Что такое наследование в Python

Наследование — это механизм ООП, позволяющий создавать новый класс на основе существующего. Дочерний класс автоматически получает все атрибуты и методы родительского класса, а также может добавлять собственные или переопределять унаследованные.

Синтаксис наследования

```
class BaseClass:
    def __init__(self, base_attr):
        self.base_attr = base_attr

    def base_method(self):
        print("Method from BaseClass")

class DerivedClass(BaseClass):
    def __init__(self, base_attr, derived_attr):
        super().__init__(base_attr)  # Вызов конструктора
родителя
        self.derived_attr = derived_attr

    def derived_method(self):
        print("Method from DerivedClass")

# Создание объекта дочернего класса
derived = DerivedClass("Base Attribute", "Derived Attribute")
derived.base_method()    # Вывод: Method from BaseClass
derived.derived_method() # Вывод: Method from DerivedClass
```

Функция super() в Python

Функция super() обеспечивает доступ к методам родительского класса, что особенно важно при переопределении методов:

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

    def get_info(self):
        return f"Employee: {self.name}, Salary: {self.salary}"


class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)  # Вызов конструктора
родителя
        self.department = department

    def get_info(self):
```

```
        base_info = super().get_info()  # Получение информации
от родителя
        return f"{base_info}, Department: {self.department}"


manager = Manager("John", 50000, "IT")
print(manager.get_info())  # Employee: John, Salary: 50000,
Department: IT
```

Полиморфизм в Python

Полиморфизм — это возможность объектов разных классов реагировать на одинаковые вызовы методов по-разному. В Python полиморфизм реализуется через переопределение методов.

Базовый пример полиморфизма

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

class Bird(Animal):
    def make_sound(self):
        return "Tweet!"

# Полиморфное использование
animals = [Dog(), Cat(), Bird()]

for animal in animals:
    print(animal.make_sound())
# Вывод:
# Woof!
# Meow!
# Tweet!
```

Практический пример: система платежей

```
class PaymentMethod:
    def process_payment(self, amount):
        raise NotImplementedError("Subclass must implement")


class CreditCard(PaymentMethod):
```

```
    def __init__(self, card_number):
        self.card_number = card_number

    def process_payment(self, amount):
        return f"Processing ${amount} via Credit Card ending in
{self.card_number[-4:]}"


class PayPal(PaymentMethod):
    def __init__(self, email):
        self.email = email

    def process_payment(self, amount):
        return f"Processing ${amount} via PayPal account
{self.email}"


class BankTransfer(PaymentMethod):
    def __init__(self, account_number):
        self.account_number = account_number

    def process_payment(self, amount):
        return f"Processing ${amount} via Bank Transfer to
{self.account_number}"


# Функция для обработки платежа любым способом
def make_payment(payment_method, amount):
    return payment_method.process_payment(amount)


# Использование
credit_card = CreditCard("1234567890123456")
paypal = PayPal("user@example.com")
bank_transfer = BankTransfer("ACC123456789")

print(make_payment(credit_card, 100))
print(make_payment(paypal, 50))
print(make_payment(bank_transfer, 200))
```

Расширенные примеры наследования и полиморфизма

Пример с геометрическими фигурами

```
import math


class Shape:
```

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

    def area(self):
        raise NotImplementedError("Subclass must implement area
method")

    def perimeter(self):
        raise NotImplementedError("Subclass must implement
perimeter method")

    def __str__(self):
        return f"{self.name} - Area: {self.area():.2f},
Perimeter: {self.perimeter():.2f}"


class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius


class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)


class Triangle(Shape):
    def __init__(self, a, b, c):
        super().__init__("Triangle")
        self.a = a
```

```
        self.b = b
        self.c = c

    def area(self):
        s = self.perimeter() / 2
        return math.sqrt(s * (s - self.a) * (s - self.b) * (s -
self.c))

    def perimeter(self):
        return self.a + self.b + self.c


# Создание коллекции фигур
shapes = [
    Circle(5),
    Rectangle(4, 6),
    Triangle(3, 4, 5)
]

# Полиморфное использование
for shape in shapes:
    print(shape)
```

Множественное наследование

Python поддерживает множественное наследование — возможность наследовать от нескольких классов:

```
class Flyable:
    def fly(self):
        return "Flying through the air"

class Swimmable:
    def swim(self):
        return "Swimming in water"

class Duck(Animal, Flyable, Swimmable):
    def make_sound(self):
        return "Quack!"

duck = Duck()
print(duck.make_sound())  # Quack!
print(duck.fly())         # Flying through the air
print(duck.swim())        # Swimming in water
```

Абстрактные классы и методы

Для создания более строгой иерархии классов используются абстрактные классы:

from abc import ABC, abstractmethod

```python
class Vehicle(ABC):
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

    def get_info(self):
        return f"{self.brand} {self.model}"


class Car(Vehicle):
    def start_engine(self):
        return "Car engine started"

    def stop_engine(self):
        return "Car engine stopped"


class Motorcycle(Vehicle):
    def start_engine(self):
        return "Motorcycle engine started"

    def stop_engine(self):
        return "Motorcycle engine stopped"


# Создание объектов
car = Car("Toyota", "Camry")
motorcycle = Motorcycle("Honda", "CBR")

print(car.get_info())  # Toyota Camry
print(car.start_engine())  # Car engine started
print(motorcycle.start_engine())  # Motorcycle engine started
```

Преимущества наследования и полиморфизма

Основные преимущества:

Повторное использование кода: Избежание дублирования кода за счет наследования общей функциональности.

Расширяемость: Легкое добавление новых классов без изменения существующего кода.

Полиморфизм: Единый интерфейс для работы с объектами разных типов.

Инкапсуляция: Скрытие деталей реализации и предоставление простого интерфейса.

Модульность: Разделение кода на логические компоненты.

Принцип подстановки Лисков

Важный принцип ООП: объекты суперкласса должны заменяться объектами подклассов без нарушения работы программы:

```
def process_shapes(shapes_list):
    total_area = 0
    for shape in shapes_list:
        total_area += shape.area()  # Работает для любого подкласса Shape
    return total_area

# Функция работает с любыми наследниками Shape
mixed_shapes = [Circle(3), Rectangle(2, 4), Triangle(3, 4, 5)]
total = process_shapes(mixed_shapes)
print(f"Total area: {total:.2f}")
```

Лучшие практики

1. Используйте композицию вместо наследования когда это уместно

```
class Engine:
    def __init__(self, power):
        self.power = power

    def start(self):
        return "Engine started"


class Car:
    def __init__(self, brand, engine):
        self.brand = brand
        self.engine = engine  # Композиция вместо наследования

    def start(self):
        return f"{self.brand}: {self.engine.start()}"
```

2. Переопределяйте методы осмысленно

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

    def __str__(self):
        return f"Animal: {self.name}"

    def __repr__(self):
        return f"Animal('{self.name}')"


class Dog(Animal):
    def __str__(self):
        return f"Dog: {self.name}"

    def __repr__(self):
        return f"Dog('{self.name}')"
```

3. Используйте isinstance() для проверки типов

```python
def handle_animal(animal):
    if isinstance(animal, Dog):
        return f"Walking the dog {animal.name}"
    elif isinstance(animal, Cat):
        return f"Feeding the cat {animal.name}"
    else:
        return f"Caring for {animal.name}"
```

Наследование и полиморфизм — это мощные инструменты Python, которые делают код более организованным, гибким и легким в сопровождении. Правильное использование этих концепций позволяет создавать масштабируемые и поддерживаемые приложения.