

# PYTHON

## Лекция 8

# ПЛАН ЛЕКЦИИ

- Классы

# СТРУКТУРА

- Ключевое слово `class`
- Имя, двоеточие, тело со смещением

```
1 # Простейшее определение
2 class C:
3     pass
```

# КЛАСС КАК ОБЪЕКТ

- Класс сам по себе является объектом
- В теле класса можно определять переменные
- И исполнять обычный код
- Этот код исполняется сразу

# КОД УРОВНЯ КЛАССА

```
1 print(1)
2
3 class C:
4     print('C.1')
5     V = 1234
6     print('C.2')
7
8     def m():
9         print(222)
10
11 print(2)
12 print(C.V)
```

# АТТРИБУТЫ КЛАССА

- Переменная, определенная на уровне класса, становится свойством класса как объекта
- Аналог статических полей класса в Java/C++
- Но не во всем
- В момент инициализации переменных класса самого класса еще нет

# ПРИМЕР

```
1 class Config:
2     __SEC_PER_MIN = 60
3
4     TIMEOUT_MIN = 100
5
6     TIMEOUT_SEC = TIMEOUT_MIN * __SEC_PER_MIN
7
8     # так нельзя - Config еще не существует
9     # TIMEOUT_SEC = Config.TIMEOUT_MIN *
10         Config.__SEC_PER_MIN
11
12 print(Config.TIMEOUT_SEC)
```

# АЛЬТЕРНАТИВА

```
1 class Config:
2
3     TIMEOUT_MIN = 100
4
5     @classmethod # детали попозже
6     def initialize(cls)
7         sec_per_min = 60
8         cls.TIMEOUT = cls.TIMEOUT_MIN * sec_per_min
9
10 Config.initialize()
11
12 print(Config.TIMEOUT)
```



# МЕТОДЫ/ФУНКЦИИ КЛАССА

- Внутри класса можно определять функции
- Это экзотичная практика, но так можно
- И вызывать их через ссылку на класс
- В схеме разрешения ссылок на переменные уровень класса не участвует
- К методам это тоже относится

# ПРИМЕР

```
1 V = 12
2 class C:
3     V = 23
4
5     def hello(name='world'):
6         print('hello', name)
7
8     def m():
9         print(V)
10        print(C.V)
11
12 C.hello()
13 C.m()
```

# СОЗДАНИЕ ОБЪЕКТА

- Вызываем класс как функцию
- Надо ли передавать параметры - зависит от наличия символа

`'__init__'`

в пространстве имен класса

- Если ничего специально не делать - его там нет
- Тогда параметры не нужны

# ПРИМЕР

```
1 class C:  
2     pass  
3  
4 c1 = C()  
5 c2 = C()  
6 print(c1, c2)
```

# СОЗДАНИЕ ОБЪЕКТА

- Наличие

```
def __init__(...):
```

внутри класса меняет дело

- Неформально это называется "конструктором"
- Я бы это назвал "инициализирующий метод"
- Потому что на входе в него объект уже сконструирован

# ССЫЛКА НА ОБЪЕКТ

- В целом в ООП метод - это функция, один из параметров которой - объект
- Того класса, чей это метод
- Почти во всех языках этот параметр передается неявно
- И называется как-то вроде

`'this'`

# ССЫЛКА НА ОБЪЕКТ

- В Python все делается явно
- У метода первый параметр - это объект
- Назвать можно как угодно
- Жесткая традиция - называть

`'self'`

# ССЫЛКА НА ОБЪЕКТ

- Если

`__init__`

определен - Python вызовет его

- Первым параметром передаст уже созданный объект
- Остальными - все, что указано в скобках
- По всем правилам именованных/позиционных

`*args/**kwargs`



# ПРИМЕР

```
1 class C:
2     def __init__(self):
3         print("self", self)
4
5 c1 = C()
6 c2 = C()
7 print(c1, c2)
```

# ПРИМЕР

```
1 class Point2D:
2     def __init__(self, x, y, /):
3         self._x = x
4         self._y = y
5
6 p1 = Point2D(1.2, 2.3)
7 p2 = Point2D(2.2, 3.1)
8
9 print(p1)
10 print(p2)
```

# ДРУГИЕ МЕТОДЫ

- Можем определять свои методы
- И работать с ними по своему усмотрению
- Например вычислять расстояние между точками
- Создать объект "прямая"
- Определять точку пересечения прямых и т.п.

# КОНТРАКТНЫЕ МЕТОДЫ

- Есть соглашения по именам методов
- Методы с определенными именами Python вызывает в определенных ситуациях
- Например, для визуального представления объекта через `print` можно определить метод

`__str__`

- Он должен вернуть строку

# ПРИМЕР

```
1 class Point2D:
2     def __init__(self, x, y):
3         self._x = x
4         self._y = y
5
6     def __str__(self):
7         return f"Point({self._x}, {self._y})"
8
9 p1 = Point2D(1.2, 2.3)
10 p2 = Point2D(2.2, 3.1)
11
12 print(p1)
13 print(p2)
```

# REPR

- Можно зайти в интерпретатор и создать экземпляр

```
Point2D
```

- Напечатать в командной строке напрямую и через

```
str
```

- Увидеть разницу

# REPR

- Для командной строки есть отдельный метод -  
`__repr__`
- И встроенная функция  
`repr`

# REPR

- Есть определенная традиция относительно `repr`
- Чтобы результат можно было скопировать в строку интерпретатора
- И получить конструктор текущего объекта
- Для `Point2D`:

```
f"Point2D({self._x}, {self._y})"
```



# CALL

- Можно определить метод `__call__`
- С произвольным набором параметров
- Объекты такого класса будут 'callable objects'
- Их можно будет "вызывать" как функции

# ПРИМЕР

```
1 class Binom:
2     def __init__(self, a, b, c):
3         self._a = a
4         self._b = b
5         self._c = c
6
7     def __call__(self, x, /):
8         return self._a * x * x + self._b * x + self._c
9
10 # .....
```

# ПРИМЕР

```
1 # .....
2
3     def __str__(self):
4         return f"{self._a}*x^2 + {self._b}*x + {self._c}"
5
6     def __repr__(self):
7         return f"Binom({self._a}, {self._b}, {self._c})"
8
9
10 b = Binom(1.2, 2.3)
11
12 print(b(5))
13 print(b(10))
```

# ДРУГИЕ: КРАТКИЙ ОБЗОР

- Можно определить арифметические операции
- Итерирование
- Извлечение объекта по индексу
- Реакцию на отсутствующие атрибуты
- Многое другое

# ОСОБЕННОСТИ МОДЕЛИ

- Нет отдельного описания схемы данных объекта класса
- Все определяется динамически исполнением кода
- Как следствие - не гарантируется одинаковый набор свойств у объектов одного класса
- Разночтения могут появиться из-за исполнения разных веток
- Или через назначение свойств объектам извне

# ПРИМЕР

```
1 class Person:
2     def __init__(self, name, birth_year, *kwargs):
3         self._name = name
4         self._birth_year = birth_year
5         if self._birth_year < 2010:
6             self._father = kwargs.get('father')
7
8
9
10 p1 = Person('Ivan', 1980)
11 p2 = Person('Vasya', 2018, father=p1)
12 p3 = Person('Natalia', 1982)
13 p2._mother = p3
```

# ОСОБЕННОСТИ МОДЕЛИ

- Каждый объект хранит ссылку на его класс
- При обращении к элементу словаря символов объекта требуемого имени может там не быть
- Тогда это имя ищется в классе
- Но можно непосредственно обратиться к элементу класса

# ПРИМЕР

```
1 class C:
2     V1 = 1
3     V2 = 2
4
5     def __init__(self):
6         self.V2 = 22
7         self.V3 = 33
8
9     def m(self):
10        print(self.V1, self.V2, self.V3)
11
12 c = C()
13 c.m()
14 C.m(c)
```



# ПРИМЕР

```
1 class C:
2     V1 = 1
3     V2 = 2
4
5     def __init__(self):
6         self.V2 = 22
7         self.V3 = 33
8
9     def m(self):
10        print(self.V1, self.V2, self.V3)
11
12    def m2(self):
13        print(self.V1, C.V4)
14    # .....
```

# ПРИМЕР

```
1      # .....
2
3      def update(self):
4          self.V3 = 44
5          C.V1 += 1
6          C.V4 = 4
7
8      c1 = C()
9      c2 = C()
10     c1.m()
11     c1.update()
12     c2.m2()
```

# МЕТОДЫ КЛАССА

- Вспомним пример с инициализацией
- Мы хотим иметь метод, в котором был параметр типа

`self`

- Только, чтобы в нем была ссылка именно на класс как на объект
- Этого добивается декоратор

`@classmethod`

- Традиционное название такого параметра -

`cls`

# ПРИМЕР

```
1 class C:
2     @classmethod
3     def cm(cls, p1, *, p2):
4         print(cls, p1, p2)
5
6
7 C.cm(1, p2=2)
8 C().cm(11, p2=22)
```

# ОБЩИЕ ПРИНЦИПЫ ОО-ДИЗАЙНА

- Анализируется предметная область
- Выделяются сущности и их отношения
- Сущности могут соответствовать предметам реального мира
- А могут - вспомогательным конструкциям, возникшим в результате декомпозиции

# ОБЩИЕ ПРИНЦИПЫ ОО-ДИЗАЙНА

- У сущностей есть состояние и поведение
- Пример сущности: пользователь соцсети
- Состояние: данные профиля, отправленные сообщения, подписки
- Поведение: действия по отправке сообщения

# ОБЩИЕ ПРИНЦИПЫ ОО-ДИЗАЙНА

- Для сущностей с однотипным поведением и состоянием заводим класс
- Сущности представляются объектами класса
- Состояние - атрибутами
- Поведение - методами

# ОБЩИЕ ПРИНЦИПЫ ОО-ДИЗАЙНА

- Детализация атрибутов зависит от задачи
- Где-то нужен текущий возраст, где-то год рождения, где-то дата
- Атрибуты могут зависеть друг от друга
- Предпочтительнее вычислять зависимые в методах, если это не дорого



# ИЗМЕНЯЕМОСТЬ/ НЕИЗМЕНЯЕМОСТЬ

- Неизменяемость объекта для дизайна предпочтительнее
- Упрощает анализ объекта
- Изменения реализуются через создание нового объекта
- Обратная сторона - цена создания нового объекта

# ИЗМЕНЯЕМОСТЬ/ НЕИЗМЕНЯЕМОСТЬ

- Для реализации градиентного спуска предпочтительнее объект с изменяемым состоянием
- Но если есть датасет, который мы обучаем - то стоит подумать о двух отдельных объектах
- Объект "датасет с моделью" порождать из объекта "просто датасет"
- Неизменяемость хорошо подходит для конвейерной обработки малых объектов

# ШАБЛОНЫ СОЗДАНИЯ ОБЪЕКТОВ

- Простые объекты создаются непосредственно конструктором
- Но это не всегда идеальный вариант
- Есть три базовых шаблона создания объектов: singleton, factory, builder
- В конечном итоге они могут вызывать конструктор

# SINGLETON

- Нам нужен только один объект данного класса
- Не обращаемся непосредственно к конструктору, а к сервисному методу
- Который либо создает объект и возвращает, либо возвращает уже созданный
- Можно вручную, можно универсально
- Есть сторонние библиотеки с декораторами

# FACTORY

- Есть узкий набор классов и их конфигураций для определенной задачи
- Например, есть несколько хранилищ данных
- Они хранят похожие данные
- Но отличаются актуальностью данных, детализацией, скоростью работы

# FACTORY

- И объекты, отвечающие за взаимодействие с хранилищами, могут быть дорогими по памяти
- И их должно быть немного, но не факт, что ровно один
- Тогда логика выбора конкретного хранилища может быть спрятана в отдельном методе
- Если добавляем логику кеширования - нужно какое-то состояние
- Получаем factory-класс

# BUILDER

- Есть объект с большим количеством атрибутов
- Не хотим писать огромный конструктор
- Хотим строить состояние по кусочкам
- Создаем вспомогательный класс
- Который в своем состоянии накапливает состояние создаваемого объекта

# ПРИМЕР

```
1 class UserProfile:
2     def __init__(self, **kwargs):
3         pass
4
5 # Синтакс не очень приспособлен
6 ProfileBuilder()\
7     .phone('11111')\
8     .address('aaaaa')\
9     .name('ccccc')\
10    .build()
```



# BUILDER

- Позволяет разделить фазу построения объекта - где может потребоваться изменяемость
- И фазу использования - где можно полагаться на неизменяемость
- Частично построенный builder можно передавать параметром и возвращать значением

# SOLID

- S - single responsibility
- В хорошем дизайне каждый класс отвечает за небольшую хорошо очерченную область
- Делегируя детали другим классам
- Важно выделять абстракции и выносить их в классы

# SOLID

- O - Open/closed
- Классы открыты к использованию, закрыты к модификации
- Интерфейс класса должен позволять использовать его в разных ситуациях
- Не должно быть внутренних костылей для разных случаев использования

# SOLID

- L - Liskov substitution
- I - interface segregation
- D - dependency inversion

