Питон
Курс лекций
Лекция одиннадцатая

© Зайцев Вадим Валерьевич, 2002–2010,
spider.vz@gmail.com

6.2 Инкапсуляция

Инкапсуляция позволяет скрывать детали реализации, на являющиеся важными для использования объекта. Как писал Ингаллс, автор объектно-ориентированного языка программирования Смолток: «Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части».

Абстракция и инкапсуляция взаимодополняют друг друга: абстрагирование направлено на придание смысла внешним свойствам объекта, а инкапсуляция устраняет влияние внутренней реализации. Инкапсуляция делает границы между абстракциями более чёткими, не позволяя абстракциям высокого уровня вмешиваться в работу абстракция более низкого уровня (а обратное влияние невозможно в принципе).

Практическое следствие объединения инкапсуляции с абстракцией означает всего лишь вот что: любой класс можно мысленно разделить на две части: интерфейс и реализацию. *Интерфейс* описывает абстракцию поведения объектов данного класса, он отражает внешнее поведение. *Реализация* же воплощает это поведение, описывая алгоритмы достижения желаемого.

Гради Буч считает, что такой принцип разделения «соответствует сути вещей», потому что интерфейс собирает всё, связанное с взаимодействием данного объекта с любыми другими, а реализация успешно скрывает все детали.

Итак, дадим такое определение: *Инкапсуляция — это процесс отделения интерфейса объекта, отражающего его внешнее поведение, от реализации, описывающей собственно алгоритмы достижения желаемого поведения объекта. Инкапсуляция изолирует контрактные обязательства абстракции от их реализации.*

Соблюдение инкапсуляции в питоне почты полностью является ответственностью программиста. Однако некоторые средства, помогающие ему, присутствуют. Согласно сложившейся традиции, свойства или методы

(то есть переменные области имён объекта) делятся на свободные, личные, скрытые и служебные. Сорт определяется именем.

- Свободные свойства и методы — это основная их масса. Они не имеют никаких ограничений на имена. Доступ к ним может получить кто угодно, хотя прямой доступ к свойствам другого объекта не считается правильным стилем программирования. Для прямого низкоуровневого общения со свойством обычно пишутся две функции: возвращающая (accessor или getter) и изменяющая (mutator или setter). Вот как это может выглядеть:

  ```
  class A:
   x=1
   def getX(self):return x
   def setX(self,n):x=n
  ```

- Личные свойства и методы содержат в начале имени один символ подчёркивания. Доступ к ним интерпретатором не запрещается, но подчёркивание подразумевает некоторую степень служебности. Поэтому такие имена обычно даются переменным, к которым вообще не предполагается никакого доступа извне, то есть переменным, являющимся частью внутренней структуры (инкапсуляция реализации абстракции). Например, как в данном коде:

  ```
  class B:
  ␣_d=''
  ␣def␣add(self,s):self._d+='␣'+s
  ␣def␣get(self):return␣self._d.split('␣')\leftbr1:\rightbr
  ```

- Скрытые свойства и методы содержат не меньше двух подчёркиваний в начале имени и не больше одного — в конце. Их инкапсуляцией занимается интерпретатор питона, но делает это достаточно своеобразно. Получить доступ к скрытой переменной можно, но для этого нужно знать имя класса, экземпляром которого является объект-носитель.

  ```
  class␣C:
  ␣def␣__init__(self):self.__x=0
  ␣def␣add(self,n):self.__x+=n
  ␣def␣sub(self,n):self.__x-=n
  ␣def␣write(self):print␣self.__x
  c=C()
  ```

  При попытке обратиться к переменной `c.__x` мы получим ошибку AttributeError. Такой переменной нет, есть переменная `ci._C__x` — реальное имя создаётся из символа подчёркивания, имени класса и

исходного имени. При этом подпрограммы объекта могут использовать краткую форму имени без указания имени класса (потому как задаются непосредственно внутри него). Это не относится к приобретённым методам (то есть методам, присвоенным классу или объекту уже после завершения его создания):

```
def␣ext(a,b):␣a.__x=b
class␣C:
␣def␣__init__(self):self.__x=0
␣def␣add(self,n):self.__x+=n
␣def␣sub(self,n):self.__x-=n
␣def␣write(self):print␣self.__x
c=C()
C.set=ext
```

В данном случае вызов, например, `c.set(10)` не приведёт к изменению переменной `c._C__x`, потому что будет изменять `c.__x` — ту переменную, непреобразованное имя которой содержит исходный текст функции. Будьте осторожны при работе со скрытыми свойствами, а не то они могут оказаться скрытыми даже от вас самих. Только подпрограммы класса, заданные при его описании, имеют право использовать краткую форму записи имени скрытого свойства.

- Служебные свойства и методы начинаются и завершаются двумя символами подчёркивания, как, например, уже используемые нами конструктор `__init__` и деструктор `__del__`. Прочие служебные свойства включают:
  - **Представление объекта строкой:** `__str__` и `__repr__` будут использоваться при запуске стандартных питоновских функций `str` и `repr` соответственно, предлагая две возможности краткого представления объекта строкой.
  - **Сравнение с другими объектами:** `__lt__` (меньше), `__le__` (меньше либо равно), `__eq__` (равно), `__ne__` (не равно), `__gt__` (больше) и `__ge__` (больше либо равно) задают правила сравнения данного объекта с другими объектами (на обязательно того же класса!). Все эти функции должны возвращать булево значение (во всяком случае, то, что они будут возвращать, будет *трактоваться* именно как булев тип). Вместо них можно задать одну обобщённую `__cmp__`, возвращающую отрицательное число, если другой объект превосходит наш, положительное, если базовый объект превосходит чужой, и ноль в случае их равенства. Таким образом, если программист запишет `x<y`, на самом деле будет выполнено `x.__lt__(y)`.

– **Размер объекта и проверка на ложность:** как мы уже знаем, ложными считаются нули, пустые строки, последовательности и объект `None`. Для определения проверки на пустоту объекта используется функция `__nonzero__`, возвращающая значение булева типа. Если такая функция не определена, вместо неё будет использоваться функция `__len__`, возвращающая длину объекта в каких-то единицах измерения. Внимание, если ни одна из этих функций не определена, все объекты данного класса будут считаться истинными! Конечно, `__len__` имеет и прямое применение (через стандартную функцию `len`).

– **Использование объекта как функции:** в питоне можно обращаться с некоторыми объектами как с функциями, при этом вызов такого вида: `x(arg1,arg2,...)` будет заменён на вызов `x.__call__(arg1,arg2,...)`

– **Использование объекта как последовательности:** аналогичным образом существуют служебные методы, вызывающиеся при попытке взять элемент объекта как массива: x[n]. Они включают: `__getitem__` для взятия элемента (приведённый пример будет на самом деле выполнен как `x.__getitem__(n)`), `__setitem__` для изменения значения элемента присваиванием и `__delitem__` для тех случаев, когда описываемая объектом последовательность поддерживает удаление элементов. Также вводятся функции `__getslice__`, `__setslice__` и `__delslice__` для сечений. Важным атрибутом последовательности может стать служебный метод `__contains__`, возвращающие истинное значение, если его аргумент принадлежит последовательности и ложное в противном случае.

Вообще, следует помнить слова Бьёрна Страуструпа: «инкапсуляция защищает от ошибок, но не от жульничества».