Модуль 3

Урок 5. Наследование классов

Поймёте, в каких случаях нужно применять наследование классов, и научитесь переопределять методы и атрибуты.

Наследование

Наследование – важная составляющая объектно-ориентированного программирования. Так или иначе мы уже сталкивались с ним, ведь объекты наследуют атрибуты своих классов. Однако обычно под наследованием в ООП понимается наличие классов и подклассов. Также их называют супер- или надклассами и классами, а также родительскими и дочерними классами.

Суть наследования здесь схожа с наследованием объектами от классов. Дочерние классы наследуют атрибуты родительских, а также могут переопределять атрибуты и добавлять свои.

Простое наследование методов родительского класса

В качестве примера рассмотрим разработку класса столов и его двух подклассов – кухонных и письменных столов. Все столы, независимо от своего типа, имеют длину, ширину и высоту. Пусть для письменных столов важна площадь поверхности, а для кухонных – количество посадочных мест. Общее вынесем в класс, частное – в подклассы.

Связь между родительским и дочерним классом устанавливается через дочерний: родительские классы перечисляются в скобках после его имени.

```
class Table:
    def __init__(self, l, w, h):
        self.lenght = l
        self.width = w
        self.height = h

class KitchenTable(Table):
    def setPlaces(self, p):
        self.places = p

class DeskTable(Table):
    def square(self):
        return self.width * self.length
```

В данном случае классы KitchenTable и DeskTable не имеют своих собственных конструкторов, поэтому наследуют его от родительского класса. При создании экземпляров этих столов, передавать аргументы для __init__() обязательно, иначе возникнет ошибка:

```
from test import *
```

```
>>> t1 = KitchenTable()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 3 required positional arguments: 'l', 'w', and 'h'
>>> t1 = KitchenTable(2, 2, 0.7)
>>> t2 = DeskTable(1.5, 0.8, 0.75)
>>> t3 = KitchenTable(1, 1.2, 0.8)
```

Несомненно можно создавать столы и от родительского класса Table. Однако он не будет, согласно неким родственным связям, иметь доступ к методам setPlaces() и square(). Точно также как объект класса KitchenTable не имеет доступа к единоличным атрибутам сестринского класса DeskTable.

```
>>> t4 = Table(1, 1, 0.5)
>>> t2.square()
1.2000000000000002
>>> t4.square()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Table' object has no attribute 'square'
>>> t3.square()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'KitchenTable' object has no attribute 'square'
```

В этом смысле терминология "родительский и дочерний класс" не совсем верна. Наследование в ООП – это скорее аналог систематизации и классификации наподобие той, что есть в живой природе. Все млекопитающие имеют четырехкамерное сердце, но только носороги – рог.

Полное переопределение метода надкласса

Что если в подклассе нам не подходит код метода его надкласса. Допустим, мы вводим еще один класс столов, который является дочерним по отношению к DeskTable. Пусть это будут компьютерные столы, при вычислении рабочей поверхности которых надо отнимать заданную величину. Имеет смысл внести в этот новый подкласс его собственный метод square():

```
class ComputerTable(DeskTable):
    def square(self, e):
        return self.width * self.length - e
```

При создании объекта типа ComputerTable по-прежнему требуется указывать параметры, так как интерпретатор в поисках конструктора пойдет по дереву наследования сначала в родителя, а потом в прародителя и найдет там метод __init__().

Однако когда будет вызываться метод square(), то поскольку он будет обнаружен в самом ComputerTable, то метод square() из DeskTable останется невидимым, т. е. для объектов класса ComputerTable он окажется переопределенным.

Результат:

```
>>> from test import ComputerTable
>>> ct = ComputerTable(2, 1, 1)
>>> ct.square(0.3)
1.7
```

Дополнение, оно же расширение, метода

Если посмотреть на вычисление площади, то часть кода надкласса дублируется в подклассе. Этого можно избежать, если вызвать родительский метод, а потом дополнить его:

```
class ComputerTable(DeskTable):
    def square(self, e):
        return DeskTable.square(self) - e
```

Здесь вызывается метод другого класса, а потом дополняется своими выражениями. В данном случае вычитанием.

Рассмотрим другой пример. Допустим, в классе KitchenTable нам не нужен метод, поле places должно устанавливаться при создании объекта в конструкторе. В классе можно создать собсвенный конструктор с чистого листа, чем переопределить родительский:

```
class KitchenTable(Table):
    def __init__(self, l, w, h, p):
        self.length = l
        self.width = w
        self.height = h
        self.places = p
```

Однако это не лучший способ, если дублируется почти весь конструктор надкласса. Проще вызвать родительский конструктор, после чего дополнить своим кодом:

```
class KitchenTable(Table):
    def __init__(self, l, w, h, p):
        Table.__init__(self, l, w, h)
        self.places = p
```

Теперь при создании объекта типа KitchenTable надо указывать в конструкторе четыре аргумента. Три из них будут переданы выше по лестнице наследования, а четвертый оприходован на месте.

Результат:

```
>>> tk = KitchenTable(2, 1.5, 0.7, 10)
```

```
>>> tk.places
10
>>> tk.width
1.5
```

Задание:

Разработайте программу по следующему описанию.

В некой игре-стратегии есть солдаты и герои. У всех есть свойство, содержащее уникальный номер объекта, и свойство, в котором хранится принадлежность команде. У солдат есть метод «иду за героем», который в качестве аргумента принимает объект типа «герой». У героев есть метод увеличения собственного уровня.

В основной ветке программы создается по одному герою для каждой команды. В цикле генерируются объекты-солдаты. Их принадлежность команде определяется случайно. Солдаты разных команд добавляются в разные списки. Каждая команда должна «следовать» за свои героем.

Измеряется длина списков солдат противоборствующих команд и выводится на экран. У героя, принадлежащего команде с более длинным списком, поднимается уровень. Уровень обоих героев и списки команд вывести на экран.