PYTHON

Лекция 14

ПЛАН ЛЕКЦИИ

- Pandas
- Продвинутое ООП

PANDAS

- Ключевые понятия: DataFrame и Series
- DataFrame: двумерная таблица
- Series: примерно "столбец"
- Никаких произвольных размерностей

PANDAS

- Основной смысл: обработка табличных данных
- Поискать, попреобразовывать, очистить
- Хорошо стыкуется с CSV
- Идейно близок к Excel, в чем-то к SQL

PANDAS

- Столбцы именованы и типизированы
- Типы победнее, чем в numpy
- Уклон в бизнес-типы: время, даты, категории
- Меньше про числа

СПОСОБЫ СОЗДАНИЯ

- Словарь столбцов
- Список строк как словарей ячеек
- Двумерный ndarray с колонками
- Из файлов

ПРИМЕР

```
d = {'x': [1, 2, 3], 'y': np.array([2, 4, 8]), 'z': 100}
pd.DataFrame(d)
"""
   x  y    z
0  1  2  100
1  2  4  100
2  3  8  100
"""
```

ПРИМЕР

```
d2 = pd.DataFrame(d, index=[100, 200, 300],
                  columns=['z', 'y', 'x'])
"""
       z  y  x
100  100  2  1
200  100  4  2
300  100  8  3
"""
```

ПРИМЕР

```
d2.index
# Index([100, 200, 300], dtype='int64')
d2.loc[150:350]
"""
        z  y  x
200   100  4  2
300   100  8  3
"""
type(d2.loc[150:350])
# <class 'pandas.core.frame.DataFrame'>
type(d2.loc[150:250])
# >class 'pandas.core.frame.DataFrame'>
# ................
```

ПРИМЕР

```
# ........................
d2.loc[200]
"""
z    100
y      4
x      2
Name: 200, dtype: int64
"""

type(d2.loc[150:250])
# <class 'pandas.core.frame.DataFrame'>
```

ПРИМЕР

```
lst = [{'x': 1, 'y': 2, 'z': 100},
       {'x': 2, 'y': 4 },
       {'x': 3, 'y': 8, 't': 123}]
pd.DataFrame(lst)
"""
   x  y      z      t
0  1  2  100.0    NaN
1  2  4    NaN    NaN
2  3  8    NaN  123.0
"""
```

ПРИМЕР

```
arr = np.array([[1, 2, 100],
                [2, 4, 100],
                [3, 8, 100]])
df = pd.DataFrame(arr, columns=['x', 'y', 'z'])
"""
   x  y    z
0  1  2  100
1  2  4  100
2  3  8  100
"""
```

ПРИМЕР

```
arr[0, 0] = 5
df
"""
   x  y    z
0  5  2  100
1  2  4  100
2  3  8  100
"""
```

SERIES

- Одномерный набор значений
- Возможно, с индексом
- Простейший пример: колонка в DataFrame
- Также: df.dtypes, df.columns, df.index

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ DF

- Квадратные скобки сразу за DF - обращение по колонке
- Только по имени
- Если имя соотвествует правилам идентификаторов - можно как к свойству
- Второе измерение через запятую - нельзя

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ DF

- Можно указать список в колонке - будет новый DF
- Но данные колонок - те же
- Если указали одну колонку не списком - получим Series
- И в нем можно обращаться по индексу в квадратных скобках
- Если списком - все повторяется

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ DF

- Если хотим начать со строк - нужны вспомогательные методы
- С причудливыми правилами
- loc - оборащение в терминах индекса
- iloc - в терминах чисел

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ DF

- Это такие view-объекты
- iloc однозначно поддерживает все, что привычно для обычных индексов
- loc - только там, где есть разумная семантика
- Оба допускают запятую и индекс по колонкам за ней
- icol - только числовой, col - допускает оба
- at, iat - аналоги, только без диапазонов

АБСТРАКТНЫЕ МЕТОДЫ И АБСТРАКТНЫЕ КЛАССЫ

- Мотивация в контексте Python: хочется как-то структурировать duck typing
- Хотим написать функцию, работающую со списками - используя обращение по индексу

АБСТРАКТНЫЕ МЕТОДЫ И АБСТРАКТНЫЕ КЛАССЫ

- Хотим работать не только с list
- Готовы принимать любую коллекцию с нумерацией элементов и реализующую соответствующие специальные методы
- А если это не так - хотим бросить свое исключение

АБСТРАКТНЫЕ МЕТОДЫ И АБСТРАКТНЫЕ КЛАССЫ

- Общее решение - научиться задавать контракт
- Контракт - ожидание от объекта своей реализации набора методов
- Реализовано через декоратор abstractmethod
- И сопутствующий класс ABC - от которого нужно наследоваться

КАК-ТО ТАК

```python
from abc import ABC, abstractmethod

class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
```

ПРИМЕРНАЯ СХЕМА РАБОТЫ

- Декоратор помечает метод как абстрактный
- При создании объекта вызывается инициализирующая логика суперкласса ABC
- Она проверяет наличие реализации всех абстрактных методов в классе создаваемого объекта
- Если хотя бы одного нет - бросает исключение

ВОЗВРАЩАЕМСЯ К ПРИМЕРУ

- Заведем абстрактный класс
- В нем опишем ожидаемые методы
- На входе будем проверять - принадлежит ли наш объект абстрактному классу
- Но есть нюансы

ПРОБЛЕМЫ

- Для разных методов нужны разные контракты
- Хочется, чтобы один объект умел реализовывать разные контракты
- То есть наследоваться от нескольких классов
- В этом помогает множественное наследование

ПРИМЕР

```
 1 class C1(ABC):
 2     @abstractmethod
 3     def m1(self, arg1):
 4         ...
 5
 6 class C2(ABC):
 7     @abstractmethod
 8     def m2(self, arg1):
 9         ...
10
11 # ........................
```

ПРИМЕР

```
1 # ......................
2 class C(C1, C2):
3     def m1(self):
4         pass
5
6     def m2(self):
7         pass
```

ДРУГАЯ ПРОБЛЕМА

- Мы не можем библиотечный класс сделать наследником нашего базового абстрактного класса
- Аналогично со сторонними реализациями похожих структур
- Стандартных структур немного, их можно явно проверить
- Для сторонних коллекций типовые конктракты определены в collections.abc

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

- У класса может быть несколько наследников
- Формально перечисляются через запятую
- Все включаются в поиск атрибута
- В каком порядке - можно узнать в атрибуте __mro__

ПРИМЕР

```
 1 class C1:
 2     pass
 3
 4 class C2(list, C1):
 5     pass
 6
 7 class C3(C1, list):
 8     pass
 9
10
11 print(C1.__mro__)
12 print(C2.__mro__)
13 print(C3.__mro__)
```

ПРИМЕР

```
 1 class C1:
 2     pass
 3
 4 class C2(C1):
 5     pass
 6
 7 class C3(C1):
 8     pass
 9
10 class C4(C2, C3):
11     pass
12
13 print(C4.__mro__)
```

ПРИМЕР

```
 1 class C1:
 2     pass
 3 class C2(C1):
 4     pass
 5 class C3(C2):
 6     pass
 7 class C4(C1):
 8     pass
 9 class C5(C4):
10     pass
11 class C6(C3, C5):
12     pass
13
14 print(C6.__mro__)
```

ХОРОШИЕ ПРАКТИКИ

- Избегаем сложных графов наследования
- Полезный вариант: много абстрактных классов
- Полезный вариант: mixin (подмешивание опционального поведения)
- Полезный вариант: общее поведение, раскладывающееся на самодостаточные части (reader/writer и т.п.)

ДЕСКРИПТОРЫ, СВОЙСТВА

- Мотивирующие примеры
- Хотим запретить обновление определенного атрибута объекта
- Или удаление
- Или хотим, чтобы любой атрибут имел значение None, если не был явно определен

BASELINE

- Специальные методы: __getattribute__ и __getattr__
- Позволяют "перехватывать" доступ к существующим атрибутам
- И обрабатывать доступ к несуществующим
- На них можно многое построить

ПРИМЕР

```
 1 class C:
 2     def __init__(self, v):
 3         self.v = v
 4 
 5     def __getattribute__(self, name):
 6         print("__getattribute__", name)
 7         return super().__getattribute__(name)
 8 
 9     def __getattr__(self, name):
10         print("__getattr__", name)
11         return None
12 
13 C(123).v
14 C(234).q
```

РАЗВИВАЕМ ИДЕЮ

- Аналогичные методы: __setattr__ и __detattr__
- Позволяют "перехватывать" присваивания и удаления атрибутов
- Можно не только запретить
- Можно, например, организовать валидацию

ЧТО НЕ ТАК

- Задачи довольно типовые
- Повторяющиеся из класса в класс - с небольшими изменениями
- Хочется эту общую логику куда-то вынести
- Чтобы она жила вне класса

ПРОТОКОЛ ДЕСКРИПТОРОВ

- Дескриптор - класс, реализующий хотя бы один специальный метод из заданного набора
- Простейший вариант: `__get__(self, obj, type=None)`
- Сам по себе класс остается обычным классом
- А объект - обычным объектом

ПРИМЕР

```
1 class C:
2     def __get__(self, obj, type=None):
3         print("obj", obj)
4         print("type", type)
5         return 42
6
7 print(C().__get__("hello"))
```

ПРИМЕР

```
 1 class D:
 2     c = C()
 3
 4 print(D.c)
 5 # obj None
 6 # type <class '__main__.D'>
 7
 8 print(D().c)
 9 # obj <__main__.D object at 0x7fc81df5beb0>
10 # type <class '__main__.D'>
```

ОСОЗНАЕМ ПРОИСХОДЯЩЕЕ

- При разрешении символа 'c' как элемента объекта дошли до уровня класса и увидели объект-дескриптор
- Во втором случае сразу обнаружили объект-дескриптор
- Сработал его метод __get__
- Результат __get__ выступает как значение атрибута

ОСОЗНАЕМ ПРОИСХОДЯЩЕЕ

ПРИМЕР

```
1 class C:
2     def __get__(self, obj, type=None):
3         print("obj", obj)
4         print("type", type)
5         return 42
6
7
8 # ........................
```

ПРИМЕР

```
 1 # ....................
 2
 3 class D:
 4     c = C()
 5
 6 class D2:
 7     c = C()
 8     d = c
 9
10 class D3:
11     c = D.c # не факт, что эффект совпадает с ожиданием
```

СОСТОЯНИЕ

- Можем хранить состояние дескриптора разными способами
- Можно в самом объекте дескрипторе
- Если непосредственно - оно будет общим для всех употреблений данного дескриптора
- Можно завести словарь, ключ которого - id объекта
- А можно завести парное свойство в объекте

ДАЛЬНЕЙШЕЕ РАЗВИТИЕ

- Логично ожидать методы для обновления и удаления
- `__set__(self, obj, value)`
- `__delete__(self, obj)`
- Получаем полный набор

РЕАЛИЗУЕМ ОГРАНИЧЕНИЯ

- Хотим для полей класса указать, какие данные мы там ожидаем
- Например, целое число из заданного диапазона
- Или одна из перечисленных строк
- Реализуем целый набор декрипторов с общим интерфейсом

ПРИМЕР

```
 1 from abc import ABC, abstractmethod
 2
 3 class Validator(ABC):
 4
 5     def __set_name__(self, owner, name):
 6         self.private_name = '_' + name
 7
 8     def __get__(self, obj, objtype=None):
 9         return getattr(obj, self.private_name)
10
11 # ....................................
```

ПРИМЕР

```
1 # ....................................
2
3     def __set__(self, obj, value):
4         self.validate(value)
5         setattr(obj, self.private_name, value)
6
7     @abstractmethod
8     def validate(self, value):
9         pass
```

ONEOF

```
 1 class OneOf(Validator):
 2     def __init__(self, *options):
 3         self.options = set(options)
 4
 5     def validate(self, value):
 6         if value not in self.options:
 7             raise ValueError(
 8     f'Expected {value!r} to be one of {self.options!r}')
 9
10 test = OneOf('green', 'red', 'blue')
11 test.validate('green')
12 test.validate('red')
13 test.validate('blue')
```

NUMBER

```
 1 class Number(Validator):
 2
 3     def __init__(self, minvalue=None, maxvalue=None):
 4         self.minvalue = minvalue
 5         self.maxvalue = maxvalue
 6
 7     def validate(self, value):
 8         if not isinstance(value, (int, float)):
 9             raise TypeError(
10                 f'Expected {value!r} to be an int or float'
11             )
12
13 # ............................
```

NUMBER

```
1 # ............................
2         if self.minvalue is not None and
3            value < self.minvalue:
4             raise ValueError(
5     f'Expected {value!r} to be at least {self.minvalue!r}'
6             )
7 # ............................
```

NUMBER

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

        if self.maxvalue is not None and
           value > self.maxvalue:
            raise ValueError(
                f"Expected {value!r} "
                 "to be no more than {self.maxvalue!r}"
            )

test = Number()
test.validate(12345)
test.validate(-12.345)
```

STRING

```
1 class String(Validator):
2
3     def __init__(self, minsize=None, maxsize=None,
4                  predicate=None):
5         self.minsize = minsize
6         self.maxsize = maxsize
7         self.predicate = predicate
8
9 # ................................
```

STRING

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

    def validate(self, value):
        if not isinstance(value, str):
            raise TypeError(
                f'Expected {value!r} to be an str'
            )
        if self.minsize is not None and
           len(value) < self.minsize:
            raise ValueError(
                f'Expected {value!r} to be no smaller '
                 'than {self.minsize!r}'
            )
# ................................
```

STRING

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

        if self.maxsize is not None and
           len(value) > self.maxsize:
            raise ValueError(
                f'Expected {value!r} to be no bigger '
                'than {self.maxsize!r}'
            )
        if self.predicate is not None and
           not self.predicate(value):
            raise ValueError(
                f'Expected {self.predicate} to be true '
                'for {value!r}'
            )
```

STRING

```
 1 class Component:
 2     name = String(minsize=3, maxsize=10,
 3                   predicate=str.isupper)
 4     kind = OneOf('wood', 'metal', 'plastic')
 5     quantity = Number(minvalue=0)
 6
 7     def __init__(self, name, kind, quantity):
 8         self.name = name
 9         self.kind = kind
10         self.quantity = quantity
11
12 c = Component('WIDGET', 'metal', 5)
13
14 print(c)
```

ОСМЫСЛИМ

- Выглядит как крен в сторону классического ООП
- Мы фактически задаем схему класса
- Именно на уровне класса
- Только работает она все равно динамически

ОСМЫСЛИМ

- Это не отказ от гибкости и динамичности
- Это попытка привнести лучшее из мира типизации
- Но "типы" существуют как объекты
- Их можно создавать, переиспользовать, настраивать

ЕЩЕ ОДИН TRADE-OFF

- В примерах в каждом классе создавался новый дескриптор-объект
- Это задавало декларативный стиль
- Но мы могли вы создать объект где-то в другом месте
- А в описании класса - сослаться на него
- Особенно если класс создается в рамках мета-фреймворка как результат работы функции

ЕЩЕ ОДИН TRADE-OFF

- Пример: переиспользование логирующего дескриптора
- Мы могли бы его переиспользовать с пользой
- Например, если хотим логировать доступ полю age в разных классах
- С одинаковыми параметрами логирования
- Которые бы отражались в параметрах конструктора

ЕЩЕ ОДИН TRADE-OFF

- Это бы работало нормально, если вы один экземпляр привязывали к одному и тому же имени
- Но мы можем это проверять в `__set_name__`
- Другой вариант: разрешать привязывать один экземпляр внутри одного класса
- Пусть даже на разные имена
- И хранить знания об этих именах
- Особая аккуратность, если у дескриптора - свое состояние

ВАЖНОЕ ОГРАНИЧЕНИЕ

- В `__set__`/`__get__` не приходит имя, инициировавшее операцию
- Это ограничивает переиспользование одного экземпляра дескриптора внутри одного класса
- Только в случаях, когда поведение не привязано к полю
- Или мы намеренно хотим сделать одно поле псевдонимом другого

LOGGEDACCESS

```
 1 import logging
 2
 3 logging.basicConfig(level=logging.INFO)
 4
 5 class LoggedAccess:
 6     def __init__(self):
 7         self._owner = None
 8
 9     def __set_name__(self, owner, name):
10         if not self._owner:
11             self.public_name = name
12             self.private_name = '_' + name
13             self._owner = owner
14 # .....................................
```

LOGGEDACCESS

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

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r',
                     self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r',
                     self.public_name, value)
        setattr(obj, self.private_name, value)
# ................................
```

LOGGEDACCESS

```
1  # ..................................
2  class Person:
3
4      name = LoggedAccess()
5      age = LoggedAccess()
6      summers = age
7
8      def __init__(self, name, age):
9          self.name = name
10         self.age = age
11
12     def birthday(self):
13         self.age += 1
14 # ..................................
```

LOGGEDACCESS

```
1 # ..................................
2 mary = Person('Mary M', 30)
3 dave = Person('David D', 40)
4
5 print(dave.age, mary.summers)
6
7 dave.summers = 35
8 mary.age -= 1
9 print(dave.summers, mary.age)
```

ORM

- ORM - object-relational mapping
- Шлюз между понятиями ООП и реляционной моделью данных
- В реляционной модели однотипные объекты хранятся в таблицах
- Поля таблицы соответствуют свойствам объекта

ORM

- Создание объекта - INSERT-запрос
- Чтение объекта - SELECT
- Изменение свойств - UPDATE
- Это в первом приближении

ORM

- Во втором приближении - нужно межтабличные взаимодействия поддерживать
- Делать JOIN под капотом объектной абстракции
- Обновлять записи как-то оптимальнее, чем одиночными UPDATE/INSERT-ами
- И какое-то итерирование

ORM

- В любом случае - хотим работать с объектами
- И чтобы это отображалось в какую-то вспомогательную логику
- Частью которой являются SQL-запросы
- И здесь помогают декораторы и дескрипторы

СХЕМА РЕАЛИЗАЦИИ

- Нужно знание о том, в какой базе все хранится
- Если это SQL, то какой именно (от этого зависит синтаксис)
- Знание о деталях доступа: хост, порт, пароли (не в исходниках)
- Это может быть класс с аттрибутами уровня класса
- С декоратором вроде `@Repository`

СХЕМА РЕАЛИЗАЦИИ

- Такой декоратор зарегистрирует репозиторий
- И зафиксирует конфигурацию для доступа
- Для классов, которые хотим хранить, будет свой декоратор, вроде `Entity`
- Декларация желания хранить объекты этого класса в репозитории

СХЕМА РЕАЛИЗАЦИИ

- Каждому классу выделяем свою таблицу
- По умолчанию имя таблицы - имя класса
- Но можно указать параметром декоратора
- Как и какие-то опции для создания таблицы

СХЕМА РЕАЛИЗАЦИИ

- Через дескрипторы указываем поля таблицы
- Там же можно указывать ограничения
- Какие поля и как индексировать
- Кто будет первичным ключом

КАК-ТО ТАК

```
class Field:
    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM '
                     f'{owner.table} WHERE {owner.key}=?;'
        self.store = f'UPDATE {owner.table} '
                     f'SET {name}=? WHERE {owner.key}=?;'

    def __get__(self, obj, objtype=None):
        return conn.execute(self.fetch,
                            [obj.key]).fetchone()[0]

    def __set__(self, obj, value):
        conn.execute(self.store, [value, obj.key])
        conn.commit()
```

КАК-ТО ТАК

```
 1 class Movie:
 2     table = 'Movies'          # Table name
 3     key = 'title'             # Primary key
 4     director = Field()
 5     year = Field()
 6
 7     def __init__(self, key):
 8         self.key = key
 9
10 # ........................
```

КАК-ТО ТАК

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

class Song:
    table = 'Music'
    key = 'title'
    artist = Field()
    year = Field()
    genre = Field()

    def __init__(self, key):
        self.key = key
```

PROPERTY

- `property` реально реализована на C
- Можно реконструировать его логику через Python-класс
- И использовать механизм дескрипторов

PROPERTY

```
 1 class Property:
 2     def __init__(self, fget=None, fset=None,
 3                  fdel=None, doc=None):
 4         self.fget = fget
 5         self.fset = fset
 6         self.fdel = fdel
 7         if doc is None and fget is not None:
 8             doc = fget.__doc__
 9         self.__doc__ = doc
10         self._name = ''
11
12     def __set_name__(self, owner, name):
13         self._name = name
14 # ..................
```

PROPERTY

```
# ..................
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError(
        f'property {self._name!r} '
        f'of {type(obj).__name__!r} object has no getter'
             )
        return self.fget(obj)
# ..................
```

PROPERTY

```
# ..................
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError(
        f'property {self._name!r} of '
        f'{type(obj).__name__!r} object has no setter'
             )
        self.fset(obj, value)
# ..................
```

PROPERTY

```
# ..................
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError(
        f'property {self._name!r} of '
        f'{type(obj).__name__!r} object has no deleter'
             )
        self.fdel(obj)
```

PROPERTY

```
 1     def getter(self, fget):
 2         prop = type(self)(fget, self.fset,
 3                           self.fdel, self.__doc__)
 4         prop._name = self._name
 5         return prop
 6
 7     def setter(self, fset):
 8         prop = type(self)(self.fget, fset,
 9                           self.fdel, self.__doc__)
10         prop._name = self._name
11         return prop
12
13 #...................
```

PROPERTY

```
1 #..................
2
3     def deleter(self, fdel):
4         prop = type(self)(self.fget, self.fset,
5                           fdel, self.__doc__)
6         prop._name = self._name
7         return prop
```