Конструктор. Unittest

В предыдущей работе для задания начальных значений атрибутов использовался специальный метод инициализации __init__(self), который вызывался неявным образом при вызове конструктора — метода, имя которого совпадает с названием класса BankAccount.

Переопределив специальный метод __new__(), можно контролировать процесс создания экземпляра класса. Этот метод вызывается до метода __init__(self) и должен вернуть новый экземпляр, либо None (тогда будет вызван __new__() родительского класса). Метод __new__() используется для управления созданием неизменчивых (immutable) объектов.

В Python имеется возможность определять функции с переменным количеством параметров. Применительно к методу инициализации __init__(self) это означает, что нет необходимости создавать несколько методов с разным количеством параметров. Изменим в классе BankAccount описание метода __init__(self) на следующее.

```
    def __init__(self, number = 0, balance = 0):
        self.number = number
        self.balance = balance
```

Передавать значения атрибутов можно через позиционные параметры.

```
ba = BankAccount(1, 100)
```

За счёт передачи аргументов по-умолчанию значения параметров можно не указывать.

```
ba1 = BankAccount()
ba2 = BankAccount(2)
ba3 = BankAccount(3, 50)
```

Можно использовать именованные параметры.

```
ba = BankAccount(balance = 100, number = 1)
```

Для отображения текущего состояния счёта добавим в класс ещё метод __str__(self), который используется для строкового представления объекта. В строке используются два позиционных параметра, на место которых будут подставлены self.number и self.balance.

```
    def __str__(self):
        return 'number: {0}, balance: {1}'.format(self.number, self.balance)
```

В Python классы не являются чем-то статическим, поэтому атрибуты можно добавить к нему в любом месте и в любое время. Например, добавим поле, хранящее текущую дату.

```
    from datetime import date
    ba.date = date.today()
    print(ba.date)
```

В классе BankAccount можно указывать любой номер счёта. Добавим счётчик для номера банковского счёта. Для этого добавим за пределами класса глобальную переменную _next и функцию _next_number(), которая будет увеличивать её на единицу.

```
_next = 0

def _next_number():
    global _next
    _next += 1
```

```
    return _next
```

Из параметров специального метода __init__() удалим number, а в атрибут self.number занесём результат вызова функции _next_number().

```
    def __init__(self, balance = 0):
        self._number = _next_number()
        self._balance = balance
```

Рассмотрим возможность объекта с помощью метода изменять данные другого объекта. Для этого создадим метод по переводу средств с одного счёта на другой.

```
    def transfer_from(self, account, amount):
        account.withdraw(amount)
        self.deposit(amount)
```

Теперь создадим два банковский счёта и выполним перевод средств.

```
    ba1 = BankAccount(100)
    ba2 = BankAccount(100)
    print(ba1)
    print(ba2)
    ba1.transfer_from(ba2, 50)
    print(ba1)
    print(ba2)
```

Добавим возможность пользователю вводить данные при работе с банковским счётом. Для этого не требуется создание отдельного класса, так что добавим несколько функций в модуль.

```
def test_deposit(account):
    print(account)
    amount = int(input('enter amount to deposit on number
{0}:'.format(account.number)))
    account.deposit(amount)
    print(account)
```

В этом методе сначала выводится информация о текущем состоянии банковского счёта, далее в переменную amount заносится введённое пользователем значение путём преобразования строки в число, вызывается метод deposit() и опять выводится информация о текущем состоянии счёта.

```
    ba = BankAccount(100)
    test_deposit(ba)
```

Модуль unittest входит в стандартную библиотеку Python и служит базовым инструментом для организации регрессионных unit-тестов. Создадим класс с набором тестов, проверяющих методы класса BankAccount по созданию счета, добавление и снятие денег в отдельном модуле.

```
from labs_oop import BankAccount
import unittest

class TestBankAccount(unittest.TestCase):

    def setUp(self):
        self.account = BankAccount(100)
```

```
    def test_account_deposit(self):
        test_balance = 150
        self.account.deposit(50)
        self.assertEqual(self.account.balance, test_balance)

if __name__ == '__main__':
    unittest.main()
```

Импортируем BankAccount из модуля labs_oop и классы для проведения модульного тестирования из unittest.

В языке Python можно поместить требуемые определения в файл и использовать их в модулях или в интерактивном режиме интерпретатора. Определения из модуля могут быть импортированы в других модулях. Модуль — это файл, содержащий определения и операторы Python. Именем файла является имя модуля с добавленным суффиксом .py. Внутри модуля, имя модуля (в качестве строки) доступно в виде значения глобальной переменной с именем __name__.

Метод setUp() – служебный. Он вызывается перед запуском каждого теста и подготавливает среду выполнения. В нашем случае метод setUp() создаёт банковский счёт и помещает на счёт 100 единиц. Имена остальных методов начинаются с «test» (необходимое условие для нахождения тестов в коде модуля).

Задания:

- используйте при описании конструкторов значения по умолчанию;
- добавьте в описание класса метод __str__() для строкового представления значений атрибутов экземпляра класса;
- используйте генерацию идентификаторов для классов, согласно индивидуального задания;
- разработайте тесты для проведения модульного тестирования созданных методов в этой работе: конструкторов по умолчанию, строкового представления значений и генерации идентификаторов.