



# Множественное наследование: шпаргалка



Множественное наследование — это когда класс делают наследником сразу нескольких суперклассов.

**Пример.** Есть класс «Амфибия»: `Amphibian`. Он должен наследовать методы и наземных животных `LandAnimal`, и водоплавающих `WaterAnimal`.

## Как написать множественное наследование



Чтобы сделать класс наследником нескольких классов, нужно перечислить названия суперклассов в скобках.

**Пример.** Вот есть два класса: `LandAnimal` с методом `walk()` и `WaterAnimal` с методом `swim()`:

```
class LandAnimal:
    def walk(self):
        print("Ходит")

class WaterAnimal:
    def swim(self):
        print("Плавает")
```

Чтобы сделать амфибий наследниками обоих классов, нужно перечислить названия суперклассов в скобках:

```
class LandAnimal:
    def walk(self):
        print("Ходит")

class WaterAnimal:
    def swim(self):
        print("Плавает")

class Amphibian(LandAnimal, WaterAnimal): # два суперкласса
    # тут будет переопределение методов
```

Допустим, конкретно в этом случае переопределять методы не нужно: никакой новой функциональности у амфибий тоже нет. Приходит на помощь оператор

`pass` .

```
class LandAnimal:
    def walk(self):
        print("Ходит")

class WaterAnimal:
    def swim(self):
        print("Плавает")

class Amphibian(LandAnimal, WaterAnimal):
    pass # так как ничего не нужно менять
```



В программировании `pass` используют как заполнитель в тех случаях, когда синтаксис требует наличия какого-то кода, но при этом вам не нужно выполнять никаких действий.

То есть подкласс всё равно наследует все свойства и методы суперкласса, а новых у него нет. Используем оператор `pass` .

Теперь объект `frog` умеет и ходить, и плавать:

```

class LandAnimal:
    def walk(self):
        print("Ходит")

class WaterAnimal:
    def swim(self):
        print("Плавает")

class Amphibian(LandAnimal, WaterAnimal):
    pass

frog = Amphibian()
frog.walk() # Ходит
frog.swim() # Плавает

```

## Переопределение в множественном наследовании

В Python подкласс наследует все методы и атрибуты суперклассов: нельзя выбрать какие-то конкретные, а остальные не наследовать. Иными словами, от обоих родителей подклассу перейдут все методы и свойства.

Хорошая новость — методы можно переопределить: заменить функциональность или вообще убрать. Всё как в единичном наследовании.

**Пример.** У `LandAnimal` есть метод `run()`. При этом нужно, чтобы животное-амфибия не умело бегать. Просто переопределим метод в подклассе: пусть он выводит «Этот вид не бежит».

```

class LandAnimal:
    def walk(self):
        print("Ходит")

    def run(self):
        print("Бежит")

class WaterAnimal:
    def swim(self):
        print("Плавает")

```

```
class Amphibian(LandAnimal, WaterAnimal):
    def run(self):
        print("Этот вид не бегаёт.")

frog = Amphibian()
frog.walk() # Ходит
frog.swim() # Плавает
frog.run()  # Этот вид не бегаёт.
```

## Множественное наследование с конструктором



Если у суперклассов есть конструкторы, нужно явно вызвать эти конструкторы в подклассе.

**Пример.** У `LandAnimal` и `WaterAnimal` есть конструкторы, которые устанавливают атрибуты `can_walk` и `can_swim`:

```
class LandAnimal:
    def __init__(self):
        self.can_walk = True # конструктор

    def walk(self):
        if self.can_walk:
            print("Ходит")
        else:
            print("Не может ходить")

class WaterAnimal:
    def __init__(self):
        self.can_swim = True # конструктор

    def swim(self):
        if self.can_swim:
            print("Плавает")
        else:
            print("Не может плавать")
```

Вот так вызвать конструкторы:

```

class LandAnimal:
    def __init__(self):
        self.can_walk = True

    def walk(self):
        if self.can_walk:
            print("Ходит")
        else:
            print("Не может ходить")

class WaterAnimal:
    def __init__(self):
        self.can_swim = True

    def swim(self):
        if self.can_swim:
            print("Плавает")
        else:
            print("Не может плавать")

class Amphibian(LandAnimal, WaterAnimal):
    def __init__(self):
        LandAnimal.__init__(self) # вызвали конструктор первого суперкласса
        WaterAnimal.__init__(self) # вызвали конструктор второго суперкласса

frog = Amphibian()
frog.walk() # Ходит
frog.swim() # Плавает

```



Если не вызывать конструктор одного из суперклассов, атрибуты суперкласса не будут инициализированы в наследнике. Всё как в единичном наследовании.

Вот что произойдёт, если не вызвать конструктор `WaterAnimal` в `Amphibian`:

```

class LandAnimal:
    def __init__(self):
        self.can_walk = True

    def walk(self):
        if self.can_walk:
            print("Ходит")

```

```

        else:
            print("Не может ходить")

class WaterAnimal:
    def __init__(self):
        self.can_swim = True

    def swim(self):
        if self.can_swim:
            print("Плавает")
        else:
            print("Не может плавать")

class Amphibian(LandAnimal, WaterAnimal):
    def __init__(self):
        LandAnimal.__init__(self)
        # WaterAnimal.__init__(self) # закомментировано

frog = Amphibian()
frog.walk() # Ходит
frog.swim() # AttributeError: 'Amphibian' object has no attribute 'can_swim'

```

Если вызвать метод `swim()` для экземпляра класса `Amphibian`, получится ошибка `AttributeError`. Это потому что атрибут `can_swim` не был инициализирован: не вызвали конструктор суперкласса `WaterAnimal`.

## Множественное наследование и `super()`



**При множественном наследовании лучше вызывать конструкторы через имя класса.** Так будет легче проконтролировать, конструктор какого класса сейчас вызовется.

Функция `super()` тоже работает, но есть особенности. Допустим, в амфибиях вызывают конструктор суперкласса через `super()`:

```

class Amphibian(LandAnimal, WaterAnimal):
    def __init__(self):
        print("Вызвали init из Amphibian")
        super().__init__() # вызов конструктора суперкласса

```

По идее `super()` должна вызвать конструктор суперкласса. Но у амфибий родителя целых два. Какой конструктор вызвать? Это нигде не указано. Раз нигде явно не указали, какой конструктор нужен, Python решит это сам. Для этого у него есть специальный алгоритм.



В Python есть специальный алгоритм, который определяет, в каком порядке вызвать методы. Он называется **MRO** — Method Resolution Order.

И этот алгоритм решит вот что: кто из родителей первый указан в скобках, конструктор того и вызовут.

В этом примере первым стоит класс `LandAnimal`:

```
class LandAnimal:
    def __init__(self):
        print("Вызвали init из LandAnimal")
        self.can_walk = True

    def walk(self):
        if self.can_walk:
            print("Ходит")
        else:
            print("Не может ходить")

class WaterAnimal:
    def __init__(self):
        print("Вызвали init из WaterAnimal")
        self.can_swim = True

    def swim(self):
        if self.can_swim:
            print("Плавает")
        else:
            print("Не может плавать")

class Amphibian(LandAnimal, WaterAnimal):
    def __init__(self):
        print("Вызвали init из Amphibian")
        super().__init__() # вызовет LandAnimal.__init__(), но не WaterAnimal.__init__()

frog = Amphibian()
```

```
# Вызвали init из Amphibian
# Вызвали init из LandAnimal

print(frog.can_walk) # True

# # Вызовет ошибку – AttributeError: 'Amphibian' object has no attribute 'can_swim',
# потому что WaterAnimal.__init__() не был вызван
print(frog.can_swim)
```

Даже если ты напишешь второй раз `super().__init__()`, он опять вызовет конструктор `LandAnimal`.

В тренажёре легко определить, какой именно конструктор будет вызван. Но на практике все классы распределены по разным файлам. Когда строишь сложные иерархии, следить за логикой наследования сложно. Поэтому лучше указать явно, какой именно конструктор нужно вызвать, — то есть обратиться через имя. А `super()` лучше использовать в одиночном наследовании.

## Миксины



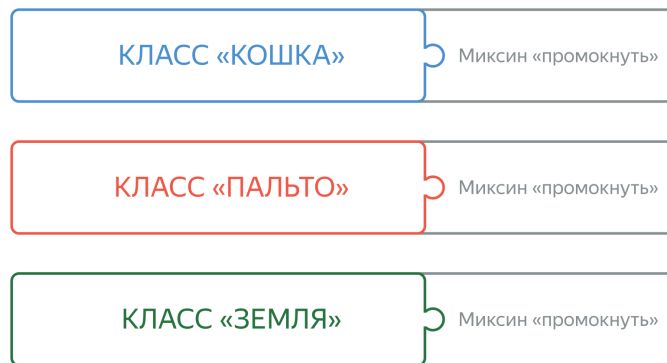
Миксины — это классы, которые создаются как готовые детали. Их встраивают в другие классы, чтобы расширить функциональность, но не создавать иерархии с наследованием.

Похоже на детали конструктора: присоединили готовую деталь, и конструкция уже умеет больше. При этом в иерархии классов ничего не меняли.

**Пример.** Есть классы «Кошка», «Пальто», «Земля». Для всех них должна быть доступна функция «промокнуть под дождём». Но при этом эти три класса никак не объединишь в иерархию: это будет нелогично.

Иначе говоря, есть некая функция, которую хочется использовать в нескольких классах, но эти классы не имеют других общих черт. Помогут **МИКСИНЫ**.





## Как создать миксин

Отдельной сущности «миксины» в Python нет. Технически это реализуется через классы и наследование, поэтому выглядит всё привычно. Просто договорились, что если класс определён как `Mixin`, то нужно использовать его соответствующим образом.

**Пример.** Нужно написать миксин, который поможет добавить в любой код функцию «засыпать» — `start_sleeping()`. Пишем его как отдельный класс, в названии которого будет `Mixin`:

```
class SleepMixin:
    def start_sleeping(self):
        print("Засыпаю...")
```

Теперь нужно добавить миксин классу «Человек». Указываем миксин в скобках — точно так же, как при наследовании:

```
class SleepMixin:
    def start_sleeping(self):
        print("Засыпаю...")

class Human(SleepMixin):
    def go_to_work(self):
        print("Иду на работу!")
```

Теперь классу `Human` доступен и его метод `go_to_work()`, и метод миксина `start_sleeping()`.

Расширим миксин до двух методов и добавим `EatMixin` с атрибутами `is_eating` и `food`. Теперь человек может просыпаться, засыпать, начинать и заканчивать есть и ходить на работу:

```
class SleepMixin:

    def start_sleeping(self):
        print("Засыпаю...")

    def stop_sleeping(self):
        print("Просыпаюсь!")

class EatMixin:
    def __init__(self):
        self.is_eating = False
        self.food = None

    def start_eating(self, food):
        self.is_eating = True
        self.food = food
        print(f"Начинаю есть {food}...")

    def stop_eating(self):
        self.is_eating = False
        print("Закончил есть!")


class Human(SleepMixin, EatMixin):
    def __init__(self):
        SleepMixin.__init__(self)
        EatMixin.__init__(self)

    def go_to_work(self):
        print("Иду на работу!")


bob = Human()
bob.start_sleeping() # Засыпаю...
bob.stop_sleeping() # Просыпаюсь!
bob.start_eating("яблоко") # Начинаю есть яблоко...
print(bob.is_eating, bob.food) # True, "яблоко"
bob.stop_eating() # Закончил есть!
print(bob.is_eating) # False
bob.go_to_work() # Иду на работу!
```

Класс `Human` «смешивает» несколько миксинов, тем самым получая возможность спать и есть. Кроме того, у него есть собственный метод `go_to_work()`.

## Чем миксины отличаются от суперклассов

 **Не участвуют в иерархии.** Хотя миксины и реализуются как классы, они не представляют собой суперкласс в обычном понимании.

Миксины не предназначены для того, чтобы использовать их в качестве полноценных классов: создавать подклассы или объекты. Это лишь некое расширение, которое ты устанавливаешь классу, чтобы были доступны определённые методы или свойства.

 **Небольшой объём и специализация.** Миксины, как правило, реализуют небольшой объём функциональности: на практике в них редко бывает больше двух методов.

## Миксины в тестировании

В автоматизированном тестировании миксины помогают объединить поведение и атрибуты, которые могут быть общими для нескольких различных тестовых случаев.

**Пример.** Есть много тестов, которые нуждаются в аутентификации. Вместо того чтобы копировать и вставлять код аутентификации в каждый тест, можно создать миксин, который содержит эту общую функциональность:

```
class AuthenticationMixin:
    def authenticate(self):
        # заполняем поля логина и пароля и аутентифицируемся на сайте
        ...

class TestDashboard(AuthenticationMixin):
    def test_dashboard_loads(self):
        self.authenticate()
        # здесь были бы проверки, что дашборд загрузился правильно

class TestProfile(AuthenticationMixin):
    def test_profile_loads(self):
        self.authenticate()
        # здесь были бы проверки, что профиль загрузился правильно
```

В этом примере два тест-кейса для `TestDashboard` и `TestProfile`. Оба требуют аутентификации перед выполнением теста: используем миксин `AuthenticationMixin`, чтобы не дублировать код.

Миксин содержит метод `authenticate()`, который выполняет процесс аутентификации.

Это лишь один из примеров, как использовать миксины в автоматизированном тестировании. В целом они пригодятся для любой функциональности, которую нужно повторно использовать в нескольких тестах.