eebb

ЯЗЫК PYTHON

Лекция 13. Классы

Мотивация

- Есть строки, числа, коллекции
- Хочется работать с объектами реального мира
- Тексты, пользователи, товары
- Можно приспособить что-то

Мотивация

- Например, словарь
- Поймем, какие атрибуты нужны
- Заведем ключи в словаре
- Значение атрибута объекта реального мира - значение по атрибуту

Пример

```
 1 text = {'source': 'To be or not to be',
 2         'n_words': 6,
 3         'n_letters': 13
 4        }
 5
 6 def init_text(txt):
 7     return {'source': txt,
 8             'n_words': len(txt.split()),
 9             'n_letters': sum(len(w) for w in txt.split())
10            }
11
12 data = init_text('To be or not to be')
13 print(data)
```

Мотивация

- Можно придумать функции, которые принимают такой словарь как аргумент
- Извлекают его атрибуты
- Что-то меняют или преобразовывают
- Сравним это со встроенными объектами

Пример

```
1 s = 'to be or not to be'
2 s.split()
3 data = [1, 2, 3]
4 data.append(123)
```

Что хотим

- Чтобы работа с атрибутами была встроена в язык
- Чтобы функции для работы с объектом одного типа определялись рядом
- Чтобы "основной" объект был не в скобках
- А перед именем функции (через точку)

Понятие класса

- Схема, по которой создаются объекты
- Содержит определения функций, работающих над объектами
- Такие функции называются методами
- Содержим описания данных
- Иногда прямо, иногда косвенно

Определение класса

- Ключевое слово class
- Имя класса
- Двоеточие
- Определения методов с отступом

Определение класса

- Есть особый метод __init__
- Принимает не меньше одного параметра
- Первый параметр обычно называется self
- Остальные - по ситуации
- Их может не быть

Пример

```
 1 class Text:
 2     print("define Text")
 3 
 4     def __init__(self, data):
 5         print("init. self:", self)
 6         print("init. data:", data)
 7 
 8 print("start")
 9 t = Text("To be or not to be")
10 print("t", t)
```

Разберем пример

- Python работает "сверху вниз"
- Видит определение класса
- Исполняет то, что идет с отступом
- В частности, читает определения функций (методов)

Разберем пример

- Принимает их к сведению
- Не исполняет
- Добавляет их куда-то
- Но не в глобальный контекст
- __init__ имеет особый смысл

Разберем пример

- Продолжается исполнение за определением класса
- Python видит что-то, похожее на вызов функции
- С именем класса на месте имени функции
- Python создает объект класса

Разберем пример

- Объект - это место в памяти
- У каждого объекта - свое место
- В переменной хранится указатель на это место
- В объекте могут быть данные
- У нас пока их нет

Разберем пример

- В классе может не быть метода __init__
- Тогда создание объекта (класса C) будет выглядеть так: C()
- В классе может быть метод __init__ с одним параметром
- По традиции его называют self
- Тогда создание объекта будет выглядеть так же

Разберем пример

- Но после создания объекта будет вызван __init__
- И в качестве параметра будет передан свежесозданный объект
- Который можно проинициализировать внутри __init__
- Но инициализировать лучше из каких-то данных

Разберем пример

- Поэтому у __init__ могут быть другие параметры
- В определении они идут следом за self
- А в месте создания объекта - они идут в скобках
- У нас уже есть параметр
- Мы его пока только печатаем

Научимся инициализировать

- В мотивирующем примере мы начинали со словаря
- У каждого Python-объекта есть за кулисами свой словарь
- В нем можно создавать элементы
- И извлекать уже созданные
- Для этого есть свой синтаксис

Научимся инициализировать

- Можем взять переменную, указывающую на объект
- Добавить точку
- И имя атрибута
- Это можно использовать в левой части присваивания
- Атрибут создается или обновляется

Научимся инициализировать

- А можно использовать в выражении
- Если атрибут есть, подставится его значение
- Если нет - получим исключение
- Это можно использовать в левой части присваивания
- Внутри __init__ переменная, указывающая на объект - это self

Пример

```
 1 class Text:
 2     def __init__(self, data):
 3         self._data = data
 4         self._n_words = len(data.split())
 5 
 6     def get_data(self):
 7         return self._data
 8 
 9     def n_words(self):
10         return self._n_words
11 
12 
13 t = Text("To be or not to be")
14 print("t", t.get_data())
15 print("t", t.n_words())
```

Разберем пример

- Передаем в __init__ один параметр
- Сохраняем его в атрибуте _data
- Подчеркивание в имени - соглашение
- Означает "для внутреннего использования"

Разберем пример

- Создаем атрибут _n_words
- Записываем в него количество слов
- Возможен другой вариант
- Считать прямо в n_words

Пример

```
 1 class Text:
 2     def __init__(self, data):
 3         self._data = data
 4 
 5     def get_data(self):
 6         return self._data
 7 
 8     def n_words(self):
 9         return len(self._data.split())
10 
11 
12 t = Text("To be or not to be")
13 print("t", t.get_data())
14 print("t", t.n_words())
```

Пример

```
 1 class Text:
 2     def __init__(self, data):
 3         self._data = data
 4         self._n_words = None
 5 
 6     def get_data(self):
 7         return self._data
 8 
 9     def n_words(self):
10         if self._n_words is None:
11             self._n_words = len(self._data.split())
12         return self._n_words
13 
14 
15 t = Text("To be or not to be")
```

Атрибуты класса

- Есть атрибуты объекта
- А бывают атрибуты класса
- Например, какие буквы считаем гласными
- Что считаем пробелами
- Свойства, относящиеся ко всем объектам

Атрибуты класса

- Методы определяются как функции внутри определения класса
- Атрибуты класса определяются как переменные внутри определения класса
- Традиционно называются большими буквами
- Можно обращаться через имя класса или объекта

Пример

```
 1 class Text:
 2
 3     VOWELS = 'auioey'
 4
 5     def __init__(self, data):
 6         self._data = data
 7         self._n_vowels = None
 8
 9     def get_data(self):
10         return self._data
11
12     def n_vowels(self):
13         if self._n_vowels is None:
14             self._n_vowels = sum(1 for c in self._data if c
15         return self._n_vowels
```

Пример

```
 1 class Text:
 2
 3     VOWELS = 'auioey'
 4
 5     def __init__(self, data):
 6         self._data = data
 7         self._n_vowels = None
 8
 9     def get_data(self):
10         return self._data
11
12     def n_vowels(self):
13         if self._n_vowels is None:
14             self._n_vowels = sum(1 for c in self._data if c
15         return self._n_vowels
```

Вспомним контексты

- Есть глобальный контекст
- В нем есть функции и глобальные переменные
- И еще классы
- Функция видит свой локальный контекст и глобальный

Вспомним контексты

- Встреченная в выражении внутри функции переменная ищется в локальном контексте
- Если не нашли, ищем в глобальном
- А если функция вложена в другую, то объемлющая - еще один контекст
- В котором ищется после локального, но перед глобальным

Поймем отличие классов

- Синтаксически класс для метода - как объемлющая функция
- И в классе есть свои переменные
- Но класс не участвует в поиске обычной переменной
- В примере нельзя в методе написать просто VOWELS

Типы методов

- Методы в примерах работают с объектами класса
- Иногда обращаются к атрибутам класса
- Могут через имя класса, могут через self
- Предположим, что в методе не хотим работать с объектом
- Только с атрибутами класса

Типы методов

- Например в некоторых атрибутах класса хранится его конфигурация
- А в методе хотим ее напечатать
- Или изменить
- Можем использовать и обычный метод
- Но для него нужен объект

Типы методов

- Объекта может не быть под рукой
- А создавать объект только ради работы с атрибутами класса не хочется
- Есть специальный вид методов - методы класса
- Задаются с помощью декораторов

Декораторы

- Чисто формально - синтаксическая конструкция
- Амперсанд + идентификатор
- Ставится перед функцией, методом или классом
- Модифицирует то, перед чем ставится

Декораторы

- Можно нацчится писать свои декораторы
- Но сейчас мы не об этом
- Есть набор уже определенных декораторов
- Один из них - @classmethod

classmethod

- Пишем перед определением метода
- Меняется интерпретация первого параметра
- Теперь это ссылка на класс
- Да, класс - это тоже объект

Пример

```
 1 class Text:
 2
 3     VOWELS = 'auioey'
 4
 5     def __init__(self, data):
 6         self._data = data
 7         self._n_vowels = None
 8
 9     @classmethod
10     def print_settings(self):
11         print(self.VOWELS)
12
13
14 Text.print_settings()
```

staticmethod

- А бывает так, что мы хотим добавить метод
- Который делает что-то вспомогательное для класса
- Но не привязан ни к объекту, ни к классу
- Это статический метод

staticmethod

- Обозначается декоратором @staticmethod
- Специальных параметров нет
- Пример: в обработке текста нужна логика разбиения строки на слова
- Обычно похитрее, чем `split`

staticmethod

- Можно реализовать как @staticmethod
- Особенно если нет привязки к атрибутам класса
- Если есть - возможно, лучше @classmethod
- Вызов статического метода - только через класс

Пример

```
 1 class Rational:
 2
 3     def __init__(self, p, q):
 4         self._p = p
 5         self._q = q
 6
 7     @staticmethod
 8     def integer(value):
 9         return Rational(value, 1)
10
11
12 print(Rational.integer(10))
```

Напечатать объект

- Хочется напечатать объект своего класса
- Можно передать в print
- Будет некрасиво
- Можно печатать значения атрибутов
- Будет неудобно

Напечатать объект

- Можно определить специальный метод - ___str__
- Принимает только self
- Возвращает строку
- Его вызовет print перед печатью объекта

Пример

```
 1 class Rational:
 2
 3     def __init__(self, p, q):
 4         d = math.gcd(p, q)
 5         self._p = p // d
 6         self._q = q // d
 7
 8
 9     def __str__(self):
10         return f'{self._p} / {self._q}'
11
12     @staticmethod
13     def integer(value):
14         return Rational(value, 1)
15
```

Специальные классы

- Классы определяют состояние и поведение
- Поведение может быть сложным
- Но может быть очень простым
- В пределе - просто храним вместе атрибуты
- И извлекаем значения

Специальные классы

- namedtuple - именованные кортежи
- Объекты, сочетающие свойства кортежей и словарей
- Неизменяемые, что не всегда удобно
- Есть аннотация @dataclass
- Создает нечто похоже, но более гибкое