IT CAREER HUB

Урок 37.
Множественное наследование

IT CAREER HUB

Множественное наследование

Множественное наследование — это механизм, при котором класс может наследовать сразу от нескольких родительских классов.

Это позволяет **объединять функциональность из разных классов**, создавая более универсальные и гибкие структуры.

Синтаксис:

```python
class Child(Parent1, Parent2):
    pass
```

- `Child` — дочерний класс
- `Parent1, Parent2` — родительские классы, перечисленные через запятую

Пример

```python
class Printable:
    def print_info(self):
        print("Печать информации...")

class Savable:
    def save(self):
        print("Сохраняем в файл...")

class Report(Printable, Savable):
    pass

r = Report()
r.print_info()  # унаследовано от Printable
r.save()        # унаследовано от Savable
```

Особенности:

- Все методы и атрибуты родительских классов доступны в дочернем
- Порядок перечисления родителей имеет значение — используется **слева направо**
- Если имена методов совпадают, будет вызван метод **первого подходящего родителя**

IT CAREER HUB

Функция hasattr

 **`hasattr()` — это встроенная функция Python, которая позволяет проверить наличие атрибута у объекта. Возвращает `True`, если атрибут существует у объекта, `False` - если отсутствует.**

Она особенно полезна при работе с объектами, структура которых может быть неизвестна заранее, или при использовании **наследования и миксинов**, где поведение может зависеть от наличия тех или иных атрибутов.

Синтаксис:

```python
hasattr(obj, name)
```

- `obj` – объект, у которого нужно проверить наличие атрибута
- `name` — строка с именем атрибута

 **Пример**

```python
class User:
    def __init__(self, username):
        self.username = username

user = User("Alice")

print(hasattr(user, "username"))  # Атрибут существует
print(hasattr(user, "email"))     # Атрибута нет
```

Когда использовать:

- При **динамическом добавлении полей**
- В **миксинах и декораторах**, чтобы понять, поддерживает ли объект нужное поведение
- В системах, где объекты приходят из разных источников и не имеют стабильной структуры

IT CAREER HUB

Миксины

Миксин — это вспомогательный класс, который добавляет дополнительное поведение другим классам через множественное наследование.

Миксины **не предназначены для самостоятельного использования**, они **не создают объекты**, а лишь «примешиваются» к другим классам, чтобы **расширить их возможности**.

Признаки миксинов

- **Не имеют собственного состояния** (`__init__()` обычно не определяют)
- Содержат **одну или несколько полезных функций**
- Используются **только вместе с другими классами**
- **Имя обычно заканчивается на** `Mixin` (например, `LoggableMixin`, `SavableMixin`)

Зачем нужны миксины

- Позволяют **повторно использовать поведение** без дублирования кода
- Упрощают структуру классов, разделяя обязанности
- Делают код **гибким и расширяемым** при помощи множественного наследования

Пример: класс объединяющий аутентификацию и отправку уведомлений

```python
class AuthMixin:
    def login(self):
        if not hasattr(self, "username"):
            raise AttributeError("Не задан username")
        print(f"{self.username} вошёл в систему.")

    def logout(self):
        print("Пользователь вышел из системы.")

class NotificationMixin:
    def send_email(self, message):
```

```
        if not hasattr(self, "email"):
            raise AttributeError("Не задан email")
        print(f"Отправка письма на {self.email}: {message}")

class UserProfile(AuthMixin, NotificationMixin):
    def __init__(self, username, email):
        self.username = username
        self.email = email

user = UserProfile("alice", "alice@example.com")
user.login()                              # alice вошёл в систему.
user.send_email("Добро пожаловать!")      # Отправка письма на
alice@example.com: Добро пожаловать!
user.logout()                             # Пользователь вышел из системы.
```

- `AuthMixin` добавляет поведение, связанное с входом в систему
- `NotificationMixin` — добавляет поведение, связанное с отправкой уведомлений
- `UserProfile` объединяет всё это и определяет нужные данные
- Классы родители определяют поведение, которое может быть полезно разным наследникам

IT CAREER HUB

Порядок поиска методов при наследовании

Когда у объекта вызывается метод, Python ищет его:

- В самом классе
- В **первом родителе** (слева направо)
- Во всей его иерархии **вглубь, от наследника к предкам**
- Далее во **втором родителе** и его иерархии
- И так далее, **слева направо**, по всем родителям
- Последним проверяется базовый класс `object`

Как только метод найден — поиск прекращается.

Пример

```python
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    pass

class C:
    def greet(self):
        print("Hello from C")

class D(B, C):
    pass

d = D()  # D и B не имеют метода greet
d.greet()
```

Как ищется метод greet():

1. D — нет
2. B — нет
3. A — **нашёл! вызывается**

4. До C Python уже не доходит

Diamond problem, или проблема ромба — это ситуация в множественном наследовании, когда класс наследует от нескольких классов, которые в свою очередь происходят от одного общего предка.

В результате образуется **ромбовидная структура наследования**, и возникает вопрос: **какую версию метода или поля из общего предка использовать?**

Схема наследования:

Класс `D` наследует и от `B`, и от `C`, а они — от `A`.

```
   A
  / \
 B   C
  \ /
   D
```

Как Python решает это?

Python использует **обход в глубину слева направо**, и только после проверяет общего родителя **единожды**.

IT CAREER HUB

Задание для закрепления 1

Какой будет порядок поиска метода?

```python
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    pass

class C(A):
    def greet(self):
        print("Hello from C")

class D(B, C):
    pass

d = D()
d.greet()
```

Посмотреть ответ

IT CAREER HUB

Порядок разрешения методов (MRO)

Когда у объекта вызывается метод, Python должен определить, в каком порядке искать его в классах. При простом наследовании это просто, но при множественном наследовании становится сложнее.

Чтобы определить однозначный порядок поиска, Python использует механизм **MRO (Method Resolution Order) — порядок разрешения методов**.

Что делает MRO?

- Определяет **в каком порядке искать методы и поля**
- Работает **автоматически** при вызове любого метода
- Учитывает **весь путь наследования**, **включая множественное**

Как посмотреть MRO?

Python предоставляет возможность узнать MRO любого класса.

Синтаксис:

```python
print(ClassName.__mro__)  # Возвращает кортеж классов
```

Пример

```python
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):
        print("Hello from B")
```

```python
class C(A):
    def greet(self):
        print("Hello from C")

class D(B, C):
    def greet(self):
        print("Hello from D")

print(D.__mro__)
```

IT CAREER HUB

Функция super в множественном наследовании

В множественном наследовании функция `super()` играет **особенно важную роль**: она позволяет **гарантированно вызывать следующий метод в цепочке MRO**, а не просто обращаться к конкретному родителю.

```python
ParentClass.method(self)  # Плохо – жёсткая привязка, не учитывает MRO
super().method()          # Хорошо – учитывает порядок разрешения методов
```

- `super()` не означает "ближайший родитель" — это **следующий класс в цепочке MRO**, у которого есть метод
- Такой подход делает код **гибким, надёжным и расширяемым**, особенно в системах с несколькими родителями

Пример

```python
class A:
    def action(self):
        print("A")

class B(A):
    def action(self):
        print("B")
        super().action()

class C(A):
    def action(self):
        print("C")
        super().action()

class D(B, C):
    def action(self):
        print("D")
```

```
        super().action()

d = D()
d.action()
```

- Метод `action()` будет вызываться **в порядке MRO**, переходя через `super()` — таким образом **выполняются все реализации метода**, а не только одна.

IT CAREER HUB

Задания для закрепления 2

1. **Для чего используется super() в наследовании?**

a. Чтобы вызвать метод только родительского класса напрямую
b. Чтобы гибко переходить к следующему методу в MRO
c. Чтобы заменить метод родителя
d. Чтобы обратиться к базовому типу object

Посмотреть ответ

2. Что напечатает код?

```python
class SaveMixin:
    def process(self):
        print("Saving...")

class PrintMixin:
    def process(self):
        print("Printing...")

class Document(SaveMixin, PrintMixin):
    pass

doc = Document()
doc.process()
```

a. `Saving…`
b. `Printing…`
c. `Ошибка`

Посмотреть ответ

Композиция и агрегация

Композиция и **агрегация** — это два способа построения отношений между объектами, при которых один объект **включает в себя** другой как часть своей структуры.

Они оба позволяют **собирать сложные объекты из более простых**, но различаются по уровню зависимости.

Композиция

Композиция означает, что один объект полностью принадлежит другому, то есть является частью его композиции.

- Один объект **владеет** другим и **отвечает за его создание и удаление**
- Связь очень тесная: **если уничтожить внешний объект — внутренний тоже исчезнет**
- Вложенный объект **не существует отдельно**

Пример: Menu создаёт и использует ExitButton

У нас есть класс `Menu`, и он **всегда содержит кнопку выхода**, которую создаёт сам. Кнопка не существует отдельно — это часть меню.

```python
class ExitButton:
    def click(self):
        print("Выход из программы")

class Menu:
    def __init__(self):
        self.exit_button = ExitButton()  # создаётся внутри

    def show(self):
        print("Меню открыто")
        self.exit_button.click()

menu = Menu()
```

```
menu.show()
```

Почему это композиция?

- `ExitButton` создаётся внутри `Menu`
- Кнопка **не существует отдельно** — она часть меню
- `Menu` **полностью управляет** жизненным циклом кнопки (создаёт и использует её)

Агрегация

Агрегация означает, что один объект использует другой, но не управляет его созданием или удалением. Связанный объект создаётся вне основного и может использоваться в других местах.

- Объект **использует** другой, но **не владеет** им напрямую
- Вложенный объект **создаётся снаружи** и передаётся при инициализации
- Если уничтожить внешний объект — вложенный может продолжать жить

Пример: Teacher использует University

Python

```python
class University:
    def __init__(self, name):
        self.name = name

    def get_info(self):
        print(f"Обучение проходит в университете: {self.name}")

class Teacher:
    def __init__(self, name, university):
        self.name = name
        self.university = university  # передаётся извне

    def introduce(self):
        print(f"Преподаватель: {self.name}")
        self.university.get_info()
```

```
# Университет создаётся один раз
uni = University("Tech University")

# Передаётся в нескольких преподавателей
t1 = Teacher("Anna", uni)
t2 = Teacher("Dmitry", uni)

t1.introduce()
t2.introduce()
```

Почему это агрегация?

- `University` создаётся отдельно
- Один объект `University` может использоваться несколькими объектами `Teacher`
- `Teacher` **не управляет** жизненным циклом университета

Вложенные классы как композиция

Иногда один класс логически **является частью** другого и **не имеет смысла вне его контекста**.

В таких случаях удобно определять вспомогательный класс **внутри** внешнего — это подчёркивает, что он используется **только как часть реализации** и не предназначен для отдельного применения.

Это тоже считается **композицией**, потому что внешний объект **создаёт и управляет** вложенным элементом.

Пример: Вложенный класс Battery внутри Smartphone

```python
class Smartphone:
    class Battery:
```

```
        def __init__(self, capacity):
            self.capacity = capacity
            self.charge = capacity

        def use(self, amount):
            self.charge = max(self.charge - amount, 0)
            print(f"Батарея: {self.charge}/{self.capacity} мАч")

    def __init__(self, model, battery_capacity):
        self.model = model
        self.battery = self.Battery(battery_capacity)

    def play_video(self):
        print(f"{self.model} воспроизводит видео...")
        self.battery.use(300)

phone = Smartphone("Pixel 9", 4000)
phone.play_video()
```

Почему это композиция?

- Класс `Battery` определён **внутри** `Smartphone`, потому что используется **только им**
- Он создаётся и контролируется внешним классом
- Нельзя создать `Battery` сам по себе и использовать вне смартфона

Главные различия композиции и агрегации

	Композиция	**Агрегация**
Создание	Объект создаёт внутренний сам	Объект получает готовый элемент извне
Связь	Жёсткая, вложенный неотделим	Слабая, вложенный может существовать отдельно
Жизненный цикл	Управляется внешним объектом	Независимый

Пример	`Car` создаёт `Engine` внутри себя	`Car` получает `Engine` как аргумент

IT CAREER HUB

Задания для закрепления 3

1. Сопоставьте понятие с описанием:

1. Композиция
2. Агрегация
3. Вложенный класс

a. Класс определён внутри другого класса
b. Объект создаётся и управляется внешним объектом
c. Объект передаётся извне и может существовать отдельно

Посмотреть ответ

2. Какая связь реализована в коде?

```python
class Course:
    def __init__(self, teacher):
        self.teacher = teacher
```

a. Композиция
b. Агрегация

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
Порядок поиска метода	Ответ: D → B → C → A → object
Задания на закрепление 2	Вернуться к заданиям
1. super() в наследовании	Ответ: b
2. Результат выполнения кода	Ответ: a
Задания на закрепление 3	Вернуться к заданиям
1. Сопоставить понятие с описанием	Ответ: 1-b, 2-c, 3-a
2. Реализация связи в коде	Ответ: b

IT CAREER HUB

Практическая работа

1. Класс Student

Создайте класс `Student`, представляющий ученика.

- При создании указываются имя и email.
- Добавьте строковое представление студента (например, только имя).
- Добавьте метод `notify(message)`, который выводит сообщение: `Email to <email>: <message>`

Проверьте создание объекта и вызов метода.

Решение:

```python
class Student:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def __str__(self):
        return self.name

    def notify(self, message):
        print(f"Email to {self.email}: {message}")

s = Student("Alice", "alice@example.com")
print(s)
s.notify("You have been enrolled.")
```

IT CAREER HUB

2. Класс Course

Создайте класс `Course`, представляющий учебный курс.

- При создании указывается название курса.
- У каждого объекта `Course` должно быть поле `students`, в котором хранится список зарегистрированных студентов.
- Добавьте метод `add_student(student)`, который принимает объект `Student` и добавляет его в курс.
- Добавьте метод `show_students()`, который выводит список имён студентов.
- Добавьте метод `notify_all(message)`, который уведомляет всех студентов курса.

Проверьте работу методов, создав курс, добавив студентов и отправив уведомление.

Пример вывода:

```python
Students enrolled in Python OOP:
- Alice
- Bob

Email to alice@example.com: Welcome to the course!
Email to bob@example.com: Welcome to the course!
```

Решение:

```python
class Course:
    def __init__(self, title):
        self.title = title
        self.students = []

    def add_student(self, student):
        self.students.append(student)

    def show_students(self):
        print(f"Students enrolled in {self.title}:")
        for student in self.students:
            print("-", student)

    def notify_all(self, message):
        for student in self.students:
            student.notify(message)

s1 = Student("Alice", "alice@example.com")
s2 = Student("Bob", "bob@example.com")

course = Course("Python OOP")
course.add_student(s1)
course.add_student(s2)

course.show_students()
course.notify_all("Welcome to the course!")
```