PYTHON

Лекция 11

ПЛАН ЛЕКЦИИ

- Классы

МЕТОДЫ/ФУНКЦИИ КЛАССА

- Методы задаются через def на уровне класса
- Чисто технически это функция, заданная в пространстве имен класса
- И может быть вызвана в таком виде

ПРИМЕР

```
 1 class C:
 2     V = 23
 3     def m1(name='world'):
 4         return ('hello', name)
 5     def m2(self, v):
 6         return (self, v)
 7
 8
 9 print(C.V)
10 print(C.m1())
11 print(C.m1('vasya'))
12 print(C.m2(12, None))
```

ПРИМЕР

```
1 C.m3 = C.m2
2 C.m4 = lambda self: self.__class__
3
4 print(C.m3('vasya'))
5 print(C.m4(None))
```

СОЗДАНИЕ ОБЪЕКТА

- Вызываем класс как функцию
- (Сам класс - вызываемый объект)
- В первом приближении - параметры вызова определяются наличием метода __init__
- Если есть - на 1 меньше, чем в нем
- Если нет - без параметров

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__`

  опеределен - 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.

СПЕЦИАЛЬНЫЕ 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

```python
class Point2D:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __str__(self):
        return f"Point({self._x}, {self._y})"

p1 = Point2D(1.2, 2.3)
p2 = Point2D(2.2, 3.1)

print(p1)
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

```
# ..............................

    def __str__(self):
        return f"{self._a}*x^2 + {self._b}*x + {self._c}"

    def __repr__(self):
        return f"Binom({self._a}, {self._b}, {self._c})"


b = Binom(1.2, 2.3)

print(b(5))
print(b(10))
```

ДРУГИЕ СПЕЦИАЛЬНЫЕ

- Можно определить арифметические операции
- Итерирование
- Извлечение объекта по индексу
- Реакцию на отсутствующие аттрибуты
- Вернемся к этому позже

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 # плохой вариант
 6         self.__class__.V4 = 4 # получше
 7
 8 c1 = C()
 9 c2 = C()
10 c1.m()
11 c1.update()
12 c2.m2()
```

МЕТОДЫ КЛАССА

- Вспомним ситуацию с инициализацией атрибута класса из метода
- Через переменную класса - плохо
- Через self.__class__ - получше
- Но вообще для этого есть методы класса
- Объявляются через декоратор @classmethod

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)
```

ПРИМЕР

```
 1 class Config:
 2     path = '/usr'
 3
 4     @classmethod
 5     def change_default(cls, path):
 6         cls.path = path
 7
 8     def m(self):
 9         self.change_default('/local/path')
10
11
12 C.path = '/opt/usr'
13 C.m()
14 C().m()
```

СТАТИЧЕСКИЕ МЕТОДЫ

- Для логики, связанной с объектами класса
- Но не привязанной к состоянию объекта
- Пример: класс для управления конфигурацией
- Умеет собирать конфигурацию из разных источников разного приоритета

СТАТИЧЕСКИЕ МЕТОДЫ

- Такому классу нужно уметь прочитать текстовый файл
- Построчно, разбивая по знаку `=`
- Логика чтения вызывается из логики формирования конфигурации
- Но она не влияет на состояние объекта-конфигурации

ПРИМЕР

```
1 class Config:
2     def __init__(cls):
3         self.global_cfg = self.read_as_map('/etc/app.cfg')
4         self.user_cfg = self.read_as_map(
5                                 '/home/user/app.cfg')
6
7     @staticmethod
8     def read_as_map(name):
9         ...
```

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 возвраст, 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
- Оборотная 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 разные 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 случае использования

SOLID

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