескольких других функций.

3. В \*\*лабораторной работе\*\* использовались различные методы асинхронного программирования, такие как asyncio в Python для создания асинхронных функций и await/async операторы для ожидания выполнения асинхронных задач.

4. \*\*Асинхронное программирование\*\* может интегрироваться с функциональным стилем в Python через использование функциональных паттернов и подходов, таких как композиция функций и передача функций в качестве аргументов.

5. При \*\*модульном программировании\*\* могут возникнуть проблемы, такие как сложность управления зависимостями между модулями, избыточное дублирование кода и сложность тестирования. Эти проблемы могут быть решены путем использования принципов SOLID, правильного проектирования интерфейсов и абстракций, а также разделения функциональности на независимые модули.

6. \*\*Композиция функций\*\* позволяет создавать более модульный и многократно используемый код, так как позволяет объединять маленькие функции в более крупные и сложные функции, не нарушая принцип единой ответственности.

7. \*\*Асинхронное программирование\*\* может быть нецелесообразным, когда задачи не имеют параллельной природы или когда асинхронная модель внесет больше сложности, чем преимуществ.

8. \*\*Управление состоянием в асинхронных приложениях\*\* может быть осуществлено через использование объектов состояния, передачу состояния через параметры функций или использование специальных библиотек для управления асинхронным состоянием.

9. \*\*Функциональные паттерны\*\* это шаблоны проектирования, которые используют функциональные концепции, такие как замыкания и высокоуровневые функции. Они могут использоваться для управления состоянием, обработки данных и реализации асинхронных операций.

10. Для \*\*асинхронного программирования в Python\*\* используются такие инструменты и библиотеки, как asyncio, aiohttp, aiofiles и др.

11. \*\*Модульный подход\*\* упрощает тестирование и отладку программ, так как позволяет изолировать части кода для тестирования и уменьшает взаимозависимость между компонентами системы.

12. \*\*Реализация асинхронности в функциональном стиле\*\* может столкнуться с вызовами, связанными с управлением состоянием и организацией асинхронных операций в виде чистых функций.

13. \*\*Функциональное программирование\*\* способствует написанию чистого асинхронного кода путем предоставления четких границ между состоянием и операциями, а также путем использования функциональных паттернов для управления асинхронными операциями.

14. Ключевые вызовы при \*\*композиции асинхронных функций\*\* могут включать управление потоком управления, управление состоянием и организацию асинхронных операций в цепочку выполнения.

15. Для \*\*улучшения архитектуры модульного асинхронного приложения\*\* можно использовать четкое определение интерфейсов между модулями, разделение сложных операций на более простые компоненты, а также использование асинхронных паттернов проектирования, таких как "Future" или "Promise".