

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

```
1 class Person:
2     def __init__(self, name):
3         self.name = name      # устанавливаем имя
4         self.age = 1         # устанавливаем возраст
5
6     def display_info(self):
7         print("Имя:", self.name, "\tВозраст:", self.age)
8
9
10    tom = Person("Том")
11    tom.name = "Человек-паук"    # изменяем атрибут name
12    tom.age = -129              # изменяем атрибут age
13    tom.display_info()          # Имя: Человек-паук      Возраст: -129
```

Инкапсуляция

Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объекта 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`.

```
1 class Person:
2     def __init__(self, name):
3         self.__name = name # устанавливаем имя
4         self.__age = 1     # устанавливаем возраст
5
6     @property
7     def age(self):
8         return self.__age
9
10    @age.setter
11    def age(self, age):
12        if age in range(1, 100):
13            self.__age = age
14        else:
15            print("Недопустимый возраст")
16
17    @property
18    def name(self):
19        return self.__name
20
```

```
21     def display_info(self):
22         print("Имя:", self.__name, "\tВозраст:", self.__age)
23
24
25 tom = Person("Том")
26
27 tom.display_info()           # Имя: Том  Возраст: 1
28 tom.age = -3486             # Недопустимый возраст
29 print(tom.age)              # 1
30 tom.age = 36
31 tom.display_info()          # Имя: Том  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).

```
1 class Person:
2     def __init__(self, name, age):
3         self.__name = name # устанавливаем имя
4         self.__age = age # устанавливаем возраст
5
```

```
25 class Employee(Person):
26
27     def details(self, company):
28         # print(self.__name, "работает в компании", company) # так нельзя, self.__name - приватный атрибут
29         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
```

```
37 class Student(Person):
38     # определение конструктора
39     def __init__(self, name, age, university):
40         Person.__init__(self, name, age)
41         self.university = university
42
43     # переопределение метода display_info
44     def display_info(self):
45         print("Студент", self.name, "учится в университете", self.university)
```

```
47 people = [Person("Tom", 23), Student("Bob", 19, "Harvard"), Employee("Sam", 35, "Google")]
48
49 for person in people:
50     person.display_info()
51     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.

```
1  Имя: Том      Возраст: 23
2
3  Студент Bob учится в университете Harvard
4
5  Имя: Sam      Возраст: 35
6  Компания: Google
```