Python

Занятие #8. Классы. Часть 2.

💬 О чем это занятие:

- ❏ Модификация поведения класса:
- ❏ “Магические” методы
- ❏ Абстрактные классы

💫 Теги:

Mixin, __repr__, __str__, __format__, __call__, __hash__, __getattr__, __getattribute__, __setattr__, __delattr__, __enter__, явные протоколы, абстрактный класс

Краткий обзор темы

- Миксины – это классы, реализованные на основе других, в название которых добавляется слово Mixin

- “Магические” методы неявно вызываются во время выполнения кода, имеют в названии двойное подчеркивание. Наиболее часто встречающиеся и полезные методы:
  __repr__ определяет строковое представление аргумента, используется разработчиками
  __call__ реализует вызов экземпляра класса как функцию,
  __hash__ используется для вычисления хэш-функции,
  __getattr__ вызывается при обращении к несуществующему атрибуту,
  __setattr__ вызывается при изменении всех атрибутов и методов класса,
  __delattr__ вызывается при удалении атрибута,
  __enter__ необходим для реализации контекстного менеджера
  и другие.

- Абстрактные классы: основная концепция, протоколы, особенности наследования от абстрактных классов, готовые абстрактные классы из коллекций Python и готовые реализации наследников.

Модификация поведения класса

Миксины

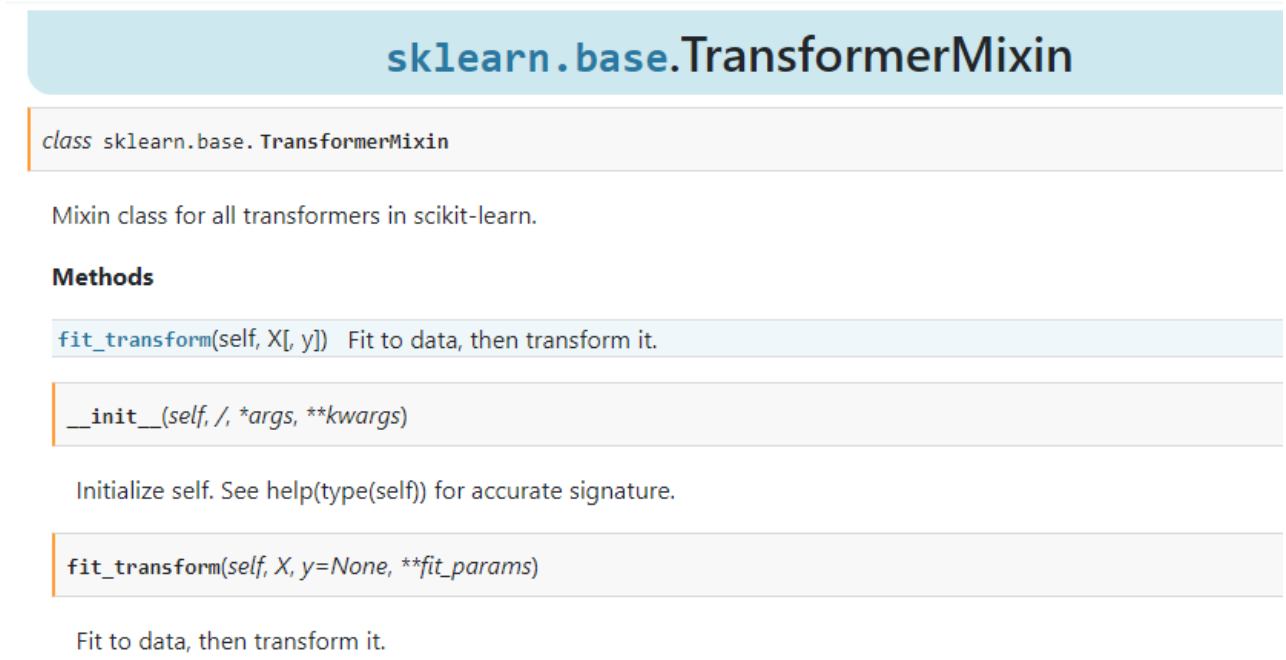
- ❏ Миксин – это обычный класс, который частично изменяет поведение текущего класса, то есть добавляется некоторая функциональность к другому классу.
- ❏ Название такого класса принято заканчивать словом Mixin
- ❏ Указывается самым левым родителем

💻**Пример:**
к классу, который отвечает за возвращение некоторого JSON[1], необходимо добавить авторизацию. Если добавить mixin, появится проверка авторизации.

```
1  # благодаря LoginRequiredMixin проверяем, что пользователь залогинен
2  class GuidesListView(LoginRequiredMixin, ListView):
3     pass
```

В качестве аналитика миксины можно встретить в scikit-learn[2]:

[1] JSON (JavaScript Object Notation) - простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером.
[2] Библиотека Scikit-learn — это библиотека Python для решения задач классического машинного обучения, основана на NumPy и SciPy.



sklearn.base.TransformerMixin

class sklearn.base. **TransformerMixin**

Mixin class for all transformers in scikit-learn.

Methods

`fit_transform`(self, X[, y]) Fit to data, then transform it.

__init__(*self, /, *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

fit_transform(*self, X, y=None, **fit_params*)

Fit to data, then transform it.

💬 миксин добавляет в класс возможность трансформировать и сразу обучить модель

👉 миксин не используется как отдельный класс, он “подмешивается” к другим классам

“Магические” методы

- Магические методы, как их называют в Python, начинаются и заканчиваются с двух подчеркиваний
- Неявно вызываются во время выполнения

  Это означает, что “вручную” пользователь их вызывает редко, они вызываются интерпретатором Python во время выполнения кода

- С их помощью реализованы различные протоколы/интерфейсы

👉 Протокол – набор методов объекта, благодаря которым он может выполнять определенную роль в системе.

Пусть имеется некоторая функциональность, для которой описаны методы, в этом случае ее можно назвать протоколом.
Пример: говорят, что если объект может реализовывать метод `str`, то он соответствует протоколу строки.

Метод __repr__

- Определяет строковое представление, удобное для изучения объекта (для разработчика)

❏ Используется отладчиками[3] и интерактивной средой REPL[4]

👉 метод `__repr__` вызывается, когда нужно представить объект в виде строки для его изучения или использования в командной строке

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

Здесь метод `__repr__` используется для вывода id и названия класса

💬 id возвращает адрес объекта класса в памяти

Метод __str__

❏ Определяет строковое представление для показа конечным пользователям (отличие от `__repr__`):

[3] Отладчик – программа, позволяющая пошаговое выполнение, с остановками на некоторых строках исходного кода
[4] REPL – read-eval-print loop

```
class Guide:
   text = '''Автомобиль продается первым официальным дилером Порше.
Спорткар Центр Порше самый большой дилерский центр в Европе.'''

   def __str__(self):
       compact_text = textwrap.shorten(self.text, 40, placeholder='...')
       return f'Guide with text: {compact_text}'

Out:
'Guide with text: Автомобиль продается первым...'
```

Здесь происходит сокращение содержимого атрибута text до 40 символов

Метод __format__

```
class Guide:
   text = 'Автомобиль Порше'

   def __format__(self, format_spec):
       return self.text.__format__(format_spec)

f'{Guide():>30}'

Out:
'              Автомобиль Порше'
```

Здесь поле text выводится в отформатированном виде

Сравнение

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

```
obj.__eq__(other)  # obj == other
obj.__ne__(other)  # obj != other
obj.__lt__(other)  # obj < other
obj.__le__(other)  # obj <= other
obj.__gt__(other)  # obj > other
obj.__ge__(other)  # obj >= other
```

👉 методы сравнения можно переопределять

Пример переопределения методов сравнения в классе:

```
@total_ordering
class Car:
    def __init__(self, max_speed):
        self.max_speed = max_speed

    def __eq__(self, other):
        return self.max_speed == other.max_speed

    def __lt__(self, other):
        return self.max_speed < other.max_speed
```

Код с `__init__` работает быстрее, чем с реализацией 3 методов, но медленнее, с реализацией всех 6

Метод __call__

Экземпляр класса может быть вызван как функция (с пустыми скобками):

```
class Lemmatizer:
    def __init__(self, lang):
        self.lang = lang

    def __call__(self):
        print("Lemmatizer.__call__")

Lemmatizer('en')()

Out:Lemmatizer.__call__
```

Если напишем `@Lemmatizer`, вызовется метод `__call__`

Метод __hash__

👉 Хеш-функция – это соответствие между элементами двух множеств, первое из которых может быть произвольной мощности, а второе фиксированной. Такое соответствие для элементов первого множества может быть представлено единственным образом.

альтернативное определение (из которого лучше понятно):
– это функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом или «ключом». Результат преобразования (выходные данные) называется «хешем».

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

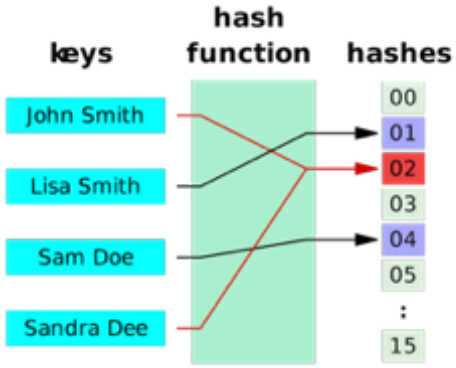
```
class Tokenizer:
   def __init__(self, text):
       self.text = text

{Tokenizer('text'), Tokenizer('text')}  # получим разные
объекты

Out:
{<__main__.Tokenizer at 0x107270e10>, <__main__.Tokenizer at
0x107270ef0>}
```

Здесь показано, что два объекта класса, инициализированные одинаковым параметром text, на самом деле не являются физически одинаковыми.

👉 Метод `__hash__` используется для вычисления значения хеш-функции

👉 значения хеш-функции объектов – это их id, он не меняется на протяжении жизни объекта.

💬 Python вызывает метод `__hash__` у объектов, когда пользователь создаёт множество `set` или словарь `dict`



Здесь два ключа имеют один и тот же хэш (образовалась коллизия). Чтобы её разрешить, описывают метод `__eq__`, в котором сравнивают значения:

```
class Tokenizer:
   def __init__(self, text):
       self.text = text

   def __eq__(self, other):
       return self.text == other.text

   def __hash__(self):
       return hash(self.text)

{Tokenizer('text'), Tokenizer('text')}

Out: {<__main__.Tokenizer at 0x1075dd5c0>}
```

Объекты с одинаковым полем text возвращают одинаковые хэш-значения, и в этом случае из двух одинаковых объектов остается один.

- При реализации метода `__hash__` необходимо реализовывать метод `__eq__` для избежания неожиданного поведения класса
- Если объекты равны, то и значения хеш-функции должны быть равны: x == y => hash(x) == hash(y)
- При реализации только `__eq__` объект становится нехэшируемым, его нельзя добавлять в словарь

Метод __getattr__

Метод вызывается при обращении к несуществующему атрибуту:

```
class Lemmatizer:
    LANGS = {'en', 'ru'}

    def __getattr__(self, item):
        print(f'access to {item}')

Lemmatizer().LANGS
Lemmatizer().unknown_attr

Out:access to unknown_attr
```

Метод __getattribute__

Метод вызывается при обращении ко всем атрибутам и методам класса:

```
class Lemmatizer:
    LANGS = {'en', 'ru'}

    def __getattribute__(self, item):
        print(f'access to {item}')

    def lemmatize(self, token: str) -> str:
        return token

Lemmatizer().LANGS, Lemmatizer().unknown_attr, Lemmatizer().lemmatize
```

```
Out: 'access to LANGS'  'access to unknown_attr' 'access to lemmatize'
```

Метод __setattr__

Метод вызывается при изменении всех атрибутов и методов класса:

```
class Lemmatizer:
   LANGS = {'en', 'ru'}

   def __setattr__(self, key, value):
       print(f'set {value} to {key}')

l = Lemmatizer()
l.LANGS = {'en', 'ru', 'es'}

Out:
set {'ru', 'es', 'en'} to LANGS
```

Чтобы добавить новый атрибут, через `__setattr__` можно обратиться к `__dict__` и добавить ключ:

```
class Lemmatizer:
   def __getattr__(self, item):
       return self.__dict__.get(item)

   def __setattr__(self, key, value):
       self.__dict__[key] = value
```

При переопределении вместе с `__getattribute__` нужно обращаться к `__getattribute__` родительского класса для предотвращения зацикливания:

```
class Lemmatizer:
   def __getattribute__(self, item):
       try:
           val = super().__getattribute__(item)
       except AttributeError:
           val = None
       return val

   def __setattr__(self, key, value):
       __dict__ = super().__getattribute__('__dict__')
       __dict__[key] = value
```

Метод __delattr__

Вызывается при удалении атрибута:

```
class Lemmatizer:
   def __init__(self, lang):
       self.lang = lang

   def __delattr__(self, item):
       del self.__dict__[item]

l = Lemmatizer('en')
del l.lang
```

Метод __enter__

Контекстный менеджер – это конструкция языка,позволяющая использовать паттерн `try...except...finally` с гарантией корректного завершения работы кода

```
fd = open('open.py', 'r')
hit_except = False

try:
   print(fd.read())
except:
   print('except')
   hit_except = True
   fd.close()
finally:
   if not hit_except:
       fd.close()

print(f'closed: {fd.closed}')
```

При работе с файлами рекомендуется использовать контекстный менеджер:

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

Для написания своего менеджера достаточно реализовать методы `__enter__` и `__exit__`:

```
class tag:
   def __init__(self, name):
       self.name = name

   def __enter__(self):
       print(f'<{self.name}>')

   def __exit__(self, exc_type, exc_val, exc_tb):
       print(f'</{self.name}>')

with tag(‘div’) as tg:
      print(‘Hello’)

Out:
<div>
Hello
</div>
```

Здесь tag(‘div’) вызывает `__init__`, конструкция `with` вызывает метод `__enter__`, а после команд внутри `with` вызывается метод `__exit__`

Название класса начинается с прописной буквы, т.к. класс реализует протокол контекстного менеджера.

Абстрактные базовые классы

Протокол

- набор методов объекта, благодаря которым он может выполнять определенную роль в системе
- определен путем документирования или соглашения
- можно реализовывать частично

```
1  class Sequence:
2      """Протокол последовательности"""
3      def __getitem__(self, key):
4          """Доступ к элементы по целочисленному индексу"""
5
6      def __len__(self):
7          """Возвращает длину последовательности"""
```

💻 **Пример:**
Реализуем частично протокол последовательности

```
class PartSeq:
   def __getitem__(self, index):
       return range(4)[index]

seq = PartSeq()
seq[3]
Out: 3
for i in seq:  # нет метода __iter__
   print(i)
Out: 0 1 2 3
3 in seq  # нет метода __contains__
Out: True
```

Здесь возможно итерироваться по объекту, хотя метод __iter__ не реализован, и использовать конструкцию in, хотя не реализован метод __contains__

Без исключения IndexError получаем бесконечный цикл for, т.е. класс бесконечной последовательности:

```
class InfSeq:
   def __getitem__(self, index):
      return 'inf'

seq = InfSeq()
for i in seq:
   print(i)
Out:
inf
inf
inf...
```

Явные протоколы

Явно зафиксировать протокол можно с помощью модуля Python, который называется ABC

```
# часть ABC Sequence из http://bit.ly/2GGN9jY
class Sequence(Reversible, Collection):
    """All the operations on a read-only sequence.
    Concrete subclasses must override __new__ or __init__,
    __getitem__, and __len__.
    """
    __slots__ = ()
    @abstractmethod
    def __getitem__(self, index):
        raise IndexError
```

Здесь говорится о том, что все классы, которые реализует протокол последовательности, должны реализовывать метод __getitem__, иначе Python выбросит исключение.

👉 Создание явного протокола: фиксация требований к новым классам, которые будут реализовывать протокол.

Абстрактный класс — это базовый класс, который не предполагает создания экземпляров. Абстрактный класс может содержать абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков.

❏ Абстрактный класс наследуется от ABC:

записывается так: abc.ABC - теперь класс abc абстрактный

- ❏ Абстрактные методы выделяются с помощью декоратора @abstractmethod
- ❏ Модуль collections.abc содержит большое количество готовых абстрактных классов

🔎 https://docs.python.org/3/library/collections.html - ABC for Containers

Абстрактный метод класса

Для задания абстрактного метода класса используют комбинацию @abstractmethod и @classmethod

```
1  class MyABC:
2     @classmethod
3     @abc.abstractmethod
4     def class_func(cls):
5         pass
```

Если бы класс MyABC был простым классом, декоратор @abc.abstractmethod был бы не нужен.

Абстрактное свойство

Для задания абстрактного свойства используют комбинацию @abstractmethod и @property

```python
class MyABC:
    @property
    @abc.abstractmethod
    def atr(cls):
        pass

    @atr.setter
    @abc.abstractmethod
    def atr(self, val):
        pass
```

Проверка на соответствие

- Если класс не реализовал все абстрактные методы, выдается ошибка при создании объекта
- Не допускается частичная реализация

```python
from collections.abc import Sequence

class PartSeq(Sequence):
    def __getitem__(self, index):
        return range(4)[index]

seq = PartSeq()
Out:

```

```
TypeError: Can't instantiate abstract class PartSeq with
abstract methods __len__
```

В абстрактном классе Sequence описаны два метода, поэтому в наследнике PartSeq должны быть реализованы оба метода, частичная реализация уже не допускается.

isinstance

Для проверки принадлежности ABC можно воспользоваться функцией isinstance. Экземпляр потомка ABC ожидаемо проходит проверку:

```
class PartSeq(Sequence):
    def __getitem__(self, index):
        return range(4)[index]

    def __len__(self):
        return 4

isinstance(PartSeq(), Sequence)
Out: True
```

Класс, реализующий все методы, но не являющийся потомком ABC, проверку не пройдет:

```
class PartSeq:
   def __getitem__(self, index):
       return range(4)[index]

   def __len__(self):
       return 4

isinstance(PartSeq(), Sequence)
Out: False
```

Поиск реализаций

В Python есть возможность получить классы, которые унаследованы от ABC или зарегистрированы как виртуальные классы:

```
import abc

class MyABC(abc.ABC): pass

class ImplSub(MyABC): pass

@MyABC.register
class ImplReg: pass

MyABC.__subclasses__() + list(MyABC._abc_registry)
Out: [__main__.ImplSub, __main__.ImplReg]
```

Встроенные коллекции

Создадим подкласс dict и переопределим метод __setitem__:

```
class MyDict(dict):
   def __init__(self, *args, **kwargs):
       super().__init__(*args, **kwargs)
       self.history = []

   def __setitem__(self, key, value):
       self.history.append(key)
       super().__setitem__(key, value)

my_dict = MyDict()
my_dict.setdefault("key01", "val01")
my_dict.history
Out: []
```

История пуста из-за того, что dict - это структура языка C, которая не вызывает перегруженный метод __setitem__

Для избежания такой проблемы можно создать класс на основе MutableMapping:

```python
class MyDict(MutableMapping):
    def __init__(self, *args, **kwargs):
        self.data = dict()
        self.history = []

    def __setitem__(self, key, value):
        self.history.append(key)
        self.data[key] = value

    # +4 метода __getitem__ __delitem__ __iter__ __len__
```

В этом случае Python потребует реализацию всех шести методов, описанных в абстрактном классе MutableMapping.

В collections существует готовая реализация класса UserDict с реализованными методами, которая не связана с dict. Этот класс:

- ❏ имитирует словарь
- ❏ производный от MutableMapping
- ❏ хранит данные в атрибуте data

Для этого класса можно описывать только те методы, которые нужны:

```python
from collections import UserDict

class MyDict(UserDict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.history = []

    def __setitem__(self, key, value):
        self.history.append(key)
        super().__setitem__(key, value)
```