

Объектно-ориентированное программирование

это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. Если постараться объяснить простыми словами, то ООП ускоряет написание кода и делает его более читаемым.

Идеология объектно-ориентированного программирования (ООП) разрабатывалась, чтобы связать поведение определенного объекта с его классом. Людям проще воспринимать окружающий мир как объекты, которые поддаются определенной классификации (например, разделение на живую и неживую природу).

Объектно-ориентированное программирование.....	1
Практическое задание.....	3
Принципы ооп.....	3

Практическое задание

Принципы ооп

Теоретическая часть

Инкапсуляция

Инкапсуляция означает скрытие деталей реализации объекта и предоставление только интерфейса для взаимодействия с ним. Это позволяет изолировать изменения в одной части программы от других частей, что делает код более надежным и устойчивым к изменениям.

Наследование

Наследование позволяет создавать новые классы на основе существующих. Это способствует повторному использованию кода и созданию иерархий классов. Наследование позволяет наследникам использовать свойства и методы предков и переопределять или расширять их, если это необходимо.

Полиморфизм

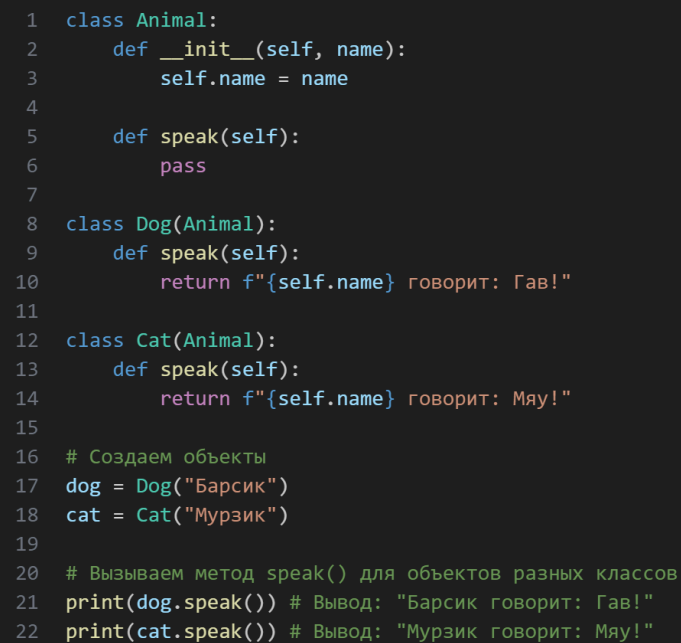
Полиморфизм означает способность объектов разных классов обладать общим интерфейсом. Это позволяет обрабатывать объекты разных типов с помощью общих методов и функций. Полиморфизм делает код более гибким и расширяемым.

Абстракция

Абстракция - это процесс выделения общих характеристик объектов и создание абстрактных классов или интерфейсов для их представления.

Абстракция помогает упростить модель системы, делая её более понятной и управляемой.

Пример:




```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         pass
7
8 class Dog(Animal):
9     def speak(self):
10         return f"{self.name} говорит: Гав!"
11
12 class Cat(Animal):
13     def speak(self):
14         return f"{self.name} говорит: Мяу!"
15
16 # Создаем объекты
17 dog = Dog("Барсик")
18 cat = Cat("Мурзик")
19
20 # Вызываем метод speak() для объектов разных классов
21 print(dog.speak()) # Вывод: "Барсик говорит: Гав!"
22 print(cat.speak()) # Вывод: "Мурзик говорит: Мяу!"
```

Практическая часть

Для начало надо определиться, чем отличаются объекты от класса. Класс это схема, при заполнений этой схемы, создается объект класса. Например, есть класса Human с атрибутами класса `height`, `age`:




Изначально атрибуты класса `height`, `age` со значением `None`. При создание объекта класса они будут наследоваться. Создаем объекты класса `human_one`, `human_two`.



```
1 class Human:
2     height = None
3     age = None
4
5
6 human_one = Human()
7 human_two = Human()
```

И проверим значение наших атрибутов класса в объекте класса.



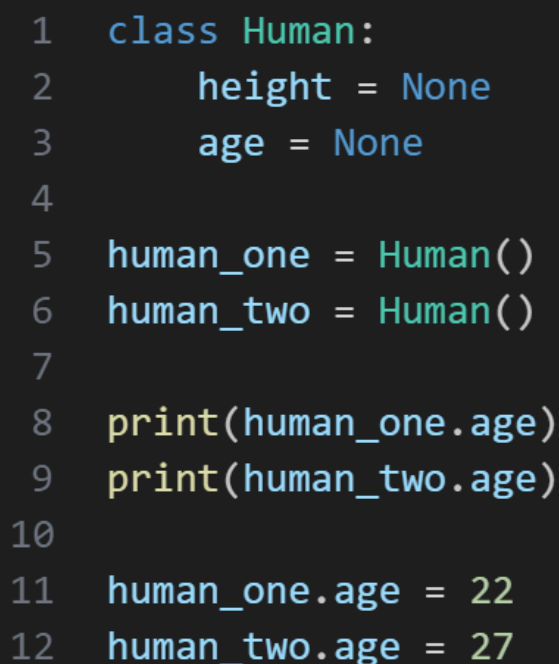
```
1 class Human:
2     height = None
3     age = None
4
5 human_one = Human()
6 human_two = Human()
7
8 print(human_one.age)
9 print(human_two.age)
```

При запуске программы, она будет выводить значение атрибута класса в объекте –

None

None

Также, если в атрибуте класса ничего нету, то не инициализировался (и его как бы не существует)
Присвоим другое значение атрибута класса в объекте. Для это обращаемся к объекту с атрибутами
класса(как выводили атрибут класса) и присваиваем другое значение.




```
1 class Human:
2     height = None
3     age = None
4
5 human_one = Human()
6 human_two = Human()
7
8 print(human_one.age)
9 print(human_two.age)
10
11 human_one.age = 22
12 human_two.age = 27
```

Теперь в атрибуте объекте будут храниться другие значение. Также давайте удалим атрибут
класса height для этого воспользуемся функцией `delattr(obj, name_attr)`, которая принимает:

Первым аргументам: объект класса или сам класс

Вторым аргументам: название атрибута в виде строки

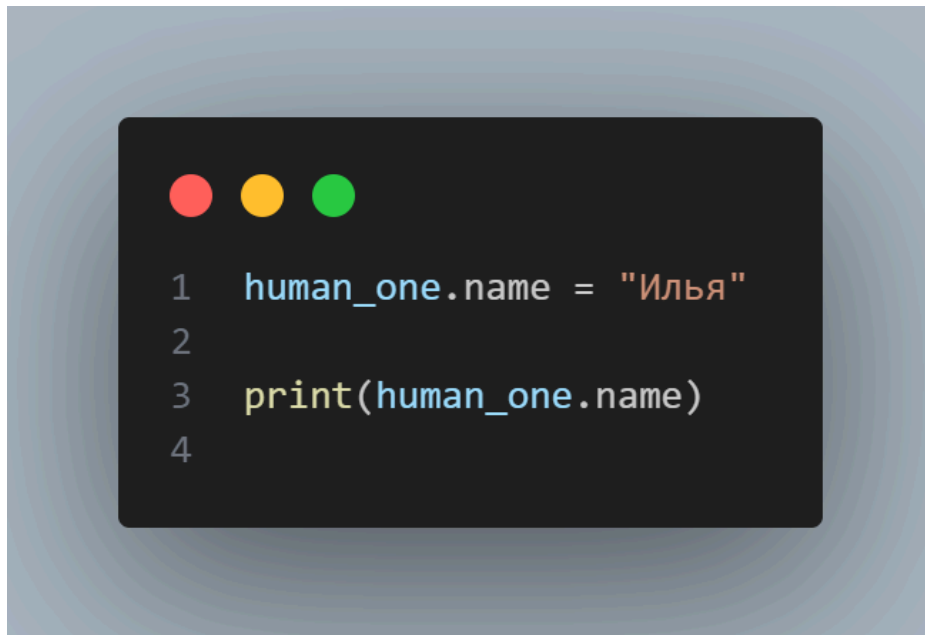
Удаляем атрибут height у обоих объектов, и с помощью функции `hasattr(obj, name_attr)`
проверим существует ли данный атрибут в объекте. При вызове данной функции она возвращает
bool значение.



```
1  class Human:
2      height = None
3      age = None
4
5  human_one = Human()
6  human_two = Human()
7
8  print(human_one.age)
9  print(human_two.age)
10
11 human_one.age = 22
12 human_two.age = 27
13
14 print(human_one.age)
15 print(human_two.age)
16
17 delattr(human_one, 'height')
18 delattr(human_two, 'height')
19
20
21 print(hasattr(human_one, 'height'))
22 print(hasattr(human_two, 'height'))
```

Теперь при вызове функции `hasattr`, она будет возвращаться `False`, потому что данного атрибута нету.

Давайте добавим новый атрибут для объекта класса, например `name`. Это можно сделать с помощью функции `setattr` или обратиться к объекту класса, и через точку написать название атрибута, а также присвоить значение.



Задание

1. Создайте класс Goods с атрибутами класса и значениями:
 - a. title Мороженое
 - b. weight 151
 - c. tp "Еда"
 - d. price 12321Изменить значение атрибута price и weight
2. Создайте пустой класс Car, и добавьте атрибуты со значениями:
 - a. model Тойота
 - b. color черный
 - c. number П34А123
3. Что делает хранит атрибут `__dict__`? (самостоятельно узнать)

Контрольные вопросы

1. напишите функции, которые удаляют атрибут класса, создают атрибут класса, проверяют существует ли атрибут класса и возвращают значение атрибута класса
2. Чем отличается объект от класса ?

Практическое задание

Методы. Параметр self

Теоретическая часть

Методы в Python

Метод — это функция, определенная внутри класса. Методы используются для того, чтобы объекты класса могли выполнять определенные действия или изменять свои внутренние данные (атрибуты). Все методы связаны с конкретным экземпляром класса и работают с его данными через параметр `self`.

Параметр `self`

`self` — это специальный параметр, который всегда передается первым в методы класса. Он является ссылкой на текущий экземпляр класса и используется для доступа к атрибутам и другим методам этого экземпляра. Это ключевой механизм, который позволяет методам "знать", с каким объектом они работают.

Основные моменты о `self`:

1. **Связь с объектом:** Через `self` метод может обращаться к атрибутам и методам конкретного экземпляра класса. Это необходимо, так как каждый объект класса может иметь свои значения атрибутов.
2. **Неявная передача:** Когда вызывается метод объекта, Python автоматически передает ссылку на этот объект как первый аргумент метода — именно в качестве `self`.
3. **Необязательное имя:** Хотя общепринятое имя для этого параметра — `self`, можно использовать любое другое, но следование стандарту помогает делать код понятнее.

Пример использования метода с self

```
1 class Dog:
2     # Инициализация объекта (без магического метода __init__)
3     def set_details(self, name, breed):
4         self.name = name    # Атрибут "name" объекта
5         self.breed = breed  # Атрибут "breed" объекта
6
7     # Метод для вывода информации об объекте
8     def display_details(self):
9         print(f"Dog's name: {self.name}, Breed: {self.breed}")
10
11 # Создание экземпляра класса Dog
12 my_dog = Dog()
13
14 # Вызов метода для установки атрибутов
15 my_dog.set_details("Buddy", "Golden Retriever")
16
17 # Вызов метода для отображения данных
18 my_dog.display_details() # Вывод: Dog's name: Buddy, Breed: Golden Retriever
```

Как работает self в примере:

1. Метод set_details:

- Использует self для установки атрибутов экземпляра (self.name и self.breed).
- Атрибуты объекта становятся доступными для других методов этого объекта.

2. Метод display_details:

- С помощью self получает доступ к атрибутам name и breed, установленных ранее, и выводит их.

Почему важно использовать self?

- **Связь метода с объектом:** Без self методы не могли бы взаимодействовать с состоянием конкретного объекта. Если бы методы не принимали self, они не смогли бы изменять и использовать данные конкретного экземпляра класса.
- **Индивидуальные объекты:** Каждый объект класса может иметь свои значения атрибутов, и через self методы работают с данными конкретного объекта, а не класса в целом.

Пример без использования self (ошибочный подход):

Если убрать self, код перестанет работать, так как Python не сможет определить, с какими именно атрибутами и методами объекта следует взаимодействовать.



```
1 class Dog:
2     # Инициализация объекта (без магического метода __init__)
3     def set_details(self, name, breed):
4         self.name = name    # Атрибут "name" объекта
5         self.breed = breed  # Атрибут "breed" объекта
6
7     # Метод для вывода информации об объекте
8     def display_details(self):
9         print(f"Dog's name: {self.name}, Breed: {self.breed}")
10
11 # Создание экземпляра класса Dog
12 my_dog = Dog()
13
14 # Вызов метода для установки атрибутов
15 my_dog.set_details("Buddy", "Golden Retriever")
16
17 # Вызов метода для отображения данных
18 my_dog.display_details() # Вывод: Dog's name: Buddy, Breed: Golden Retriever
```

Этот пример вызовет ошибку, так как без `self` Python не понимает, к каким атрибутам или методам обращаться.

Практическая часть

В этой практической сделаем два класса. Первый класс будет обрабатывать txt файл с атрибутами и значениями, а затем создавать новый класс с атрибутами и значениями из файла.

Создаем классы UserSchem и DataBase. Класс UserSchem создаем пустым используя оператор pass, в DataBase создаем методы(методы это такая же функция, но внутри класса) get_data, serializers и create_user.

1. метод get_data – возвращать объект файла в кодировке UTF-8
2. метод serializers – обрабатывает объект файла, и возвращает его виде списка со словарями
3. метод create – объекты класс с определенными атрибутами и значениями, которые взяты из файла

Структура txt файла:

```
id 1 name Илья age 23
id 2 name Михаил age 32
id 4 name Дмитрий age 25
```

Метод get_data:

```
1 def get_data(self, url):
2     with open(url, 'r', encoding='UTF-8') as f:
3         result = f.readlines()
4         f.close()
5     return result
```

В методе get_data – с помощью встроенной функции open для взаимодействия с файлами. Функция принимает путь файл, типа взаимодействия(в данном примере – чтение “r”), а также необязательный аргумент это тип кодировки(так как у нас кириллица, используем utf-8). Используем with...as, чтобы определенное значение использовать в некой области, без создания переменной. Используем метод, который возвращают все строки в файле.

Метод serializers:

```
1 def serializers(self, data:TextIOWrapper):
2     content = []
3     for i in data:
4         schema=dict()
5         line = [i for i in re.split(r'\s',i) if i != '']
6         for index in range(0,len(line)-1, 2):
7             schema[line[index]] = line[index+1]
8         content.append(schema)
9     return content
```

Обрабатывает данные из файла в виде словаря с помощью регулярных выражений.

метод create:

```
1 def create(self, data):
2     for i in data:
3         user = UserSchem()
4         for key, item in i.items():
5             setattr(user,key, item)
```

create создает объекты, но нигде не сохраняют. Как вообще сохраняются объекты? Объекты сохраняются, тогда когда они используются в коде.

Задание

1. Измените метод `create`, чтобы он сохранял все созданные объекты в виде списка в атрибуте класса
2. Добавьте метод `search`, который находит определенный объект по атрибутам
3. Создайте класс `Translator` с методами `add`, `remove`, `translate`.
 - `add(self, eng, rus)` – добавляет в словарь(dict) перевод английского слова на русский язык (если перевод слова на английском существует, то добавляет его в список со всеми переводами данного слова. Перевод в списке не должен повторяться). Также создам атрибут класса `tr`, где будет храниться словарь
 - `remove(self, eng)` – удаляет слова из словаря со списком перевода
 - `translate(self, eng)` – возвращает перевод слова с первым значением из списка

Контрольные вопросы

1. Что называется методом класса?
2. Какую роль играет параметр `self` в методах класса?

Практическое задание

Магические методы.

Магический метод `__init__`

Магический метод `__del__`

Магический метод `__new__`

Теоретическая часть

