Урок 7

Интерфейсы

Интерфейсы (перегрузка операторов). Интерфейс итерации (iter(), next()). Метод как атрибут. Статические методы. Расширение встроенных типов.

Перегрузка операторов

Перегрузка операторов позволяет объектам, созданным из классов, перехватывать операции и участвовать в тех, что применяются к встроенным типам: сложение, вычитание, получение среза, вывод и так далее. По большей части это автоматический механизм: при выполнении выражений и других встроенных операций интерпретатор передаёт управление реализации классов.

Мы можем полностью реализовать поведение класса в виде методов. Однако перегрузка операторов позволяет объектам действовать так же, как действуют встроенные объекты, потому что она создаёт менее противоречивые и более простые в изучении интерфейсы объектов. Обеспечивается возможность обрабатывать объекты, созданные из классов, программным кодом, который предполагает взаимодействие со встроенными типами.

Имена методов, начинающиеся и заканчивающиеся двумя символами подчеркивания (__X__), имеют специальное назначение. Перегрузка операторов в языке Python реализуется за счёт создания методов со специальными именами для перехвата операций. Язык Python определяет фиксированные и неизменяемые имена методов для каждой из операций. Классы могут переопределять большинство встроенных методов, такие методы вызываются автоматически. Например, если объект экземпляра наследует метод __add__, этот метод будет вызываться всякий раз, когда объект будет появляться в операции сложения (+). Возвращаемое значение метода становится результатом соответствующей операции.

```
class Vector:
    def __init__(self, pos):
        self.x = pos[0]
        self.y = pos[1]

# Перегружаем оператор +
    def __add__(self, other):
        return Vector((self.x + other.x, self.y + other.y))

    def as_point(self):
        return self.x, self.y

# Формируем удобное отображение объекта при выводе функции print()
    def __str__(self):
        return "V(x:{} y:{})".format(self.x, self.y)

# Создаём экземпляры класса (объекты)
v1 = Vector((10, 15))
v2 = Vector((12, 10))
# Наши объекты участвуют в операции сложения (+)
v3 = v1 + v2
# На самом деле это работает так:
# v3 = v1.__add__(v2)
# Благодаря перегрузке мы можем использовать более удобную и привычную запись:
# v3 = v1+v2
```

Обратите внимание, что метод __add__ создаёт и возвращает новый объект экземпляра этого класса.

Аналогичным образом можно перегружать практически все встроенные операции. Полный список magic-методов (именно так они называются в официальной литературе) с описанием можно посмотреть здесь.

Благодаря перегрузке операторов, объекты, реализованные на базе классов, действуют подобно встроенным типам, тем самым обеспечивают непротиворечивые и совместимые интерфейсы.

Конструктор __init__ также является примером перегрузки операции и вызывается автоматически при создании экземпляра класса.

Перегрузка операторов является частью механизма полиморфизма, который мы уже рассматривали.

Интерфейс итерации

С итераторами мы знакомы. Разберём, как они устроены внутри и научимся создавать собственные объекты-итераторы.

Итераторы — это специальные объекты, представляющие последовательный доступ к данным из контейнера.

При этом немаловажную роль играет то, что память фактически не тратится, так как промежуточные данные выдаются по мере необходимости при запросе, поэтому фактически в памяти останутся только исходные данные и конечный результат, да и их можно читать и записывать, используя файл на диске.

С итераторами работают:

1. Цикл for in.
2. Встроенные функции map(), filter(), zip() и прочие.
3. Операция распаковки *.

На самом деле все эти инструменты будут работать с любым объектом, который поддерживает интерфейс итерации. Т.е. предоставляет определённый набор методов, которыми пользуются инструменты, работающие с итераторами.

В python понятие интерфейса несколько отличается от интерфейса в других языках. Если вы можете применить определённый инструмент (функцию, инструкцию, оператор), значит, этот объект поддерживает данный интерфейс. Наличие конкретного интерфейса определяется наличием нужного метода. Да, всё так просто!

Вернёмся к итераторам.

```
test_list = [1, 2, 3, 4, 'END']
# Когда вы пишете
for el in test_list:
    print(test_list)
```

Python выполняет:

1. Вызывает метод __iter__(): test_list.__iter__().
   Метод __iter__() должен вернуть объект, у которого есть метод __next__().
2. Цикл for in каждую итерацию вызывает метод __next__().
   __next__() при каждом вызове возвращает следующий элемент итератора.
3. Когда элементы итератора заканчиваются, метод __next__() **возбуждает** исключение StopIteration.
   Цикл for in завершает свою работу, когда перехватывает это исключение.

Список сам по себе не итератор, у него есть метод, который возвращает объект-итератор.

Важной особенностью любого объекта-итератора является то, что по нему можно пройти только один раз. Т.е. после вызова next() итератор запоминает своё состояние. Если хотите выполнить итерацию по списку ещё раз, нужно снова получить новый объект-итератор.

Собственный итератор

Пример создания объекта, поддерживающего интерфейс итерации:

```
class IterObj:
    """
    Объект итератор
    """
    def __init__(self, start=0):
        self.i = start
    # Объект считается итератором - если у него есть метод __next__

    def __next__(self):
        self.i += 1
        if self.i <= 5:
            return self.i
        else:
            raise StopIteration

class Iter:
    """
    Объект, поддерживающий интерфейс итерации
    """
    def __init__(self, start=0):
        self.start = start - 1
    def __iter__(self):
        # Метод __iter__ должен возвращать объект-итератор
        return IterObj(self.start)

obj = Iter(start=2)
for el in obj:
    print(el)
print("Еще раз ...")
for el in obj:
    print(el)
print('sum(obj) -->', sum(obj))
# Функция map() возвращает объект-итератор
map_iter = map(int, '123')
print('next(map_iter) --> ', next(map_iter))
print('next(map_iter) --> ', next(map_iter))
# Цикл for in продолжает перебор элементов, т.к. map_iter является итератором
for el in map_iter:
    print("el in for in -->", el)

class Iter2:
    def __init__(self, start=0):
        self.i = start
    def __iter__(self):
# Метод __iter__ должен возвращать объект-итератор
        return self
    def __next__(self):
        self.i += 1
        if self.i <= 5:
            return self.i
        else:
            raise StopIteration
print("Demo Iter2")
obj = Iter2(start=2)
for el in obj:
    print(el)
print("Еще раз ...")
for el in obj:
    print(el)
```

Обратите внимание! Экземпляры Iter() — объекты, возвращающие объект итератор. Экземпляры Iter2() — сами по себе итераторы, и пройти по ним можно только один раз.

Свойство @property

@property — это декоратор, но с декораторами мы познакомимся только в следующей части курса по python, пока воспринимайте их как инструкции, которые добавляют дополнительные возможности.

С декоратором @property мы уже встречались на прошлом уроке, когда разбирали инкапсуляцию. И тут есть предостережение, почему этим инструментом не стоит злоупотреблять.

Например, мы использовали @property так:

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    @property
    def full_name(self):
        return " ".join([self.first_name, self.last_name])
person = Person("Andrey", "Popp")
person.full_name # "Andrey Popp"
```

Это реализация attribute getter/setter-шаблона. То есть, обращаясь к full_name как к атрибуту, мы неявно вызываем соответствующий метод, его результат как бы становится значением атрибута. Почему как бы? Потому что он будет заново вычисляться при каждом обращении к full_name.

Операция обращения к атрибуту (а именно так выглядит работа с property) по своей семантике должна быть «дешёвой». Со свойствами можно сделать следующее — представьте себе, что вы, находясь в консоли Python, обращаетесь к некоему атрибуту myobj.some_info, в результате чего отправляется запрос в БД, вычисление свойства происходит в течении n секунд — крайне неприятно. Получается, если обратиться к такому свойству в цикле из 100 итераций, то будет сделано 100 отдельных запросов к БД, при этом программист, читающий код, может этого совершенно не заметить.

Отсюда вердикт: если для вычисления свойства нужно произвести побочные эффекты, то лучше сделать это свойство методом.

Статические методы

На прошлом уроке, когда мы разбирали механизм наследования, использовали класс Учителя.

```
class Teacher(People):
    def __init__(self, name, surname, birth_date, school, teach_classes):
        People.__init__(self, name, surname, birth_date, school)
        self.teach_classes = list(map(self.convert_class, teach_classes))
 # Уникальный метод Учителя
    def convert_class(self, class_room):
        """
             '<class_num> <class_int>' --> {'class_num': <class_num>, 'class_int':
<class_int>}
        """
        return {'class_num': int(class_room.split()[0]),
                'class_char': class_room.split()[1]}
```

Метод convert_class() принимает автоматический аргумент self, но его не использует. Подобные методы рекомендуется делать статическими. Для этого используется декоратор @staticmethod.

По сути, статический метод не принимает автоматический аргумент self и тем самым говорит программисту, что использование данного метода никак не связано с обработкой атрибутов/методов текущего экземпляра.

```
@staticmethod
def convert_class(class_room):
        """
              '<class_num> <class_int>' --> {'class_num': <class_num>, 'class_int':
<class_int>}
        """
        return {'class_num': int(class_room.split()[0]),
                'class_char': class_room.split()[1]}
```

Расширение встроенных типов

Помимо реализации объектов новых типов, классы иногда используются для расширения функциональных возможностей встроенных типов языка Python с целью обеспечения поддержки более экзотических структур данных.

```
# Расширяем стандартный class dict
class my_dict(dict):
# Добавляем свой метод
    def new_method(self):
        return "I'm new_method"
# Добавляем дополнительный функционал к существующему методу
    def __setitem__(self, key, value):
        print('Setting %r to %r' % (key, value))
        return super().__setitem__(key, value)
m_dict = my_dict({1: 2, 2: 3})
print(m_dict)
# Данная операция вызывает метод __setitem__
m_dict["new_key"] = "new_value"
print(m_dict)
# print(m_dict.keys())
print(m_dict.new_method())
```

Обратите внимание! При использовании механизма наследования можно не только добавлять и переопределять методы, но и расширять (модифицировать) существующие. Подробно о функции super() поговорим во второй части курса.

Еще один пример. Предположим, вас не устраивает тот факт, что стандартные списки начинают отсчёт элементов с 0, а не с 1. Это не проблема — вы всегда можете создать свой подкласс, который изменит эту характерную особенность списков.

```
class MyList(list):
    """
    Список - индексы которого начинаются с 1, а не с 0
    """
    def __getitem__(self, offset):
        print('(indexing % s at % s)' % (self, offset))
        return list.__getitem__(self, offset - 1)
x = MyList('abc')  # __init__ наследуется из списка
print(x)                # __repr__ наследуется из списка
print(x[1])             # MyList.__getitem__
print(x[3])             # Изменяет поведение метода суперкласса
x.append('spam')
print(x)                # Атрибуты, унаследованные от суперкласса list
x.reverse()
print(x)
```

Домашнее задание

1. Смотреть здесь https://github.com/GeekBrainsTutorial/Python_lessons_basic/tree/master/lesson07.

Большинство заданий делятся на три категории — easy, normal и hard:

- easy — простенькие задачи на понимание основ;
- normal — если вы делаете эти задачи, то вы хорошо усвоили урок;
- hard — наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи — это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard — не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

Дополнительные материалы

Всё то, о чём сказано здесь, но подробнее:

1. Перегрузка операторов.
2. Итераторы.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Самоучитель по python.
2. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).