СТРУКТУРИ ДАНИХ ТА ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Питання 8.5

Сортуючі списки (Sorting lists)

- Якщо потрібно впорядкувати об’єкти, зазвичай вони поміщаються в список та сортуються.
  - Спеціальний метод __lt__() – "less than" – повинен визначатись у класі, щоб зробити об’єкти порівнюваними.
  - Метод sort() для списку отримуватиме доступ до цього методу для кожного об’єкта списку, щоб перевірити його місце у відсортованому варіанті.

```
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)
```

- Метод __repr__() спрощує перегляд двох значень при друкуванні списку.
- Реалізація методу __lt__() порівнює об’єкт з іншим екземпляром цього класу (або іншим качинотипізованим об’єктом, який має атрибути string, number та sort_num).

Результати роботи

```
>>> a = WeirdSortee('a', 4, True)
>>> b = WeirdSortee('b', 3, True)
>>> c = WeirdSortee('c', 2, True)
>>> d = WeirdSortee('d', 1, True)
>>> l = [a,b,c,d]
>>> l
[a:4, b:3, c:2, d:1]
>>> l.sort()
>>> l
[d:1, c:2, b:3, a:4]
>>> for i in l:
...     i.sort_num = False
...
>>> l.sort()
>>> l
[a:4, b:3, c:2, d:1]
```

- При першому виклику sort() сортування відбувається за числами, оскільки sort_num() повертає True для всіх об'єктів, які порівнюються.
- Вдруге відбувається сортування за літерами.
  - Метод __lt__() – єдиний необхідний для задіювання сортування.
  - Проте, технічно, при його реалізації мають бути також реалізації подібних методів __gt__(), __eq__(), __ne__(), __ge__() та __le__(), які відповідають операторам <, >, ==, !=, >= та <=.

Це можна отримати задарма, реалізувавши методи __lt__() та __eq__(), а потім застосувавши декоратор класу @total_ordering

```
from functools import total_ordering

@total_ordering
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return"{}:{}".format(self.string, self.number)

    def __eq__(self, object):
        return all((
            self.string == object.string,
            self.number == object.number,
            self.sort_num == object.number
        ))
```

- Це корисно тоді, коли потрібно мати змогу використовувати оператори над нашими об'єктами.
  - Проте для налаштування порядку сортування навіть цього недостатньо.
  - Тоді метод sort() повинен могти приймати опційний аргумент key – функцію, здатну перетворювати кожен об'єкт зі списку в порівнюваний об'єкт.
  - Наприклад, str.lower()

```
>>> l = ["hello", "HELP", "Helo"]
>>> l.sort()
>>> l
['HELP', 'Helo', 'hello']
>>> l.sort(key=str.lower)
>>> l
['hello', 'Helo', 'HELP']
```

Існує кілька ключів сортування, настільки поширених, що команда Python вбудувала їх

- Наприклад, часто потрібно відсортувати список кортежів за чимось відмінним, від першого елементу в списку.
  - У якості ключа використовується метод operator.itemgetter():

```
>>> from operator import itemgetter
>>> l = [('h', 4), ('n', 6), ('o', 5), ('p', 1), ('t', 3), ('y', 2)]
>>> l.sort(key=itemgetter(1))
>>> l
[('p', 1), ('y', 2), ('t', 3), ('h', 4), ('o', 5), ('n', 6)]
```

  - Функція itemgetter() найбільш поширена (працює і для словників), проте інколи можна використати attrgetter() та methodcaller(), які повертають атрибути об'єкта та результати викликів методу над об'єктом.
  - Додаткова інформація в документації модуля operator.

Порожні об’єкти

- Технічно, можна інстанціювати об’єкт без записування підкласу:

```
>>> o = object()
>>> o.x = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'x'
```

  - щоб зберегти пам’ять.
    - Коли Python дозволяє об’єкту мати довільні атрибути, об’єкт отримує частину системної пам’яті, щоб слідкувати за володінням атрибутами, зберіганням їх назв та значень.
    - Навіть без атрибутів виділяється пам’ять для потенційних нових атрибутів.

```
class MyObject:
    pass
```

```
>>> m = MyObject()
>>> m.x = "hello"
>>> m.x
'hello'
```

- Основна причина писати порожній клас – застовпити місце, щоб потім додати поведінку.
  - Набагато простіше адаптувати поведінки у класі, ніж заміняти структуру даних в об’єкті та всі посилання на неї.

Розширення вбудованих структур даних

- Потрібно застосувати наслідування, якщо необхідно змінити принципи роботи контейнера.
  - Наприклад, бажаємо перевірити, щоб кожен елемент списку був 5-символьним рядком.
  - Слід розширити list та переозначити метод append(), щоб викидати виняток при некоректному вводі.
  - Також потрібно мінімально переозначити спеціальний метод __setitem__(self, index, value), який викликається при застосуванні синтаксису x[index] = "value", а також метод extend().
- Списки є об’єктами.
  - Python-програмісти погоджуються, що не ОО синтаксис простіше читати та писати.

```
c = a + b
c = a.add(b)
```

```
l[0] = 5
l.setitem(0, 5)
d[key] = value
d.setitem(key, value)
```

```
for x in alist:
    #do something with x
it = alist.iterator()
while it.has_next():
    x = it.next()
    #do something with x
```

Різні форми роботи зі списками

```
class SillyInt(int):
    def __add__(self, num):
        return 0
```

- Об’єктно-орієнтована форма запису довша.
  - Проте решта форм запису перетворюються компілятором в ООП-форму в вигляді ОО методів.
  - Ці методи мають спеціальні назви: __назва__.
  - Також це дає причину переозначувати (override) їх.

```
>>> a = SillyInt(1)
>>> b = SillyInt(2)
>>> a + b
0
```

- Якщо бажано використовувати синтаксис «x in myobj» для кастомізованого об’єкта, можна реалізувати __contains__().
  - Для синтаксису «myobj[i] = value» постачається метод __setitem__().
  - Для синтаксису «something = myobj[i]» реалізують метод __getitem__().
- Для списків представлено 33 спеціальних методи:

```
>>> dir(list)

['__add__', '__class__', '__contains__', '__delattr__','__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__
new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__
subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort'
```

Довідка зі спеціальних методів

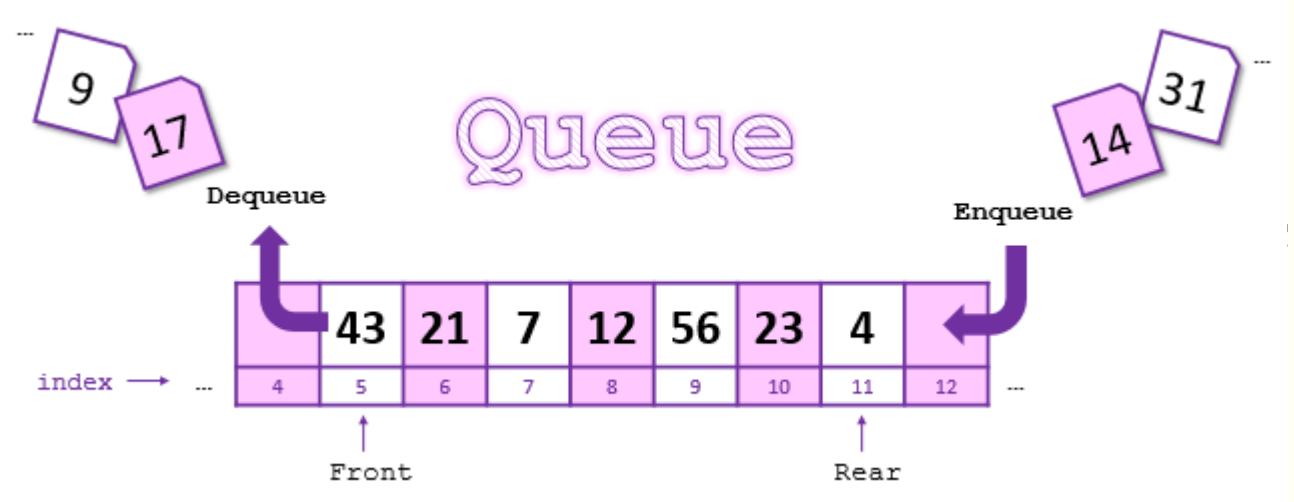
- Для додаткової інформації щодо роботи методів можна використати функцію help():

```
>>> help(list.__add__)
Help on wrapper_descriptor:

__add__(self, value, /)
    Return self+value.
```

- Часто потреба в розширенні вбудованих структур даних вказує на помилковий вибір типу даних.
  - Це не завжди так, проте слід ретельно розглянути доречні альтернативні типи даних.

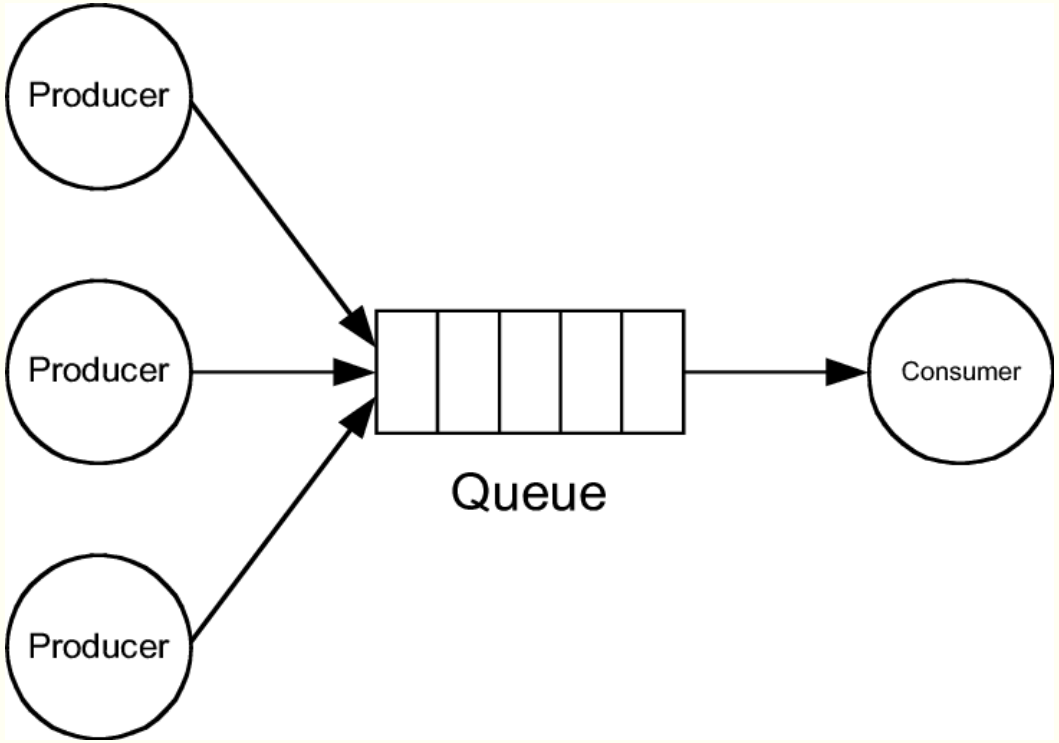
Черги (Queue)



- Python пропонує три види структур даних на базі черг.
  - Всі вони використовують один API, проте відрізняються поведінкою та структурними особливостями.
- Списки Python мають багато сценаріїв для використання:
  - Підтримують ефективний довільний доступ до будь-якого елементу списку;
  - Встановлюють строгий порядок елементів;
  - Підтримують ефективну операцію додавання (append) елементів.
- Загалом списки досить повільні в ситуаціях, коли потрібно додавати елементи в їх кінець.
  - Черга представляє структуру даних з властивістю FIFO (First In First Out) – перший прийшов, перший пішов.

Клас Queue

- Реалізує чергу.
  - Зазвичай використовується як середовище комунікації при виробництві-споживанні (producer-consumer) об’єктів, ймовірно, з різною швидкістю.
  - Уявіть месенджер, в якому надходить багато повідомлень, проте може відобразити лише одне повідомлення за раз.
  - Інші повідомлення буферизуються в чергу в порядку надходження.
  - FIFO-черги часто використовуються в багатопоточних додатках.
  - Клас Queue доречний, коли немає потреби отримувати доступ до довільного елемента структури даних, крім наступного.
  - Список буде менш ефективним, оскільки вставка та видалення даних з початку списку вимагає зсуву решти елементів списку.



Черги мають досить простий API

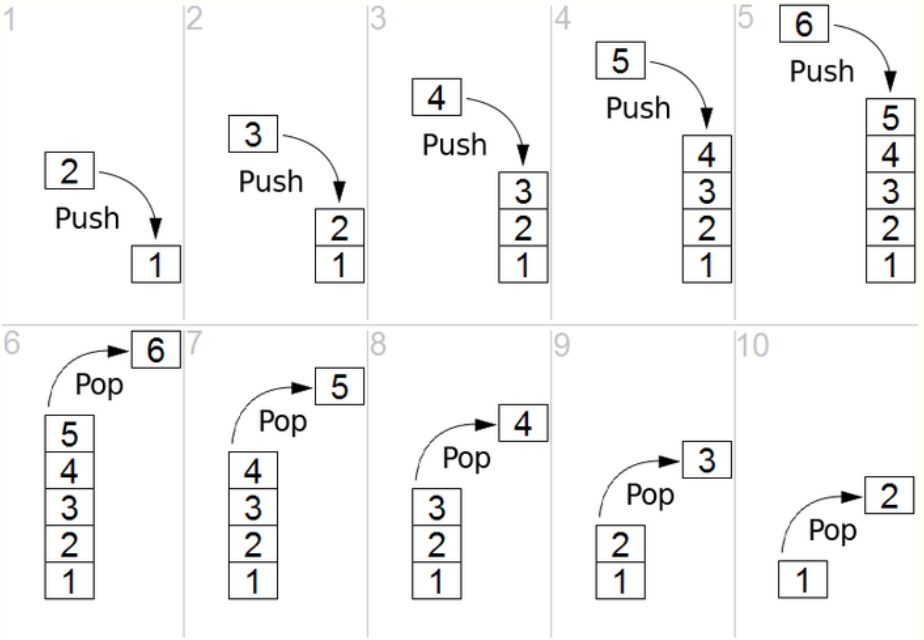
- Місткість черги на базі класу Queue може бути «нескінченною» (поки є пам’ять), проте частіше вона обмежується певним максимальним розміром.
  - Основні методи – put() та get() – додають елемент у кінець лінії та отримують елементи з початку відповідно.
  - Обидва методи приймають опційні аргументи, які визначають реакцію на неуспішне виконання операції через порожність або переповнення черги.
  - Поведінка за умовчанням – виконати блокування або постійно очікувати, поки в Queue-об’єкті не з’являться дані або місце.
  - Викинути виняток можна за умови передачі параметру block=False.
  - Для обмежування часу очікування перед викиданням винятку слід передавати параметр timeout.
  - Клас також має методи для перевірки пустоти (empty()) та переповнення (full()) черги та кілька додаткових методів для багатопоточного доступу.

Методи в дії

```
>>> from queue import Queue
>>> lineup = Queue(maxsize=3)
>>> lineup.get(block=False)
Traceback (most recent call last):
  File "<ipython-input-5-a1c8d8492c59>", line 1, in <module>
    lineup.get(block=False)
  File "/usr/lib64/python3.3/queue.py", line 164, in get
    raise Empty
queue.Empty
>>> lineup.put("one")
>>> lineup.put("two")
>>> lineup.put("three")
>>> lineup.put("four", timeout=1)
Traceback (most recent call last):
  File "<ipython-input-9-4b9db399883d>", line 1, in <module>
    lineup.put("four", timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 144, in put
    raise Full
queue.Full
>>> lineup.full()
True
>>> lineup.get()
'one'
>>> lineup.get()
'two'
>>> lineup.get()
'three'
>>> lineup.empty()
True
```

LIFO-черги

- **LIFO** (**Last In First Out**) черги частіше називають ***стеками (stack)***.
  - Уявляйте стек, як стопку паперу.
  - Взяти та покласти папір можна тільки нагору.
- Традиційні назви операцій зі стеком – push (додавання) та pop (видалення), проте модуль queue використовує одноіменні методи put() і get().
  - Вони працюють з вершиною стеку замість початку та кінця черги.
  - Це чудовий приклад поліморфізму.



```
>>> from queue import LifoQueue
>>> stack = LifoQueue(maxsize=3)
>>> stack.put("one")
>>> stack.put("two")
>>> stack.put("three")
>>> stack.put("four", block=False)
Traceback (most recent call last):
  File "<ipython-input-21-5473b359e5a8>", line 1, in <module>
    stack.put("four", block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
queue.Full

>>> stack.get()
'three'
>>> stack.get()
'two'
>>> stack.get()
'one'
>>> stack.empty()
True
>>> stack.get(timeout=1)
Traceback (most recent call last):
  File "<ipython-input-26-28e084a84a10>", line 1, in <module>
    stack.get(timeout=1)
  File "/usr/lib64/python3.3/queue.py", line 175, in get
    raise Empty
queue.Empty
```

Робота зі стеком у Python

- Кілька причин віддати перевагу LifoQueue, а не списку:
  - LifoQueue підтримує чистий багатопоточний доступ з багатьох потоків.
  - LifoQueue задіює інтерфейс стеку: неможливо випадково вставити елемент не туди.

Черги з пріоритетами (Priority queues)

- Мають дуже відмінний принцип впорядкування в порівнянні з іншими чергами.
  - Вони мають ті ж методи get() і put(), проте заміняють критерій отримання елементів з порядку їх надходження на їх «важливість».
  - За угодою, найбільш важливим (пріоритетним) елементом є той, що має найменший пріоритет при порівнянні оператором less than.
  - Зазвичай зсередини в черзі з пріоритетами зберігаються кортежі – пари «пріоритет-значення».
  - Інший поширений варіант – реалізувати метод __lt__().
  - Абсолютно нормально мати кілька елементів з однаковим пріоритетом, проте немає гарантії, який з них буде віддаватись першим.
- Чергу з пріоритетом можна застосовувати, наприклад, для пошукового двигуна, який ранжуватиме найбільш популярні веб-сторінки до того, як почати пошук на сайтах.
  - Інструмент рекомендації товарів може використати чергу з пріоритетами для показу інформації про найбільш популярні товари, поки підвантажуватиме інформацію щодо менш популярних.

Черги з пріоритетами (Priority queues)

```
>>> heap.put((3, "three"))
>>> heap.put((4, "four"))
>>> heap.put((1, "one") )
>>> heap.put((2, "two"))
>>> heap.put((5, "five"), block=False)
Traceback (most recent call last):
  File "<ipython-input-23-d4209db364ed>", line 1, in <module>
    heap.put((5, "five"), block=False)
  File "/usr/lib64/python3.3/queue.py", line 133, in put
    raise Full
Full
>>> while not heap.empty():
    print(heap.get())
(1, 'one')
(2, 'two')
(3, 'three')
(4, 'four')
```

Черга з пріоритетами завжди повертатиме поточний найважливіший елемент.

- Метод get() блокуватиме (за умовчанням), якщо черга порожня, проте якщо в черзі щось є, він не блокуватиме потік та очікуватиме вставку елементу з вищим пріоритетом.
- Черга не знає нічого про елементи, які ще не були додані (або навіть про елементи, які до цього були видалені). Вона оперує пріоритетами лише наявних у черзі елементів.

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Основи модульного тестування коду мовою Python