IT CAREER HUB

Урок 35.
Атрибуты объекта и класса

Поля класса 2
Доступ к полям 3
Поля объекта по умолчанию 7
Задания для закрепления 8
Классовые методы 9
Статические методы 12
Магический метод __str__ 15
Магический метод __repr__ 17
Задания для закрепления 18
Ответы на задания 19
Практическая работа 20

IT CAREER HUB

Поля класса

До этого мы работали с **полями объектов** — данными, которые принадлежат каждому конкретному экземпляру. Но в Python можно также создавать **поля класса**.

Поле класса (атрибут класса) — это переменная, которая принадлежит самому классу, а не отдельному объекту. Все экземпляры класса могут получить к ней доступ, и при этом значение хранится в одном месте — внутри класса.

Синтаксис:

```python
class ClassName:
    class_attribute = value  # поле класса
```

- `class_attribute` — поле класса, общее для всех объектов
- `value` — значение этого поля

Пример

```python
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title    # поле объекта
        self.author = author  # поле объекта
```

Когда использовать:

- Для хранения **настроек, категорий, счётчиков, общих значений**
- Когда значение **одинаково для всех объектов**

IT CAREER HUB

Доступ к полям

Поля класса можно читать и изменять как **внутри класса**, так и **снаружи — через класс или объект**. Но при этом поведение может отличаться — особенно при изменении значений.

1. Чтение поля класса

Поле класса можно прочитать:

- напрямую через класс: `ClassName.attribute`
- через объект: `object_name.attribute`
- внутри методов класса — через `self.attribute` или `ClassName.attribute`

Пример

```python
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title
        self.author = author

    def show_library(self):
        print(self.library_name)  # доступ через self
        print(Book.library_name)  # доступ через имя класса

print(Book.library_name)  # доступ через класс
book = Book("1984", "George Orwell")
print(book.library_name)  # доступ через объект
book.show_library()       # доступ изнутри
```

2. Изменение значения через класс

Если нужно **изменить значение для всех объектов**, это делается через класс.

Пример

```python
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title
        self.author = author

book = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")
Book.library_name = "City Library"
print(Book.library_name)
print(book.library_name)      # у объекта тоже изменилось
print(book2.library_name)     # у объекта тоже изменилось
```

3. Создание поля через объект

Если попытаться изменить значение поля через объект, на самом деле будет создано **новое поле в конкретном объекте**, не затронув поле класса.

Пример

```python
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title
        self.author = author

book = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")

book.library_name = "Private Shelf"  # создаёт новое поле в book
print(book.library_name)  # Private Shelf — новое значение в объекте
```

```
print(Book.library_name)  # поле класса не изменилось
print(book2.library_name) # также прочитает поле класса
```

4. Механизм поиска атрибутов

При обращении к полю через объект (`object.attribute`), Python сначала **проверяет, есть ли такое поле у самого объекта**. Если не находит — продолжает поиск **в классе**, из которого объект был создан.

5. Магическое поле __dict__

`__dict__` — это **специальный атрибут** (магическое поле) объекта или класса, в котором хранятся **все явно заданные атрибуты** в виде словаря.

- `object.__dict__` показывает **все поля**, которые были присвоены **этому конкретному объекту**
- `ClassName.__dict__` содержит **все атрибуты класса**, включая методы и поля класса, но не отображает поля, присвоенные конкретным объектам

Пример

Python

```
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title
        self.author = author

book = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")

book.library_name = "Private Shelf"  # создаёт новое поле в book

print(book.__dict__)  # у book1 появилось собственное поле
print(book2.__dict__) # у book2 такого поля нет
```

```
print(Book.__dict__)  # поля и методы класса
```

6. Добавление новых полей извне класса

Python позволяет **в любой момент добавить новое поле** как к классу, так и к объекту. Но **делать это не рекомендуется** — такие поля **неочевидны**, их сложно отследить, и они **могут привести к ошибкам**, особенно в больших проектах или при командной разработке.

Пример

Python

```
class Book:
    library_name = "Central Library"  # поле класса

    def __init__(self, title, author):
        self.title = title
        self.author = author

book = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")

book.book_color = "black"            # создание поля объекта вне класса
Book.default_language = "eng"        # создание поля класса вне класса
print(book.book_color)
# print(book2.book_color)            # ошибка, такого поля нет
print(book.default_language)
```

IT CAREER HUB

Поля объекта по умолчанию

Иногда нужно, чтобы у всех объектов было **базовое значение** для некоторого поля. Например, формат книги, статус пользователя, язык интерфейса и т.д. При этом важно сохранить возможность **присвоить другое значение** при создании объекта, если нужно.

В Python это реализуется через **значения по умолчанию** в методе `__init__` — ведь он работает как обычная функция.

Пример:

```python
class Book:
    def __init__(self, title, author, language="English"):
        self.title = title
        self.author = author
        self.language = language

book1 = Book("1984", "George Orwell")                                    # язык по умолчанию
book2 = Book("Cien años de soledad", "Gabriel García Márquez", "Spanish") # другой язык

print(book1.language)
print(book2.language)
```

IT CAREER HUB

Задания для закрепления

1. **Укажи верные утверждения о полях класса:**

a. Поле класса общее для всех объектов
b. Поле класса создаётся внутри метода `__init__()`
c. Изменение поля через объект всегда меняет поле класса
d. Поле класса можно прочитать через объект

Посмотреть ответ

IT CAREER HUB

Классовые методы

Классовый метод — это метод, который работает не с конкретным объектом, а с самим классом. Он получает доступ не к `self`, а к `cls` — ссылке на класс, из которого был вызван.

Чтобы объявить такой метод, используется **декоратор** `@classmethod`.

Зачем нужны классовые методы?

- Чтобы **работать с полями класса** (например, счётчиками или общими настройками)
- Чтобы **создавать объекты особым образом** — через альтернативные конструкторы
- Чтобы **добавлять поведение, относящееся к самому классу**, а не к отдельному экземпляру

Синтаксис:

```python
class ClassName:
    @classmethod
    def method_name(cls, ...):
        ...
```

- `@classmethod` — декоратор, помечающий метод как классовый
- `cls` — ссылка на сам класс (аналог `self`, но для класса)

Пример: счётчик созданных объектов

```python
class Book:
    total_books = 0  # поле класса
```

```
    def __init__(self, title, author):
        self.title = title
        self.author = author
        Book.total_books += 1

    @classmethod
    def show_total(cls):
        print(f"Total books: {cls.total_books}")
        # print(cls.title)  # Нельзя обратиться к полю объекта

Book.show_total()     # Вызов через класс
book1 = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")

Book.show_total()     # Вызов через класс
book1.show_total()    # Вызов через объект
```

Пример: альтернативный конструктор (создание объекта по шаблону)

Иногда удобно создавать объекты **не через обычный** `__init__`, а другим способом — например, **из строки** или **словаря**. Для этого используется **классовый метод как альтернативный конструктор**.

Python

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @classmethod
    def from_string(cls, data):
        title, author = data.split(" - ")
        # cls — это Book, поэтому cls(title, author) эквивалентно Book(title,
author)
        return cls(title, author)  # создаёт и возвращает новый объект

book = Book.from_string("1984 - George Orwell")
```

```
print(book.title)
print(book.author)
```

Пример: поведение, связанное с классом

Классовые методы также полезны, когда нужно **выполнить действие, относящееся ко всему классу**,например, вывести настройки по умолчанию или сбросить глобальные параметры.

Python

```
class Book:
    default_format = "PDF"
    default_language = "en"

    @classmethod
    def show_defaults(cls):
        print(f"Format: {cls.default_format}, Language:
{cls.default_language}")

    @classmethod
    def reset_defaults(cls):
        cls.default_format = "PDF"
        cls.default_language = "en"

Book.default_format = "EPUB"  # Изменение формата
Book.show_defaults()          # Просмотр текущих настроек
Book.reset_defaults()         # Сброс до базовых настроек
Book.show_defaults()          # Просмотр текущих настроек
```

IT CAREER HUB

Статические методы

Статический метод — это метод, который не зависит ни от объекта, ни от самого класса. Он работает как обычная функция, но помещён внутрь класса для логической связанности с ним.

Чтобы объявить такой метод, используется **декоратор** `@staticmethod`.

Когда использовать статические методы?

- Когда функция **логически относится к классу**, но **не использует ни** `self`**, ни** `cls`
- Чтобы **сгруппировать связанную логику внутри класса**, а не выносить её в отдельные функции вне класса

Синтаксис:

```python
class ClassName:
    @staticmethod
    def method_name(...):
        ...
```

- `@staticmethod` — декоратор, превращающий метод в статический
- Такой метод **не получает ни** `self`**, ни** `cls`

Пример: вспомогательная функция внутри класса

```python
class Book:
    def __init__(self, title):
        self.title = title

    @staticmethod
    def is_title_valid(title):
```

```
        return isinstance(title, str) and len(title) > 0

print(Book.is_title_valid("1984"))       # True
print(Book.is_title_valid(""))           # False
print(Book.is_title_valid(123))          # False
```

- Такой метод можно вызывать и через класс (`Book.is_title_valid()`), и через объект — результат будет одинаков.

Пример: логически связанные, но независимые операции

```
Python
class MathHelper:
    @staticmethod
    def square(x):
        return x * x

    @staticmethod
    def cube(x):
        return x * x * x

print(MathHelper.square(5))
print(MathHelper.cube(3))
```

Сравнение типов методов

Тип метода	Обычный метод	Классовый метод	Статический метод
Декоратор	–	`@classmethod`	`@staticmethod`
Первый аргумент	`self`	`cls`	–
Доступ	К полям и методам **объекта**	К полям и методам **класса**	-

Когда использовать	Когда метод работает с конкретным экземпляром	Когда метод работает с классом или создаёт объекты по шаблону	Когда функция логически относится к классу, но ничего не использует

IT CAREER HUB

Магический метод __str__

Когда объект выводится через `print()`, Python вызывает магический метод `__str__()`, чтобы получить **строковое представление объекта**, которое будет показано пользователю.

По умолчанию, если метод `__str__()` не определён, Python выводит **техническую информацию** — тип и адрес объекта в памяти, что не информативно.

Определение метода `__str__()` позволяет **вернуть читаемую строку** о содержимом объекта.

Используйте метод `__str__()`, чтобы показать **основную информацию** об объекте.

Пример: техническая информация

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book = Book("1984", "George Orwell")
print(book)
```

Пример: содержимое объекта

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"
```

```
book = Book("1984", "George Orwell")
print(book)       # вызывает __str__
print(str(book))  # вызывает __str__
```

Особенности:

- Не должен ничего выводить сам — **только возвращать строку**
- Вызывается автоматически при `print(obj)`или `str(obj)`

IT CAREER HUB

Магический метод __repr__

Магический метод `__repr__()` возвращает **техническое представление объекта**, удобное для **разработчика**. Оно используется при выводе в коллекциях, отладке и в интерактивной оболочке.

Пример:

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __repr__(self):
        return f"Book(title={self.title!r}, author={self.author!r})"

book1 = Book("1984", "George Orwell")
book2 = Book("Brave New World", "Aldous Huxley")
print(repr(book1))
books = [book1, book2]
print(books)
```

Особенности:

- Если `__str__()` отсутствует, Python использует `__repr__()` как запасной вариант при `print()`.
- Может использоваться **в логах**, **интерактивных сессиях** и отладчиках.
- В идеале, `__repr__()` должен возвращать строку, из которой **можно воссоздать объект**.

IT CAREER HUB

Задания для закрепления

1. **Укажи верные утверждения о @classmethod:**
   a. Метод `@classmethod` получает первым аргументом `self`
   b. Метод `@classmethod` может создавать объект класса
   c. Метод `@classmethod` имеет доступ к атрибутам конкретного объекта
   d. Метод `@classmethod` вызывается только через класс, а не через объект
   e. Метод `@classmethod` имеет доступ к классу

Посмотреть ответ

2. Найдите ошибку в коде:

```python
class Person:
    @staticmethod
    def greet(self):
        print("Hello!")
```

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Утверждения о полях класса	Ответ: a, d
Задания на закрепление 2	Вернуться к заданиям
1. Утверждения о `@classmethod`	Ответ: b, e
2. Ошибка в коде	Ответ: у статического метода не должно быть параметра `self`

IT CAREER HUB

Практическая работа

1. Расстояние между городами

Создайте класс `City`, представляющий город с координатами.

- У каждого города есть поля `name`, `latitude`, `longitude`.
- Добавьте строковое представление объекта.
- Добавьте метод `distance(city1, city2)`, который возвращает кортеж `(latitude, longitude)` между двумя городами.
- Проверьте расстояние между двумя городами.

Пример вывода:

```python
City: Berlin (52.52, 13.4)
City: Paris (48.85, 2.35)
Distance: 14.72
```

Решение:

```python
class City:
    def __init__(self, name, latitude, longitude):
        self.name = name
        self.latitude = latitude
        self.longitude = longitude

    def __str__(self):
        return f"City: {self.name} ({self.latitude}, {self.longitude})"

    @staticmethod
    def distance(city1, city2):
        lat_diff = round(city1.latitude - city2.latitude, 2)
        lon_diff = round(city1.longitude - city2.longitude, 2)
        return lat_diff, lon_diff

berlin = City("Berlin", 52.52, 13.40)
paris = City("Paris", 48.85, 2.35)

print(berlin)
print(paris)
print("Distance:", City.distance(berlin, paris))
```

IT CAREER HUB

2. Создание объекта из строки

Доработайте класс `City`.

- Добавьте метод `from_string(data)`, который создаёт объект из строки вида `"Rome:41.89,12.51"`.
- Проверьте создание нового объекта через этот метод и выведите его.

Пример вывода:

```python
City: Rome (41.89, 12.51)
```

Решение:

```python
class City:
    def __init__(self, name, latitude, longitude):
        self.name = name
        self.latitude = latitude
        self.longitude = longitude

    def __str__(self):
        return f"City: {self.name} ({self.latitude}, {self.longitude})"

    @classmethod
    def from_string(cls, data):
        name_part, coords_part = data.split(":")
        lat, lon = map(float, coords_part.split(","))
        return cls(name_part, lat, lon)

    @staticmethod
    def distance(city1, city2):
        lat_diff = round(city1.latitude - city2.latitude, 2)
        lon_diff = round(city1.longitude - city2.longitude, 2)
        return lat_diff, lon_diff

rome = City.from_string("Rome:41.89,12.51")
print(rome)
```