Лекция 3. Понятие инкапсуляции. Виды атрибутов и методов

**Оглавление**

[**Введение** 3](#_Toc188620714)

[**1.** **Декораторы в Python.** 4](#_Toc188620715)

[**2.** **Понятие инкапсуляции. Применение инкапсуляции.** 10](#_Toc188620716)

[**3.** **Методы и атрибуты. Геттеры и сеттеры. Модификаторы доступа.** 11](#_Toc188620717)

[**Заключение** 12](#_Toc188620718)

[**Список литературы** 13](#_Toc188620719)

**Понятие инкапсуляции. Виды атрибутов и методов**

**Цель:** формирование знаний по одному из базовых элементов объектно-ориентированного программирования – инкапсуляции, его назначению и способам его применения.

**План лекции:**Введение.1. Декораторы в Python.  
2. Понятие инкапсуляции. Применение инкапсуляции.  
3. Методы и атрибуты. Геттеры и сеттеры. Модификаторы доступа.Заключение.Список литературы.

# **Введение**

Инкапсуляция является одним из ключевых принципов объектно-ориентированного программирования (ООП), который помогает управлять доступом к данным и методам объектов. В Python инкапсуляция реализуется через использование атрибутов и методов, а также при помощи специальных инструментов, таких как декораторы, геттеры, сеттеры и модификаторы доступа.

Прежде чем погружаться в саму концепцию инкапсуляции, важно понять, как работают декораторы в Python. Декораторы — это мощный инструмент, который позволяет модифицировать поведение функций или методов без изменения их исходного кода. Декораторы часто используются для оборачивания функций дополнительной функциональностью, и они могут быть полезны при реализации геттеров и сеттеров.

Геттеры и сеттеры позволяют контролировать доступ к атрибутам объекта, обеспечивая возможность проверки данных перед их изменением или получения их значения. В Python геттеры и сеттеры часто реализуются при помощи декораторов @property и @<атрибут>.setter.

Инкапсуляция позволяет скрывать внутреннее состояние объекта и предоставлять интерфейс для взаимодействия с этим состоянием. Мы рассмотрим, как использование модификаторов доступа (таких как \_ и \_\_) помогает ограничить доступ к атрибутам и методам класса, делая их защищенными или приватными.

В ходе лекции мы обсудим:

* Основные концепции декораторов и их применение для геттеров и сеттеров.
* Принципы инкапсуляции и её преимущества.
* Реализацию геттеров и сеттеров для управления доступом к атрибутам.
* Использование модификаторов доступа для защиты данных.

Итак, приступим к изучению этих важных концепций и их практическому применению в языке Python.

# **Декораторы в Python.**

Декораторы в Python позволяют расширять и изменять поведение вызываемых объектов (функций, методов и классов) без постоянного изменения самого вызываемого объекта.

Любая достаточно общая функциональность, которую можно «прикрепить» к поведению существующего класса или функции, является отличным примером использования декораторов. Сюда входит:

* журналирование,
* обеспечение контроля доступа и аутентификации,
* инструментарий и функции управления временем,
* ограничение скорости,
* кэширование и многое другое.

Для лучшего понимания декораторов вспомним свойства функций:

* Функции являются объектами — они могут быть присвоены переменным, переданы другим функциям и возвращены из них.
* Функции могут быть определены внутри других функций, и дочерняя функция может захватывать локальное состояние родительской функции (лексические замыкания).

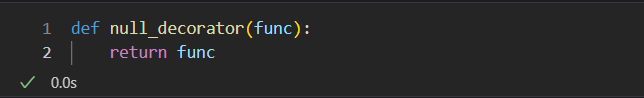
### **Основы декораторов Python**

Итак, что же такое декораторы на самом деле? Они «декорируют» или «оборачивают» другую функцию и позволяют выполнять код до и после выполнения обернутой функции.

Декораторы позволяют определять повторно используемые модули, которые могут изменять или расширять поведение других функций. При этом они позволяют делать это без постоянного изменения самой обернутой функции. Поведение функции меняется только тогда, когда она *декорируется*.

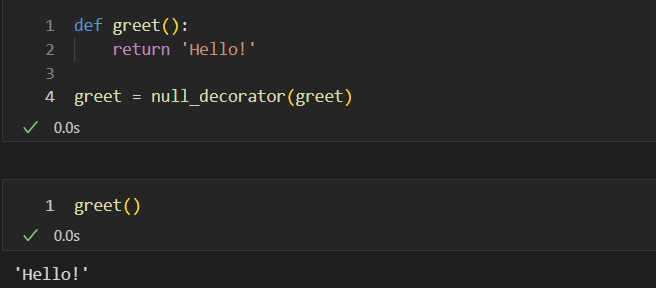
Итак, как выглядит реализация простого декоратора? В общих чертах декоратор — это вызываемый объект, который принимает на вход вызываемый объект и возвращает другой вызываемый объект.

Следующая функция обладает этим свойством и может считаться самым простым декоратором, который только можно написать:



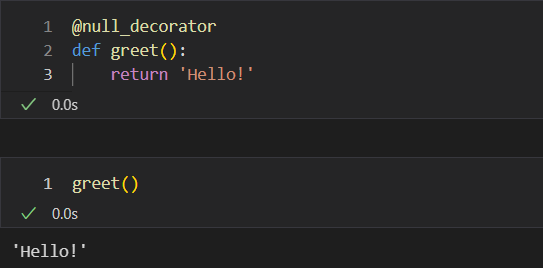
Как видите, null\_decorator является вызываемым объектом, он принимает на вход другой вызываемый объект и возвращает тот же самый входной объект, не изменяя его.

Давайте используем его для декорирования (или обертывания) другой функции:



В этом примере я определил функцию greet, а затем сразу же декорировал ее, прогнав ее через функцию null\_decorator. Я знаю, пока это не выглядит чем-то очень полезным (*мы ведь специально разработали декоратор null, чтобы он был бесполезным*), но через некоторое время это прояснит, как работает синтаксис декораторов в Python.

Вместо того, чтобы явно вызывать null\_decorator для greet, а затем переназначать переменную greet, можно использовать синтаксис Python @ для декорирования функции за один шаг:



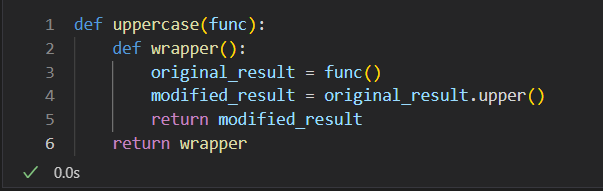
Разместить строки @null\_decorator перед определением функции — это то же самое, что сначала определить функцию, а затем применить к ней декоратор. Использование синтаксиса @ — это просто синтаксический сахар и сокращение для этого часто используемого шаблона.

Обратите внимание, что использование синтаксиса @ декорирует функцию непосредственно во время определения. Это затрудняет доступ к недекорированному оригиналу без хрупких хаков. Поэтому вы можете декорировать некоторые функции вручную, чтобы сохранить возможность вызова недекорированной функции.

### **Декораторы могут изменять поведение**

Теперь, когда мы немного познакомились с синтаксисом декораторов, давайте напишем еще один декоратор, который *действительно* что-то делает и изменяет поведение декорируемой функции.

Вот немного более сложный декоратор, который преобразует результат декорированной функции в заглавные буквы:

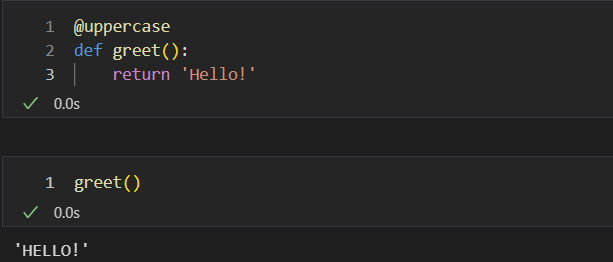


Вместо того, чтобы просто возвращать входную функцию, как это делал декоратор null, декоратор uppercase определяет новую функцию на лету (замыкание) и использует ее для обертывания входной функции, чтобы изменить ее поведение во время вызова.

Замыкание wrapper имеет доступ к недекорированной входной функции и может свободно выполнять дополнительный код до и после вызова входной функции. (Технически, ей вообще не нужно вызывать входную функцию).

Обратите внимание, что до сих пор декорированная функция никогда не выполнялась. На самом деле вызов входной функции в этот момент не имеет никакого смысла — декоратор должен иметь возможность изменять поведение своей входной функции, когда она будет вызвана.

Пришло время увидеть декоратор uppercase в действии. Что произойдет, если декорировать им исходную функцию greet?



В отличие от null\_decorator, декоратор uppercase возвращает *другой объект функции*, когда он декорирует функцию.

Как мы видели ранее, это необходимо для того, чтобы изменить поведение декорированной функции, когда она будет вызвана. Декоратор uppercase сам является функцией. И единственный способ повлиять на «будущее поведение» входной функции, которую он декорирует, — это заменить (или обернуть) входную функцию замыканием.

Вот почему uppercase определяет и возвращает другую функцию (замыкание), которую можно вызвать позднее, запустить исходную входную функцию и изменить ее результат.

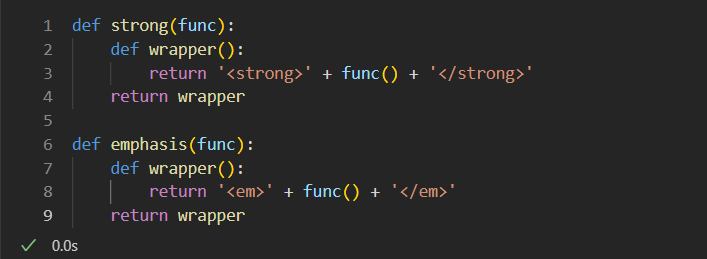
Декораторы изменяют поведение вызываемого объекта с помощью обертки, поэтому нам не нужно постоянно изменять оригинал. Вызываемый объект не подвергается постоянным изменениям — его поведение меняется только при декорировании.

Это позволяет «присоединять» к существующим функциям и классам повторно используемые модули, такие как журналирование и другие инструменты. Именно это делает декораторы такой мощной функцией в Python, которая часто используется в стандартной библиотеке и в пакетах сторонних разработчиков.

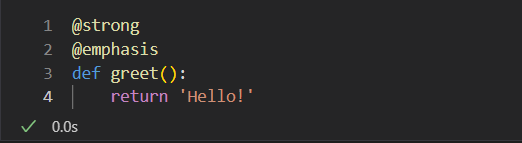
### **Применение нескольких декораторов к одной функции**

Возможно, неудивительно, что к функции можно применить более одного декоратора. Это накапливает их эффекты, и именно это делает декораторы настолько полезными, как повторно используемые модули.

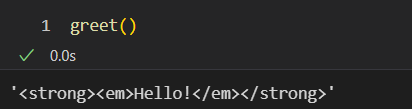
Вот пример. Следующие два декоратора оборачивают выводимую строку декорируемой функции в HTML-теги. Посмотрев на то, как вложены теги, можно увидеть, в каком порядке Python применяет несколько декораторов:



Теперь давайте возьмем эти два декоратора и применим их к нашей функции greet одновременно. Для этого можно использовать обычный синтаксис @ и просто «уложить» несколько декораторов поверх одной функции:



Какой результат вы ожидаете увидеть, если запустите декорированную функцию? Будет ли декоратор @emphasis первым добавлять свой тег <em> или @strong имеет приоритет? Вот что происходит, когда вы вызываете декорированную функцию:



Здесь хорошо видно, в каком порядке применялись декораторы: *снизу вверх*. Сначала входная функция была обернута декоратором @emphasis , а затем результирующая (декорированная) функция была снова обернута декоратором @strong.

Чтобы запомнить этот порядок снизу вверх, мне нравится называть такое поведение «стеком декораторов». Вы начинаете строить стек снизу, а затем продолжаете добавлять новые блоки сверху, чтобы проделать путь наверх.

Если разбить приведенный выше пример и не использовать синтаксис @ для применения декораторов, то цепочка вызовов функций декораторов выглядит следующим образом:

**decorated\_greet = strong(emphasis(greet))**

Здесь снова видно, что сначала применяется декоратор emphasis, а затем полученная обернутая функция снова оборачивается декоратором strong.

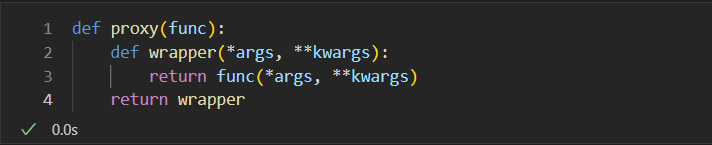
Это также означает, что глубокие уровни наложения декораторов в конечном итоге будут влиять на производительность, поскольку они продолжают добавлять вызовы вложенных функций. Обычно это не является проблемой на практике, но об этом следует помнить, если вы работаете над кодом, требующим высокой производительности.

### **Декорирование функций, принимающих аргументы**

Все примеры до сих пор декорировали только простую нульарную функцию greet, которая не принимала никаких аргументов. Поэтому декораторы, которые вы видели здесь до сих пор, не имели дела с передачей аргументов во входную функцию.

Если вы попытаетесь применить один из этих декораторов к функции, принимающей аргументы, он будет работать неправильно. Как декорировать функцию, принимающую произвольные аргументы?

Здесь на помощь приходит функция Python \*args *и* \*\*kwargsдля работы с переменным количеством аргументов. Декоратор proxy использует эту возможность:

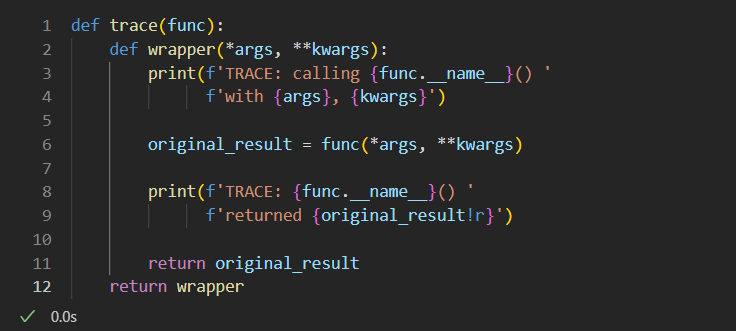


В этом декораторе есть два примечательных момента:

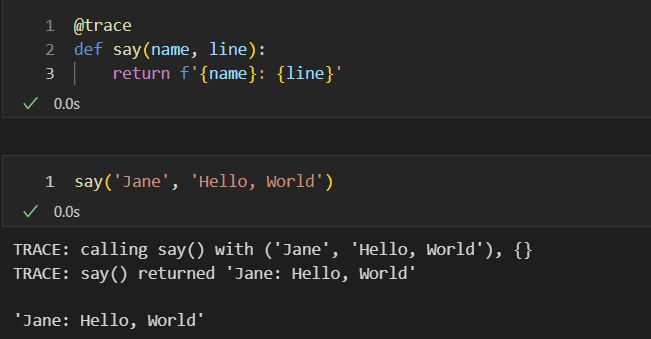
* Он использует операторы \* и \*\* в определении замыкания wrapper для сбора всех позиционных и ключевых аргументов и хранения их в переменных (args и kwargs).
* Затем замыкание wrapper передает собранные аргументы исходной входной функции с помощью операторов «распаковки аргументов» \* и \*\*.

(*Немного жаль, что значение операторов "звездочка" и "двойная звездочка" перегружено и меняется в зависимости от контекста, в котором они используются. Но я надеюсь, что вы поняли идею.*)

Давайте расширим технику, заложенную в декораторе proxy, на более полезный практический пример. Вот декоратор trace, который выводит аргументы и результаты функции во время ее выполнения:



Декорировав функцию с помощью trace и вызвав ее, можно вывести аргументы, переданные декорированной функции, и ее возвращаемое значение. Это все еще в некоторой степени игрушечный пример, но в крайнем случае он станет отличным подспорьем для отладки:

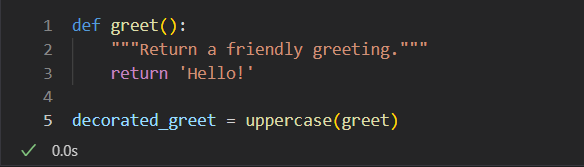


Кстати, об отладке — есть несколько моментов, которые следует иметь в виду при отладке декораторов.

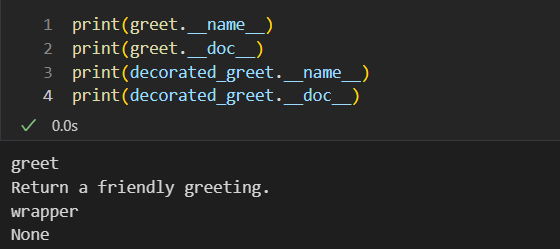
### **Как писать «отлаживаемые» декораторы**

Когда вы используете декоратор, на самом деле вы заменяете одну функцию другой. Одним из недостатков этого процесса является то, что он «скрывает» некоторые метаданные, прикрепленные к исходной (недекорированной) функции.

Например, оригинальное имя функции, ее документационная строка (docstring) и список параметров скрываются замыканием:

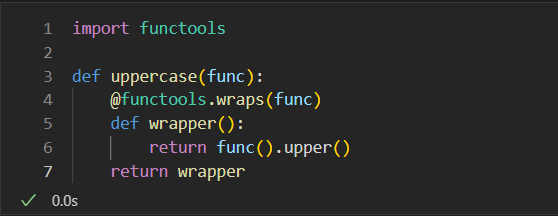


Если вы попытаетесь получить доступ к любым метаданным этой функции, то вместо них вы увидите метаданные замыкания wrapper:



Это делает отладку и работу с интерпретатором Python неудобной и сложной. К счастью, для этого есть быстрое решение: декоратор functools.wraps, включенный в стандартную библиотеку Python.

Вы можете использовать functools.wraps в своих собственных декораторах, чтобы скопировать потерянные метаданные из недекорированной функции в замыкание декоратора. Вот пример:



Применение functools.wraps к замыканию обертки, возвращаемому декоратором, переносит документационную строку и другие метаданные входной функции:



Я бы рекомендовал использовать functools.wraps во всех декораторах, которые вы пишете сами. Это не займет много времени и избавит вас (и других) от головной боли при отладке в будущем.

# **Понятие инкапсуляции. Применение инкапсуляции.**

Значение термина «инкапсуляция» расплывчато и отличается от источника к источнику. Принято считать, что инкапсуляция — один из основополагающих принципов ООП, хотя некоторые научные статьи вовсе упускают инкапсуляцию из списка. К примеру, Джон Митчелл в книге «Концепты в языках программирования» при перечислении основных концептов в ООП упоминает только абстракцию — термин который принято считать близким к инкапсуляции по значению, но все-же более обширным и высокоуровневым. С другой стороны, Роберт Мартин в его книге «Чистая архитектура» явно говорит о том, что инкапсуляция, наследование и полиморфизм считается фундаментом ООП.

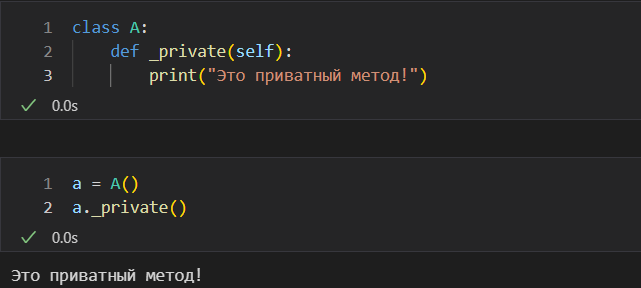
Разнообразие определений, данных термину «инкапсуляция», сложно привести к общему знаменателю. В целом можно выделить два подхода к значению этого термина. Инкапсуляция может быть рассмотрена как:

* связь данных с методами которые этими данными управляют;
* набор инструментов для управления доступом к данным или методам которые управляют этими данными.

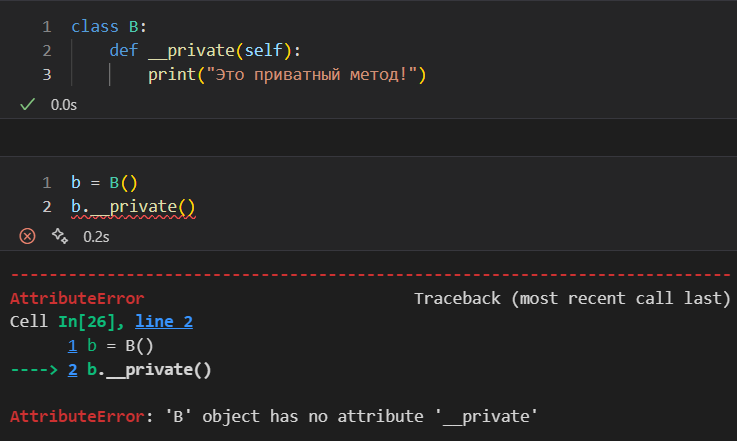
Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

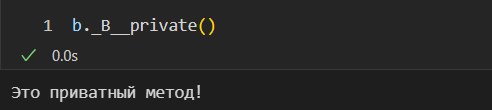
Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.



Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.



Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем \_ИмяКласса\_\_ИмяАтрибута:



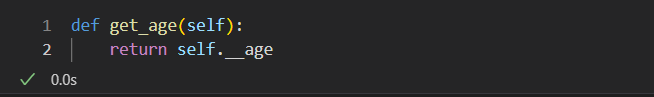
# **Методы и атрибуты. Геттеры и сеттеры. Модификаторы доступа.**

### **Геттеры и сеттеры**

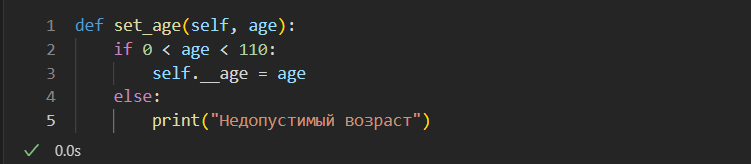
Может возникнуть вопрос, как обращаться к подобным приватным атрибутам. Для этого обычно применяются специальные методы доступа. Геттер позволяет получить значение атрибута, а сеттер установить его. Так, изменим выше определенный класс, определив в нем методы доступа:

|  |  |
| --- | --- |
|  |  |

Для получения значения возраста применяется метод get\_age:



Для изменения возраста определен метод set\_age:



Причем опосредование доступа к атрибутам через методы позволяет задать дополнительную логику. Так, в зависимости от переданного возраста мы можем решить, надо ли переустанавливать возраст, так как переданное значение может быть некорректным.

Также необязательно создавать для каждого приватного атрибута подобную пару методов. Так, в примере выше имя человека мы можем установить только из конструктора. А для получение определен метод get\_name.

### **Аннотации свойств**

Выше мы рассмотрели, как создавать методы доступа. Но Python имеет также еще один - более элегантный способ - свойства. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация @property.

Для создания свойства-сеттера над свойством устанавливается аннотация имя\_свойства\_геттера.setter.

Перепишем класс Person с использованием аннотаций:



Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение tom.age.

При этом можно определить только геттер, как в случае с свойством name - его нельзя изменить, а можно лишь получить значение.

# **Заключение**

Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования, которая обеспечивает безопасность данных и удобство их использования. В ходе этой лекции мы узнали, как декораторы помогают нам создавать геттеры и сеттеры, позволяя контролировать доступ к атрибутам объектов. Мы рассмотрели основные принципы инкапсуляции и преимущества, которые она предоставляет, такие как защита данных и улучшение читаемости кода.

Мы также познакомились с использованием модификаторов доступа, таких как \_ и \_\_, для ограничения доступа к атрибутам и методам класса, что помогает скрывать внутреннюю реализацию и защищать данные от несанкционированного доступа. Геттеры и сеттеры позволяют нам создавать интерфейс для взаимодействия с приватными данными, обеспечивая контроль над их изменением и доступом.

Инкапсуляция позволяет разработчикам создавать более устойчивые и удобные для поддержки программы. Используя инкапсуляцию, вы можете защищать данные, обеспечивать их целостность и улучшать структуру вашего кода. Этот принцип является важной частью ООП, и его понимание и применение сделают ваш код более качественным и надежным.

# **Список литературы**

1. https://habr.com/ru/companies/otus/articles/727590/
2. https://habr.com/ru/articles/444338/
3. https://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html
4. https://sky.pro/wiki/python/inkapsulyaciya-v-python/
5. https://sky.pro/wiki/python/oop-v-python-inkapsulyaciya/
6. https://proglib.io/p/samouchitel-po-python-dlya-nachinayushchih-chast-18-osnovy-oop-inkapsulyaciya-i-nasledovanie-2023-04-17
7. https://younglinux.info/oopython/encapsulation
8. https://metanit.com/python/tutorial/7.2.php
9. https://ya.zerocoder.ru/pgt-ponimanie-inkapsulyacii-v-python/
10. https://blog.skillfactory.ru/glossary/inkapsulyacziya/
11. John C. Mitchell, Concepts in programming languages
12. Robert C. Martin, Clean Architecture, A Craftsman’s Guide to Software Structure and Design