Методы контейнеров

Ранее мы рассматривали классы, оперирующие с простыми типами данных: int, float, str. Но есть контейнеры, то есть списки, словари, множества, которые содержат эти простые типы внутри себя. Как быть с ними? Как их создавать и определять их магические методы? Ведь для работы с контейнерами нужны совсем другие операции.

Мы уже говорили, что в Python все есть объект, а объект есть класс. То есть любой тип данных, например, int, — это тоже класс, в котором определены методы для операций с ним.

Получается, любой контейнер — тоже класс, в котором находятся экземпляры других классов, и у всех есть свои магические методы... Однако для контейнеров нужны методы, которых нет у атомарных типов данных. Рассмотрим их.

Для того чтобы ваш класс мог работать как изменяемый или неизменяемый контейнер, у него должны быть определены специальные методы. Для неизменяемого контейнера достаточно определить методы, действие которых понятно из названия: `__len__()` и `__getitem__()`, поскольку с ним ничего больше сделать нельзя, а для изменяемого контейнера — еще и метод для изменения элемента контейнера (`__setitem__()`) и метод для удаления элемента (`__delitem__()`). А чтобы можно было перебирать элементы, например, в цикле for, нужен метод `__iter__()`.

Давайте напишем класс, в котором грибы расположены в некоторой последовательности, например, нанизаны на прутик:

```
class Mushrooms:
    def __init__(self, values):
        self.values = values[:]

    def __len__(self):
        return len(self.values)

    def __getitem__(self, item):
        return self.values[item]

    def __iter__(self):
        return iter(self.values)

    def __setitem__(self, key, value):
        self.values[key] = value


ms = Mushrooms(['cep', 'puff ball', 'stubby-stalk'])
print(len(ms))
for item in ms:
    print(item)
```

```
print()
ms[1] = 'russule'
print(ms[1])
3
cep
puff ball
stubby-stalk

russule
```

Обратите внимание: при инициализации мы копируем переданный список, чтобы избежать изменения внешней переменной.

Методы сравнения

В случае, если экземпляры класса могут сравниваться, обязательно нужно реализовать методы, которые укажут, как именно это нужно делать. Например, как сравнить два карандаша? Больше тот, что длиннее, или тот, что толще? А может, красный больше синего? Это все можно прописать при реализации методов сравнения.

Все 6 методов сравнения расписывать необязательно, потому что они попарно обратны друг другу. Например, если a > b, то (a >= b) == not (a < b).

```
class Pencil:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def __lt__(self, other):
        return (self.length, self.width) < (other.length,
other.width)

    def __ge__(self, other):
        return not self.__lt__(other)


p_1 = Pencil(12, 1.5)
p_2 = Pencil(12, 2)
print(p_1 < p_2)  # True
print(p_1 >= p_2)  # False
```

В нашем примере карандаши сначала сравниваются по длине, а в случае равной длины — по толщине.

Другие специальные методы
Специальных методов слишком много, чтобы рассмотреть их все на этом уроке. Мы приведем лишь небольшой их список.

Специальные методы сравнения

Метод	Описание
`__eq__(self, other)`	Оператор равенства ==. Будет вызвано: x.__eq__(y)
`__ne__(self, other)`	Оператор неравенства !=
`__lt__(self, other)`	Оператор меньше <. Будет вызвано: x.__lt__(y)
`__gt__(self, other)`	Оператор больше >
`__le__(self, other)`	Оператор меньше или равно <=
`__ge__(self, other)`	Оператор больше или равно >=

Специальные методы контейнеров

Метод	Описание
`__len__(self)`	Оператор получения длины последовательности
`__getitem__(self, key)`	Оператор получения элемента по ключу
`__setitem__(self, key, value)`	Оператор присваивания значения элементу
`__delitem__(self, key)`	Оператор удаления элемента по ключу
`__iter__(self)`	Возвращает итератор
`__next__(self)`	Должен вернуть следующий элемент в последовательности