Урок 13.  
Линейные структуры данных.

короткая линия

# План урока

1. Стек
2. Очередь
3. Дек
4. Встроенная реализация

# Стек

Этот урок посвящен структурам данных, которые мы будем реализовывать самостоятельно, так как для них в Python не предусмотрены стандартные типы. Стек, очередь и дек, так же как и список, относятся к *линейным* структурам данных, то есть их элементы упорядочены в зависимости от того, как они добавлялись или удалялись.

Стек (Stack) – это упорядоченная коллекция элементов, где добавление нового или удаление существующего элемента всегда происходит только на одном из концов. Проще представить стек как стопку коробок. Добавляя элемент (коробку), мы ставим его на вершину стопки. Разгружая коробки, мы поочередно снимаем их с вершины, пока не добираемся до коробки в основании стопки.

Эта простая иллюстрация объясняет устройство стека. Создавая стек, мы добавляем элемент на вершину. В каждый конкретный момент времени мы можем посмотреть на элемент на вершине, удалить его или положить сверху еще один элемент. Таким образом, стек реализует принцип организации элементов, который называется LIFO (“Last In, First Out” – “последним пришёл – первым ушёл”).

Еще несколько примеров:

* стопка книг на столе: Вы можете увидеть обложку только самой верхней книги. Удаляя книгу с вершины стека, можно увидеть обложку книги, которая раньше лежала ниже.
* браузеры сохраняют стек посещенных URL, чтобы пользователь мог воспользоваться кнопкой “Назад”. На вершине стека расположен текущий URL, ниже – страницы, посещенные до этого.

Итак, определим действия, которые нужны для реализации стека:

* добавление элемента на вершину (push)
* удаление элемента с вершины (pop)
* просмотр элемента на вершине без удаления (top)
* получение информации о размере стека (size)
* очистка стека (clear)

Реализовывать стек мы будем на основе списка, определив для каждого действия отдельную функцию:

|  |
| --- |
| **def** push(value):  stack.append(value)  **def** pop():  **return** stack.pop()  **def** top():  **return** stack[-1]  **def** size():  **return** len(stack)  **def** clear():  stack.clear()  stack = []  *# моделирование книжной полки* push(**"Война и мир"**) push(**"Гордость и предубеждение"**) push(**"Над пропастью во ржи"**) print(top()) *# "Над пропастью во ржи"* print(size()) *# 3* pop() print(top()) *# "Гордость и предубеждение"* print(size()) *# 2* clear() print(size()) *# 0* |

Возможно, Вы уже реализовывали стек в задаче “Ответ от сервера” из урока 1.9 (<https://stepik.org/lesson/201702/step/15?unit=175778>). Решим эту задачу, используя нашу модель стека:

|  |
| --- |
| **def** push(value):  stack.append(value)  **def** pop():  **return** stack.pop()  **def** top():  **return** stack[-1]  **def** size():  **return** len(stack)  **def** clear():  stack.clear()  stack = []  command = input() **while** command != **"."**:  **if** command.startswith(**"POST"**):  push(command[5:])  **elif** command.startswith(**"GET"**):  print(top())  **elif** command.startswith(**"DELETE"**):  pop()  command = input() print(**" "**.join(stack)) |

**Задача “Работаем эффективно”**

**Задача “TO-DO List”**

**Задача “Правильная скобочная последовательность”**

**Задача “Обратная польская нотация”**

# Очередь

В отличие от стека очередь (queue) реализует принцип организации данных под названием FIFO (“First In First Out” – “первым пришел – первым ушел”), моделируя таким образом известную нам из жизни очередь. Добавляются элементы в “хвост” очереди, а извлекаются с “головы” очереди. Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Действия, которые необходимо реализовать для очереди:

* добавление элемента в “хвост” очереди (push)
* удаление элемента из “головы” очереди (pop)
* просмотр элемента в “голове” очереди без его удаления (top)
* получение информации о размере очереди (size)
* очистка очереди (clear)

Реализовывать очередь мы будем снова на основе списка:

|  |
| --- |
| **def** push(value):  queue.append(value)  **def** pop():  **return** queue.pop(0)  **def** top():  **return** queue[0]  **def** size():  **return** len(queue)  **def** clear():  queue.clear()  queue = [] *# моделируем очередь в ресторане* push(**"Стейк"**) push(**"Салат"**) push(**"Десерт"**) print(top()) *# Стейк* print(size()) *# 3* pop() print(top()) *# Салат* print(size()) *# 2* clear() print(size()) *# 0* |

Из этой реализации видно, что “хвост” и “голова” очереди – это соответственно конец и начало списка queue.

**Задача “Очередь в ресторане”**

# Дек

Дек (deque – “**D**ouble-**E**nded **Que**ue”) – это некий гибрид очереди и стека, которые часто иначе называют двусвязной (или двухсторонней) очередью. Дек также имеет “голову” и “хвост”, однако добавлять и извлекать элементы можно с любой стороны. Таким образом, количество действий, которые нужно реализовать, заметно возрастает:

* добавление элемента к “хвост” (push\_back)
* добавление элемента в “голову” (push\_front)
* удаление элемента из “хвоста” (pop\_back)
* удаление элемента из “головы” (pop\_front)
* просмотр элемента в “хвосте” без его удаления (back)
* просмотр элемента в “голове” без его удаления (top)
* получение информации о размере дека (size)
* очистка дека (clear)

Приведем реализацию дека:

|  |
| --- |
| **def** push\_back(value):  dequeue.append(value)  **def** push\_front(value):  dequeue.insert(0, value)  **def** pop\_back():  **return** dequeue.pop()  **def** pop\_front():  **return** dequeue.pop(0)  **def** top():  **return** dequeue[0]  **def** back():  **return** dequeue[-1]  **def** size():  **return** len(dequeue)  **def** clear():  dequeue.clear()  dequeue = [] |

Чтобы разобраться с применением дека, решим задачу определения палиндрома (<https://stepik.org/lesson/219715/step/11?unit=211832>). Решая задачу, до написания кода полезно разложить ее на подзадачи и составить алгоритм. Обозначим последовательность действий:

* удалим пробелы, переведем строку в нижний регистр, а затем – в список символов;
* поочередно добавим каждый символ в “хвост” дека;
* после добавления всех символов в “голове” дека – первый символ строки, а в “хвосте” – последний;
* пока дек не пустой или не содержит единственный элемент, удаляем элементы сразу и из “хвоста”, и из “головы” и сравниваем их. Если символы одинаковые, повторяем этот шаг. Если различные, останавливаем программу;
* если после выполнения цикла дек пуст или содержит единственный элемент, то строка является палиндромом, иначе – нет.

Теперь реализуем этот алгоритм на Python.

|  |
| --- |
| **def** push\_back(value):  dequeue.append(value)  **def** push\_front(value):  dequeue.insert(0, value)  **def** pop\_back():  **return** dequeue.pop()  **def** pop\_front():  **return** dequeue.pop(0)  **def** top():  **return** dequeue[0]  **def** back():  **return** dequeue[-1]  **def** size():  **return** len(dequeue)  **def** clear():  dequeue.clear()  dequeue = []  s = list(input().lower().replace(**" "**, **""**)) **for** char **in** s:  push\_back(char)  palindrome = **True** **while** size() > 1 **and** palindrome:  first, last = pop\_front(), pop\_back()  **if** first != last:  palindrome = **False**  print(**"YES"** **if** palindrome **else** **"NO"**) |

Возможно, этот пример покажется намного более громоздким решением этой несложной задачи, однако дек может выручить при решении гораздо более сложных задач.

# Встроенная реализация

В реальной разработке Вам редко придется самостоятельно реализовывать структуры данных. Теперь, когда Вы понимаете их устройство и назначение, ради экономии времени можно воспользоваться реализацией, встроенной в язык Python.

Так как дек предоставляет обширный список функций, он может реализовывать как стек, так и очередь. Поэтому Python позволяет **импортировать** именно эту структуру данных и использовать ее, как Вам угодно.

Мы остановимся подробней на импортировании модулей и библиотеках позже, однако Вы, возможно, уже использовали некоторые модули: например, *math*. Модули расширяют стандартные средства Python и добавляют новые полезные функции (так, *math* добавляет тригонометрические функции, функцию извлечения корня и т. д.)

Дек находится в модуле collections, поэтому, чтобы начать его использовать, необходимо написать

|  |
| --- |
| **from** collections **import** deque  *# буквально читаем: "из модуля collections импортировать deque"*  *# теперь можно использовать дек, используя новое ключевое слово deque*  items = deque([**"Стейк"**, **"Салат"**, **"Десерт"**]) *# в круглых скобках можно передать начальные значения* |

Методы этой встроенной реализации идентичны функциям, которые мы написали; они имеют лишь немного другие названия. Полный список методов, а также примеры их использования можно найти в документации:

<https://docs.python.org/3.7/library/collections.html#deque-objects>