Программирование на Python. Часть 7: Специальные методы и атрибуты классов

Сергей Яковлев
Консультант
независимый специалист

01.07.2010

Классы в питоне имеют большой набор встроенных методов и атрибутов, которые позволяют гибко использовать модель объектно-ориентированного программирования и упрощают решение стандартных задач и алгоритмов.

Мы продолжаем изучать классы в Python. Специальные зарезервированные методы здесь имеют префикс — двойной символ подчеркивания. С их помощью реализованы такие механизмы, как конструкторы, последовательности, итераторы, проперти, слоты и т.д.

Сегодня мы рассмотрим следующие темы.

1. Объекты классов и специальные методы.
2. Экземпляры классов и специальные методы.
3. Экземпляры классов в качестве последовательностей.
4. Приведение объектов к базовым типам.
5. Bound и unbound методы.
6. Метод super.
7. Статические методы.
8. Итератор.
9. Property.
10. Singleton.
11. Слоты.
12. Функтор.
13. Дескриптор.
14. Sequence.

1. Объекты классов и специальные методы

Объект-класс создается с помощью определения класса. Объекты-классы имеют следующие атрибуты:

`__name__` — имя класса;

`__module__` — имя модуля;

`__dict__` — словарь атрибутов класса, можно изменять этот словарь напрямую;

`__bases__` — кортеж базовых классов в порядке их следования;

`__doc__` — строка документации класса.

2. Экземпляры классов и специальные методы

Экземпляр (инстанс) класса возвращается при вызове объекта-класса. Объект у класса может быть один, экземпляров (или инстансов) — несколько. Экземпляры имеют следующие атрибуты:

`__dict__` — словарь атрибутов класса, можно изменять этот словарь напрямую;

`__class__` — объект-класс, экземпляром которого является данный инстанс;

`__init__` — конструктор. Если в базовом классе есть конструктор, конструктор производного класса должен вызвать его;

`__del__` — деструктор. Если в базовом классе есть деструкор, деструктор производного класса должен вызвать его;

`__cmp__` — вызывается для всех операций сравнения;

`__hash__` — возвращает хеш-значение объекта, равное 32-битному числу;

`__getattr__` — возвращает атрибут, недоступный обычным способом;

`__setattr__` — присваивает значение атрибуту;

`__delattr__` — удаляет атрибут;

`__call__` — срабатывает при вызове экземпляра класса.

3. Экземпляры классов в качестве последовательностей

Экземпляры классов можно использовать для эмуляции последовательностей. Для такой реализации есть встроенные методы:

`__len__` — возвращает длину последовательности;

`__getitem__` — получение элемента по индексу или ключу;

`__setitem__` — присваивание элемента с данным ключом или индексом;

`__delitem__` — удаление элемента с данным ключом или индексом;

`__getslice__` — возвращает вложенную последовательность;

`__setslice__` — заменяет вложенную последовательность;

`__delslice__` — удаляет вложенную последовательность;

`__contains__` — реализует оператор in.

4. Приведение объектов к базовым типам

Объекты классов можно привести к строковому или числовому типу.

`__repr__` — возвращает формальное строковое представление объекта;

`__str__` — возвращает строковое представление объекта;

`__oct__ , __hex__ , __complex__ , __int__ , __long__ , __float__` — возвращают строковое представление в соответствующей системе счисления.

5. Bound и unbound методы

Рассмотрим конкретный пример. Есть базовый класс Cat, и есть производный от него класс Barsik:

```
class Cat:
  def __init__(self):
      self.hungry = True
  def eat(self):
      if self.hungry:
          print 'I am hangry...'
          self.hungry = False
      else:
          print 'No, thanks!'

class Barsik(Cat):
  def __init__(self):
      self.sound = 'Aaaammm!'
      print self.sound
```

Создаем экземпляр производного класса:

```
>>> brs = Barsik()
Aaaammm!
>>> brs.eat()
AttributeError: Barsik instance has no attribute 'hungry'
```

На первый взгляд — странная ошибка, поскольку атрибут hungry есть в базовом классе.
На самом деле, конструктор производного класса — перегруженный, при этом конструктор базового класса не вызывается, и его нужно явно вызвать. Это можно сделать двумя путями. Первый вариант считается устаревшим:

```
class Barsik(Cat):
  def __init__(self):
      Cat.__init__(self)
      self.sound = 'Aaaammm!'
      print self.sound
```

Здесь мы напрямую вызываем конструктор базового класса, не создавая инстанс базового класса Cat — поэтому такой базовый конструктор относится к категории unbound-методов, в пику методам, которые вызываются для инстансов классов и называются bound-методами. Для вызова bound-метода в качестве первого параметра методу нужно передать инстанс класса.

6. Метод super

Второй вариант: в начале программы нужно определить метакласс , который указывает на то, что класс реализован в так называемом новом стиле — new-style. Затем нужно вызвать стандартный метод super для базового конструктора:

```
__metaclass__ = type
...
class Barsik(Cat):
    def __init__(self):
      super(Barsik, self).__init__()
      self.sound = 'Aaaammm!'
      print self.sound

>>> brs = Barsik()
>>> brs.eat()
Aaaammm!
I am hangry...
```

7. Статические методы

Статический метод — функция, определенная вне класса и не имеющая атрибута self:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

def printNumInstances( ):
    print "Number of instances created: ", Spam.numInstances

>>> a=Spam()
>>> b=Spam()
>>> printNumInstances()
Number of instances created:  2
```

Статический метод может быть определен и внутри класса — для этого используется ключевое слово `staticmethod`, причем метод может быть вызван как статически, так и через инстанс:

```
class Multi:
    def imeth(self, x):
        print self, x
    def smeth(x):
        print x
    def cmeth(cls, x):
        print cls, x
    smeth = staticmethod(smeth)
    cmeth = classmethod(cmeth)

>>> Multi.smeth(3)
3
>>> obj=Multi()
>>> obj.smeth(5)
5
```

Методы класса определяются с помощью ключевого слова `classmethod` — здесь автоматически питон передает в качестве первого параметра сам класс (cls):

```
>>> Multi.cmeth(7)
__main__.Multi 7
>>> obj.cmeth(10)
__main__.Multi 10
```

8. Итератор

Итераторы хороши там, где списки не подходят в силу того, что занимают много памяти, а итератор возвращает его конкретное значение. В классе нужно определить два стандартных метода — `__iter__` и `next`. Метод `__iter__` будет возвращать объект через метод `next`:

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('12345'):
>>>   print char
5
4
3
2
1
```

Итератор можно сконвертировать в список:

```
>>> rvr = list(Reverse('12345'))
>>> rvr
['5', '4', '3', '2', '1']
```

9. Property

Property — атрибут класса, возвращаемый через стандартную функцию property, которая в качестве аргументов принимает другие функции класса:

```
class DateOffset:
    def __init__(self):
        self.start = 0

    def _get_offset(self):
        self.start +=  5
        return self.start

    offset = property(_get_offset)


>>> d = DateOffset()
>>> d.offset
5
>>> d.offset
10
```

10. Singleton

Данный паттерн позволяет создать всего один инстанс для класса. Используется метод `__new__`:

```
class Singleton(object):
  def __new__(cls, *args, **kw):
      if not hasattr(cls, '_instance'):
        orig = super(Singleton, cls)
        cls._instance = orig.__new__(cls, *args, **kw)
      return cls._instance


>>> one = Singleton()
>>> two = Singleton()
>>> id(one)
3082687532
>>> id(two)
3082687532
```

11. Слоты

Слоты — это список атрибутов, задаваемый в заголовке класса с помощью `__slots__`. В инстансе необходимо назначить атрибут, прежде чем пользоваться им:

```
class limiter(object):
    __slots__ = ['age', 'name', 'job']

>>> x=limiter()
>>> x.age = 20
```

12. Функтор

Функтор — это класс, имеющий метод `__call__` — при этом объект можно вызвать как функцию.

Пример. Пусть у нас имеется класс Person, имеется коллекция объектов этого класса-people, нужно отсортировать эту коллекцию по фамилиям. Для этого можно использовать функтор Sortkey:

```
class SortKey:
  def __init__(self, *attribute_names):
      self.attribute_names = attribute_names

  def __call__(self, instance):
      values = []
      for attribute_name in self.attribute_names:
          values.append(getattr(instance, attribute_name))
      return values

class Person:
  def __init__(self, forename, surname, email):
      self.forename = forename
      self.surname = surname
      self.email = email

>>> people=[]
>>> p=Person('Petrov','','')
>>> people.append(p)
>>> p=Person('Sidorov','','')
>>> people.append(p)
>>> p=Person(u'Ivanov','','')
>>> people.append(p)
>>> for p in people:
...   print p.forename
Petrov
Sidorov
Ivanov
>>> people.sort(key=SortKey("forename"))
>>> for p in people:
...   print p.forename
Ivanov
Petrov
Sidorov
```

13. Дескриптор

Дескриптор — это класс, который хранит и контролирует атрибуты других классов. Вообще любой класс, который имплементирует один из специальных методов — `__get__ , __set__ , __delete__`, является дескриптором.

Пример:

```
class ExternalStorage:
  __slots__ = ("attribute_name",)
  __storage = {}

  def __init__(self, attribute_name):
      self.attribute_name = attribute_name

  def __set__(self, instance, value):
      self.__storage[id(instance), self.attribute_name] = value

  def __get__(self, instance, owner=None):
      if instance is None:
          return self
      return self.__storage[id(instance), self.attribute_name]
```

```
class Point:
  __slots__ = ()
  x = ExternalStorage("x")
  y = ExternalStorage("y")
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

>>> p1=Point(1,2)
>>> p2=Point(3,4)
```

В данном случае класс Point не имеет собственных атрибутов x, y, хотя вызывает их так, как будто они есть — на самом деле они хранятся в дескрипторе ExternalStorage.

14. Sequence

Последовательность реализуется с помощью методов `__getitem__`, `__setitem__`. В данном примере класс MySequence возвращает по индексу элемент последовательности неопределенной длины, представляющей собой арифметическую прогрессию вида: `1 3 5 7 ...` Здесь нельзя применить стандартные методы `__del__` , `__len__`:

```
class MySequence:
  def __init__(self, start=0, step=1):
      self.start = start
      self.step = step
      self.changed = {}
  def __getitem__(self, key):
      return self.start + key*self.step
  def __setitem__(self, key, value):
      self.changed[key] = value

>>> s = MySequence(1,2)
>>> s[0]
1
>>> s[1]
3
>>> s[100]
201
```

Заключение

Сегодня мы узнали, что классы в питоне имеют большой набор встроенных методов и атрибутов, которые позволяют гибко использовать модель объектно-ориентированного программирования и упрощают решение стандартных задач и алгоритмов. Методы могут быть статическими в зависимости от природы объекта, что позволяет смешивать объектно-ориентированную и функциональную архитектуру. Вызов методов базового класса имеет собственную семантику. Последовательности и мапы, реализованные на базе итераторов, экономят ресурсы и память. Проперти упрощают сложную реализацию атрибутов класса. Функторы обращаются с коллекцией объектов пользовательского типа так, как будто это стандартные типы. Дескрипторы реализуют различную логику хранения атрибутов класса. В продолжение цикла расскажем о работе с файловой системой средствами Python.

Код примеров проверен на версии питона 2.6.

< Предыдущая статья. Следующая статья >

Об авторе

Сергей Яковлев

Яковлев Сергей — независимый разработчик с многолетним опытом прикладного и системного программирования; вносит вклад в развитие open-source на своем персональном сайте www.iakovlev.org. Консультант.