# Самое необходимое о классах ②

Роман Афанаскин, 2022

# Правила

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

# План лекции

## ✅ Применение классов

- Варианты реализации
- Сложное поведение

## ✅ Синтаксис

- Атрибуты
- Методы
- Свойства

# План лекции

- Наследование
  - Поиск атрибутов и методов
  - Переопределение методов
  - Множественное наследование
- Мини практика
- Перерыв
- Миксины
  - Решаемая проблема
  - Особенности
- “Магические” методы
  - `__str__`, `__getattr__`
  - Контекстный менеджер
- Мини практика

# Наследование

# Наследование: синтаксис

Родительский класс указывается в скобках после названия класса

```
1  class BaseEstimator:
2      def to_str(self):
3          return f'{self.__class__.__name__}'
4
5
6  class CountVectorizer(BaseEstimator):
7      def __init__(self, lowercase=True):
8          self.lowercase = lowercase
9          self._vocabulary = {}
10
11 CountVectorizer().to_str()
12 Out: 'CountVectorizer'
```

# Наследование: поиск атрибутов и методов

- сначала в экземпляре
- потом в классе
- в родительских классах

```
1  class BaseEstimator:
2     in_base = 'in_base'
3
4  class CountVectorizer(BaseEstimator):
5     in_class = 'in_class'
6
7     def __init__(self):
8         self.in_object = 'in_object'
9
10 vec.in_object, vec.in_class, vec.in_base
11 Out: ('in_object', 'in_class', 'in_base')
```

# Наследование: переопределение методов

- Позволяет расширить поведение метода
- Позволяет заменить поведение метода

```
1  class BaseEstimator:
2      def to_str(self):
3          return f'{self.__class__.__name__}'
4
5  class CountVectorizer(BaseEstimator):
6      def to_str(self):
7          base_str = BaseEstimator.to_str(self)  # есть способ лучше
8          return f'[{base_str}]'
9
10 CountVectorizer().to_str()
12 Out: '[CountVectorizer]'
```

# Наследование: `super`

- Возвращает прокси-объект, передающий вызов методов в родительский класс
- Не нужно указывать явно родителя
- Не нужно передавать `self`

```
1  class CountVectorizer(BaseEstimator):
2     def to_str(self):
3         base_str = super().to_str()
4         return f'[{base_str}]'
5
6  CountVectorizer().to_str()
7  Out: '[CountVectorizer]'
```

# Наследование: множественное

- Можно указать более одного родительского класса

```
1  class BaseEstimator:
2     def to_str(self):
3         return f'{self.__class__.__name__}'
```

```
1  class _VectorizerMixin():
2     def get_stop_words(self):
3         return self._stop_words
```

```
1  class CountVectorizer(_VectorizerMixin, BaseEstimator):
2     def __init__(self):
3         self._stop_words = ('and', 'or', 'the')
4
5  vec = CountVectorizer()
6  vec.get_stop_words()
7  Out: ('and', 'or', 'the')
8
10 vec.to_str()
11 Out: 'CountVectorizer'
```

# Наследование: `mro`

- Родительские классы линеаризуются с помощью алгоритма C3
- Посмотреть линеаризованный список можно с помощью `cls.mro()`
- Сохраняет порядок родителей указанный в объявлении класса

```
1
2
class A:
    pass
```

```
1
2
class B(A):
    pass
```

```
1
2
class C:
    pass
```

```
1
2
class D(C, B):
    pass
```

```
1  D.mro()  # D.__mro__
2
3  Out:
4  [__main__.D, __main__.C, __main__.B, __main__.A, object]
```

http://bit.ly/2Scy7sE - The Python 2.3 Method Resolution Order

# Наследование: ещё раз о `super`

Делегирует вызов метода следующему классу в `mro`

```
1  class A:
2      def process(self):
3          print('A.process')
```

```
1  class B(A):
2      def process(self):
3          print('B.process')
4          super().process()
```

```
1  class C(A):
2      def process(self):
3          print('C.process')
4          super().process()
```

```
1  class D(C, B):
2      pass
```

```
1  D().process()  # [__main__.D, __main__.C, __main__.B, __main__.A, object]
2  Out: ?
```

# Наследование: ещё раз о `super`

Делегирует вызов метода следующему классу в `mro`

```python
class A:
    def process(self):
        print('A.process')
```

```python
class B(A):
    def process(self):
        print('B.process')
        super().process()
```

```python
class C(A):
    def process(self):
        print('C.process')
        super().process()
```

```python
class D(C, B):
    pass
```

```python
D().process()  # [__main__.D, __main__.C, __main__.B, __main__.A, object]
Out: C.process B.process A.process
```

# Наследование: предостережения

- Множественное наследование усложняет понимание кода
- Результат линеаризации не всегда тривиален, поэтому использовать сложные иерархии множественного наследования не рекомендуется

# Наследование: предостережения

Можно так отнаследоваться, что `mro` построить не получится

```
1  class A:
2     pass
```

```
1  class B(A):
2     pass
```

```
1  class C(A, B):
2     pass
3
4  Out:
5  TypeError: Cannot create a consistent method resolution
6  order (MRO) for bases A, B
```

# Наследование: предостережения

Если родительский класс определяет `__slots__`, а наследник нет, то используется `__dict__`

```
1  class WithSlots:
2      __slots__ = ('a', 'b')
3
4  class WithDict(WithSlots):
5      pass
6
7  class AlsoWithSlots(WithSlots):
8      __slots__ = ()
```

# Наследование: проверка принадлежности

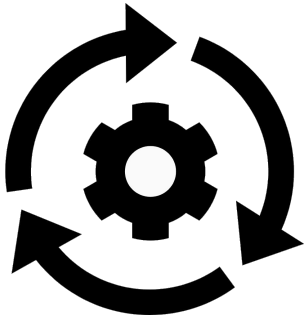
- `isinstance(obj, clsinfo)` - проверяет, что `obj` является экземпляром класса или кортежа классов
- `issubclass(cls, clsinfo)` - проверяет, что `cls` является потомком класса или кортежа классов

```
1  isinstance('obj', str)
2  Out: True
3
4  issubclass(KeyError, (LookupError, Exception))
5  Out: True
```

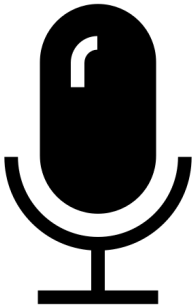
# Практика #1

# План

Получаем задание



Реализуем



Демонстрируем

# Задание #2: Pokemon/базовый класс

Реализуйте класс BasePokemon

- устанавливает атрибуты `name` и `category`
- имеет метод to_str, выводящий строку из атрибутов `name` и `category`

```
1  base_charmander = BasePokemon(name='Charmander', category='Lizard')
2  base_charmander.to_str()
3
4  Out: 'Charmander/Lizard'
```

# Задание #2: Pokemon/наследование

Отнаследуйтесь от класса BasePokemon в классе Pokemon, удалив ненужный код

```
charmander = Pokemon(
    name='Charmander',
    category='Lizard',
    weaknesses=(water, ground, rock)
)

charmander = Pokemon(name='Charmander', category='Lizard')
charmander.to_str()

Out: 'Charmander/Lizard'
```

# Перерыв #1

До 19:05

# Миксины

# Миксины: решаемая проблема

Допустим, реализуем класс `MinMaxScaler` для нормализации значений

```
1  class MinMaxScaler(BaseEstimator):
2     """Transform features by scaling each feature to a given range."""
3
4     def fit_transform(self, features):
5         return self.fit(features).transform(features)
```

# Миксины: решаемая проблема

И реализуем преобразования в tf-idf в классе `TfidfTransformer`

```
1  class TfidfTransformer(BaseEstimator):
2      """Transform a count matrix to a normalized tf-idf representation"""
3
4      def fit_transform(self, features):
5          return self.fit(features).transform(features)
```

# Что можно улучшить?

```
class MinMaxScaler(BaseEstimator):
    """Transform features by scaling each ..."""

    def fit_transform(self, features):
        return self.fit(features).transform(features)
```

```
class TfidfTransformer(BaseEstimator):
    """Transform a count matrix to a ..."""

    def fit_transform(self, features):
        return self.fit(features).transform(features)
```

# Миксины: решаемая проблема

Вынесем метод `fit_transform` в базовый класс

И теперь нет дублирования в классах `MinMaxScaler` и `TfidfTransformer`

```
1  class BaseEstimator:
2     """Base class for all estimators"""
3
4     def fit_transform(self, features):
5         return self.fit(features).transform(features)
```

# Миксины: решаемая проблема

Но у нас есть класс `KMeans`, реализующий кластеризацию методом K-средних и являющийся наследником `BaseEstimator`

Теперь он тоже имеет ненужный ему метод `fit_transform` ¯_(ツ)_/¯

```
1  class KMeans(BaseEstimator):
2      """K-Means clustering."""
3
4      def fit_predict(self, features):
5          self.fit(features)
           return self.labels_
```

Как можно избавиться от ненужного метода `fit_transform`?

# Миксины: решаемая проблема

Вынесем из `BaseEstimator` логику метода `fit_transform` в класс `TransformerMixin`

```
1  class TransformerMixin:
2     """Mixin class for all transformers."""
3
4     def fit_transform(self, features):
5         return self.fit(features).transform(features)
```

# Миксины: решаемая проблема

И унаследуем его в `MinMaxScaler` и `TfidfTransformer`

```
1  class MinMaxScaler(TransformerMixin, BaseEstimator):
2     """Transform features by scaling each feature to a given range."""
3
4
5  class TfidfTransformer(TransformerMixin, BaseEstimator):
6     """Transform a count matrix to a normalized tf-idf representation"""
```

# Миксины: особенности

- Обычный класс
- Самостоятельно не может использоваться
- Принято, что название заканчивалось на `Mixin`
- Частично изменяет поведение класса
- Содержат дополнительную функциональность, которую можно “подмешать” к текущему классу
- Обычно указывается самым левым родителем

# Почему указывается самым левым родителем?

# Миксины: особенности

По mro методы и атрибуты будут искать в этом классе первыми

И в них можно получить финальный результат от всех остальных родителей

```
1  class PercentageMixin:
2      scale = 100
3
4      def fit_transform(self, features):
5          transformed = super().fit_transform(features)
6          return transformed * self.scale
```

# “Магические” методы

# “Магические” методы: определение

- Начинаются и заканчиваются с двух подчеркиваний - `dunder methods`
- Неявно вызываются во время выполнения
- С их помощью реализованы различные протоколы/интерфейсы

Протокол - набор методов объекта, благодаря которым, он может выполнять определенную роль в системе

# “Магические” методы: `__repr__`

- Определяет строковое представление удобное для изучения объекта
- Используется отладчиками и интерактивной средой REPL

```
1  class Guide:
2      def __repr__(self):
3          return f'id: {id(self)}; class: {self.__class__.__name__}'
4
5  Guide()
6
7  Out:
8  'id: 4550051936; class: Guide'
```

# “Магические” методы: `__repr__`

Отладчик - программа, позволяющая пошаговое выполнение, с остановками на некоторых строках исходного кода

REPL - read-eval-print loop

# “Магические” методы: __str__

Определяет строковое представление, для показа конечным пользователям

```
1  class Guide:
2     text = '''Автомобиль продается первым официальным дилером Порше.
3  Спорткар Центр Порше самый большой дилерский центр в Европе.'''
4
5     def __str__(self):
6         compact_text = textwrap.shorten(self.text, 40, placeholder='...')
7         return f'Guide with text: {compact_text}'
8
   Out:
   'Guide with text: Автомобиль продается первым...'
```

# “Магические” методы: __format__

Определяет форматированное строковое представление

```
1  class Guide:
2      text = 'Автомобиль Порше'
3
4      def __format__(self, format_spec):
5          return self.text.__format__(format_spec)
6
7  f'{Guide():>30}'
8
9  Out:
10 '                Автомобиль Порше'
```

# “Магические” методы: сравнение

Для поддержания всех операторов сравнения можно реализовать 3 из 6 “магических” методов. Недостающие будут вызваны у `other`

```
1  obj.__eq__(other)  # obj == other
2  obj.__ne__(other)  # obj != other
3  obj.__lt__(other)  # obj < other
4  obj.__le__(other)  # obj <= other
5  obj.__gt__(other)  # obj > other
6  obj.__ge__(other)  # obj >= other
```

# “Магические” методы: сравнение

Можно воспользоваться декоратором `functools.total_ordering`, который добавит недостающие методы

```
1  @total_ordering
2  class Car:
3      def __init__(self, max_speed):
4          self.max_speed = max_speed
5
6      def __eq__(self, other):
7          return self.max_speed == other.max_speed
8
9      def __lt__(self, other):
10         return self.max_speed < other.max_speed
```

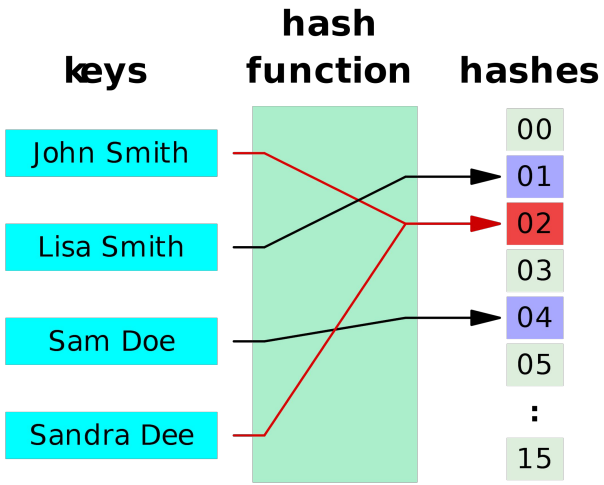
# “Магические” методы: __hash__

По умолчанию одинаковые значения хеш-функции только у физически одинаковых объектов

```
1  class Tokenizer:
2     def __init__(self, text):
3         self.text = text
4
5  {Tokenizer('text'), Tokenizer('text')}  # получим разные объекты
6  Out:
7  {<__main__.Tokenizer at 0x107270e10>, <__main__.Tokenizer at 0x107270ef0>}
```

# “Магические” методы: __hash__

Хеш-функция - **соответствие** между элементами двух **множеств**, первое из которых может быть **произвольной мощности**, а второе **фиксированной** и такое соответствие для элементов первого множества **одно**



# “Магические” методы: __hash__

Используется для вычисления значения хеш-функции

```
1  class Tokenizer:
2      def __init__(self, text):
3          self.text = text
4
5      def __eq__(self, other):
6          return self.text == other.text
7
8      def __hash__(self):
9          return hash(self.text)
10
11 {Tokenizer('text'), Tokenizer('text')}
12 Out: {<__main__.Tokenizer at 0x1075dd5c0>}
```

# “Магические” методы: __hash__

- При реализации метода `__hash__` нужно реализовывать метод `__eq__`, для избежания неожиданного поведения в сетах и словарях
- Если объекты равны, то и значения  хеш-функции должны быть равны
  `x == y => hash(x) == hash(y)`
- При реализации только `__eq__`, объект становиться нехэшируемым

# “Магические” методы: `__getattr__`

Вызывается при обращение к несуществующему атрибуту

```
1  class Lemmatizer:
2      LANGS = {'en', 'ru'}
3
4      def __getattr__(self, item):
5          print(f'access to {item}')
6
7  Lemmatizer().LANGS
8  Lemmatizer().unknown_attr
9
10 Out:
11 access to unknown_attr
```

# “Магические” методы: __getattribute__

Вызывается при обращение ко всем атрибутам и методам

```
1  class Lemmatizer:
2      LANGS = {'en', 'ru'}
3
4      def __getattribute__(self, item):
5          print(f'access to {item}')
6
7      def lemmatize(self, token: str) -> str:
8          return token
9
10 Lemmatizer().LANGS, Lemmatizer().unknown_attr, Lemmatizer().lemmatize
11
12 Out: 'access to LANGS'  'access to unknown_attr' 'access to lemmatize'
```

# “Магические” методы: `__setattr__`

Вызывается при изменение всех атрибутов и методов

```
1  class Lemmatizer:
2      LANGS = {'en', 'ru'}
3
4      def __setattr__(self, key, value):
5          print(f'set {value} to {key}')
6
7  l = Lemmatizer()
8  l.LANGS = {'en', 'ru', 'es'}
9
10 Out:
11 set {'ru', 'es', 'en'} to LANGS
```

# “Магические” методы: `__setattr__`

Нужно обращаться к `__dict__` для избежания зацикливания

```
1  class Lemmatizer:
2      def __getattr__(self, item):
3          return self.__dict__.get(item) # так не надо getattr(self, item)
4
5      def __setattr__(self, key, value):
6          self.__dict__[key] = value  # так не надо setattr(self, key, value)
7
```

# “Магические” методы: __setattr__

При переопределение вместе с __getattribute__ нужно обращаться к __getattribute__ родительского класса для избежания зацикливания

```
1  class Lemmatizer:
2      def __getattribute__(self, item):
3          try:
4              val = super().__getattribute__(item)
5          except AttributeError:
6              val = None
7          return val
8
9      def __setattr__(self, key, value):
10         self.__dict__[key] = value
11
```

# “Магические” методы: __delattr__

Вызывается при удаление атрибута

```
1  class Lemmatizer:
2     def __init__(self, lang):
3         self.lang = lang
4
5     def __delattr__(self, item):
6         del self.__dict__[item]
7
8  l = Lemmatizer('en')
9  del l.lang
```

# “Магические” методы: `__enter__`

Контекстный менеджер - конструкция языка, позволяющая оборачивать код, с целью переиспользования паттерна **`try…except…finally`**

```
1   fd = open('open.py', 'r')
2   hit_except = False
3
4   try:
5       print(fd.read())
6   except:
7       print('except')
8       hit_except = True
9       fd.close()
10  finally:
11      if not hit_except:
12          fd.close()
13
14  print(f'closed: {fd.closed}')
```

# “Магические” методы: `__enter__`

Работать с файлами лучше используя контекстный менеджер

```
1  with open('open.py', 'r') as fd:
2      print(fd.read())
```

# “Магические” методы: __enter__

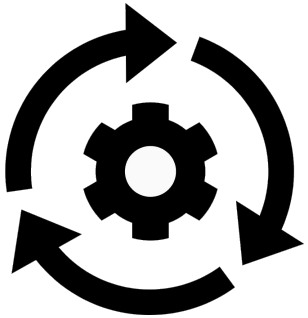
Для написания своего менеджера, достаточно реализовать методы __enter__ и __exit__

```
1  class tag:
2      def __init__(self, name):
3          self.name = name
4
5      def __enter__(self):
6          print(f'<{self.name}>')
7
8      def __exit__(self, exc_type, exc_val, exc_tb):
9          print(f'</{self.name}>')
```

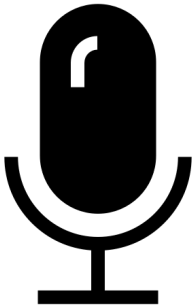
# Практика #2

# План

Получаем задание



Реализуем



Демонстрируем

# Задание #1: Pokemon/макический метод

Дан класс Pokemon

```
class Pokemon:
    def __init__(self, name: str, poketype: str):
        self.name = name
        self.poketype = poketype

    def to_str(self):
        return f'{self.name}/{self.poketype}'
```

Замените метод to_str

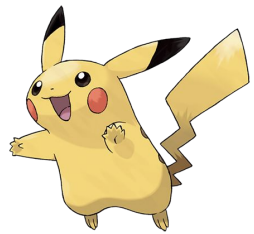
```
bulbasaur = Pokemon(name='Bulbasaur', poketype='grass')
print(bulbasaur)
Out: 'Bulbasaur/grass'
```

# Задание #2: Pokemon/миксин

Напишите миксин `EmojiMixin`, который модифицирует метод `__str__`
Заменяет категорию покемона на эмоджи

- **`grass`** => 🌿
- **`fire`** => 🔥
- **`water`** => 🌊
- **`electric`** => ⚡

```
1  pikachu = Pokemon(name='Pikachu', category='electric')
2  print(pikachu)
3  Out: 'Pikachu/⚡'
```



# Результаты

Что было:

- Наследование
  - Поиск атрибутов и методов
  - Переопределение методов
  - Множественное наследование
- Миксины
  - Решаемая проблема
  - Особенности
- “Магические” методы
  - `__str__`, `__getattr__`
  - Контекстный менеджер

# Результаты

Что будет:

- Практика по “магическим” методам

Спасибо за внимание!