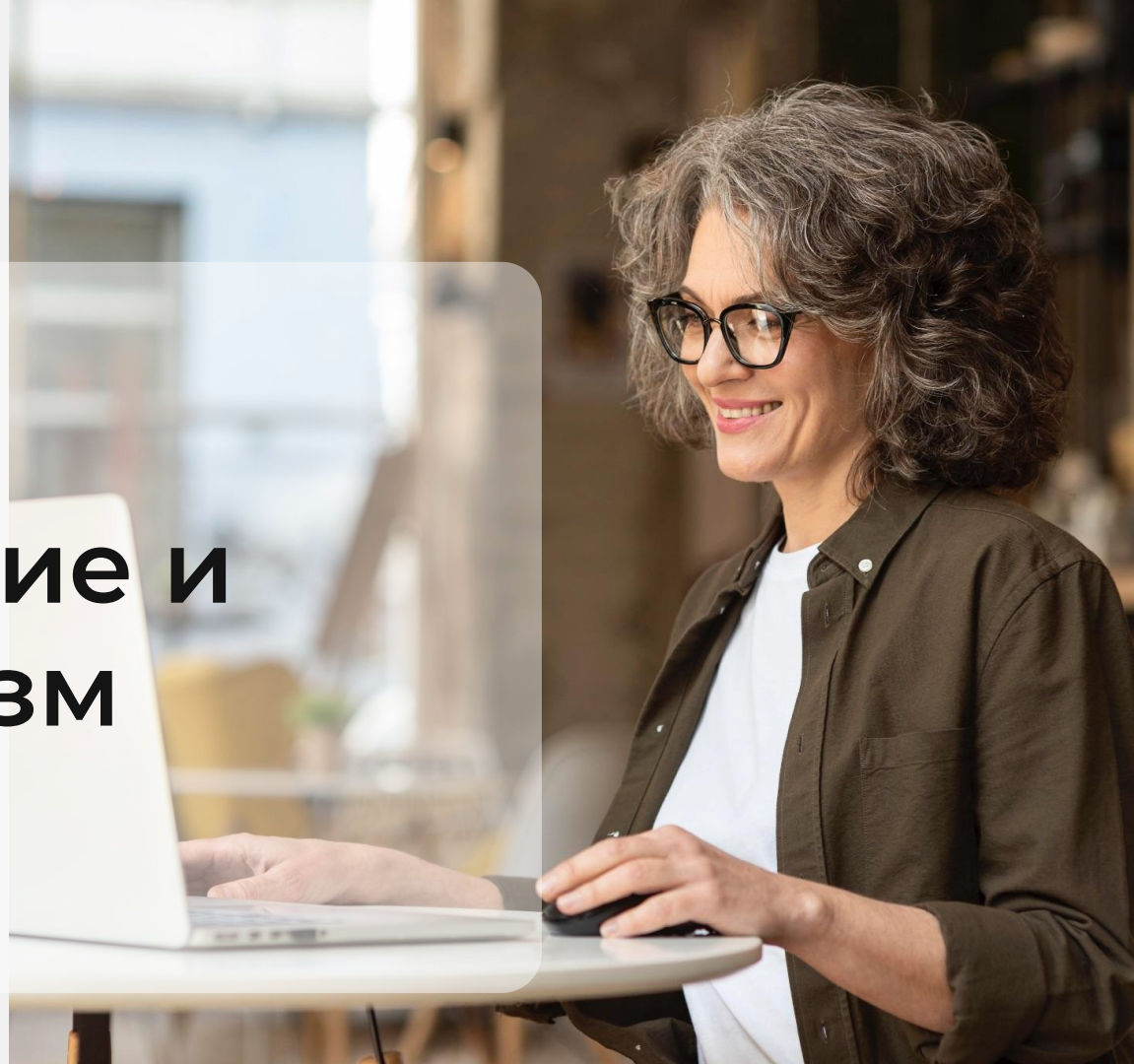
IT CAREER HUB

Python

Наследование и полиморфизм



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] Поля класса
- [x] Доступ к полям
- [x] Поля объекта по умолчанию
- [x] Классовые методы
- [x] Статические методы
- [x] Магический метод __str__
- [x] Магический метод __repr__

План занятия

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

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

IT CAREER HUB

Принципы ООП

Принципы ООП

Наследование

Один класс может **унаследовать** свойства и поведение другого. Это позволяет **избегать дублирования** кода и **переиспользовать логику**

Инкапсуляция

Объект **скрывает внутреннее устройство** и предоставляет **только нужный интерфейс**. Позволяет защищать данные и контролировать доступ к ним

Полиморфизм

Одинаковый интерфейс может **вести себя по-разному** в зависимости от типа объекта. Позволяет использовать **одинаковый код для разных классов**

Абстракция

Выделение **общих характеристик и поведения**, чтобы задать интерфейс для будущих реализаций в наследниках

Знание принципов

- Помогает понять, как правильно проектировать классы и объекты
- Упрощают понимание механизмов наследования, переопределения и взаимодействия объектов
- Позволяют писать гибкий, расширяемый и поддерживаемый код

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Наследование

Полезно знать

ICH
IT CAREER HUB

Наследование

Это механизм, который позволяет одному классу перенять свойства и поведение другого. Это один из ключевых принципов ООП, который помогает избегать дублирования кода и повторно использовать общую логику.

Наследование

Родительский класс, базовый класс или суперкласс	Дочерний класс, производный класс или подкласс
Класс, от которого наследуются	Класс, который наследует

Польза наследования

- Избежание дублирования кода, выделив общее поведение в один базовый класс
- Удобство расширять существующую логику без переписывания старых классов
- Работа с разными объектами одинаковым образом, если они наследуются от общего предка

Наследование: синтаксис

```
class Parent:
    # родительский класс
    ...

class Child(Parent):
    # дочерний класс, наследует всё от Parent
    ...
```

Наследование

Пример

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

    def work(self):
        print(f"{self.name} is working...")

class Programmer(Employee):
    pass

class Manager(Employee):
    pass

programmer = Programmer("Alice")
manager = Manager("Bob")

programmer.work()
manager.work()
```

Пояснение

- `Employee` — базовый класс, который хранит имя сотрудника и умеет работать.
- `Programmer` и `Manager` **наследуют всё от** `Employee`: и поле `name`, и метод `work()`.
- Каждый из них может использовать эти возможности **без повторного определения**.

Особенности

- Дочерний класс наследует все поля и методы родительского класса
- Можно добавлять свои методы и поля, не затрагивая родителя
- Можно переопределять поведение, если нужно изменить работу унаследованного метода

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Полиморфизм

Полезно знать

Полиморфизм

(от греч. «много форм»)
Это возможность использовать один и тот же интерфейс для разных типов объектов.

Польза полиморфизма

- Позволяет писать универсальный код, не заботясь о типе объекта
- Упрощает расширение программы — можно добавлять новые типы, не меняя старую логику
- Делает код гибким и читаемым

Механизмы полиморфизма

Переопределение методов (overriding)	Перегрузка (overloading)
Когда дочерний класс заново определяет метод, унаследованный от родителя. При вызове такого метода используется реализация из дочернего класса.	Когда один и тот же метод может вести себя по-разному в зависимости от переданных аргументов

Пример: переопределение метода

```python
class Employee:

    def __init__(self, name):
        self.name = name

    def work(self):
        print(f"{self.name} is working...")

class Programmer(Employee):
    def work(self):
        print(f"{self.name} writing code...")

class Manager(Employee):
    def work(self):
        print(f"{self.name} managing team...")


staff = [Programmer("Alice"), Manager("Bob"), Programmer("Bill")]

for person in staff:
    person.work()  # Поведение зависит от конкретного типа
```

- У всех сотрудников есть метод `work()`, но он реализован по-разному.
- В списке `staff` мы храним объекты разных классов: `Programmer`, `Manager`.
- При вызове `person.work()` Python **автоматически вызывает нужную версию метода** в зависимости от того, к какому классу относится объект.
- Это и есть **полиморфизм**: **один вызов → разное поведение в зависимости от типа объекта**.

Пример: попытка перегрузки

```
class Math:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

m = Math()
print(m.add(1, 2, 3))   # Работает
print(m.add(1, 2))      # Ошибка!
```

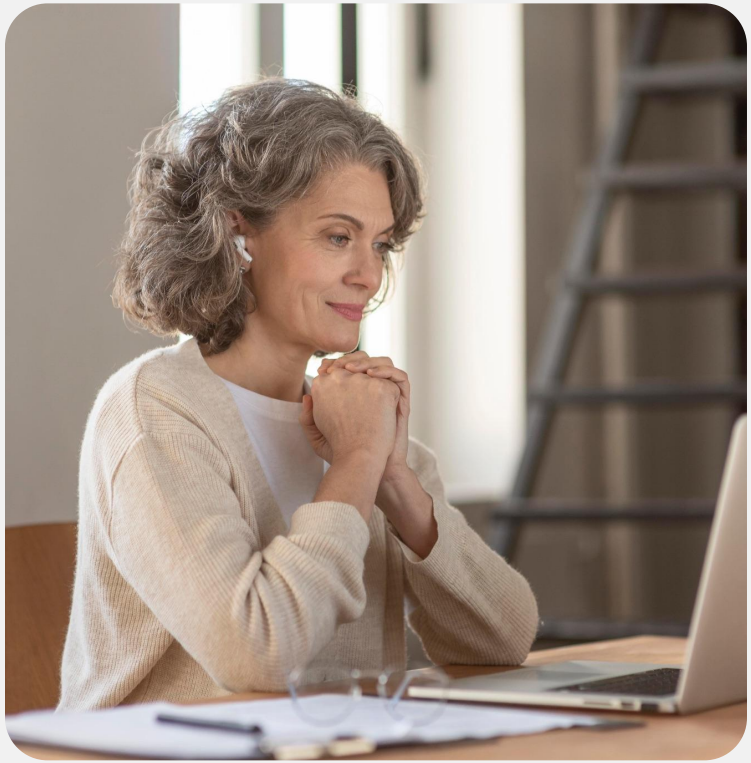
- Python не поддерживает **перегрузку** методов по количеству аргументов.
- В примере выше вторая версия `add()` просто **заменяет** первую — Python запоминает только последнее определение.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ



Выберите верный вариант ответа

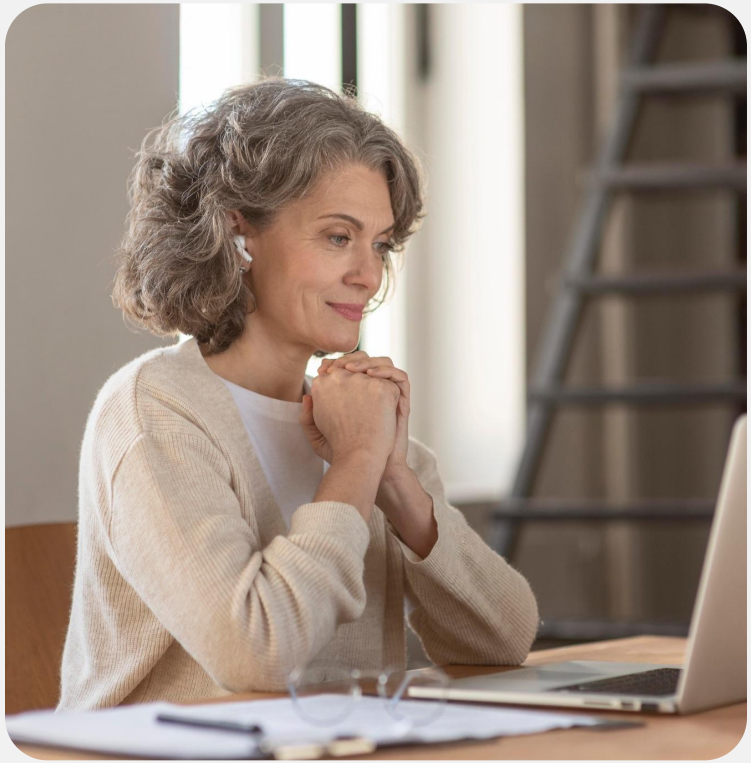
1. **Что произойдёт при выполнении следующего кода?**

```
class Employee:
    def work(self):
        print("Employee is working")

class Manager(Employee):
    pass

person = Manager()
person.work()
```

a. Возникнет ошибка, так как `Manager` не имеет метода `work()`
b. Будет напечатано: `Employee is working`
c. Будет напечатано: `None`
d. Метод `work()` выполнится, но ничего не выведет



Выберите верный вариант ответа

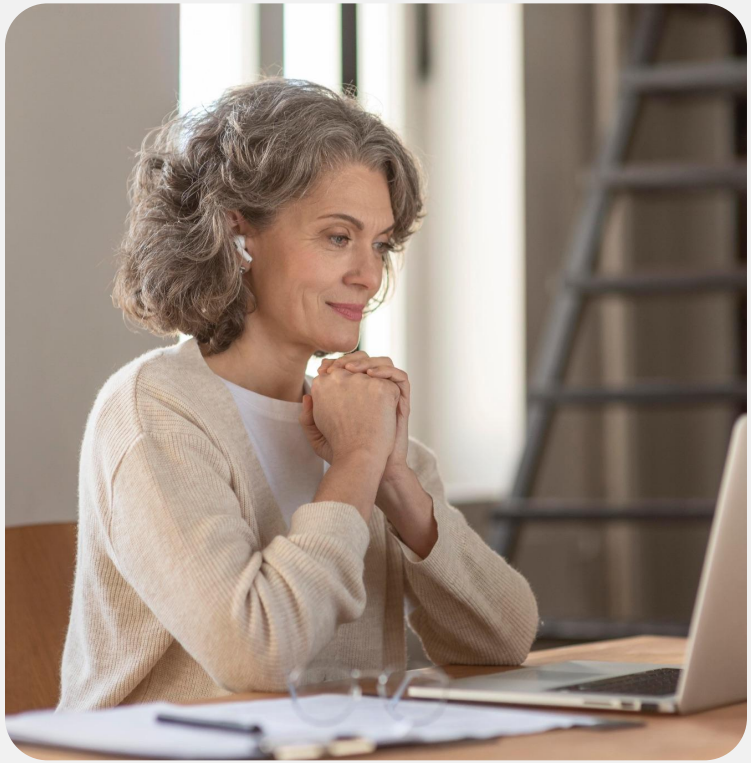
1. **Что произойдёт при выполнении следующего кода?**

```
class Employee:
    def work(self):
        print("Employee is working")

class Manager(Employee):
    pass

person = Manager()
person.work()
```

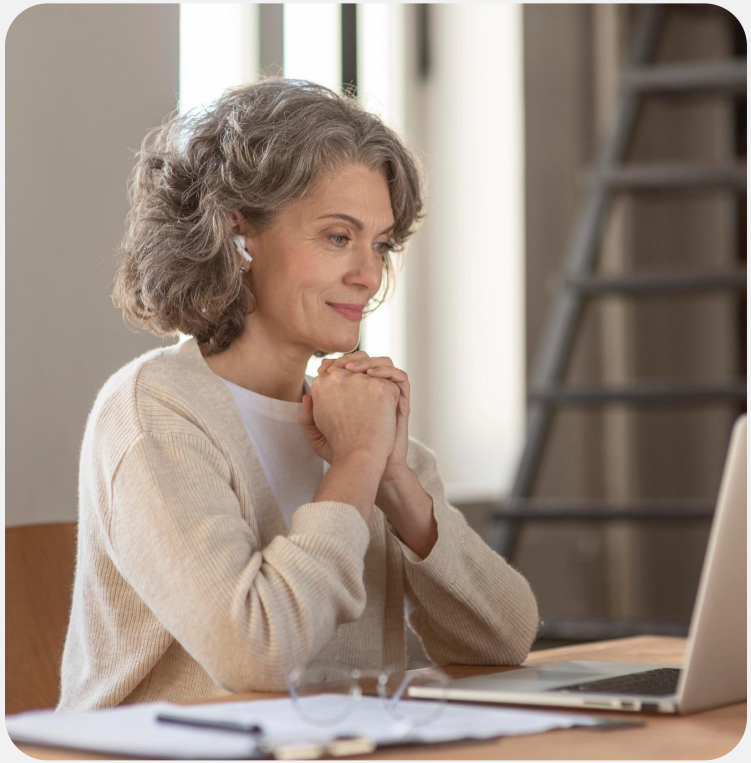
a. Возникнет ошибка, так как `Manager` не имеет метода `work()`
b. Будет напечатано: `Employee is working`
c. Будет напечатано: `None`
d. Метод `work()` выполнится, но ничего не выведет



Выберите верные варианты ответа

2. Укажи, что относится к преимуществам наследования:

a. Повышение безопасности данных
b. Возможность писать меньше кода
c. Возможность скрыть поля от внешнего доступа
d. Возможность переиспользовать логику



Выберите верные варианты ответа

2. Укажи, что относится к преимуществам наследования:

a. Повышение безопасности данных
b. Возможность писать меньше кода
c. Возможность скрыть поля от внешнего доступа
d. Возможность переиспользовать логику

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Функция super

Важно

При наследовании часто возникает ситуация, когда родительский класс уже задаёт общие поля в `__init__()`, а в дочернем классе необходимо добавить новые поля. Чтобы **не дублировать одинаковый код** используется функция `super()` — она вызывает ближайший метод родителя.

Пример: неправильный способ - дублируем код родителя

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

class Programmer(Employee):
    def __init__(self, name, language):
        self.name = name  # повторяем то же, что уже делает
родитель
        self.language = language

class Manager(Employee):
    def __init__(self, name, department):
        self.name = name  # снова повторяем
        self.department = department
```

Такой код **нарушает DRY** (Don't Repeat Yourself) и делает поддержку сложнее: если поведение `Employee.__init__` поменяется — `Programmer` об этом не узнает

Пример: альтернативный неправильный способ - вызываем код родителя

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

class Programmer(Employee):
    def __init__(self, name, language):
        Employee.__init__(self, name)  # явный вызов
родителя
        self.language = language
```

Этот способ работает, но считается **плохой практикой**:

- Привязан к имени родительского класса
- Может некорректно работать при изменении иерархии
- Не учитывает порядок разрешения методов

Пример: правильный способ - использование super()

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

class Programmer(Employee):
    def __init__(self, name, language):
        super().__init__(name)  # вызываем родительский __init__
        self.language = language

class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)  # вызываем родительский __init__
        self.department = department

p = Programmer("Alice", "Python")
print(p.name)
print(p.language)
```

Функция super в цепочке наследования

Когда у нас есть **несколько уровней наследования**, `super` позволяет вызывать инициализацию каждого уровня **без жёсткой привязки к имени родительского класса**.

Пример: многоуровневое наследование

```
class Person:
    def __init__(self, name):
        self.name = name
        print(f"Init Person: {self.name}")

class Employee(Person):
    def work(self):
        print(f"{self.name} is working...")


class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)
        self.department = department
        print(f"Init Manager: {self.name} manages {self.department}")

m = Manager("Alice", "Development")
```

Функция super в обычных методах

Функция `super` полезна не только в `__init__()`, но и **в любых других методах**, где нужно **расширить поведение родителя**, а не полностью его заменить.

Пример: расширение обычного метода

```
class Employee:
    def work(self):
        print("Employee is doing general tasks.")

class Programmer(Employee):
    def work(self):
        super().work()  # вызываем метод родителя
        print("Programmer is writing code.")

class Manager(Employee):
    def work(self):
        super().work()
        print("Manager is holding a meeting.")

staff = [Programmer(), Manager()]
for person in staff:
    person.work()
    print()
```

Эффективность super

- Автоматически находит метод в ближайшем родителе
- Позволяет избежать дублирования кода
- Гарантирует корректную работу при изменении иерархии классов

Использование super

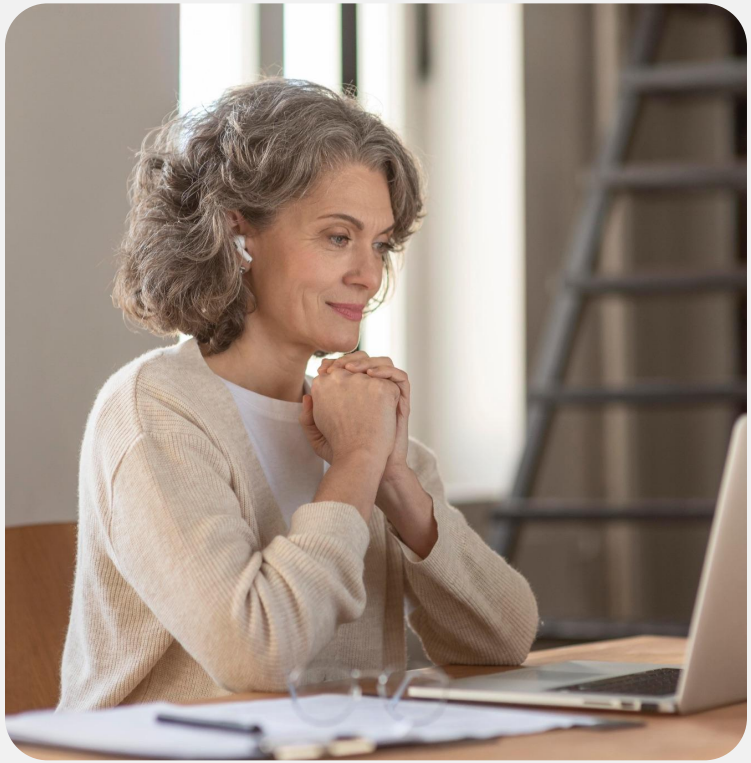
- Когда **родитель уже делает нужную инициализацию**, и мы хотим её сохранить
- Когда **расширяем** (а не полностью заменяем) поведение метода
- Особенно важно при **множественном наследовании** — `super()` автоматически учитывает порядок вызова

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ

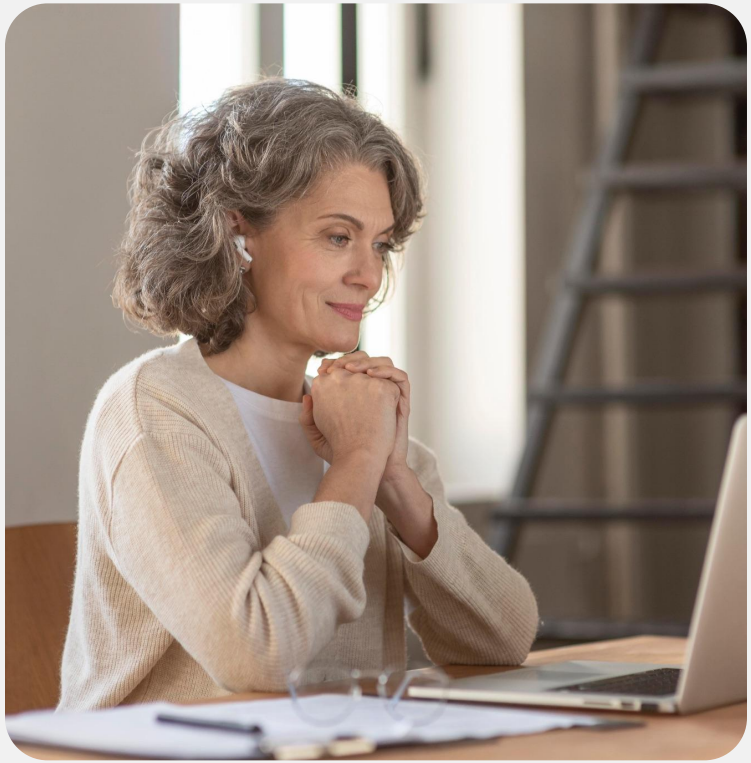


Выберите верные варианты ответа

1. **Найди ошибку в коде:**

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

class Programmer(Employee):
    def __init__(self, name, language):
        super(name).__init__()
        self.language = language
```



Выберите верные варианты ответа

1. **Найди ошибку в коде:**

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

class Programmer(Employee):
    def __init__(self, name, language):
        super(name).__init__()
        self.language = language
```

Ответ: Неверное использование `super`: должно быть `super().__init__(name)`

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

Наследование от object

Полезно знать

ICH
IT CAREER HUB

Класс object

Это корневой класс всей иерархии классов в Python. Он предоставляет набор базовых методов, таких как `__str__()`, `__init__()` и другие.

Класс object

Пример

```
class Book:
    pass
```

Пояснение

Python воспринимает это так:

```
class Book(object):
    pass
```

Методы от object

Пример

```
class Book:
    pass

b = Book()

print(b.__str__())  # Вызов метода
__str__ напрямую
print(b)            # Вызов метода
__str__
```

Пояснение

Метод `__str__()` **присутствует в классе без собственной реализации**, поскольку он унаследован от базового класса `object`

Преимущества наследования от object

- Все классы получают **единое поведение по умолчанию**
- Все объекты можно безопасно использовать с базовыми функциями и методами
- Это делает классы совместимыми со встроенными механизмами Python (например, `print()` вызывает `__str__()` из `object`, если не переопределено)

IT CAREER HUB

ВОПРОСЫ

ICH IT CAREER HUB

Функции isinstance и issubclass

Важно

Python предоставляет удобные встроенные функции для **проверки типов и иерархии классов**, что особенно полезно при работе с **наследованием** и **полиморфизмом**.

Полезно знать

isinstance

Это встроенная функция Python, которая позволяет проверить, является ли объект экземпляром определённого класса или его подкласса, чтобы выбрать правильное поведение

Особенности isinstance

Если объект принадлежит **указанному классу или его наследнику**, функция возвращает `True`, в противном случае — `False`.

Функция `isinstance` также работает со **встроенными типами** (`str`, `list`, `int` и т.д.)

Функция isinstance

Синтаксис

```
isinstance(obj, some_class)
isinstance(obj, tuple_of_classes)
```

Пояснение

- `obj` — объект, который необходимо проверить
- `some_class` — класс, принадлежность к которому необходимо проверить
- `tuple_of_classes` — кортеж из нескольких классов, принадлежность к любому из которых необходимо проверить

Пример

```
class Employee:
    pass

class Programmer(Employee):
    pass

class Manager(Employee):
    pass

e = Employee()
p = Programmer()
m = Manager()

print(isinstance(p, Programmer))  # экземпляр класса
print(isinstance(p, Employee))    # экземпляр наследника
print(isinstance(p, Manager))     # Manager не находится выше в иерархии
print(isinstance(p, object))      # True – все классы наследуют от object
```

Пример использования в коде

```python
class Employee:
    def work(self):
        print("Выполняет общие задачи")

class Programmer(Employee):
    def write_code(self):
        print("Пишет код")

class BackendDeveloper(Programmer):
    def write_code(self):
        print("Пишет серверный код")

class FrontendDeveloper(Programmer):
    def write_code(self):
        print("Пишет интерфейс")
```

1

```python
class Manager(Employee):
    def work(self):
        print("Проводит собрание")


staff = [
    Programmer(),
    BackendDeveloper(),
    FrontendDeveloper(),
    Manager(),
    Employee()
]

for person in staff:
    if isinstance(person, Programmer):
        person.write_code()
    else:
        person.work()
```

2

Проверка на несколько классов

Пример

```
print(isinstance("hello", (str, int)))  # Строка принадлежит к одному из классов
```

Пояснение

Можно проверить, принадлежит ли объект **к одному из нескольких классов**, передав кортеж

Полезно знать

IT CAREER HUB

issubclass

Это встроенная функция Python, которая позволяет проверить, является ли один класс подклассом другого, то есть наследуется ли он от указанного класса

Особенности issubclass

Функция возвращает `True`, если класс **наследуется от другого класса напрямую или через цепочку**, и `False` — если нет.

Функция issubclass

Синтаксис

```
issubclass(class_a, class_b)
issubclass(class_a, tuple_of_classes)
```

Пояснение

- `class_a` — класс, который мы проверяем
- `class_b` — предполагаемый родительский класс
- `tuple_of_classes` — кортеж из нескольких классов

Пример

```
class Employee:
    pass

class Programmer(Employee):
    pass

class Manager(Employee):
    pass

class BackendDeveloper(Programmer):
    pass

print(issubclass(Programmer, Employee))         # Прямой потомок
print(issubclass(BackendDeveloper, Programmer)) # Прямой потомок
print(issubclass(BackendDeveloper, Employee))   # Потомок через цепочку
print(issubclass(Manager, Programmer))          # Разные ветки иерархии
print(issubclass(Employee, object))             # Все классы наследуют от object
```

Пример использования в коде

```
class Employee:
    def send_welcome_email(self):
        print("Добро пожаловать в компанию!")

class Programmer(Employee):
    def send_welcome_email(self):
        print("Добро пожаловать, разработчик! Не
забудьте подключиться к репозиторию.")

class Manager(Employee):
    def send_welcome_email(self):
        print("Добро пожаловать, менеджер!
Сегодня у вас первое собрание с командой.")
```

1

```
def hire_employee(cls):
    if not issubclass(cls, Employee):
        raise ValueError("Можно нанимать только
классы, основанные на Employee")

    person = cls()
    person.send_welcome_email()

hire_employee(Programmer)
hire_employee(Manager)
# hire_employee(str)  # ValueError: Не наследник
Employee
```

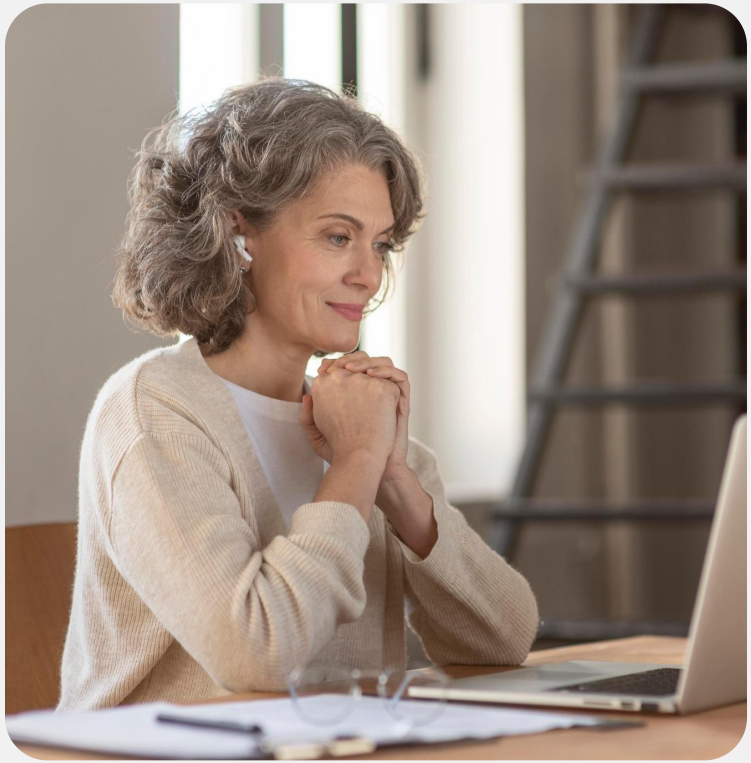
2

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ

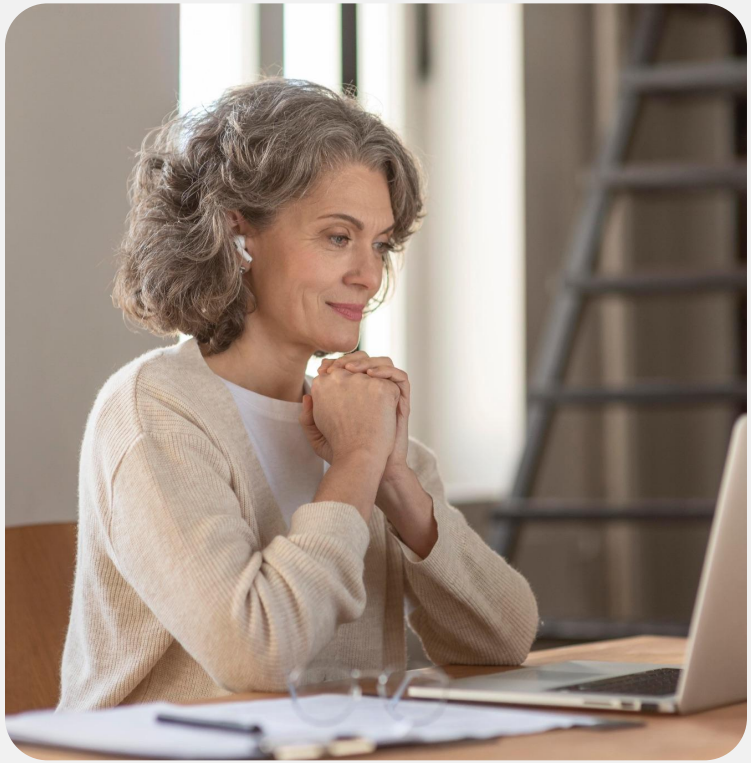


Выберите верные варианты ответа

1. **Что вернёт функция issubclass() и почему?**

```
class Book:
    pass

b = Book()
print(issubclass(b, object))
```



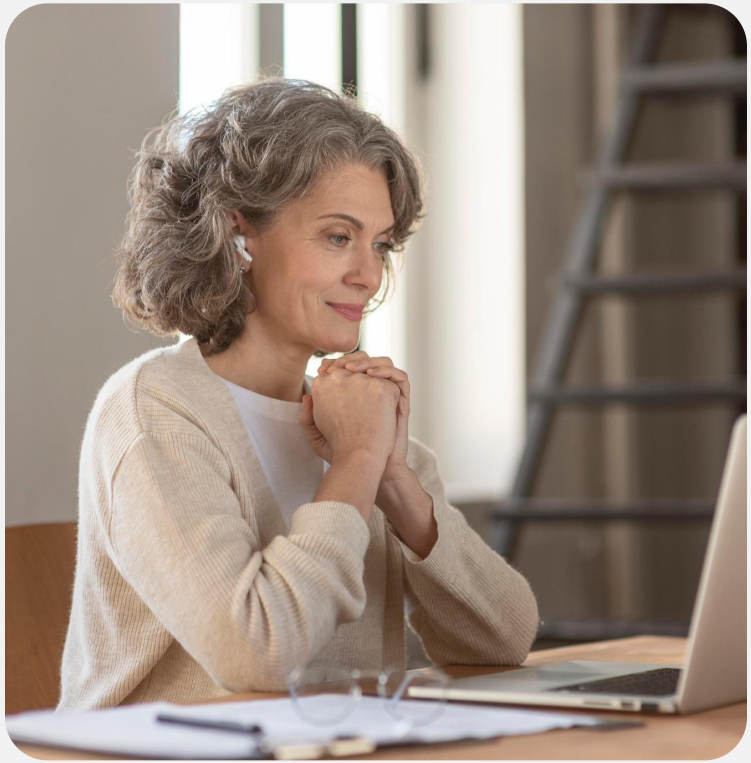
Выберите верные варианты ответа

1. **Что вернёт функция issubclass() и почему?**

```
class Book:
    pass

b = Book()
print(issubclass(b, object))
```

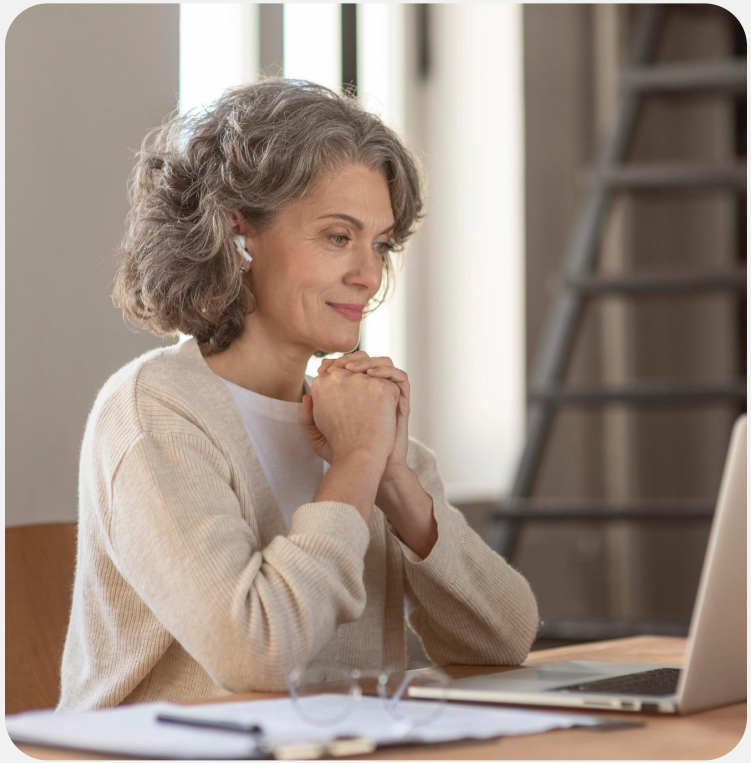
Ответ: `TypeError`, так как функция `issubclass()` принимает класс, а не объект.



Выберите верные варианты ответа

2. Для чего используется функция isinstance?

a. Проверяет, является ли объект экземпляром заданного класса или его наследников
b. Проверяет, что класс унаследован от object
c. Проверяет тип переменной только для встроенных типов



Выберите верные варианты ответа

2. Для чего используется функция isinstance?

a. Проверяет, является ли объект экземпляром заданного класса или его наследников
b. Проверяет, что класс унаследован от object
c. Проверяет тип переменной только для встроенных типов

Ответ: a

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ПРАКТИЧЕСКАЯ РАБОТА

1. Класс Employee

Создайте класс `Employee`, представляющий сотрудника.

- У каждого объекта должно быть поле `name`.
- Метод `work()` выводит строку: `<имя> is working...`.
- Проверьте работу класса, создав сотрудника и вызвав метод `work()`.

Пример вывода:

```
Alice is working...
```

2. Класс Developer

Создайте класс `Developer`, который расширяет `Employee`.

- Добавьте дополнительное поле `language`.
- Переопределите метод `work()`, чтобы он включал сообщение из родительского метода и добавлял строку:
  `<имя> writes <язык> code.`
- Проверьте работу, создав объект `Developer` и вызвав метод `work()`.

Пример вывода:

```
Bob is working...
Bob writes Python code.
```

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. Класс Person

Создайте класс `Person`, представляющий человека.

- Каждый человек должен иметь имя.
- Добавьте метод `introduce()`, который выводит приветствие с именем.

Пример вывода:

```
Hello, my name is Alice.
```

Домашнее задание

2. Класс Student

На основе класса `Person` создайте класс `Student`.

- Студент должен иметь имя и номер курса.
- Метод `introduce()` должен сначала выводить базовое приветствие, а затем строку: `I'm on course <номер_курса>.`

Пример вывода:

```
Hello, my name is Alice.
I'm on course 2.
```

Домашнее задание

3. Класс Teacher и список людей

На основе класса `Person` создайте класс `Teacher`.

- У преподавателя есть имя и предмет.
- Метод `introduce()` должен выводить имя и предмет.
- Метод `introduce()` должен выводить строку: `Hello, I am professor <имя>. My subject is <предмет>.`
- Создайте список, в котором будут `Student` и `Teacher`, и вызовите у всех метод `introduce()`.

Пример вывода:

```
Hello, my name is Alice.
I'm on course 2.
Hello, I am professor Bob.
My subject is Mathematics
```

Заключение