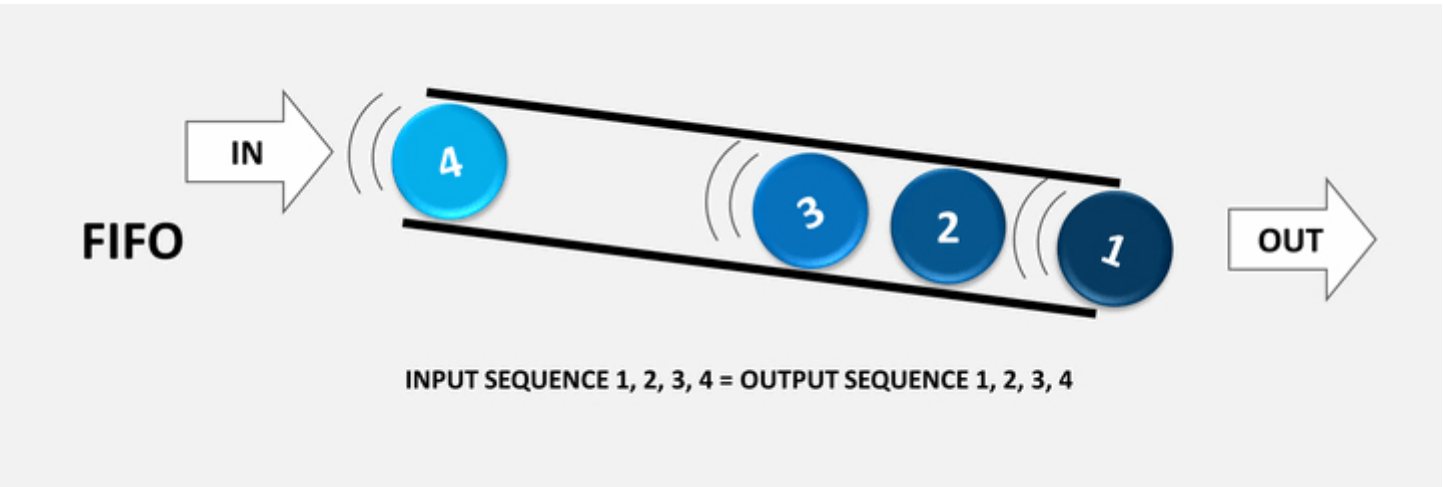
Очереди в Python

Как запустить FIFO структуру данных в Python, пользуясь только встроенными типами данных и классами из **стандартной библиотеки**. Очередь – это набор объектов, который поддерживает быструю семантику first-in, first-out (FIFO) для вставки и удаления. Операции вставки и удаления иногда называют **enqueue** и **dequeuer**. В отличие от списков и массивов, очереди, как правило, не пропускают случайный доступ к содержащимся объектам.



Аналогия очереди first-in, first-out на пальцах

> Представьте ряд питонистов, в ожидании получения бейджиков на конференции в день регистрации на **PyCon**. Новые дополнения в ряде находятся в конце очереди, так как новые люди становятся в очередь на конференцию в ожидании бейджиков. Сокращение очереди (подача) происходит в её начале, так как разработчики получают свои бейджи с новыми кепками с надписью SWAG и покидают очередь.

Другой способ описать такую структуру данных, как очередь, это представить её как трубку:

> Новые объекты (молекулы воды, шарики для тенниса, и т.д.) помещаются в один конец трубки, и двигаются к другому, когда вы, или что-то другое пропихиваете их вдоль трубки. Пока объекты находятся в очереди (в нашем случае это трубка), вы не можете достать их. Единственный способ взаимодействия с объектами в очереди, это добавить новые объекты в конце очереди (**enqueuer**) или убрать существующие из начала (**dequeuer**) трубки.

Очереди похожи на стеки, но отличаются способом **удаления элементов**:

В случае с очередью вы удаляете объект, который был недавно добавлен (по принципу первый вошел – первым вышел first-in, first-out или **FIFO**), в случае сто стеком вы удаляете последний добавленный элемент (последним зашел – первым вышел, last-in, first-out or **LIFO**).

С точки зрения производительности, верная реализация очереди заключается в выделении O(1) времени для вставки и удаления операций. Существует две основные операции, используемые в очереди и они должны выполнять работу быстро и корректно. Очереди Python имеют широкий ряд приложений в алгоритмах, а также все необходимое для составления графиков, а также решения параллельных проблем в программировании. Короткий и удобный алгоритм под названием **BFS** (breadth-first search) годится для работы с такими структурами данных как **tree** и **graph**.

Алгоритмы планирования часто используют **приоритетные очереди** изнутри. Существуют определенные виды очередей: вместо получения следующего элемента в соответствии с указанным временем вставки, приоритетная очередь извлекает элемент с самым высоким приоритетом. Приоритет отдельных элементов определяется очередью на основании порядка применяемых к ним ключам. Однако, обычная очередь, не будет переопределять порядок объектов, которые в ней находятся. Вы получаете только то, и в том порядке, в каком вы помещали объекты в очередь (пример с трубкой). Python позволяет выполнить несколько реализаций очереди, каждая из которых имеет небольшие отличия. Давайте рассмотрим их:

Встроенный список

Мы можем использовать обычный список в качестве очереди, но это не очень эффективно с точки зрения производительности. Списки слишком медленные для этой задачи, так как вставка и удаление элемента с начала требует сдвига всех прочих элементов по одному, на это уходит *O(n)* времени. В связи с этим, я не спешу рекомендовать список в качестве временной очереди в Python (разве что вы имеете дело с небольшим количеством элементов).

Python

```python
# Как использовать список Python в качестве очереди FIFO:

q = []

q.append('eat')
q.append('sleep')
q.append('code')

print(q)
# ['eat', 'sleep', 'code']

# Осторожнее: медленно работает!
print(q.pop(0)) # 'eat'
```

Есть вопросы по Python?

На нашем форуме вы можете задать любой вопрос и получить ответ от всего нашего сообщества!

Python Форум Помощи

Telegram Чат & Канал

Вступите в наш дружный **чат по Python** и начните общение с единомышленниками! Станьте частью большого сообщества!

Чат

Канал

Класс collections.deque

Класс **deque** реализует двухконечную очередь, которая поддерживает добавление и удаление элементов с обоих концов в течение O(1) времени. Объекты **deque** представлены в виде двусвязных списков, что дает им превосходную производительность для входящих и выходящих элементов, но при этом у него **плохая производительность** O(n) при работе со случайно принимаемыми элементами в середине очереди. В связи с тем, что **deque** поддерживает вставку и удаление элементов одинаково хорошо, они могут поддерживать и очереди и стеки. collections.deque это отличное решение, если вы ищите структуру данных очереди в Python в **стандартной библиотеке**.

Python

```python
# Как использовать collections.deque в качестве очереди FIFO:

from collections import deque
q = deque()

q.append('eat')
q.append('sleep')
q.append('code')

print(q)
# deque(['eat', 'sleep', 'code'])

print(q.popleft()) # 'eat'
print(q.popleft()) # 'sleep'
print(q.popleft()) # 'code'

print(q.popleft())
IndexError: "pop from an empty deque"
```

Класс queue.Queue

Такая реализация очереди в стандартной библиотеке является **синхронизированной** и предоставляет блокировку семантики для поддержки нескольких конкурирующих производителей и потребителей. Модуль queue содержит несколько других классов, реализующих мульти-продюсера, очереди мульти-

потребители , которые полезны в параллельных вычислениях. В зависимости от причины использования, закрытие семантики может быть полезным, или просто привести к ненужному перерасходу. В данном случае, вам лучше будет использовать **collections.deque** как очередь общего назначения.

```python
# Как использовать queue.Queue в качестве очереди FIFO:

from queue import Queue
q = Queue()

q.put('eat')
q.put('sleep')
q.put('code')

print(q)
# <queue.Queue object at 0x1070f5b38>

print(q.get()) # 'eat'
print(q.get()) # 'sleep'
print(q.get()) # 'code'

print(q.get_nowait()) # queue.Empty

q.get()
# Блокирование / вечное ожидание...
```

Класс multiprocessing.Queue

Это реализация очереди с разделенными функциями, которые позволяет находящимся в очереди объектам быть обработанными параллельно несколькими одновременно работающими процессами. Основанная на процессах паралеллизация очень популярна в Python, из-за глобального блокиратора интерпретатора GIL. **multiprocessing.Queue** используется при разделении данных между процессами и может хранить любой пригодный объект.

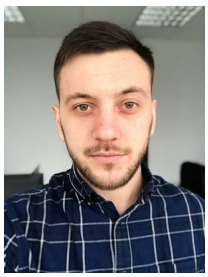
Как использовать multiprocessing.Queue в качестве очереди FIFO:

```python
# Как использовать multiprocessing.Queue в качестве очереди FIFO:

from multiprocessing import Queue
q = Queue()

q.put('eat')
q.put('sleep')
q.put('code')
```

```
9
10 print(q)
11 # <multiprocessing.queues.Queue object at 0x1081c12b0>
12
13 print(q.get()) # 'eat'
14 print(q.get()) # 'sleep'
15 print(q.get()) # 'code'
16
17 q.get()
18 # Блокировка / вечное ожидание...
```

Вердикт

Неплохой выбор – это **collections.deque**. Если вас не интересует поддержка параллельной обработки, реализация очереди, выполняемая **collections.deque** – отличный дефолтный выбор для создания структуры данных очереди **FIFO** в Python. У него хорошие характеристики, которые нужны при реализации очереди, также он может быть использован в качестве стека.



Vasile Buldumac

Являюсь администратором нескольких порталов по обучению языков программирования Python, Golang и Kotlin. В составе небольшой команды единомышленников, мы занимаемся популяризацией языков программирования на русскоязычную аудиторию. Большая часть статей была адаптирована нами на русский язык и распространяется бесплатно.

E-mail: vasile.buldumac@ati.utm.md

Образование
Universitatea Tehnică a Moldovei (*utm.md*)

2014 — 2018 Технический Университет Молдовы, ИТ-Инженер. Тема дипломной работы «*Автоматизация покупки и продажи криптовалюты используя технический анализ*»

2018 — 2020 Технический Университет Молдовы, Магистр, Магистерская диссертация «*Идентификация человека в киберпространстве по фотографии лица*»