Эмуляция защиты атрибутов экземпляра: часть 1

В следующем коде (private0.py) демонстрируется другой сценарий использования таких инструментов. Он обобщает предыдущий пример, чтобы позволить каждому подклассу иметь собственный список закрытых имен, которым экземпляры не могут присваивать значения (и применяет определяемый пользователем класс исключения, что обсуждается в части VII):

```
class PrivateExc(Exception): pass           # Исключения подробно
                                            #   рассматриваются в части VII

class Privacy:
    def __setattr__(self, attrname, value): # Вызывается для
                                            #   self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)  # Сгенерировать определяемое
                                            #   пользователем исключение
        else:
            self.__dict__[attrname] = value   # Избежать зацикливания,
                                            #   используя ключ словаря

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'       # Чтобы сделать лучше,
                                            #   обратитесь в главу 39!

if __name__ == '__main__':
    x = Test1()
    y = Test2()

    x.name = 'Bob'                          # Работает
    #y.name = 'Sue'                         # Терпит неудачу
    print(x.name)

    y.age = 30                              # Работает
    #x.age = 40                             # Терпит неудачу
    print(y.age)
```

Фактически это первое пробное решение для реализации *защиты атрибутов* в Python – запрет вносить изменения в имена атрибутов за пределами класса. Хотя Python не поддерживает закрытые объявления как таковые, методики вроде показанной здесь способны эмулировать большую часть их предназначения.

Тем не менее, решение получилось неполное и даже неуклюжее. Чтобы сделать его более эффективным, мы должны дополнить классы возможностью устанавливать свои закрытые атрибуты более естественным путем, не проходя каждый раз через __dict__, как обязан поступать конструктор во избежание запуска метода __setattr__ и генерации исключения. Лучший и более совершенный подход может требовать класса-оболочки ("посредника") для препятствования операциям доступа к закрытым атрибутам, выполняемым только за пределами класса, а также метода __getattr__ для проверки операций извлечения атрибутов.

Мы отложим полное решение по защите атрибутов до главы 39, где будем использовать *декораторы классов* для более общего перехвата и проверки атрибутов. Однако, несмотря на то, что защиту атрибутов можно эмулировать подобным образом, на практике так почти никогда не поступают. Программисты на Python в состоянии