|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Язык программирования Python»  Вариант 9 | | |
|  | | Работу выполнил  студент группы ИТ-6-2024 1 курса  Плеханов Д. В.  «12» июня 2025 г. |
| Работу проверил  \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фамилия И.О.  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2024 г. |
| Пермь 2025 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#_Toc153130027)

[Алгоритм решения 3](#_Toc153130028)

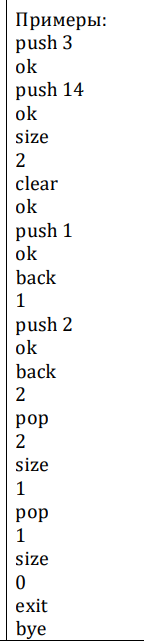
[Тестирование 4](#_Toc153130029)

[Код программы 13](#_Toc153130030)

[Инструкция по применению стилей и оформлению работы 17](#_Toc153130031)

# Постановка задачи

# Реализуйте структуру данных "стек". Напишите программу, содержащую описание стека и моделирующую работу стека, реализовав все указанные здесь методы. Программа считывает последовательность команд и в зависимости от команды выполняет ту или иную операцию. После выполнения каждой команды программа должна вывести одну строчку. Возможные команды для программы: push n - Добавить в стек число n (значение n задается после команды). Программа должна вывести ok. pop - Удалить из стека последний элемент. Программа должна вывести его значение. back - Программа должна вывести значение последнего элемента, не удаляя его из стека. size - Программа должна вывести количество элементов в стеке. clear - Программа должна очистить стек и вывести ok. exit - Программа должна вывести bye и завершить работу. Входные данные: команды управления стеком вводятся в описанном ранее формате по 1 на строке. Гарантируется, что набор входных команд удовлетворяет следующим требованиям: максимальное количество элементов в стеке в любой момент не превосходит100, все команды pop и back корректны, то есть при их исполнении в стеке содержится хотя бы один элемент. Выходные данные: протокол работы со стеком, по 1 сообщению в строке.



# Алгоритм решения

# Постановка задачи

# Требуется реализовать структуру данных "стек" и программу, моделирующую его работу. Программа должна обрабатывать следующие команды:

# push n – добавить число n в стек (вывод "ok")

# pop – удалить и вернуть последний элемент стека (вывод значения)

# back – вернуть последний элемент стека без удаления (вывод значения)

# size – вывести количество элементов в стеке

# clear – очистить стек (вывод "ok")

# exit – завершить работу программы (вывод "bye")

# Ограничения:

# Максимальное количество элементов в стеке – 100.

# Команды pop и back всегда корректны (стек не пуст).

Подробный алгоритм решения задачи "Реализация стека"

1. \*\*Анализ задачи и проектирование структуры данных\*\*

- \*\*Стек\*\* - структура данных, работающая по принципу LIFO (Last In, First Out).

- \*\*Требуемые операции\*\*:

- Добавление элемента (`push`)

- Удаление и возврат последнего элемента (`pop`)

- Просмотр последнего элемента без удаления (`back`)

- Получение размера стека (`size`)

- Очистка стека (`clear`)

- \*\*Ограничения\*\*:

- Максимальный размер стека - 100 элементов

- Гарантируется корректность команд `pop` и `back` (стек не пуст)

2. \*\*Проектирование класса Stack\*\*

- \*\*Атрибуты\*\*:

- `\_\_data` - список для хранения элементов (инкапсуляция)

- `\_\_size` - счетчик элементов

- \*\*Методы\*\*:

1. `\_\_init\_\_()` - инициализация пустого стека

2. `push(n)`:

- Проверка на переполнение (если размер ≥ 100 → исключение)

- Добавление элемента в конец списка

- Увеличение счетчика `\_\_size`

3. `pop()`:

- Проверка на пустоту стека (если `\_\_size` == 0 → исключение)

- Уменьшение счетчика `\_\_size`

- Возврат и удаление последнего элемента

4. `back()`:

- Проверка на пустоту стека

- Возврат последнего элемента без удаления

5. `size()`:

- Возврат текущего значения `\_\_size`

6. `clear()`:

- Очистка списка `\_\_data`

- Сброс счетчика `\_\_size` в 0

3. \*\*Проектирование обработки команд\*\*

- \*\*Основной цикл программы\*\*:

1. Чтение ввода пользователя

2. Разделение строки на команду и аргументы

3. Обработка возможных ошибок ввода

4. Выполнение соответствующей операции

5. Вывод результата

- \*\*Логика обработки команд\*\*:

- `push n`:

1. Проверка наличия аргумента

2. Проверка, что аргумент - число

3. Вызов `stack.push(n)`

4. Вывод "ok"

- `pop`:

1. Вызов `stack.pop()`

2. Вывод удаленного элемента

- `back`:

1. Вызов `stack.back()`

2. Вывод последнего элемента

- `size`:

1. Вызов `stack.size()`

2. Вывод количества элементов

- `clear`:

1. Вызов `stack.clear()`

2. Вывод "ok"

- `exit`:

1. Вывод "bye"

2. Завершение программы

- Неизвестная команда:

1. Вывод сообщения об ошибке

4. \*\*Обработка ошибок\*\*

- \*\*Типы возможных ошибок\*\*:

1. Некорректный аргумент для `push` (не число)

2. Отсутствие аргумента для `push`

3. Переполнение стека (при добавлении 101-го элемента)

4. Пустой стек при `pop` или `back` (хотя по условию не должно возникать)

- \*\*Механизм обработки\*\*:

- Использование блоков `try-except`

- Генерация понятных сообщений об ошибках

5. \*\*Детальная последовательность выполнения\*\*

1. Создание экземпляра класса `Stack`

2. Запуск бесконечного цикла:

a. Ожидание ввода команды

b. Разделение введенной строки на части

c. Определение типа команды

d. Для `push`:

- Проверка количества аргументов

- Попытка преобразования аргумента в число

- Проверка переполнения стека

- Вызов метода `push`

e. Для других команд - вызов соответствующих методов

f. Перехват и обработка возможных исключений

g. Вывод результата операции

3. При получении команды `exit` - выход из цикла

6. \*\*Особенности реализации\*\*

- \*\*Инкапсуляция данных\*\*:

- Использование приватных атрибутов `\_\_data` и `\_\_size`

- \*\*Модульность\*\*:

- Вынесение логики стека в отдельный класс

- \*\*Гибкость\*\*:

- Возможность легко расширить функционал (например, добавить проверку на переполнение)

- \*\*Защита от дурака\*\*:

- Проверка всех возможных ошибок ввода

- Генерация понятных сообщений об ошибках

7. \*\*Блок-схема алгоритма\*\* (текстовое описание)

1. Начало программы

2. Инициализация стека

3. Цикл:

a. Ввод команды

b. Если команда == "push":

- Проверить аргумент

- Добавить в стек

- Вывести "ok"

c. Если команда == "pop":

- Удалить элемент

- Вывести его значение

d. ... (аналогично для других команд)

e. Если команда == "exit":

- Вывести "bye"

- Выйти из цикла

f. Иначе:

- Сообщение о неизвестной команде

4. Конец программы

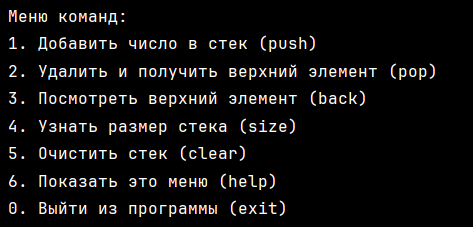
8. \*\*Оптимизации\*\*

- Быстрый доступ к последнему элементу через индексацию списка

- Эффективное управление памятью (использование списка)

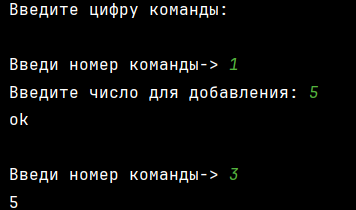
- Минимальные проверки для команд `pop` и `back` (по условию они всегда корректны)

# Тестирование

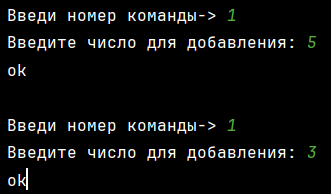
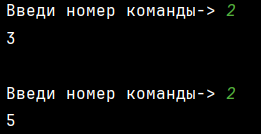


**Тесты для класса Stack:**

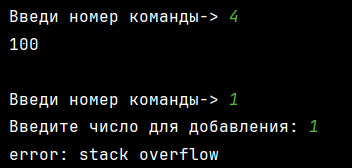
1. **Тест метода push:**
   * Добавление одного элемента и проверка, что он действительно добавлен.



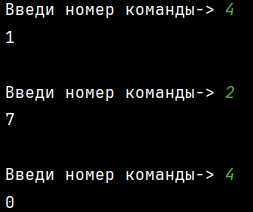
* + Добавление нескольких элементов и проверка их порядка.

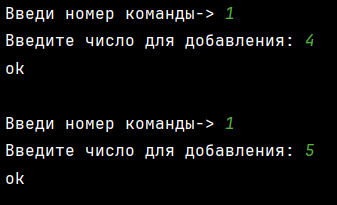
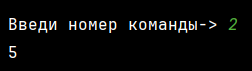
* + Попытка добавления элемента в заполненный стек (должна возвращать ошибку переполнения).



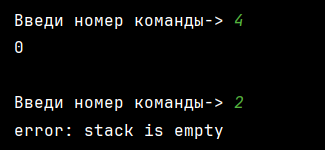
1. **Тест метода pop:**
   * Удаление элемента из стека с одним элементом и проверка, что стек стал пустым.



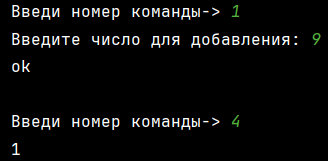
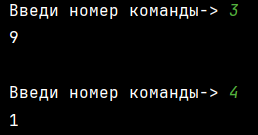
* + Удаление элемента из стека с несколькими элементами и проверка, что удален последний добавленный элемент.

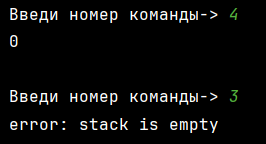
* + Попытка удаления из пустого стека (должна возвращать ошибку).



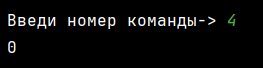
1. **Тест метода back:**
   * Проверка значения верхнего элемента без его удаления.

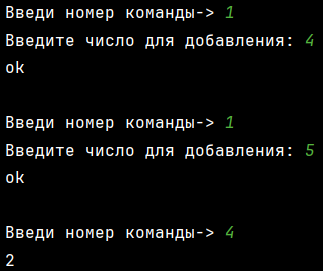
* + Проверка на пустом стеке (должна возвращать ошибку).



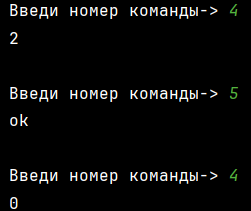
1. **Тест метода size:**
   * Проверка размера пустого стека.



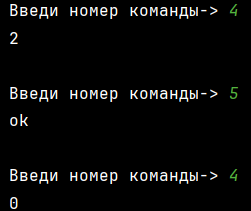
* + Проверка размера после добавления нескольких элементов.



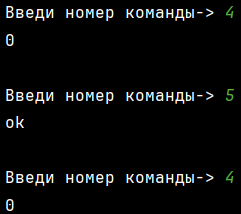
* + Проверка размера после очистки стека.



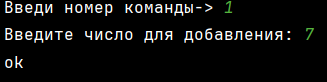
1. **Тест метода clear:**
   * Очистка непустого стека и проверка, что размер стал равен 0.



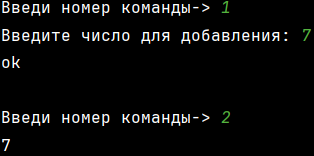
* + Очистка пустого стека.



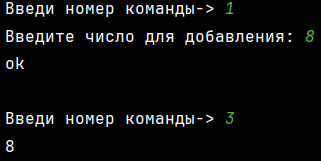
1. **Тесты для основной программы (main):**
2. **Тест обработки команд:**
   * Проверка реакции на команду push 7 (должна выводить "ok").



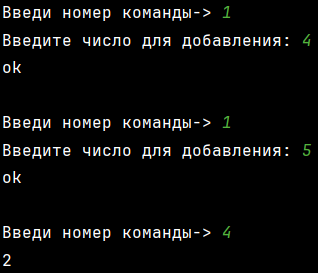
* + Проверка реакции на команду pop (должна выводить удаленный элемент).



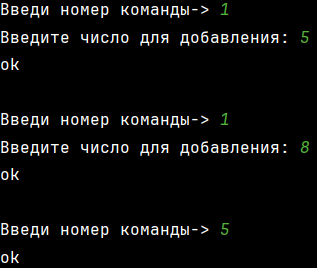
* + Проверка реакции на команду back (должна выводить последний элемент).



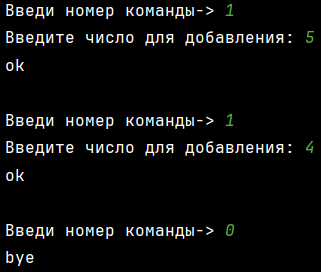
* + Проверка реакции на команду size (должна выводить текущий размер стека).



* + Проверка реакции на команду clear (должна выводить "ok").



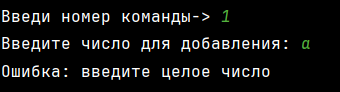
* + Проверка реакции на команду exit (должна выводить "bye" и завершать работу).



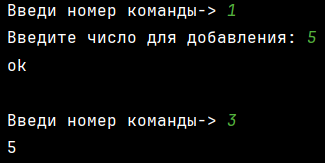
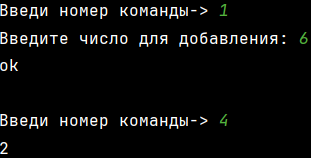
1. **Тест обработки некорректного ввода:**
   * Проверка реакции на ввод несуществующей команды.



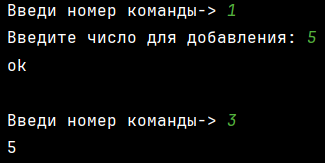
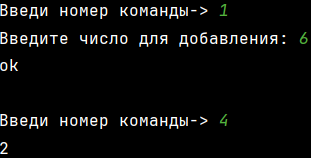
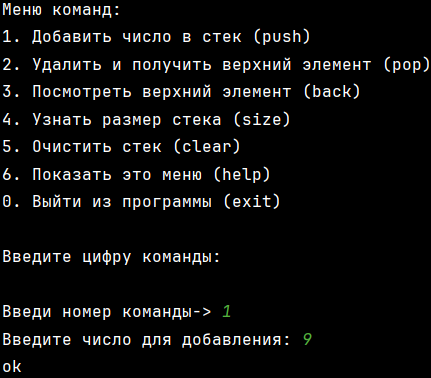
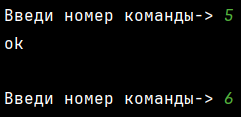
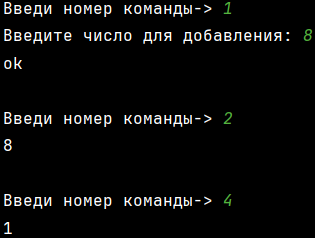
* + Проверка реакции на ввод некорректного значения для push (не число).



1. **Тест последовательности команд:**
   * Последовательное выполнение нескольких команд и проверка корректности вывода после каждой.

* + Проверка работы стека после очистки и повторного добавления элементов.

# Код программы

class Stack:  
 *"""Класс, реализующий структуру данных стек."""* def \_\_init\_\_(self):  
 *"""Инициализирует стек с максимальным размером 100 элементов."""* self.\_\_items = []  
 self.\_\_max\_size = **100** def push(self**,** value):  
 *"""  
 Добавляет элемент в стек.  
  
 Args:  
 value: Число для добавления  
  
 Returns:  
 "ok" при успехе, сообщение об ошибке при переполнении  
 """* if len(self.\_\_items) >= self.\_\_max\_size:  
 return "error: stack overflow"  
 self.\_\_items.append(value)  
 return "ok"  
  
 def pop(self):  
 *"""  
 Удаляет и возвращает верхний элемент стека.  
  
 Returns:  
 Значение элемента или сообщение об ошибке  
 """* if not self.\_\_items:  
 return "error: stack is empty"  
 return str(self.\_\_items.pop())  
  
 def back(self):  
 *"""  
 Возвращает верхний элемент без удаления.  
  
 Returns:  
 Значение элемента или сообщение об ошибке  
 """* if not self.\_\_items:  
 return "error: stack is empty"  
 return str(self.\_\_items[-**1**])  
  
 def size(self):  
 *"""  
 Возвращает количество элементов в стеке.  
  
 Returns:  
 Количество элементов в виде строки  
 """* return str(len(self.\_\_items))  
  
 def clear(self):  
 *"""Очищает стек и возвращает подтверждение."""* self.\_\_items = []  
 return "ok"  
  
  
def display\_menu():  
 *"""Выводит меню команд с цифрами."""* print("\nМеню команд:")  
 print("1. Добавить число в стек (push)")  
 print("2. Удалить и получить верхний элемент (pop)")  
 print("3. Посмотреть верхний элемент (back)")  
 print("4. Узнать размер стека (size)")  
 print("5. Очистить стек (clear)")  
 print("6. Показать это меню (help)")  
 print("0. Выйти из программы (exit)")  
 print("\nВведите цифру команды:")  
  
  
def main():  
 *"""Основная функция программы."""* stack = Stack()  
 print("Добро пожаловать в программу 'Стек'!")  
 display\_menu()  
  
 while True:  
 try:  
 user\_input = input("\n> ").strip()  
  
 if not user\_input:  
 continue  
  
 if user\_input == "1": # push  
 try:  
 value = int(input("Введите число для добавления: "))  
 print(stack.push(value))  
 except ValueError:  
 print("Ошибка: введите целое число")  
  
 elif user\_input == "2": # pop  
 print(stack.pop())  
  
 elif user\_input == "3": # back  
 print(stack.back())  
  
 elif user\_input == "4": # size  
 print(stack.size())  
  
 elif user\_input == "5": # clear  
 print(stack.clear())  
  
 elif user\_input == "6": # help  
 display\_menu()  
  
 elif user\_input == "0": # exit  
 print("bye")  
 break  
  
 else:  
 print("Неизвестная команда. Введите цифру от 0 до 6")  
  
 except KeyboardInterrupt:  
 print("\nПрограмма завершена")  
 break  
 except Exception:  
 print("Произошла ошибка. Попробуйте снова")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

# Инструкция по применению стилей и оформлению работы

Для оформления частей отчёта следует использовать заранее созданные стили. Все стили, которые могут пригодиться, начинаются с «ЛР. 1 …».



**ЛР. 1 Обычный** – для оформления текста задания и алгоритма решения.

**ЛР. 1 Текст программы** – для оформления кода программы.

**ЛР. 1 Заголовок 1** – заголовок первого уровня (для того, чтобы озаглавить основные разделы отчета).

**ЛР. 1 Заголовок 2** – заголовок второго уровня (для того, чтобы озаглавить подразделы).

Для того, чтобы перенести текст следующего блока на другую страницу, необходимо воспользоваться инструментов «Разрыв страницы» в разделе «Вставка».

