Тема 10. Объектно-ориентированное программирование в Python

Объектно-ориентированный подход в Python предоставляет гибкие средства для организации кода и построения масштабируемых приложений. Предыдущая тема охватывала базовые принципы (наследование, инкапсуляцию, полиморфизм) и демонстрировала их базовую реализацию в языке. В данной теме рассматриваются дополнительные аспекты, которые позволяют оптимизировать архитектуру кода, обеспечить единообразное хранение состояния, выполнять динамическое управление атрибутами и настраивать работу с ресурсами с помощью контекстных менеджеров.

Здесь внимание уделяется статическим переменным и методам класса, создающим единый центр управления общими данными, а также статическим методам, выступающим в роли утилитарных функций, не зависящих от конкретных экземпляров. Кроме того, разбираются инструменты динамической модификации атрибутов и механизмы интроспекции, позволяющие программе изучать свою структуру и адаптироваться к меняющимся условиям. Особое место отводится специальным (магическим) методам, которые дают возможность переопределять операторную семантику, управлять процессом итерации и создавать безопасные контекстные менеджеры для управления ресурсами.

Методы классов и статические переменные и методы

Объектно-ориентированное программирование в Python предоставляет разработчикам инструменты для управления общим состоянием объектов и организации кода на уровне классов. В отличие от методов экземпляра, которые работают с индивидуальными данными каждого объекта, методы классов и статические методы позволяют решать задачи, связанные с общими характеристиками или независимыми операциями. Статические переменные, определённые на уровне класса, становятся единым хранилищем данных, доступным всем экземплярам, что делает их удобными для учёта общих параметров или конфигураций. Методы классов, использующие декоратор `@classmethod`, дают возможность оперировать самим классом, например, для создания альтернативных конструкторов или управления глобальными настройками. Статические методы, в свою очередь, оформленные с помощью `@staticmethod`, представляют собой утилитарные функции, логически связанные с классом, но не зависящие от его состояния. Эти механизмы позволяют централизовать управление данными и упростить архитектуру программы, обеспечивая её масштабируемость и читаемость.

Статические переменные (переменные класса)

Определение и назначение

Статическая переменная — это конструкция, позволяющая хранить данные, общие для всех экземпляров одного класса, что делает её центральным элементом при организации единого состояния в программе. При разработке программного обеспечения зачастую возникает необходимость в наличии переменных, значение которых должно оставаться единым для всех объектов, независимо от того, сколько из них создано в ходе выполнения программы. Такая переменная определяется на уровне класса, а не конкретного объекта, что означает, что её объявление происходит внутри определения класса, но вне любых методов, и в результате она существует в единственном экземпляре для всего класса. Это позволяет обеспечить консистентность данных, избежать дублирования информации и централизовать управление состоянием, которое относится ко всей группе объектов.

Представим ситуацию, когда необходимо вести учёт количества созданных объектов для последующего анализа или принятия решений в рамках бизнес-логики. Каждый раз, когда создаётся новый объект, значение статической переменной может увеличиваться на единицу, что позволяет в реальном времени отслеживать динамику создания экземпляров класса. Такой подход на практике находит применение в системах регистрации пользователей или мониторинга активности на веб-сайте. Например, если разработчик создаёт класс `Visitor` для учёта посетителей сайта, статическая переменная `count` может использоваться как глобальный счётчик — каждый раз при вызове конструктора нового объекта увеличивается значение `count`, а обращение к нему через имя класса (`Visitor.count`) всегда возвращает актуальное число посетителей. Это наглядно демонстрирует, как единое хранилище данных позволяет быстро и без лишних вычислений получать агрегированную информацию, что особенно важно в системах, где данные используются для анализа производительности или нагрузки.

Другой пример использования статической переменной можно наблюдать в разработке систем, требующих единой конфигурации для всех объектов. Допустим, существует класс `DatabaseConnection`, отвечающий за установление соединения с базой данных. Вместо того чтобы у каждого объекта хранить свои копии настроек подключения — адрес сервера, имя пользователя, параметры безопасности — можно использовать статическую переменную для хранения этих общих параметров. При таком подходе изменения в конфигурации вносятся лишь в одном месте, что значительно упрощает сопровождение кода и предотвращает появление ошибок, связанных с рассинхронизацией настроек между различными объектами. Это актуально, например, в больших корпоративных системах, где изменение одного параметра требует немедленного отражения во всех точках программы.

Для более конкретного понимания рассмотрим следующий пример. Пусть имеется класс `Order`, представляющий заказ в системе интернет-магазина, где важным требованием является учёт общего количества заказов для формирования статистики продаж. При каждом создании нового заказа в конструкторе класса происходит увеличение значения статической переменной `order_count`, которая изначально инициализируется нулём. Такой приём позволяет любому компоненту системы в любой

момент обратиться к переменной через имя класса — `Order.order_count` — и получить точное число заказов, что может быть использовано для анализа загруженности системы, расчёта финансовых показателей или для запуска процессов автоматизации.

In [ ]:
```python
class Order:
    order_count = 0

    def __init__(self, order_id, items):
        self.order_id = order_id
        self.items = items
        Order.order_count += 1
```

В этом примере видно, как статическая переменная `order_count` выступает в роли глобального счётчика, обновляющегося автоматически при каждом создании объекта. Такой механизм не требует дополнительных усилий по синхронизации данных между объектами, поскольку все они обращаются к одному и тому же месту в памяти, где хранится значение переменной. Это существенно упрощает архитектуру программы и позволяет избегать лишнего кода, связанного с передачей информации от одного объекта к другому.

Таким образом, статическая переменная является неотъемлемой частью объектно-ориентированного программирования, поскольку она предоставляет удобный способ для хранения и управления данными, общими для всех экземпляров класса. Её использование способствует повышению эффективности кода, централизует управление состоянием и позволяет разработчику сосредоточиться на реализации основной логики программы, не отвлекаясь на избыточное дублирование данных. Именно благодаря таким свойствам статические переменные находят широкое применение в реальных проектах — от простых систем учёта до сложных корпоративных приложений, где важна согласованность и единообразие данных во всех компонентах системы.

Объявление статических переменных

Объявление статических переменных в Python представляет собой интуитивно понятный и одновременно мощный механизм, позволяющий задать параметры, общие для всех объектов данного класса. Такие переменные объявляются непосредственно внутри тела класса, но вне каких-либо методов, что означает, что они создаются один раз в момент определения класса и остаются неизменными для всех его экземпляров до явного изменения. Это даёт возможность централизованно хранить данные, которые не зависят от конкретного состояния каждого объекта, — например, настройки приложения, счётчики или константные параметры, необходимые для функционирования системы.

При объявлении статической переменной разработчик использует стандартный синтаксис присваивания, как если бы создавалась обычная переменная, но размещает это присваивание в пространстве имён класса. Такой подход подчёркивает логическую связь переменной с самим классом, а не с отдельными его экземплярами. Таким образом, если рассмотреть класс `AppConfig`, отвечающий за базовые настройки программы, то объявление переменных вроде версии приложения или максимально допустимого количества пользователей выглядит максимально естественно и прозрачно:

In [ ]:
```python
class AppConfig:
    version = "1.0.0"
    max_users = 100
```

В данном примере переменные `version` и `max_users` объявлены внутри определения класса, что означает их общедоступность для всех объектов класса `AppConfig`. При этом, если в какой-либо части программы потребуется получить доступ к версии приложения, достаточно обратиться к переменной через имя класса — `AppConfig.version`. Такое решение позволяет избежать дублирования кода и обеспечивает единообразие при использовании общих параметров, поскольку все экземпляры разделяют одно и то же значение, заданное в момент объявления.

Ещё одним важным аспектом объявления статических переменных является то, что они создаются при интерпретации определения класса, а не при создании каждого нового объекта. Это означает, что при импорте модуля, содержащего данный класс, статическая переменная инициализируется автоматически, и её значение становится доступным для всех последующих операций. Такая особенность особенно ценна в тех случаях, когда необходимо гарантировать, что параметр не изменяется случайно при каждом создании экземпляра, а остаётся единым для всех объектов. Например, в системах учёта, где требуется хранить общее количество обработанных заказов или зарегистрированных пользователей, объявление счётчика в виде статической переменной позволяет добиться надёжности и целостности данных.

Важно отметить, что обращение к статической переменной возможно как через имя класса, так и через экземпляр, однако попытка изменить значение переменной через объект может привести к созданию нового атрибута в пространстве экземпляра, который будет скрывать оригинальную переменную класса. Таким образом, чтобы гарантировать изменение именно общей переменной, следует обращаться к ней через имя класса. Это помогает избежать путаницы и обеспечивает предсказуемое поведение программы, поскольку разработчик точно знает, что изменение происходит в глобальном для класса контексте.

Объявление статических переменных в Python — это не просто синтаксическая возможность, а важный инструмент для организации общего состояния в программных системах. Он позволяет централизовать управление данными, избежать избыточности и обеспечить консистентность значений во всех экземплярах класса, что особенно важно при разработке сложных приложений, где требуется единое представление о критических параметрах работы программы.

Доступ к статическим переменным

Доступ к статическим переменным в Python — это тонкий механизм, который следует понимать для грамотного управления общим состоянием класса. При обращении к переменной, определённой на уровне класса, интерпретатор сначала ищет её в пространстве имён конкретного объекта, а затем в пространстве имён самого класса. Если объект не содержит локальной копии переменной, возвращается значение, заданное в классе, что позволяет работать с общими данными, независимо от того, сколько экземпляров создано. Обращение к переменной через имя класса, например, `MyClass.variable`, подчёркивает, что мы имеем дело именно с глобальной для класса информацией. Такой подход не только упрощает чтение кода, но и устраняет неоднозначность, когда возникает ситуация, если объект имеет одноимённый атрибут, скрывающий статическую переменную.

При выполнении операции присваивания через экземпляр — например, запись `instance.variable = новое_значение` — интерпретатор не изменяет значение переменной в пространстве имён класса, а создаёт в объекте локальный атрибут с именем `variable`. Этот локальный атрибут затем начинает перекрывать статическую переменную, что может привести к неожиданным результатам. Если же требуется изменить именно общее значение, следует обращаться к переменной через имя класса, записывая, скажем, `MyClass.variable = другое_значение`. Такой механизм даёт разработчику возможность избирательно управлять, изменяется ли общее состояние для всех объектов или создаётся индивидуальное значение для конкретного экземпляра.

Особое внимание стоит уделить порядку поиска атрибутов в Python. При обращении к атрибуту через объект интерпретатор сперва обращается к его внутреннему словарю атрибутов — `instance.__dict__`. Если нужного ключа там нет, поиск продолжается в классе, затем в базовых классах, если таковые имеются. Такой процесс называется разрешением имён, и он обеспечивает возможность наследования и полиморфизма. Если разработчик по ошибке присваивает значение через объект, то изменение будет видно только в этом объекте, а не во всей группе, использующей статическую переменную. Для тех, кто работает с системами, где важна синхронизация общего состояния, например, при ведении учёта количества обращений к определённому ресурсу, важно следить за тем, чтобы изменения производились именно через имя класса.

Рассмотрим более практический пример. Пусть имеется класс `Logger`, отвечающий за ведение журнала событий в приложении. В классе определена статическая переменная `log_level`, которая указывает уровень детализации логирования. Если обратиться к переменной через `Logger.log_level`, можно быть уверенным, что в системе используется единое значение, влияющее на работу всех экземпляров логгера. Однако если один из объектов выполнит операцию `logger_instance.log_level = 'DEBUG'`, то для него будет создана собственная копия этого атрибута, а другие объекты по-прежнему будут ссылаться на значение из класса. Это может вызвать путаницу при отладке или анализе логов, поскольку в одном и том же приложении окажутся два различных уровня логирования. Поэтому специалисты рекомендуют для изменения глобальных параметров использовать именно синтаксис `Logger.log_level = новое_значение`, что обеспечивает неизменность структуры и поведение всей системы.

При работе с большими проектами, где классы могут иметь сложную иерархию и несколько уровней наследования, понятие доступа к статическим переменным приобретает ещё большее значение. Например, в системе управления пользователями класса `User` может быть статическая переменная `default_role`, определяющая роль, назначаемую при регистрации. Если её изменять через конкретного пользователя, код может вести себя непредсказуемо, особенно если другие модули программы опираются на единое значение. Обращаясь же к переменной через `User.default_role`, разработчик явно демонстрирует намерение работать с глобальным параметром, что повышает читаемость и поддерживаемость кода.

Таким образом, понимание механизмов доступа к статическим переменным позволяет создавать более устойчивые и предсказуемые архитектуры программ. Правильное использование имени класса для обращения к переменной гарантирует, что общее состояние будет контролироваться централизованно, а осознанное создание локальных копий через экземпляры даёт гибкость в случае, когда требуется изменить поведение отдельного объекта без влияния на весь класс. Эта двойственность делает Python гибким инструментом для решения разнообразных задач в области объектно-ориентированного программирования.

Примеры использования

Статические переменные находят применение в самых разнообразных сценариях, и приводимые ниже примеры иллюстрируют их практическое использование в реальных задачах разработки. Рассмотрим ситуацию, когда необходимо вести учёт общего количества созданных объектов. Пусть имеется класс, отвечающий за представление посетителя на веб-сайте, и требуется фиксировать число обращений. Для этого в классе объявляется статическая переменная, которая инициализируется нулём, а при каждом вызове конструктора происходит её инкрементирование. Это позволяет в любой момент обратиться к переменной через имя класса и узнать общее количество посетителей.

In [ ]:
```python
class Visitor:
    count = 0

    def __init__(self, name):
        self.name = name
        Visitor.count += 1
```

In [ ]:
```python
v1 = Visitor("Алексей")
v2 = Visitor("Анна")
```

In [ ]:
```
print(Visitor.count)
```

```
2
```

В этом примере видно, что при создании объектов `v1` и `v2` переменная `count` увеличивается на единицу, что позволяет точно отследить общее число посетителей независимо от количества созданных экземпляров.

Другой распространённый случай — управление общими настройками приложения. Если требуется, чтобы все компоненты программы использовали единый набор параметров, можно определить их как статические переменные. Представим класс, отвечающий за конфигурацию, в котором параметр, определяющий режим отладки, объявлен на уровне класса. Обращение к нему происходит через имя класса, что гарантирует, что изменение значения параметра затронет всё приложение.

In [ ]:
```
class AppConfig:
    debug_mode = False
```

In [ ]:
```
print(AppConfig.debug_mode)
```

```
False
```

In [ ]:
```
AppConfig.debug_mode = True
print(AppConfig.debug_mode)
```

```
True
```

Здесь изменение значения переменной `debug_mode` через имя класса позволяет легко переключить режим работы приложения без необходимости модификации каждого отдельного объекта.

Ещё один практический пример связан с автоматической генерацией уникальных идентификаторов для объектов. Если необходимо, чтобы каждый создаваемый экземпляр имел уникальный номер, можно использовать статическую переменную в качестве счётчика. Например, класс, моделирующий сотрудника компании, может содержать переменную, хранящую следующий доступный идентификатор. При инициализации нового объекта этому сотруднику присваивается текущее значение переменной, после чего счётчик увеличивается.

In [ ]:
```
class Employee:
    next_id = 1

    def __init__(self, name):
        self.name = name
        self.id = Employee.next_id
        Employee.next_id += 1
```

In [ ]:
```
emp1 = Employee("Карина")
emp2 = Employee("Дмитрий")
```

In [ ]:
```
print(emp1.id, emp2.id)
```

```
1 2
```

Этот приём гарантирует, что каждому сотруднику будет присвоен уникальный идентификатор.

Приведённые примеры демонстрируют, как статические переменные помогают централизованно управлять данными, общими для всех объектов класса. Такой подход упрощает организацию кода, снижает риск дублирования информации и обеспечивает согласованное поведение системы при изменении глобальных параметров.

Методы классов

Декоратор `@classmethod`

Декоратор `@classmethod` позволяет объявить метод, который оперирует самим классом, а не отдельными его экземплярами. При этом первым параметром такого метода всегда выступает ссылка на класс — её принято именовать `cls`. Это даёт возможность получать доступ к атрибутам, определённым в теле класса, а также управлять статическими переменными. Например, когда данные для создания объекта поступают в виде строки, можно оформить альтернативный конструктор, который преобразует входные данные в параметры для основного конструктора.

Рассмотрим класс `Product`, предназначенный для представления товара. В нём реализован метод `from_string`, принимающий строку с данными о товаре, разделёнными запятыми, и возвращающий новый объект на основе этих данных:

In [ ]:
```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    @classmethod
    def from_string(cls, product_str):
        parts = product_str.split(',')
```

```
        name = parts[0].strip()
        price = float(parts[1].strip())
        return cls(name, price)
```

In [ ]:
```
product_data = "Ноутбук, 155050.50"
```

In [ ]:
```
new_product = Product.from_string(product_data)
```

In [ ]:
```
print(new_product.name)
print(new_product.price)
```

```
Ноутбук
155050.5
```

Метод `from_string` получает строку, разбивает её на части, приводит цену к числовому типу и создаёт новый объект, используя ссылку на класс `cls`.

Ещё один практический случай использования `@classmethod` связан с управлением глобальным состоянием класса. Пусть имеется класс, отвечающий за конфигурацию приложения, в котором необходимо централизованно обновлять общие настройки. Например, класс `AppConfig` хранит параметры, влияющие на поведение всего приложения — режим отладки и версию программы. При необходимости изменения этих настроек можно определить метод, который принимает новые параметры и обновляет статические переменные класса. Пример ниже демонстрирует, как можно организовать подобное обновление:

In [ ]:
```
class AppConfig:
    debug_mode = False
    version = "1.0.0"

    @classmethod
    def update_config(cls, debug=None, version=None):
        if debug is not None:
            cls.debug_mode = debug
        if version is not None:
            cls.version = version
```

In [ ]:
```
print(AppConfig.debug_mode)
print(AppConfig.version)
```

```
False
1.0.0
```

In [ ]:
```
AppConfig.update_config(debug=True, version="2.0.0")
```

In [ ]:
```
print(AppConfig.debug_mode)
print(AppConfig.version)
```

```
True
2.0.0
```

В этом примере класс `AppConfig` содержит статические переменные `debug_mode` и `version`, задающие основные параметры работы приложения. Метод `update_config`, оформленный декоратором `@classmethod`, принимает аргументы для обновления настроек. При вызове `AppConfig.update_config(debug=True, version="2.0.0")` статические переменные обновляются непосредственно в классе, и все компоненты, использующие эти параметры, автоматически получают актуальное значение.

Использование `@classmethod` в этих примерах демонстрирует, как метод может выступать в роли альтернативного конструктора или инструмента для централизованного управления настройками, обеспечивая единообразное поведение всей системы.

Применение методов класса

Методы класса находят применение там, где необходимо работать с самим классом, а не с отдельными его экземплярами, что открывает дополнительные возможности для организации кода. Одной из распространённых ситуаций является создание альтернативных конструкторов. Когда информация о создаваемом объекте поступает в виде строки или в другой нестандартной форме, класс может предоставить специальный метод, который принимает данные в нужном формате, выполняет их обработку и возвращает новый объект. Это помогает отделить логику преобразования входных данных от основной процедуры создания экземпляра, делая код более структурированным и удобным для расширения. Примером служит класс, описывающий прямоугольник, в котором кроме обычного конструктора, принимающего ширину и высоту, реализован метод для создания квадрата:

In [ ]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @classmethod
```

```
    def square(cls, side):
        return cls(side, side)
```

In [ ]:
```
rect = Rectangle(3, 4)
```

In [ ]:
```
sq = Rectangle.square(5)
```

In [ ]:
```
print(f'Прямоугольник: {rect.width}x{rect.height}')
```

```
Прямоугольник: 3x4
```

In [ ]:
```
print(f'Квадрат: {sq.width}x{sq.height}')
```

```
Квадрат: 5x5
```

Другим направлением применения методов класса является управление общим состоянием. Если в классе присутствуют статические переменные, отражающие глобальные настройки или счётчики, то методы, помеченные декоратором `@classmethod`, могут использоваться для централизованного обновления этих переменных. Например, рассмотрим класс, отвечающий за ведение журнала событий, где переменная `log_level` определяет уровень детализации. Метод класса может принимать новое значение и менять его для всего класса, что сразу становится доступным для всех объектов, использующих этот параметр:

In [ ]:
```
class Logger:
    log_level = "INFO"

    @classmethod
    def set_log_level(cls, level):
        cls.log_level = level
```

In [ ]:
```
print(Logger.log_level)
```

```
INFO
```

In [ ]:
```
Logger.set_log_level("DEBUG")
```

In [ ]:
```
print(Logger.log_level)
```

```
DEBUG
```

Методы класса полезны и в случаях, когда необходимо производить преднастройку или обновление конфигурационных параметров без создания экземпляра. Такое решение оправдано в приложениях, где изменение глобальных настроек должно сразу отражаться во всех компонентах системы. Возможность обращаться к атрибутам класса через параметр `cls` позволяет обеспечить гибкость и централизованное управление, избавляя от необходимости дублировать логику обновления состояния в каждом объекте.

Применение методов класса не ограничивается только альтернативными конструкторами или обновлением глобальных переменных — они могут служить универсальным инструментом для реализации фабричных методов, генераторов объектов по заданным критериям или даже для кэширования часто используемых данных, что позволяет улучшить производительность приложения. Возможности, предоставляемые методами класса, делают их важным элементом при проектировании архитектуры программ, где требуется тонкое управление логикой создания и состояния объектов.

Пример метода класса

Рассмотрим пример метода класса, который иллюстрирует, как с его помощью можно расширить функциональность конструктора и упростить создание объектов из данных, представленных в иной системе измерения. Пусть имеется класс, предназначенный для представления температуры в градусах Цельсия. Стандартный конструктор принимает значение температуры в Цельсиях, однако часто данные могут поступать в градусах Фаренгейта, что требует предварительного преобразования.

Для реализации этой задачи в классе объявлен метод, оформленный декоратором `@classmethod`, который принимает значение температуры в Фаренгейтах, выполняет пересчёт по формуле — результатом которого является температура в Цельсиях — и возвращает новый объект класса. При этом метод обращается к классу через параметр `cls`, что позволяет использовать его для создания объектов без предварительного вызова конструктора непосредственно через имя класса.

In [ ]:
```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @classmethod
    def from_fahrenheit(cls, fahrenheit):
        celsius = (fahrenheit - 32) * 5 / 9
        return cls(celsius)

    def __str__(self):
        return f"{self.celsius:.2f}°C"
```

In [ ]:
```
temp = Temperature.from_fahrenheit(100)
```

In [ ]:
```
print(temp)
```

```
37.78°C
```

В этом примере метод `from_fahrenheit` принимает значение 100 градусов Фаренгейта, преобразует его в 37.78 градусов Цельсия и возвращает новый объект класса `Temperature`. Использование `@classmethod` гарантирует, что логика преобразования является неотъемлемой частью класса, а обращение к ней осуществляется через имя класса, что делает вызов метода интуитивно понятным и удобным для поддержки.

Статические методы

Декоратор `@staticmethod`

Декоратор `@staticmethod` позволяет объявить метод, не зависящий от состояния экземпляра или класса, и функционирующий как независимая утилитарная функция, логически связанная с классом. При использовании этого декоратора метод не принимает ни параметра `self`, ни параметра `cls` — он получает только те аргументы, которые явно передаются при вызове. Такая организация кода удобна, когда требуется инкапсулировать функциональность, не влияющую на внутреннее состояние объекта, но при этом относящуюся к предметной области, представленной классом.

Представьте себе класс, отвечающий за математические операции, где логично объединить в одном месте функции сложения, вычитания и т.д. Метод, оформленный декоратором `@staticmethod`, может быть вызван как через имя класса, так и через его экземпляр, однако вне зависимости от способа вызова он работает одинаково, не имея доступа ни к атрибутам экземпляра, ни к атрибутам самого класса. Такой метод, по сути, обычная функция, помещённая внутрь класса для удобства организации кода и отражения логической связи с предметной областью.

In [ ]:
```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b
```

In [ ]:
```
result = MathUtils.add(10, 5)
```

In [ ]:
```
print(result)
```

```
15
```

В данном примере метод `add` не зависит от состояния какого-либо экземпляра класса, и его вызов через имя класса делает намерения разработчика максимально очевидными.

Можно отметить, что статические методы способствуют поддержанию чистой архитектуры приложения, то есть они позволяют разделить утилитарную логику от логики, связанной с конкретным состоянием объектов.

Ещё один пример использования статического метода может касаться валидации входных данных. Представьте класс, отвечающий за обработку строк, где необходимо проверять, удовлетворяет ли переданная строка определённым условиям. Метод, помеченный `@staticmethod`, может принять строку, выполнить проверку и вернуть результат в виде булевого значения. Такой метод не изменяет состояние объекта и не зависит от него, что делает его универсальным инструментом для валидации.

In [ ]:
```
class StringProcessor:
    @staticmethod
    def is_uppercase(s):
        return s.isupper()
```

In [ ]:
```
StringProcessor.is_uppercase("ПРИВЕТ")
```

Out[ ]:
```
True
```

In [ ]:
```
StringProcessor.is_uppercase("Привет")
```

Out[ ]:
```
False
```

В этом примере метод `is_uppercase` предоставляет возможность проверить, состоит ли строка полностью из заглавных букв, не обращаясь к атрибутам экземпляра или класса. Он служит простой и понятной функцией, которая, будучи размещённой внутри класса, подчёркивает логическую связь с операциями обработки строк, что упрощает структуру кода и делает его более читаемым.

Использование декоратора `@staticmethod` позволяет создавать компактные, независимые от состояния функции, что удобно в больших проектах, где требуется разделять утилитарную логику от функциональности, зависящей от конкретных данных объектов. Этот приём улучшает организацию кода, облегчает его тестирование и поддержку, а также способствует созданию модульных решений, отвечающих принципам чистой архитектуры.

Назначение статических методов

Итак, статические методы создаются для реализации функциональности, не зависящей от состояния конкретных объектов или самого класса. Они позволяют сгруппировать утилитарные функции, логически связанные с предметной областью, но не требующие доступа к атрибутам экземпляров. Функция, оформленная декоратором `@staticmethod`, по своей сути является обычной функцией, вынесенной внутрь класса для удобства организации кода и повышения его читаемости. Например, если требуется проверить корректность идентификатора книги (ISBN), можно оформить алгоритм проверки как статический метод, поскольку он работает только с входными данными и не зависит от свойств конкретного экземпляра.

Рассмотрим пример класса, отвечающего за работу с книгами, где реализован статический метод для проверки ISBN-10. Метод принимает строку с идентификатором, очищает её от лишних символов и вычисляет контрольную сумму по известному алгоритму. Если сумма удовлетворяет условию деления на 11, метод возвращает значение `True`, иначе — `False`:

In [ ]:
```
class Book:
    @staticmethod
    def is_valid_isbn(isbn):
        # Удаляем тире и пробелы для получения непрерывной строки цифр
        isbn = isbn.replace('-', '').replace(' ', '')
        if len(isbn) != 10:
            return False
        total = 0
        # Обрабатываем первые 9 символов: они должны быть цифрами
        for i in range(9):
            if not isbn[i].isdigit():
                return False
            total += int(isbn[i]) * (10 - i)
        # Последний символ может быть цифрой или буквой 'X', обозначающей 10
        last = isbn[9]
        if last == 'X':
            total += 10
        elif last.isdigit():
            total += int(last)
        else:
            return False
        return total % 11 == 0
```

In [ ]:
```
isbn_example = "0-306-40615-2"
```

In [ ]:
```
print(f"ISBN {isbn_example} корректен:", Book.is_valid_isbn(isbn_example))
```

```
ISBN 0-306-40615-2 корректен: True
```

В этом примере метод `is_valid_isbn` не зависит от состояния объектов класса `Book` и может быть вызван напрямую через имя класса. Он последовательно обрабатывает входную строку, вычисляя сумму произведений цифр на их веса, и проверяет корректность по модулю 11.

Ещё один пример демонстрирует применение статического метода в классе, отвечающем за преобразование времени. Допустим, требуется преобразовать количество секунд в часы, минуты и секунды. Поскольку для вычислений не нужен доступ к атрибутам класса или экземпляров, целесообразно оформить данную операцию как статический метод:

In [ ]:
```
class TimeConverter:
    @staticmethod
    def seconds_to_hms(seconds):
        hours = seconds // 3600
        minutes = (seconds % 3600) // 60
        secs = seconds % 60
        return hours, minutes, secs
```

In [ ]:
```
time_sec = 3672
```

In [ ]:
```
h, m, s = TimeConverter.seconds_to_hms(time_sec)
```

In [ ]:
```
print(f"{time_sec} секунд — это {h} ч, {m} мин, {s} сек")
```

```
3672 секунд — это 1 ч, 1 мин, 12 сек
```

Здесь метод `seconds_to_hms` принимает значение времени в секундах и с помощью целочисленного деления и операции взятия остатка вычисляет число часов, минут и оставшихся секунд. Его вызов через имя класса подчёркивает независимость метода от каких-либо конкретных данных, хранящихся в объектах, что делает его универсальным инструментом для преобразований в различных частях приложения.

Применение статических методов позволяет централизовать логически связанные операции, не требующие сохранения состояния, и существенно упрощает поддержку кода. Такие методы, будучи вызванными напрямую через имя класса, обеспечивают ясность и предсказуемость поведения, что особенно ценно при разработке масштабируемых и модульных систем.

Отличия от методов экземпляра и класса

Методы экземпляра, методы класса и статические методы существенно различаются по своему поведению, назначению и способу получения аргументов при вызове. Метод экземпляра, определённый без специальных декораторов, всегда получает первым параметром ссылку на конкретный объект — обычно именуемую `self`. Благодаря этому он имеет доступ к атрибутам, хранящимся в объекте, и может изменять его состояние. В отличие от него, метод класса, помеченный декоратором `@classmethod`, получает первым параметром ссылку на сам класс — обозначаемую как `cls` — и работает с данными, общими для всех экземпляров. Наконец, статический метод, оформленный декоратором `@staticmethod`, не получает ни `self`, ни `cls` — он функционирует как обычная функция, логически связанная с классом, но не зависящая от его состояния или состояния его объектов.

Рассмотрим пример, который иллюстрирует эти различия на примере класса, моделирующего сотрудника предприятия. В нём метод экземпляра возвращает информацию о конкретном сотруднике, метод класса обновляет глобальный коэффициент повышения зарплаты для всех сотрудников, а статический метод проверяет, является ли переданная дата рабочим днём.

In [ ]:
```
import datetime
```

In [ ]:
```
class Employee:
    raise_factor = 1.04

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

    def get_details(self):
        return f"Сотрудник {self.name} получает зарплату {self.salary}"

    @classmethod
    def update_raise_factor(cls, new_factor):
        cls.raise_factor = new_factor
        return f"Новый коэффициент повышения установлен: {cls.raise_factor}"

    @staticmethod
    def is_workday(date_obj):
        return date_obj.weekday() < 5
```

In [ ]:
```
emp = Employee("Алексей", 60000)
```

In [ ]:
```
emp.get_details()
```

Out[ ]:
```
'Сотрудник Алексей получает зарплату 60000'
```

In [ ]:
```
Employee.update_raise_factor(1.06)
```

Out[ ]:
```
'Новый коэффициент повышения установлен: 1.06'
```

In [ ]:
```
today = datetime.date.today()
```

In [ ]:
```
print("Сегодня рабочий день?", Employee.is_workday(today))
```

```
Сегодня рабочий день? True
```

В этом примере метод `get_details` является обычным методом экземпляра — он получает `self` и использует индивидуальные атрибуты объекта, а именно, `name` и `salary`.

Метод `update_raise_factor` благодаря `@classmethod` работает с переменной `raise_factor`, которая является общей для всех объектов класса `Employee` — изменение этого коэффициента отражается на всей системе, связанной с оплатой труда.

Статический же метод `is_workday` не зависит ни от состояния конкретного сотрудника, ни от состояния класса — он служит для проверки календарной информации, являясь чистой утилитарной функцией.

Эти различия демонстрируют, как выбор типа метода позволяет точно разграничивать ответственность: методы экземпляра управляют данными отдельного объекта, методы класса обеспечивают управление общим состоянием, а статические методы аккуратно группируют функциональность, не связанную с внутренним состоянием класса или его объектов.

Пример статического метода

Рассмотрим пример, который продемонстрирует, как можно использовать статический метод для вычисления евклидова расстояния между двумя точками на плоскости. Каждая точка задаётся кортежем с координатами (x, y), а само расстояние вычисляется по известной формуле: корень квадратный из суммы квадратов разностей координат. Функция не обращается ни к данным экземпляра, ни к переменным класса, поэтому она оформляется с помощью декоратора `@staticmethod` и вызывается непосредственно через имя класса.

In [ ]:
```python
import math
```

In [ ]:
```python
class GeometryUtil:
    @staticmethod
    def calculate_distance(point_a, point_b):
        x1, y1 = point_a
        x2, y2 = point_b
        return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
```

In [ ]:
```python
point1 = (2, 3)
point2 = (7, 11)
```

In [ ]:
```python
distance = GeometryUtil.calculate_distance(point1, point2)
```

In [ ]:
```python
print(f"Расстояние между точками {point1} и {point2} составляет {distance:.2f}")
```

```
Расстояние между точками (2, 3) и (7, 11) составляет 9.43
```

В этом примере статический метод `calculate_distance` принимает два аргумента — кортежи с координатами двух точек. Он последовательно извлекает значения координат и выполняет вычисления по формуле, известной как теорема Пифагора, что позволяет получить расстояние между точками.

Здесьй не требуется создавать экземпляр класса, поскольку логика расчёта является чистой функцией, зависящей лишь от переданных параметров. Вызов метода происходит через имя класса — `GeometryUtil.calculate_distance` — что подчёркивает утилитарный характер данного метода.

Управление доступом к атрибутам класса

В Python управление доступом к атрибутам класса строится на принципах соглашений и гибкости, что отличает его от строгих модификаторов доступа других языков программирования. Публичные атрибуты, не имеющие специальных префиксов, предназначены для свободного использования, что упрощает взаимодействие с объектами в системах, где оперативность важнее строгой инкапсуляции. Защищённые атрибуты, обозначенные одним подчёркиванием, сигнализируют о необходимости ограничить прямой доступ извне, оставляя их доступными для подклассов и подчёркивая внутреннюю природу данных. Приватные атрибуты с двумя подчёркиваниями используют механизм name mangling, затрудняя случайное вмешательство, что особенно важно для критических данных. Свойства, реализованные через декоратор `@property`, дополняют этот арсенал, позволяя контролировать получение, установку и удаление значений атрибутов с помощью геттеров, сеттеров и делитеров. Такой подход обеспечивает баланс между открытостью и безопасностью, предоставляя разработчику инструменты для точной настройки взаимодействия с данными объекта.

Модификаторы доступа

Публичные атрибуты

Публичные атрибуты отражают дизайн-философию Python, где принцип «доверься разработчику» реализуется через соглашения об именовании. Если атрибут объявлен без ведущего подчёркивания, он предназначен для свободного использования и изменения из любой части программы. Это ценно в системах, где требуется оперативное управление данными, а архитектура предусматривает прямое взаимодействие с объектами без посредничества методов доступа.

Рассмотрим пример, связанный с управлением серверной инфраструктурой в дата-центре. В реальных условиях специалисты отвечают за мониторинг состояния серверов, оперативное обновление их конфигурации и реакцию на возникающие неисправности. Пусть в системе имеется класс, описывающий сервер, где публичные атрибуты задают основные параметры — имя хоста, IP-адрес и текущий статус работы. Эти данные могут изменяться напрямую, например, при переходе сервера в режим обслуживания или восстановлении после аварии.

In [ ]:
```python
class Server:
    def __init__(self, hostname, ip_address, status):
        self.hostname = hostname
        self.ip_address = ip_address
        self.status = status
```

Использование класса в контексте управления дата-центром может выглядеть следующим образом:

In [ ]:
```python
server1 = Server("server1.example.com", "192.168.1.10", "online")
print(f"Сервер: {server1.hostname} с IP {server1.ip_address} находится в статусе {server1.status}")
```

```
Сервер: server1.example.com с IP 192.168.1.10 находится в статусе online
```

In [ ]:
```python
server1.status = "maintenance"
print(f"Обновлённый статус сервера {server1.hostname}: {server1.status}")
```

```
Обновлённый статус сервера server1.example.com: maintenance
```

В этом примере атрибуты `hostname`, `ip_address` и `status` являются публичными и доступны для чтения и изменения

напрямую.

Прямое изменение публичного атрибута, как в строке `server1.status = "maintenance"`, обеспечивает мгновенное обновление данных, что критично для систем, где задержка даже на несколько секунд может повлиять на качество обслуживания.

Использование публичных атрибутов оправдано в тех случаях, когда корректность данных контролируется на более высоком уровне архитектуры системы или когда изменения производятся в рамках чётко регламентированных процессов. При этом разработчик осознаёт, что ответственность за поддержание консистентности значений лежит на нём, а не на самом языке программирования. Такой подход делает код прозрачным и гибким, позволяя быстро интегрировать изменения в систему управления, будь то автоматизированное обновление статуса серверов в режиме реального времени или ручное вмешательство оператора при возникновении сбоев.

Публичные атрибуты, оформленные в виде открытых переменных экземпляра, играют важную роль в построении понятного и легко поддерживаемого интерфейса объектов, особенно в сложных прикладных системах, где оперативное изменение состояния является неотъемлемой частью работы.

Защищенные атрибуты

Защищённые атрибуты представляют собой соглашение, принятое в сообществе Python, которое сигнализирует о том, что определённые данные предназначены исключительно для внутреннего использования классом и его наследниками. Такие атрибуты именуются с одной ведущей подчёркиванием, например, `_attribute`, и служат для разделения публичного интерфейса объекта и его внутренней реализации. Разработчик, встречая имя с одним подчёркиванием, понимает, что напрямую изменять или использовать этот атрибут не стоит, так как его корректное функционирование зависит от специальных методов класса.

Рассмотрим ситуацию из финансовой сферы, где система управления банковскими операциями должна гарантировать целостность данных. Пусть имеется класс, описывающий банковский счёт, в котором баланс хранится во внутреннем атрибуте `_balance`. При этом изменение баланса должно происходить только посредством специально разработанных методов, например, `deposit` и `withdraw`, чтобы гарантировать, что все транзакции проходят валидацию и записываются корректно. Прямое изменение `_balance` извне может привести к нарушению логики учёта операций, поэтому имя с ведущим подчёркиванием служит предупредительным знаком для разработчиков.

In [ ]:
```python
class BankAccount:
    def __init__(self, owner, initial_balance):
        self.owner = owner
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return f"Счёт пополнен на {amount}. Текущий баланс: {self._balance}"
        return "Сумма пополнения должна быть положительной."

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            return f"Со счёта снято {amount}. Остаток: {self._balance}"
        return "Недостаточно средств или неверная сумма для снятия."

    def get_balance(self):
        return self._balance
```

В данном примере защищённый атрибут `_balance` используется для хранения текущего баланса счёта. Методы `deposit` и `withdraw` инкапсулируют логику изменения баланса, гарантируя, что сумма операции проверяется перед внесением изменений. Пользователь класса может узнать значение баланса с помощью метода `get_balance`, но прямое изменение `_balance` (например, присвоение `account._balance = 10000`) нарушит логику работы класса и может привести к некорректным результатам.

Такой подход широко применяется в критически важных системах, где контроль над внутренним состоянием объекта играет ключевую роль. Соглашение об именовании с одним подчёркиванием помогает разработчикам поддерживать чистую архитектуру кода, минимизируя риск случайных изменений внутренних данных. Оно позволяет ясно разграничивать интерфейс, предназначенный для внешнего взаимодействия, и детали реализации, изменять которые следует лишь через специально предусмотренные методы. Это актуально в системах, где соблюдение бизнес-логики имеет первостепенное значение, например, при ведении бухгалтерского учёта или управлении транзакциями в банковской сфере.

Приватные атрибуты

Приватные атрибуты представляют собой данные, имена которых начинаются с двух подчёркиваний, например, `__secret`. Такой способ именования задуман для того, чтобы скрыть детали реализации от внешнего мира и предотвратить случайное или несанкционированное изменение этих данных. Механизм, называемый name mangling, автоматически преобразует имя

приватного атрибута, добавляя к нему префикс в виде `_ИмяКласса`. Это делает прямой доступ к атрибуту извне затруднительным и позволяет избежать конфликтов имён при наследовании.

Представим систему, отвечающую за управление доступом к электронному сейфу. Допустим, класс `SecureVault` хранит конфиденциальный код, который должен быть скрыт от посторонних глаз. Приватный атрибут, содержащий код, объявляется с двумя подчёркиваниями – `__pin`. Внутри класса реализуются методы для проверки введённого кода и безопасного доступа к его значению, если это необходимо. Попытка обратиться к `__pin` извне приведёт к ошибке, поскольку имя атрибута преобразовано механизмом name mangling. Однако, зная правила преобразования, можно обратиться к нему как `_SecureVault__pin`, хотя такой приём нарушает принципы инкапсуляции и используется крайне редко.

In [ ]:
```
class SecureVault:
    def __init__(self, pin):
        self.__pin = pin

    def verify_pin(self, input_pin):
        return self.__pin == input_pin

    def reveal_pin(self):
        return f"Доступ разрешён. Ваш PIN: {self.__pin}"
```

In [ ]:
```
vault = SecureVault("1234")
print(vault.reveal_pin())
```

```
Доступ разрешён. Ваш PIN: 1234
```

Попытка обратиться к приватному атрибуту напрямую вызывает ошибку:

In [ ]:
```
try:
    print(vault.__pin)
except AttributeError as e:
    print("Ошибка доступа к приватному атрибуту:", e)
```

```
Ошибка доступа к приватному атрибуту: 'SecureVault' object has no attribute '__pin'
```

При желании можно обратиться к приватному атрибуту через механизм name mangling:

In [ ]:
```
print(vault._SecureVault__pin)
```

```
1234
```

В данном примере атрибут `__pin` скрыт от прямого доступа, что предохраняет его от случайного изменения извне. Метод `verify_pin` использует этот атрибут для сравнения с введённым значением, гарантируя, что проверка производится корректно. Вызов `vault.__pin` приведёт к ошибке, поскольку имя атрибута преобразовано в `_SecureVault__pin`.

Пример использования

В реальных системах, где безопасность и корректное управление данными имеют первостепенное значение, важно не только разделять уровни доступа к атрибутам, но и обеспечивать инкапсуляцию критически важной информации.

Рассмотрим пример класса `UserAccount`, который иллюстрирует использование публичных, защищённых и приватных атрибутов. Здесь публичные данные — имя пользователя и электронная почта — доступны для внешнего взаимодействия, а конфиденциальный пароль хранится в приватном атрибуте, имя которого начинается с двух подчёркиваний. Дополнительно защищённый атрибут отвечает за подсчёт неудачных попыток входа в систему. Управление приватными данными происходит исключительно через специально разработанные методы, что предотвращает их случайное или несанкционированное изменение.

In [ ]:
```
class UserAccount:
    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self._login_attempts = 0
        self.__password = password

    def attempt_login(self, password_attempt):
        if password_attempt == self.__password:
            self._login_attempts = 0
            return "Доступ разрешён"
        else:
            self._login_attempts += 1
            return f"Неверный пароль. Попыток: {self._login_attempts}"

    def change_password(self, old_password, new_password):
        if old_password == self.__password:
            self.__password = new_password
            return "Пароль изменён успешно"
        return "Неверный старый пароль"

    def get_account_info(self):
```

```
        return f"Пользователь: {self.username}, Email: {self.email}"

    def get_login_attempts(self):
        return self._login_attempts
```

Пример использования данного класса демонстрирует, как методы обеспечивают корректное взаимодействие с приватными данными:

In [ ]:
```
user = UserAccount("ivan", "ivan@gmail.com", "secret123")
```

In [ ]:
```
print(user.get_account_info())
```

```
Пользователь: ivan, Email: ivan@gmail.com
```

In [ ]:
```
print(user.attempt_login("wrongpass"))
```

```
Неверный пароль. Попыток: 1
```

In [ ]:
```
print(user.attempt_login("secret123"))
```

```
Доступ разрешён
```

In [ ]:
```
print(user.change_password("secret123", "newpass456"))
```

```
Пароль изменён успешно
```

In [ ]:
```
print(user.change_password("wrongpass", "anotherpass"))
```

```
Неверный старый пароль
```

В данном примере приватный атрибут `__password` остаётся скрытым благодаря механизму name mangling, что предотвращает прямой доступ извне. Любые операции, связанные с проверкой или изменением пароля, выполняются через методы `attempt_login` и `change_password`, которые инкапсулируют всю необходимую логику. Защищённый атрибут `_login_attempts` аккумулирует количество неудачных попыток входа, что может использоваться для реализации механизмов блокировки или уведомления службы безопасности.

Свойства

Декоратор `@property`

Декоратор `@property` позволяет объявить метод, к которому можно обращаться как к обычному атрибуту, без круглых скобок. При этом метод становится так называемым **геттером** — специальной функцией для получения значения скрытого или вычисляемого атрибута объекта. Геттер обеспечивает доступ к данным, позволяя не раскрывать детали их хранения или вычисления, что упрощает взаимодействие с объектом и повышает уровень инкапсуляции.

Представим реальную ситуацию из области промышленной автоматизации. Допустим, в системе контроля температуры на производственной линии используется датчик, измеряющий температуру в градусах Цельсия. Значение, получаемое с датчика, может требовать предварительной обработки или преобразования. Чтобы пользователь системы мог получить данные напрямую, без необходимости знать, как происходит вычисление, создаётся геттер с помощью декоратора `@property`.

Например, класс `SmartThermostat` хранит температуру в защищённом атрибуте, а геттер позволяет получить это значение как простое свойство объекта. При этом, если потребуется добавить дополнительную логику (например, округление или валидацию), она будет выполнена внутри геттера, оставаясь невидимой для внешнего кода.

In [ ]:
```
class SmartThermostat:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def temperature(self):
        return round(self._celsius, 1)

    @property
    def temperature_f(self):
     return self._celsius * 9/5 + 32
```

In [ ]:
```
thermostat = SmartThermostat(23.567)
```

In [ ]:
```
print(f"Температура: {thermostat.temperature}°C")
print(f"В Фаренгейтах: {thermostat.temperature_f:.2f}°F")
```

```
Температура: 23.6°C
В Фаренгейтах: 74.42°F
```

В данном примере метод `temperature` является геттером, который возвращает значение защищённого атрибута `_celsius` в виде округлённого до одной десятой числа. Пользователь класса может обращаться к `temperature` как к простому атрибуту, не зная о внутренней реализации хранения данных. Аналогично, геттер `temperature_f` выполняет вычисление температуры в Фаренгейтах по известной формуле.

Сеттеры и делитеры

Сеттеры и делитеры представляют собой механизм управления свойствами объекта, позволяющий не просто получать значение, но и контролировать его установку и удаление.

Сеттер, определяемый через декоратор `@имя_свойства.setter`, перехватывает операцию присваивания, давая возможность проверить и, при необходимости, модифицировать новое значение перед сохранением.

Делитер, оформляемый с помощью `@имя_свойства.deleter`, задаёт действия, которые выполняются при удалении свойства — это может быть логирование, очистка ресурсов или сброс состояния.

Рассмотрим класс `Product`, описывающий товар в системе управления складом. В этом классе цена товара хранится в защищённом атрибуте `_price`. Сеттер свойства `price` следит за тем, чтобы новая цена не была отрицательной, и генерирует исключение, если условие нарушается. Делитер, в свою очередь, сообщает о том, что данные о цене удаляются, и осуществляет сам процесс удаления. Это позволит поддерживать корректность данных и централизованно контролировать изменения критичных параметров.

In [ ]:
```
class Product:
    def __init__(self, name, price):
        self.name = name
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price < 0:
            raise ValueError("Цена не может быть отрицательной.")
        self._price = new_price

    @price.deleter
    def price(self):
        print(f"Удаляем информацию о цене для товара: {self.name}.")
        del self._price
```

Рассмотрим, как можно воспользоваться реализованными сеттером и делитером:

In [ ]:
```
product = Product("Ноутбук", 75000)
print(f"Начальная цена товара '{product.name}': {product.price} руб.")
```

```
Начальная цена товара 'Ноутбук': 75000 руб.
```

In [ ]:
```
product.price = 70000
print(f"Обновлённая цена товара '{product.name}': {product.price} руб.")
```

```
Обновлённая цена товара 'Ноутбук': 70000 руб.
```

In [ ]:
```
try:
    product.price = -5000
except ValueError as error:
    print("Ошибка изменения цены:", error)
```

```
Ошибка изменения цены: Цена не может быть отрицательной.
```

In [ ]:
```
print(f"Цена товара '{product.name}': {product.price} руб.")
```

```
Цена товара 'Ноутбук': 70000 руб.
```

In [ ]:
```
del product.price
```

```
Удаляем информацию о цене для товара: Ноутбук.
```

In [ ]:
```
try:
    print(product.price)
except AttributeError as error:
    print("После удаления цена недоступна:", error)
```

```
После удаления цена недоступна: 'Product' object has no attribute '_price'
```

В этом примере класс `Product` хранит в защищённом атрибуте `_price` информацию о цене товара и предоставляет к ней доступ через свойство `price`. Благодаря декоратору `@property` внешний код может обращаться к `price` так же, как к любому открытому атрибуту, однако «под капотом» вызывается соответствующий метод.

Сеттер (`@price.setter`) реагирует на попытки изменить цену, проверяя, не является ли новое значение отрицательным, и выбрасывая исключение `ValueError` при нарушении этого условия.

Делитер (`@price.deleter`) обеспечивает выполнение дополнительной логики (например, вывод сообщения в консоль) при удалении атрибута цены и физически удаляет `_price` из объекта.

В примере использования при создании объекта устанавливается валидная цена (75000), которая затем обновляется до 70000. Попытка назначить отрицательную цену приводит к срабатыванию проверки и выбросу исключения, а поскольку при этом никакое изменение не вносится, стоимость остаётся прежней.

Наконец, вызов `del product.price` инициирует делитер, который выводит уведомление об удалении цены и удаляет `_price` из объекта; последующая попытка обратиться к цене уже вызывает `AttributeError`, поскольку соответствующего атрибута больше не существует.

Применение свойств

Свойства в Python предоставляют способ «обернуть» атрибут объекта, позволяя контролировать его получение и изменение. При применении геттеров, сеттеров и делитеров можно встроить проверку значений, выполнить преобразования или даже автоматически обновлять связанные данные, что помогает избежать ошибок при непосредственном присваивании. Этот приём часто используют для обеспечения валидации данных и для того, чтобы скрыть детали внутренней реализации.

Представим ситуацию из автомобильной отрасли. При разработке системы мониторинга транспортных средств класс, описывающий автомобиль, должен хранить скорость, измеряемую в километрах в час. Чтобы избежать установки некорректных значений, например, отрицательной скорости или чисел, превышающих допустимый предел, свойство `speed` оформляют с геттером и сеттером. Геттер позволяет получать текущее значение, а сеттер проверяет новое значение перед его сохранением. Если пользователь пытается установить скорость, которая не удовлетворяет условиям (например, отрицательное число или число, превышающее максимально допустимое значение), система сразу сообщает об ошибке, выбрасывая исключение.

In [ ]:
```python
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self._speed = speed

    @property
    def speed(self):
        return self._speed

    @speed.setter
    def speed(self, value):
        MAX_SPEED = 300
        if not isinstance(value, (int, float)):
            raise TypeError("Скорость должна быть числом.")
        if value < 0:
            raise ValueError("Скорость не может быть отрицательной.")
        if value > MAX_SPEED:
            raise ValueError(f"Скорость не может превышать {MAX_SPEED} км/ч.")
        self._speed = value

    @speed.deleter
    def speed(self):
        print(f"Сброс значения скорости для автомобиля {self.brand}.")
        del self._speed
```

Использование данного класса может выглядеть следующим образом:

In [ ]:
```python
car = Car("Toyota", 120)
print(f"Начальная скорость {car.brand}: {car.speed} км/ч")
```

```
Начальная скорость Toyota: 120 км/ч
```

In [ ]:
```python
car.speed = 150
print(f"Обновлённая скорость {car.brand}: {car.speed} км/ч")
```

```
Обновлённая скорость Toyota: 150 км/ч
```

In [ ]:
```python
try:
    car.speed = -50
except ValueError as e:
    print("Ошибка установки скорости:", e)
```

```
Ошибка установки скорости: Скорость не может быть отрицательной.
```

In [ ]:
```python
try:
    car.speed = 350
except ValueError as e:
    print("Ошибка установки скорости:", e)
```

```
Ошибка установки скорости: Скорость не может превышать 300 км/ч.
```

In [ ]:
```python
del car.speed
try:
    print(car.speed)
except AttributeError as e:
    print("После удаления доступа к скорости нет:", e)
```

```
Сброс значения скорости для автомобиля Toyota.
После удаления доступа к скорости нет: 'Car' object has no attribute '_speed'
```

В этом примере геттер позволяет внешнему коду обращаться к свойству `speed` как к обычному полю, а сеттер выполняет проверку перед присвоением нового значения. Делитер, в свою очередь, задаёт поведение при удалении свойства, например, сбрасывая скорость и информируя об этом пользователя.

Пример использования

Свойства позволяют не только скрыть внутреннее представление данных, но и обеспечить автоматическую проверку вводимых значений, а также динамический расчёт параметров объекта. Такой подход широко применяется в системах, где точность финансовых показателей и корректность данных напрямую влияют на принятие управленческих решений.

Рассмотрим класс, моделирующий заявку на кредит, где свойства используются для валидации основных параметров и вычисления дополнительных характеристик заявки. В классе `LoanApplication` задаются такие поля, как доход заявителя, запрашиваемая сумма кредита и кредитный рейтинг. Геттеры и сеттеры для этих полей гарантируют, что значения соответствуют допустимым диапазонам: доход не может быть отрицательным, сумма кредита должна быть положительной, а кредитный рейтинг — находиться в диапазоне от 300 до 850. Кроме того, свойство `interest_rate` вычисляет процентную ставку по заявке в зависимости от кредитного рейтинга, а свойство `eligibility` определяет возможность одобрения кредита по простейшим критериям. Делитер для свойства `eligibility` может использоваться для сброса статуса, если данные заявки подвергаются пересмотру.

In [ ]:
```python
class LoanApplication:
    def __init__(self, applicant, income, requested_amount, credit_score):
        self.applicant = applicant
        self._income = income
        self._requested_amount = requested_amount
        self._credit_score = credit_score

    @property
    def income(self):
        return self._income

    @income.setter
    def income(self, value):
        if value < 0:
            raise ValueError("Доход не может быть отрицательным.")
        self._income = value

    @property
    def requested_amount(self):
        return self._requested_amount

    @requested_amount.setter
    def requested_amount(self, value):
        if value <= 0:
            raise ValueError("Запрашиваемая сумма должна быть положительной.")
        self._requested_amount = value

    @property
    def credit_score(self):
        return self._credit_score

    @credit_score.setter
    def credit_score(self, value):
        if not (300 <= value <= 850):
            raise ValueError("Кредитный рейтинг должен быть от 300 до 850.")
        self._credit_score = value

    @property
    def interest_rate(self):
        if self._credit_score >= 750:
            return 5.0
        elif self._credit_score >= 650:
            return 7.5
        else:
            return 10.0

    @property
    def eligibility(self):
        return (self._income >= 3 * self._requested_amount and
                self._credit_score > 600)

    @eligibility.deleter
    def eligibility(self):
        print("Статус заявки на кредит сброшен. Требуется повторная оценка.")

    def __str__(self):
        return (
```

```
            "Заявка на кредит:\n"
            f"  Заявитель: {self.applicant}\n"
            f"  Доход: {self.income} руб.\n"
            f"  Запрашиваемая сумма: {self.requested_amount} руб.\n"
            f"  Кредитный рейтинг: {self.credit_score}\n"
            f"  Процентная ставка: {self.interest_rate}%\n"
            f"  Одобрена заявка: {self.eligibility}"
            )
```

В следующем фрагменте показано применение класса `LoanApplication`. Сначала создаётся объект заявки на кредит для заявителя «Анна Иванова» с указанными параметрами дохода, запрашиваемой суммы и кредитного рейтинга. При вызове `print(loan_app)` автоматически вызывается метод `__str__`, который возвращает отформатированную строку с полной информацией о заявке. Затем демонстрируется попытка установить недопустимое значение для дохода — сеттер `income` проверяет новое значение и, при обнаружении ошибки, выбрасывает исключение. После этого обновляются значения запрашиваемой суммы и кредитного рейтинга, что приводит к пересчёту динамических свойств — `interest_rate` и `eligibility`. В завершении применяется делитер для свойства `eligibility`, который сбрасывает статус заявки, информируя об этом пользователя.

In [ ]:
```
loan_app = LoanApplication(
    "Анна Иванова",
    income=120000,
    requested_amount=30000,
    credit_score=720
    )
```

In [ ]:
```
print(loan_app)
```

```
Заявка на кредит:
  Заявитель: Анна Иванова
  Доход: 120000 руб.
  Запрашиваемая сумма: 30000 руб.
  Кредитный рейтинг: 720
  Процентная ставка: 7.5%
  Одобрена заявка: True
```

In [ ]:
```
try:
    loan_app.income = -50000
except ValueError as e:
    print("Ошибка при изменении дохода:", e)
```

```
Ошибка при изменении дохода: Доход не может быть отрицательным.
```

In [ ]:
```
loan_app.requested_amount = 35000
loan_app.credit_score = 680
```

In [ ]:
```
print(loan_app)
```

```
Заявка на кредит:
  Заявитель: Анна Иванова
  Доход: 120000 руб.
  Запрашиваемая сумма: 35000 руб.
  Кредитный рейтинг: 680
  Процентная ставка: 7.5%
  Одобрена заявка: True
```

In [ ]:
```
del loan_app.eligibility
```

```
Статус заявки на кредит сброшен. Требуется повторная оценка.
```

Таким образом, свойства в классе `LoanApplication` позволяют централизованно контролировать ввод данных и динамически вычислять дополнительные характеристики (процентную ставку и возможность одобрения заявки), а метод `__str__` обеспечивает удобное форматирование вывода всей информации об объекте.

Динамические операции с атрибутами и интроспекция

Современные приложения часто требуют гибкости в управлении объектами, что делает динамические операции с атрибутами и интроспекцию важными аспектами программирования в Python. Функции вроде `getattr()`, `setattr()`, `hasattr()` и `delattr()` позволяют обращаться к атрибутам, изменять их или удалять, используя имена в виде строк, что особенно полезно при работе с данными, структура которых определяется в процессе выполнения. Интроспекция, в свою очередь, даёт программе способность анализировать собственную структуру: через функцию `dir()` можно получить список доступных атрибутов и методов, а доступ к `__dict__` раскрывает внутреннее состояние объекта. Модуль `inspect` расширяет эти возможности, предоставляя детальную информацию о сигнатурах методов, документации и даже исходном коде. Эти инструменты делают Python мощным языком для создания адаптивных систем, где объекты могут динамически изменяться, а код — самонастраиваться в зависимости от текущих требований и состояния.

Динамические операции

Функция `getattr()`

Функция `getattr()` — это встроенная функция Python, которая позволяет динамически получать значение атрибута объекта по его имени, заданному в виде строки. Этот метод может понадобиться, когда имена атрибутов неизвестны заранее или определяются в ходе выполнения программы, например, при обработке конфигурационных данных или при работе с объектами, структура которых может варьироваться.

Функция принимает три аргумента: объект, строку с именем атрибута и необязательное значение по умолчанию, которое возвращается, если атрибут отсутствует. Это позволяет избежать выброса исключения `AttributeError` и делает код более универсальным.

Рассмотрим систему управления умными устройствами, где класс `SmartHomeDevice` описывает устройство с обязательными атрибутамик `device_id`, `device_type` и `status`, а также с дополнительным атрибутом `battery_level`, который может отсутствовать у проводных устройств. При обработке информации о таких устройствах можно использовать `getattr()`, чтобы в цикле пройти по списку ожидаемых атрибутов и получить их значения, при этом для отсутствующих атрибутов возвращать значение по умолчанию, например, `"Нет данных"`. Это поможет динамически формировать отчёты или проводить валидацию данных без жесткого кодирования имён атрибутов.

В приведённом ниже коде создаётся объект устройства, после чего в цикле с помощью `getattr(device, attr, "Нет данных")` извлекаются значения атрибутов из списка, что демонстрирует, как можно легко адаптировать обработку данных в условиях изменяющейся структуры объекта.

In [ ]:
```python
class SmartHomeDevice:
    def __init__(self, device_id, device_type, status, battery_level=None):
        self.device_id = device_id
        self.device_type = device_type
        self.status = status
        if battery_level is not None:
            self.battery_level = battery_level
```

In [ ]:
```python
device = SmartHomeDevice("DEV001", "Thermostat", "Online")
```

In [ ]:
```python
attributes = ["device_id", "device_type", "status", "battery_level"]
```

In [ ]:
```python
for attr in attributes:
    value = getattr(device, attr, "Нет данных")
    print(f"{attr}: {value}")
```

```
device_id: DEV001
device_type: Thermostat
status: Online
battery_level: Нет данных
```

В этом примере цикл проходит по списку атрибутов и с помощью `getattr()` получает их значения из объекта `device`. Если, например, атрибут `battery_level` отсутствует, функция возвращает строку `"Нет данных"`, что предотвращает ошибку и позволяет коду работать гибко независимо от структуры объекта.

Данный пример демонстрирует, что `getattr()` является незаменимым инструментом для динамической обработки данных, позволяющим адаптировать код к различным сценариям и упрощать работу с объектами, структура которых может изменяться во время выполнения программы.

Функция `setattr()`

Функция `setattr()` позволяет программно установить значение атрибута объекта, когда имя атрибута передаётся в виде строки, что особенно удобно при динамическом формировании набора свойств объекта или обработке внешних данных, где имена атрибутов могут меняться во время выполнения программы.

Функция принимает три аргумента: объект, имя атрибута в виде строки и новое значение для этого атрибута. Если указанный атрибут уже существует, его значение обновляется; если же такого атрибута нет, он создаётся автоматически, что позволяет гибко модифицировать состояние объекта без явного обращения к его внутреннему словарю.

Рассмотрим пример, где класс `SmartDevice` представляет устройство с базовыми характеристиками `name`, `status` и `mode`. При создании объекта эти свойства инициализируются начальными значениями, а затем с помощью `setattr()` можно изменять их или добавлять новые свойства, например, `temperature`, если устройство поддерживает измерение температуры. Данный подход позволяет адаптировать объект к изменяющимся условиям, получая данные с датчиков или от удалённых сервисов, и изменять его конфигурацию без жесткого кодирования имён атрибутов.

In [ ]:
```python
class SmartDevice:
    def __init__(self, name):
        self.name = name
        self.status = "offline"
        self.mode = "normal"
```

In [ ]:
```
device = SmartDevice("Thermostat")
```

In [ ]:
```
setattr(device, "status", "online")
setattr(device, "mode", "energy_saving")
```

In [ ]:
```
setattr(device, "temperature", 22.5)
```

In [ ]:
```
print(
    f"Устройство: {device.name}\n"
    f"Статус: {device.status}\n"
    f"Режим: {device.mode}\n"
    f"Температура: {device.temperature}°C"
    )
```

```
Устройство: Thermostat
Статус: online
Режим: energy_saving
Температура: 22.5°C
```

В данном примере объект `device` класса `SmartDevice` сначала создаётся с начальными значениями для атрибутов `name`, `status` и `mode`. Затем с помощью `setattr(device, "status", "online")` изменяется значение атрибута `status`, а вызов `setattr(device, "mode", "energy_saving")` обновляет режим работы устройства.

Динамическое добавление атрибута происходит при вызове `setattr(device, "temperature", 22.5)`, что создаёт новый атрибут `temperature` и присваивает ему значение 22.5. После этого с помощью одного оператора `print` выводится полная информация об устройстве, демонстрируя, как `setattr()` позволяет изменять и расширять объект в ходе выполнения программы.

Функция `hasattr()`

Функция `hasattr()` предоставляет механизм для динамической проверки наличия определённого атрибута в объекте. Она принимает два аргумента: сам объект и имя атрибута в виде строки. Если объект имеет указанный атрибут, функция возвращает `True`, в противном случае — `False`. Это позволяет, прежде чем обращаться к атрибуту, убедиться в его существовании, что особенно полезно в тех случаях, когда структура объекта может варьироваться в зависимости от входных данных или конфигурации системы. Например, в системе управления персоналом не все сотрудники могут иметь указанный номер телефона. При обработке данных можно использовать `hasattr()`, чтобы проверить наличие атрибута `phone_number` у каждого объекта и, в зависимости от результата, либо вывести информацию о номере, либо сообщить об отсутствии данных. Рассмотрим следующий пример:

In [ ]:
```
class Employee:
    def __init__(self, name, position, phone_number=None):
        self.name = name
        self.position = position
        if phone_number is not None:
            self.phone_number = phone_number
```

In [ ]:
```
emp1 = Employee("Иван", "Разработчик", phone_number="123-456-7890")
emp2 = Employee("Мария", "Дизайнер")
```

In [ ]:
```
for emp in [emp1, emp2]:
    if hasattr(emp, "phone_number"):
        print(f"Сотрудник {emp.name} имеет номер телефона: {emp.phone_number}")
    else:
        print(f"Сотрудник {emp.name} не указал номер телефона.")
```

```
Сотрудник Иван имеет номер телефона: 123-456-7890
Сотрудник Мария не указал номер телефона.
```

В этом примере для каждого объекта класса `Employee` с помощью `hasattr(emp, "phone_number")` проверяется, существует ли атрибут `phone_number`. Если атрибут присутствует, выводится его значение, а если отсутствует — генерируется соответствующее сообщение. Это помогает избежать ошибок доступа к несуществующим атрибутам и писать гибкий, адаптивный код, который корректно работает с объектами, имеющими различную структуру данных.

Функция `delattr()`

Функция `delattr()` предоставляет возможность программно удалить атрибут из объекта, когда его имя передаётся в виде строки. Эта встроенная функция принимает два аргумента: объект и строку с именем удаляемого атрибута. При наличии указанного атрибута он исключается из объекта, что позволяет изменять его внутреннюю структуру во время выполнения программы. Если же атрибут отсутствует, генерируется исключение, что помогает обнаруживать рассинхронизацию между ожидаемой и фактической структурой объекта.

Пусть класс `Employee` создаёт объект сотрудника с тремя свойствами: `name`, `position` и `salary`. Если в процессе работы оказывается, что информация о должности больше не нужна или должна быть изменена, можно удалить атрибут `position` с помощью `delattr()`. После удаления функция `hasattr()` позволяет проверить, что атрибут действительно

отсутствует, предотвращая ошибки при последующем обращении к нему.

In [ ]:
```
class Employee:
    def __init__(self, name, position, salary):
        self.name = name
        self.position = position
        self.salary = salary
```

In [ ]:
```
emp = Employee("Иван", "Разработчик", 70000)
```

In [ ]:
```
print(f"Имя: {emp.name}\nДолжность: {emp.position}\nЗарплата: {emp.salary}")
```

```
Имя: Иван
Должность: Разработчик
Зарплата: 70000
```

In [ ]:
```
delattr(emp, "position")
```

In [ ]:
```
if not hasattr(emp, "position"):
    print("Атрибут 'position' успешно удалён.")
else:
    print(f"Должность: {emp.position}")
```

```
Атрибут 'position' успешно удалён.
```

В этом примере объект `emp` класса `Employee` создаётся с тремя свойствами. После вызова `delattr(emp, "position")` свойство `position` исключается из объекта, и последующая проверка через `hasattr()` подтверждает, что атрибут отсутствует. Таком образом, можно динамически изменять набор данных, хранящихся в объекте, что полезно при адаптации структуры объекта к изменяющимся требованиям приложения.

Применение

Динамические операции, то есть использование функций `getattr()`, `setattr()`, `hasattr()` и `delattr()`, открывают широкие возможности для изменения структуры объектов в рантайме и позволяют создавать адаптивные классы, способные реагировать на внешние данные и изменяющиеся условия работы приложения.

Представим класс `DynamicEntity`, который на этапе инициализации принимает произвольное число именованных аргументов и устанавливает их в качестве атрибутов посредством `setattr()`. Метод `update_attributes()` позволяет обновлять существующие свойства или добавлять новые, а метод `remove_attribute()` проверяет наличие атрибута с помощью `hasattr()` и, если он найден, удаляет его с помощью `delattr()`. Такой дизайн делает объекты максимально гибкими — их можно динамически настраивать в зависимости от поступающих данных, будь то конфигурация устройств, мониторинг состояния или работа с изменяемыми наборами параметров. Ниже приведён пример, демонстрирующий, как с помощью этих функций можно создавать и изменять объект, а также удалять ненужные атрибуты:

In [ ]:
```
class DynamicEntity:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    def update_attributes(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

    def remove_attribute(self, attr_name):
        if hasattr(self, attr_name):
            delattr(self, attr_name)
        else:
            print(f"Attribute '{attr_name}' does not exist.")

    def __str__(self):
        return ", ".join(f"{key}={value}" for key, value in self.__dict__.items())
```

In [ ]:
```
entity = DynamicEntity(name="Device1", status="active", battery=95)
print("Начальное состояние:", entity)
```

```
Начальное состояние: name=Device1, status=active, battery=95
```

In [ ]:
```
entity.update_attributes(status="inactive", location="Room A")
print("После обновления:", entity)
```

```
После обновления: name=Device1, status=inactive, battery=95, location=Room A
```

In [ ]:
```
entity.remove_attribute("battery")
print("После удаления 'battery':", entity)
```

```
После удаления 'battery': name=Device1, status=inactive, location=Room A
```

В этом примере объект класса `DynamicEntity` изначально создаётся с атрибутами `name`, `status` и `battery`. Затем метод `update_attributes()` изменяет значение `status` и добавляет новый атрибут `location`, что иллюстрирует

динамическое изменение состояния объекта. Метод `remove_attribute()` проверяет наличие атрибута `battery` и, если он обнаружен, удаляет его с помощью `delattr()`. Итоговое строковое представление объекта, сформированное методом `__str__`, показывает актуальные свойства, демонстрируя, как динамические операции позволяют создавать гибкие и легко настраиваемые классы, способные адаптироваться к изменяющимся требованиям приложения.

Интроспекция

Понятие интроспекции

Интроспекция — это способность программы во время выполнения анализировать свою собственную структуру, получать информацию о своих классах, объектах, методах, атрибутах и других составляющих. Эта возможность позволяет программе «заглянуть внутрь себя», узнать, какие компоненты ей доступны, и на основе полученных данных принимать решения или динамически изменять своё поведение.

В динамических языках программирования, таких как Python, интроспекция играет ключевую роль, поскольку объекты могут изменять свою структуру в процессе работы, а типы проверяются не жестко, а по наличию необходимых методов и свойств. Благодаря функциям, например, `dir()`, можно получить список всех атрибутов и методов объекта, а доступ к внутреннему словарю через `__dict__` позволяет увидеть все данные, хранящиеся в объекте, что полезно для отладки и анализа.

Модуль `inspect` расширяет эти возможности, предоставляя инструменты для получения подробной информации о сигнатурах функций, структуре классов и модулей, что облегчает разработку адаптивных систем, где новые компоненты могут автоматически интегрироваться в уже существующую архитектуру.

Интроспекция позволяет реализовать динамическое создание и модификацию объектов, поскольку программа может изучать собственную конфигурацию и на её основе корректировать логику работы. Это особенно важно в сложных приложениях, где своевременный анализ и корректировка структуры данных способствуют более эффективной отладке, расширению функционала и повышению устойчивости системы. Таким образом, интроспекция становится мощным инструментом, который позволяет программам быть самосознательными, способными адаптироваться к изменениям и обеспечивать гибкое управление своими компонентами в режиме реального времени.

Использование функции `dir()`

Интроспекция позволяет программе исследовать свою структуру во время выполнения, что особенно ценно для разработки гибких и адаптивных систем. С помощью функции `dir()` можно получить список всех имен, определённых в объекте, что даёт возможность быстро понять, какие атрибуты и методы доступны для работы. Такой механизм может использоваться, например, в системе управления серверами, когда необходимо динамически собрать информацию о сервере для отладки или автоматической настройки.

Представим класс `Server`, описывающий сервер с основными характеристиками – именем, IP-адресом и нагрузкой, а также с методами для перезапуска и обновления конфигурации. Используя `dir()`, можно получить список всех публичных элементов объекта, отфильтровать их, разделив на атрибуты и методы, и таким образом сформировать подробный отчёт о состоянии объекта. Ниже приведён пример кода, демонстрирующий этот подход:

In [ ]:
```
class Server:
    def __init__(self, hostname, ip_address, load):
        self.hostname = hostname
        self.ip_address = ip_address
        self.load = load

    def restart(self):
        print("Сервер перезапущен.")

    def update_config(self, config):
        print("Конфигурация сервера обновлена.")
```

In [ ]:
```
server = Server("server1.example.ru", "192.168.0.1", 0.85)
```

Получаем список всех имен в объекте с помощью `dir()`:

In [ ]:
```
all_names = dir(server)
```

Фильтруем список, оставляя только публичные имена (без системных __имён__)

In [ ]:
```
public_names = [name for name in all_names if not name.startswith('__')]
```

In [ ]:
```
Разделяем публичные имена на методы и атрибуты:
```

In [ ]:
```
methods = [name for name in public_names if callable(getattr(server, name))]
attributes = [name for name in public_names if not callable(getattr(server, name))]
```

In [ ]:
```
print("Публичные атрибуты и методы объекта server:")
```

```
print("Атрибуты:", attributes)
print("Методы:", methods)
```

```
Публичные атрибуты и методы объекта server:
Атрибуты: ['hostname', 'ip_address', 'load']
Методы: ['restart', 'update_config']
```

В этом примере функция `dir(server)` возвращает полный список имен, доступных в объекте `server`. Затем с помощью спискового включения отбираются только те имена, которые не начинаются с двойного подчёркивания, что позволяет исключить системные элементы. После этого происходит разделение полученного списка: проверка с помощью `callable(getattr(server, name))` позволяет определить, какие имена соответствуют методам, а какие — данным. Итоговый вывод демонстрирует, какие именно атрибуты (например, `hostname`, `ip_address`, `load`) и методы (например, `restart`, `update_config`) доступны для объекта, что облегчает дальнейшую динамическую работу с ним. Данный способ интроспекции предоставляет разработчику мгновенный обзор структуры объекта, что может оказаться незаменимым при отладке, автоматической генерации документации или динамической интеграции модулей в систему.

Доступ к атрибутам через `__dict__`

У большинства объектов Python имеется специальный атрибут `__dict__`, представляющий собой словарь, где хранятся все изменяемые атрибуты объекта в виде пар «имя–значение». Это означает, что ключами словаря являются имена атрибутов, а значениями — их текущие состояния.

Такой прямой доступ к внутреннему пространству имён объекта позволяет не только проводить подробный анализ его состояния для отладки и документирования, но и динамически модифицировать, добавлять или удалять атрибуты в рантайме, что может быть особенно полезно в адаптивных системах, где структура объектов изменяется в зависимости от внешних данных.

Рассмотрим следующий пример, иллюстрирующий работу с `__dict__` на примере класса `Car`:

In [ ]:
```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.status = "New"
```

In [ ]:
```
car = Car("Toyota", "Camry", 2020)
```

In [ ]:
```
print("Начальное состояние объекта:", car.__dict__)
```

```
Начальное состояние объекта: {'brand': 'Toyota', 'model': 'Camry', 'year': 2020, 'status': 'New'}
```

Модифицируем атрибуты объекта напрямую через `__dict__`:

In [ ]:
```
car.__dict__["status"] = "Used"
car.__dict__["color"] = "Black"
del car.__dict__["year"]
```

In [ ]:
```
print("Изменённое состояние объекта:", car.__dict__)
```

```
Изменённое состояние объекта: {'brand': 'Toyota', 'model': 'Camry', 'status': 'Used', 'color': 'Black'}
```

Здесь класс `Car` создаёт объект с начальными атрибутами: `brand`, `model`, `year` и `status`. Обратившись к `car.__dict__`, мы получаем словарь, в котором перечислены все установленные атрибуты. Далее мы напрямую изменяем значение атрибута `status`, добавляем новый атрибут `color` и удаляем атрибут `year` через манипуляцию словарём. Этот способ доступа позволяет динамически управлять состоянием объекта без необходимости определения специальных методов, что может понадобиться, например, при отладке или при создании объектов, структура которых меняется в зависимости от входных данных или конфигурации системы.

Модуль `inspect`

Модуль `inspect` предоставляет разработчику возможность глубоко анализировать объекты во время выполнения программы, раскрывая их внутреннюю структуру, атрибуты, методы, а также параметры функций. Эта функциональность позволяет «заглянуть под капот» кода и получить подробную информацию о том, какие компоненты определены в классе или функции, какие аргументы принимает функция, а также извлечь документацию и даже исходный код. Благодаря этому модуль широко используется для создания инструментов автоматизированной отладки, генерации документации, тестирования и динамической адаптации программ, когда система должна работать с объектами, структура которых может изменяться во время выполнения.

Например, функция `inspect.getmembers()` возвращает список всех членов объекта, что позволяет составить детальную карту его методов и атрибутов, а `inspect.getdoc()` извлекает строку документации, которая описывает назначение функции или класса. Функция `inspect.signature()` возвращает объект сигнатуры, демонстрируя, какие параметры принимает функция, а `inspect.getsource()` позволяет получить исходный код, что может помочь при анализе алгоритмов или поиске ошибок. Следующий пример демонстрирует возможности модуля `inspect` для анализа класса и функции:

In [ ]:
```python
import inspect
```

In [ ]:
```python
class ServerConfig:
    """
    Класс для демонстрации возможностей модуля inspect.
    Представляет конфигурацию сервера с различными параметрами.
    """
    def __init__(self, hostname, ip, port):
        self.hostname = hostname
        self.ip = ip
        self.port = port

    def restart(self):
        """Перезапускает сервер."""
        print("Сервер перезапущен.")

    @classmethod
    def get_default_config(cls):
        """Возвращает конфигурацию сервера по умолчанию."""
        return cls("localhost", "127.0.0.1", 8080)
```

In [ ]:
```python
def process_config(config):
    """
    Функция для обработки конфигурации сервера.
    Принимает объект ServerConfig и возвращает строку с информацией.
    """
    return f"Сервер {config.hostname} работает на {config.ip}:{config.port}"
```

Получим список членов класса ServerConfig, причем выведем только пользовательские атрибуты и методы, исключая встроенные элементы:

In [ ]:
```python
members = inspect.getmembers(ServerConfig)
print("Члены класса ServerConfig:")
for name, member in members:
    if not name.startswith('__'):
        print(f"  {name}: {member}")
```

```
Члены класса ServerConfig:
  get_default_config: <bound method ServerConfig.get_default_config of <class '__main__.ServerConfig'>>
  restart: <function ServerConfig.restart at 0x7e607d615800>
```

Извлечём документацию класса и функции:

In [ ]:
```python
class_doc = inspect.getdoc(ServerConfig)
func_doc = inspect.getdoc(process_config)
print("\nДокументация класса ServerConfig:")
print(class_doc)
print("\nДокументация функции process_config:")
print(func_doc)
```

```
Документация класса ServerConfig:
Класс для демонстрации возможностей модуля inspect.
Представляет конфигурацию сервера с различными параметрами.

Документация функции process_config:
Функция для обработки конфигурации сервера.
Принимает объект ServerConfig и возвращает строку с информацией.
```

Получим сигнатуру функции `process_config`:

In [ ]:
```python
signature = inspect.signature(process_config)
print("\nСигнатура функции process_config:")
print(signature)
```

```
Сигнатура функции process_config:
(config)
```

Наконец, получим исходный код функции `process_config`:

In [ ]:
```python
source_code = inspect.getsource(process_config)
print("\nИсходный код функции process_config:")
print(source_code)
```

```
Исходный код функции process_config:
def process_config(config):
    """
    Функция для обработки конфигурации сервера.
    Принимает объект ServerConfig и возвращает строку с информацией.
    """
    return f"Сервер {config.hostname} работает на {config.ip}:{config.port}"
```

В данном примере класс `ServerConfig` определяет несколько атрибутов и методов, а функция `process_config` обрабатывает объект конфигурации. Сначала с помощью `inspect.getmembers(ServerConfig)` собирается список всех членов класса, после чего фильтруются встроенные элементы. Затем функции `inspect.getdoc()` извлекают описания класса и функции, что помогает понять их назначение. Функция `inspect.signature()` показывает, какие аргументы принимает `process_config`, а `inspect.getsource()` возвращает полный исходный код функции. Таким образом, модуль `inspect` позволяет динамически исследовать объекты, что является важным инструментом для отладки, анализа и адаптивного управления программным обеспечением.

Применение интроспекции

Интроспекция даёт возможность программе наблюдать за собственной структурой, что открывает широкие перспективы для отладки и динамического управления объектами во время выполнения. Благодаря таким функциям, как `dir()`, доступ к `__dict__`, модулю `inspect` и другим методам анализа, разработчик может узнать, какие атрибуты и методы присутствуют у объекта, какие значения им присвоены, а также динамически изменять или создавать новые атрибуты, адаптируя поведение программы к текущим условиям.

Например, в процессе отладки можно вывести список всех атрибутов объекта, чтобы убедиться в корректности его внутреннего состояния, а затем, используя доступ к `__dict__` или функции `setattr()` и `delattr()`, внести необходимые изменения без перезапуска приложения. Эта возможность особенно важна в системах, где структура объектов может изменяться в зависимости от поступающих данных или внешних параметров, позволяя не только выявлять ошибки, но и динамически обновлять конфигурацию.

Рассмотрим следующий пример, который демонстрирует, как с помощью интроспекции можно не только анализировать объект для целей отладки, но и создавать гибкие, настраиваемые структуры:

In [ ]:
```
class DynamicConfig:
    def __init__(self, **settings):
        for key, value in settings.items():
            setattr(self, key, value)

    def add_setting(self, key, value):
        setattr(self, key, value)

    def remove_setting(self, key):
        if hasattr(self, key):
            delattr(self, key)
        else:
            print(f"Настройка '{key}' отсутствует.")

    def list_settings(self):
        return self.__dict__

    def __str__(self):
        settings = "\n".join(f"  {k}: {v}" for k, v in self.__dict__.items())
        return f"DynamicConfig:\n{settings}"
```

Создаём объект конфигурации с начальными настройками:

In [ ]:
```
config = DynamicConfig(theme="dark", timeout=30, retries=3)
print("Начальное состояние объекта:")
print(config)
```

```
Начальное состояние объекта:
DynamicConfig:
  theme: dark
  timeout: 30
  retries: 3
```

Выводим список всех атрибутов с помощью `__dict__`:

In [ ]:
```
print("Состояние объекта через __dict__:")
print(config.__dict__)
```

```
Состояние объекта через __dict__:
{'theme': 'dark', 'timeout': 30, 'retries': 3}
```

Добавляем новую настройку и обновляем существующую:

In [ ]:
```
config.add_setting("debug", True)
config.add_setting("timeout", 60)
```

Удаляем настройку `retries`:

In [ ]:
```
config.remove_setting("retries")
print("Обновлённое состояние объекта:")
print(config)
```

```
Обновлённое состояние объекта:
DynamicConfig:
  theme: dark
  timeout: 60
  debug: True
```

В этом примере класс `DynamicConfig` инициализируется с набором настроек, переданных в виде именованных аргументов. Метод `list_settings()` возвращает внутренний словарь `__dict__`, позволяя увидеть текущие параметры, а метод `__str__` формирует удобное для чтения представление объекта. При помощи метода `add_setting()` новые атрибуты добавляются динамически, а `remove_setting()` проверяет наличие атрибута с помощью `hasattr()` и удаляет его через `delattr()`.

Видно, как интроспекция помогает в отладке — можно быстро оценить состояние объекта — и в динамическом управлении, когда программа должна адаптироваться к изменяющимся требованиям, например, подгружая новые конфигурационные параметры без остановки работы системы.

Специальные методы (магические методы)

Магические методы в Python открывают широкие возможности для настройки поведения объектов, позволяя переопределять стандартные операции и адаптировать их к специфике предметной области. Метод `__init__` задаёт начальное состояние объекта, обеспечивая его готовность к работе, а методы `__str__` и `__repr__` формируют строковое представление для пользователей и разработчиков. Перегрузка арифметических операторов, например, `__add__` или `__eq__`, позволяет объектам участвовать в вычислениях и сравнениях, приближая код к математической или логической интуиции. Методы контейнеров вроде `__getitem__` и `__iter__` превращают пользовательские классы в полноценные коллекции, поддерживающие индексацию и итерацию, а контекстные менеджеры с `__enter__` и `__exit__` гарантируют безопасное управление ресурсами. Эти механизмы делают язык выразительным и универсальным, предоставляя разработчикам средства для создания объектов с поведением, максимально соответствующим задачам приложения.

Методы инициализации и представления

Метод `__init__`

Метод `__init__` является основным инструментом для подготовки нового объекта к работе в Python. Этот метод, называемый конструктором, автоматически вызывается сразу после создания экземпляра класса и отвечает за установку начальных значений атрибутов, что обеспечивает корректное состояние объекта с самого начала его жизненного цикла.

При определении `__init__` разработчик указывает, какие параметры необходимы для создания объекта, а внутри метода можно выполнить любые действия, от простого присваивания значений атрибутам до проведения проверок или выполнения сложных вычислений.

Обязательным параметром всегда является `self` — ссылка на создаваемый объект, что позволяет методам класса работать с его внутренним состоянием. Таким образом, `__init__` задаёт фундамент, на котором строится дальнейшее поведение объекта, гарантируя, что каждый новый экземпляр будет инициализирован в согласованном и ожидаемом виде.

In [ ]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        if age < 0:
            raise ValueError("Возраст не может быть отрицательным.")
```

In [ ]:
```python
person = Person("Алексей", 32)
```

In [ ]:
```python
print(f"Имя: {person.name}, Возраст: {person.age}")
```

```
Имя: Алексей, Возраст: 32
```

В другом примере рассмотрим класс, моделирующий банковский счёт. Здесь `__init__` не только принимает обязательный параметр для владельца, но и задаёт баланс со значением по умолчанию, если он не передан, что позволяет создавать объекты с минимальным набором необходимых данных.

In [ ]:
```python
class BankAccount:
    def __init__(self, owner, balance=0.0):
        self.owner = owner
        self.balance = balance
```

In [ ]:
```python
account = BankAccount("Ирина")
print(f"Владелец: {account.owner}, Баланс: {account.balance} руб.")
```

```
Владелец: Ирина, Баланс: 0.0 руб.
```

В обоих примерах метод `__init__` обеспечивает, что после создания объекта все необходимые атрибуты будут установлены и доступны для последующего использования. Это помогает поддерживать целостность данных и упрощает дальнейшую работу

с объектами, поскольку они сразу находятся в предсказуемом и корректном состоянии.

Методы `__str__` и `__repr__`

Методы `__str__` и `__repr__` отвечают за строковое представление объектов, позволяя разработчику задать, как экземпляры класса будут отображаться в виде строк.

Метод `__str__` предназначен для конечного пользователя – он должен возвращать понятное и удобочитаемое описание объекта, которое используется, например, при вызове функции `print()`. В свою очередь, метод `__repr__` создаёт официальное строковое представление объекта, которое, по возможности, должно быть корректным выражением Python, позволяющим восстановить объект (или хотя бы давать исчерпывающую информацию для разработчика). Таким образом, `__str__` ориентирован на человекочитаемость, а `__repr__` – на точное описание для отладки и разработки.

Рассмотрим пример на основе класса `Person`. В этом примере метод `__str__` возвращает краткую информацию о человеке, удобную для отображения пользователю, а метод `__repr__` выдаёт детальную строку, содержащую имена и значения атрибутов, что может помочь разработчику при отладке кода.

In [ ]:
```
class Person:
    def __init__(self, name, age, occupation):
        self.name = name
        self.age = age
        self.occupation = occupation

    def __str__(self):
        return f"{self.name}, {self.age} лет"

    def __repr__(self):
        return (f"Person(name='{self.name}', "
                f"age={self.age}, "
                f"occupation='{self.occupation}')")
```

In [ ]:
```
person = Person("Алексей", 35, "Инженер")
```

In [ ]:
```
print("Вывод с __str__:", person)
```

```
Вывод с __str__: Алексей, 35 лет
```

In [ ]:
```
print("Вывод с __repr__:", repr(person))
```

```
Вывод с __repr__: Person(name='Алексей', age=35, occupation='Инженер')
```

В этом примере при вызове `print(person)` будет использован метод `__str__`, который выдаст строку вида `"Алексей, 30 лет"`, что удобно для восприятия пользователем. В то же время, функция `repr(person)` возвращает строку `"Person(name='Алексей', age=30, occupation='Инженер')"`, которая подробно описывает объект и позволяет понять его внутреннее состояние.

Такая разница позволяет использовать один и тот же класс как для представления данных в пользовательском интерфейсе, так и для глубокого анализа состояния объекта в процессе отладки или логирования. Это существенно упрощает процесс разработки и делает взаимодействие с объектами более гибким и информативным.

В следующем примере реализован класс `FinancialTransaction`, моделирующий финансовую транзакцию с такими параметрами, как уникальный идентификатор (`transaction_id`), дата (`date`), сумма (`amount`), валюта (`currency`) и описание операции (`description`).

Метод `__init__` инициализирует объект, задавая его начальное состояние, а метод `__str__` возвращает компактное и понятное представление транзакции для конечного пользователя, например, `"Transaction TX123: 1000.0 USD on 2023-08-01"`, в то время как метод `__repr__` предоставляет подробное строковое описание объекта, включающее имена всех атрибутов, что облегчает отладку и анализ состояния объекта.

In [ ]:
```
class FinancialTransaction:
    def __init__(self, transaction_id, date, amount, currency, description):
        self.transaction_id = transaction_id
        self.date = date
        self.amount = amount
        self.currency = currency
        self.description = description

    def __str__(self):
        return (f"Transaction {self.transaction_id}: "
                f"{self.amount} {self.currency} on {self.date}")

    def __repr__(self):
        return (f"FinancialTransaction("
                f"transaction_id={self.transaction_id!r}, "
                f"date={self.date!r}, "
                f"amount={self.amount!r}, "
```

```
                f"currency={self.currency!r}, "
                f"description={self.description!r})")
```

In [ ]:
```
transaction = FinancialTransaction("TX123",
                                   "2023-08-01",
                                   1000.0,
                                   "USD",
                                   "Salary payment"
                                   )
```

In [ ]:
```
print("Вывод с использованием __str__:", transaction, sep='\n')
```

```
Вывод с использованием __str__:
Transaction TX123: 1000.0 USD on 2023-08-01
```

In [ ]:
```
print("Вывод с использованием __repr__:", repr(transaction), sep='\n')
```

```
Вывод с использованием __repr__:
FinancialTransaction(transaction_id='TX123', date='2023-08-01', amount=1000.0, currency='USD', description='Sala
ry payment')
```

В этом примере метод `__init__` задаёт начальное состояние объекта, присваивая переданные значения соответствующим атрибутам, что гарантирует корректную инициализацию транзакции. Метод `__str__` формирует краткое описание, которое удобно выводить пользователю, а метод `__repr__` предоставляет детальное представление, полезное для отладки и логирования, позволяющее разработчику увидеть все параметры объекта.

Перегрузка операторов

Арифметические операторы

Метод `__add__`

Метод `__add__` позволяет переопределить поведение оператора сложения (`+`) для объектов пользовательского класса. Это означает, что при попытке сложить два объекта вашего класса Python вызовет метод `__add__`, который должен определить, как именно объединяются данные из этих объектов. Обычно в этом методе проводится проверка типа второго операнда и производится соответствующая операция, результатом которой создаётся новый объект. Такой метод используется для реализации семантики сложения, соответствующей предметной области, например, для векторной арифметики, работы с комплексными числами или объединения данных.

Рассмотрим пример класса `Vector`, представляющего вектор в пространстве. Метод `__init__` инициализирует вектор списком координат, а метод `__add__` реализует поэлементное сложение двух векторов, возвращая новый объект класса `Vector`. В примере предусмотрена проверка, что второй операнд является экземпляром класса `Vector` и имеет ту же размерность. В противном случае выбрасывается исключение, сигнализирующее о невозможности выполнить операцию.

In [ ]:
```
class Vector:
    def __init__(self, components):
         self.components = components

    def __add__(self, other):
        if isinstance(other, Vector):
            if len(self.components) != len(other.components):
                raise ValueError("Векторы должны иметь одинаковую размерность для сложения.")
            new_components = [a + b for a, b in zip(self.components, other.components)]
            return Vector(new_components)
        else:
            raise TypeError("Операция сложения возможна только между объектами Vector.")

    def __str__(self):
        return f"Vector({self.components})"
```

In [ ]:
```
v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
```

In [ ]:
```
v3 = v1 + v2
```

In [ ]:
```
print(f"Сумма векторов: {v3}")
```

```
Сумма векторов: Vector([5, 7, 9])
```

В этом примере при выполнении операции `v1 + v2` Python автоматически вызывает метод `__add__` объекта `v1`, передавая в качестве аргумента `v2`. Метод проверяет, что `v2` является объектом `Vector`, и затем поэлементно складывает соответствующие координаты, используя функцию `zip()`. Результатом является новый объект `Vector`, содержащий сумму координат, что позволяет работать с векторной арифметикой естественным способом. Если операнды имеют различную размерность или тип второго операнда не является `Vector`, генерируется соответствующее исключение, что помогает избежать ошибок и обеспечивает корректность вычислений.

Метод `__sub__`

Метод `__sub__` позволяет переопределить оператор вычитания (`-`) для объектов пользовательского класса. Когда мы пишем выражение `v1 - v2` , Python автоматически вызывает метод `__sub__` первого объекта, передавая второй объект в качестве аргумента.

В реализации этого метода обычно проводится проверка, что операнд справа является объектом нужного класса, а также что оба объекта имеют совместимые структуры (например, одинаковую размерность в случае векторной арифметики). После этого производится поэлементное вычитание соответствующих компонент, и результатом является новый объект, представляющий разность исходных значений. Если операнды не соответствуют ожидаемым требованиям, метод генерирует исключения, например, `TypeError` или `ValueError` .

Ниже приведён пример класса `Vector` , в котором метод `__sub__` реализует операцию вычитания для векторов:

In [ ]:
```
class Vector:
    def __init__(self, components):
        self.components = components

    def __sub__(self, other):
        if not isinstance(other, Vector):
            raise TypeError("Операция вычитания возможна только между объектами Vector.")
        if len(self.components) != len(other.components):
            raise ValueError("Векторы должны иметь одинаковую размерность для вычитания.")
        new_components = [a - b for a, b in zip(self.components, other.components)]
        return Vector(new_components)

    def __str__(self):
        return f"Vector({self.components})"
```

In [ ]:
```
v1 = Vector([10, 20, 30])
v2 = Vector([1, 2, 3])
```

In [ ]:
```
v3 = v1 - v2
```

In [ ]:
```
print(f"Результат вычитания: {v3}")
```

```
Результат вычитания: Vector([9, 18, 27])
```

В этом примере объект `v1` создаётся с координатами `[10, 20, 30]` , а объект `v2` — с координатами `[1, 2, 3]` . При выполнении операции `v1 - v2` Python автоматически вызывает метод `__sub__` объекта `v1` , который сначала проверяет, что `v2` является экземпляром класса `Vector` , затем убеждается, что длина списков координат обоих векторов совпадает. После этого происходит поэлементное вычитание: 10 минус 1, 20 минус 2 и 30 минус 3, что приводит к новому списку `[9, 18, 27]` . Результатом операции является новый объект `Vector` , который затем выводится с помощью метода `__str__` . Если же попытаться выполнить вычитание с объектом другого типа или с векторами разной размерности, метод генерирует соответствующие исключения, предотвращая некорректное выполнение операции.

Другие арифметические методы

Метод `__mul__` позволяет переопределить оператор умножения для объектов, а аналогичные арифметические методы расширяют эту возможность, реализуя операции умножения, деления, целочисленного деления, вычисления остатка и возведения в степень. Другими словами, помимо сложения и вычитания, можно задать, как объекты нашего класса будут реагировать на другие арифметические операторы, что существенно повышает гибкость и выразительность кода.

Ниже приведён пример расширенного класса `Vector` , который, помимо методов `__add__` и `__sub__` , включает реализацию следующих специальных методов: `__mul__` , `__truediv__` , `__floordiv__` , `__mod__` и `__pow__` .

В реализации метода `__mul__` предусмотрена проверка типа операнда: если он является числом, выполняется скалярное умножение, возвращающее новый вектор, а если это другой вектор и его размерность совпадает с размерностью первого, то выполняется скалярное произведение с возвращением числового результата.

Методы `__truediv__` , `__floordiv__` и `__mod__` определяют поэлементное деление, целочисленное деление и вычисление остатка соответственно, при этом проверка гарантирует, что операция производится только с числовым операндом, а метод `__pow__` осуществляет возведение каждого компонента вектора в заданную степень.

Такое переопределение позволяет использовать операторы `*` , `/` , `//` , `%` и `**` непосредственно с объектами класса, делая код, работающий с векторной арифметикой, более интуитивно понятным и близким к математической нотации.

In [ ]:
```
class Vector:
    def __init__(self, components):
         self.components = components

    def __add__(self, other):
        if isinstance(other, Vector):
            if len(self.components) != len(other.components):
```

```
                raise ValueError("Векторы должны иметь одинаковую размерность для сложения.")
            new_components = [a + b for a, b in zip(self.components, other.components)]
            return Vector(new_components)
        else:
            raise TypeError("Операция сложения возможна только между объектами Vector.")

    def __sub__(self, other):
        if not isinstance(other, Vector):
            raise TypeError("Операция вычитания возможна только между объектами Vector.")
        if len(self.components) != len(other.components):
            raise ValueError("Векторы должны иметь одинаковую размерность для вычитания.")
        new_components = [a - b for a, b in zip(self.components, other.components)]
        return Vector(new_components)

    def __mul__(self, other):
        if isinstance(other, (int, float)):
            new_components = [a * other for a in self.components]
            return Vector(new_components)
        elif isinstance(other, Vector):
            if len(self.components) != len(other.components):
                raise ValueError("Векторы должны иметь одинаковую размерность для умножения.")
            return sum(a * b for a, b in zip(self.components, other.components))
        else:
            raise TypeError("Умножение возможно только с числом или другим объектом Vector.")

    def __truediv__(self, other):
        if isinstance(other, (int, float)):
            if other == 0:
                raise ZeroDivisionError("Деление на ноль невозможно.")
            new_components = [a / other for a in self.components]
            return Vector(new_components)
        else:
            raise TypeError("Деление возможно только на число.")

    def __floordiv__(self, other):
        if isinstance(other, (int, float)):
            if other == 0:
                raise ZeroDivisionError("Деление на ноль невозможно.")
            new_components = [a // other for a in self.components]
            return Vector(new_components)
        else:
            raise TypeError("Целочисленное деление возможно только на число.")

    def __mod__(self, other):
        if isinstance(other, (int, float)):
            new_components = [a % other for a in self.components]
            return Vector(new_components)
        else:
            raise TypeError("Остаток от деления возможен только с числом.")

    def __pow__(self, exponent):
        if isinstance(exponent, (int, float)):
            new_components = [a ** exponent for a in self.components]
            return Vector(new_components)
        else:
            raise TypeError("Показатель степени должен быть числом.")

    def __str__(self):
        return f"Vector({self.components})"
```

Рассмотрим использование расширенного класса `Vector` для демонстрации работы этих методов:

In [ ]:
```
v1 = Vector([10, 20, 30])
v2 = Vector([1, 2, 3])
```

In [ ]:
```
v_mul_scalar = v1 * 2
print(f"Умножение на скаляр: {v_mul_scalar}")
```

```
Умножение на скаляр: Vector([20, 40, 60])
```

In [ ]:
```
dot_product = v1 * v2
print(f"Скалярное произведение: {dot_product}")
```

```
Скалярное произведение: 140
```

In [ ]:
```
v_div = v1 / 2
print(f"Деление на число: {v_div}")
```

```
Деление на число: Vector([5.0, 10.0, 15.0])
```

In [ ]:
```
v_floordiv = v1 // 3
print(f"Целочисленное деление: {v_floordiv}")
```

```
Целочисленное деление: Vector([3, 6, 10])
```

In [ ]:
```
v_mod = v1 % 7
print(f"Остаток от деления: {v_mod}")
```

```
Остаток от деления: Vector([3, 6, 2])
```

In [ ]:
```
v_pow = v1 ** 2
print(f"Возведение в степень: {v_pow}")
```

```
Возведение в степень: Vector([100, 400, 900])
```

В этом примере метод `__mul__` обрабатывает два типа операций: умножение вектора на число и скалярное произведение двух векторов, возвращая либо новый объект `Vector`, либо число. Методы `__truediv__`, `__floordiv__`, `__mod__` выполняют деление каждого компонента вектора на заданное число, причем целочисленное деление использует оператор `//`, а операция остатка – `%`. Метод `__pow__` реализует возведение каждого элемента вектора в заданную степень.

Таким образом, благодаря переопределению этих методов, объекты класса `Vector` могут участвовать в арифметических операциях, аналогичных стандартным числовым типам, что позволяет писать код, максимально приближенный к математической записи, и упрощает реализацию алгоритмов, работающих с векторными данными.

Операторы сравнения

Метод `__eq__`

Метод `__eq__` позволяет задавать поведение оператора равенства (`==`) для объектов пользовательского класса. При реализации определяется, на каких основаниях два объекта считаются равными — обычно сравниваются ключевые атрибуты или внутреннее состояние. Вызов `obj1 == obj2` приводит к автоматическому выполнению `obj1.__eq__(obj2)`. Возвращение `True` означает равенство объектов, `False` — различие. Рекомендуется проверять тип второго операнда и возвращать `NotImplemented` при несоответствии ожидаемому типу, что обеспечивает корректную обработку сравнения или вызов симметричного метода другого объекта. Переопределение `__eq__` часто применяется для сравнения сложных структур данных, где равенство определяется содержимым, а не ссылками. Это важно для сортировки, фильтрации или проверки уникальности элементов в коллекциях.

Пример реализации показан на классе `Polynomial`, моделирующем многочлен с коэффициентами в виде списка чисел. Два многочлена считаются равными при совпадении нормализованных списков коэффициентов, причём завершающие нули не влияют на результат. Нормализация выполняется вспомогательным методом `_normalize()`, убирающим ненужные нули в конце списка. При несовпадении типа второго операнда с классом `Polynomial` возвращается `NotImplemented`.

In [ ]:
```
class Polynomial:
    def __init__(self, coefficients):
      self.coefficients = coefficients[:]
      self._normalize()

    def _normalize(self):
        while len(self.coefficients) > 1 and self.coefficients[-1] == 0:
            self.coefficients.pop()

    def __eq__(self, other):
        if not isinstance(other, Polynomial):
            return NotImplemented
        return self.coefficients == other.coefficients

    def __str__(self):
        terms = []
        for power, coeff in enumerate(self.coefficients):
            if coeff == 0:
                continue
            if power == 0:
                terms.append(f"{coeff}")
            elif power == 1:
                terms.append(f"{coeff}*x")
            else:
                terms.append(f"{coeff}*x^{power}")
        return " + ".join(terms) if terms else "0"
```

При создании объекта методом `__init__` принимается список коэффициентов, копируется в атрибут `self.coefficients`, а затем вызывается метод `_normalize`. Этот метод удаляет завершающие нули из списка, пока его длина превышает единицу и последний элемент равен нулю. Метод `__eq__` определяет равенство двух многочленов: если второй операнд не относится к классу `Polynomial`, возвращается `NotImplemented`; в противном случае сравниваются списки коэффициентов. Для удобного отображения многочлена метод `__str__` создаёт строку, где ненулевые коэффициенты преобразуются в слагаемые: для степени ноль — просто число, для степени один — запись вида `coeff*x`, для остальных — `coeff*x^power`. Слагаемые соединяются знаком плюс, а если слагаемых нет, возвращается `"0"`.

Пример использования:

In [ ]:
```
p1 = Polynomial([1, 2, 0, 0])
```

```
p2 = Polynomial([1, 2])
p3 = Polynomial([1, 2, 3])
```

In [ ]:
```
print("p1:", p1)
print("p2:", p2)
print("p3:", p3)
```

```
p1: 1 + 2*x
p2: 1 + 2*x
p3: 1 + 2*x + 3*x^2
```

In [ ]:
```
print("p1 == p2:", p1 == p2)
```

```
p1 == p2: True
```

In [ ]:
```
print("p1 == p3:", p1 == p3)
```

```
p1 == p3: False
```

Переопределение оператора равенства в данном классе позволяет точно определить, когда два многочлена представляют одно и то же математическое выражение, даже если они заданы разными списками коэффициентов.

Методы `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__`

Методы `__ne__`, `__lt__`, `__gt__`, `__le__` и `__ge__` переопределяют стандартные операторы сравнения для объектов пользовательского класса. Они позволяют определить, когда один объект меньше, больше или не равен другому, основываясь на выбранных атрибутах или логике сравнения. Реализация этих методов полезна, когда объекты необходимо сортировать или сравнивать для поиска, фильтрации и других операций. При этом важно учитывать, что сравнение должно корректно обрабатывать ситуацию, когда второй операнд имеет неверный тип, для чего обычно возвращается значение `NotImplemented`.

Рассмотрим пример класса `Employee`, в котором объекты сравниваются сначала по возрасту, а если возраст равен, то по заработной плате. Таким образом, сотрудник считается «меньше», если его возраст ниже, или при равных возрастах – если его зарплата меньше.

In [ ]:
```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    def __eq__(self, other):
        if not isinstance(other, Employee):
            return NotImplemented
        return (self.age, self.salary) == (other.age, other.salary)

    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return NotImplemented
        return not result

    def __lt__(self, other):
        if not isinstance(other, Employee):
            return NotImplemented
        return (self.age, self.salary) < (other.age, other.salary)

    def __le__(self, other):
        if not isinstance(other, Employee):
            return NotImplemented
        return self < other or self == other

    def __gt__(self, other):
        if not isinstance(other, Employee):
            return NotImplemented
        return not (self <= other)

    def __ge__(self, other):
        if not isinstance(other, Employee):
            return NotImplemented
        return not (self < other)

    def __str__(self):
        return f"{self.name}, возраст: {self.age}, зарплата: {self.salary}"
```

Здесь метод `__eq__` сравнивает сотрудников по кортежу `(age, salary)`, что означает, что два объекта считаются равными, если у них одинаковый возраст и зарплата. Метод `__ne__` возвращает логическое отрицание результата `__eq__`. Оператор «меньше» (`__lt__`) сначала сравнивает возраст, а при равном возрасте — зарплату, что позволяет упорядочить объекты по двум критериям одновременно. Остальные методы (`__le__`, `__gt__`, `__ge__`) реализованы через уже определённые операторы для избежания дублирования логики.

Пример использования класса `Employee`:

In [ ]:
```
emp1 = Employee("Иван", 30, 50000)
emp2 = Employee("Мария", 35, 60000)
emp3 = Employee("Петр", 30, 55000)
emp4 = Employee("Алексей", 30, 50000)
```

In [ ]:
```
print(f"{emp1} == {emp4}:", emp1 == emp4)
print(f"{emp1} != {emp3}:", emp1 != emp3)
print(f"{emp1} < {emp2}:", emp1 < emp2)
print(f"{emp1} < {emp3}:", emp1 < emp3)
print(f"{emp2} > {emp3}:", emp2 > emp3)
print(f"{emp1} <= {emp4}:", emp1 <= emp4)
print(f"{emp3} >= {emp1}:", emp3 >= emp1)
```

```
Иван, возраст: 30, зарплата: 50000 == Алексей, возраст: 30, зарплата: 50000: True
Иван, возраст: 30, зарплата: 50000 != Петр, возраст: 30, зарплата: 55000: True
Иван, возраст: 30, зарплата: 50000 < Мария, возраст: 35, зарплата: 60000: True
Иван, возраст: 30, зарплата: 50000 < Петр, возраст: 30, зарплата: 55000: True
Мария, возраст: 35, зарплата: 60000 > Петр, возраст: 30, зарплата: 55000: True
Иван, возраст: 30, зарплата: 50000 <= Алексей, возраст: 30, зарплата: 50000: True
Петр, возраст: 30, зарплата: 55000 >= Иван, возраст: 30, зарплата: 50000: True
```

В этом примере объекты класса `Employee` сравниваются с использованием стандартных операторов (`==` , `!=` , `<` , `>` , `<=` , `>=`), которые переопределены методами `__eq__` , `__ne__` , `__lt__` , `__le__` , `__gt__` и `__ge__` . Это позволяет проводить комплексное сравнение по нескольким критериям, делая объекты пригодными для сортировки и других операций, где важна упорядоченность.

Методы контейнеров и последовательностей

Индексирование и срезы

Метод `__getitem__`

Метод `__getitem__` отвечает за получение элемента или группы элементов из объекта с помощью квадратных скобок, что делает пользовательский класс похожим на встроенные коллекции (например, списки или кортежи). При его переопределении можно обеспечить поддержку как одиночного индексирования, так и работы с срезами, что позволяет получать подмножество данных. Такой способ удобно применять при создании моделей реальных данных, когда нужно работать с отдельными элементами, например, в медиатеках, плейлистах или каталогах товаров.

Рассмотрим класс `Playlist` , который представляет список музыкальных треков. Каждый трек описывается словарём с информацией о названии, исполнителе и длительности. Класс `Playlist` сохраняет список треков и переопределяет метод `__getitem__` , чтобы при обращении через квадратные скобки возвращался либо конкретный трек, либо новый объект `Playlist` , содержащий выбранные треки. Это позволяет гибко управлять данными в плейлисте: можно легко получить отдельную композицию или выделить подсписок треков для создания мини-плейлиста.

In [ ]:
```
class Playlist:
    def __init__(self, tracks):
        self.tracks = tracks[:]

    def __getitem__(self, index):
        if isinstance(index, slice):
            return Playlist(self.tracks[index])
        else:
            return self.tracks[index]

    def __str__(self):
        track_list = ", ".join(f"'{track['title']}' by {track['artist']}" for track in self.tracks)
        return f"Playlist: {len(self.tracks)} tracks - {track_list}"
```

In [ ]:
```
tracks = [
    {"title": "Imagine", "artist": "John Lennon", "duration": 183},
    {"title": "Bohemian Rhapsody", "artist": "Queen", "duration": 354},
    {"title": "Hotel California", "artist": "Eagles", "duration": 391},
    {"title": "Stairway to Heaven", "artist": "Led Zeppelin", "duration": 482},
    {"title": "Hey Jude", "artist": "The Beatles", "duration": 431}
    ]
```

In [ ]:
```
playlist = Playlist(tracks)
```

In [ ]:
```
print(f"Трек с индексом 2: {playlist[2]}")
```

```
Трек с индексом 2: {'title': 'Hotel California', 'artist': 'Eagles', 'duration': 391}
```

In [ ]:
```
print("Подплейлист с индексами 1-3:")
print(playlist[1:4])
```

```
Подплейлист с индексами 1-3:
Playlist: 3 tracks - 'Bohemian Rhapsody' by Queen, 'Hotel California' by Eagles, 'Stairway to Heaven' by Led Zep
pelin
```

In [ ]:
```python
print("Полный плейлист:")
print(playlist)
```

```
Полный плейлист:
Playlist: 5 tracks - 'Imagine' by John Lennon, 'Bohemian Rhapsody' by Queen, 'Hotel California' by Eagles, 'Stai
rway to Heaven' by Led Zeppelin, 'Hey Jude' by The Beatles
```

В этом примере класс `Playlist` демонстрирует, как метод `__getitem__` позволяет получать доступ к данным внутри объекта: если передается одиночное число, возвращается конкретный трек (словарь с информацией о композиции), а если передается срез, создается новый объект `Playlist`, содержащий подмножество треков. Это делает работу с плейлистом удобной и интуитивно понятной, позволяя динамически извлекать как отдельные элементы, так и группы элементов.

Метод `__setitem__`

Метод `__setitem__` переопределяет возможность установки (обновления) элемента в объекте с помощью квадратных скобок, то есть при операции `obj[index] = value` Python автоматически вызывает именно этот метод. Это позволяет объектам, которые моделируют коллекции или последовательности, поддерживать динамическое изменение своего содержимого.

При реализации метода `__setitem__` можно добавить валидацию новых данных, проверку типа, диапазона индексов и прочие условия, что обеспечивает корректное обновление внутреннего состояния объекта. Такой механизм широко применяется в реальных приложениях, когда необходимо обновлять информацию в структуре данных, например, изменять детали товара в корзине покупок, корректировать статус заказа или обновлять информацию о записи в каталоге.

Класс `ShoppingCart` моделирует корзину покупок. Корзина хранит список товаров, где каждый товар представлен словарём с ключами `name` и `price`. Метод `__setitem__` позволяет обновлять информацию о товаре по индексу, при этом происходит проверка, что новый элемент является корректным словарём с обязательными ключами. Благодаря этому данные в корзине сохраняют согласованность и точность.

In [ ]:
```python
class ShoppingCart:
    def __init__(self, items):
        self.items = items[:]

    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        if not isinstance(value, dict):
            raise TypeError("Товар должен быть представлен словарём.")
        if 'name' not in value or 'price' not in value:
            raise ValueError("Словарь товара должен содержать ключи 'name' и 'price'.")
        self.items[index] = value

    def __str__(self):
        item_strs = [f"{i + 1}. {item['name']} (₽{item['price']})" for i, item in enumerate(self.items)]
        return "Shopping Cart:\n" + "\n".join(item_strs)
```

Пример использования класса `ShoppingCart`:

In [ ]:
```python
initial_items = [
    {"name": "Ноутбук", "price": 120000},
    {"name": "Смартфон", "price": 80000},
    {"name": "Наушники", "price": 15000}
    ]
```

In [ ]:
```python
cart = ShoppingCart(initial_items)
```

In [ ]:
```python
print("Начальное состояние корзины:")
print(cart)
```

```
Начальное состояние корзины:
Shopping Cart:
1. Ноутбук (₽120000)
2. Смартфон (₽80000)
3. Наушники (₽15000)
```

In [ ]:
```python
cart[1] = {"name": "Смартфон", "price": 75000}
```

In [ ]:
```python
print("После обновления второго товара:")
print(cart)
```

```
После обновления второго товара:
Shopping Cart:
1. Ноутбук (₽120000)
2. Смартфон (₽75000)
3. Наушники (₽15000)
```

In [ ]:
```
try:
    cart[0] = {"title": "Ноутбук", "cost": 110000}
except ValueError as e:
    print("Ошибка при обновлении товара:", e)
```

```
Ошибка при обновлении товара: Словарь товара должен содержать ключи 'name' и 'price'.
```

В этом примере класс `ShoppingCart` обеспечивает возможность изменять содержимое корзины через синтаксис индексирования. Метод `__setitem__` проверяет, что новый элемент является словарём с нужными ключами, и только после этого заменяет старый товар. Если передается некорректное значение (например, словарь без обязательных ключей), генерируется исключение, что помогает предотвратить ошибки в данных.

Метод `__delitem__`

Метод `__delitem__` переопределяет операцию удаления элемента из объекта с помощью оператора `del` и квадратных скобок, например, при вызове `del obj[index]` Python автоматически обращается к методу `obj.__delitem__(index)`. Это необходимо для классов, которые моделируют коллекции или последовательности, поскольку позволяет удалять элементы из внутреннего хранилища данных по индексу. При реализации метода можно добавить дополнительные проверки, например, корректности индекса или выполнение сопутствующих действий (например, обновление внутреннего состояния), гарантируя, что структура объекта остаётся согласованной после удаления элемента.

Расширим класс `ShoppingCart`. Метод `__delitem__` реализован для удаления товара по указанному индексу, что позволяет использовать команду `del cart[index]` для динамического изменения содержимого корзины. При этом объект продолжает работать как обычная коллекция: после удаления элемента метод `__str__` выводит обновлённое состояние корзины.

In [ ]:
```
class ShoppingCart:
    def __init__(self, items):
        self.items = items[:]

    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        if not isinstance(value, dict):
            raise TypeError("Товар должен быть представлен словарём.")
        if 'name' not in value or 'price' not in value:
            raise ValueError("Словарь товара должен содержать ключи 'name' и 'price'.")
        self.items[index] = value

    def __delitem__(self, index):
        del self.items[index]

    def __str__(self):
        item_strs = [f"{i + 1}. {item['name']} (₽{item['price']})" for i, item in enumerate(self.items)]
        return "Shopping Cart:\n" + "\n".join(item_strs)
```

In [ ]:
```
initial_items = [
    {"name": "Ноутбук", "price": 120000},
    {"name": "Смартфон", "price": 80000},
    {"name": "Наушники", "price": 15000}
    ]
```

In [ ]:
```
cart = ShoppingCart(initial_items)
```

In [ ]:
```
print("Начальное состояние корзины:")
print(cart)
```

```
Начальное состояние корзины:
Shopping Cart:
1. Ноутбук (₽120000)
2. Смартфон (₽80000)
3. Наушники (₽15000)
```

In [ ]:
```
del cart[1]
```

In [ ]:
```
print("Состояние корзины после удаления товара с индексом 1:")
print(cart)
```

```
Состояние корзины после удаления товара с индексом 1:
Shopping Cart:
1. Ноутбук (₽120000)
2. Наушники (₽15000)
```

В данном примере класс `ShoppingCart` инициализируется списком товаров, каждый из которых представлен словарём с информацией о названии и цене. Метод `__delitem__` позволяет удалить товар по индексу, что реализовано через вызов `del cart[1]`. После удаления внутренний список `items` обновляется, и метод `__str__` возвращает актуальное состояние

корзины, отображая оставшиеся товары. Этот механизм демонстрирует, как с помощью переопределения метода `__delitem__` можно сделать пользовательский класс динамичным и похожим на встроенные коллекции, поддерживая интуитивно понятный синтаксис удаления элементов.

Итераторы

Метод `__iter__`

Метод `__iter__` — это ключ к тому, чтобы объект стал итерабельным, то есть чтобы можно было использовать его в циклах или передавать в функции, ожидающие коллекцию. При вызове `iter(obj)` Python ищет этот метод и использует его для получения итератора, который по очереди возвращает элементы объекта через вызовы `next()`. Обычно реализация метода `__iter__` подразумевает возврат генератора или объекта, реализующего протокол итератора. Это важно для классов, моделирующих динамические наборы данных, когда требуется не просто хранить информацию, но и фильтровать или преобразовывать её на лету.

Например, если в классе, описывающем очередь заказов в интернет-магазине, необходимо выдавать только те заказы, которые находятся в ожидании обработки, метод `__iter__` может вернуть генератор, который отфильтровывает список заказов по статусу. При этом можно добавить дополнительные методы для управления очередью: добавление нового заказа, обновление статуса, отмена заказа и т.д. Такой подход упрощает работу с данными, скрывая детали их обхода внутри объекта, а внешний код может получать актуальные элементы через обычную итерацию.

Рассмотрим класс `OrderQueue`, моделирующий очередь заказов. Класс инициализируется списком заказов, где каждый заказ представлен словарём с ключами `order_id`, `customer`, `status` и `amount`. Помимо метода `__iter__`, который возвращает только заказы со статусом `"pending"`, класс включает методы для добавления заказа (`add_order`), обновления статуса заказа (`update_order_status`) и отмены заказа (`cancel_order`). Примеры использования демонстрируют, как можно динамически добавлять заказы, изменять их статус, и затем итерироваться по очереди, получая только актуальные заказы для обработки.

In [ ]:
```
class OrderQueue:
    def __init__(self, orders=None):
        self.orders = orders[:] if orders is not None else []

    def add_order(self, order):
        required_keys = {'order_id', 'customer', 'status', 'amount'}
        if not isinstance(order, dict) or not required_keys.issubset(order.keys()):
            raise ValueError("Заказ должен быть словарем с ключами 'order_id', 'customer', 'status' и 'amount'.
        self.orders.append(order)

    def update_order_status(self, order_id, new_status):
        for order in self.orders:
            if order.get("order_id") == order_id:
                order["status"] = new_status
                return True
        return False

    def cancel_order(self, order_id):
        return self.update_order_status(order_id, "cancelled")

    def __iter__(self):
        return (order for order in self.orders if order.get("status") == "pending")

    def __str__(self):
        lines = [f"OrderQueue: {len(self.orders)} заказов"]
        for order in self.orders:
            lines.append(
                f"  {order['order_id']}: {order['customer']} - {order['status']} (₽{order['amount']})"
            )
        return "\n".join(lines)
```

Пример использования:

In [ ]:
```
initial_orders = [
    {"order_id": "A001", "customer": "Иванов", "status": "pending", "amount": 2500.0},
    {"order_id": "A002", "customer": "Петров", "status": "shipped", "amount": 3000.0},
    {"order_id": "A003", "customer": "Сидоров", "status": "pending", "amount": 1500.0},
    ]
```

In [ ]:
```
queue = OrderQueue(initial_orders)
```

In [ ]:
```
print("Исходное состояние очереди заказов:")
print(queue)
```

```
Исходное состояние очереди заказов:
OrderQueue: 3 заказов
  A001: Иванов - pending (₽2500.0)
  A002: Петров - shipped (₽3000.0)
  A003: Сидоров - pending (₽1500.0)
```

In [ ]:
```python
queue.add_order({"order_id": "A004", "customer": "Кузнецов", "status": "pending", "amount": 4000.0})
```

In [ ]:
```python
queue.update_order_status("A002", "pending")
```

Out[ ]:
```
True
```

In [ ]:
```python
queue.cancel_order("A003")
```

Out[ ]:
```
True
```

In [ ]:
```python
print("Обновленное состояние очереди заказов:")
print(queue)
```

```
Обновленное состояние очереди заказов:
OrderQueue: 4 заказов
  A001: Иванов - pending (₽2500.0)
  A002: Петров - pending (₽3000.0)
  A003: Сидоров - cancelled (₽1500.0)
  A004: Кузнецов - pending (₽4000.0)
```

In [ ]:
```python
print("Заказы, ожидающие обработки:")
for order in queue:
    print(f"Заказ {order['order_id']} от {order['customer']} на сумму ₽{order['amount']} (статус: {order['statu
```

```
Заказы, ожидающие обработки:
Заказ A001 от Иванов на сумму ₽2500.0 (статус: pending)
Заказ A002 от Петров на сумму ₽3000.0 (статус: pending)
Заказ A004 от Кузнецов на сумму ₽4000.0 (статус: pending)
```

Здесь метод `__iter__` возвращает генератор, фильтрующий заказы по статусу `"pending"`, благодаря чему цикл `for` обрабатывает только актуальные заказы для выполнения. Это демонстрирует, как интроспекция внутренней структуры (в данном случае, динамическое формирование итератора на основе текущего состояния объекта) позволяет создавать гибкие и настраиваемые классы, способные адаптироваться к изменяющимся требованиям приложения.

Метод `__next__`

Метод `__next__` является неотъемлемой частью протокола итератора в Python. Когда объект является итератором, вызов функции `next()` приводит к автоматическому вызову его метода `__next__`, который должен вернуть следующий элемент последовательности. Если элементов больше нет, метод `__next__` должен вызвать исключение `StopIteration`, чтобы сообщить циклу (или другой конструкции, работающей с итераторами), что итерация завершена.

В реальных приложениях этот метод может использоваться для последовательного извлечения данных из различных источников: очередей задач, логов, результатов запросов или других динамических наборов данных. Рассмотрим класс `TaskQueueIterator`, который реализует итератор для списка задач. Каждый вызов метода `__next__` возвращает следующую задачу из внутреннего списка, а по достижении конца списка выбрасывается исключение `StopIteration`, что позволяет корректно завершить итерацию. При этом класс может включать дополнительные методы для управления очередью задач, но именно реализация `__next__` отвечает за выдачу следующего элемента при обходе в цикле.

In [ ]:
```python
class TaskQueueIterator:
    def __init__(self, tasks):
        self.tasks = tasks[:]
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.tasks):
            task = self.tasks[self.index]
            self.index += 1
            return task
        else:
            raise StopIteration
```

In [ ]:
```python
tasks = [
    "Обработать заказ №1001",
    "Отправить уведомление клиенту",
    "Сформировать отчет по продажам",
    "Обновить данные о складе",
    ]
```

In [ ]:
```python
task_iterator = TaskQueueIterator(tasks)
```

In [ ]:
```
print("Начало обработки задач:")
for task in task_iterator:
    print("Выполняется:", task)
```

```
Начало обработки задач:
Выполняется: Обработать заказ №1001
Выполняется: Отправить уведомление клиенту
Выполняется: Сформировать отчет по продажам
Выполняется: Обновить данные о складе
```

In [ ]:
```
print("Повторная попытка обработки:")
for task in task_iterator:
    print("Выполняется:", task)
```

```
Повторная попытка обработки:
```

Класс `TaskQueueIterator` инициализируется списком задач и задает начальное значение индекса равным 0. Метод `__iter__` возвращает сам объект, делая его итератором, а метод `__next__` проверяет, не превышает ли текущий индекс длину списка задач. Если есть следующий элемент, он возвращается, и индекс увеличивается. Когда все задачи пройдены, метод вызывает исключение `StopIteration`, что сигнализирует о завершении итерации.

Данный пример демонстрирует, как цикл `for` автоматически вызывает метод `__next__` для получения каждой задачи, и как после полного обхода итерация не возвращает элементов при повторном запуске.

Рассмотрим, как реализовать собственные итераторы и контролировать процесс обхода последовательностей.

Класс `Range` демонстрирует, как можно создать пользовательский итератор, имитирующий поведение встроенной функции `range()`. Он генерирует последовательность чисел в заданном диапазоне, поддерживая возможность итерации в цикле. При этом в конструкторе можно задавать начальное значение, конечное значение и шаг (по умолчанию шаг равен 1). Метод `__iter__` возвращает сам объект, а метод `__next__` отвечает за выдачу следующего элемента последовательности. Когда текущее значение достигает или превосходит конечное значение (при положительном шаге) или становится меньше конечного (при отрицательном шаге), метод `__next__` генерирует исключение `StopIteration`, сигнализируя о завершении итерации.

In [ ]:
```
class Range:
    def __init__(self, start, stop=None, step=1):
        if stop is None:
            self.start = 0
            self.stop = start
        else:
            self.start = start
            self.stop = stop

        if step == 0:
            raise ValueError("Шаг не может быть равен 0.")
        self.step = step

        self.current = self.start

    def __iter__(self):
        return self

    def __next__(self):
        if (self.step > 0 and self.current >= self.stop) or (self.step < 0 and self.current <= self.stop):
            raise StopIteration

        result = self.current
        self.current += self.step
        return result
```

Пример использования класса Range:

In [ ]:
```
print("Пример range(5):")
for number in Range(5):
    print(number, end=" ")
```

```
Пример range(5):
0 1 2 3 4
```

In [ ]:
```
print("Пример range(2, 10, 2):")
for number in Range(2, 10, 2):
    print(number, end=" ")
```

```
Пример range(2, 10, 2):
2 4 6 8
```

In [ ]:
```
print("Пример range(10, 2, -2):")
for number in Range(10, 2, -2):
    print(number, end=" ")
```

```
Пример range(10, 2, -2):
10 8 6 4
```

В этом примере класс `Range` реализует все необходимые методы для итерации. Конструктор принимает либо один аргумент (тогда последовательность начинается с 0 и заканчивается перед заданным значением), либо два аргумента (начальное и конечное значения). Также предусмотрена возможность задания шага, который может быть как положительным, так и отрицательным; нулевой шаг приводит к ошибке. Метод `__iter__` возвращает сам объект, позволяя использовать его в циклах, а метод `__next__` возвращает текущее значение и обновляет его на величину шага. Когда условие продолжения итерации нарушается, генерируется исключение `StopIteration`, корректно завершая обход последовательности.

Длина и проверка на вхождение

Метод `__len__`

Метод `__len__` переопределяет поведение встроенной функции `len()`, возвращая количество элементов, содержащихся в объекте. Это требуется для классов, моделирующих коллекции или контейнеры, так как именно с помощью `__len__` можно обеспечить интуитивно понятное измерение «размера» объекта.

При вызове `len(obj)` Python автоматически обращается к методу `obj.__len__()`, ожидая получить целое число, которое отражает число элементов, находящихся в объекте. Реализация метода `__len__` может включать дополнительную логику, например, фильтрацию элементов или динамический подсчёт, что делает класс гибким и адаптивным к изменениям внутреннего состояния.

Создадим класс `Bag`, моделирующий мешок, который хранит уникальные элементы. В данном случае конструктор принимает необязательный список элементов и сохраняет только уникальные значения, исключая повторения. Метод `__len__` возвращает количество уникальных элементов, что позволяет использовать функцию `len(bag)` для получения «размера» мешка.

In [ ]:
```python
class Bag:
    def __init__(self, items=None):
        if items is None:
            self.items = []
        else:
            seen = set()
            self.items = [item for item in items if not (item in seen or seen.add(item))]

    def add(self, item):
        if item not in self.items:
            self.items.append(item)

    def remove(self, item):
        if item in self.items:
            self.items.remove(item)

    def __len__(self):
        return len(self.items)

    def __str__(self):
        return f"Bag: {self.items}"
```

In [ ]:
```python
planets = [
    "Меркурий", "Венера", "Земля", "Марс",
    "Земля", "Юпитер", "Сатурн", "Уран",
    "Нептун", "Марс"
    ]
```

In [ ]:
```python
bag = Bag(planets)
```

In [ ]:
```python
print(f"Начальное состояние мешка:\n{bag}")
```

```
Начальное состояние мешка:
Bag: ['Меркурий', 'Венера', 'Земля', 'Марс', 'Юпитер', 'Сатурн', 'Уран', 'Нептун']
```

In [ ]:
```python
bag.add("Плутон")
bag.add("Венера")
```

In [ ]:
```python
print(f"После добавления новых элементов:\n{bag}")
```

```
После добавления новых элементов:
Bag: ['Меркурий', 'Венера', 'Земля', 'Марс', 'Юпитер', 'Сатурн', 'Уран', 'Нептун', 'Плутон']
```

In [ ]:
```python
bag.remove("Сатурн")
```

In [ ]:
```python
print(f"После удаления элемента:\n{bag}")
```

```
После удаления элемента:
Bag: ['Меркурий', 'Венера', 'Земля', 'Марс', 'Юпитер', 'Сатурн', 'Уран', 'Нептун', 'Плутон']
```

In [ ]:
```
print(f"Количество уникальных элементов в мешке: {len(bag)}")
```

```
Количество уникальных элементов в мешке: 9
```

Метод `_normalize_` реализован через фильтрацию с использованием множества `seen`, что позволяет сохранить только первые вхождения каждого элемента. Метод `__len__` возвращает длину внутреннего списка, определяя количество уникальных планет в мешке. Пример демонстрирует добавление нового элемента (`"Плутон"`), повторное добавление уже существующего элемента (`"Венера"`) и удаление элемента (`"Сатурн"`), после чего функция `len(bag)` возвращает актуальное количество уникальных элементов. Таким образом можно адаптировать классы для работы с коллекциями, обеспечивая динамическое управление содержимым и корректное измерение их «размера» через переопределение метода `__len__`.

Метод `__contains__`

Метод `__contains__` задаёт логику работы оператора `in` для пользовательских объектов, позволяя определить, считается ли элемент принадлежащим коллекции. При выполнении `item in obj` Python вызывает `obj.__contains__(item)`, который должен вернуть `True` или `False` в зависимости от наличия элемента. Этим подходом можно воспользоваться, когда объект представляет собой набор данных и требуется нестандартный способ проверки, например, поиск товара по названию в каталоге или проверка, активен ли заказ в системе.

Пример ниже демонстрирует класс `ProductCatalog`, представляющий каталог товаров, где каждый товар хранится в виде словаря с ключами `id` и `name`. Метод `__contains__` переопределён так, что переданное значение рассматривается как название товара, а поиск выполняется в списке товаров. Если среди них есть элемент с совпадающим значением `name`, возвращается `True`, иначе — `False`.

In [ ]:
```
class ProductCatalog:
    def __init__(self, products):
        self.products = products[:]

    def __contains__(self, item):
        for product in self.products:
            if product.get("name") == item:
                return True
        return False

    def __str__(self):
        product_names = ", ".join(product.get("name") for product in self.products)
        return f"Каталог товаров: [{product_names}]"
```

In [ ]:
```
catalog = ProductCatalog([
    {"id": 101, "name": "Ноутбук"},
    {"id": 102, "name": "Смартфон"},
    {"id": 103, "name": "Планшет"},
    {"id": 104, "name": "Наушники"}
    ])
```

In [ ]:
```
print("Ноутбук" in catalog)
```

```
True
```

In [ ]:
```
print("Фотоаппарат" in catalog)
```

```
False
```

In [ ]:
```
print(catalog)
```

```
Каталог товаров: [Ноутбук, Смартфон, Планшет, Наушники]
```

В этом примере класс `ProductCatalog` хранит список товаров, переданный в конструктор, и обеспечивает собственную логику проверки наличия товара через метод `__contains__`. При обращении к выражению `"Ноутбук" in catalog` происходит итерация по списку товаров, и если среди них находится товар, у которого значение ключа `name` равно `"Ноутбук"`, метод возвращает `True`. Аналогичная проверка для `"Фотоаппарат"` возвращает `False`, так как такого товара в каталоге нет. Метод `__str__` предоставляет удобное для восприятия представление каталога, выводя список названий товаров.

Контекстные менеджеры

Метод `__enter__`

Метод `__enter__` является неотъемлемой частью протокола контекстного менеджера в Python, который обеспечивает безопасное выделение и последующее освобождение ресурсов. Когда объект используется в конструкции `with`, Python автоматически вызывает его метод `__enter__` перед выполнением кода внутри блока `with`. Результат этого метода затем присваивается переменной, указанной после ключевого слова `as`. Это позволяет выполнить такие операции, как открытие файла, установка соединения с базой данных или блокировка ресурса, гарантируя, что все необходимые подготовительные действия завершены до начала основного кода, а затем, в конце блока, метод `__exit__` обеспечивает корректное

освобождение этих ресурсов даже при возникновении исключений.

Пример ниже показывает класс `FileManager`, который использует метод `__enter__` для безопасного открытия файла. В конструкторе задаются имя файла и режим работы, а `__enter__` открывает файл и возвращает его, позволяя работать с ним внутри блока `with`.

In [ ]:
```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

Представим, что требуется обработать лог-файл в системе мониторинга, где важно гарантировать, что файл будет закрыт после завершения операций чтения или записи, независимо от того, возникнут ли ошибки в процессе. Пример использования класса `FileManager` демонстрирует этот механизм: файл открывается и возвращается в переменную, после чего с ним можно работать как с обычным объектом файла, а по завершении блока `with` ресурс автоматически закрывается, что предотвращает утечки ресурсов и ошибки доступа к файлам.

In [ ]:
```
with FileManager("system.log", "r") as logfile:
    for line in logfile:
        print(line.strip())
```

В этом примере блок `with` гарантирует, что метод `__enter__` откроет файл `system.log` в режиме чтения, а возвращённый объект файла будет доступен внутри блока. После завершения работы с файлом, даже если возникнет ошибка при обработке данных, метод `__exit__` автоматически закроет файл, обеспечивая корректное освобождение ресурса. Это демонстрирует, как использование метода `__enter__` в контекстном менеджере повышает надёжность и чистоту кода, избавляя от необходимости явно закрывать файлы и обрабатывать исключения, связанные с ресурсами.

Метод `__exit__`

Метод `__exit__` вызывается автоматически при выходе из блока `with` и отвечает за завершение работы с ресурсом, освобождение или закрытие его, независимо от того, произошло ли в блоке исключение. Этот метод принимает три параметра: `exc_type`, `exc_val` и `exc_tb`, которые представляют тип исключения, его значение и трассировку стека соответственно. Если блок `with` завершается без ошибок, все три параметра будут равны `None`. В противном случае они будут содержать информацию об исключении, что позволяет в методе `__exit__` выполнить необходимую обработку (например, логирование или корректное закрытие ресурса) и, при необходимости, подавить исключение, вернув `True`. Обычно в методе `__exit__` реализуют обязательное освобождение ресурсов, чтобы избежать утечек, а также могут добавлять дополнительную логику для обработки ошибок, возникающих в блоке `with`.

Класс `FileManager` демонстрирует использование метода `__exit__`. В конструкторе задаются имя файла и режим работы, `__enter__` открывает файл и возвращает его, а `__exit__` закрывает его после выхода из блока `with`. Если в процессе работы возникает исключение, метод выводит информацию о нём и не подавляет ошибку (возвращает `False`), позволяя передать её дальше для обработки.

In [ ]:
```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
        if exc_type is not None:
            print(f"Произошло исключение: {exc_type.__name__}: {exc_val}")
            return False
        return False
```

Пример использования FileManager:

In [ ]:
```
try:
    with FileManager("file.txt", "r") as f:
```

```
        data = f.read()
        print(data)
except Exception as e:
    print("Ошибка при работе с файлом:", e)
```

```
Ошибка при работе с файлом: [Errno 2] No such file or directory: 'file.txt'
```

В этом примере объект класса `FileManager` используется в конструкции `with`, что гарантирует вызов метода `__enter__` для открытия файла и получения объекта файла для работы внутри блока. После завершения работы с файлом (или при возникновении исключения) вызывается метод `__exit__`, который закрывает файл. Если в блоке `with` возникает исключение, информация о нем выводится на экран, а возвращаемое значение `False` позволяет передать исключение дальше, чтобы внешний код смог его обработать. Такой механизм гарантирует, что ресурс (файл) будет корректно освобожден, даже если во время его обработки произойдет ошибка.

Контекстные менеджеры обеспечивают удобный способ управления ресурсами, гарантируя их освобождение даже в случае возникновения исключений во время выполнения кода. Преимущество использования контекстных менеджеров заключается в том, что они гарантируют выполнение завершающих операций, таких как закрытие файлов, разрыв сетевых соединений или освобождение блокировок, благодаря чему программисту не нужно заботиться об этом вручную, что существенно снижает вероятность утечек ресурсов и ошибок в управлении ими.

Такой подход особенно ценен в системах, где стабильность работы критична, а ресурсы ограничены. Например, при работе с файлами контекстный менеджер, реализованный через конструкцию `with`, автоматически открывает и затем закрывает файл независимо от того, возникло ли исключение в процессе чтения или записи. Аналогичным образом можно управлять сетевыми соединениями: после установления соединения `with` гарантирует его корректное завершение, даже если во время обмена данными происходит сбой, что позволяет избежать подвешенных сокетов или блокировок портов.

Наконец, в многопоточных приложениях контекстные менеджеры применяются для работы с блокировками, обеспечивая автоматическое освобождение блокировки после завершения критической секции, что предотвращает взаимные блокировки и снижает риск возникновения состояния гонки. Таким образом, использование контекстных менеджеров делает код более чистым, надёжным и лёгким в сопровождении, поскольку освобождение ресурсов происходит гарантированно, а программная логика остаётся сосредоточенной на основной функциональности.

Кейс: построение иерархии классов

При проектировании программного обеспечения одним из ключевых этапов является выбор предметной области, которая определяет не только логику работы системы, но и её дальнейшее развитие. В качестве примера рассмотрим создание системы управления сотрудниками в компании — задачу, актуальную для многих организаций, стремящихся автоматизировать процессы учёта персонала, распределения обязанностей и формирования отчётности. Такая система позволяет сократить рутинную работу, повысить прозрачность процессов и заложить основу для интеграции с другими корпоративными сервисами, например, расчётом заработной платы или планированием ресурсов. Управление сотрудниками интересно тем, что сочетает общие характеристики, присущие всем работникам (например, `name` или базовый доход), с индивидуальными особенностями разных ролей, будь то разработчики, менеджеры или бухгалтеры. Это делает предметную область идеальной для демонстрации принципов объектно-ориентированного подхода, где общие черты можно выделить в базовую структуру, а специфические — реализовать через наследование и полиморфизм.

Прежде чем приступить к написанию кода, необходимо определить, какие задачи должна решать система. В основе её работы лежит потребность хранить данные о сотрудниках в единой структуре, включающей `name`, уникальный идентификатор (`employee_id`) и `salary`. Однако одной общей структуры недостаточно, поскольку роли сотрудников различаются: разработчикам важно знать язык программирования, менеджерам — размер команды, а другим должностям могут понадобиться свои параметры. Система должна поддерживать единые действия (выполнение работы или предоставление информации о сотруднике), причём эти действия могут отличаться в зависимости от роли. Например, разработчик пишет код, а менеджер координирует команду. Важно также обеспечить гибкость: если в будущем появятся новые роли или функции, их можно будет добавить без переписывания всей системы. Наконец, код должен быть понятным, удобным для тестирования и готовым к расширению, чтобы в дальнейшем подключать дополнительные возможности, например, учёт отпусков или расчёт бонусов.

Работу над системой начинают с создания базового класса `Employee`, который станет основой для всех сотрудников. Этот класс задаёт общие атрибуты и поведение, которые затем можно уточнять для конкретных ролей. Рассмотрим его реализацию:

In [ ]:
```
class Employee:
    def __init__(self, name, employee_id, salary):
        self.name = name
        self.employee_id = employee_id
        self.salary = salary

    def work(self):
        print(f"{self.name} выполняет свою работу.")

    def get_details(self):
        return f"{self.name}, ID: {self.employee_id}, зарплата: {self.salary}"
```

Конструктор класса принимает три параметра — `name`, `employee_id` и `salary`, — которые сохраняются в

соответствующих атрибутах объекта. Метод `work` описывает базовое действие сотрудника и на данном этапе имеет простую реализацию, выводя сообщение о том, что сотрудник занят делом. Это своего рода заготовка, которую подклассы смогут переопределить в соответствии со своими задачами. Метод `get_details` возвращает текстовое представление основных данных сотрудника — имя, идентификатор и зарплата, — в удобном для чтения виде. Например, создание объекта `emp = Employee("Алексей", "E001", 60000)` и вызов `emp.get_details()` вернёт строку `"Алексей, ID: E001, зарплата: 60000"`. Такой подход обеспечивает универсальность: независимо от роли сотрудника, система может работать с его базовыми характеристиками и действиями.

Далее необходимо учесть специфику разных должностей, для чего используются подклассы. Рассмотрим реализацию класса `Developer`, представляющего разработчиков. Поскольку разработчикам свойственна работа с конкретным языком программирования, в этот класс добавляется соответствующий атрибут, а методы адаптируются под их задачи:

In [ ]:
```
class Developer(Employee):
    def __init__(self, name, employee_id, salary, programming_language):
        super().__init__(name, employee_id, salary)
        self.programming_language = programming_language

    def work(self):
        print(f"{self.name} пишет код на {self.programming_language}.")

    def get_details(self):
        base_info = super().get_details()
        return f"{base_info}, язык: {self.programming_language}"
```

Класс `Developer` наследует все атрибуты и методы от `Employee`, вызывая конструктор базового класса через `super().__init__`, а затем добавляет поле `programming_language`, например, `"Python"` или `"Java"`. Метод `work` переопределяется, чтобы отражать деятельность разработчика: вместо общей фразы он сообщает, что сотрудник пишет код на определённом языке. Метод `get_details` расширяет базовую версию, добавляя информацию о языке программирования. Если создать объект `dev = Developer("Екатерина", "D001", 80000, "Python")`, то вызов `dev.work()` выведет `"Екатерина пишет код на Python"`, а `dev.get_details()` вернёт `"Екатерина, ID: D001, зарплата: 80000, язык: Python"`. Таким образом, подкласс сохраняет общую структуру, но уточняет поведение и данные в соответствии с ролью.

Аналогичным образом создаётся класс `Manager` для менеджеров, у которых важным параметром является размер команды. Его реализация выглядит следующим образом:

In [ ]:
```
class Manager(Employee):
    def __init__(self, name, employee_id, salary, team_size):
        super().__init__(name, employee_id, salary)
        self.team_size = team_size

    def work(self):
        print(f"{self.name} руководит командой из {self.team_size} человек.")

    def get_details(self):
        base_info = super().get_details()
        return f"{base_info}, команда: {self.team_size} человек"
```

Здесь конструктор добавляет атрибут `team_size`, указывающий количество подчинённых, а метод `work` переопределяется, чтобы показать, что менеджер управляет командой. Метод `get_details` дополняется информацией о размере команды. При создании объекта `mgr = Manager("Игорь", "M001", 95000, 5)` вызов `mgr.work()` выведет `"Игорь руководит командой из 5 человек"`, а `mgr.get_details()` вернёт `"Игорь, ID: M001, зарплата: 95000, команда: 5 человек"`. Такой подход позволяет сохранить общий интерфейс, унаследованный от `Employee`, одновременно отражая специфику управленческой роли.

Чтобы проверить, как система работает в целом, можно создать несколько сотрудников разных типов и протестировать их поведение. Рассмотрим пример:

In [ ]:
```
dev1 = Developer("Екатерина", "D001", 80000, "Python")
dev2 = Developer("Дмитрий", "D002", 85000, "Java")
mgr1 = Manager("Игорь", "M001", 95000, 5)
mgr2 = Manager("Ольга", "M002", 100000, 8)
```

In [ ]:
```
staff = [dev1, dev2, mgr1, mgr2]
```

In [ ]:
```
for person in staff:
    print(person.get_details())
    person.work()
    print("-" * 55)
```

```
Екатерина, ID: D001, зарплата: 80000, язык: Python
Екатерина пишет код на Python.
--------------------------------------------------------
Дмитрий, ID: D002, зарплата: 85000, язык: Java
Дмитрий пишет код на Java.
--------------------------------------------------------
Игорь, ID: M001, зарплата: 95000, команда: 5 человек
Игорь руководит командой из 5 человек.
--------------------------------------------------------
Ольга, ID: M002, зарплата: 100000, команда: 8 человек
Ольга руководит командой из 8 человек.
--------------------------------------------------------
```

При выполнении этого кода создаются два разработчика и два менеджера, которые затем помещаются в список `staff`. Цикл проходит по списку, вызывая для каждого объекта методы `get_details` и `work`. Результат демонстрирует, как единый код обрабатывает разные типы сотрудников: разработчики выводят информацию о языке программирования и сообщают о написании кода, а менеджеры — о размере команды и управлении. Вывод будет включать строки вроде `"Екатерина, ID: D001, зарплата: 80000, язык: Python"` и `"Екатерина пишет код на Python"` для разработчиков, а для менеджеров — `"Игорь, ID: M001, зарплата: 95000, команда: 5 человек"` и `"Игорь руководит командой из 5 человек"`. Это наглядно иллюстрирует полиморфизм: несмотря на общий интерфейс, каждый подкласс реализует методы по-своему.

Система легко расширяется. Например, если в компании появляется бухгалтер, можно добавить новый подкласс `Accountant`:

In [ ]:
```python
class Accountant(Employee):
    def __init__(self, name, employee_id, salary, department):
        super().__init__(name, employee_id, salary)
        self.department = department

    def work(self):
        print(f"{self.name} готовит отчёты для отдела {self.department}.")

    def get_details(self):
        base_info = super().get_details()
        return f"{base_info}, отдел: {self.department}"
```

Создание объекта `acc = Accountant("Марина", "A001", 70000, "финансы")` и добавление его в список `staff` позволяет без изменений в основном коде вывести `"Марина, ID: A001, зарплата: 70000, отдел: финансы"` и `"Марина готовит отчёты для отдела финансы"`. Это показывает, что структура системы остаётся устойчивой при добавлении новых ролей.

Преимущества такого подхода очевидны. Во-первых, общие характеристики сотрудников собраны в классе `Employee`, что исключает дублирование кода и упрощает его сопровождение. Во-вторых, подклассы позволяют гибко учитывать различия между ролями, добавляя только необходимые атрибуты и поведение. В-третьих, полиморфизм обеспечивает единообразную обработку сотрудников независимо от их типа: один цикл может работать с разработчиками, менеджерами и бухгалтерами, вызывая соответствующие версии методов. Наконец, система готова к развитию — новые функции, такие как учёт проектов или бонусов, можно добавить через дополнительные атрибуты и методы, не затрагивая существующую логику.

Этот пример демонстрирует, как иерархия классов помогает решать задачу управления сотрудниками. Подобный подход применим и в других областях: например, в интернет-магазине общая структура может описывать товары, а подклассы — уточнять особенности продуктов, таких как книги или электроника. В любом случае данные остаются согласованными, а система — открытой для изменений, что делает её надёжной основой для практического применения и изучения объектно-ориентированного программирования.

Задания для самостоятельного решения

1. Разработка системы управления библиотечным каталогом

Постановка задачи

Необходимо разработать программную систему для управления библиотечным каталогом, позволяющую учитывать различные типы изданий — книги и журналы. Система должна обеспечивать хранение данных об изданиях, возможность сравнения их по количеству страниц и предоставление информации в удобном формате. Архитектура системы должна быть гибкой, чтобы в будущем можно было расширить функциональность для поддержки новых типов изданий (например, аудиокниг). Реализация обязана использовать принципы объектно-ориентированного программирования, включая наследование, полиморфизм и магические методы Python для сравнения и строкового представления объектов.

Требования к структуре

Создать иерархию классов с базовым классом `LibraryItem`, представляющим любое библиотечное издание, и двумя подклассами: `Book` для книг и `Magazine` для журналов. Базовый класс должен определять общие атрибуты и поведение, а подклассы — дополнять их специфическими характеристиками и переопределять методы в соответствии с особенностями

каждого типа издания.

Спецификация классов

- **Класс `LibraryItem`:**

  Атрибуты:

  - `title` (строка) — название издания;
  - `item_id` (строка) — уникальный идентификатор (например, `"B001"`, `"M001"`);
  - `pages` (целое число) — количество страниц.

  Методы:

  - `__init__(title, item_id, pages)` — конструктор для инициализации атрибутов;
  - `describe()` — возвращает строку с описанием, например, `"Издание 'title' с ID item_id"`;
  - `__str__()` — возвращает строковое представление вида `"title (ID: item_id), страниц: pages"`;
  - `__eq__(other)` — магический метод для сравнения изданий по количеству страниц; возвращает `True`, если количество страниц равно, и `NotImplemented`, если `other` не является экземпляром `LibraryItem`;
  - `__lt__(other)` — магический метод для определения, меньше ли текущее издание по количеству страниц; возвращает `NotImplemented`, если `other` не является `LibraryItem`.

- **Класс `Book` (наследуется от `LibraryItem`):**

  Атрибут:

  - `author` (строка) — имя автора книги.

  Методы:

  - `__init__(title, item_id, pages, author)` — вызывает `__init__` базового класса и задаёт `author`;
  - `describe()` — возвращает строку вида `"Книга 'title' от автора author"`;
  - `__str__()` — возвращает строковое представление вида `"title от author (ID: item_id), страниц: pages"`;
  - `__add__(other)` — магический метод, который объединяет две книги одного автора в новое издание. Новое издание имеет сумму страниц исходных книг и название вида `"title1 и title2"`. Метод должен возвращать новый объект `Book`. Если авторы разные или `other` не является экземпляром `Book`, необходимо выбросить исключение `ValueError`.

- **Класс `Magazine` (наследуется от `LibraryItem`):**

  Атрибут:

  - `issue_number` (целое число) — номер выпуска журнала.

  Методы:

  - `__init__(title, item_id, pages, issue_number)` — вызывает `__init__` базового класса и задаёт `issue_number`;
  - `describe()` — возвращает строку вида `"Журнал 'title', выпуск issue_number"`;
  - `__str__()` — возвращает строковое представление вида `"title, выпуск issue_number (ID: item_id), страниц: pages"`;
  - `__gt__(other)` — магический метод для сравнения журналов по номеру выпуска; возвращает `True`, если номер выпуска текущего объекта больше, и `NotImplemented`, если `other` не является экземпляром `Magazine`.

Критерии проверки

Для тестирования системы следует использовать следующий сценарий:

1. Создать объект:
   - `book1 = Book("Война и мир", "B001", 1200, "Л. Толстой")`

- `mag1 = Magazine("Наука", "M001", 50, 15)`

2. Вывести строковое представление:
   - `print(book1)` должно вывести:
     `"Война и мир от Л. Толстой (ID: B001), страниц: 1200"`.
   - `print(mag1)` должно вывести:
     `"Наука, выпуск 15 (ID: M001), страниц: 50"`.
3. Выполнить сравнения:
   - `book1 == mag1` → ожидается `False`.
   - `book1 < mag1` → ожидается `False`.
4. Создать объект и объединить книги:
   - `book2 = Book("Анна Каренина", "B002", 800, "Л. Толстой")`
   - `book3 = book1 + book2`
   - `print(book3)` должно вывести:
     `"Война и мир и Анна Каренина от Л. Толстой (ID: None), страниц: 2000"`
     (При этом `item_id` нового объекта может быть опущен или задан как `None`).
5. Создать объект журнала и выполнить сравнение:
   - `mag2 = Magazine("Природа", "M002", 60, 10)`
   - Проверить выражение `mag1 > mag2`, которое должно вернуть `True`.

---

2. Разработка системы управления транспортными средствами

Постановка задачи

Необходимо разработать программную систему для управления транспортными средствами транспортной компании, включающую учёт автомобилей и мотоциклов. Система должна поддерживать сравнение транспортных средств по грузоподъёмности и топливному запасу, а также предоставлять подробную информацию о каждом объекте. Архитектура должна быть расширяемой для добавления новых типов транспорта (например, грузовиков) и реализована с применением объектно-ориентированного подхода, наследования, полиморфизма и магических методов Python.

Требования к структуре

Создать иерархию классов с базовым классом `Vehicle`, представляющим любое транспортное средство, и двумя подклассами: `Car` для автомобилей и `Motorcycle` для мотоциклов. Базовый класс определяет общие характеристики и поведение, а подклассы дополняют их специфическими атрибутами и адаптируют методы в соответствии с особенностями каждого типа транспорта.

Спецификация классов

- **Класс `Vehicle`:**

  Атрибуты:

  - `model` (строка) — модель транспортного средства;
  - `vin` (строка) — уникальный идентификатор (VIN-код);
  - `fuel_capacity` (число с плавающей точкой) — ёмкость топливного бака в литрах;
  - `load_capacity` (число с плавающей точкой) — грузоподъёмность в килограммах.

  Методы:

  - `__init__(model, vin, fuel_capacity, load_capacity)` — конструктор для инициализации атрибутов;
  - `info()` — возвращает строку вида `"Транспорт: model (VIN: vin)"`;
  - `__str__()` — возвращает строковое представление вида `"model (VIN: vin), топливо: fuel_capacity л, груз: load_capacity кг"`;
  - `__eq__(other)` — магический метод для сравнения транспортных средств по грузоподъёмности; возвращает `True`, если значения равны, и `NotImplemented`, если `other` не является экземпляром `Vehicle`;
  - `__le__(other)` — магический метод для проверки, меньше или равна ли грузоподъёмность текущего объекта грузоподъёмности другого; возвращает `NotImplemented`, если `other` не является экземпляром `Vehicle`.

- **Класс `Car` (наследуется от `Vehicle`):**

  Атрибут:

  - `seats` (целое число) — количество мест в автомобиле.

  Методы:

  - `__init__(model, vin, fuel_capacity, load_capacity, seats)` — вызывает `__init__` базового класса и задаёт `seats`;
  - `info()` — возвращает строку вида `"Автомобиль: model, мест: seats"`;
  - `__str__()` — возвращает строковое представление вида `"model (VIN: vin), топливо: fuel_capacity л, груз: load_capacity кг, мест: seats"`;
  - `__add__(other)` — магический метод, который увеличивает грузоподъёмность и объём топливного бака, «добавляя» другой автомобиль как прицеп. Метод должен возвращать новый объект `Car` с той же моделью, суммарными значениями топливного бака и грузоподъёмности. Если `other` не является экземпляром `Car`, необходимо выбросить исключение `TypeError`.

- **Класс `Motorcycle` (наследуется от `Vehicle`):**

  Атрибут:

  - `has_sidecar` (логическое значение) — наличие коляски (`True` / `False`).

  Методы:

  - `__init__(model, vin, fuel_capacity, load_capacity, has_sidecar)` — вызывает `__init__` базового класса и задаёт `has_sidecar`;
  - `info()` — возвращает строку вида `"Мотоцикл: model, с коляской"` или `"Мотоцикл: model, без коляски"`, в зависимости от значения `has_sidecar`;
  - `__str__()` — возвращает строковое представление вида `"model (VIN: vin), топливо: fuel_capacity л, груз: load_capacity кг, с коляской"` или `"model (VIN: vin), топливо: fuel_capacity л, груз: load_capacity кг, без коляски"`;
  - `__gt__(other)` — магический метод для сравнения мотоциклов по топливному запасу; возвращает `True`, если текущий объект имеет больший запас топлива, и `NotImplemented`, если `other` не является экземпляром `Motorcycle`.

Критерии проверки

Для тестирования системы предусмотрен следующий сценарий:

1. Создать объекты:
   - `car1 = Car("Toyota Camry", "VIN123", 50.0, 500.0, 5)`
   - `moto1 = Motorcycle("Honda CBR", "VIN456", 15.0, 150.0, False)`
2. Вывести строковое представление:
   - `print(car1)` должно вывести:
     `"Toyota Camry (VIN: VIN123), топливо: 50.0 л, груз: 500.0 кг, мест: 5"`.
   - `print(moto1)` должно вывести:
     `"Honda CBR (VIN: VIN456), топливо: 15.0 л, груз: 150.0 кг, без коляски"`.
3. Выполнить сравнения:
   - `car1 == moto1` → ожидается `False`.
   - `car1 <= moto1` → ожидается `False`.
4. Создать и объединить автомобили:
   - `car2 = Car("Ford Focus", "VIN789", 45.0, 400.0, 4)`
   - `car3 = car1 + car2`
   - `print(car3)` должно вывести:
     `"Toyota Camry (VIN: VIN123), топливо: 95.0 л, груз: 900.0 кг, мест: 5"`.
5. Создать объект мотоцикла и выполнить сравнение:
   - `moto2 = Motorcycle("Yamaha R1", "VIN999", 20.0, 200.0, True)`
   - Проверить выражение `moto2 > moto1`, которое должно вернуть `True`.