IT CAREER HUB

Python

Множественное наследование

Преподаватель

Портрет

Имя Фамилия

Текущая должность

Количество лет опыта

Какой у Вас опыт - ключевые кейсы

Самые яркие проекты

Дополнительная информация по вашему усмотрению

Корпоративный e-mail

Социальные сети (по желанию)

Важно

- Камера должна быть включена на протяжении всего занятия
- В течение занятия вопросы задавать в чате или когда преподаватель спрашивает, есть ли у Вас вопросы
- Вести себя уважительно и этично по отношению к остальным участникам занятия
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя

Повторение

- [x] Принципы ООП
- [x] Наследование
- [x] Полиморфизм
- [x] Функция super
- [x] Наследование от object
- [x] Функции isinstance и issubclass

План занятия

- Множественное наследование
- Функция hasattr
- Миксины
- Порядок поиска методов при наследовании
- Порядок разрешения методов (MRO)
- Функция super в множественном наследовании
- Композиция и агрегация

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

ICH IT CAREER HUB

Множественное наследование

Полезно знать

ICH
IT CAREER HUB

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.

Множественное наследование

Синтаксис

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

Пояснение

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

Пример

```
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 множественного наследования

- 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

ВОПРОСЫ

IT CAREER HUB

Функция hasattr

Полезно знать

hasattr()

Это встроенная функция Python, которая позволяет проверить наличие атрибута у объекта. Возвращает True, если атрибут существует у объекта, False - если отсутствует.

Функция hasattr

Синтаксис

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

Пояснение

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

Пример

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

user = User("Alice")

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

Использование функции hasattr

- 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

ВОПРОСЫ

IT CAREER HUB

Миксины

Полезно знать

ICH
IT CAREER HUB

Миксин

Это вспомогательный класс, который добавляет дополнительное поведение другим классам через множественное наследование

Важно

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`)

Польза миксинов

- 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}")
```

1

```python
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.
```

2

- `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

ВОПРОСЫ

IT CAREER HUB

Порядок поиска методов при наследовании

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

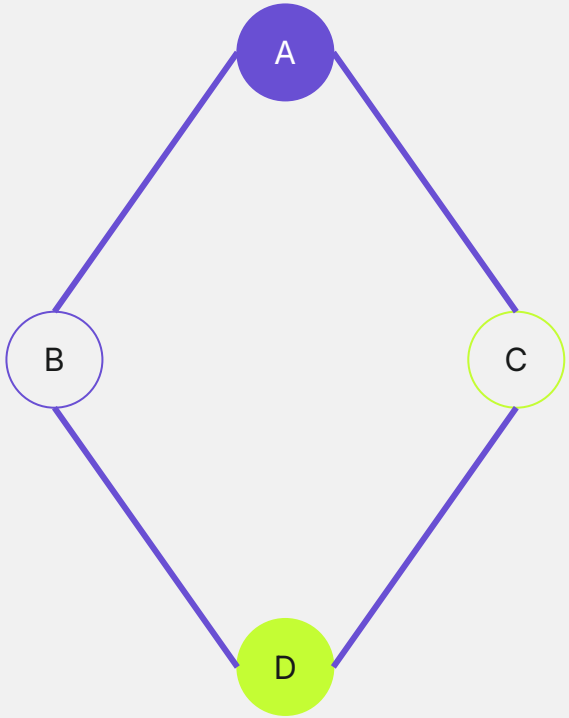
Полезно знать

ICH
IT CAREER HUB

Diamond problem
Проблема ромба

Это ситуация в множественном наследовании, когда класс наследует от нескольких классов, которые в свою очередь происходят от одного общего предка

Схема наследования



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

Важно

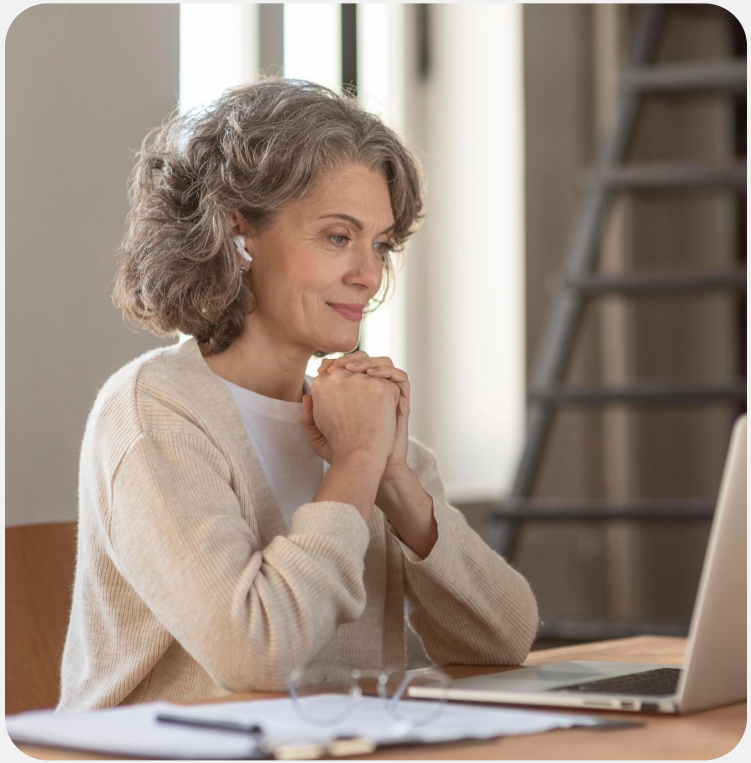
Python использует **обход в глубину слева направо**, и только после проверяет общего родителя **единожды**.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ



Ответьте на вопрос

Какой будет порядок поиска метода?

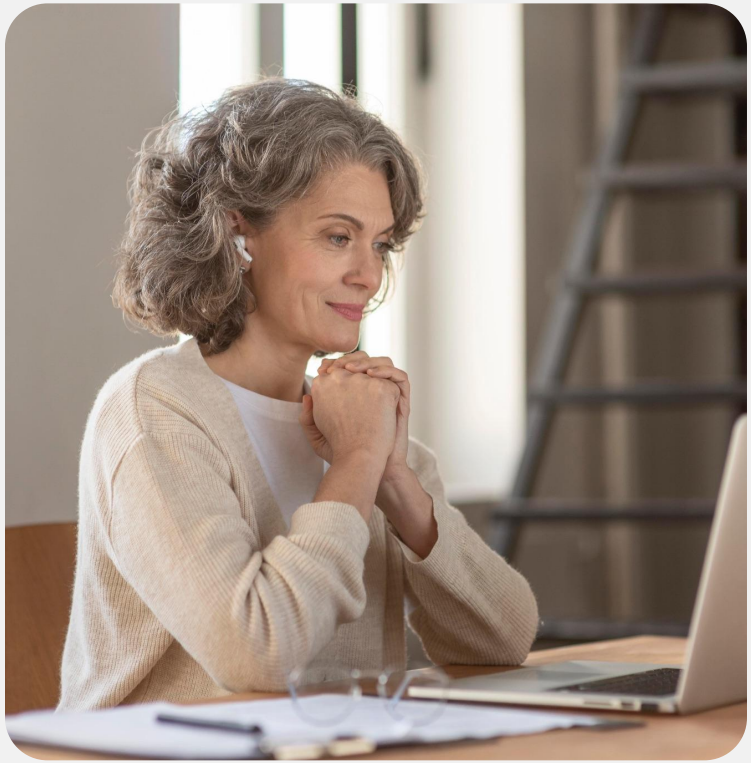
```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    pass

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

class D(B, C):
    pass

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



Ответьте на вопрос

Какой будет порядок поиска метода?

```
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()
```

Ответ: D → B → C → A → object

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Порядок разрешения методов (MRO)

Важно

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

Работа MRO

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

Как посмотреть MRO

Синтаксис

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

Пояснение

Python предоставляет возможность узнать MRO любого класса

Пример

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

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

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

ВОПРОСЫ

ICH IT CAREER HUB

Функция super в множественном наследовании

Важно

В множественном наследовании функция `super()` играет **особенно важную роль**: она позволяет **гарантированно вызывать следующий метод в цепочке MRO**

Функция super

Пример

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

Пояснение

- `super()` — 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

```
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()
```

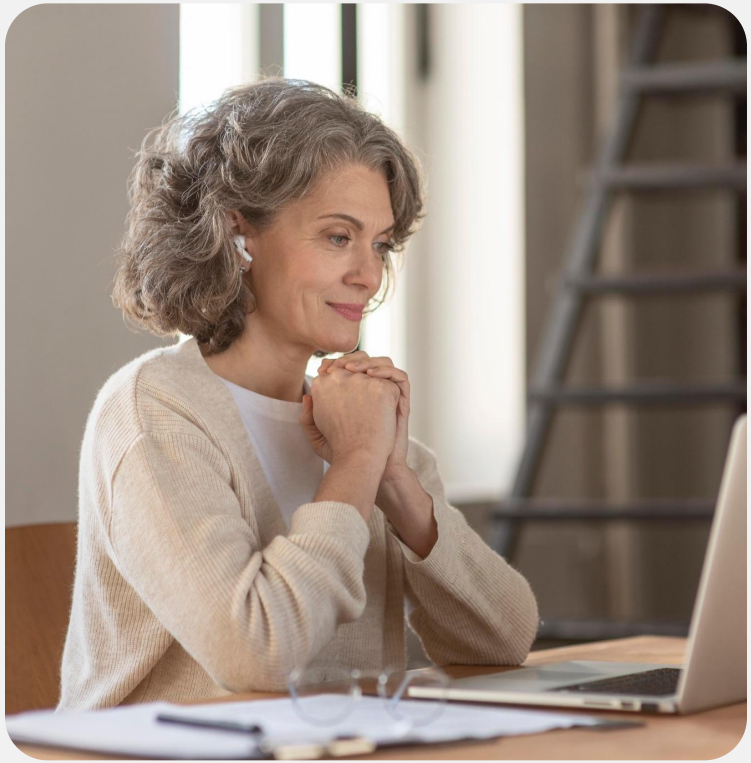
Метод `action()` будет вызываться **в порядке MRO**, переходя через `super()` — таким образом **выполняются все реализации метода**, а не только одна.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

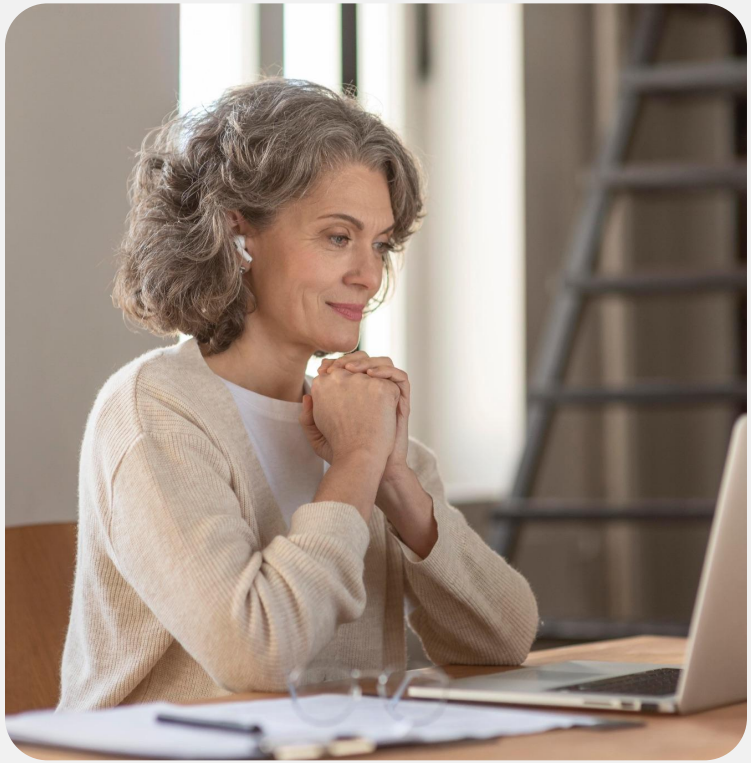
ЗАДАНИЯ



Выберите верный вариант ответа

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



Выберите верный вариант ответа

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

Выберите верный вариант ответа

2. Что напечатает код?

```
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

Выберите верный вариант ответа

2. Что напечатает код?

```
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

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Композиция и агрегация

Важно

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

Полезно знать

ICH
IT CAREER HUB

Композиция

Отношения между объектами, когда один объект полностью принадлежит другому, то есть является частью его композиции

Особенности 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

```
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()
```

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

Полезно знать

ICH
IT CAREER HUB

Агрегация

Это отношения между объектами, когда один объект использует другой, но не управляет его созданием или удалением. Связанный объект создаётся вне основного и может использоваться в других местах.

Особенности агрегации

- 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
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()
```

1

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

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

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

2

- `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 тоже считается **композицией**

Пример

Вложенный класс Battery внутри Smartphone

```
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")
```

1

```
#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()
```

2

- 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

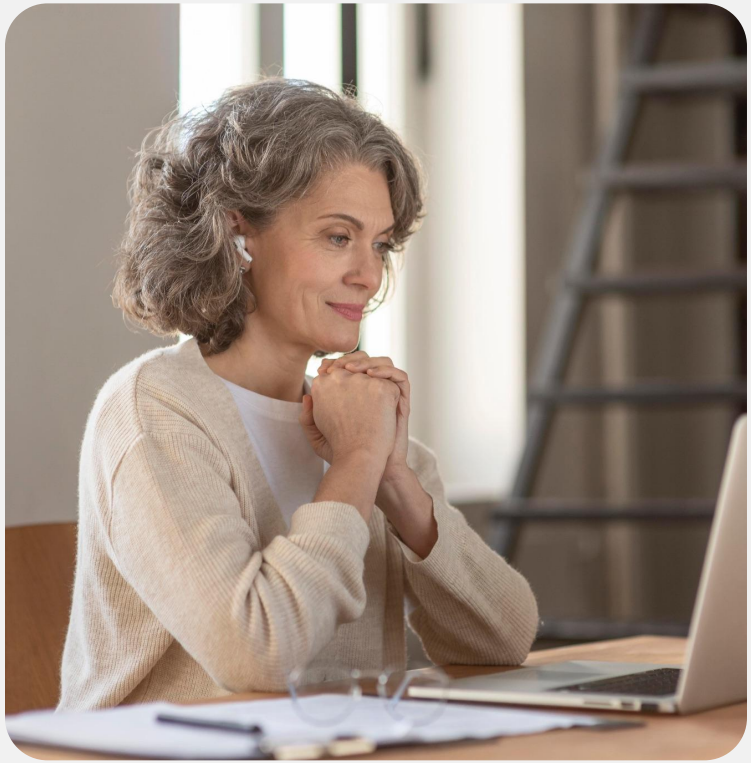
	Композиция	**Агрегация**
Создание	Объект создаёт внутренний сам	Объект получает готовый элемент извне
Связь	Жёсткая, вложенный неотделим	Слабая, вложенный может существовать отдельно
Жизненный цикл	Управляется внешним объектом	Независимый
Пример	`Car` создаёт `Engine` внутри себя	`Car` получает `Engine` как аргумент

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ

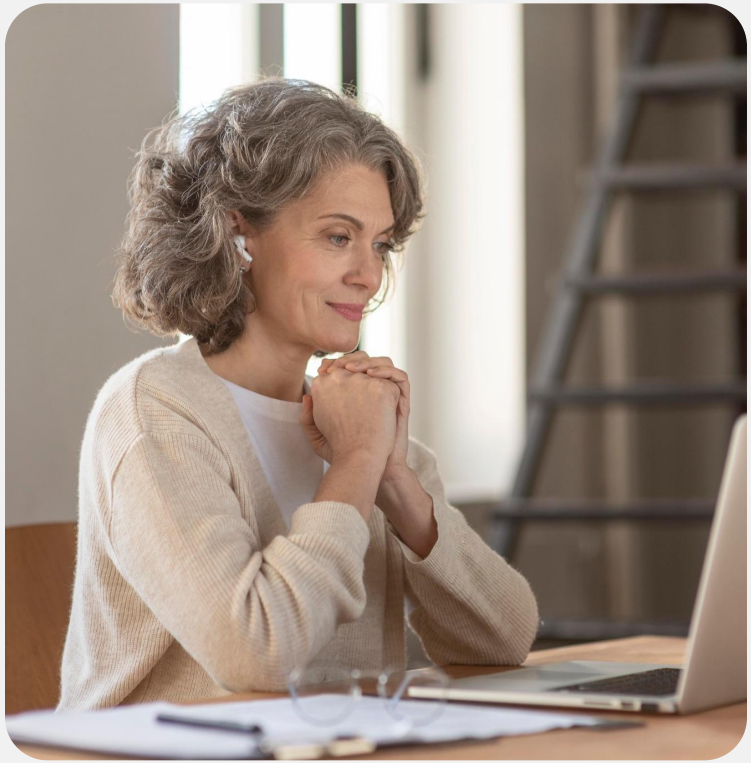


Выполните задание

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



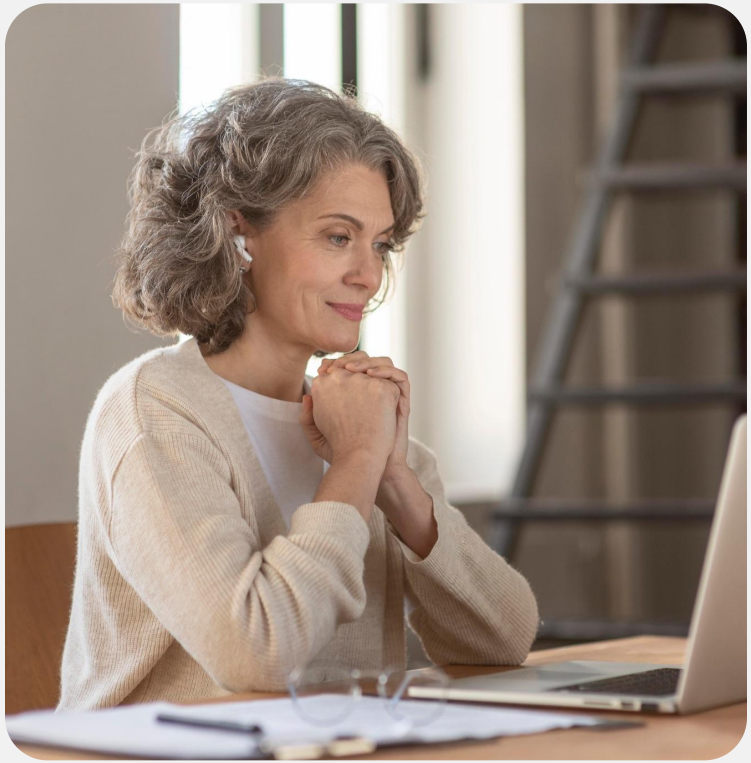
Выполните задание

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

Ответ: 1-b, 2-c, 3-a

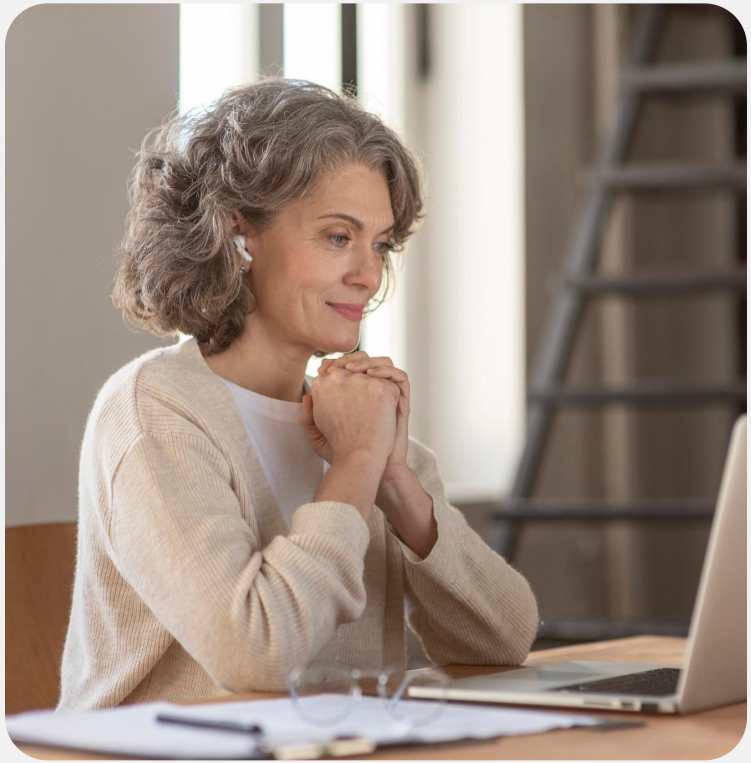


Выберите верный вариант ответа

2. Какая связь реализована в коде?

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

a. Композиция
b. Агрегация



Выберите верный вариант ответа

2. Какая связь реализована в коде?

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

a. Композиция
b. Агрегация

IT CAREER HUB

ВОПРОСЫ

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.

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:

```
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!
```

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. Воспроизведение мультимедиа

Создайте два класса:

Класс 1

`AudioFileMixin` — требует наличие поля `audio_tracks` (список треков).

Метод `play_audio()` выводит:

```
Воспроизведение аудио для <НазваниеКласса>:
    <название трека>
    <название трека>
```

Класс 2

`VideoFileMixin` – требует наличие поля `video_files` (список видео).

Метод `play_video()` выводит:

```
Воспроизведение видео для <НазваниеКласса>:
    <название видео>
    <название видео>
```

Если нужное поле отсутствует – выбрасывайте `AttributeError`.

Домашнее задание

2. Устройства

Создайте два класса:

- `MediaPlayer` — поддерживает только аудио. Принимает список треков.
- `Laptop` — поддерживает аудио и видео. Принимает списки треков и видео.

Проверьте работу классов, вызвав методы воспроизведения.

Заключение