Самое необходимое о классах

Роман Афанаскин, 2022

Правила

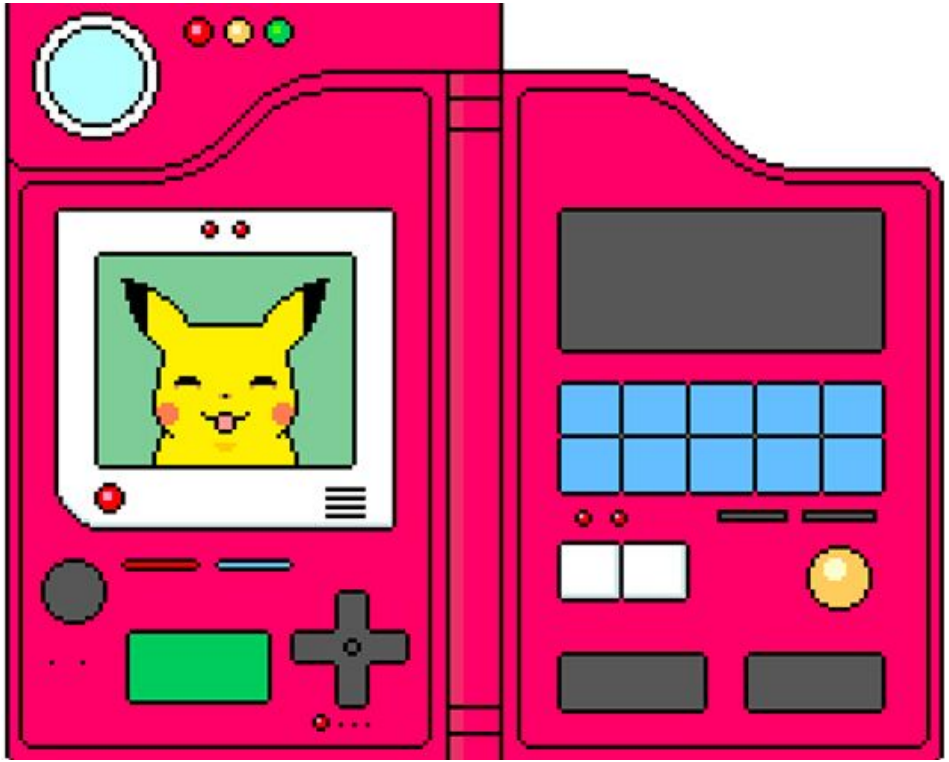
- Задаем любые вопросы
- Ведем диалог
- Будем писать код и показывать результаты

План лекции

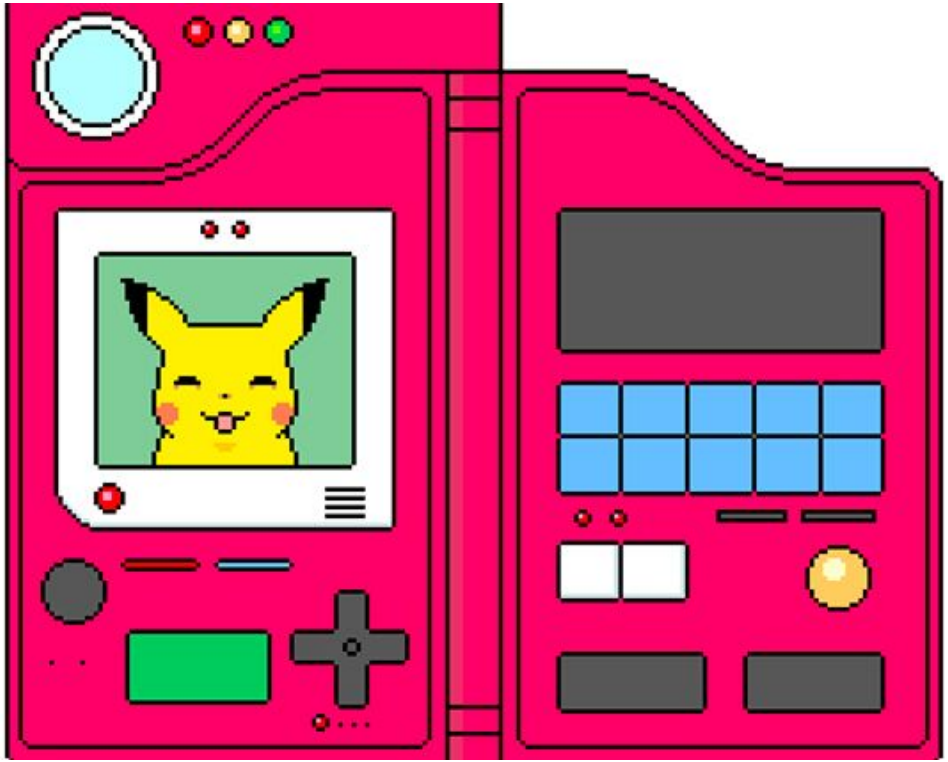
- Применение классов
  - Варианты реализации
  - Сложное поведение
- Мини практика
- Перерыв

- Синтаксис
  - Атрибуты
  - Методы
  - Свойства
- Мини практика
- Перерыв

Что это?



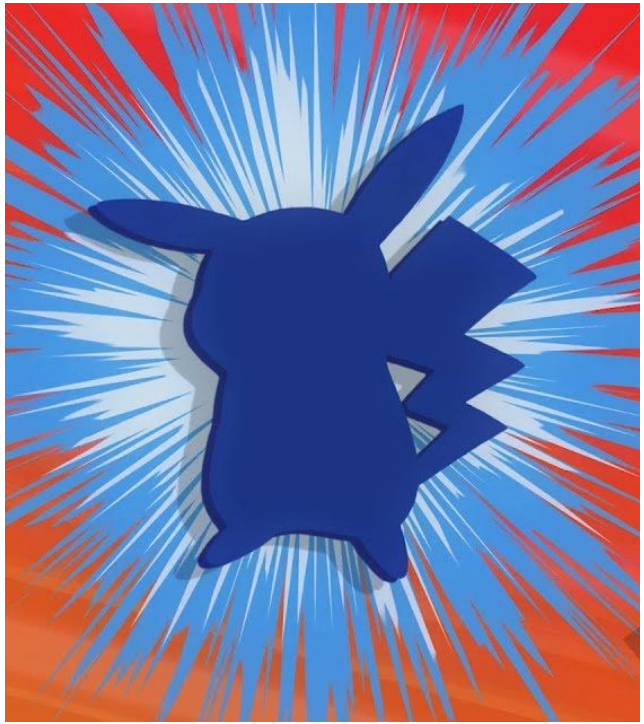
Pokedex



Возможности:

- Поиск по названию
- Просмотр детальной информации
- Добавление/редактирование/изменение

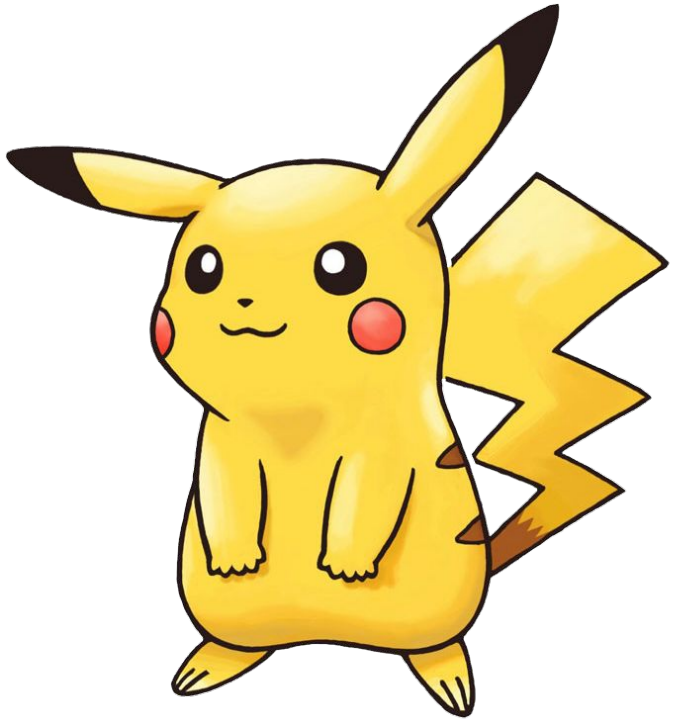
Покемон



Покемон

- Имя
- Категория
- Рост
- Вес
- Сильные стороны
- Слабые стороны

Покемон



Покемон

- Имя: Pikachu
- Категория: Mouse
- Рост: 40.64 см
- Вес: 5.99 кг
- Сильные стороны: Electric
- Слабые стороны: Ground

Покемон

```python
# Tuple
('Pikachu', 'Mouse', 5.99, 40.64, ['Electric'], ['Ground'])


# Dict
{
  'name': 'Pikachu',
  'category': 'Mouse',
  'height': 40.64,
  'weight': 5.99,
  'strengths': ['Electric'],
  'weaknesses': ['Ground'],
}
```

Покемон

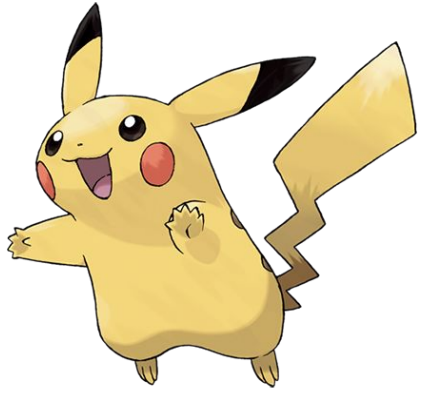
```python
# Namedtuple
from collections import namedtuple

attributes = ['name', 'category', 'height', 'weight', 'strengths', 'weaknesses']
Pokemon = namedtuple('Pokemon', attributes)

pikachu = Pokemon(
  name='Pikachu',
  category='Mouse',
  height=40.64,
  weight=5.99,
  strengths=['Electric'],
  weaknesses=['Ground'],
)
```

Покемон

```python
from typing import List


class Pokemon:
    def __init__(self, name: str, category: str, height: float, weight: float, strengths: List[str],
                 weaknesses: List[str], next_generation=None):
        self.name = name
        self.category = category
        self.height = height
        self.weight = weight
        self.strengths = strengths
        self.weaknesses = weaknesses
        self.next_generation = next_generation

    def evolve(self):
        if self.next_generation:
            pass
```

Покемон

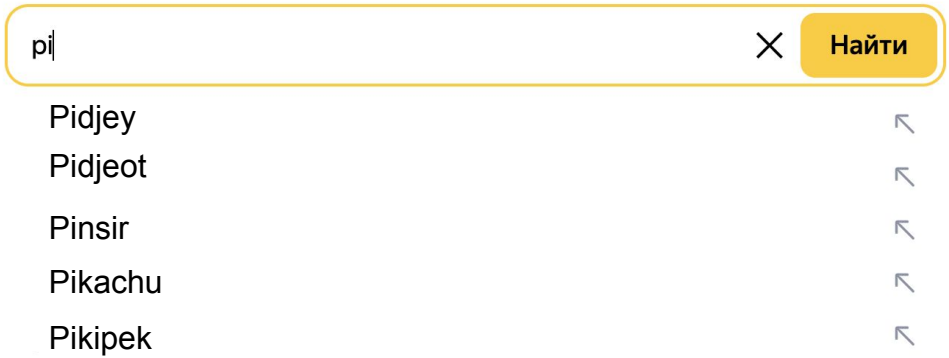
```python
raichu = Pokemon(
  name='Raichu',
  category='Mouse',
  height=78.74,
  weight=29.98,
  strengths=['Electric'],
  weaknesses=['Ground'],
)

pikachu = Pokemon(
  name='Pikachu',
  category='Mouse',
  height=40.64,
  weight=5.99,
  strengths=['Electric'],
  weaknesses=['Ground'],
  next_generation=raichu,
)
```



Перерыв #1

До 19:00

Каталог



Каталог:

- Коллекция покемонов
- Структура для поиска

- Найти по префиксу(префикс)
- Добавить покемона(покемон)
- Удалить покемона(покемон)

Каталог

```
1  # module catalog
2  from typing import List
3
4
5  # Коллекция покемонов
6  pokemons = {
7    'Pikachu': pikachu,
8    'Raichu': raichu,
9  }
10
11 # Структура для поиска
12 trie = {}
```

Каталог

```
1   def suggest(prefix: str) -> List[Pokemon]:
2     """
3     Поиск по имени топ-5 покемонов по префиксу
4     """
5     node = trie
6     for ch in prefix:
7       if ch not in node.nxt:
8         return []
9
10      node = node.nxt[ch]
11
12    return [pokemons[name] for name in node.top5]
```

Каталог

```
1   def append(pokemon: Pokemon) -> None:
2     """
3     Добавление покемона в каталог
4     """
5     pokemons[pokemon.name] = Pokemon
6     _trie_append(pokemon.name)
7
8
9
10  def _trie_append(name: str) -> None:
11    """
12    Добавление имени покемона в структуру для поиска
13    """
14    pass
```

Каталог

```
1  def delete(pokemon: Pokemon) -> None:
2      """
3      Удаление покемона из каталога
4      """
5      pokemons.pop(pokemon.name)
6      _trie_delete(pokemon.name)
7
8
9  def _trie_delete(name: str) -> None:
10     """
11     Удаление имени покемона из структуры для поиска
12     """
13     pass
```

Каталог

```
21      # Коллекция покемонов
22      pokemons = {...}
26
27      # Структура для поиска
28      trie = {}
29
30      def suggest(prefix: str) -> List[Pokemon]:...
42
43      def append(pokemon: Pokemon) -> None:...
49
50      def _trie_append(name: str) -> None:...
55
56      def delete(pokemon: Pokemon) -> None:...
62
63      def _trie_delete(name: str) -> None:...
68
```

Каталог

```
1   class Trie:
2     def __init__(self):
3       self.root = Node('')
4
5     def suggest(self, prefix: str) -> List[str]:
6       node = self.root
7       for ch in prefix:
8         if ch not in node.nxt:
9           return []
10
11        node = node.next[ch]
12
13      return node.top5
14
15    def append(self, word: str) -> None:
16      pass
17
18    def delete(self, word: str) -> None:
19      pass
```

Каталог

```
1  class Node:
2    def __init__(self, char: str, parent: Optional['Node'] = None, top5:
3  Optional[List[str]] = None):
4        self.char = char
5        self.parent = parent or []
6        self.next = {}
7        self.top5 = top5 or []
```

Каталог

```python
class Catalog:
    def __init__(self):
        self._trie = Trie()
        self.storage = {}

    def suggest(self, prefix: str) -> List[Pokemon]:
        names = self._trie.suggest(prefix)
        return [self.storage[name] for name in names]

    def append(self, pokemon: Pokemon) -> None:
        self.storage[pokemon.name] = pokemon
        self._trie.append(pokemon.name)

    def delete(self, pokemon: Pokemon) -> None:
        self.storage.pop(pokemon.name)
        self._trie.delete(pokemon.name)
```

Синтаксис

Пример: CountVectorizer

```
1   class CountVectorizer():
2       """Convert a collection of tex..."""
3       stop_words = ("the", "a", "and")
4
5       def __init__(self, lowercase=True):
6           self.lowercase = lowercase
7
8       def fit_transform(self, corpus):
9           """Learn the vocabulary di..."""
10          pass
```

Название класса

Докстринг

Атрибут класса

Конструктор

Атрибут

Метод

Экземпляр

Синтаксис: исследование

Можно получить докстринг, название класса или класс объекта

```
1  CountVectorizer.__doc__
2  Out: 'Convert a collection of tex...'
3
4  CountVectorizer.__name__
5  Out: 'CountVectorizer'
6
7  CountVectorizer().__class__
8  Out: __main__.CountVectorizer
```

Синтаксис: `__init__`

Конструктор `__init__`. Вызывается при создании объектов

```
5  def __init__(self, lowercase=True):
6      self.lowercase = lowercase
```

Синтаксис: `self`

- экземпляр класса
- принято называть именно `self`
- явно указывается первым аргументом в методах
- неявно передается при вызове методов

Не смотря на то, что синтаксис языка не запрещает использовать другое название вместо `self`, НЕ ДЕЛАЙТЕ этого

Синтаксис: `self`

Сравним идентификаторы объектов

```
1  class CountVectorizer():
2     def get_id(self):
3         return id(self)
4
5  vec = CountVectorizer()
6  assert vec.get_id() == id(vec)
7  print(vec.get_id())
8
9  Out: 4324201024
```

https://docs.python.org/3/library/functions.html#id - `id(object)`

Синтаксис: атрибуты

Атрибуты - для хранения значений

```
5       def __init__(self, lowercase=True):
6           self.lowercase = lowercase
```

Синтаксис: атрибуты экземпляра

- задаются посредством присвоения к self

```
self.lowercase = lowercase

# присвоение к экземпляру
vec.lowercase = True
```

- хранят значения принадлежащие только ему

```
vec1, vec2 = CountVectorizer(lowercase=False),
CountVectorizer(lowercase=False)
vec1.lowercase = True
print(vec1.lowercase, vec2.lowercase)

Out: (True, False)  # значение в vec2.lowercase так и осталось False
```

Синтаксис: атрибуты класса

- задаются посредством объявления в теле класса

```
1  class CountVectorizer():
2     stop_words = ('the', 'a', 'and')
3
4  CountVectorizer.stop_words = ('и', 'или') # прямое присвоение к классу
```

- хранят значения доступные всем экземплярам

```
1  vec1, vec2 = CountVectorizer(), CountVectorizer()
2
3  CountVectorizer.stop_words = ('и', 'или')
4  print(vec1.stop_words, vec2.stop_words)
5  Out: ('и', 'или') ('и', 'или')
```

Синтаксис: приватные атрибуты

- Отсутствуют модификаторы доступа
- Соглашение об именование приватных атрибутов: добавляем нижнее подчеркивание в начало названия

```
1  class CountVectorizer():
2     def __init__(self, lowercase=True):
3         self.lowercase = lowercase
4         self._vocabulary = {}
```

https://stackoverflow.com/a/52903693/2190638 - _ and __

Синтаксис: приватные атрибуты

- Для предотвращения случайного доступа к атрибуту используют два нижних подчеркивания
- Для доступа нужно указать название класса

```
1  class CountVectorizer():
2     def __init__(self, lowercase=True):
3         self.lowercase = lowercase
4         self.__vocabulary = {}
5
6  vec = CountVectorizer()
7  vec._CountVectorizer__vocabulary # _[class_name]__[attr_name]
8  Out: {}
```

Синтаксис: `obj.__dict__`

- атрибуты объекта доступны в виде словаря `__dict__`
- доступ можно получить также с помощью `vars`

```
1  vec = CountVectorizer()
2  vec.some_attr = 'some_attr_val'
3  vec.__dict__
4  Out: {'lowercase': True, 'some_attr': 'some_attr_val'}
5
6  vars(vec) is vec.__dict__
7  Out: True
```

Синтаксис: `obj.__dict__`

изменяя словарь `__dict__`, меняются атрибуты объекта

```
1  vec = CountVectorizer()
2  vec.some_attr
3  Out: AttributeError: 'CountVectorizer' object has no attribute 'some_attr'
4
5  vec.__dict__['some_attr'] = 'some_attr_val'
6  vec.some_attr
7  Out: 'some_attr_val'
```

Синтаксис: `obj.__slots__`

- фиксирует множество возможных атрибутов объекта
- `__dict__` после не доступен

```
1  class CountVectorizer():
2      __slots__ = ('lowercase',)
3
4  vec = CountVectorizer()
5  vec.some_attr = 'some_attr_val'
6  Out: AttributeError: 'CountVectorizer' object has no attribute 'some_attr'
7
8
9  vec.__dict__
10 Out: AttributeError: 'CountVectorizer' object has no attribute '__dict__'
```

Синтаксис: `obj.__slots__`

- объекты с `__slots__` занимают меньше памяти
- более быстрый доступ к атрибутам

```
$ python -m memory_profiler mem_prof.py

Line #    Mem usage    Increment   Line Contents
================================================
    13     48.1 MiB     48.1 MiB   @profile
    14                             def mem_usage():
    15    102.5 MiB      3.7 MiB       s = [Gs('s') for _ in range(int(1e6))]
    16    267.3 MiB      6.6 MiB       d = [Gd('d') for _ in range(int(1e6))]
```

Синтаксис: `cls.__dict__`

содержит атрибуты класса

```
1  class CountVectorizer():
2     """Convert a collection of tex..."""
3     stop_words = ("the", "a", "and")
4
5
6  CountVectorizer.__dict__
7
8  Out:
9  mappingproxy({'__module__': '__main__',
10               '__doc__': 'Convert a collection of tex...',
11               'stop_words': ('the', 'a', 'and'),
12               '__dict__': <attribute '__dict__' of 'CountVectorizer' objects>,
13               '__weakref__': <attribute '__weakref__' of 'CountVectorizer' objects>})
```

Синтаксис: `cls.__dict__`

`mappingproxy` защищает атрибуты класса от изменений

```
1  CountVectorizer.__dict__['some_cls_attr'] = 'some_cls_attr_val'
2  Out: TypeError: 'mappingproxy' object does not support item assignment
3
4
5  CountVectorizer.__dict__ = {}
6
7  Out:
   AttributeError: attribute '__dict__' of 'type' objects is not writable
```

http://bit.ly/2TqVWJJ - `types.MappingProxyType`

Синтаксис: методы

Методы - это функции, принадлежащие определенному классу

```
1  class CountVectorizer():
2      def __init__(self, lowercase=True):
3          self.lowercase = lowercase
4          self._vocabulary = {}
5
6      def get_feature_names(self):
7          return self._vocabulary.items()
8
9  CountVectorizer.get_feature_names
10
11 Out: <function __main__.CountVectorizer.get_feature_names(self)>
```

Синтаксис: связанные методы

Первый аргумент связанного метода зафиксирован и равен связанному объекту

```
1  vec = CountVectorizer()
2  vec.get_feature_names
3
4  Out:
5  <bound method CountVectorizer.get_feature_names of
   <__main__.CountVectorizer object at 0x10f8421d0>>
6
7  vec.get_feature_names.__self__ is vec
8  Out: True
```

Синтаксис: связанные методы

Для вызова метода передавать экземпляр не нужно

```
vec.get_feature_names()

Out: dict_items([])
```

Для вызова функции из класса нужно явно передавать экземпляр
Но так обычно делать не стоит

```
CountVectorizer.get_feature_names(vec)

Out: dict_items([])
```

Синтаксис: статические методы

- Не надо указывать `self` первым аргументом
- Вызывается одинаково у класса и у объекта

```
1  class CountVectorizer():
2     @staticmethod
3     def split(text: str) -> [str]:
4         return text.split()
5
6  CountVectorizer.split('Crock Pot Pasta')
7  Out: ['Crock', 'Pot', 'Pasta']
8
9  vec = CountVectorizer()
10 vec.split('Crock Pot Pasta')
11 Out: ['Crock', 'Pot', 'Pasta']
```

Синтаксис: методы класса

- Первым аргументом указывается ссылка на класс cls
- Используется как альтернативный конструктор

```
1  class CountVectorizer():
2      @classmethod
3      def from_vocabulary(cls, vocabulary):
4          vec = cls()
5          vec._vocabulary = vocabulary
6          return vec
7
8  vec = CountVectorizer.from_vocabulary({'Crock': 0, 'Pot': 1, 'Pasta': 2})
9  vec._vocabulary
10
11 Out: {'Crock': 0, 'Pot': 1, 'Pasta': 2}
```

Синтаксис: свойства

Позволяют определять атрибуты, вычисляющие значения во время обращения

```
1  class CountVectorizer():
2      @property
3      def feature_names(self):
4          return self._vocabulary.items()
5
6
7
8  vec = CountVectorizer()
9  vec.feature_names
10
11
12 Out: dict_items([])
```

Синтаксис: свойства

Можно изменить поведение, не меняя интерфейс

```
1  class CountVectorizer():
2     def __init__(self, lowercase=True):
3         self.lowercase = lowercase
4         self.vocabulary = {}
5
6  # обращаемся к атрибуту
7  CountVectorizer().vocabulary
```

```
1  class CountVectorizer():
2     def __init__(self, lowercase=True):
3         self.lowercase = lowercase
4         # решили сделать приватным
5         self._vocabulary = {}
6
7     @property
8     def vocabulary(self):
9         return self._vocabulary
10
11 # обращаемся к свойству, как будто
12 # ничего не менялось
13 CountVectorizer().vocabulary
```

Синтаксис: свойства

Добавить проверку при изменение

```
    @property
    def vocabulary(self):
        return self._vocabulary

    @vocabulary.setter
    def vocabulary(self, new_vocabulary):
        assert new_vocabulary, 'empty vocabulary'

        extra_words = new_vocabulary.keys() - self._vocabulary.keys()
        assert not extra_words, 'has extra words'

        self._vocabulary = new_vocabulary
```

Синтаксис: свойства

Добавить логику при удалении

```
    @vocabulary.deleter
    def vocabulary(self):
        self._vocabulary = {}

vec = CountVectorizer()
vec._vocabulary = {'Crock': 0, 'Pot': 1, 'Pasta': 2} # хак для примера
Vec.vocabulary
Out: {'Crock': 0, 'Pot': 1, 'Pasta': 2}

del vec.vocabulary
vec.vocabulary
Out: {}
```

Синтаксис: тело класса

Объявление атрибутов и методов не является специальным синтаксисом

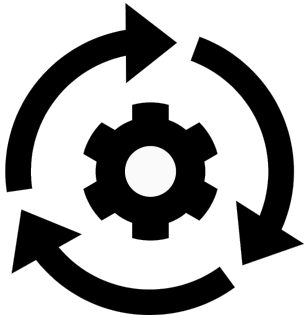
```
1  class StrangeCountVectorizer():
2      number1, number2 = 1, 2
3      for i in range(3):
4          number1 += i
5
6  StrangeCountVectorizer.__dict__
7
8  Out: mappingproxy({'number1': 4, 'number2': 2, 'i': 2})
```

http://bit.ly/2Tug5ym - statements in the body of a class

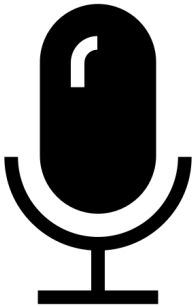
Практика #1

План

Получаем задание



Реализуем



Демонстрируем

Задание #1: Pokemon/атрибуты

Реализуйте класс Pokemon, имеющий атрибуты

- name
- category

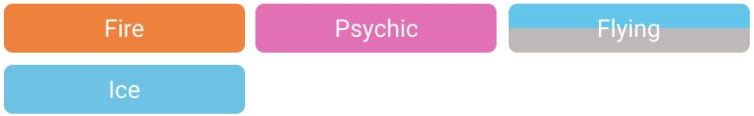
```
1  bulbasaur = Pokemon(name='Bulbasaur', category='seed')
2  bulbasaur.category
3
4  Out: 'seed'
```

Задание #2: Pokemon/приватные атрибуты

Добавьте в класс Pokemon

- приватный атрибут _weaknesses: Tuple[str]
- метод получения слабостей, который возвращает только первую

```
bulbasaur = Pokemon(
    name='Bulbasaur',
    category='seed',
    weaknesses=('fire', 'psychic', 'flying', 'ice')
)
bulbasaur.get_weaknesses()

Out: ('fire',)
```



Задание #3: Pokemon/свойства

Добавьте в класс Pokemon

- свойство weaknesses, которое возвращает только первую слабость

```
bulbasaur = Pokemon(
    name='Bulbasaur',
    category='seed',
    weaknesses=('fire', 'psychic', 'flying', 'ice')
)
bulbasaur.weaknesses

Out: ('fire',)
```

Спасибо за внимание!