Конспект: Наследование, инкапсуляция и полиморфизм

1. Инкапсуляция

В Python нет строгой инкапсуляции, как в некоторых других языках (например, Java или C++). Однако существуют соглашения и механизмы, которые помогают достичь похожего эффекта.

- Мы используем защищенные и приватные атрибуты (и методы) для затруднения доступа к ним. К сожалению, доступ все-равно возможен, и это больше работает как сигнал другим разработчикам – не трогайте эти атрибуты
- Мы используем геттеры и сеттеры для умного контроля доступа к атрибутам

Пример:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner  # Публичный атрибут
        self._balance = balance  # Защищенный атрибут
        self.__pin = 1234  # Приватный атрибут (PIN-код)

    # Геттер для защищенного атрибута _balance
    @property
    def balance(self):
        return self._balance

    # Сеттер для защищенного атрибута _balance
    @balance.setter
    def balance(self, value):
        if not isinstance(value, (int, float)):
            print("Balance must be a number")
        elif value < 0:
            print("Balance cannot be negative")
        else:
            self._balance = value
```

```
    # Метод для вывода баланса
    def display_balance(self, pin):
        if pin != self.__pin:
            print("Incorrect PIN")
        else:
            print(f"Balance: {self._balance}")

    # Метод для смены PIN-кода
    def change_pin(self, old_pin, new_pin):
        if old_pin != self.__pin:
            print("Incorrect old PIN")
        else:
            self.__pin = new_pin
            print("PIN changed successfully")
```

2. Наследование

Наследование помогает переиспользовать имеющийся код, изменять ненужное поведение на нужное и выстраивать иерархию классов

Пример:

```
# Базовый класс для дебетового счета
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner  # Владелец счета
        self.balance = balance  # Баланс

    # Метод для пополнения счета
    def deposit(self, amount):
        if amount <= 0:
            print("Deposit amount must be positive")
        else:
            self.balance += amount
            print(f"Deposited {amount}. New balance: {self.balance}")

    # Метод для снятия средств
    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive")
        elif amount > self.balance:
            print("Insufficient funds")
```

```
        else:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance: {self.balance}")

    # Метод для вывода информации о счете
    def info(self):
        return f"Owner: {self.owner}, Balance: {self.balance}"

# Дочерний класс для кредитного счета
class CreditAccount(Account):
    def __init__(self, owner, balance=0, credit_limit=1000):
        super().__init__(owner, balance)  # Вызов конструктора родительского
класса
        self.credit_limit = credit_limit  # Лимит кредита

    # Переопределение метода снятия средств
    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive")
        if amount > self.balance + self.credit_limit:
            print("Exceeds credit limit")
        else:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance: {self.balance}")
```

В Python возможно множественное наследование и mro (порядок разрешения методов, Method Resolution Order) помогает не запутаться

Пример:

```
class User:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def show(self):
        print(f'Я {self.name}. Мой телефон {self.phone}')

class Friend(User):
    def show(self):
        print(f'Мой друг {self.name}. Его телефон {self.phone}')

class Spider:
```

```
    def show(self):
        print(f'Я паук')

    def throw_web(self):
        print('Вжуууууух!')

class SpiderMan(Spider, Friend):
    pass

timur = User('Тимур', 9345326536)
oleg = Friend('Олег', 234235236523)
piter = SpiderMan('Питер', None)
timur.show()
oleg.show()
piter.throw_web()
piter.show()
print(SpiderMan.mro())
```

3. Полиморфизм

- Полиморфизм, реализованный через наследование и переопределение методов

Пример:

```
# Базовый класс
class Animal:
    def speak(self):
        return "Some generic animal sound"

# Подкласс Dog
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Подкласс Cat
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Функция, принимающая любой объект типа Animal
def make_animal_speak(animal):
    print(animal.speak())
```

```
# Создание объектов
dog = Dog()
cat = Cat()
# Вызов функции
make_animal_speak(dog)  # Woof!
make_animal_speak(cat)  # Meow!
```

- Перегрузка операторов через магические методы

Пример:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # Переопределение оператора +
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):  # Переопределение вывода объекта
        return f"Point({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2  # Используется переопределённый метод __add__
print(p3)  # Point(4, 6)
```

- Утиная типизация

Пример:

```
class Duck:
    def quack(self):
        return "Quack!"

class Person:
    def quack(self):
        return "I'm pretending to be a duck!"

def make_quack(obj):
    print(obj.quack())

duck = Duck()
person = Person()
```

```
make_quack(duck)    # Quack!
make_quack(person)  # I'm pretending to be a duck!
```

- Пример комплексного использования полиморфизма

Пример:

```
from math import radians, sin, cos, acos


class Point:
    def __init__(self, latitude, longitude):
        self.longitude = radians(longitude)
        self.latitude = radians(latitude)

    def distance(self, other):  # other - другой объект класса Point
        cos_d = sin(self.latitude) * sin(other.latitude) \
                + cos(self.latitude) * cos(other.latitude) * cos(self.longitude -
other.longitude)
        return round(6371 * acos(cos_d), 2)

    def __sub__(self, other):
        cos_d = sin(self.latitude) * sin(other.latitude) \
                + cos(self.latitude) * cos(other.latitude) * cos(self.longitude -
other.longitude)
        return round(6371 * acos(cos_d), 2)

class City(Point):
    def __init__(self, latitude, longitude, name, population):
        super().__init__(latitude, longitude)
        self.name = name
        self.population = population

    def __str__(self):
        return f'Город: {self.name}, население: {self.population}'

class Mountain(Point):
    def __init__(self, latitude, longitude, name, height):
        super().__init__(latitude, longitude)
        self.name = name
        self.height = height

    def __str__(self):
        return f'Гора: {self.name}, высота: {self.height}'
```

```
moscow = City(55, 37, 'Москва', 14_000_000)
everest = Mountain(27, 86, 'Эверест', 8849)
print(moscow)
print(everest)
print(moscow.distance(everest))
print(moscow - everest)
print(everest - moscow)
```