Python. Уровень 1

Урок 6

Объектно-ориентированное программирование

Введение в ООП. Классы. Инкапсуляция, полиморфизм, наследование.

Словарь как структура данных

Иногда вам нужно работать с некоей сущностью, у которой можно выделить именованные параметры. Например, со студентами: каждый студент имеет именованные параметры, такие как имя, фамилия, дата рождения и т.д. Для описания и работы с такой структурой подходит словарь.

```
student = {
    "name": "Александр",
    "surname": "Иванов",
    "birth_date": '10.11.1998',
    "school": "8 гимназия",
    "class_room": "5 А"
}
student2 = {
    "name": "Александр",
    "surname": "Иванов",
    "birth_date": '10.11.1998',
    "school": "8 гимназия",
    "class_room": "5 А"
}
```

Потом возникает необходимость изменять и обрабатывать данные каждого студента, для этих задач вы начинаете писать функции.

```
# но функция - не очень удобное решение
def next_class(current_class):
    return str(int(current_class.split()[0]) + 1) + \
           ' ' + current_class.split()[1]
# т.к. количество функций постоянно увеличивается
def change_class(current_class):
    pass
```

И сложно запомнить, какая функция применяется к какому объекту.

Представьте, что у вас в программе 20 разных структур и для каждой по 10 функций обработки (для реальной программы это совсем немного). Получается 200 (!) перемешанных друг с другом функций, в которых вы рано или поздно запутаетесь.

Гораздо удобнее работать со сгруппированными структурами данных с помощью классов.

Введение в ООП

Весь окружающий нас мир состоит из объектов: стол, парта, доска — привычные нам объекты. У объектов есть определённые параметры (свойства): цвет, размеры, вес, материал, координаты и прочее. Рассматривая ООП-подход в программировании, будем опираться на аналогии с объектами реального мира.

Чтобы научиться писать полноценные ООП-программы, нужно понять суть этого подхода и преимущества, которые он даёт, проникнуться философией ООП.

Можно ли жить без ООП

Однозначно можно. Есть языки программирования, вообще не поддерживающие концепцию ООП (например, Pascal). Любую программу, написанную с ООП-подходом, можно написать и без него. Так зачем же тогда тратить столько времени на изучение ООП?

Запомните: ООП — это всего лишь инструмент, созданный для того, чтобы оказать вам помощь в написании больших программ.

Python является полностью объектно-ориентированным языком, в python всё является объектами. Только поняв ООП, вы сможете писать красивые, функциональные и легко расширяемые программы. Не ждите, что вы поймете всю суть ООП за 2 либо 3 недели, это довольно ёмкая концепция. Читайте лекции, выполняйте задания, пробуйте применять ООП-подход в своих проектах — и со временем всё станет на свои места. Как говорится: «Дорогу осилит идущий».

Классы

Что такое класс или тип? В реальном мире стол — это объект. Но когда его изготавливают, то руководствуются определённым описанием (знанием), что такое стол. Я могу сказать «стол», не имея в виду какой-то конкретный, но большинство поймёт, о чём идет речь, т.к. знает особенности этого предмета (крышка, четыре ножки и т. п.).

Класс — это описание объектов определённого типа. В каком-то смысле это абстракция без материального воплощения, которая позволяет систематизировать объекты той или иной системы.

На основе классов создаются объекты. Может быть множество объектов, принадлежащих одному классу. С другой стороны, может быть класс без объектов, реализованных на его основе.

Класс — это пользовательский тип, состоящий из методов и атрибутов.

С точки зрения ООП, класс представляет собой коллекцию данных. Использование классов даёт нам, прежде всего, преимущества абстрактного подхода в программировании.

Экземпляр класса (объект) создается путём вызова имени класса как функции с параметрами. Объект состоит из атрибутов и методов.

Атрибут — это переменная класса, **метод** — это функция. Отличия метода от функции в том, что у него есть первый параметр — **self**, который является ссылкой на тот объект, для которого был вызван метод. Т.е. метод всегда имеет доступ (через self) к атрибутам и другим методам своего объекта.

Основные свойства ООП — полиморфизм, наследование, инкапсуляция.

Примеры

```
# class - шаблон для создания объектов
# Классы содержат атрибуты - данные, и методы - функции для обработки данных
class Student:
# функция-конструктор - запускается автоматически при создании объекта
# (экземпляра класса)
    def __init__(self, name, surname, birth_date, school, class_room):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
        self.class_room = class_room
 # метод
    def next_class(self):
        self.class_room = str(int(self.class_room.split()[0]) + 1) + ' ' +\
                          self.class_room.split()[1]
    def get_full_name(self):
        return self.name + ' ' + self.surname
    def set_name(self, new_name):
        self.name = new_name
```

Используя класс как шаблон, можно создать любое количество объектов (экземпляров класса).

```
student1 = Student("Александр", "Иванов", '10.11.1998', "8 гимназия", "5 А")
student2 = Student("Петр", "Сидоров", '10.01.1995', "8 гимназия", "8 Б")
```

Аргументы передаются в функцию-конструктор.

Теперь в памяти существуют два независимых объекта типа (класса) Student, но с разным набором атрибутов. Класс — это просто шаблон, он описывает структуру объекта и его поведение.

Для обращения к атрибутам объектов используется точечный синтаксис.

```
# Выводим текущий класс первого ученика
print(student1.class_room)
# Вызываем метод, который переводит ученика в следующий класс
student1.next_class()
# Проверяем, изменился ли класс
print(student1.class_room)
# Выводим текущий класс второго ученика.
# Классы имеют общую структуру и методы, но различные данные
print(student1.class_room)
```

Методы — это обычные функции, но получающие в качестве первого аргумента self-ссылку на экземпляр класса.

```
# Вызываем метод, для получения полного отображаемого имени студента
print(student1.get_full_name())
print(student2.get_full_name())
```

По сути, вызов student1.get_full_name() равносилен вызову get_full_name(student1), только в случае с методами ссылка на экземпляр передаётся автоматически.

```
# Можно изменить значение любого атрибута, присвоив ему новое значение
student1.name = 'Вася'
print(student1.name)
```

Создавая новый класс, мы практически создаём новый тип данных. Между классами и типами есть разница, но пока вы можете считать эти понятия равнозначными.

Основные свойства ООП

Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонентов доступными только внутри класса.

Цель инкапсуляции — скрыть внутреннюю реализацию класса от пользователей. Для работы с данными, хранящимися в классе, определяются публичные методы.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

```
# Исходный класс Студента
class Student:
    def __init__(self, name, surname, birth_date, school, class_room):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
        self.class_room = class_room
    def next_class(self):
        self.class_room = str(int(self.class_room.split()[0]) + 1) + ' ' + \
                          self.class_room.split()[1]
    def get_full_name(self):
        return self.name + ' ' + self.surname
    def set_name(self, new_name):
        self.name = new_name
```

Теперь мы пользуемся классом Student, я его импортирую в свои различные программы.

```
students = [Student("Александр", "Иванов", '10.11.1998', "8 гимназия", "5 А"),
            Student("Петр", "Сидоров", '10.01.1995', "8 гимназия", "8 Б"),
            Student("Иван", "Петров", '12.11.1999', "8 гимназия", "4 В"),
            ]
# Учебный год закончился
for student in students:
    student.next_class()
for num, student in enumerate(students, start=1):
           print("{})   {}   класс:   {}".format(num,   student.get_full_name(),
student.class_room))
# ... и ещё много кода с использованием Student
```

Но потом вы заметили, что реализация метода .next_class() смотрится очень плохо. Чтобы написать хороший код, решили хранить информацию о классе по-другому.

```
class Student:
    def __init__(self, name, surname, birth_date, school, class_room):
#         ...
        self.class_room = {'class_num': int(class_room.split()[0]),
                           'class_char': class_room.split()[1]}
#     Класс храним в виде словаря {номер класса, буква класса}
#         ...
#     Теперь метод выглядит понятно и легкочитаемо
    def next_class(self):
        self.class_room['class_num'] += 1
```

Чтобы использовать новую реализацию класса, вам придётся переписать много готового кода, т.к. атрибут class_room — это словарь, а не строка.

Чтобы этого избежать, поступаем следующим образом:

```
class Student:
    def __init__(self, name, surname, birth_date, school, class_room):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
# Символ нижнего подчеркивания говорит пользователям класса,
# что атрибут для внутреннего использования
        self._class_room = {'class_num': int(class_room.split()[0]),
                            'class_char': class_room.split()[1]}
 # @property - позволяет обращаться к методу как к атрибуту
 # .class_room() --> .class_room
    @property
    def class_room(self):
                        return   "{}   {}".format(self._class_room['class_num'],
self._class_room['class_char'])
    def next_class(self):
        self._class_room['class_num'] += 1
    def get_full_name(self):
        return self.name + ' ' + self.surname
    def set_name(self, new_name):
        self.name = new_name
```

Старый код продолжает работать с новым классом без изменений.

Посмотреть данный пример и его работу, а также ещё несколько демонстраций инкапсуляции вы можете в демонстрационном проекте, ссылка на который дана в конце лекции.

Наследование

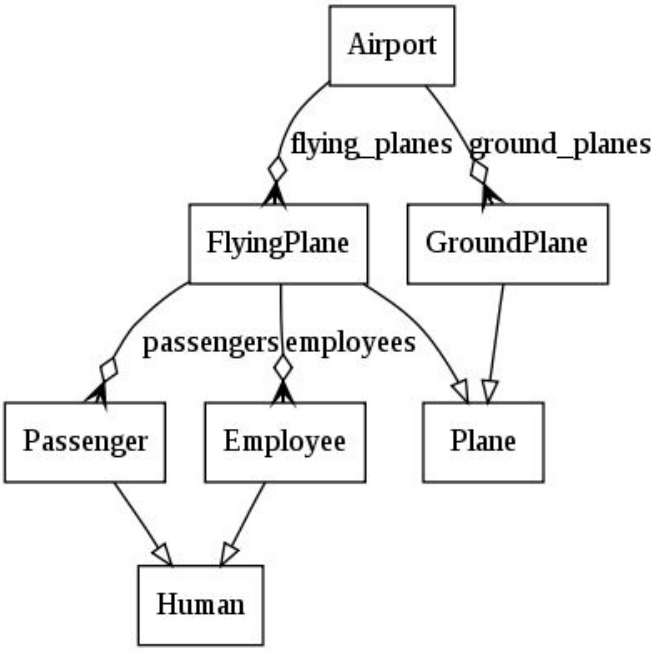
Наследование позволяет создавать специализированные классы на основе базовых. Это даёт возможность избегать написания повторного кода.

Например, у вас могут быть классы Студент и Преподаватель, оба являются людьми. Можно выделить общие атрибуты и методы, присущие людям, и вынести их в отдельный класс, это позволит избежать дублирования кода.

```
class Student:
    def __init__(self, name, surname, birth_date, school, class_room):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
        self._class_room = {'class_num': int(class_room.split()[0]),
                            'class_char': class_room.split()[1]}
    @property
    def class_room(self):
       return "{} {}".format(self._class_room['class_num'],
                             self._class_room['class_char'])
    def next_class(self):
        self._class_room['class_num'] += 1

    def get_full_name(self):
        return self.name + ' ' + self.surname

    def set_name(self, new_name):
        self.name = new_name

class Teacher:
    def __init__(self, name, surname, birth_date, school, teach_classes):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
        self.teach_classes = list(map(self.convert_class, teach_classes))

    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]}

    def get_full_name(self):
        return self.name + ' ' + self.surname

    def set_name(self, new_name):
        self.name = new_name

# Эти Классы описывают два различных объекта.
# Но часть информации у них общая (атрибуты, методы).
# Общую информацию выносим в Класс-предок (родитель):
class Person:
    def __init__(self, name, surname, birth_date, school):
        self.name = name
        self.surname = surname
        self.birth_date = birth_date
        self.school = school
    def get_full_name(self):
        return self.name + ' ' + self.surname
    def set_name(self, new_name):
        self.name = new_name

# Сами классы наследуем
```

```
class Student(Person):
    def __init__(self, name, surname, birth_date, school, class_room):
    # Явно вызываем конструктор родительского класса
        People.__init__(self, name, surname, birth_date, school)
    # Добавляем уникальные атрибуты
        self._class_room = {'class_num': int(class_room.split()[0]),
                            'class_char': class_room.split()[1]}
    # И уникальные методы
    @property
    def class_room(self):
       return "{} {}".format(self._class_room['class_num'],
                             self._class_room['class_char'])
    def next_class(self):
        self._class_room['class_num'] += 1

class Teacher(Person):
    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]}
```

Таким образом, практически любая серьёзная программа с использованием ООП-подхода представляет собой большое дерево классов.



Полиморфизм

Полиморфизм заключается в том, что в разных объектах одна и та же операция может выполнять различные функции. Слово «полиморфизм» имеет греческую природу и означает «имеющий многие формы». Простым примером полиморфизма может служить функция count(), выполняющая одинаковое действие для различных типов объектов: 'abc'.count('a') и [1, 2, 'a'].count('a'). Оператор плюс полиморфичен при сложении чисел и при сложении строк.

Полиморфизм позволяет нам работать с различными типами объектов так, что нам не нужно задумываться о том, к какому типу они принадлежат.

```
>>> 2 + 4
: 6
>>> '2' + '4'
: '24'
>>> (2, 4)+(2, 4)
:(2, 4, 2, 4)
```

Результат операции сложения (+) зависит от объектов, к которым применяется, т.е. от реализации метода, соответствующего применяемой операции.

Удобство полиморфизма в том, что вы можете не заботиться о том, с какими именно объектами работаете, главное — знать, что данные объекты поддерживают текущую операцию.

Подробнее об этой особенности — на следующем уроке.

Домашнее задание

1. Смотреть здесь https://github.com/GeekBrainsTutorial/Python_lessons_basic/tree/master/lesson06.

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. Введение в ООП.
2. Основные свойства ООП.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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