Классы в Python

Темы на сегодня:

- Зачем классы нужны аналитику
- Классы и объектно-ориентированное программирование
- Работа с классами

ЗАЧЕМ ЭТО ЗНАТЬ АНАЛИТИКУ

Когда речь заходит о классах в языках программирования, то сразу же возникает вопрос: «А зачем это знать аналитикам данных?»

- Во-первых, потому что аналитики не работают сами по себе. Часто они решают задачи в ходе работы над проектом. Основной труд в нём приходится на разработчиков, которые практически всегда используют классы для упрощения и масштабирования. Поскольку аналитикам приходится контактировать с ними (например, потому что разработчики оптимизируют код, написанный аналитиком для решения задачи), то полезно понимать помимо базового синтаксиса ещё и классы.
- Во-вторых, по причине того, что требования к аналитикам растут. Если 15 лет назад было достаточно уметь считать и работать в Excel, то сейчас для аналитиков даже проводят code review.

Возникает другой вопрос: «Для упрощения используются функции – неужели их недостаточно?»

Рассмотрим на примере: допустим, в масштабах какого-то проекта существует код по получению курса некоторой Валюты_1 с Ресурса_1. Код работает, но вдруг возникают новые требования:

- Нужно получать другие валюты
- Нужно получать не только курс валюты
- Нужно получать информацию с другого ресурса
- Нужно, чтобы код выполнял какую-то аналогичную задачу

В принципе, можно всё это учесть в функциях. Но такой код:

- Избыточен
- Имеет много повторений
- Сложен для понимания
- Сложен для изменения и масштабирования

Именно в этот момент нужно начать смотреть с сторону классов.

КЛАССЫ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

С понятием класса неразрывно связано понятие **объектно-ориентированного программирования** (ООП) – своеобразного подхода к написанию кода, где программа состоит из **экземпляров классов** – объектов, работа которых определяется, собственно, **классами** – наборами объединённых функций и свойств (если очень упрощённо). Эти функции называются **методами** и отвечают за, собственно, всю вычислительную работу.

В жизни класс можно сравнить с чертежом какого-то определённого изделия (например, пылесосов). А готовый пылесос, созданный по такому чертежу – экземпляр класса. У этого пылесоса есть технические параметры (цвет, мощность) – инициализирующие значения. Кроме того, у пылесоса есть функции «обычный режим», «турбо-режим», «моющий режим», которые определяют его работу – методы.

Есть 3 кита, на которых базируется такой подход:

- **Наследование** – принцип, по которому свойства и методы класса передаются дочернему классу (классу, который определяет более частный вариант). Например, все возможности пылесосов есть у промышленных пылесосов, но у последних есть и свои собственные функции (методы). Например, работать в обратную сторону для очистки пылесборника.
- **Инкапсуляция** – механизм защиты внутреннего устройства класса от внешнего воздействия. На примере того же пылесоса: он будет представлять собой цельное изделие, которое нельзя легко разобрать.
- **Полиморфизм** – возможность переопределить метод класса в нём и в его потомках. Например, в пылесосах по умолчанию будет обычное всасывание через трубу, а в премиум-пылесосе всасывание будет происходить через систему фильтров. Действие осталось прежним, но его принцип изменился.

Аналитикам наиболее важно первое свойство. Другие же задействуются довольно редко.

При работе с большим количеством экземпляров класса (много пылесосов) может возникнуть вопрос: каким образом так получается, что методы «узнают», что нужно работать именно с этим экземпляром класса. Допустим, есть 2 пылесоса с функциями «вкл» и «выкл». Как на чертеже показать, что каждый пылесос должен включать и выключать именно себя, а не другой, когда к нему обращаешься? Для этого в классе (чертеже) есть особый метод **__init__** (создание). Этот метод называется **конструктором класса**. В него записываются свойства, которые задаются при создании экземпляра (на примере пылесоса – цвет, мощность и тд.) + свойство **self** – своеобразный указатель на самого себя. Именно благодаря этому свойству и решается проблема доступа.

РАБОТА С КЛАССАМИ

Итак, инициализация класса в общем виде выглядит так:

```
class Имя_класса:
      def __init__(self, параметры_инициализации):
            свойство_1
            ...
            свойство_N

      def метод_1(self, параметры)
            Тут что-то происходит

      ...

      def метод_N(self, параметры)
            Тут что-то тоже происходит
```

После этого можно создавать экземпляры класса и работать с ними:

```
переменная_1 = Имя_класса(переданное_свойство_1, …, переданное_свойство_N)
переменная_1.метод_1(параметры)              # Результат выполнения метода
```

Дочерние классы инициализируются следующим образом. По сути, внутри конструктора дочернего класса вызывается родительский класс с определёнными параметрами.

```
class Имя_дочернего_класса(имя_родительского класса):
      def __init__(self):
            super().__init__(self, параметры_родительского класса)
            свойство_дочернего_класса_1
            ...
            свойство_дочернего_класса_N

      Методы
```

Пример:

```
# Класс Vacuum_cleaner (Пылесос)
class Vacuum_cleaner:
      # При создании передаём имя пылесоса и модель
      def __init__(self, name):
            self.name = name
            self.state = 'Off'       # По умолчанию каждый пылесос выключен
```

```
    # Метод включения-выключения. Нужно передать команду ‘on’ или ‘off’
    def switch(self, command):
        if command == ‘on’:
            self.state = ‘On’
        else:
            self.state = ‘Off’

    # Метод, возвращающий строку вида «Имя пылесоса: состояние»
    def state(self):
        result = self.name + ‘: ’ + self.state
        return result

# Дочерний класс
class Mini_vacuum_cleaner(Vacuum_cleaner):
    def __init__(self)
        super().__init__(self, name)

    def is_it_mini(self)
        return true

# Создаём экземпляры класса (пылесосы) и запускаем методы
cleaner_1 = Vacuum_cleaner(‘Philips’)
cleaner_2 = Vacuum_cleaner(‘Dyson’)
cleaner_3 = Mini_vacuum_cleaner(‘Kitfort’)
cleaner_1.switch(‘on’)
cleaner_2.switch(‘on’)
cleaner_3.switch(‘on’)
cleaner_1.switch(‘off’)
print(cleaner_1.state())          # Philips: Off
print(cleaner_2.state())          # Dyson: On
print(cleaner_3.state())          # Kitfort: On
print(cleaner_3.is_it_mini())     # true
```