



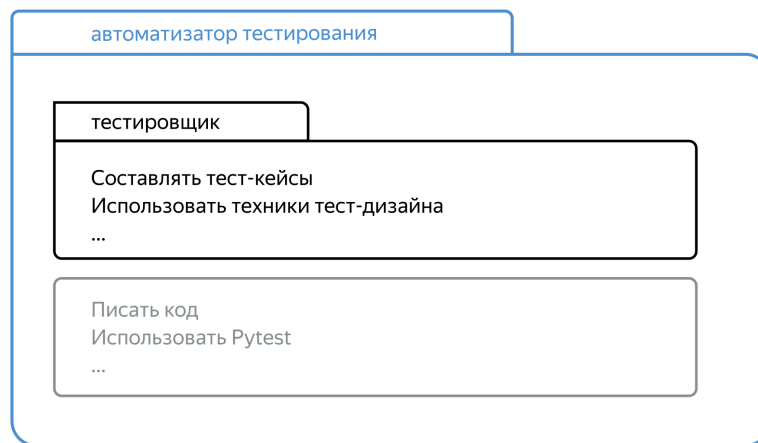
Наследование: шпаргалка



Наследование — это когда ты берёшь один класс и создаёшь на его основе другой. Например, на основе «Тестировщика» — «Автоматизатора тестирования». Новенький класс наследует всё, что есть у родителя. Плюс ему можно добавить дополнительные параметры и методы. Это помогает не дублировать код.

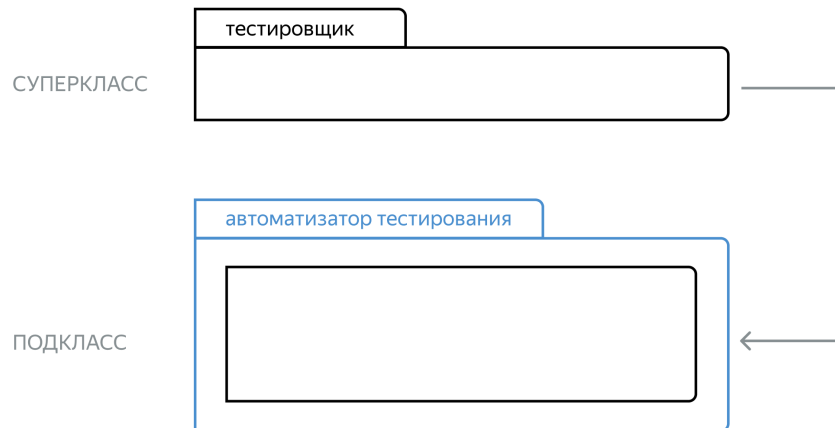
Пример. Есть классы «Тестировщик» и «Автоматизатор тестирования». Автоматизатор умеет всё, что умеет тестировщик. При этом у него есть особые знания и навыки: он может писать автотесты.

Получается, что автоматизатор — это тестировщик, но с расширенным функционалом. То есть в классе «Автоматизатор» будет тот же код, что и в классе «Тестировщик». Плюс несколько новых методов.



Чтобы не писать одинаковый код, в таких случаях используют **наследование**.

В Python класс-родитель называют **суперклассом**, а класс-наследник — **подклассом**.



Как создать подкласс

Когда создают подкласс, в круглых скобках указывают имя суперкласса. Вот так:

```
class <имя подкласса>(<имя суперкласса>):
    ... # тут будут методы подкласса
```

Так класс `Animal` выглядит с наследованием:

```
class Organism:
    def eat(self):
        return "Я ем..."

    def sleep(self):
        return "Я сплю..."

class Animal(Organism):
    # класс Animal – наследник класса Organism
    def move(self):
        # метод move() – уникальный метод подкласса
        return "Я двигаюсь..." # общие методы описывать не пришлось
```

Наследование с конструктором

В классе может быть метод `__init__()`. Он называется «конструктор», потому что помогает построить объект — собрать его из разных параметров.



В Python метод `__init__()` не наследуется автоматически. Если в суперклассе есть конструктор, его нужно явно вызвать в подклассе.

Пример. Класс `StringInstrument` описывает струнные инструменты. У него два свойства:

- `name` — определяет название инструмента,
- `number_of_strings` — отвечает за количество струн.

Параметры задаются через конструктор: именно он помогает присвоить индивидуальные значения каждому объекту.

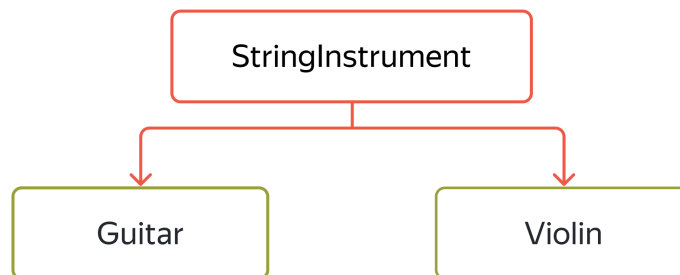
```
class StringInstrument:
    def __init__(self, name, number_of_strings):
        self.name = name
        self.number_of_strings = number_of_strings
```

Эти свойства используют методы `play()` и `tune()`. Они отвечают за игру на инструменте и за его настройку:

```
class StringInstrument:
    def __init__(self, name, number_of_strings):
        self.name = name
        self.number_of_strings = number_of_strings
    # Метод для игры
    def play(self):
        return f"Играем на инструменте {self.name}"
    # Метод для настройки
    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"
```

Теперь представь, что у класса `StringInstrument` нужно создать два подкласса:

- `Guitar` — гитара. Он наследует все свойства и методы `StringInstrument`, плюс есть уникальный метод `play_chords()` — «играть аккорды».
- `Violin` — наследует все свойства и методы `StringInstrument`.



И вот тут важный момент. У суперкласса есть конструктор `__init__()`:

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

Значит, его нужно будет вызвать в подклассе. Это можно сделать двумя способами.

Способ 1. Вызвать конструктор через имя суперкласса

```
class SuperClass:
    def __init__(self, arg1, arg2):

class SubClass(SuperClass):
    def __init__(self, arg1, arg2, arg3):
        SuperClass.__init__(self, arg1, arg2)
```

→ Суперкласс

→ Вызов конструктора суперкласса

→ Подкласс

→ Вызов конструктора подкласса

→ Вызов конструктора суперкласса

Что происходит в самом вызове:

1. Пишем имя суперкласса — `SuperClass`. Это помогает явно указать, какой конструктор нужно вызывать.
2. Пишем конструктор `__init__()`.
3. Передаём `self` в качестве первого аргумента. Это означает, что нужно инициализировать свойства текущего объекта подкласса.
4. Передаём остальные аргументы `arg1`, `arg2` — это параметры, которые требуются конструктором суперкласса. Если их не передать, Python выдаст ошибку.

Пример. Вот как это выглядит для инструментов:

```
# суперкласс
class StringInstrument:
    def __init__(self, name, number_of_strings):
        self.name = name
        self.number_of_strings = number_of_strings

    def play(self):
        return f"Играем на инструменте {self.name}"

    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"

# подкласс Guitar
class Guitar(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Гитара'
        StringInstrument.__init__(self, self.name, number_of_strings) #вызов конструктора

    def play_chords(self):
        return f"Играем аккорды на инструменте {self.name}"
```

```

# суперкласс
class StringInstrument:
    def __init__(self, name, number_of_strings):
        self.name = name
        self.number_of_strings = number_of_strings

    def play(self):
        return f"Играем на инструменте {self.name}"

    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"

# подкласс Guitar
class Guitar(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Гитара'
        StringInstrument.__init__(self, self.name, number_of_strings)

    def play_chords(self):
        return f"Играем аккорды на инструменте {self.name}"

```

В конструкторе подкласса вызываем конструктор суперкласса

Если убрать `StringInstrument.__init__(self, self.name, number_of_strings)`, код работать не будет.

Пример 2. Вот как будет выглядеть вызов конструктора суперкласса для скрипки:

```

class StringInstrument:
    def __init__(self, name, number_of_strings):
        self.name = name
        self.number_of_strings = number_of_strings

    def play(self):
        return f"Играем на инструменте {self.name}"

    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"

# подкласс Violin
class Violin(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Скрипка'
        StringInstrument.__init__(self, self.name, number_of_strings) # Вызов конструктора суперкласса

```

Способ 2. Вызвать конструктор через функцию `super()`



`super()` — это встроенная функция Python. Она возвращает объект суперкласса.

Когда используешь `super()`, передавать `self` в конструктор не нужно.

```
super().init(self.name, number_of_strings)
```

self первым аргументом
передавать не нужно

Пример. Вот как выглядит вызов конструктора через `super()` в коде целиком — для гитары и скрипки:

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

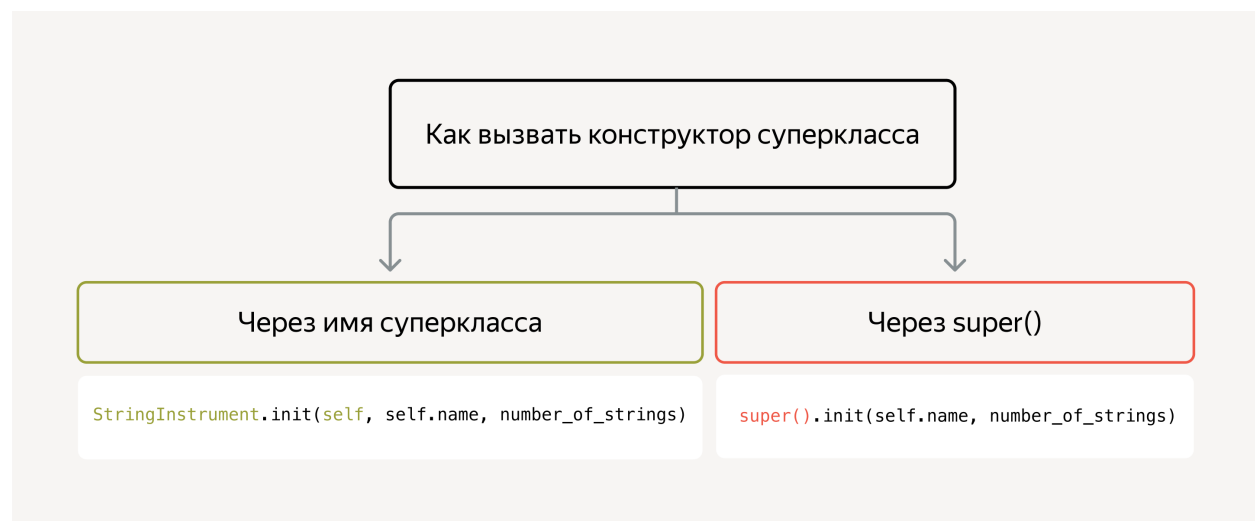
    def play(self):
        return f"Играем на инструменте {self.name}"

    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"

# подкласс Violin
class Violin(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Скрипка'
        super().__init__(self.name, number_of_strings) # вызов конструктора суперкласса

# подкласс Guitar
class Guitar(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Гитара'
        super().__init__(self.name, number_of_strings) # вызов конструктора суперкласса
```

Когда какой способ использовать

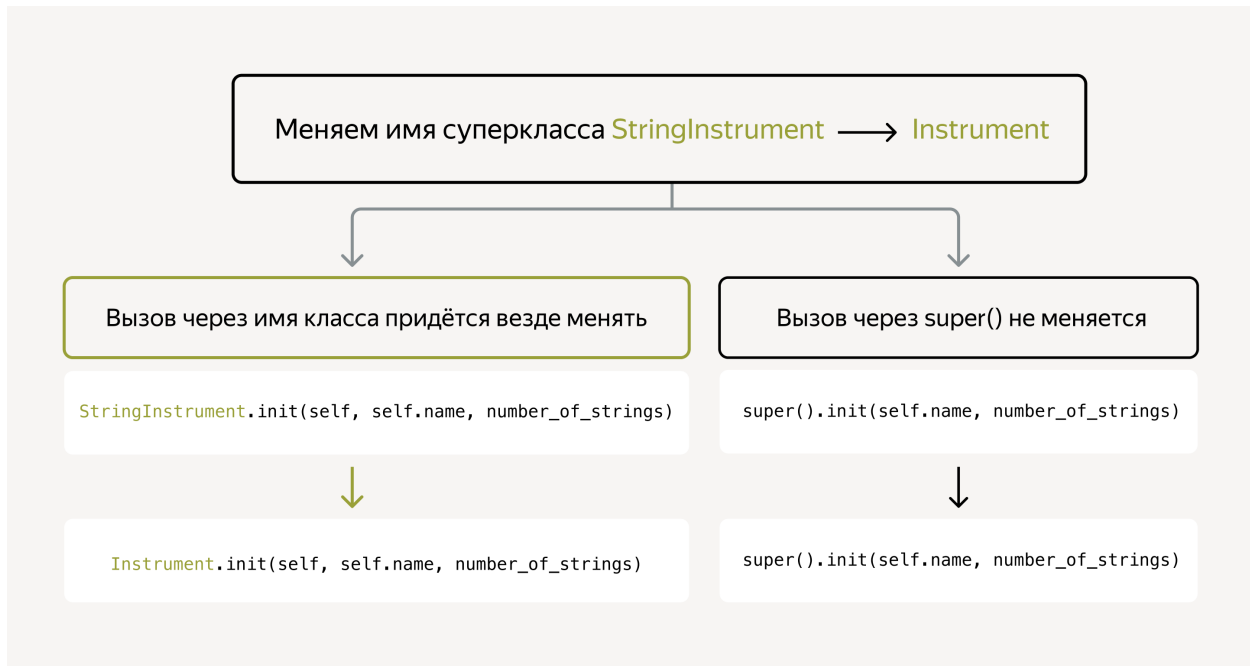




В одиночном наследовании лучше использовать `super()`, чем вызывать метод класса через имя. Это более универсально, и поддерживать код проще.

Допустим, имя суперкласса поменялось. Если использовать прямой вызов конструктора через имя —

`StringInstrument.__init__(self, self.name, number_of_strings)`, — придётся переписывать эту строку в каждом подклассе. А вот с `super()` ничего менять не нужно.



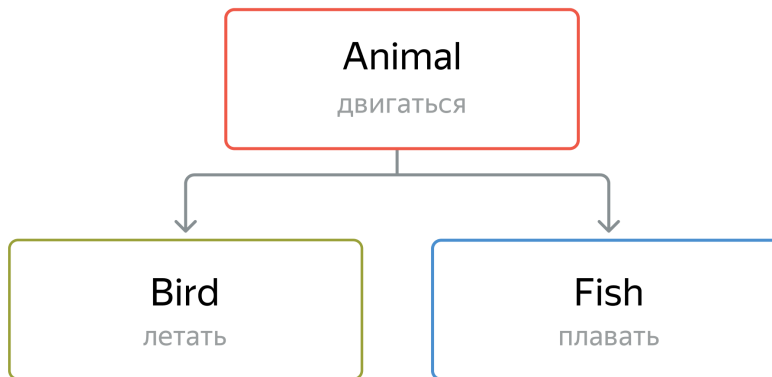
Во множественном наследовании нужно вызывать конструктор суперкласса через имя. Иначе легко запутаться, где к чему обращались.

Переопределение методов

Подкласс наследует все методы и свойства суперкласса, плюс у него было несколько своих уникальных. Но бывает так, что метод нужно не просто унаследовать, но ещё и адаптировать.

Пример. Есть суперкласс `Animal` с методом «двигаться». Все подклассы его унаследуют. Но у каждого животного свои особенности: птицы летают, рыбы плавают. Получается, нужно немного изменить первоначальный метод в каждом из наследников.

Для этого методы нужно **переопределить**.



Как переопределить метод

Чтобы переопределить метод в подклассе, нужно задать ему такое же имя, как в суперклассе. А вот дальше уже поменять: написать свою реализацию.

Пример. Вот как переопределить метод `move()` для птицы:

```
class Animal():
    def move(self):
        return "Я двигаюсь..."

class Bird(Animal):          # класс Bird – подкласс Animal
    def move(self):
        return "Я летаю..." # реализация метода поменялась
```

А вот переопределили метод сразу в двух подклассах:

```
class Animal():
    def move(self):
        return "Я двигаюсь..."

class Bird(Animal):          # класс Bird – подкласс Animal
    def move(self):
        return "Я летаю..." # реализация метода поменялась

class Fish(Animal):          # класс Fish – подкласс Animal
    def move(self):
        return "Я плаваю..." # реализация метода поменялась
```

Теперь этот метод будет выводить своё для каждого класса:

```
# создаём объекты
sparrow = Bird()
tuna = Fish()

# вызываем методы
print(sparrow.move()) # Я летаю...
print(tuna.move())    # Я плаваю...
```


Пример 2. У класса `StringInstrument` есть подкласс `Violin`:

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

    def play(self):
        return f"Играем на инструменте {self.name}"

    def tune(self):
        return f"Настраиваем инструмент {self.name}. Количество струн: {self.number_of_strings}"

# подкласс Violin
class Violin(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Скрипка'
        StringInstrument.__init__(self, self.name, number_of_strings) # вызов конструктора суперкласса
```

Допустим, нужно, чтобы звук игры на скрипке отличался от других струнных инструментов. Для этого можно переопределить метод `play()`:

```
class Violin(StringInstrument):
    def __init__(self, number_of_strings):
        self.name = 'Скрипка'
        super().__init__(self.name, number_of_strings)

    def play(self):
        return f"Играем на инструменте {self.name} с особенной техникой"

violin = Violin(4)
print(violin.play()) # Играем на инструменте Скрипка с особенной техникой
```

Теперь при вызове `play()` для объекта `Violin` будет возвращаться строка `"Играем на инструменте Скрипка с особенной техникой"`. При этом метод `play()` для объектов других подклассов `StringInstrument` остаётся без изменений, потому что его переопределили только в классе `Violin`.

Как добавить методу уникальные параметры

Нужно просто дописать их в скобках, когда объявляешь переопределённый метод.

Пример. Есть класс `ElectronicDevice`. В нём метод `consume_power()` показывает, сколько энергии потребляет устройство. У разных устройств это рассчитывается по-разному:

- для компьютера `Laptop` учитываются часы работы;
- для кондиционера `AirConditioner` учитываются часы охлаждения помещения.

Соответственно, нужно переопределить метод `consume_power()` в обоих подклассах. Но просто поменять тело метода недостаточно. Нужно добавить параметры:

- для компьютера метод будет принимать дополнительный параметр `working_hours`;
- для кондиционера метод будет принимать дополнительный параметр `cooling_hours`.

```
# метод суперкласса, который переопределяем
def consume_power(self, hours):
    return self.standby_power * hours

# метод для компьютера
def consume_power(self, hours, working_hours):
    return self.standby_power * (hours - working_hours) + self.working_power * working_hours

# метод для кондиционера
def consume_power(self, hours, cooling_hours):
    return self.standby_power * (hours - cooling_hours) + self.cooling_power * cooling_hours
```

Получается, у переопределённых методов другая сигнатура. Вот как это выглядит целиком:

```
class ElectronicDevice:
    def __init__(self, standby_power):
        self.standby_power = standby_power

    def consume_power(self, hours):
        return self.standby_power * hours

class Laptop(ElectronicDevice):
    def __init__(self, standby_power, working_power):
        super().__init__(standby_power) # вызов конструктора суперкласса
        self.working_power = working_power

    def consume_power(self, hours, working_hours): # переопределение метода
        return self.standby_power * (hours - working_hours) + self.working_power * working_hours

class AirConditioner(ElectronicDevice):
    def __init__(self, standby_power, cooling_power):
        super().__init__(standby_power) # вызов конструктора суперкласса
        self.cooling_power = cooling_power

    def consume_power(self, hours, cooling_hours): # переопределение метода
        return self.standby_power * (hours - cooling_hours) + self.cooling_power * cooling_hours

laptop = Laptop(standby_power=10, working_power=100)
air_conditioner = AirConditioner(standby_power=5, cooling_power=150)

# расход энергии для ноутбука за 24 часа с 8 рабочими часами
print(laptop.consume_power(hours=24, working_hours=8)) # 960

# расход энергии для кондиционера за 24 часа с 12 часами охлаждения
print(air_conditioner.consume_power(hours=24, cooling_hours=12)) # 1860
```

Как видишь, имя метода оставили таким же, а параметры и тело поменяли.

Как полностью заменить параметры

Можно не просто добавить дополнительные параметры, а заменить их полностью: просто написать в скобках новые.

Пример. Есть суперкласс `Instrument` с методом `play()`. Метод принимает два параметра: `melody` и `duration`.

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

    def play(self, melody, duration):
        return f'Играем "{melody}" на инструменте "{self.name}" длительностью {duration} минут.'
```

Нужно создать подкласс `PercussionInstrument` — для ударных. У него тоже будет метод `play()`, но он должен принимать только один параметр — `rhythm`. Ударные инструменты обычно играют ритм, а не мелодию.

Получается, метод `play()` переопределили в подклассе так: убрали параметры `melody` и `duration`, добавили новый — `rhythm`. Тело тоже изменили:

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

    def play(self, melody, duration):
        return f'Играем "{melody}" на инструменте "{self.name}" длительностью {duration} минут.'
```

```
class PercussionInstrument(Instrument):
    def __init__(self, name):
        super().__init__(name)
```

```
def play(self, rhythm):
    return f'Играем "{rhythm}" ритм на инструменте "{self.name}".'
```

```
guitar = Instrument('гитара')
print(guitar.play('Sweet dreams', 5)) # Играем Sweet dreams на инструменте гитара длительностью 5 минут.
```

```
drums = PercussionInstrument('барабаны')
print(drums.play('basic rock')) # Играем basic rock ритм на инструменте барабаны.
```



Нужно внимательно следить за тем, где какие аргументы ты передаёшь методам. Если сигнатура метода в подклассе отличается от сигнатуры в суперклассе, легко запутаться.

Например, метод суперкласса пытаются вызвать с параметрами, которые вообще-то нужны подклассу. Допустим, `consume_power()` для объекта `laptop` без параметра `working_hours`. Получим ошибку `TypeError`: она указывает на то, что в метод не передали нужные аргументы.

`Super()` для переопределения методов



Поможет, если в переопределённом методе нужно оставить часть функциональности из метода суперкласса.

Пример. Есть суперкласс `Animal`, который определяет базовые характеристики и поведение животного. У животного есть определённое количество энергии и метод `move()`. Он симулирует движение животного и расход энергии на это.

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

    def move(self, distance):
        self.energy -= distance * 0.1 # каждый метр движения требует 0.1 единицы энергии

        return self.energy
```

Создадим подкласс `Bird`. У птиц есть особенность в движении: они могут восстанавливать энергию, когда летают с течением ветра.

Получается, чтобы рассчитать энергию для птицы, надо взять метод для животного и добавить ему дополнительные настройки: способность восстанавливать энергию, когда птица движется по ветру.

Можно просто переписать метод заново и повторить строчку `self.energy -= distance * 0.1`. Но легче обратиться к ней с помощью `super()`. Вот так:

```
def move(self, distance):
    self.energy = super().move(distance) # начинаем движение так же, как и другие животные
    self.energy += self.wind * 0.05 * distance # но затем восстанавливаем часть энергии, если есть ветер

    return self.energy
```

В коде целиком:

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

    def move(self, distance):
        self.energy -= distance * 0.1 # каждый метр движения требует 0.1 единицы энергии

        return self.energy
```

```
class Bird(Animal):
    def __init__(self, energy, wind):
        super().__init__(energy)
        self.wind = wind # скорость ветра может влиять на энергию птицы во время движения

    def move(self, distance):
        self.energy = super().move(distance) # начинаем движение так же, как и другие животные
        self.energy += self.wind * 0.05 * distance # но затем восстанавливаем часть энергии, если есть ветер

    return self.energy
```

В начале переопределённого метода используем `super()`, чтобы вызвать реализацию `move()` из суперкласса. После этого добавили новую логику, уникальную для класса `Bird`, — восстановление энергии.

```
def move(self, distance):
    self.energy = super().move(distance)
    self.energy += self.wind * 0.05 * distance

    return self.energy
```

• обратились к методу суперкласса

Теперь движение птицы учитывает как общую логику движения животного, так и уникальную логику движения птицы.

Полиморфизм

Есть три принципа ООП:





Полиморфизм — принцип, который позволяет объектам с одним и тем же набором методов или свойств вести себя по-разному. Проще говоря, это когда одно и то же действие работает по-разному для разных объектов.

Слово «полиморфизм» происходит от греческих слов *poly* — «много» и *morphos* — «форма». То есть мы имеем дело с «многими формами». Это как умение одного и того же актёра играть разные роли. Кусочек кода — это актёр, а фильмы — разные части программы. В разных частях программы один и тот же кусочек кода играет разные роли, по-разному себя ведёт.

Переопределение методов и полиморфизм тесно связаны: именно переопределение позволяет изменить поведение метода для подкласса. Но это не единственный пример того, как работает полиморфизм в Python.

Пример полиморфизма: оператор

В Python оператор  — полиморфный. Это означает, что он может вести себя по-разному в зависимости от контекста. Например, при сложении двух чисел  выполняет арифметическую операцию:

```
print(2 + 3) # 5
```

Но если мы применим `+` к двум строкам, он будет работать как оператор конкатенации:

```
print("Привет, " + "Рим!") # Привет, Рим!
```

Полиморфизм функций

В Python есть некоторые функции, которые могут принимать аргументы разных типов. Например, функция `len()` возвращает длину. Она работает с различными типами данных.

Со строками:

```
print(len("Полиморфизм")) # 11
```

Со списками:

```
print(len([5, 10, 5, 4, 5, 10, 5])) # 7
```

Со словарями:

```
print(len({'Длина': 100, 'Ширина': 80, 'Высота': 18})) # 3
```

Функция полиморфная: её работа зависит от того, аргумент какого типа ей передать.

Чем полезен полиморфизм

Полиморфизм делает код проще и чище:

- **Код можно переиспользовать.** Одну и ту же часть кода можно применять в разных местах программы.
- **Код легче понять.** Один и тот же кусочек кода всегда будет выглядеть одинаково, даже если он делает разные вещи.
- **Легче вносить изменения в код.** Можно добавить новую функциональность кусочку кода, не затрагивая его старое использование.