IT CAREER HUB

# Урок 39.
# Абстрактные классы и пользовательские исключения

IT CAREER HUB

# Абстрактные классы

**Абстрактный класс — это класс, который не предназначен для создания объектов напрямую. Он служит как шаблон для других классов, задавая структуру и обязательные методы, которые должны быть реализованы в дочерних классах.**

### Зачем нужны абстрактные классы

- Чтобы **определить единый интерфейс** для группы классов
- Чтобы **заставить наследников реализовать нужные методы**
- Чтобы **описать общую логику**, но **запретить создание "незаконченных" объектов**

### Отличие от обычного класса

| **Обычный класс** | **Абстрактный класс** |
|---|---|
| Можно создавать объекты | Создание объектов запрещено |
| Все методы можно переопределить, но не обязательно | Некоторые методы **обязательны к реализации** |
| Используется напрямую | Используется как основа для наследования |

### Модуль abc и декоратор @abstractmethod

В Python абстрактные классы создаются с помощью модуля `abc` (*Abstract Base Classes*) и декоратора `@abstractmethod`:

- Класс должен **наследоваться от** `ABC` (из модуля abc)
- Абстрактные методы помечаются декоратором `@abstractmethod`

**Пример**

```python
from abc import ABC, abstractmethod

class Employee(ABC):  # Абстрактный класс

    @abstractmethod
    def work(self):
        pass  # Метод без реализации

e = Employee()  # TypeError: не реализован абстрактный метод
```

**Пример: Создание подкласса, реализующего метод**

```python
class Programmer(Employee):
    def work(self):
        print("Write code")

p = Programmer()
p.work()
```

**Особенности:**

- Абстрактный класс не может быть использован напрямую — он является **шаблоном**
- Класс может **содержать как абстрактные, так и обычные (реализованные) методы** — при этом он остаётся абстрактным, пока не будут реализованы все `@abstractmethod` в наследниках

IT CAREER HUB

# Документация классов

В Python для каждого класса можно (и нужно) писать **документацию**, которая объясняет **назначение класса**, его **поведение**, **поля** и **методы**. Это упрощает чтение кода, помогает другим разработчикам и отображается в `help()`.

### Как оформляется документация

Документация класса пишется в виде **многострочной строки** (`docstring`) сразу **под объявлением класса** в тройных кавычках `"""..."""`.

**Пример**

```python
class Book:
    """
    Represents a book.

    Attributes:
        title (str): The title of the book.
        author (str): The author of the book.

    Methods:
        get_info(): Returns a brief description of the book.
    """

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

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

print(Book.__doc__)
```

**Рекомендации по документации:**

- Кратко опишите, **что делает класс**
- Перечислите **основные атрибуты и методы**
- Используйте **одинаковый стиль** по всему проекту

IT CAREER HUB

# Задания для закрепления 1

1. **Укажите верные утверждения об абстрактных классах:**

a. Абстрактный класс можно использовать для создания объектов напрямую
b. Абстрактный класс задаёт структуру для наследников
c. Абстрактный класс может содержать как реализованные, так и абстрактные методы
d. Абстрактный класс запрещает наследование

Посмотреть ответ

**2. Что произойдет при выполнении следующего кода?**

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    pass

d = Dog()
d.speak()
```

Посмотреть ответ

IT CAREER HUB

# Пользовательские исключения

В Python можно не только использовать встроенные типы ошибок (`ValueError`, `TypeError`, `ZeroDivisionError` и др.), но и **создавать собственные исключения**, которые лучше отражают **контекст конкретной задачи**.

Такие исключения называются **пользовательскими**. Они позволяют точно указать, **что именно пошло не так**, и делают код **более читаемым и контролируемым**.

### Зачем нужны пользовательские исключения

- Чтобы **ясно отделить** ошибки своей логики от стандартных
- Чтобы **точно указать причину ошибки**, например `InvalidTemperatureError`, `LoginFailedError`
- Чтобы можно было **отлавливать их отдельно** в `try/except`

### Создание пользовательских исключений

Пользовательское исключение — это **обычный класс**, который **наследуется от** `Exception` или одного из его подклассов.

**Синтаксис:**

```python
class CustomError(Exception):
    pass

raise CustomError("Error message")
```

- `CustomError` — имя нового класса ошибки (по соглашению заканчивается на `Error`)
- `Exception` — базовый класс, от которого наследуется логика

**Особенности:**

- Пользовательские исключения должны наследоваться от `Exception` **или его подклассов** (`ValueError`, `RuntimeError` и т.д.)

- Не следует наследоваться от `BaseException`, так как он предназначен для системных исключений
- Класс может быть пустым (`pass`) или содержать собственную логику
- Названия пользовательских исключений принято заканчивать на `Error`

**Пример: ограничение доступа по возрасту**

```python
class AccessDeniedError(Exception):
    """Вызывается, если пользователь слишком молод для доступа."""
    pass


def check_age(age):
    if age < 18:
        raise AccessDeniedError("Access denied: age must be at least 18")
    print("Access granted")

try:
    check_age(16)
except AccessDeniedError as e:  # можно отловить собственную ошибку
    print("Ошибка:", e)
```

- Класс `AccessDeniedError` **наследуется от** `Exception` — это базовый подход для пользовательских ошибок
- Внутри `try/except` можно **перехватить собственную ошибку**, не затрагивая другие

**Дополнительные поля в исключениях**

Пользовательские исключения можно не только использовать с сообщением, но и **сохранять в них данные**, которые помогут при обработке ошибки: например, **аргументы**, вызвавшие сбой.

Это позволяет:

- делать исключения **более информативными**
- передавать в обработчик **контекст ошибки**
- облегчить **логирование и отладку**

### Пример: пользовательская ошибка с данными

```python
class AccessDeniedError(Exception):
    """Вызывается, если пользователь слишком молод для доступа."""

    def __init__(self, age):
        self.age = age
        super().__init__(f"Access denied: age {age} is too low")


def check_age(age):
    if age < 18:
        raise AccessDeniedError(age)
    print("Access granted")


try:
    check_age(15)
except AccessDeniedError as e:
    print("Error:", e)
    print("Age:", e.age)
```

- Можно сохранять **дополнительные данные внутри исключения**
- Удобно при логировании, отладке и тестировании
- `super().__init__()` передаёт сообщение в базовый `Exception`, чтобы его можно было отобразить обычным `print(e)`

### Наследование от стандартных исключений

Если ваше исключение относится к определённому типу ошибки — например, связано с **неправильным значением** или **типом данных**, — то его лучше делать **наследником подходящего встроенного исключения**, а не просто `Exception`.

IT CAREER HUB

## Пример: собственная ошибка значения (ValueError)

Python

```python
class TemperatureTooLowError(ValueError):
    pass

def set_temperature(value):
    if value < -273.15:
        raise TemperatureTooLowError("Temperature cannot be below absolute zero")
    print(f"Temperature set to {value}°C")

set_temperature(15)
set_temperature(-300)  # вызовет ошибку
```

IT CAREER HUB

# Задания для закрепления 2

**Укажите, в каких случаях имеет смысл создавать пользовательские исключения:**

a. Чтобы обработать ошибку, связанную с некорректным типом
b. Чтобы явно разделить ошибки бизнес-логики от стандартных ошибок
c. Чтобы ускорить выполнение программы

Посмотреть ответ

IT CAREER HUB

# Магические методы

Магические методы (dunder-методы) позволяют классу **встраиваться в поведение самого языка Python** и вести себя **как встроенные типы данных** (строки, списки, числа и т.д.) в стандартных ситуациях.

### Что делают магические методы

- Управляют **созданием и инициализацией** объектов
- Отвечают за **представление объекта**
- Определяют поведение **при сравнении**, **арифметике**, **в коллекциях** и т.п.
- Позволяют делать объекты **итерируемыми**, **вызываемыми** и т.п.

### Примеры ситуаций, где вызываются магические методы

| Выражение | Вызываемый метод |
|---|---|
| `str(obj) или print()` | `__str__()` |
| `len(obj)` | `__len__()` |
| `obj1 == obj2` | `__eq__()` |
| `obj1 + obj2` | `__add__()` |
| `item in obj` | `__contains__()` |
| `obj()` | `__call__()` |
| `bool(obj)` | `__bool__()` |

IT CAREER HUB

# Магические методы итерации

Чтобы объекты можно было использовать в `for`, `in`, `list()`, `sum()` и других **итерационных конструкциях**, они должны поддерживать **протокол итерации**.

Для этого нужно реализовать **два магических метода**:

| Метод | Назначение |
|---|---|
| `__iter__()` | Возвращает **итератор** — объект с `__next__()` |
| `__next__()` | Возвращает **следующее значение** или выбрасывает `StopIteration` |

**Пример: итератор, с нарастающей суммой элементов**

```python
class CumulativeSum:
    def __init__(self, numbers):
        self.numbers = numbers
        self.index = 0
        self.total = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.numbers):
            raise StopIteration
        self.total += self.numbers[self.index]
        self.index += 1
        return self.total


data = [3, 5, 2, 4]
acc = CumulativeSum(data)

for value in acc:
    print(value)
```

IT CAREER HUB

# Задания для закрепления 3

**В каких случаях необходимо определить метод __iter__()?**

a. Когда объект должен поддерживать итерацию через `for`
b. Чтобы объект можно было передавать в функцию `len()`

Посмотреть ответ

IT CAREER HUB

# Ответы на задания

| **Задания на закрепление 1** | Вернуться к заданиям |
|---|---|
| 1. Верные утверждения об абстрактных классах | Ответ: b, c |
| 2. Результат выполнения кода | Ответ: Произойдет ошибка при попытке создать объект `Dog` |
| **Задания на закрепление 2** | Вернуться к заданиям |
| Использование пользовательских исключений | Ответ: b |
| **Задания на закрепление 3** | Вернуться к заданиям |
| метод `__iter__()` | Ответ: a |

IT CAREER HUB

# Практическая работа

**1. Онлайн-платёжные системы**

Создайте абстрактный класс `PaymentProcessor`.

- В классе должен быть метод `pay(amount)`.
- Реализуйте два класса:
    - `PaypalPayment`, который печатает `"Paid <amount> via PayPal"`.
    - `CreditCardPayment`, который печатает `"Paid <amount> via Credit Card"`.

**Решение:**

```python
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class PaypalPayment(PaymentProcessor):
    def pay(self, amount):
        print(f"Paid {amount} via PayPal.")

class CreditCardPayment(PaymentProcessor):
    def pay(self, amount):
        print(f"Paid {amount} via Credit Card.")

# Пример использования
payment1 = PaypalPayment()
payment2 = CreditCardPayment()

payment1.pay(100)
payment2.pay(200)
```

IT CAREER HUB

**2. Проверка платежей**

Доработайте систему:

- Создайте пользовательское исключение `InvalidPaymentError`.
- В каждом платёжном классе метод pay(amount) должен проверять сумму:
    - Если сумма меньше или равна нулю, выбрасывать `InvalidPaymentError`.
    - Иначе проводить платёж.

IT CAREER HUB

**Решение:**

```python
from abc import ABC, abstractmethod

class InvalidPaymentError(Exception):
    """Raised when the payment amount is invalid."""
    pass

class PaymentProcessor(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

    @staticmethod
    def _validate_amount(amount):
        if amount <= 0:
            raise InvalidPaymentError("Payment amount must be positive.")

class PaypalPayment(PaymentProcessor):
    def pay(self, amount):
        self._validate_amount(amount)
        print(f"Paid {amount} via PayPal.")

class CreditCardPayment(PaymentProcessor):
    def pay(self, amount):
        self._validate_amount(amount)
        print(f"Paid {amount} via Credit Card.")

# Пример использования
payments = [PaypalPayment(), CreditCardPayment()]

for payment in payments:
    try:
        payment.pay(100)
        payment.pay(-50)  # Ошибка
    except InvalidPaymentError as e:
        print("Error:", e)
```