Python

ООП

Инкапсуляция

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его

```
class Person:
    def __init__(self, name):
        self.name = name    # устанавливаем имя
        self.age = 1        # устанавливаем возраст

    def display_info(self):
        print("Имя:", self.name, "\tВозраст:", self.age)


tom = Person("Tom")
tom.name = "Человек-паук"       # изменяем атрибут name
tom.age = -129                  # изменяем атрибут age
tom.display_info()              # Имя: Человек-паук     Возраст: -129
```

Инкапсуляция

Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода.

Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются свойствами.

Инкапсуляция

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: self.__name. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса.

```
1 tom.__age = 43
```

Меняем на

```
1 def get_age(self):
2     return self.__age
```

Данный метод еще часто называют геттер или аксессор.

Здесь мы уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод еще называют сеттер или мьютейтор (mutator).

```
1  def set_age(self, value):
2      if value in range(1, 100):
3          self.__age = value
4      else:
5          print("Недопустимый возраст")
```

Аннотации свойств

Для создания свойства-геттера над свойством ставится аннотация @property.

Для создания свойства-сеттера над свойством устанавливается аннотация имя_свойства_геттера.setter.

```python
class Person:
    def __init__(self, name):
        self.__name = name  # устанавливаем имя
        self.__age = 1      # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
    def name(self):
        return self.__name
```

```
    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)


tom = Person("Tom")

tom.display_info()      # Имя: Tom  Возраст: 1
tom.age = -3486         # Недопустимый возраст
print(tom.age)          # 1
tom.age = 36
tom.display_info()      # Имя: Tom  Возраст: 36
```

Наследование

Наследование позволяет создавать новый класс на основе уже существующего класса

Подкласс наследует от суперкласса все публичные атрибуты и методы.

Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

```python
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

```

```python
class Employee(Person):

    def details(self, company):
        # print(self.__name, "работает в компании", company) # так нельзя, self.__name - приватный атрибут
        print(self.name, "работает в компании", company)
```

Наследование

Класс Employee полностью перенимает функционал класса Person и в дополнении к нему добавляет метод details().

Стоит обратить внимание, что для Employee доступны через ключевое слово self все методы и атрибуты класса Person, кроме закрытых атрибутов типа __name или __age.

При создании объекта Employee мы фактически используем конструктор класса Person. И кроме того, у этого объекта мы можем вызвать все методы класса Person.

Полиморфизм

Предполагает способность к изменению функционала, унаследованного от базового класса.

```
25 class Employee(Person):
26     # определение конструктора
27     def __init__(self, name, age, company):
28         Person.__init__(self, name, age)
29         self.company = company
30 
31     # переопределение метода display_info
32     def display_info(self):
33         Person.display_info(self)
34         print("Компания:", self.company)
35 
```

```python
class Student(Person):
    # определение конструктора
    def __init__(self, name, age, university):
        Person.__init__(self, name, age)
        self.university = university

    # переопределение метода display_info
    def display_info(self):
        print("Студент", self.name, "учится в университете", self.university)
```

```python
people = [Person("Tom", 23), Student("Bob", 19, "Harvard"), Employee("Sam", 35, "Google")]

for person in people:
    person.display_info()
    print()
```

Полиморфизм

В основной части программы создается список из трех объектов Person, в котором два объекта также представляют классы Employee и Student. И в цикле этот список перебирается, и для каждого объекта в списке вызывается метод display_info. На этапе выполнения программы Python учитывает иерархию наследования и выбирает нужную версию метода display_info() для каждого объекта.

```
Имя: Tom    Возраст: 23

Студент Bob учится в университете Harvard

Имя: Sam    Возраст: 35
Компания: Google
```