

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 |
| Теоретическая часть..... | 3 |
| Практическая часть..... | 5 |
| Задание..... | 9 |
| Контрольные вопросы..... | 9 |
| Практическое задание..... | 10 |
| Теоретическая часть..... | 10 |
| Практическая часть..... | 13 |
| Задание..... | 15 |
| Практическое задание..... | 16 |
| Теоретическая часть..... | 16 |
| Практическая часть..... | 19 |
| Практическое задание..... | 22 |
| Теоретическая часть..... | 22 |

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

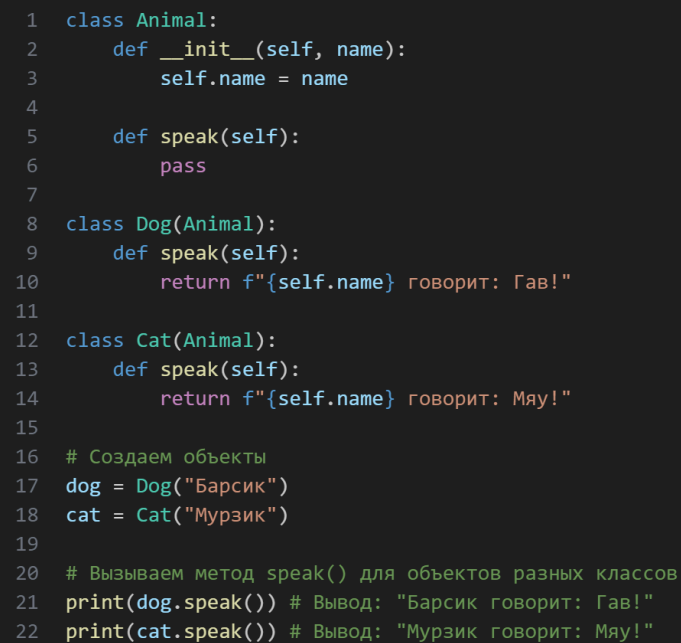
Полиморфизм означает способность объектов разных классов обладать общим интерфейсом. Это позволяет обрабатывать объекты разных типов с помощью общих методов и функций. Полиморфизм делает код более гибким и расширяемым.

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 рост(height), возраст(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.



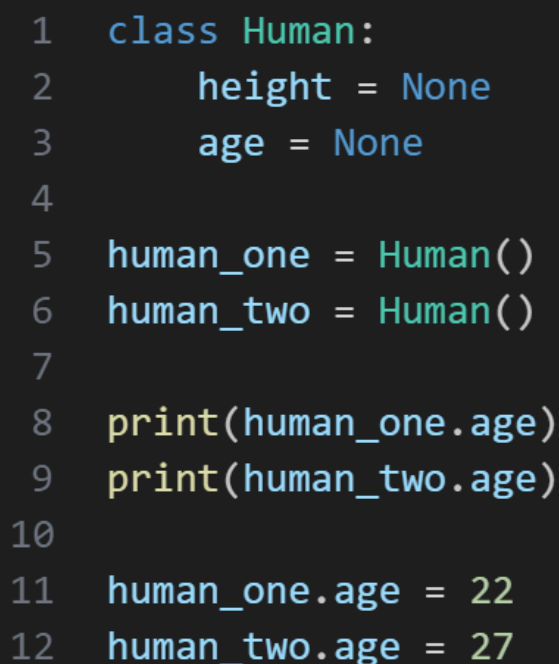
```
1 class Human:
2     height = None
3     age = None
4
5 human_one = Human()
6 human_two = Human()
7
8 print(human_one.age)
9 print(human_two.age)
```

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

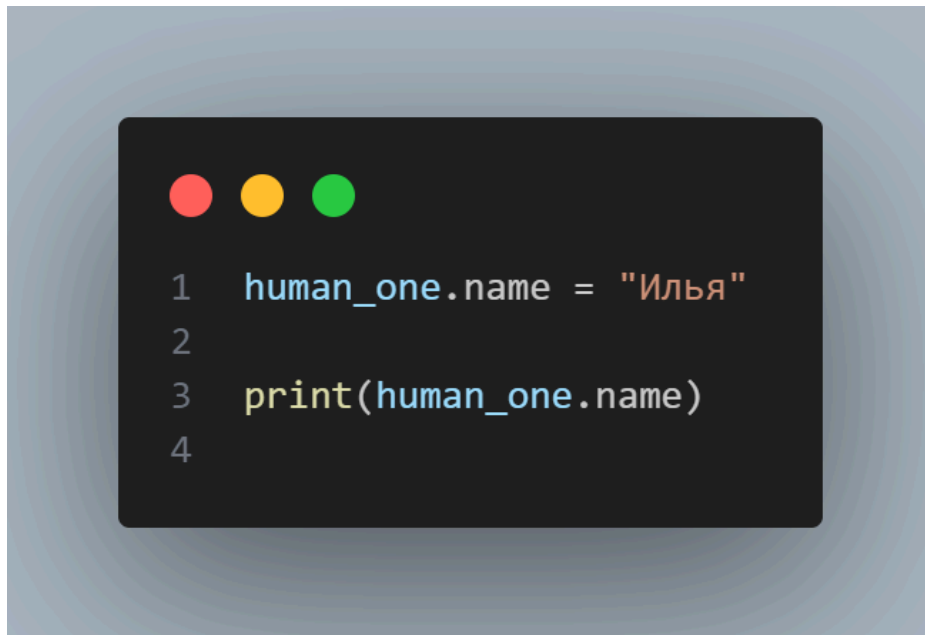
Удаляем атрибут height у обоих объектов, и с помощью функции `hasattr(obj, name_attr)`
проверим существует ли данный атрибут в объекте. При вызове данной функции она возвращает
bool значение.



```
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.



```
1 human_one.name = "Илья"
2
3 print(human_one.name)
4
```

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`.

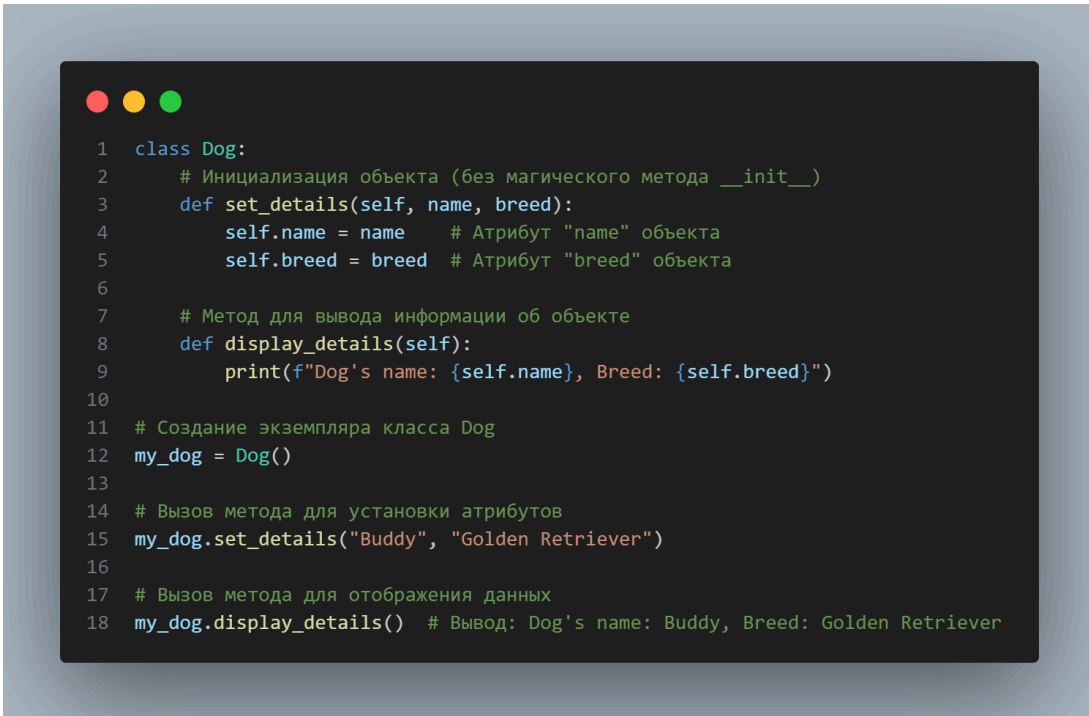
Параметр `self`

`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

A screenshot of a code editor with a dark background and light-colored text. The code defines a class 'Dog' with two methods: 'set_details' and 'display_details'. It also shows the creation of an instance 'my_dog' and the use of both methods. Comments in Russian explain each step. The code is as follows:

```
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.

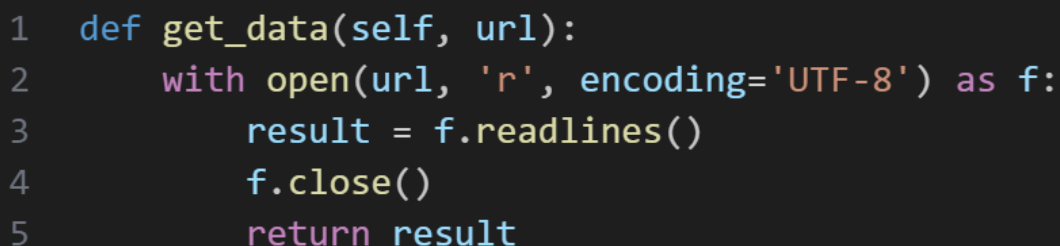
Создаем классы UserSchem и DataBase. Класс UserSchem создаем пустым используя оператор pass, в DataBase создаем методы(методы это такая же функция, но внутри класса) get_data, serializers и create_user.

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__`.

Теоретическая часть

Магические методы в Python, также известные как dunder-методы (от английского "double underscore"), представляют собой специальные методы, которые начинаются и заканчиваются двумя подчеркиваниями (например, `__init__`, `__str__`). Эти методы позволяют вам определять поведение ваших объектов при использовании встроенных функций и операторов. Магические методы делают ваш код более читаемым и позволяют использовать объекты классов так, как если бы они были встроенными типами данных. Например, вы можете использовать магические методы для создания объектов, которые ведут себя как списки или словари, или для определения поведения объектов при выполнении арифметических операций.

Магические методы играют ключевую роль в Python, так как они позволяют разработчикам создавать более интуитивно понятные и гибкие классы. Они позволяют вам переопределять стандартное поведение объектов, что может быть особенно полезно при создании пользовательских структур данных или при работе с объектно-ориентированным программированием. В этой статье мы рассмотрим основные магические методы, их назначение и примеры использования, чтобы вы могли лучше понять, как и когда их использовать.

Метод `__init__`.

Метод `__init__` — это так называемый «инициализатор». Он вызывается после создания объекта и служит для инициализации его атрибутов.



```
1 class Car:
2     def __init__(self, color):
3         self.color = color
4
5 car = Car("red") # создание объекта
6 print(car.color) # вывод: "red"
```

В этом примере метод `__init__` принимает два аргумента: `self` (это обычное название для ссылки на объект, с которым работает метод) и `color`. После создания объекта `car`, `__init__` вызывается автоматически с `car` в качестве `self` и строкой «red» в качестве `color`.

Метод `__del__`.

Метод `__del__` можно определить в классе, и он будет вызван автоматически, когда все ссылки на объект будут удалены или когда программа завершится.



```
1 class MyClass:
2     def __del__(self):
3         print("Удаление экземпляра")
4
5 obj = MyClass() # создание экземпляра
6 del obj # удаление экземпляра, вызывает __del__
```

В этом примере `__del__` просто выводит сообщение при удалении объекта. На практике `__del__` может быть использован для более сложных задач, таких как закрытие открытых файлов, освобождение сетевых ресурсов и т.д.

Практическая часть

В этой практической создадим класс Resource, который будет принимать значение название ресурса и тип ресурса, а также выводить их после удаление объекта.

1. Создаем класс Resource и добавляем магический метод `__init__`. С помощью магического метода `__init__` будем добавлять атрибуты объекта. В метод `__init__` добавляем аргументы `name`(название ресурса) и `resource_type`(тип ресурса). Теперь в самом методе надо присвоить значение из аргументов в сам объект, чтобы присвоить данные в объект используем параметр `self`.

```
1 class Resource:
2
3
4     def __init__(self, name: str, resource_type:str):
5         self.name = name
6         self.resource_type = resource_type
```

2. Создаем объекты класса Resource. Затем в круглы скобках вносим данные.

```
1 r1 = Resource("Соединение1", "подключение к базе данных")
2 r2 = Resource("Соединение2", "подключение к базе данных")
```


(чтобы обратиться к атрибуту объекта, напишите сам объект и через точку название атрибута объекта)

3. Добавляем магический метод `__del__`, который активируется при удалении объекта из памяти. Давайте его переопределим, и сделаем так, чтобы при удалении объекта он выводил данные из атрибутов.



```
1 def __del__(self):
2     print(f"Ресурс {self.name} типа {self.resource_type} удалён.")
```

4. Теперь запускаем файл с объектами класса. После запуска у вас сразу будет выводиться данные об объектах, это потому что объекты нигде не используются и их удаляет сборщик мусора(в python его называют GIL)
5. Давайте попробуем сами удалить объект класса с помощью оператора del. Создаем цикл for, который будет проводить 10 итераций, и на 5 будет удалять объект r2



```
1 for _ in range(1,11):
2     print(_)
3     if _ == 5:
4         del r2
```

После запуска программы в терминале выводится сначала пять чисел, а затем удаляется объект и выводит данные об объекте.

Задание

1. Создайте класс Node с атрибутами объекта data и next.
2. Создайте класс LinkedList.

Атрибутами класса:

- start - хранит начальный объект Node
- end - хранит конечный объект Node

Методы:

- len - возвращает количество связей
- search(data) - возвращает определенный объект из связанного списка по значению
- append(obj) - добавляет новую связь в конце связанного списка и изменяет атрибут класса end
- remove(index) - удаляет определенный объект по индексу, если удаляет первый объект по индексу, то изменяет атрибут класса start

Контрольные вопросы

1. Как работают метод __init__ и __del__.
2. Какая последняя строка вывелась после выполнения программы из практической части?

Практическое задание

Параметры cls.
Магический метод `__new__`.

Теоретическая часть

Метод `__new__`.

Метод `__new__` — это настоящий конструктор в Python. Он вызывается до `__init__` и отвечает за создание (и возврат) нового объекта.

По умолчанию `__new__` создает экземпляр класса и возвращает его, но вы можете переопределить его для более сложного поведения. Например, вы можете использовать `__new__` для реализации паттернов проектирования, таких как singleton или factory.

```
1 class Singleton:
2     _instance = None # атрибут класса для хранения экземпляра
3
4     def __new__(cls, *args, **kwargs):
5         if cls._instance is None:
6             cls._instance = super().__new__(cls)
7         return cls._instance
8
9 s1 = Singleton()
10 s2 = Singleton()
11 print(s1 is s2) # True
```

В этом примере `__new__` проверяет, существует ли уже экземпляр Singleton, и если нет, вызывает `super().__new__(cls)` для создания нового. Если экземпляр уже существует, `__new__` просто возвращает его. Это гарантирует, что может существовать только один экземпляр Singleton.

Аргумент `cls`

Аргумент `cls` — это ссылка на класс, который вызывается. Он аналогичен аргументу `self`, который используется в методах экземпляра и указывает на конкретный объект.

Использование `cls` позволяет работать с классом в контексте методов класса, таких как `__new__` и другие классовые методы.

Применение `cls`:

- Позволяет создавать новые экземпляры класса.
- Используется для доступа к атрибутам и методам самого класса.

Метод `super`

Метод `super` предоставляет доступ к методам родительского класса. Он используется для вызова методов, определенных в родительских классах, что особенно полезно в многократном наследовании. `super` помогает избежать явного указания имени родительского класса и обеспечивает более гибкое и безопасное взаимодействие между классами.

Основные моменты о `super`:

- **Вызывается для родительского класса:** Позволяет вызывать методы родительского класса без указания его имени.
- **Удобство в многократном наследовании:** Обеспечивает правильный порядок разрешения методов (Method Resolution Order, MRO).
- **Применение в `__init__`:** Часто используется для инициализации родительского класса в конструкторе.

Практическая часть

Создаем класс Singleton, в классе создаем атрибут класс `__instance`, который будет хранить объект класса.

```
1 class Singleton:
2     __instance = None
3
4
5     def __new__(cls, *args, **kwargs):
6         if cls.__instance is None:
7             cls.__instance = super().__new__(cls)
8         return cls.__instance
```

Переопределяем магический метод `__new__`. В методе `__new__` есть обязательный параметр `cls`, который ссылается на сам класс, а не на объект класса (`self` ссылается на объект класса). Также в параметрах, есть `*args` и `**kwargs`, они хранят все значение атрибута объекта в `set` и `dict`. Теперь в методе, пишем условие, которое будет проверят атрибут класса `__instance`, если в атрибуте объекта класса. При отсутствии в атрибуте класса, объекта класса будет создавать объект класса с помощью класса `super()` (класс `super()` - позволяет обратиться к родительскому классу и вызывать его методы, при создание класса они наследуются от объекта `object`, в котором находятся все магические методы). Вызываем метод `__new__`, чтобы создать объект класса.

Теперь создаем два объекта класса `Singleton`. Проверяем их идентичность это можно сделать с помощью оператора `is` или вывести объекты в терминал.

```
print(s1 is s2)
print(s2)
print(s1)
```



```
True
<__main__.Singleton object at 0x000001A690CC2A90>
<__main__.Singleton object at 0x000001A690CC2A90>
```

Задание

1. Создай класс, который мог создавать только пять объектов, если количество классов закончилось, то создавалась бы последнее созданный объект класса.
2. Создай класс Book:

Атрибуты объекта:

title

author

year

Который нельзя будет создать, одинаковыми атрибутами объектов title и author.

Контрольные вопросы

1. Чем отличаются метод `__new__` от `__init__`?
2. Для чего нужен класс `super()`?