Лекция 7: аттрибуты объектов, дескрипторы, json и регулярные выражения

Работа с аттрибутами объектов

- По-умолчанию каждый объект имеет словарь полученный от своего класса, в котором хранятся поля объекта.
- Словарь доступен как ObjectClass.__dict__.
- При чтении (value = obj.name) или записи (obj.name = value) по имени обращение автоматически происходит к объекту в словаре по ключу name.
- Поведение при работе с аттрибутами объекта (полями, доступными по имени) можно переопределять.

Переопределение доступа к полям по имени

- __getattr__(self, name) - переопределить доступ к аттрибуту (a = x.name) при попытке чтения.
  - Вызывается только тогда, когда не удалось найти аттрибут обычными способами (не является полем текущего объекта или какого-либо класса в иерархии предков).
  - Должен либо вернуть сформированное значение по указанному имени, либо вызвать AttributeError.
  - Важно: используется только, если поле не найдено по имени другим способом.

- __setattr__(self, name, value) - при попытке записи данных в аттрибут (x.name = a).
  - Вызывается при попытке присваивания, перекрывает обычный механизм присваивания (в отличие от __getattr__).
  - Для присваивания аттрибута текущему объекту над вызвать __setattr__() у базового класса (self.name = value не сработает).
  - Важно: в отличие от предыдущего __getattr__ перекрывает стандартный механизм.

- __delattr__(self, name) - удалить аттрибут по имени.
  - Реализовывать стоит только в том случае, если del x.name имеет смысл для текущего объекта.

- __getattribute__(self, name) - переопределить доступ к аттрибуту при чтении.
  - Подобен __getattr__, но в отличие от него перекрывает обычный механизм безусловно.
  - В случае, если также определён __getattr__, то последний вызовется только если его явно вызвать из __getattribute__ или бросить исключение AttributeError в __getattribute__.
  - Для получения аттрибута у текущего объекта, во избежание рекурсии, надо вызывать __getattribute__ у базового класса.

Определение property полей

Декоратор property позволяет определять виртуальные поля, задавая (или нет) отдельные методы в качестве геттера, сеттера, удаляющего, а также определить документацию.

In [12]:
```python
class PseudoMeter(object):
    def __init__(self):
        self.__temperature = 0

    def __get_temperature(self):
        print("Asked for temperature")
        return self.__temperature

    def __set_temperature(self, value):
        print("Got new value")
        if not isinstance(value, int):
            raise TypeError("int value required")
        self.__temperature = value

    temperature = property(__get_temperature, __set_temperature)

meter = PseudoMeter()
meter.temperature = 10
print(meter.temperature)
print(meter._PseudoMeter__temperature)
```

```
Got new value
Asked for temperature
10
10
```

Это был простейший пример. Можно делать аккуратнее и, зависимости от конкретного случая, использовать данный декоратор различными способами.

In [13]:
```python
class PseudoMeter(object):
    def __init__(self):
        self.__temperature = 0

    @property
    def temperature(self):
        print("Asked for temperature")
        return self.__temperature

meter = PseudoMeter()
print(meter.temperature)
print(meter._PseudoMeter__temperature)
```

```
Asked for temperature
0
0
```

In [14]:
```python
class PseudoMeter(object):
    def __init__(self):
        self.__temperature = 0

    @property
    def temperature(self):
        print("Asked for temperature")
        return self.__temperature

    @temperature.setter
    def temperature(self, value):
        print("Got new value")
        if not isinstance(value, int):
            raise TypeError("int value required")
        self.__temperature = value

meter = PseudoMeter()
meter.temperature = 10
print(meter.temperature)
print(meter._PseudoMeter__temperature)
```

```
Got new value
Asked for temperature
10
10
```

In [15]:
```python
class PseudoMeter(object):
    def __init__(self):
        self.__temperature = 0

    def get_temperature(self):
        print("Asked for temperature")
        return self.__temperature

    def set_temperature(self, value):
        print("Got new value")
        if not isinstance(value, int):
            raise TypeError("int value required")
        self.__temperature = value

    temperature = property(get_temperature, set_temperature)


meter = PseudoMeter()
meter.temperature = 10
print(meter.temperature)
print(meter._PseudoMeter__temperature)
```

```
Got new value
Asked for temperature
10
10
```

Последний способ не рекомендуется к написанию и показан лишь для примера.

Дескрипторы

- Дескриптор данных (data descriptor) - если определены __get__ и __set__.
- Дескриптор не-данных (non-data descriptor) - если определен __get__, но не __set__.
- Дескриптор данных имеет приоритет перед поиском в словаре текущего объекта, в котором он находится.
- Дескриптор не-данных запрашивается только после того, как по имени в текущем объекте не было найдено аттрибутов.
- Read-only дескриптор данных - реализовать __set__, но выбрасывать из него AttributeError.

Протокол дескрипторов

- descr.__get__(self, instance, owner) --> value
  - Вызывается при обращении для чтения к аттрибуту класса-владельца (class attribute) или объекта-владельца (instance attribute).
  - instance - объект-владелец, для которого вызвано получение аттрибута (None, если вызвано для класса).
  - owner - класс-владелец.
  - Должен либо возвращать значение, либо бросать AttributeError.

- descr.__set__(self, instance, value) --> None

  - Вызывается для присваивания аттрибуту объекта-владельца instance значения value.
- descr.__delete__(self, instance) --> None

  - Вызывается для удаления аттрибута из объекта-владельца instance.

Варианты вызова дескрипторов

- Напрямую - descr.__get__(instance, type(instance))
  - Обычно не используется.
- От объекта - obj.descr:
  - type(instance).__dict__['descr'].__get__(instance, type(instance))
- От класса - obj.descr
  - owner.__dict__['descr'].__get__(None, owner)
- От super - super(owner, instance).descr
  - Ищет в instance.__class__.__mro__ ближайший базовый класс-предок ownerBase и вызывает дескриптор.
  - Вызвает так: ownerBase.__dict__['descr'].__get__(instance, instance.__class__)

Замечания про дескрипторы и переопределение доступа к полям

- Преобразования из предыдущего списка автоматически делает __getattribute__.
- Если его переопределить, то больше не будет работать автоматическое использование дескрипторов.
- В зависимост от варианта вызова используется object.__getattribute__ (для объекта) или type.__getattribute__ (для класса).

Примеры дескрипторов в языке Python

Функции staticmethod, classmethod и property (также используемые как декораторы) реализованы с помощью протокола дескрипторов.

Пример property на чистом Python (в стандартном интерпретаторе реализована нативно):

In [16]:
```python
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

Функции и методы

- ООП в Python основывается на использовании функций.
- Для объединения этих двух частей воедино используются дескрипторы не-данных.

Как это работает:

- В словарях полей классов методы хранятся просто как функции, реализованные через def и lambda.
- Единственное отличие - соглашение о том, что первый аргумент соответствует вызывающему объекту self.
- Все объекты-функции имеют __get__, что делает их дескрипторами не-данных.
- При доступе к объекту-функции в зависимости от того, как он произведён (объект-владелец или через класс-владелец) возвращает связанные или несвязанные методы соответственно.

In [17]:
```python
class Owner(object):
    def func(self, value):
        return value

instance = Owner()
print(Owner.__dict__['func'])
print(Owner.func)
print(instance.func)
```

```
<function Owner.func at 0x7f7e3ef2ad30>
<function Owner.func at 0x7f7e3ef2ad30>
<bound method Owner.func of <__main__.Owner object at 0x7f7e3e63c730>>
```

Посмотрим ещё примеры реализации версий staticmethod и classmethod на чистом Python:

In [18]:
```python
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

In [19]:
```python
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)

        def newfunc(*args):
            return self.f(klass, *args)

        return newfunc
```

JSON

- JSON - JavaScript Object Notation - формат передачи данных.
- Не зависит от языка - поддержка есть практически для всех языках программирования.
- Имеет человекочитаемый вид.
- Позволяет просто и понятно описывать структурированные данные.
- Значения: объект, массив, строка, число, true, false, null.

In [20]:
```python
response_message = """
{
  "data": [
    {
      "type": "articles",
      "id": "1",
      "attributes": {
        "title": "JSON API paints my bikeshed!",
        "body": "The shortest article. Ever.",
        "created": "2015-05-22T14:56:29.000Z",
        "updated": "2015-05-22T14:56:28.000Z"
      },
      "relationships": {
        "author": {
          "data": {
            "id": "42",
            "type": "people"
          }
        }
      }
    }
  ],
  "included": [
    {
      "type": "people",
      "id": "42",
      "attributes": {
        "name": "John",
        "age": 80,
        "gender": "male"
      }
    }
  ]
}
"""
```

В языке Python есть модуль стандартной библиотеки для работы с JSON:

In [21]:
```python
import json
dir(json)
```

Out[21]:
```
['JSONDecodeError',
 'JSONDecoder',
 'JSONEncoder',
 '__all__',
 '__author__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '_default_decoder',
 '_default_encoder',
 'codecs',
 'decoder',
 'detect_encoding',
 'dump',
 'dumps',
 'encoder',
 'load',
 'loads',
 'scanner']
```

С помощью этого модуля можно превратить json в объекты стандартные объекты языка Python.

In [23]:
```python
# convert from json text to python data
data = json.loads(response_message)
from pprint import pprint

pprint(data)
```

```
{'data': [{'attributes': {'body': 'The shortest article. Ever.',
                          'created': '2015-05-22T14:56:29.000Z',
                          'title': 'JSON API paints my bikeshed!',
                          'updated': '2015-05-22T14:56:28.000Z'},
           'id': '1',
           'relationships': {'author': {'data': {'id': '42',
                                                  'type': 'people'}}},
           'type': 'articles'}],
 'included': [{'attributes': {'age': 80, 'gender': 'male', 'name': 'John'},
               'id': '42',
               'type': 'people'}]}
```

In [24]:
```python
print(data.keys(), type(data))
print(data["data"][0]["attributes"])
```

```
dict_keys(['data', 'included']) <class 'dict'>
{'title': 'JSON API paints my bikeshed!', 'body': 'The shortest article. Ever.', 'created': '2015-05-22T14:56:29.000Z', 'updated': '2015-05-22T14:56:28.000Z'}
```

Или наоборот:

In [25]:
```python
target_description = {
    "id": 100500,
    "name": "ahaha",
    "marks": [5, 7, 5, 8],
    "is_valid": True
}

print(json.dumps(target_description))
```

```
{"id": 100500, "name": "ahaha", "marks": [5, 7, 5, 8], "is_valid": true}
```

In [26]:
```python
print(json.dumps(target_description, indent=4))
```

```
{
    "id": 100500,
    "name": "ahaha",
    "marks": [
        5,
        7,
        5,
        8
    ],
    "is_valid": true
}
```

Кроме этого:

- Аналогично есть функции load/dump, которые вместро строк работают с file-like объектами.
- Можно унаследоваться от JSONEncoder/JSONDecoder и определить свои правила кодирования-декодирования json с помощью функций loads/load/dump/dumps.

Подробнее про json:

- http://www.json.org/
- https://en.wikipedia.org/wiki/JSON
- https://docs.python.org/3/library/json.html

Регулярные выражения

- Регулярное выражение - формальный язык поиска и манипуляций над текстом, представлен грамматикой со специальными метасимволами, задающими определённые правила преобразования над последовательностями символов.
- Шаблон (pattern) - совокупность символов и метасимволов задающее правила поиска и преобразований в его процессе.
- В стандартной библиотеке Python есть библиотека для удобной работы с регулярными выражениями.
- Библиотека называется re.

In [27]:
```python
import re

print(dir(re))
```

```
['A', 'ASCII', 'DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'Match', 'Pattern', 'RegexFlag', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE',
'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version__', '_cache
', '_compile', '_compile_repl', '_expand', '_locale', '_pickle', '_special_chars_map', '_subx', 'compile', 'copyreg', 'enum', 'error', 'escape', 'findall', 'finditer',
'fullmatch', 'functools', 'match', 'purge', 'search', 'split', 'sre_compile', 'sre_parse', 'sub', 'subn', 'template']
```

In [28]:
```python
import re

pattern = "abc"
match = re.match(pattern, "abc")
print(match.pos)
```

```
0
```

In [29]:
```python
print(re.match("^abc$", "abc"))
print(re.match("^abc$", "abcd"))
```

```
<re.Match object; span=(0, 3), match='abc'>
None
```

In [30]:
```python
print(re.match("abc", "dabc"))
print(re.search("abc", "dabc"))
```

```
None
<re.Match object; span=(1, 4), match='abc'>
```

In [31]:
```python
help(re.match)
```

```
Help on function match in module re:

match(pattern, string, flags=0)
    Try to apply the pattern at the start of the string, returning
    a Match object, or None if no match was found.
```

In [32]:
```python
help(re.search)
```

```
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a Match object, or None if no match was found.
```

In [34]:
```python
print(re.match("a[0-9]b", "a1bcd"))
```

```
<re.Match object; span=(0, 3), match='a1b'>
```

In [35]:
```python
print(re.match("a[0-9]b", "a1124234234b"))
print(re.match("a[0-9]*b", "a1124234234b"))
```

```
None
<re.Match object; span=(0, 12), match='a1124234234b'>
```

In [39]:
```python
match = re.match("(.+)@(.+\.com)", "user@domain.com")
print(match.group())
print(match.group(1))
print(match.group(2))
```

```
user@domain.com
user
domain.com
```

Чтиво про регулярные выражения:

- https://ru.wikipedia.org/wiki/Регулярные_выражения
- https://ru.wikibooks.org/wiki/Регулярные_выражения
- https://docs.python.org/3/library/re.html
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions
- http://www.regexr.com/
- http://habrahabr.ru/post/115825/
- Книга "Mastering Regular Expressions", by Jeffrey Friedl
- https://swtch.com/~rsc/regexp/regexp1.html
- https://swtch.com/~rsc/regexp/regexp2.html
- https://swtch.com/~rsc/regexp/regexp3.html

Важно понимать, когда стоит употреблять:

"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems."

- https://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/
- https://programmers.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions
- https://blog.codinghorror.com/regex-use-vs-regex-abuse/

Полезно на будущее для самостоятельного чтения:

- Для понимания языков программирования, низкоуровневых идей в основе парсеров и того, как работают регулярные выражения: книга "Введение в теорию автоматов, языков и вычислений.". Авторы: Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман.