Лекция 4. Определение полиморфизма. Виды аргументов

**Оглавление**

[**Введение** 3](#_Toc188951228)

[**1.** **Понятие полиморфизма. Применение полиморфизма.** 4](#_Toc188951229)

[**Примеры полиморфизма в Python** 5](#_Toc188951230)

[**Преимущества использования полиморфизма** 6](#_Toc188951231)

[**Формы полиморфизма** 7](#_Toc188951232)

[**2.** **Абстракция в Python. Абстрактные классы** 9](#_Toc188951233)

[**Абстрактные классы в Python** 9](#_Toc188951234)

[**Заключение** 13](#_Toc188951235)

[**Список литературы** 14](#_Toc188951236)

**Понятие полиморфизма. Виды аргументов**

**Цель:** формирование знаний по базовым элементам объектно-ориентированного программирования – полиморфизма и абстракции

**План лекции:**Введение.1. Понятие полиморфизма. Применение полиморфизма.  
2. Абстракция в Python. Абстрактные классы.Заключение.Список литературы.

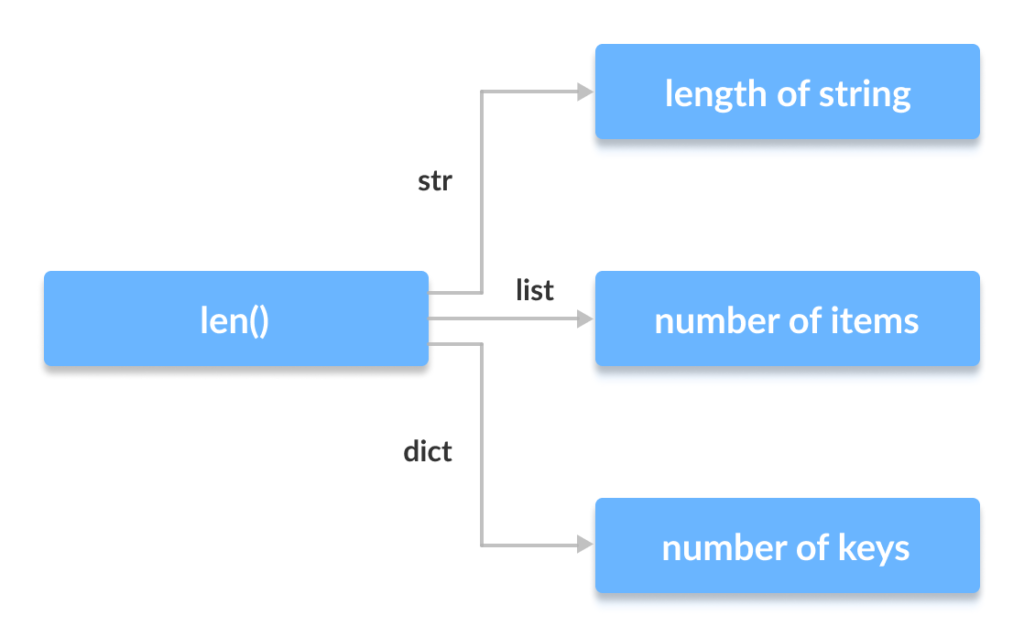
# **Введение**

В мире программирования, Python зарекомендовал себя как один из наиболее популярных и гибких языков. Это связано с его простотой, мощностью и широкой применимостью в различных областях, от веб-разработки до научных исследований. Сегодняшняя лекция посвящена трем ключевым концепциям объектно-ориентированного программирования (ООП): полиморфизму, абстракции и абстрактным методам. Полиморфизм позволяет программам работать с объектами различных типов, предоставляя универсальные методы для их обработки. Абстракция, в свою очередь, помогает скрыть сложные детали реализации и позволяет сосредоточиться на важных аспектах задачи. Абстрактные методы служат основой для создания интерфейсов, которые могут быть реализованы в различных подклассах, обеспечивая гибкость и модульность кода. Все эти концепции играют важную роль в создании устойчивых и поддерживаемых программных решений, и их понимание является неотъемлемой частью навыков любого программиста, работающего с Python.

# **Понятие полиморфизма. Применение полиморфизма.**

Полиморфизм — это одна из ключевых концепций объектно-ориентированного программирования (ООП). Слово "полиморфизм" происходит от греческих слов "поли" (много) и "морф" (форма), что буквально означает "много форм". В контексте программирования полиморфизм позволяет методам и функциям работать с объектами разных типов, не зная заранее, какого типа объект будет передан. Это делает код более гибким и расширяемым, так как позволяет использовать один и тот же интерфейс для различных типов данных. Это особенно полезно при разработке больших и сложных систем, где важно минимизировать зависимость между компонентами.

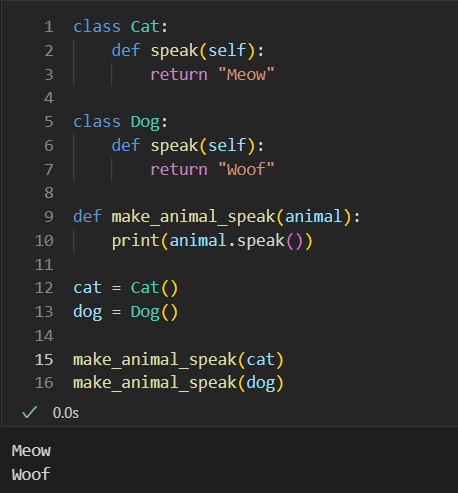
Полиморфизм в программировании помогает разработчикам создавать более универсальные и адаптируемые решения. Например, представьте, что у вас есть несколько классов, представляющих различные виды животных, и каждый из этих классов имеет метод speak. Благодаря полиморфизму, вы можете вызвать метод speak для любого объекта животного, не зная его конкретного типа, и получить правильный результат. Это позволяет писать более общий и переиспользуемый код, что особенно важно в крупных проектах.



Важно отметить, что в «Питоне» не поддерживается такой вариант **method overriding**, как создание методов с тем же самым именем, однако с различными типами аргументов.

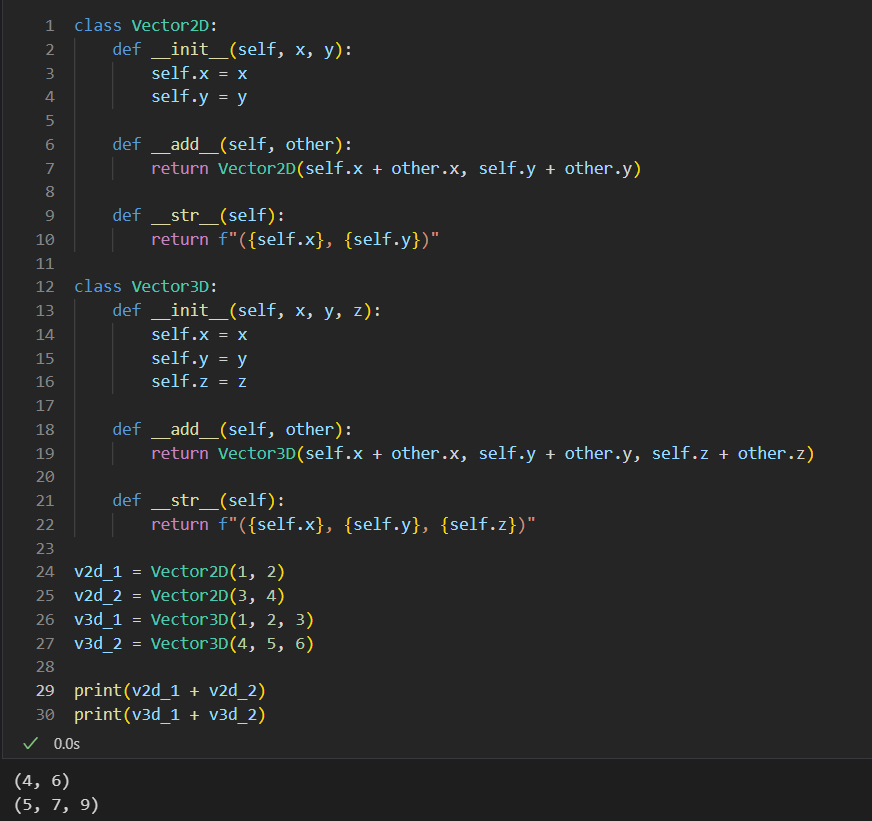
## **Примеры полиморфизма в Python**

Полиморфизм в Python можно продемонстрировать с помощью методов и операторов, которые ведут себя по-разному в зависимости от типа объекта. Рассмотрим простой пример с двумя классами — Cat и Dog, которые имеют метод speak. Этот пример иллюстрирует, как один и тот же метод может выполнять разные действия в зависимости от объекта, который его вызывает.



В этом примере функция make\_animal\_speak принимает объект и вызывает его метод speak. В зависимости от типа объекта (Cat или Dog), метод speak возвращает разные строки. Это демонстрирует, как полиморфизм позволяет использовать один и тот же интерфейс для работы с различными объектами, что упрощает и делает код более гибким.

Полиморфизм также можно использовать с операторами. Например, оператор + может быть перегружен для выполнения различных задач в зависимости от типов операндов. Рассмотрим пример с классами Vector2D и Vector3D.



В этом примере оператор + перегружен для классов Vector2D и Vector3D, что позволяет складывать векторы различных размерностей. Это еще один пример полиморфизма, где один и тот же оператор выполняет разные действия в зависимости от типов операндов.

## **Преимущества использования полиморфизма**

Полиморфизм имеет множество преимуществ, среди которых:

* Упрощение кода: Полиморфизм позволяет писать более общий и менее детализированный код, который легче поддерживать и расширять. Это особенно полезно в больших проектах, где изменение одного компонента может затронуть множество других частей кода.
* Гибкость: Код становится более гибким и может работать с объектами разных типов без изменения основной логики. Это позволяет легко адаптировать существующий код к новым требованиям и условиям.
* Расширяемость: Легко добавлять новые классы и методы, не изменяя существующий код. Это упрощает процесс добавления новых функций и возможностей в программу.
* Повторное использование кода: Полиморфизм способствует повторному использованию кода, так как один и тот же метод или функция может работать с различными типами объектов. Это уменьшает количество дублирующегося кода и облегчает его поддержку.
* Улучшение читаемости кода: Полиморфизм делает код более читаемым и понятным, так как он позволяет использовать более абстрактные и обобщенные конструкции. Это облегчает понимание и поддержку кода, особенно в больших проектах.

## **Формы полиморфизма**

Существуют разные виды полиморфизма. Вообще-то классификация довольно широкая и начинающему легко в ней запутаться, поэтому мы решили ограничиться только основными формами.

**Полиморфизм подтипов.** Это полиморфность «по умолчанию»: когда в ООП говорят о полиморфизме, обычно имеют в виду его. Выше мы рассказывали именно про такой тип. Это возможность использовать одни и те же команды, или интерфейсы, для разных сущностей — подтипов.

**Параметрический полиморфизм.** Его еще называют обобщенным полиморфизмом. В нем для команды не имеет значения, какую сущность ей прислали: для всех возможных классов будет использоваться один код. Такой полиморфизм считается «истинным» и делает код универсальнее, но реализовать его сложнее.

**Полиморфизм ad hoc.** Этот вид полиморфизма еще называют специализированным. Его иногда противопоставляют параметрическому: идея ad hoc — разный код при одинаковом названии. Часто такой полиморфизм реализуют с помощью перегрузки методов: несколько раз пишут метод с одним и тем же названием, но разным кодом.

# **Абстракция в Python. Абстрактные классы**

Одна из основных целей использования абстракции в ООП – повышение гибкости и упрощение разработки. Абстрактный подход помогает создавать интерфейсы и классы, которые определяют только те свойства и методы, которые необходимы для выполнения определенной задачи. Это позволяет создавать более гибкие и масштабируемые приложения, которые легко поддаются изменению и расширению.

Предположим, что нам нужно написать программу, которая работает с графическими объектами разных типов. Для решения этой задачи удобно создать абстрактный класс **Shape** (фигура), определяющий абстрактные методы, которые могут быть использованы для работы с любой фигурой. Затем мы можем создать конкретные классы для конкретных типов фигур – окружность, квадрат, треугольник и т.д., которые расширяют базовый класс Shape. При этом мы можем использовать только те свойства и методы, которые необходимы для выполнения конкретной задачи, игнорируя детали реализации, которые не имеют значения в данном контексте.

Абстрактный подход помогает эффективно решать ряд сложных задач:

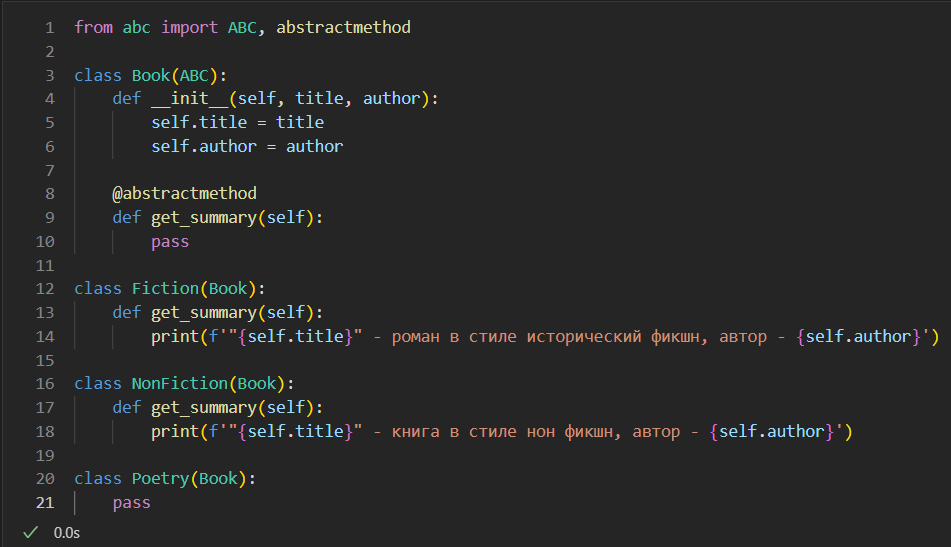
* Позволяет выделять существенные характеристики объекта, игнорируя все незначительные детали.
* Принуждает подклассы к реализации конкретных методов или к выполнению определенных требований путем определения абстрактных методов или свойств. Таким образом, абстракция позволяет определять общие интерфейсы для классов, но при этом гарантирует, что каждый подкласс будет реализовывать свою версию этих методов или свойств.
* Позволяет создавать общие модели объектов, которые могут использоваться для создания конкретных объектов.
* Упрощает работу со сложными системами, которые включают множество взаимодействующих компонентов, и позволяет создавать расширяемые, модульные приложения.

## **Абстрактные классы в Python**

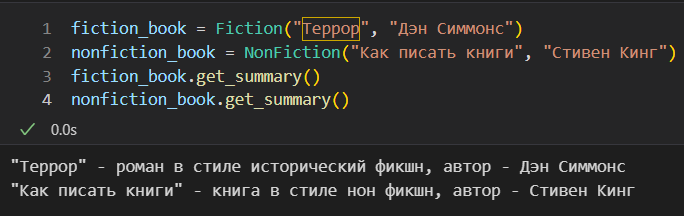
Для работы с абстрактными классами в Python используют модуль **abc**. Он предоставляет:

* **abc.ABC** – базовый класс для создания абстрактных классов. Абстрактный класс содержит один или несколько абстрактных методов, то есть методов без определения (пустых, без кода). Эти методы необходимо переопределить в подклассах.
* **abc.abstractmethod** – декоратор, который указывает, что метод является абстрактным. Этот декоратор применяется к методу внутри абстрактного класса. Класс, который наследует свойства и методы от абстрактного класса, должен реализовать все абстрактные методы, иначе он также будет считаться абстрактным.

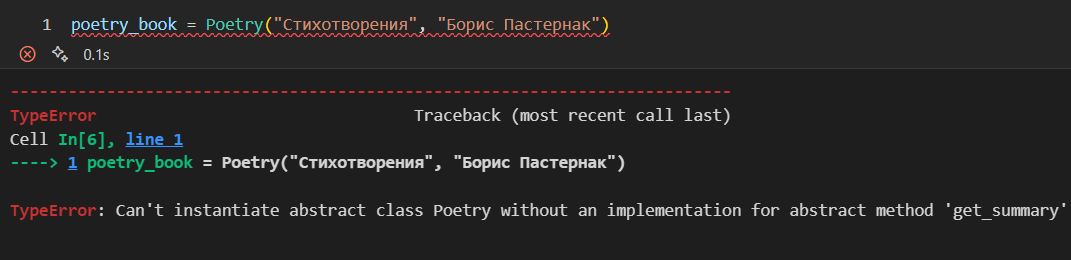
Рассмотрим пример абстрактного класса **Book**:



Класс **Book** имеет абстрактный метод **get\_summary()**. Два подкласса Book (**Fiction** и **NonFiction**) реализуют метод **get\_summary()**, а третий подкласс **Poetry** – нет. Когда мы создаем экземпляры Fiction и NonFiction и вызываем их методы get\_summary(), получаем ожидаемый результат:



А вот вызов **Poetry** приведет к ошибке, поскольку в этом подклассе метод get\_summary() не реализован:



Приведенный выше пример показывает, что семейство родственных классов (**Fiction** и **NonFiction** в нашем случае) может иметь общий интерфейс (метод **get\_summary()**), но реализация этого интерфейса может быть разной. Мы также убедились, что любой подкласс **Book** должен реализовать метод **get\_summary()**, чтобы обеспечить согласованную, безошибочную работу приложения.

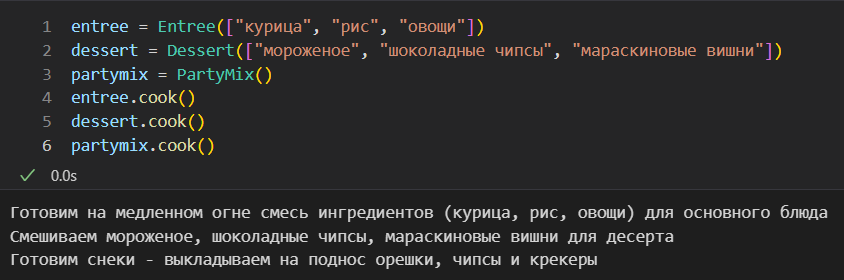
Теперь рассмотрим чуть более сложный пример, который продемонстрирует, как можно комбинировать абстракцию с другими концепциями ООП. Определим абстрактный класс **Recipe** (рецепт), который имеет абстрактный метод **cook()**. Затем создадим три подкласса **Entree**, **Dessert** и **Appetizer** (основное блюдо, десерт и закуска). **Entree** и **Dessert** имеют свои собственные методы **cook()**, в отличие от **Appetizer** и **PartyMix**. **PartyMix** (орешки, чипсы, крекеры) является подклассом **Appetizer** и имеет свою реализацию **cook()**:



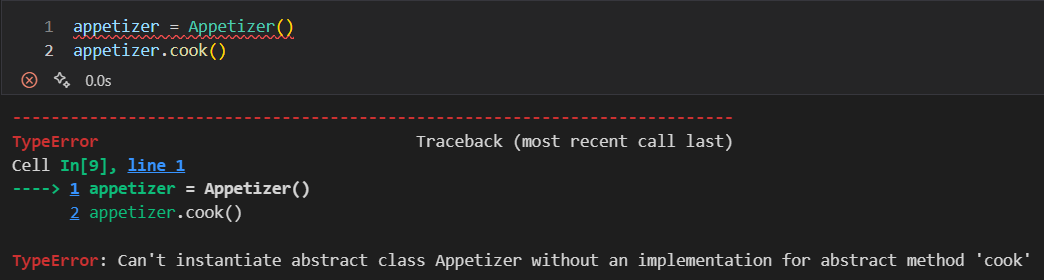
В этом примере наряду с абстракцией используются концепции полиморфизма и наследования.

Наследование заключается в том, что подклассы Entree, Dessert и PartyMix наследуют абстрактный метод cook() от абстрактного базового класса Recipe. Это означает, что все они имеют ту же сигнатуру (название и параметры) метода cook(), что и абстрактный метод, определенный в классе Recipe.

Полиморфизм проявляется в том, что каждый подкласс класса Recipe реализует метод cook() по-разному. Например, Entree реализует cook() для вывода инструкций по приготовлению основного блюда на медленном огне, а Dessert реализует cook() для вывода инструкций по смешиванию ингредиентов десерта. Эта разница в реализации является примером полиморфизма, когда различные объекты могут рассматриваться как объекты, которые относятся к одному типу, но при этом ведут себя по-разному:



Вызов метода cook() для подкласса Appetizer приведет к ожидаемой ошибке:



# **Заключение**

На протяжении этой лекции мы рассмотрели фундаментальные концепции полиморфизма, абстракции и абстрактных методов в Python, и как они помогают улучшить качество и структуру программного обеспечения. Полиморфизм предоставляет возможность создавать универсальные функции и методы, которые могут работать с объектами разных типов, что увеличивает гибкость и расширяемость кода. Абстракция помогает скрыть сложные детали реализации, предоставляя программисту инструменты для работы с более высокоуровневыми концепциями. Это, в свою очередь, упрощает разработку и сопровождение программных продуктов, так как позволяет концентрироваться на решении конкретных задач, а не на деталях реализации. Абстрактные методы задают шаблоны для создания интерфейсов, которые могут быть реализованы в различных подклассах, обеспечивая гибкость и модульность. Понимание и правильное применение этих концепций позволяет создавать мощные, масштабируемые и поддерживаемые программные решения. Для успешного использования данных концепций необходимо не только теоретическое знание, но и практические навыки их реализации в коде. Поэтому, я настоятельно рекомендую вам не только изучать данные концепции, но и применять их в ваших проектах. Практика является ключевым элементом в овладении этими инструментами и поможет вам стать более эффективным и квалифицированным программистом.

# **Список литературы**

1. https://habr.com/ru/articles/552922/
2. https://sky.pro/wiki/python/polimorfizm-v-python/
3. https://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html
4. https://younglinux.info/oopython/polymorphism
5. https://sky.pro/wiki/python/oop-v-python-polimorfizm/
6. https://otus.ru/journal/polimorfizm-v-pajton/
7. https://blog.skillfactory.ru/glossary/polimorfizm/
8. https://proproprogs.ru/python\_oop/python-polimorfizm-i-abstraktnye-metody