

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ Python

Урок 9

Вступ до ООП

Зміст

1. Вступ до ООП.....	4
1.1. Поняття ООП.....	4
1.2. Поліморфізм	11
2. Типи даних, що визначаються користувачем.....	13
3. Специфікатори доступу.....	22
3.1. Рівень доступу Public (загальний)	22
3.2. Рівень доступу Private (приватний)	24
3.3. Рівень доступу Protected (внутрішній або захищений)	27
4. Успадкування та інкапсуляція.....	28
4.1. Методи: загальний, внутрішній (захищений), приватний	37
4.2. Рівень доступу Public (загальний)	38
4.3. Рівень доступу Private (приватний)	39

4.4. Рівень доступу Protected (внутрішній або захищений)	43
4.5. Статичні методи і методи класу.....	47
4.6. Статичні методи.....	49
4.7. Методи класу	54
4.8. Множинне успадкування та MRO (порядок вирішення методів)	63
5. Поліморфізм.....	75
5.1. Перевантаження операторів.....	77
5.2. Magic-методи, конструктори	79
5.3. Реалізація магічних методів.....	81
6. Створення і керування поведінкою екземплярів класу	88
6.1. Функтори	88
6.2. Декоратори	96
6.3. Керовані атрибути об'єкта — property()	103
6.4. Властивості — property()	113
6.5. Дескриптори	121
7. Метакласи.....	133
7.1. Модель метакласів.....	133
7.2. Метод-конструктор __new__().....	140
7.3. Протоколи в Python	148

1. Вступ до ООП

1.1. Поняття ООП

Раніше ми з вами вже знайомилися з концепцією функціонального програмування, в якому основна логіка завдання (алгоритм) представлена набором функцій. І кожна з таких функцій реалізує окремі кроки загального алгоритму.

Поки ми реалізуємо прості завдання і нам достатньо наявних (вбудованих) типів даних, функціонального підходу достатньо. Однак у проєктах для предметних областей зі складною структурою, в якій потрібно обов'язково враховувати особливості взаємодії (зв'язку) між її об'єктами, нам потрібна інша методологія. Потрібен такий підхід, який дозволить описати зв'язок між даними та функціями їх обробки (що неможливо у функціональному програмуванні). Наприклад, [функція_1](#) має працювати тільки з даними одного типу, а [функція_2](#) лише з іншими. І більше того, нам потрібно створити таку [функцію_3](#), яка з даними першого типу працює за одним алгоритмом, а для даних другого типу надає зовсім іншу функціональність.

Розглянемо цю ситуацію з прикладу. Припустимо, що нам необхідно оцінити рівень успішності за критеріями «відмінно», «добре», «задовільно», «незадовільно» для учня та студента. При цьому у студента оцінка варіюється в діапазоні від [0](#) до [100](#), а у учня — від [1](#) до [12](#), тобто діапазони критеріїв успішності ніяк не співпадають. Для студента «відмінно» — це бал від [90](#) до [100](#),

а для учня — від 10 до 12. Крім того, студентські бали зі значеннями вище 12 взагалі є помилковими даними для студента.

При використанні функціонального підходу нам потрібно як мінімум створювати дві окремі функції з різними іменами.

```
def checkStudentSuccess(name, score):
    if score >= 90:
        print("{} has excellent level".format(name))
    elif 75 <= score < 90:
        print("{} has good level".format(name))
    elif 60 <= score < 75:
        print("{} has average level".format(name))
    else:
        print("{} has poor level".format(name))

def checkPupilSuccess(name, score):
    if score >= 10:
        print("{} has excellent level".format(name))
    elif 7 <= score < 10:
        print("{} has good level".format(name))
    elif 4 <= score < 7:
        print("{} has average level".format(name))
    else:
        print("{} has poor level".format(name))

checkStudentSuccess("Jane", 78)
checkPupilSuccess("Bob", 6)
```

А якщо потім в завданні додасться ще й учень молодших класів, у якого оцінки від 1 до 5? Тоді доведеться створити ще одну, третю функцію.

А як взагалі перевірити, чиї це бали: Студента чи учня? За ім'ям «Jane» або «Bob» це точно не визначити. А це

потрібно знати, щоб визначити, яку функцію викликати в програмі: `checkStudentSuccess()` або `checkPupilSuccess()`? Тобто нам потрібен зв'язок «об'єкт — його дані — його функціональність». Такий тип зв'язку даних та поведінки, функціональне програмування не може забезпечити простим та прозорим способом.

Розглянемо ще один приклад: припустимо, що нам потрібно реалізувати заповнення нового списку на основі елементів існуючого списку. Є два списки: один містить логіни користувачів, і потрібно перевести їх у нижній регістр, а другий — роки народження користувачів, і потрібно розрахувати їх вік (наприклад, для обмеження доступу до деякого контенту).

Звісно, що нам доведеться створити дві окремі функції, кожна з яких приймає список як вхідний аргумент і на основі його даних заповнює результуючий список. Тобто зовні функції виглядають дуже схожими і повертають однаковий результат — список, причому навіть однакової довжини (адже кількість років народження у другому списку збігається з кількістю користувачів у першому). Проте, алгоритм роботи функцій повністю залежить від «змісту» її вхідних даних.

```
userLogs=['Admin123','superUSER','GOODstudent']
userBYears=[2000, 2010, 2005]

def listMaker1(myList):
    result=[]
    for item in myList:
        result.append(item.lower())
    return result
```

```
def listMaker2(myList):
    result=[]
    for item in myList:
        result.append(2022-item)
    return result

newList1=listMaker1(userLogs)
newList2=listMaker2(userBYears)
print(newList1)
print(newList2)
```

Результат:

```
['admin123', 'superuser', 'goodstudent']
[22, 12, 17]
```

Рисунок 1

Як бачимо, правильність використання потрібної функції залежить від уважності та відповідальності розробника. Адже від випадкової помилки такого типу:

```
newList1=listMaker1(userBYears)
newList2=listMaker2(userLogs)
```

коли ми просто сплутали аргументи функцій

AttributeError ×

Рисунок 2

'int' object has no attribute 'lower'

Рисунок 3

розробник від цього не «застрахований».

А коли таких ситуацій (функцій) стає багато (що є природним для реальних проєктів), програміст може заплутатися в цій масштабній неоднозначності (навіть за умови зрозумілих імен функцій та змінних). Однак функціональне програмування не є єдиним підходом до розробки програмного забезпечення. Не менш популярною, і навіть більш поширеною, є парадигма об'єктно-орієнтованого програмування (ООП). Її популярність також викликана тим фактом, що використання об'єктно-орієнтованого програмування дозволяє практично однозначно відобразити сутність (елементи) реального світу (предметної галузі, завдання) в структурі програми. При цьому зберігаються особливості їх структури, зв'язку (залежності) між ними та деталі їх поведінки. Можна сказати, що ООП допомагає нам розробляти програму з таким вмістом, який найбільш схожий на наше звичне уявлення про завдання (фрагмент реального світу).

Об'єктно-орієнтоване програмування (ООП) — це такий підхід до розробки програм (парадигма програмування), в якому окремі компоненти системи (програми) представлені об'єктами. Звичайно, основними концепціями ООП є поняття об'єкта та класу.

Більшість сутностей (компонентів, частин) будь-якої предметної області — це об'єкти. Наприклад, клієнт банку, який має ім'я, прізвище, вік і ще якісь характеристики, є об'єктом реального світу. Клієнт банку може відкрити рахунок, перевірити стан рахунку, поповнити рахунок і т. д., тобто у нього є поведінка (що відображає ті завдання, які він має вирішувати у цій предметній галузі). За допомогою ООП ми можемо відобразити (виконати моделювання)

об'єкт із реального світу до програмного об'єкта, який має певні дані і який може виконувати певні функції. І це ще не все: ООП дозволяє також моделювати відносини між об'єктами (наприклад, між компаніями та її співробітниками, студентами та викладачами і т. д.).

Таке загальне уявлення, яке включає інформацію про характеристики об'єкта і про особливості (алгоритми) його дій ми будемо називати класом.

Клас — це певний шаблон (макет), що описує структуру та можливу поведінку об'єкта. Можна сказати, що клас — це схема (креслення), використовуючи яку можна створити конкретний об'єкт.

Конкретна реалізація (втілення) такого шаблону — це об'єкт (названий також екземпляром класу). Один об'єкт може відрізнятися від іншого також, як один клієнт банку відрізняється від іншого: прізвищем, віком і т. д.

Розглянемо з прикладу тварини — кішки. У кожної кішки є голова, лапи, хвіст, вуха, шерсть, тобто описуючи якусь окрему породу кішки, нам необхідно описувати ці характеристики. Також кожна кішка вміє нявкати, стрибати, сидіти. Таким чином, кішка — це клас з названими властивостями та зазначеною поведінкою.

А от сіамська кішка, у якої голова нагадує трикутник, що звужується прямими лініями до витонченої морди, вуха крупні й загострені, це об'єкт (тобто екземпляр класу «[Кішка](#)»).

Таким чином, клас — це користувацький тип даних, що описує те, які властивості й поведінку будуть мати змінні цього типу. А об'єкт — це змінна класу, тобто екземпляр з конкретним значенням цих властивостей.

Більш детально поняття «клас» та «об'єкт» ми обговоримо трохи пізніше. А зараз розглянемо основні засади (так звані стовпи) ОО-парадигми.

Інкапсуляція — одна з фундаментальних концепцій об'єктно-орієнтованого програмування, яка дозволяє приховувати деталі внутрішньої реалізації об'єктів. Можна сказати, що згідно з цим принципом, клас має розглядатися, як якийсь чорний ящик. Користувач не знає, що знаходиться всередині (особливості реалізації), і взаємодіє з ним тільки за допомогою наданого інтерфейсу.

Розглянемо з прикладу. Для того, щоб водити сучасний автомобіль з автоматичною коробкою передач, не потрібно знати, як влаштований і працює його двигун, що таке бензонасос, де розташована система охолодження двигуна. Всі ці особливості та дії окремих механізмів автомобіля приховані від водія і, в той же час дозволяють йому крутити кермо і натискати на педалі газу або гальма, не замислюючись, що в цей час відбувається під капотом.

Таке приховування внутрішнього пристрою та процесів, що реалізують роботу автомобіля, гарантують простоту його використання та ефективність керування. Навіть для водіїв, які не мають значного досвіду водіння та не є професійними автомеханіками.

Таким чином, інкапсуляція забезпечує безпеку внутрішнього вмісту класу. Адже коли ми у такий спосіб приховуємо від користувача внутрішній пристрій об'єкта, ми забезпечуємо його надійну роботу.

Успадкування — це можливість використовувати дані та функціональність чогось вже існуючого для створення нового (на цій існуючій основі).

У контексті ООП, успадкування — це здатність одного класу отримувати (успадкоувати) властивості та поведінку іншого класу. Успадкування є основним механізмом повторного використання коду в ОО-парадигмі.

Похідний (дочірній) клас — це клас, який успадковує властивості іншого класу, розширюючи його функціональність та/або характеристики.

Базовий (батьківський) клас — це клас, властивості та поведінка якого успадковуються.

Уявімо, що ми є працівниками цеху з виробництва меблів. Наш цех виготовляє стандартні стільці з жорстким сидінням, які добре зарекомендували себе у покупців. Але надійшло замовлення на випуск стільців з м'яким сидінням. Звичайно, раціональним рішенням є використання як основа для нової моделі вже наявної моделі стандартного стільця, яка користується популярністю у клієнтів. До того ж, наше обладнання вже повністю налаштоване на випуск таких стандартних стільців. Тому після виготовлення стандартного стільця ми додамо м'яку оббивку сидіння, тобто наша модифікація буде мати більшу частину властивостей колишньої моделі.

У цьому прикладі стандартний стілець є базовим класом, а стілець з м'яким сидінням — похідним, який успадковує всі характеристики батьківського класу та додає нову властивість — м'яку оббивку сидіння.

1.2. Поліморфізм

У буквальному значенні термін поліморфізм буквально позначає наявність кількох форм одного й того ж. В ОО-парадигмі поліморфізм надає можливість використовувати

той самий інтерфейс функції (з різними особливостями поведінки) для різних об'єктів.

Розглянемо поняття «поліморфізму» на прикладі кнопки — вимкнення/ввімкнення для пральної машини, пульта кондиціонера та електрочайника. На всіх перерахованих пристроях кнопка виглядає (реалізована) по-різному (хоча зображення на ній для пральної машини та пульта кондиціонера може бути схожим). При цьому результат її натискання той самий прилад включається і починає працювати відповідно до своєї програми функціонування.

У програмуванні поліморфізм реалізується за допомогою механізму навантаження методу класу. Ми вже знайомі з перевантаженням функцій, що дозволяє створювати кілька функцій з однаковими іменами, але різним набором параметрів та різною логікою поведінки. При цьому інтерпретатор сам визначає, який код працюватиме, залежно від кількості чи типу аргументів під час виклику функції.

2. Типи даних, що визначаються користувачем

Як ми обговорювали раніше, використання класів дозволить нам зберігати інформацію про сутності предметної галузі (особливості структури, значення характеристик, деталі поведінки) відповідно до їх уявлення у реальному світі. Тому можна сказати, що класи використовуються для створення власних структур даних, описують будову об'єктів предметної області.

Особливості структури (які характеристики і властивості має об'єкт) описуються переліком атрибутів класу (змінними, визначеними всередині класу та належними йому). А можлива поведінка — методами класу, які фактично є функціями (включеними до складу класу), логіка (алгоритм) яких реалізує особливості цієї поведінки.

Як ми пам'ятаємо, клас — це тільки макет для створення об'єктів (примірників класу), який, в основному, не містить конкретних значень. Розглянемо загальний синтаксис для створення класу в Python:

```
class className:
    <classStatement_1>
    <classStatement_2>
    ...
    <classStatement_N>
```

Давайте створимо клас, який представляє студента:

```
class Student:
    pass
```

Зазвичай, ім'я класу завжди пишеться з великої літери (**Student**).

Поки що у класі **Student** не визначено атрибутів та методів. Однак, згідно з правилами синтаксису Python, при створенні класу має бути щось визначене. Тому ми використали оператор **pass**, який часто використовується як заповнювач програми там, де пізніше буде додаватися потрібний фрагмент коду. Такий підхід позбавить нас зараз від виникнення помилки (згідно з синтаксисом Python після рядка оголошення класу, що закінчується символом «:», необхідно описати його склад, який може бути невідомий).

Тепер давайте спробуємо додати до нашого класу кілька характеристик студента, наприклад, вік та ім'я. Адже кожен студент має ім'я та вік. Однак синтаксис Python не дозволяє просто перерахувати назви характеристик (імена змінних) всередині класу, для них потрібно встановити значення. Наприклад, так:

```
class Student:
    age=20
    name="Bob"
```

Однак, як ми пам'ятаємо, клас — це тільки макет для створення об'єкта, який представляє конкретний екземпляр класу (конкретний студент у нашому прикладі).

Після того як ми визначили клас, ми можемо створювати об'єкти цього класу.

Для створення об'єкта (примірника класу) використовується спеціальна функція — конструктор класу (яка є за замовчуванням у будь-якому класі, і яка не має параметрів). Для запуску цієї функції (створення об'єкта) нам потрібно вказати ім'я нашого класу з наступними круглими дужками. В результаті роботи такого конструктора класу ми отримуємо об'єкт (екземпляр даного класу).

```
student1=Student()
```

Таким чином ми створили об'єкт `student1`, який є екземпляром класу `Student`. У прикладі вище ми визначили у класу `Student` два атрибути `age` та `name`.

Для доступу до атрибуту необхідно вказати ім'я об'єкта, після символу поставити крапку і вказати ім'я потрібного атрибута:

```
print(student1.name)
```

Ці атрибути входять до складу кожного екземпляра класу `Student` разом зі своїми значеннями, тобто у всіх екземплярів класу значення атрибута `age` буде `20`, а значення атрибута `name` — «*Bob*».

Давайте створимо ще один об'єкт (ще одного студента), а потім виведемо вік та ім'я в обох:

```
class Student:
    age=20
    name="Bob"
```

```

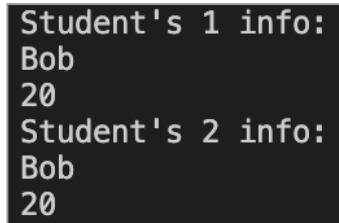
student1=Student()
student2=Student()

print("Student's 1 info:")
print(student1.name)
print(student1.age)

print("Student's 2 info:")
print(student2.name)
print(student2.age)

```

Результат:



```

Student's 1 info:
Bob
20
Student's 2 info:
Bob
20

```

Рисунок 4

Однак ця ситуація нам не підходить: у кожного студента мають бути свої значення віку та імені.

Для того, щоб створювати об'єкт із заданими характеристиками, потрібно включити до складу класу власну функцію-конструктор. Як нам відомо, всі класи за замовчуванням мають функцію-конструктор. Ім'я цієї функції `__init__()` і вона не має параметрів (знову ж таки, за замовчуванням). Ми будемо використовувати її для присвоєння значень властивостям об'єкта, вказавши їх імена як параметри.

Основна відмінність властивостей від атрибутів у тому, що властивості можуть бути різні у різних об'єктів

(такі, як ми зазначимо у момент створення). А от значення атрибутів (як ми вже показали у нашому прикладі) будуть однаковими для всіх.

Аналізуючи наш приклад, легко зрозуміти, що такі характеристики студента, як його ім'я та вік не можуть бути атрибутами класу, а є його властивостями.

Додаймо до класу конструктор `__init__()`, щоб надати значення властивостям `age` та `name`. А загальною характеристикою для всіх студентів може бути, наприклад, назва спеціальності, за якою вони навчаються. Цю характеристику, значення якої у всіх однакове (`spec="Computer science"`), потрібно додати до класу у вигляді атрибуту:

```
class Student:
    spec="Computer science"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Ви можете вказати в `__init__()` будь-яку кількість параметрів (майбутніх властивостей об'єкта), але першим параметром завжди буде змінна з ім'ям `self`. Коли створюється новий екземпляр класу, він автоматично передається параметру `self` в `__init__()`, щоб для об'єкта можна було визначити нові властивості. Фактично, змінна `self` — це посилання на поточний об'єкт. Саме використовуючи її, ми можемо всередині класу звертатися до його атрибутів та властивостей.

Примітка: заголовок методу `__init__()` у нашому прикладі має відступ у чотири пробіли від початку рядка, а його тіло — вісім пробілів. Таке

зміщення важливе, оскільки таким чином ми повідомляємо про те, що метод `__init__()` належить класу `Student`.

У нашому прикладі, в тілі методу `__init__()`, ми використовували параметр `self` у двох інструкціях для створення властивостей об'єкта (`name` та `age`) та встановлення їх значень на основі значень другого та третього параметрів методу:

- команда `self.name = name` створює атрибут з ім'ям `name` і надає йому значення другого параметра методу — `name`;
- команда `self.age = age` створює атрибут з ім'ям `age` і надає йому значення третього параметра методу — `age`.

Як ми пам'ятаємо, конструктор створює екземпляр класу (об'єкт), тому значення властивостей, визначені за допомогою параметрів конструктора, будуть відноситися до конкретного об'єкту. Таким чином, всі екземпляри класу `Student` матимуть властивості `name` та `age`, але їх значення у кожного об'єкта будуть свої — такі, які ми вкажемо в момент створення об'єкта:

```
student1=Student("Bob",20)
student2=Student("Jane", 18)

print(student1.showInfo())
print(student1.showMsg("Hello!"))

print(student2.showInfo())
print(student2.showMsg("Hi!"))

print("Student's 1 info:")
```

```
print(student1.name)

print(student1.age)
print(student1.spec)

print("Student's 2 info:")
print(student2.name)

print(student2.age)
print(student1.spec)
```

Результат:

```
Student's 1 info:
Bob
20
Computer science
Student's 2 info:
Jane
18
Computer science
```

Рисунок 5

Кожен об'єкт має такі характеристики:

- **Стан:** представлений атрибутами об'єкта і також відбиває властивості об'єкта.
- **Ідентичність:** кожен екземпляр класу має унікальне ім'я (унікальне ім'я змінної, яку ми створюємо в момент створення самого об'єкта), яке дозволяє одному об'єкту взаємодіяти з іншими об'єктами.
- **Поведінка:** показує можливу функціональність об'єкта (також відбиває реакцію об'єкта на інші об'єкти), представлена методами класу.

З першими двома характеристиками об'єкта ми вже познайомилися. Тепер розглянемо особливості та процес створення методів класу.

Методи класу — це функції, які визначені всередині класу і можуть бути викликані лише з екземпляра цього класу. Як і в методі — конструкторі `__init__()`, при визначенні будь-якого методу класу першим параметром завжди потрібно вказувати [self](#). При цьому, при виклику методу, ми не надаємо йому значення, оскільки його (посилання на поточний об'єкт) автоматично надає сам інтерпретатор Python.

Таким чином, коли ми викликаємо певний метод `methodName` об'єкта `myObject` подібним рядком коду:

```
myObject.methodName(argName1, argName2)
```

вона автоматично перетворюється Python на таку:

```
myClass.methodName(myObject, argName1, argName 2).
```

Додаймо до нашого класу [Student](#) два методи: перший для виведення інформації про студента (значень атрибутів і властивостей об'єкта), а другий для висновку потрібного тексту привітання.

```
class Student:

    spec="Computer science"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
def showInfo(self):  
    return f"Student {self.name}  
           is {self.age} years old."  
  
def showMsg(self, msgText):  
    return f"Student {self.name}  
           says '{msgText}'."  
  
student1=Student("Bob",20)  
student2=Student("Jane", 18)  
  
print(student1.showInfo())  
print(student1.showMsg("Hello!"))  
  
print(student2.showInfo())  
print(student2.showMsg("Hi!"))
```

Результат:

```
Student Bob is 20 years old.  
Student Bob says 'Hello!'.  
Student Jane is 18 years old.  
Student Jane says 'Hi!'.
```

Рисунок 6

3. Специфікатори доступу

В об'єктно-орієнтованих мовах програмування використовуються специфікатори (модифікатори) доступу, які дозволяють встановлювати рівень (обмеження) доступу до властивостей та методів класу.

Більшість мов програмування (і Python також) мають три види модифікаторів доступу: **public** (загальний), **protected** (внутрішній або захищений) та **private** (приватний).

3.1. Рівень доступу Public (загальний)

Усі компоненти (члени) класу, оголошені як **public**, доступні з будь-якої частини програми. В деяких мовах програмування, наприклад, в C# і Java, необхідно явно вказувати вид модифікаторів доступу. У Python усі властивості та методи класу є загальними за умовчанням, тобто явно вказувати ключове слово «**public**» нам не потрібно.

У нашому класі **Person** усі три властивості (ім'я, прізвище та вік) мають рівень доступу **public**. Тому вони доступні всередині класу (використовуються в методах **getInfo()** і **getHi()**), і за його межами. Давайте змінимо значення властивості «вік» об'єкта, щоб перевірити це:

```
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
```

```

        self.age = age

    # public methods
    def getInfo(self):
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."

person1=Person("Joe", "Black", 30)
print(person1.getInfo())
person1.age=35
print(person1.getInfo())

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Person first name - Joe; last name - Black; age - 35.
Hi! I am Joe.

```

Рисунок 7

Як бачимо, зміна віку у *Joe* пройшло вдало. Повторний виклик методу `getInfo()` показав вже оновлене значення якості.

Public-метод може викликатися всередині інших методів цього класу. Вище ми показали виклик методів класу ззовні (з допомогою його об'єктів). Тепер додаєм до методу `getHi()` виклик методу `getInfo()`:

```

class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName

```

```

        self.lastName = lastName
        self.age = age

    # public methods
    def getInfo(self):
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."

    def getHi(self, msgText):
        personInf=self.getInfo()
        return f"{personInf}\n{msgText}! I am
                {self.firstName}."

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.
Hi! I am Joe.

```

Рисунок 8

Тепер під час виклику методу `getHi()` виводиться вся інформація про людину (а ми показали доступність public-методу всередині інших методів класу).

3.2. Рівень доступу Private (приватний)

Приватні члени класу (властивості або методи) доступні лише всередині класу. Ми не можемо використовувати private-властивості або викликати private-методи за межами класу. Реалізація private-рівня доступу властивості або методу класу забезпечується подвійним символом підкреслення «`__`», який потрібно помістити безпосередньо перед ім'ям властивості (без пропуску), наприклад `__privateProp`, `__privateMethod`.

Додамо до базового класу `Person` private-властивість `personID`, що генерується діапазоном від 1 до 100 всередині конструктора, та private-метод `getID()`.

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __getID(self):
        return f"{self.__personID}\n"

    # public methods
    def getInfo(self):
        return f"{self.__getID()}Person first name - {self.firstName}; last name - {self.lastName}; age - {self.age}."

    def getHi(self, msgText):
        personInf=self.getInfo()
        return f"{personInf}\n{msgText}! I am {self.firstName}."

person1=Person("Joe", "Black", 30)
print(person1.getInfo())
```

Результат:

```
72
Person first name - Joe; last name - Black; age - 30.
```

Рисунок 9

Як бачимо, всередині класу (у тілі методу `getID()`) ми маємо доступ до `private`-властивості `personID`. І виклик `private`-методу `getID()` також можливий всередині класу (ми виконали його в тілі методу `getInfo()`).

Давайте тепер спробуємо вивести значення `personID` і викликати метод `getID()` використовуючи об'єкт `person1` (тобто перевіримо доступ до `private`-властивості та `private`-методом за межами класу, в якому вони визначені).

Рядок коду:

```
print(person1.__personID)
```

ВИКЛИЧЕ ВИНЯТОК

AttributeError ✕

Рисунок 10

з причини:

'Person' object has no attribute '__personID'

Рисунок 11

Така ж ситуація буде й при спробі викликати `private` — метод за межами класу:

```
person1.__getID()
```

Виникнення винятка:

AttributeError ✕

Рисунок 12

з причини:

'Person' object has no attribute '__showID'

Рисунок 13

Спроба змінити значення private-властивості `personID` не викличе виняток

```
person1=Person("Joe", "Black", 30)
print(person1.getInfo())

person1.__personID=100
print(person1.getInfo())
```

однак не дасть результату: значення `personID` не зміниться:

```
30
Person first name - Joe; last name - Black; age - 30.
30
Person first name - Joe; last name - Black; age - 30.
```

Рисунок 14

Таким чином, private-властивості та методи дійсно доступні лише зі свого класу.

3.3. Рівень доступу Protected (внутрішній або захищений)

Якщо нам треба обмежити доступ до властивостей та методів таким чином, щоб їх було «видно» тільки всередині самого класу або класів, похідних від нього, тоді нам потрібен рівень доступу `Protected`. Особливості створення protected-властивостей та методів ми розглянемо трохи пізніше — після базових основ з реалізації успадкування в Python.

4. Успадкування та інкапсуляція

Як ми вже знаємо, успадкування — це здатність одного класу отримувати (успадковувати) властивості та поведінку іншого класу. У цьому механізмі ми завжди маємо базовий та похідний клас (або декілька таких).

Похідний (дочірній) клас — це клас, який успадковує властивості іншого класу, розширюючи його функціональність та/або характеристики.

Базовий (батьківський) клас — це клас, властивості та поведінка якого успадковуються.

Уявімо, що нам необхідно розробити такі типи транспортних засобів, як автобус, вантажівка та автомобіль-седан. Кожен з перерахованих автомобілів має чотири колеса, кермо, педаль газу. Кожен автомобіль надає можливості (функціональність) перегляду витрат палива, гальмування, поворот передніх коліс відповідно до положення керма.

Автобус	Вантажівка	Седан
Колеса Двигун Кермо	Колеса Двигун Кермо	Колеса Двигун Кермо
Перегляд витрат палива() Поворот коліс() Гальмування()	Перегляд витрат палива() Поворот коліс() Гальмування()	Перегляд витрат палива() Поворот коліс() Гальмування()

Рисунок 15

Якщо нам необхідно виконати моделювання (реалізувати) усі перелічені властивості та поведінку програмно, тоді доведеться написати усі ці функції у кожному з трьох класів. Зробимо це за такою схемою (рис. 15).

Як бачимо, нам фактично потрібно виконати дублювання коду три рази. Крім проблеми дублювання коду (надмірної кількості даних) також збільшується ймовірність помилок при реалізації.

Тепер подивимося, як механізм успадкування дозволить уникнути такої ситуації.

Спочатку ми створимо базовий клас «Транспортний засіб», який має характеристики: «колеса», «кермо», «двигун», і три перелічені функції (поведінки). Далі створюємо інші класи: «Автобус», «Вантажівка», «Седан», й успадковуємо від класу «Транспортний засіб».



Рисунок 16

Як легко можна помітити, ми можемо просто уникнути дублювання даних та підвищити можливість повторного використання. Тепер розглянемо, як механізм успадкування можна реалізувати, використовуючи засоби мови Python. Будь-який клас може бути батьківським, тому синтаксис створення батьківського класу аналогічний до створення будь-якого іншого.

Давайте створимо базовий клас «Людина», яка має такі властивості: ім'я, прізвище, вік. Також визначимо у ньому два методи: «Привітатись», який виводить повідомлення «Привіт! Мене звать [ім'я]» та метод «Показати інформацію», який виводить всю інформацію (ім'я, прізвище, вік) про певну особу.

```
class Person:
    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
               last name - {self.lastName};\n\
               age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."
```

Тепер створимо два об'єкти (екземпляри класу `Person`) і запустимо їх методи `getInfo()` та `getHi()`:

```
person1=Person("Joe", "Black", 30)
person2=Person("Kate", "Smith", 20)
```

```
print(person1.getInfo())
print(person2.getInfo())

print(person1.getHi("Hi"))
print(person2.getHi("Hello"))
```

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Person first name - Kate; last name - Smith; age - 20.
Hi! I am Joe.
Hello! I am Kate.
```

Рисунок 17

На основі нашого (базового) класу `Person` створимо дочірній клас `Student`, в якому крім властивостей «ім'я», «прізвище», «вік» буде атрибут «спеціальність» і новий метод, що визначає успішність студента за середнім балом (параметр методу).

Для того, щоб у Python створити клас, який успадковує функціональність від іншого класу, потрібно передати батьківський клас, як параметр, дочірньому класу:

```
class Student(Person):
    spec="Computer Science"

    def isSuccessful(self, meanScore):
        return True if meanScore>=75 else False
```

У нашому прикладі клас `Student` не має свого конструктора (функції `__init__()`). Тому при створенні його об'єктів буде викликатися конструктор батьківського класу `Person`.

Створимо першого студента — екземпляр класу **Student**. Оскільки ми будемо використовувати конструктор базового класу **Person** (через відсутність власного), тоді нам потрібно задавати такий самий перелік виводу даних, як і при створенні екземпляра класу **Person**:

```
student1=Student("Joe","Black",30)

print(student1.getInfo())
print(student1.getHi("Morning"))

print(f"Is {student1.firstName} successful student?.
      {student1.isSuccessful(85)}")
```

Як бачимо, екземпляр класу **Student** може викликати усі методи свого базового класу (**getInfo()** та **getHi()** у нашому прикладі). І, звичайно, має всі властивості батьківського класу (ми виводимо значення властивості **firstName** у базовому класі при формуванні рядка з інформацією про успішність нашого студента).

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Morning! I am Joe.
Is Joe successful student?.True
```

Рисунок 18

А якщо нам, крім атрибутів, необхідно також додати також нові властивості у клас **Student**? Наприклад, найкращим рішенням для реалізації методу **isSuccessful()** було б передавати йому середній бал студента, як властивість самого класу. Для цього нам необхідно додати в клас **Student** властивість, як середній бал.

Таким чином, нам потрібно перевизначити конструктор базового класу так, щоб до його вхідних параметрів, крім імені, прізвища, віку, входив ще й середній бал. При цьому нам потрібно не скопіювати функціональність (рядки коду) конструктора батьківського класу, а розширити його. І тому ми всередині конструктора дочірнього класу спочатку маємо викликати конструктор батьківського класу (щоб дочірній клас мав такі ж властивості, як і батьківський клас), а потім додати нову властивість.

Для звернення до базового класу із похідного використовується вираз `super()`. Тоді виклик конструктора батьківського класу здійснюватиметься так:

```
super().__init__(propName1, propName2,... propNameN),
```

де `propName1, propName2,... propName` — список імен властивостей базового класу.

Внесемо потрібні зміни до нашого класу `Student`:

```
class Student(Person):
    spec="Computer Science"

    def __init__(self, firstName, lastName, age, score):
        super().__init__(firstName, lastName, age)
        self.score = score

    def isSuccessful(self):
        if self.score>=75:
            return True
        else:
            return False

student1=Student("Joe","Black",30, 78)
```

```
print(student1.getInfo())
print(student1.getHi("Morning"))

print(f"Is {student1.firstName} successful student?.
      {student1.isSuccessful()}")
```

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Morning! I am Joe.
Is Joe successful student?.True
```

Рисунок 19

Однак, як можна помітити з отриманого результату, під час виклику методу `getInfo()` виводиться не вся інформація про студента: немає його середнього балу, що ускладнює пояснення його успішності.

Це сталося тому, що ми не визначили в дочірньому класі цей метод. А у базовому класі інформація про середній бал була відсутня, тому й не виводилася базовим методом `getInfo()`.

Знову ж таки, наш новий (перевизначений) метод `getInfo()` має містити всю функціональність своєї версії з базового класу. Тому, в тілі перевизначеного методу ми спочатку викличемо базовий метод, а потім напишемо рядки коду, що реалізують додаткові можливості:

```
class Student(Person):
    spec="Computer Science"

    def __init__(self, firstName, lastName, age, score):
        super().__init__(firstName, lastName, age)
```

```

        self.score = score

    def getInfo(self):
        return super().showInfo()+f"score - {self.score}"

    def isSuccessful(self):
        if self.score>=75:
            return True
        else:
            return False

student1=Student("Joe","Black",30, 78)

print(student1.getInfo())

```

Результат:

```

Person first name - Joe; last name - Black; age - 30.score - 78

```

Рисунок 20

Створимо ще один похідний клас [Employee](#) на основі нашого (базового) класу [Person](#).

В новому класі-нащадку [Employee](#), крім властивостей: «ім'я», «прізвище», «вік», будуть властивості, що зберігають інформацію про назву посади, стаж роботи та розмір заробітної плати.

Нам необхідно перевизначити (доповнити) метод базового класу [getInfo\(\)](#) для врахування нової інформації. Також в класі міститься новий метод [getSickLeavePerc\(\)](#), що дозволяє за стажем роботи визначити відсоток заробітної плати, який виплачуватиметься, якщо працівник піде на лікарняний.

```

class Person:
    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."

class Employee (Person):
    def __init__(self, firstName, lastName, age,
                  jobTitle, salary, seniority):
        super().__init__(firstName, lastName, age)
        self.jobTitle=jobTitle
        self.salary=salary
        self.seniority=seniority

    def getInfo(self):
        return super().getInfo()+f"\njobTitle -
                {self.jobTitle}; salary -
                {self.salary}; seniority -
                {self.seniority}."

    def getSickLeavePerc(self):
        if self.seniority>5:
            return 1
        elif 3<self.seniority<=5:
            return 0.75
        else:
            return 0.5

employee1=Employee("Kate","Smith",20,"HR",2500,7)

```

```
print(employee1.getHi("Hello"))
print(employee1.getInfo())

print("The percentage of salary in case of sick
      leave will be {}".format(employee1.
      getSickLeavePerc()*100))
```

Результат:

```
Hello! I am Kate.
Person first name - Kate; last name - Smith; age - 20.
jobTitle - HR; salary - 2500; seniority - 7.
The percentage of salary in case of sick leave will be 100%
```

Рисунок 21

4.1. Методи: загальний, внутрішній (захищений), приватний

Ми вже навчилися успадковувати властивості або поведінку будь-якого класу, використовуючи його як базовий. Проте, а якщо деякі дані (які зберігаються у властивостях або утворюються методами класу) не повинні успадковуватися, тобто мають бути недоступні з похідного класу. У цьому випадку нам потрібний підхід для обмежень доступу до деяких властивостей та методів класу.

Як нам вже відомо, в Python є три форми модифікаторів доступу: **public** (загальний), **protected** (внутрішній або захищений) та **private** (приватний).

Використовуючи модифікатори, ми можемо вказати, чи буде доступна (чи ні) властивість (метод) класу тільки всередині самого класу, для його класів-нащадків та інших класів.

Короткий опис рівня доступу, який надає певний модифікатор, наведено у таблиці.

Таблиця 1

Модифікатор	Всередині класу (в його методах)	Похідні класи	За межами класу і його нащадків
Public	+	+	+
Protected	+	+	-
Private	+	-	-

4.2. Рівень доступу Public (загальний)

Усі компоненти (члени) класу, оголошені як **Public**, доступні з будь-якої частини програми. Усі властивості та методи класу за умовчанням є загальнодоступними.

Раніше ми вже розглядали можливості доступу до **public** — властивостям та методам всередині класу та за його межами. А що щодо доступу з методів класів — нащадків **Person**?

Створимо похідний клас **Employee**, який крім основної інформації та базової поведінки, отриманих від батьківського класу, має властивість **salary** (розмір заробітної плати). Також у нього буде метод **isRetiree()**, який на основі даних про вік визначає, чи є працівник пенсіонером. Саме в рамках цього нового методу ми перевіримо доступ до **public**-властивостей базового класу всередині методів похідного класу:

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary
```

```

def isRetiree(self):
    print(self.getInfo())
    if self.age>60:
        print(f"{self.firstName} is retiree")
    else:
        print(f"{self.firstName} is not retiree")

employee1=Employee("Joe","Black",30, 3000)
employee1.isRetiree()

```

Результат:

```

Person first name – Joe; last name – Black; age – 30.
Joe is not retiree

```

Рисунок 22

Також в рамках методу `isRetiree()` ми показали наявність доступу до `public`-методів базового класу та методів похідного.

4.3. Рівень доступу `Private` (приватний)

Ми вже знаємо, що приватні члени класу (властивості або методи) доступні лише всередині класу, де вони визначені. Тому ми не можемо використовувати `private`-властивості або викликати `private`-методи за межами класу. Однак це також неможливо і для класів, похідних від нього. Цю особливість ми розглянемо зараз.

Давайте пригадаємо, що реалізація `private`-рівня доступу властивості або методу класу забезпечується подвійним символом підкреслення «`__`», який потрібно помістити безпосередньо перед ім'ям властивості (без пробілу), наприклад `__privateProp`, `__privateMethod`.

У нашому базовому класі `Person` є private-властивість `personID`, та Private-метод `showID()`.

```
import random
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self.age}."

    def getHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."
```

Створимо екземпляр класу-нащадка `Employee` (який ми створили в прикладах раніше). І подивимось, яку інформацію про працівника (як об'єкт класу) ми зможемо вивести:

```
class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
```



```

    super().__init__(firstName, lastName, age)
    self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

employee1=Employee("Joe","Black",30, 3000)
employee1.isRetiree()

```

Результат:

```

44
Person first name - Joe; last name - Black; age - 30.
Joe is not retiree

```

Рисунок 23

Зверніть увагу, що інформація про `personID` було виведено. Однак це сталося внаслідок виклику методу базового класу `showInfo()`, який і забезпечив виведення цієї інформації. Якщо ж ми спробуємо отримати доступ до `private`-властивості `personID` напямую, наприклад, створивши в класі-нащадка метод `showEmployeeID()` для виведення цієї інформації:

```

class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)

```

```

        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

    def showEmployeeID(self):
        print(self.__personID)

employee1=Employee("Joe","Black",30, 3000)
employee1.showEmployeeID()

```

то отримаємо виняток

AttributeError ✕

Рисунок 24

з причини:

'Employee' object has no attribute '_Employee__personID'

Рисунок 25

Спробуємо змінити наш новий метод `showEmployeeID()`, викликавши в його тілі Private-метод `showID()`:

```

def showEmployeeID(self):
    self.__showID()

employee1.showEmployeeID()

```

Отримаємо такий самий виняток

AttributeError ✕

Рисунок 26

з причини:

`'Employee' object has no attribute '_Employee__showID'`

Рисунок 27

Таким чином, Private-властивості та методи дійсно доступні лише зі свого класу.

4.4. Рівень доступу Protected (внутрішній або захищений)

Якщо нам необхідно обмежити доступ до властивостей і методів таким чином, щоб їх було «видно» тільки всередині самого класу або класів, похідних від нього, нам потрібен рівень доступу **protected**. В Python створення захищеного рівня доступу властивості або методу класу реалізується за допомогою символу підкреслення «`_`», який потрібно помістити безпосередньо перед ім'ям властивості (без пробілу), наприклад `_propName`, `_methodName`.

Давайте змінимо рівень доступу для властивості «вік» у базовому класі:

```
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
```

```

    #protected properties
    self._age = age

    # public methods
    def getInfo(self):
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self._age}."

    def getHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."

```

Спробуємо змінити цю властивість всередині методу класу-нащадка (додавши метод `changeAge()`):

```

class Employee(Person):
    def __init__(self, firstName, lastName, age, salary):
        super().__init__(firstName, lastName, age)
        self.salary = salary

    def isRetiree(self):
        print(self.getInfo())
        if self._age>60:
            print(f"{self.firstName} is retiree")
        else:
            print(f"{self.firstName} is not retiree")

    def changeAge(self, newAge):
        self._age=newAge

employee1=Employee("Joe","Black",30, 3000)
employee1.isRetiree()

employee1.changeAge(65)
employee1.isRetiree()

```

Результат:

```
Person first name - Joe; last name - Black; age - 30.
Joe is not retiree
Person first name - Joe; last name - Black; age - 65.
Joe is retiree
```

Рисунок 28

А тепер перевіримо, чи зможемо ми отримати до нього доступ за межами базового класу і не в похідному:

```
employee1._age=20
print(employee1.showInfo())
```

Результат:

```
Person first name - Joe; last name - Black; age - 20.
```

Рисунок 29

Отримані результати показують, що ми можемо отримати доступ до захищених членів класу за межами класу, ми навіть можемо змінити значення нашої захищеної властивості «вік».

Тепер перевіримо, як справи із захищеними методами. Для цього додаємо до нашого базового класу `Person` protected-метод `_getFullName()`:

```
import random
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName
```

```

#protected properties
self._age = age

#private properties
self.__personID=random.randint(1,100)

#protected methods
def _getFullName(self):
    return f"Person full name:{self.firstName}
        {self.lastName}"

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
        last name - {self.lastName};
        age - {self._age}."
def getHi(self, msgText):
    print(self.showInfo())
    return f"{msgText}! I am {self.firstName}."

```

Спробуємо викликати його з об'єкта класу (тобто за межами базового класу):

```

person1=Person("Joe","Black",30)
print(person1._getFullName())

```

Результат:

```

Person full name:Joe Black

```

Рисунок 30

Так само, як і при роботі із захищеними властивостями, ми можемо отримати доступ до захищених методів за межами класу.

Виникає сумнів: чи можуть модифікатори доступу фактично змінювати рівень доступу? Чи вони тільки додають нові правила синтаксису?

Насправді модифікатори захищеного доступу розроблені таким чином, щоб відповідальний програміст міг ідентифікувати їх за згодою про імена (наявність символу «`_`» спочатку імені властивості або методу) і виконувати необхідну операцію із цими захищеними властивостями лише в межах свого класу, або похідного від нього.

Тобто модифікатор `protected` — це певна угода (декларація) про те, що дані властивості та методи не потрібно використовувати за межами класу та його нащадків, оскільки це може порушити роботу класу. Рішення про дотримання цих правил-домовленостей — це відповідальність самого програміста. Наприклад, водій (навіть із солідним досвідом водіння) не буде окремо запускати якісь компоненти двигуна своєї машини (хоча він може відкрити капот і отримати до них доступ), тому що такі дії можуть призвести до негативних наслідків.

4.5. Статичні методи і методи класу

В Python можна створювати три види методів: статичні методи, методи класу та методи екземпляра класу.

З методами екземпляра класу ми вже знайомі. В усіх розглянутих раніше прикладах використовувався саме цей тип методів, які ми викликали, використовуючи екземпляр класу (звичайно, вже після його створення).

Наприклад, у нашому класі `Person` ми мали два загальних методи: `getInfo()` та `sayHi()`.

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self._age}."

    def sayHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."
```

І ось так ми викликали їх, використовуючи об'єкт класу:

```
person1=Person("Joe", "Black", 30)
print(person1.getInfo())
```


Методи екземпляра класу є найпоширенішим типом. Як ви пам'ятаєте, першим параметром таких методів завжди є сам екземпляр класу (який позначається як `self`), за допомогою якого ми всередині методу отримуємо доступ до всіх складових даного класу: властивостей, атрибутів та інших методів.

Таким чином, використовуючи методи екземплярів класу, ми можемо змінювати як стан конкретного об'єкта, так і самого класу.

4.6. Статичні методи

Уявімо, що нам необхідно створити метод, алгоритм якого не передбачає роботу із складником даного класу, тобто нам зовсім не потрібно передавати екземпляр класу (`self`) як параметр. Давайте спробуємо оголосити в нашому класі `Person` метод `sayGreetings()` без параметрів:

```
import random
class Person:
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)
```

```
# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

def sayGreetings():
    print("Nice to meet you!")
```

Однак, коли ми спробуємо викликати цей новий метод звичним способом, використовуючи екземпляр класу:

```
person1=Person("Joe", "Black", 30)
person1.sayGreetings()
```

Ми отримаємо помилку:

TypeError ✕

Рисунок 31

Виходить, що методи, які не мають екземпляр класу як параметр, не можуть викликатися об'єктом. Як тоді створити звичайний метод з поведінкою звичайної функції всередині класу?

Звичайно, ми можемо просто створити окрему функцію за межами класу:

```
def sayGreetings():
    print("Nice to meet you!")
```

Однак, якщо в класі **Person** ця функція має виводити повідомлення «*Nice to meet you*», а в класі **Student** — «*Hello!*». Тобто тут нам все ж таки потрібен зв'язок «функція — клас». В Python можна реалізувати подібний механізм, використовуючи статичні методи, які описуються за допомогою декоратора `@staticmethod`:

```
import random
class Person:

    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

    #private methods
    def __showID(self):
        print (self.__personID)

    # public methods
    def getInfo(self):
        self.__showID()
        return f"Person first name - {self.firstName};
                last name - {self.lastName};
                age - {self._age}."

    def sayHi(self, msgText):
        print(self.getInfo())
        return f"{msgText}! I am {self.firstName}."
```

```
#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

person1=Person("Joe","Black",30)
person1.sayGreetings()
```

Результат:

Nice to meet you!

Рисунок 32

Розглянемо ще один приклад, який показує корисність статичних методів у класі. Припустимо, що нам потрібно створити клас, який надає певний набір методів обробки. Наш клас витягуватиме всі числа з рядка (як цілі, так і дробові) і виконуватиме над ними вибрану операцію. Зручно зібрати всю необхідну функціональність в одному класі (а не реалізовувати окремими функціями), оскільки при такому підході можна зберегти визначення класу в окремому файлі (на зразок зовнішньої бібліотеки). Тоді ми здійснюватимемо його імпорт, коли виникає необхідність використання одного з його методів так, як, наприклад, ми підключаємо модуль *random*, коли потрібно використовувати якийсь метод генерації значень. Такий клас не матиме властивостей, лише методи, що реалізують обробку (або якесь обчислення) вхідного значення. Причому цим методам абсолютно не потрібен екземпляр класу як параметр, вони працюватимуть лише з вхідним значенням, над яким потрібно виконати певну операцію.

```

import re
class myOperator:
    @staticmethod
    def incrementer(str):
        numbers=[float(s) for s
                    in re.findall(r'[-?\d+\.\d*', str)]
        result=[]
        for number in numbers:
            result.append(number+1)
        return result

    @staticmethod
    def decrementer(str):
        numbers=[float(s) for s
                    in re.findall(r'[-?\d+\.\d*', str)]
        result=[]
        for number in numbers:
            result.append(number-1)
        return result

userStr="Extract 500 , 100.45, 23.1 and 1000
        from my string"
print(myOperator.incrementer(userStr))
print(myOperator.decrementer(userStr))

```

Результат:

```

[501.0, 101.45, 24.1, 1001.0]
[499.0, 99.45, 22.1, 999.0]

```

Рисунок 33

Як бачимо, метод `incrementer()` витягнув усі числа з рядка і збільшив кожне на одиницю, після чого додав змінену кількість до результуючого списку. Такі ж дії виконав і метод `decrementer()`, але з єдиною відмінністю — кожне число було зменшено на одиницю.

Фактично, в Python статичні методи є звичайними функціями, які визначені в класі і, відповідно, перебувають у просторі імен цього класу. При цьому, статичні методи не можуть змінювати стан а ні самого класу, ні його об'єктів.

4.7. Методи класу

Як методу екземпляра класу завжди першим параметром передається сам екземпляр, так і методу класу першим параметром завжди передається сам клас (що позначається як `cls`). Саме через цей параметр метод класу отримує доступ до всіх класів (до всіх його складових).

Для того, щоб створити метод класу, потрібно використовувати декоратор `@classmethod`. Якщо ми створимо метод класу, що змінює, наприклад, значення атрибута класу, то після виклику даного методу всі об'єкти (екземпляри цього класу) матимуть оновлений атрибут.

Додаймо до нашого класу `Person` атрибут «хобі» зі значенням «`Cooking`». Також створимо метод класу `setDefaultHobby(cls)`, який змінюватиме значення на «`Drawing`»:

```
import random
class Person:
    hobby="Cooking"
    def __init__(self, firstName, lastName, age):
        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protecte properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)
```

```

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

#class methods
@classmethod
def setDefaultHobby(cls):
    cls.hobby="Drawing"

```

Тепер створимо кілька об'єктів. Але після створення першого об'єкта викличемо метод класу `setDefaultHobby(cls)`, щоб перевірити, чи змінилося значення атрибута «хобі» в усіх об'єктів, створених після його виклику.

```

person1=Person("Joe", "Black", 30)
print(person1.hobby)

Person.setDefaultHobby()

```

```

person2=Person("Kate","Smith",20)
print(person2.hobby)

person3=Person("Bob","Gordon",40)
print(person3.hobby)

```

Результат:

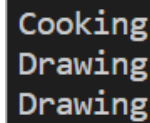


Рисунок 34

Тобто методи класу можуть змінювати стан класу, що вплине на всіх об'єктах (екземплярах цього класу), але при цьому вони не можуть змінювати стан конкретного об'єкта.

Тепер розглянемо ще одну розповсюджену ситуацію, коли застосування методів класу дозволяє створювати об'єкти класу з урахуванням різних даних. Припустимо, що у нас немає інформації про вік деяких осіб (яка є обов'язковим параметром для створення об'єкта класу `Person`). Натомість нам відомий рік їх народження.

Створимо метод класу `basedOnYear()`, який в результаті своєї роботи поверне екземпляр класу, ґрунтуючись на даних про ім'я, прізвище та рік народження:

```

import random
from datetime import date
class Person:
    hobby="Cooking"
    def __init__(self, firstName, lastName, age):

```



```

    # public properties
    self.firstName = firstName
    self.lastName = lastName

    #protecte properties
    self._age = age

    #private properties
    self.__personID=random.randint(1,100)

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

#class methods
@classmethod
def setDefaultHobby(cls):
    cls.hobby="Drawing"

@classmethod
def basedOnBYear(cls,firstName, lastName, bYear):

```

```

        personAge=date.today().year - bYear
        return cls(firstName, lastName, personAge)
person1=Person("Joe", "Black", 30)
print(person1.getInfo())
person2=Person.basedOnBYear("Kate", "Smith", 2000)
print(person2.getInfo())

```

Результат:

```

27
Person first name - Joe; last name - Black; age - 30.
64
Person first name - Kate; last name - Smith; age - 22.

```

Рисунок 35

Тепер припустимо, що у нас немає окремої інформації про ім'я, прізвище та вік користувача, але є рядок, який містить усе це, наприклад, «*Kate Smith 25*».

Створимо ще один метод класу, який на основі такого рядка поверне об'єкт класу **Person**:

```

import random
from datetime import date
class Person:
    hobby="Cooking"
    def __init__(self, firstName, lastName, age):

        # public properties
        self.firstName = firstName
        self.lastName = lastName

        #protected properties
        self._age = age

        #private properties
        self.__personID=random.randint(1,100)

```

```

#private methods
def __showID(self):
    print (self.__personID)

# public methods
def getInfo(self):
    self.__showID()
    return f"Person first name - {self.firstName};
           last name - {self.lastName};
           age - {self._age}."

def sayHi(self, msgText):
    print(self.getInfo())
    return f"{msgText}! I am {self.firstName}."

#static methods
@staticmethod
def sayGreetings():
    print("Nice to meet you!")

#class methods
@classmethod
def setDefaultHobby(cls):
    cls.hobby="Drawing"

@classmethod
def basedOnBYear(cls,firstName, lastName, bYear):
    personAge=date.today().year - bYear
    return cls(firstName, lastName, personAge)

@classmethod
def basedOnStr(cls,strInf):
    firstName, lastName,age = strInf.split(' ')
    return cls(firstName, lastName,age)

person3=Person.basedOnStr("Kate Smith 25")
print(person3.getInfo())

```

Результат:

74

Person first name - Kate; last name - Smith; age - 25.

Рисунок 36

Також методи класу можуть бути корисними, коли нам потрібно змінити логіку (алгоритм) роботи методу об'єкта. Припустимо, що нам потрібно створити метод, який розраховує зарплату працівника (наприклад, на основі базового окладу та процентної ставки). Розмір базового окладу залежить від категорії працівника: **Junior**, **Middle**, **Senior**, **Lead**.

Створимо клас **Developer** — похідний клас від нашого класу **Person**:

```
class Person:
    def __init__(self, firstName, lastName, age):
        self.firstName = firstName
        self.lastName = lastName
        self.age = age

    def getInfo(self):
        return f"Person first name - {self.firstName};\n\
               last name - {self.lastName};\n\
               age - {self.age}."

    def getHi(self, msgText):
        return f"{msgText}! I am {self.firstName}."
```

Ми можемо створити в класі **Developer** private-власивість «**run**g» та метод класу **setRun**g(), який дозволить змінити його і, таким чином, «перемикатиме» категорію працівника для всіх об'єктів, створених після його виклику.

```

class Developer (Person):
    def __init__(self, firstName, lastName, age, jobTitle):
        super().__init__(firstName, lastName, age)
        self.jobTitle=jobTitle
        self.__salary=0

    def setBasicSalary(self):
        if self.__rung=='Junior':
            self.__salary=1000
        elif self.__rung=='Middle':
            self.__salary=3000
        elif self.__rung=='Senior':
            self.__salary=5000
        elif self.__rung=='Lead':
            self.__salary=10000

    @classmethod
    def setRung(cls, rungName):
        cls.__rung=rungName

    def calculateSalary(self,perc):
        return self.__salary+self.__salary*perc

    def getInfo(self):
        return super().getInfo()+
            f"\njobTitle - {self.jobTitle};\n"
            f"\nrung - {self.__rung};\n"
            f"\nbasic salary - {self.__salary}."

```

Встановимо ранг розробника в «**Junior**»:

```
Developer.setRung("Junior")
```

Тепер створимо кілька об'єктів, які представляють розробників рівня «**Junior**»; встановимо їм базовий оклад, розрахуємо зарплату та виведемо інформацію про кожного:

```
jun1=Developer("Kate","Smith",22,
               "UI (user interface) designer")
jun2=Developer("Joe","Smith",25,"Back-end developer")

for jun in zip((jun1, jun2), (0.25,0.3)):
    jun[0].setBasicSalary()
    print(jun[0].getInfo())
    print(f"expexted salary:{jun[0].
          calculateSalary(jun[1])}")
```

Результат:

```
Person first name - Kate; last name - Smith; age - 22.
jobTitle - UI (user interface) designer;
rung - Junior;
basic salary - 1000.
expexted salary:1250.0
Person first name - Joe; last name - Smith; age - 25.
jobTitle - Back-end developer;
rung - Junior;
basic salary - 1000.
expexted salary:1300.0
```

Рисунок 37

Далі «переключимо» клас на рівень «[Middle](#)» та повторимо процедуру.

```
Developer.setRung("Middle")
midl1=Developer("Bob","Dilan",32,"Web developer")
midl2=Developer("Anna","Morning",35,"Data scientist")
midl3=Developer("Helen","Adams",42,"Systems analyst")

for midl in zip((midl1, midl2, midl3), (0.2,0.37, 0.15)):
    midl[0].setBasicSalary()
    print(midl[0].getInfo())
    print(f"expexted salary:{midl[0].
          calculateSalary(midl[1])}")
```

Результат:

```

Person first name – Bob; last name – Dilan; age – 32.
jobTitle – Web developer;
rung – Middle;
basic salary – 3000.
expected salary:3600.0
Person first name – Anna; last name – Morning; age – 35.
jobTitle – Data scientist;
rung – Middle;
basic salary – 3000.
expected salary:4110.0
Person first name – Helen; last name – Adams; age – 42.
jobTitle – Systems analyst;
rung – Middle;
basic salary – 3000.
expected salary:3450.0

```

Рисунок 38

Таким чином, методи класу можуть використовуватись для зміни (налаштування) деяких параметрів (властивостей) класу, які будуть враховані в усіх наступних екземплярах класу (об'єктів, створених після виклику даних методів класу). А ці змінені властивості далі спричинять зміну алгоритму поведінки цих нових об'єктів.

4.8. Множинне успадкування та MRO (порядок вирішення методів)

У розглянутих раніше прикладах, усі наші похідні класи успадковували властивості (методи) лише одного базового класу. Однак в Python також передбачені засоби для реалізації множинного успадкування.

Множинне успадкування — це можливість похідного класу успадкувати особливості та функціональність більш, ніж одного базового класу.

Таким чином, при множинному успадкуванні абсолютно всі властивості та методи всіх батьківських класів успадковуються класом-нащадком.

Синтаксис множинного спадкування не відрізняється від синтаксису одиночного спадкування — ми просто прописуємо в круглих дужках імена всіх базових класів, після імені похідного класу.

Припустимо, що в нас є два базових класи `baseClass1` та `baseClass2`, а нам необхідно створити похідний клас `derivedClass`, який буде успадковувати всю функціональність обох класів:

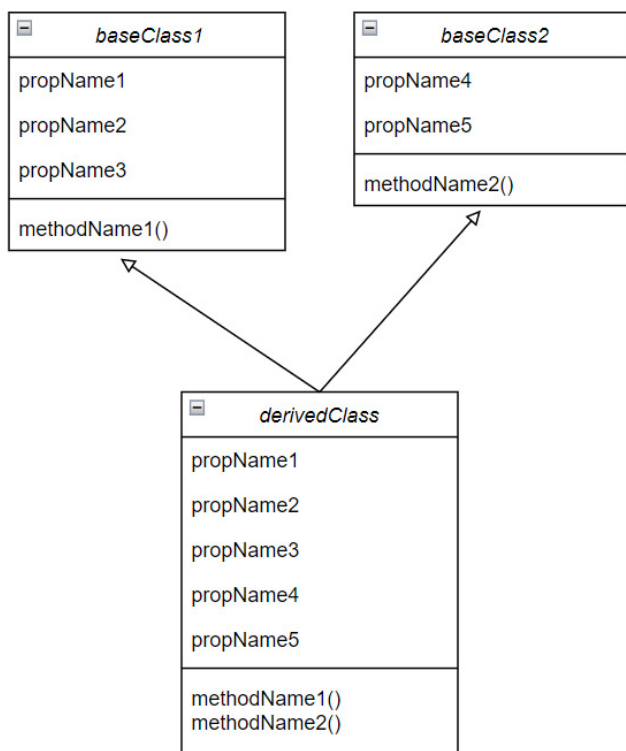


Рисунок 39

Загальний синтаксис множинного успадкування в цьому випадку матиме вигляд:

```
class baseClass1:
    pass
class baseClass2:
    pass
class derivedClass (baseClass1, baseClass2):
    pass
```

Розглянемо з прикладу. Припустимо, що в нас є клас «**Книга**» і клас «**Файл**». На їх основі нам потрібно створити похідний клас «**Електронна книга**»:

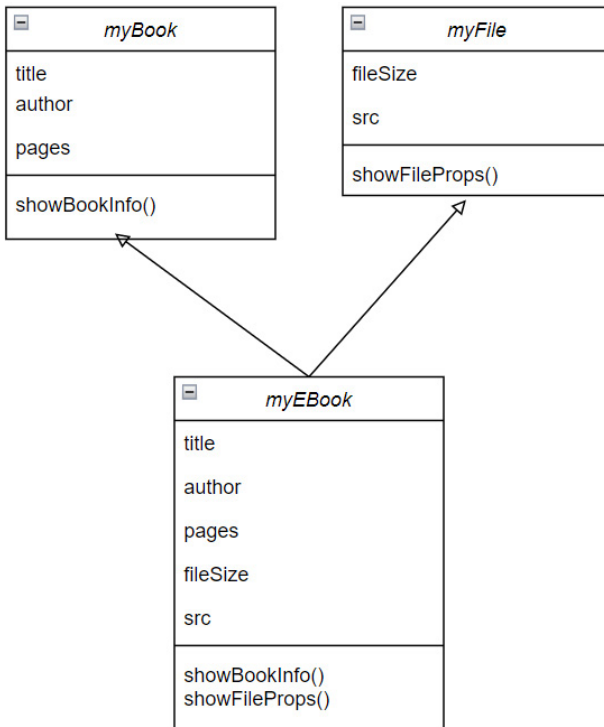


Рисунок 40

```

class myBook:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
    def showBookInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

class myFile:
    def __init__(self, fileSize, src):
        self.fileSize = fileSize
        self.src = src
    def showFileProps(self):
        print("File size: {}".format(self.fileSize))
        print("File source: {}".format(self.src))

class myEBook(myBook, myFile):
    pass

```

Тепер створимо екземпляр похідного класу `myEBook`:

```
eBook1=myEBook()
```

Однак, цей рядок коду створить нам виняток:

TypeError ×

Рисунок 41

І причина його така:

```

myBook.__init__() missing 3 required positional arguments:
    'title', 'author', and 'pages'

```

Рисунок 42

Давайте розберемося, у чому справа.

У нашому прикладі тіло похідного класу відсутнє, у ньому не визначено навіть його конструктора. У цьому випадку буде викликано конструктор того базового класу, ім'я якого ми вказали першим при визначенні похідного класу (тобто буде викликано конструктор класу `myBook`). Як впливає з визначення класу `myBook`, його конструктор має три параметри: `title`, `author`, `pages`, які перелічені вище в повідомлення про помилку.

Додаймо до коду необхідні для конструктора аргументи та виклинемо метод `showBookInfo()` для відображення інформації про назву, автора книги та про кількість сторінок у ній:

```
eBook1=myEBook("Python Crash Course", "Eric Matthes", 624)
eBook1.showBookInfo()
```

```
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
```

Рисунок 43

Однак, нам також необхідно викликати конструктор класу `myFile` для заповнення тих властивостей електронної книги, які пов'язані з файлом.

Адже якщо ми спробуємо викликати метод `showFileProps()` для перевірки, що знаходиться у відповідних властивостях:

```
eBook1.showFileProps()
```

Тоді цей рядок коду викличе виняток

AttributeError ×

Рисунок 44

'myEBook' object has no attribute 'fileSize'

Рисунок 45

Таким чином, нам потрібно створити в нашому подібному класі конструктор, в якому будемо послідовно викликати конструктори батьківських класів:

```
class myEBook(myBook,myFile):
    def __init__(self,title, author, pages,fileSize, src):
        myBook.__init__(self,title, author, pages)
        myFile.__init__(self,fileSize, src)

eBook1=myEBook("Python Crash Course","Eric Matthes",
               624, 1024,"https://www.amazon.co.uk/
               dp/1593276036/?tag=adnruk-21")

eBook1.showBookInfo()
eBook1.showFileProps()
```

Результат:

```
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
File size: 1024
File source: https://www.amazon.co.uk/dp/
            '1593276036/?tag=adnruk-21'
```

Рисунок 46

Тепер розглянемо ситуацію, коли у кожному з класів-предків є метод з однаковою назвою. Змінимо назви методів `showBookInfo()` та `showFileProps()` на `showInfo()`, після чого викличемо даний метод нашого об'єкта `eBook1`.

```
class myBook:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

class myFile:
    def __init__(self, fileSize, src):
        self.fileSize = fileSize
        self.src = src

    def showInfo(self):
        print("File size: {}".format(self.fileSize))
        print("File source: {}".format(self.src))

class myEBook(myBook, myFile):
    def __init__(self, title, author, pages, fileSize, src):
        myBook.__init__(self, title, author, pages)
        myFile.__init__(self, fileSize, src)

eBook1 = myEBook("Python Crash Course", "Eric Matthes",
                 624, 1024, "https://www.amazon.co.uk/dp/1593276036/?tag=adnruk-21")

eBook1.showInfo()
```

Результат:

```
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
```

Рисунок 47

Як бачимо, фактично був викликаний метод `showInfo()` класу `myBook`. Це сталося відповідно до лінеаризації класу `myEBook`.

Лінеаризація класу — це механізм формування списку імен класів при успадкуванні, який задаватиме порядок пошуку властивостей та методів при зверненні до них.

Для нашого класу `myEBook` цей список буде таким: `[myEBook, myBook, myFile, object]`. Такий порядок також відомий, як порядок вирішення методів (*Method Resolution Order*, MRO).

Таким чином, при виклику методу `showInfo()`, для екземпляра класу `myEBook`, інтерпретатор спочатку виконає пошук у поточному класі (`myEBook`). Якщо пошук не дасть результату, то він буде продовжений у базових класах у тому порядку, як вони були вказані при визначенні похідного класу (тобто спочатку буде виконано пошук у класі `myBook`). Оскільки на цьому етапі результат пошуку буде успішним, то відбудеться виконання коду методу `showInfo()` класу `myBook`.

Для того, щоб перевірити, в якому порядку буде проінспектовано базові класи, скористаємося методом класу `mro()`:

```
print(myEBook.mro())
```

```
[<class '__main__.myEBook'>, <class '__main__.myBook'>,  
<class '__main__.myFile'>, <class 'object'>]
```

Рисунок 48

Отже, ми з вами розглянули базову (і водночас, найпоширенішу) ситуацію множинного успадкування.

А якщо у кожного з базових класів, від яких успадковується наш похідний клас, також є класи-предки? До того ж цей клас-предок у базових класів спільний. Така ситуація відома, як «проблема ромбоподібного (алмазного) успадкування» (*diamond problem*), загальний вигляд якого ми бачимо на рисунку 49:

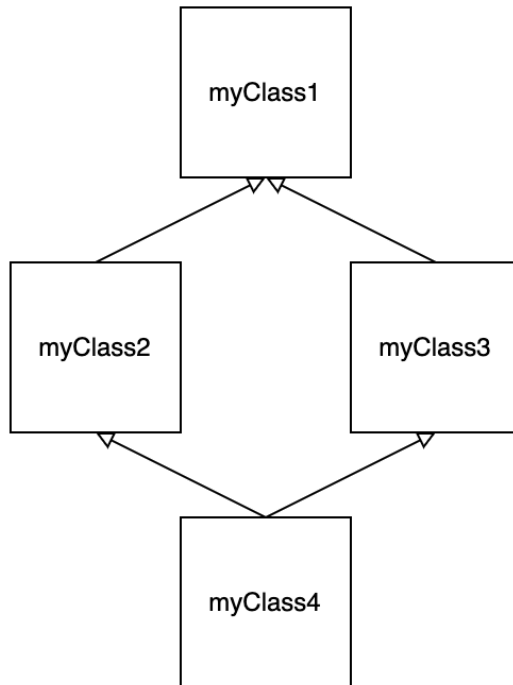


Рисунок 49

```

class myClass1:
    def sayHi(self):
        print("Hi from Class1")

class myClass2(myClass1):
    def sayHi(self):
        print("Hi from Class2")

class myClass3(myClass1):
    def sayHi(self):
        print("Hi from Class3")

class myClass4(myClass2, myClass3):
    pass

```

Якщо є певний метод `sayHi()`, який перевизначено в одному із класів `myClass2` та `myClass3`, або в обох одразу, то виникає неоднозначність: який із варіантів методів `sayHi()` має наслідувати `myClass4`?

```

myObj = myClass4()
myObj.sayHi()

```

Результат:

```
Hi from Class2
```

Рисунок 50

Як бачимо, був викликаний метод `sayHi()` із класу `myClass2`, тому що у списку MRO цей клас-предок йде раніше, ніж `myClass3`, і в ньому визначено потрібний нам метод `sayHi()`.

Тепер, давайте видалимо з класу `myClass2` метод `sayHi()`:


```

class myClass1:
    def sayHi(self):
        print("Hi from Class1")

class myClass2(myClass1):
    pass

class myClass3(myClass1):
    def sayHi(self):
        print("Hi from Class3")

class myClass4(myClass2, myClass3):
    pass

```

Результат:

Hi from Class3

Рисунок 51

На цей раз був викликаний метод `sayHi()` з класу `myClass3`, оскільки незважаючи на те, що в списку MRO цей клас-предок йде раніше, ніж `myClass3`, у ньому не знайшлося методу `sayHi()`, а от у визначенні базового класу `myClass3`, який йде наступним за MRO-списком, цей метод був знайдений.

Головна перевага множинного успадкування — це можливість успадковувати особливості та функціональність більше, ніж одного базового класу. Але його суттєвий недолік — це неоднозначність (плутанина), яка виникає у ситуації, коли декілька базових класів (або й похідний клас теж) реалізують метод з тим самим ім'ям (як у наших прикладах вище). У зв'язку з цим, множинне успадкування

може ускладнити розуміння та підтримку коду, в тому числі й через складні відносини між класами.

Тому дозвіл (синтаксисом мови) множинного успадкування дуже ускладнює правила навантаження методів, що призводить до значного підвищення вимог до рівня програмістів — розробників таких класів.

У деяких мовах програмування, таких як Java, JavaScript, взагалі не підтримується механізм множинного успадкування. У C++ та Python така можливість існує. Більш того, Python має добре продуманий і чітко контрольований підхід до множинного успадкування. Однак, правильне використання цих засобів вимагає високого рівня Python-розробника зі знаннями усіх правил та особливостей механізму множинного успадкування.

5. Поліморфізм

Поліморфізм — це один з основних принципів ООП, який надає можливість використовувати один і той же інтерфейс для різних базових форм. У поліморфізмі метод (оператор) може обробляти об'єкти різним способом залежно від типу класу або типу даних.

Спочатку розглянемо цю можливість на простих прикладах, щоб краще зрозуміти всі нюанси.

Ми знаємо, що оператор «+» є одним із найпоширеніших у програмуванні. Однак у Python він має одне функціональне призначення. Для числових даних такий оператор реалізує арифметичне додавання, а для рядкових — конкатенацію (об'єднання рядків). Однак, незважаючи на різні типи даних, суть виконуваної операції приблизно однакова — складання операндів.

```
number1=2
number2=5.7
print(number1+number2) #7.7

str1="Hello!"
str2="Python"
print(str1+str2) #Hello!Python
```

Вбудована функція `len()` обчислює довжину об'єкта залежно від його класу. Якщо об'єкт є рядком, то `len()` повертає кількість символів, а якщо об'єкт є списком — кількість елементів у списку. Якщо ж ми передаємо `len()` як аргумент-словник, результат роботи рядка — це кількість

ключів словника. Тобто, як і у випадку з «+», суть результатів однакова — кількість.

```
print(len("Student")) #7
print(len(["Python", "C#", "JavaScript"])) #3
print(len({"firstName": "Jane", "lastName": "Smith"})) #2
```

Концепція поліморфізму активно використовується під час створення методів класу. Python дозволяє різним класам мати методи з однаковими іменами. Саме завдяки цій особливості ми зможемо потім узагальнити виклик цих методів (наприклад, для кількох об'єктів, обробляючи їх у циклі), не надаючи особливого значення об'єкту, з яким ми працюємо на поточному кроці. Давайте розглянемо на прикладі. Створимо два класи: **Film** та **Book**, які мають схожу структуру, та метод **showInfo()**.

```
class Film:
    def __init__(self, originalTitle, director, genre):
        self.originalTitle = originalTitle
        self.director = director
        self.genre = genre

    def showInfo(self):
        print("Original title: {}".format(self.originalTitle))
        print("Director: {}".format(self.director))
        print("Genre: {}".format(self.genre))

class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
```

```

def showInfo(self):
    print("Title: {}".format(self.title))
    print("Author: {}".format(self.author))
    print("Pages: {}".format(self.pages))

film1=Film("The Godfather","Francis Ford Coppola",
           "Crime, drama")
book1=Book("Python Crash Course","Eric Matthes", 624)

for item in (film1, book1):
    item.showInfo()

```

Результат:

```

Original title: The Godfather
Director: Francis Ford Coppola
Genre: Crime, drama
Title: Python Crash Course
Author: Eric Matthes
Pages: 624

```

Рисунок 52

Зверніть увагу, що у прикладі ми не поєднували ці класи будь-яким чином. Ми просто «запакували» ці два різні об'єкти в кортеж і вивели інформацію про кожен з об'єктів. Це можливе саме завдяки поліморфізму.

5.1. Перевантаження операторів

Як ми вже помітили, аналізуючи функціональність оператора «+», в Python існують оператори, які виконують різні операції залежно від типу операндів. Це результат навантаження операторів.

Перевантаження операторів у Python — це можливість змінювати (варіювати) їх функціональність (виконані ними операції) залежно від контексту (класу), використовуючи спеціальні методи у класах.

Отже, у нас є клас `Book` і два об'єкти (дві книги, які прочитав користувач).

```
class Book:

    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))

book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 170)
```

Припустимо, що нам необхідно дізнатися загальну кількість прочитаних користувачем сторінок, тобто додати дві книги.

Однак, в результаті такого рядка коду:

```
print(book1+book2)
```

отримаємо помилку

TypeError ×

Рисунок 53

```
unsupported operand type(s) for +: 'Book' and 'Book'
```

Рисунок 54

Інтерпретатор Python не знає, як «додавати» два об'єкти `Book`. Адже оператор «+» не визначений для об'єктів такого класу, як, наприклад, він визначений для чисел та рядків. Для вирішення цього завдання потрібно «навчити» оператор «+» працювати з операндами типу `Book`.

І для цього ми познайомимося з «магічними» методами Python.

5.2. Magic-методи, конструктори

Magic-методи — це спеціальні методи, назви яких обрамлені двома нижніми підкресленнями (починаються і закінчуються подвійним підкресленням). Тому їх часто називають dunder-методами, як скорочення від фрази «*double underscore*».

Magic-методи відповідають за реалізацію деяких стандартних здібностей об'єктів і автоматично викликаються при використанні цих здібностей.

Наприклад, коли у прикладі вище ми виконували операцію `number1 + number2`, то фактично це перетворювалося на виклик методу `number1.__add__(number2)`. Тобто magic-метод `__add__` реалізує операцію додавання.

Насправді у Python багато вбудованих операторів і функцій викликають magic-методи. Додаючи (перевизначаючи) ці magic-методи в наші об'єкти, ми зможемо застосовувати потрібні нам вбудовані оператори та функції над об'єктами. Ми вже використовували один з «магічних» методів — конструктор `__init__()`, з допомогою якого ми

визначаємо особливості ініціалізації наших об'єктів. Однак, коли ми створюємо екземпляр нашого класу:

```
objName = someClassName()
```

то спочатку викликається не метод `__init__()`.

Першим викликається метод `__new__()`, який створює екземпляр нашого класу. Даний метод приймає клас (який прийнято позначати як `cls`) як аргумент. Основне призначення даного методу — це саме створення екземпляра класу (об'єкта). Цей новий об'єкт далі ініціалізується за допомогою методу `__init__()`.

Давайте детальніше розглянемо цей процес з прикладу, продемонструвавши послідовність викликів цих двох методів під час створення об'єкта.

```
class Class1:
    def __new__(cls):
        print("Hi! I am __new__ magic method.")
        return super(Class1, cls).__new__(cls)

    def __init__(self):
        print("Hi! I am __init__ magic method.")

obj1=Class1()
```

Результат:

```
Hi! I am __new__ magic method.
Hi! I am __init__ magic method.
```

Рисунок 55

Однак, найбільшою перевагою магічних методів в Python є те, що вони надають нам механізм, з якого ми

можемо «навчити» наші об'єкти поведінці вбудованих типів. Розглянемо найпопулярніші з них.

5.3. Реалізація магічних методів

Повернемося до нашого класу `Book`:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
```

Якщо ми спробуємо передати наш об'єкт `book1` у функцію `print()`, то результат не буде інформативним для користувача (ми не виведемо інформацію про значення властивостей нашого об'єкта):

```
book1=Book("Python Crash Course","Eric Matthes", 624)
print(book1)
```

Результат:

```
<__main__.Book object at 0x000001BDD220BB20>
```

Рисунок 56

Однак якщо ми визначимо в нашому класі магічний метод `__str__()`, який відповідає за рядкове представлення об'єкта, то ми можемо вирішити цю проблему (оскільки цей метод викликається автоматично при виклику функції `print()`, яка виводить рядкове представлення переданого їй аргументу, тобто результат роботи методу `__str__()`).

Метод `__str__()` приймає екземпляр класу як аргумент і повертає рядок, який ми сформуємо в тілі методу потрібним нам чином.

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"Title: {self.title},\n        author: {self.author},\n        pages: {self.pages}"

print(book1)
```

Результат:

```
Title: Python Crash Course, author: Eric Matthes, pages: 624
```

Рисунок 57

Тепер припустимо, що нам потрібно порівняти дві книги. Якщо просто спробувати порівняти два об'єкти напямую:

```
book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 170)

print(book1==book2) #False
```

результат завжди буде `False`, тому що це абсолютно різні сутності. Справа в тому, що при створенні екземпляра

змінного типу даних (до яких належать і класи користувача), Python виділяє новий фрагмент пам'яті та призначає унікальний ідентифікатор (**id**) для кожного об'єкта. І порівняння екземплярів змінюваних типів даних за умовчанням відбувається саме за їх унікальними ідентифікаторами.

Але проблема навіть не в цьому, а в тому, що для користувача порівняння двох книг представляється звичайним завданням: він може порівнювати їх за ціною, за автором, за кількістю сторінок. Головне для користувача — розуміти, що є критерієм порівняння. Тому в програмі, яка «працює» з книгами, він очікує наявності такої ж можливості. Нам необхідно реалізувати такий механізм порівняння книжок, які у нашій програмі представлені об'єктами, тобто відобразити ці особливості у поведінці об'єктів-книг.

У Python є безліч магічних методів, призначених для реалізації інтуїтивно зрозумілих операцій порівняння для об'єктів, які дозволяють реалізовувати порівняння за допомогою звичних операторів: `==`, `<`, `>`, `<=`, `>=`, `!=`.

Таблиця 2

Magic- метод	Оператор порівняння, поведінка якого визначається методом
<code>__eq__</code>	<code>==</code>
<code>__ne__</code>	<code>!=</code>
<code>__lt__</code>	<code><</code>
<code>__le__</code>	<code><=</code>
<code>__gt__</code>	<code>></code>
<code>__ge__</code>	<code>>=</code>

Припустимо, що книги співпадають, якщо збігаються їх назви та автор. Одна книга більша за іншу, якщо кількість сторінок в ній більша.

Додаємо ці можливості до нашого класу **Book**:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"Title: {self.title},
                author: {self.author},
                pages: {self.pages}"

    def __eq__(self, otherObj):
        if self.author==otherObj.author and
            self.title==otherObj.title:
            return True
        else:
            return False

    def __gt__(self, otherObj):
        if self.pages>otherObj.pages:
            return True
        else:
            return False

book1=Book("Python Crash Course","Eric Matthes", 624)
book2=Book("JavaScript: The Good Parts",
            "Douglas Crockford", 170)
book3=Book("Python Crash Course","Eric Matthes", 700)
print(book1==book3)
print(book1>book2)
```

Результат:



Рисунок 58

Припустимо, що у нас є інформація про кількість відгуків на книгу протягом останнього року. Додаємо до нашого класу властивість `feedbacksN` — список, що містить дванадцять цілих значень (скільки було відгуків на книгу у визначений місяць, тобто позиція значення у списку відповідає номеру місяця). Також внесемо зміни у метод `__str__()`, щоб інформація про відгуки також відображалася, як і інші властивості.

```
class Book:
    def __init__(self, title, author, pages, feedbacksN):
        self.title = title
        self.author = author
        self.pages = pages
        self.feedbacksN=feedbacksN

    def __str__(self):
        return f"Title: {self.title},
               author: {self.author},
               pages: {self.pages},
               feedbacksN:{self.feedbacksN}"
```

Було б зручніше, як би об'єкт (певна книга) мав можливість отримувати доступ до інформації (кількості відгуків) за конкретний місяць за допомогою звичного індексування, наприклад, так `book1[monthNum]`. Для вирішення цього завдання нам потрібно визначити у класі `Book` метод `__getitem__(self,key)`, який визначає поведінку під час доступу до елемента, використовуючи нотацію `self[key]`.

```
import random
class Book:
    def __init__(self, title, author, pages, feedbacksN):
```

```

        self.title = title
        self.author = author
        self.pages = pages
        self.feedbacksN = feedbacksN

    def __str__(self):
        return f"Title: {self.title},
                author: {self.author},
                pages: {self.pages},
                feedbacksN:{self.feedbacksN}"

    def __eq__(self, otherObj):
        if self.author == otherObj.author and
            self.title == otherObj.title:
            return True
        else:
            return False

    def __gt__(self, otherObj):
        if self.pages > otherObj.pages:
            return True
        else:
            return False

    def __getitem__(self, ind):
        if 0 <= ind <= 11:
            return self.feedbacksN[ind]
        else:
            return -1

PythonFeedbacks=[random.randint(50,300) for i
                  in range(12)]

book1 = Book("Python Crash Course", "Eric Matthes",
             624, PythonFeedbacks)

print(book1[2]) #265

```

Якщо нам потрібно забезпечити також оновлення (запис інформації) про кількість відгуків за певний місяць (у форматі `book1[someInd]=newInf`), то в цьому допоможе магічний метод `__setitem__(self, key, value)`.

Додаткову інформацію з магічних методів у Python можна переглянути за [посиланням](#).

6. Створення і керування поведінкою екземплярів класу

6.1. Функтори

Раніше ми вже познайомилися з поняттям, призначенням та особливостями застосування замикань у Python.

Давайте згадаємо основні нюанси на прикладі:

```
def doExercise1(var1):  
    def doExercise2(var2):  
        return var1**var2  
    return doExercise2
```

Функція-замикання `doExercise2()` використовує в своєму тілі тільки змінну `var1` (аргумент охоплюючої функції `doExercise1()`). Тому значення змінної `var1` (число 2) «запам'ятається» (незважаючи на те, що виконання функції `doExercise1(2)` було завершено).

```
case1=doExercise1(2)  
  
print(case1(5)) #32  
print(case1(10)) #1024
```

У рядку виклику «`case1(5)`» програма використовує для обчислення «запам'ятоване» значення змінної `var1` (2), а значення «5» використовується, як аргумент функції-замикання `doExercise2()`.

Також і при виклику `case1(10)` використовується «запам'ятоване» значення змінної `var1` (2) і аргумент для функції-замикання `doExercise2()` — значення «10». Основна перевага та призначення цього підходу — це «запам'ятовування» необхідних даних.

В об'єктно-орієнтованій парадигмі таке «запам'ятовування» також може виявитися корисним. Наприклад, нам потрібно створити клас `UserPlayer`, в якому передбачається зберігати та оновлювати стан «гаманця» користувача у грі. Звичайно, ми також можемо легко створити властивість «`wallet`» (в цьому випадку краще використовувати модифікатор доступу `private`) і метод для його оновлення `updateWallet()`.

```
class UserPlayer:
    def __init__(self, name):
        self.name=name
        self.__wallet=100

    def updateWallet(self,coins):
        self.__wallet+=coins

    def showWallet(self):
        print(f"You have {self.__wallet} coins now.")

user1=UserPlayer("Joe")
user1.updateWallet(50)
user1.showWallet()
```

Результат:

You have 150 coins now.

Рисунок 59

Проте, якщо на наступному кроці нам потрібно створити клас гравця-бота `BotPlayer`, для якого також необхідно зберігати та оновлювати «гаманець» за таким самим алгоритмом?

Також нам може знадобиться можливість встановлювати початковий стан «гаманець». Проблема в тому, що властивість «`wallet`» є `private`, тому доведеться створити окремий метод і викликати його для встановлення початкового стану «гаманця». Зручніше реалізувати алгоритм оновлення стану «гаманця» окремо, наприклад, у вигляді окремого класу. Більш того, такий підхід дозволить нам також приховати справжню реалізацію алгоритму оновлення «гаманця».

У цьому завданні дуже корисними виявляються функтори.

Функтори — це об'єкти (екземпляри класів-функторів), які можна викликати як звичайні функції. У Python будь-який клас, в якому визначено спеціальний «магічний» метод `__call__()` (який дозволяє перевантажити оператор `()`) є функтором. Основна перевага функторів полягає в тому, що вони можуть зберігати інформацію про стан — те, що нам зараз і потрібно.

Створимо функтор для встановлення початкового стану «гаманця» та його оновлення на вказану кількість монет:

```
class WalletFunctor:
    def __init__(self, startCoins=100):
        self.__startCoins = startCoins

    def __call__(self, coins=0):
        return self.__startCoins+coins
```

Наш метод `__call__()` повинен обов'язково повертати нове значення `__startCoin` (а не просто оновлювати його), оскільки спочатку в нашій ідеї функтор `WalletFuncтор` — це стан (пам'ять) «гаманця» (викликали функтор — отримали стан).

Тепер створимо об'єкти — екземпляри цього класу-функтора для гравця — користувача та гравця-бота (з різним початковим станом «гаманця»). І оновимо «гаманці» бота і користувача.

```
userWallet = WalletFuncтор()
print(f"Start state of user wallet: {userWallet()} coins")
print(f"State of user wallet after updating to 50 coins:
      {userWallet(50)} coins")

botWallet = WalletFuncтор(50)
print(f"Start state of bot wallet: {botWallet()} coins")
print(f"State of user bot after updating to 20 coins:
      {botWallet(20)} coins")
```

Результат:

```
Start state of user wallet: 100 coins
State of user wallet after updating to 50 coins: 150 coins
Start state of bot wallet: 50 coins
State of user bot after updating to 20 coins: 70 coins
```

Рисунок 60

Тепер створимо клас `UserPlayer` з двома private-властивостями: «гаманцем» — `__wallet` та «налаштувачем гаманця» — екземпляром класу — функтора `WalletFuncтор`, який задає алгоритм початкового налаштування та подальшого оновлення «гаманця».

```

class UserPlayer:

    def __init__(self, name):
        self.name=name
        self.__walletSetter=WalletFunctor()
        self.__wallet=self.__walletSetter()

    def updateWallet(self,coins=0):
        self.__wallet=self.__walletSetter(coins)

    def showWallet(self):
        print(f"{self.name}!
              You have {self.__wallet}
              coins now.")

```

Тепер створимо об'єкт гравця, покажемо початковий стан його гаманця, а потім, після оновлення на 50 монет, ще раз перевіримо «гаманець» гравця.

```

user1=UserPlayer("Joe")
user1.showWallet()
user1.updateWallet(50)
user1.showWallet()

```

Результат:

```

Joe! You have 100 coins now.
Joe! You have 150 coins now.

```

Рисунок 61

Повна послідовність всіх виконаних кроків представлена на рисунку 62.

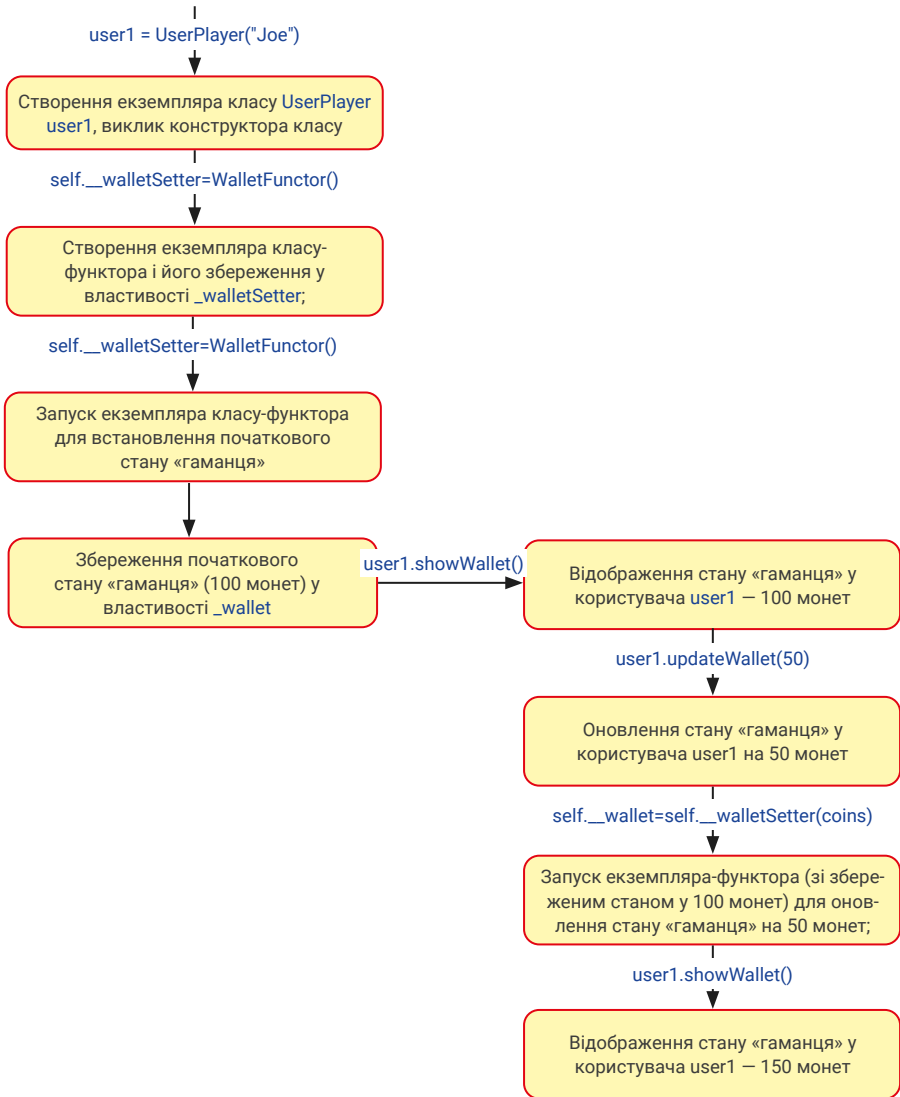


Рисунок 62

Все пройшло вдало: початковий стан «гаманець» та його подальше оновлення виконується, причому алгоритм цих процесів прихований.

Тепер створимо клас `BotPlayer` і додамо до нього такі ж можливості для роботи з «гаманцем».

```
import random
class BotPlayer:
    def __init__(self):
        self.__name=random.choice(["SuperBot",
                                    "MegaBot", "Bot-player"])
        self.__walletSetter=WalletFuncтор(50)
        self.__wallet=self.__walletSetter()

    def updateWallet(self,coins=0):
        self.__wallet=self.__walletSetter(coins)

    def showWallet(self):
        print(f"{self.__name}!
              You have {self.__wallet} coins now.")

bot1=BotPlayer()
bot1.showWallet()
bot1.updateWallet(20)
bot1.showWallet()
```

Результат:

```
MegaBot! You have 50 coins now.
MegaBot! You have 70 coins now.
```

Рисунок 63

Як бачите, ми можемо «підіграти» гравцю-користувачу, встановивши початковий стан «гаманця» бота в 50 монет (тоді, як значення за замовчуванням, які ми використовували для гравця-користувача, становить 100 монет). Однак, таке приховування реалізації певного алгоритму не єдина перевага використання функторів.

Припустимо, що алгоритм оновлення «гаманців» змінився: тепер, за кожне поповнення «гаманця», його власнику нараховується на рахунок +5% стану рахунку. Оскільки наш алгоритм оновлення «гаманця» знаходиться в окремому класі, то цю зміну легко виконати, при цьому не беручи до уваги структури кожного з класів, які використовують «гаманець» (тобто нам не потрібно буде вносити зміни до класів `BotPlayer` та `UserPlayer`).

Розглянемо ще один приклад: припустимо, що нам потрібно зберігати кількість запусків гри, яка виконується з одного акаунту гравця. При цьому, коли створюється об'єкт гравця (наприклад, під час реєстрації у грі), запуск гри відбувається автоматично.

Фактично нам потрібно реалізувати лічильник викликів функції (у нашому прикладі — методу для старту гри з певного облікового запису).

Ми створимо функтор `callCounter`, який зберігатиме і оновлюватиме (збільшуватиме на одиницю при кожному виклику функтора) число «запусків».

```
class callCounter:
    def __init__(self, n=0):
        self.__n = n

    def __call__(self):
        self.__n+=1
        return self.__n
```

У класі `Game` ми створимо метод `startGame()`, який при своєму виклику запускатиме екземпляр функтора `callCounter`. При цьому в конструкторі класу `Game` ми також

спочатку створюємо екземпляр функтора `callCounter`, а потім викликаємо його (оскільки під час реєстрації користувача, створення об'єкта гравця, гра запускається, тобто лічильник запусків дорівнює одиниці).

```
class Game:
    def __init__(self, playerName):
        self.playerName=playerName
        self.__gameCounter=callCounter()
        self.__Ncalls=self.__gameCounter()
    def startGame(self):
        self.__Ncalls=self.__gameCounter()
    def showInfo(self):
        print(f"User {self.playerName} has launched
              the game {self.__Ncalls} times already!")

user1=Game("Bob")
for i in range(5):
    user1.startGame()
    user1.showInfo()
```

Результат:

```
User Bob has launched the game 2 times already!
User Bob has launched the game 3 times already!
User Bob has launched the game 4 times already!
User Bob has launched the game 5 times already!
User Bob has launched the game 6 times already!
```

Рисунок 64

6.2. Декоратори

Ми з вами вже знайомі з декораторами-функціями, такими функціями-«обгортками», які дозволяють нам розширити функціональність існуючої функції без прямої

зміни коду в її тілі. Згадаймо особливості їх застосування на прикладі.

Припустимо, що в нас є список цін на товари в доларах. І функція, яка переводить ціну товару в доларах у відповідний гривневий еквівалент:

```
pricesUSD=[100.34, 35, 67.99, 25.5]
print (pricesUSD)

USDrate=27.5

def toPriceNew(priceList):
    return list(map(lambda x: x*USDrate, priceList))
```

Однак зараз на всі товари діє знижка (наприклад, 15%) і нам потрібно перевести ціни у еквівалент гривні і ще додатково врахувати знижку. Знижка — непостійна характеристика товару, тому змінювати код функції `toPriceNew()` нам немає сенсу.

Ми створимо декоратор, який «застосує» знижку після переведення ціни в іншу валюту:

```
pricesUSD=[100.34, 35, 67.99, 25.5]
print (pricesUSD)

def changePriceDecorator_v1(myFunction):
    print("Hello! Let's change your prices...")
    def simpleWrapper(argList):
        print("I've got list of prices with {} elements.
              Function starts working...".
              format(len(argList)))
        resutl=myFunction(argList)
        resutlwithDisc=list(map(lambda x: x*(1-0.15),
                                resutl))
```

```

        print("Let's set a discount..")
        return resutlwithDisc
    return simpleWrapper

pricesToGRN = changePriceDecorator_v1(toPriceNew)
print(toPriceNew(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts working...
Let's set a discount..
[2345.4474999999998, 818.125, 1589.26625, 596.0625]

```

Рисунок 65

Ми також знаємо і другий спосіб «декорування» функцій — це використання інструкції «`@ім'я_декоратора`», яку потрібно помістити над оголошенням функції, що «декорується».

```

@changePriceDecorator_v1
def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))

print(toPriceNew(pricesUSD))

```

В Python декораторами може бути не лише функції, а й класи. Ми вже помітили, що методи в Python фактично є звичайними функціями, в яких першим аргументом має бути посилання на екземпляр класу (об'єкт — `self`). Таким чином, ми можемо декорувати й метод класу, розширивши його функціональність без зміни початкового коду методу.

Розглянемо декорування методів класу на прикладі нашого класу `Book`:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def showInfo(self):
        print("Title: {}".format(self.title))
        print("Author: {}".format(self.author))
        print("Pages: {}".format(self.pages))
```

Припустимо, що нам перед виведенням інформації про книгу за допомогою методу `showInfo()` потрібно вивести фразу «*General information:*». Для цього створимо відповідну функцію-декоратор:

```
def methodDecorator(method_to_decorate):
    def wrapper(self):
        print("General information:")
        method_to_decorate(self)
    return wrapper
```

І перед оголошенням методу `showInfo()` у класі `Book` додамо інструкцію `@methodDecorator`:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    @methodDecorator
```

```
def showInfo(self):
    print("Title: {}".format(self.title))
    print("Author: {}".format(self.author))
    print("Pages: {}".format(self.pages))

book1=Book("Python Crash Course","Eric Matthes", 624)
book1.showInfo()
```

Результат:

```
General information:
Title: Python Crash Course
Author: Eric Matthes
Pages: 624
```

Рисунок 66

А чи можна створювати клас-декоратор? Адже ми вже знайомі з функторами і знаємо, що об'єкти (екземпляри класів-функторів) можна викликати, як звичайні функції, отже, їм можна «доручити» процес декорування.

Розглянемо це завдання на простому прикладі. Припустимо, що ми маємо функцію, яка виконує додавання двох чисел і повертає отриманий результат. Давайте розглянемо сценарій, в якому ми хотіли б додати до цієї функції додаткову функціональність (піднести значення, що повертається до квадрата), не змінюючи початковий код. Ми можемо досягти цього за допомогою класу-декоратора. Для цього ми визначимо у класі-декораторі два методи: `__init__()` та `__call__()`. Коли ми декоруємо функцію класом, то функція автоматично передається конструктору як аргумент `__init__()`. Ми створимо у нашому класі `private`-властивість для зберігання цієї функції.

Також декоратор мусить мати можливості викликаної функції, яка приймає функцію, яку необхідно декорувати і повертає її декоровану версію (яку можна запустити). Тому нам потрібно визначити в класі-декораторі метод `__call__()`, у тілі якого ми якраз і додамо потрібну нам нову функціональність.

```
class myDecorator:
    def __init__(self, fn):
        self.fn=fn

    def __call__(self, num1, num2):
        return self.fn(num1, num2)**2

@myDecorator
def addNums(x,y):
    return x+y

print(addNums(2,3)) #25
```

У методі `__call__()` ми, крім першого аргументу `self` (самого об'єкта) вказали ще два аргументи, тому що для нашої початкової функції, яку ми декоруватимемо, потрібно два аргументи. Всередині методу ми викликаємо нашу функцію (посилання на неї встановлено через властивість класу `fn`) з двома переданими значеннями, після чого піднесемо отриманий результат її роботи до квадрата.

Далі, перед оголошенням нашої функції `addNums()`, ми додали інструкцію `@myDecorator`. Тепер в результаті виклику нашої функції `addNums(2,3)` ми отримаємо результат роботи її декорованої версії — число 25 $((2+3)**2 = 25)$.

Ми пам'ятаємо, що функтори (як і замикання) дозволяють «запам'ятовувати» потрібні дані, зокрема й результати попередніх викликів функції. Клас-декоратор має можливості функтора (бо в ньому визначено метод `__call__()`), тому ми можемо додати до нашого класу `myDecorator` властивість — перелік `memoryCall`, який накопичуватиме результати роботи декорованої функції.

Спочатку (при виклику конструктора) `memoryCall` буде пустим списком. А потім (всередині методу `__call__()`) ми будемо записувати в нього результати викликів декорованої функції.

Також нам потрібно додати метод `showMemoryState()` для перегляду стану «пам'яті».

```
class myDecorator:
    def __init__(self, fn):
        self.fn=fn
        self.__memoryCall=[]

    def __call__(self, num1, num2):
        self.__memoryCall.append(self.fn(num1, num2)**2)
        return self.fn(num1, num2)**2

    def showMemoryState(self):
        print(f"Current memory state:{self.__memoryCall}")

@myDecorator
def addNums(x, y):
    return x+y

print(addNums(2,3))
addNums.showMemoryState()
print(addNums(3,3))
```

```
addNums.showMemoryState()
print(addNums(4,3))
addNums.showMemoryState()
```

Результат:

```
25
Current memory state:[25]
36
Current memory state:[25, 36]
49
Current memory state:[25, 36, 49]
```

Рисунок 67

Спочатку ми запустили декоровану версію функції `addNums(2,3)`, отримавши результат `25`, який після свого обчислення був доданий до списку `memoryCall` (до його повернення функцією). Виклик методу `showMemoryState()` вивів поточний вміст списку `memoryCall` — `[25]`, який на даному етапі містить лише один результат (після одного виклику декорованої версії функції `addNums()`)

Наступний виклик `addNums(3,3)` поповнив список `memoryCall` на результат другого виклику функції, тепер `memoryCall` вже містить два значення: `[25, 36]`.

І останній (третій) виклик `addNums(4,3)` додав у `memoryCall` третє значення: `[25, 36, 49]`.

6.3. Керовані атрибути об'єкта — `property()`

Як ми знаємо, атрибути об'єкта використовуються для опису основних характеристик даного екземпляра класу (конкретного об'єкта), які задаються в момент створення цього об'єкта. Досить часто, щоб не плутати атрибути

об'єкта з атрибутами класу (значення яких однакові для всіх екземплярів класу) їх називають властивостями. І ми також практикували такий підхід у багатьох прикладах раніше.

Наприклад, для класу **Product** (товар) такими атрибутами об'єкта (властивостями) можуть бути назва товару, його ціна, розмір знижки, кількість відгуків, а атрибутами класу (відомі також, як змінні класу) — якісь загальні характеристики для всіх (або багатьох) об'єктів, наприклад, категорія товару «побутова техніка».

Сукупність усіх атрибутів об'єкта відображає стан (*state*) об'єкта, яка у багатьох завданнях потрібно аналізувати і навіть змінювати (керувати станом об'єкта). Наприклад, нам потрібно змінити розмір знижки на певний товар залежно від кількості відгуків, тобто ми перевіряємо стан даного об'єкта (поточна кількість відгуків про товар) та змінюємо стан (оновлюємо розмір знижки).

Як правило, в нас є як мінімум два способи керування атрибутом: напряду (звертаючись до нього на ім'я через відповідний об'єкт) або за допомогою методів класу. Ми вже знайомі з модифікаторами доступу **public**, **private** та **protected**, і знаємо, як режим доступу впливає на спосіб роботи (взаємодії) з атрибутом.

Давайте пригадаємо, що в Python за замовчуванням всі атрибути об'єкта (як і методи) мають рівень доступу **public**. Слід зазначити, що у більшості практичних випадків ми використовуємо саме цей модифікатор. Таким чином, атрибути стають частиною загальнодоступного інтерфейсу (API) для роботи з нашим класом, тобто кожен розробник отримує доступ до них і зможе змінювати їх значення у своєму коді. Проблеми виникають, коли нам

потрібно модифікувати внутрішню реалізацію цього атрибута об'єкта, але ми не маємо права порушити (фактично змінивши) цю частину API-класу, оскільки це вплине на стабільність та працездатність всього коду, пов'язаного з використанням цього атрибуту. Розглянемо цю ситуацію з прикладу.

Припустимо, що в нас є клас **Product** — товар в онлайн-супермаркеті.

```
class Product:
    def __init__(self, name, price, discount=0.25):
        self.name = name
        self.price = price
        self.discount = discount

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.discount)} grn.")

    def toUSD(self, usdExchRate):
        return self.price*(1-self.discount)/usdExchRate
```

Кожен товар має атрибут «**discount**» — розмір знижки, що є числом в діапазоні від 0 до 1 (відсоток знижки вже переведений у число для скорочення обчислень). Інформація про ціну товару, в тому числі і долари (для конвертації ціни передбачено окремий метод **toUSD()**) виводиться з урахуванням знижки:

```
item1=Product("Lipton Peach Iced", 42, 0.3)
item2=Product("Pure Apple Juice", 28)

item1.showInfo()
```

```
print(f"Price in USD$:{item1.toUSD(30)}")
item2.showInfo()
print(f"Price in USD$:{item2.toUSD(30)}")
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
```

Рисунок 68

Ми можемо виводити і, за необхідності, змінювати знижку на товар (примірника класу **Product**):

```
item1.discount=0.5
print(f"New discount for {item1.name} -
      {item1.discount*100}%.")
```

Тепер припустимо, що наш важливий замовник прийшов з новою вимогою — знижка на товар має зберігатися у вигляді відсотків (25%, а не 0.25) у властивості «**discountPercentage**», яка також має бути загальною. На даному етапі видалення атрибуту «**discount**» та додавання атрибуту «**discountPercentage**» призведе до поломки не тільки двох методів, а й усіх рядків коду, які безпосередньо працювали з атрибутом «**discount**». Звісно, нам потрібно інше вирішення цієї проблеми, а не видалення атрибуту «**discount**».

В деяких мовах програмування (Java та C++) для запобігання подібних проблем рекомендується не створювати загальні атрибути, а забезпечувати доступ до них

тільки через спеціальні методи: гетери (*Getters*) — для отримання значення атрибуту та сеттерів (*Setters*) — для його встановлення. При такому підході рядків коду, в яких розробник напряму звертається до атрибуту, просто не буде у програмі. Відповідно, складнощів, пов'язаних з множинними правками коду через зміну назви атрибута об'єкта, в нас не буде.

Спробуємо застосувати цей механізм до нашого завдання. Спочатку створимо клас **Product** з атрибутом об'єкта «**discount**», а потім розглянемо, наскільки просто нам буде перейти на використання атрибуту «**discountPercentage**» зі значенням знижки у вигляді відсотків.

```
class Product:
    def __init__(self, name, price, discount=0.25):
        self.name = name
        self.price = price
        self._discount = discount

    def getDiscount(self):
        return self._discount

    def setDiscount(self, value):
        self._discount=value

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.getDiscount())}
              grn.")

    def toUSD(self, usdExchRate):
        return self.price*(1-self.getDiscount())/
              usdExchRate

item1=Product("Lipton Peach Iced", 42, 0.3)
```

```

item2=Product("Pure Apple Juice", 28)

item1.showInfo()
print(f"Price in USD$:{item1.toUSD(30)}")

item2.showInfo()
print(f"Price in USD$:{item2.toUSD(30)}")

item1.setDiscount(0.5)
print(f"New discount for {item1.name} -
      {item1.getDiscount()*100}%.")

```

Результат:

```

name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
New discount for Lipton Peach Iced – 50.0%.

```

Рисунок 69

Зверніть увагу, що ми створили наш атрибут об'єкта «**discount**» рівня доступу «**protected**» (вказавши перед іменем атрибута один символ нижнього підкреслення «**_**»), щоб воно було доступне похідним класам, але при цьому розробник розумів, що напямучо до цього атрибуту звертатися не можна, тобто рядків коду на кшталт:

```

item1._discount=0.5
print(f"New discount for {item1.name} -
      {item1._discount*100}%.")

```

в нашій програмі не буде.

Тепер, коли з'являється вимога про видалення атрибута «discount» та додавання атрибуту «discountPercentage», то ці зміни торкнуться лише самого класу **Product**:

```
class Product:
    def __init__(self, name, price,
                  discountPercentage=25):
        self.name = name
        self.price = price
        self._discountPercentage = discountPercentage

    def getDiscount(self):
        return self._discountPercentage/100

    def setDiscount(self, value):
        self._discountPercentage=value*100

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.getDiscount())}
              grn.")

    def toUSD(self, usdExchRate):
        return self.price*(1-self.getDiscount())/
              usdExchRate
```

а не усіх рядків коду, які працювали із зазначеним атрибутом:

```
item1=Product("Lipton Peach Iced", 42, 30)
item2=Product("Pure Apple Juice", 28)

item1.showInfo()
print(f"Price in USD$: {item1.toUSD(30)}")

item2.showInfo()
```

```
print(f"Price in USD$:{item2.toUSD(30)}")

item1.setDiscount(0.5)

print(f"New discount for {item1.name} -
      {item1.getDiscount()*100}%.")
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Price in USD$:0.98
name:Pure Apple Juice, price with discount:21.0 grn.
Price in USD$:0.7
New discount for Lipton Peach Iced - 50.0%.
```

Рисунок 70

Ми навіть можемо додати до сеттера перевірку коректності значення атрибута об'єкта «[discountPercentage](#)»: наприклад, щоб в якості нового значення атрибуту приймалося лише число в діапазоні від 10 до 75 (тобто розмір знижки на товар менше 10% і більше 75% встановлювати не можна):

```
def setDiscount(self,value):

    if 0.1<=value<=0.75:
        self._discountPercentage=value*100
    else:
        print(f"Discount value less than 10% or
              greater than 75% is not possible!")
```

Тоді подібний рядок коду (зі спробою встановити розмір знижки 80%):

```
item1=Product("Lipton Peach Iced", 42, 30)
item1.showInfo()
item1.setDiscount(0.8)
```

викличе повідомлення про неприпустимість такої знижки для продукту:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
Discount value less than 10% or greater than 75% is not possible!
```

Рисунок 71

Однак ми пам'ятаємо, що модифікатор «`protected`» — це лише угода про те, що напряду звертатися до «`protected`» — атрибутам не можна. Фактично наявність цього модифікатора не заважає нам або іншим програмістам звертатися до цих атрибутів:

```
item1._discountPercentage=90
item1.showInfo()
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
name:Lipton Peach Iced, price with discount:4.199999999999999 grn.
```

Рисунок 72

При цьому наявність сеттера в класі не допоможе у виключенні цієї неприпустимої ситуації.

Крім цього, ми можемо встановити неприпустиме значення розміру знижки і при виклику конструктора (наприклад, просто забувши, що ми вже перейшли від «`discount`» до «`discountPercentage`» з іншими одиницями

вимірювання, і що при виклику конструктора потрібно вказувати % знижки, тобто значення від 10 до 75):

```
item2=Product("Pure Apple Juice", 28, 0.8)
item2.showInfo()
```

Тут ми отримуємо одразу дві помилки: не ті одиниці виміру для знижки (число замість відсоткового значення) та неприпустимий розмір знижки (фактично ми ж хотіли встановити 80% знижки, що не передбачено для наших товарів). Жодна з цих проблемних ситуацій не була виявлена сеттером:

```
name:Pure Apple Juice, price with discount:27.776 grn.
```

Рисунок 73

І, нарешті, ми все ще не виконали вимогу нашого замовника щодо загальнодоступності атрибуту об'єкта «discountPercentage» для будь-якого коду, який «працюватиме» з класом Product.

Саме у зв'язку із зазначеними проблемами (вірніше, з їх «не вирішенням» сеттерами та гетерами) зазначений підхід не є поширеним в Python.

Однак більш істотним недоліком є той факт, що всі рядки коду, в яких програміст традиційно міг використовувати для звернення до атрибуту виразу типу `obj.PropName`, мають бути змінені на `obj.getPropName`. Так само і замість `obj.PropName = value`, доведеться використовувати `obj.setPropName(value)`. Ця незручність може призвести до значної кількості помилок, оскільки

порушується традиційний механізм для роботи з атрибутами об'єктів (причому при роботі з `public`-атрибутами ми дотримуємося стандартного підходу, що ще більше посилить плутанину).

6.4. Властивості — `property()`

На відміну від Java та C++, в Python існує спосіб зручного вирішення вищезгаданої проблеми — це використання `property()`. Цей підхід дозволяє створювати методи, які «ведуть себе» як властивості, не порушуючи традиційного підходу роботи з ними. Фактично, функція `property()` дозволяє створювати керовані атрибути об'єкта, уникаючи необхідності роботи з гетерами та сеттерами. Така функція є вбудованою, тому нам не потрібно виконувати жодні кроки, пов'язані з імпортом.

Загальний синтаксис `property()`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Перші два аргументи приймають функціональні об'єкти (найчастіше імена методів поточного класу), які виконуватимуть роль геттера (`fget`) та сеттера (`fset`). Таким чином, за допомогою аргументу `fget` ми задаємо ім'я методу, який викликатиметься при використанні виразу типу `obj.PropName`, а аргумент `fset` задає ім'я методу, яке викликається виразом `obj.PropName = value`.

Таким чином, за допомогою `property()` ми можемо прикріпити геттер і сеттер до потрібної властивості класу, приховуючи таким чином внутрішню логіку (реалізацію) цієї властивості та забезпечуючи стабільний API для будь-якої необхідної модифікації.

Використовуючи `property()` ми також можемо вказати спосіб обробки процесу видалення атрибуту і надати відповідний рядок документації для нього.

Не завжди зручно використовувати таку довгу назву терміна, як «керовані атрибути об'єкта», тому для скорочення їх часто називають просто «властивостями» (через використання функції `property()` для їх створення).

Розглянемо їх використання на прикладі з класом `Product`.

```
class Product:
    def __init__(self, name, price, discountPercentage=25):
        self.name = name
        self.price = price
        self.__discountPercentage = discountPercentage

    def getDiscount(self):
        return self.__discountPercentage

    def setDiscount(self, value):
        if 0.1<=value<=0.75:
            self.__discountPercentage=value*100
        elif 10<=value<=75:
            self.__discountPercentage=value
        else:
            print(f"Discount value less than 10% or
                  greater than 75% is not possible!")

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.getDiscount()/100)}
              grn.")

    def toUSD(self, usdExchRate):
        return self.price*(1-self.getDiscount())/
              usdExchRate
```

```
discountPercentage = property(
    fget=getDiscount,
    fset=setDiscount
)
```

Спочатку ми змінили тип модифікатора на `private`, щоб уникнути ситуацій прямого звернення до властивості, при якому не відбувається виклик методу `getDiscount()` — проблема, з якою ми стикалися при некоректному поводженні з `protected`-властивостями.

Також ми забезпечили вимогу загальної властивості «`discountPercentage`», тобто можливість використання виразу типу `obj.discountPercentage` або `obj.discountPercentage = value`:

```
item1=Product("Lipton Peach Iced", 42, 30)
item1.showInfo()

print(item1.discountPercentage)
item1.discountPercentage=0.25
print(item1.discountPercentage)
item1.discountPercentage=60
print(item1.discountPercentage)
item1.discountPercentage=85
print(item1.discountPercentage)
```

Результат:

```
name:Lipton Peach Iced, price with discount:29.4 grn.
30
25.0
60
Discount value less than 10% or greater than 75% is not possible!
60
```

Рисунок 74

Як бачимо, незалежно від одиниць виміру (0.25 чи 60) розмір знижки встановлюється правильно. Також програмою (неявно викликаним методом `setDiscount()` у виразі `obj.discountPercentage=value`) виявляється некоректне значення (85) розміру знижки, яке не встановлюється в нашому товарі (збережено попереднє значення знижки в -60%).

Однак при використанні некоректного значення для аргументу-знижки при виклику конструктора:

```
item2=Product("Pure Apple Juice", 28, 0.5)
```

виклик методу `setDiscount()` не відбувається автоматично.

Для усунення цього моменту, який може призвести до встановлення некоректного значення розміру знижки, внесемо зміни до конструктора класу:

```
def __init__(self, name, price, discountPercentage=25):
    self.name = name
    self.price = price
    self.__discountPercentage = discountPercentage*100
    if 0<discountPercentage<1
    else discountPercentage
```

Тепер розмір знижки буде правильно встановлюватися конструктором як при використанні відсоткового значення (наприклад, 30), так і при використанні числа (наприклад, 0.5, тобто 50% знижки):

```
item2=Product("Pure Apple Juice", 28, 0.5)
item2.showInfo()
print(item2.discountPercentage)
```

```
name:Pure Apple Juice, price with discount:14.0 grn.
50.0
```

Рисунок 75

Нам залишилося познайомитись ще з двома аргументами функції `property()`: `fdel` та `doc`.

Іноді нам потрібно видалити існуючий атрибут об'єкта, причому саме у конкретного екземпляра класу, а не самого класу. Наприклад, у товару «Coffee» немає знижки взагалі. І щоб унеможливити (наприклад, випадкове) встановлення знижки для цього товару, ми можемо просто видалити атрибут «`discountPercentage`» у конкретного об'єкта-товару «Coffee» (вже після його створення за допомогою конструктора класу).

За допомогою аргументу `fdel` ми можемо визначити метод класу, який буде викликатись у разі видалення властивості через вираз `del obj.ім'я_властивості`. А аргумент `doc` може бути корисним для створення опису властивості, яку можна побачити при використанні вбудованої функції `help()`.

Для властивості «`discountPercentage`» створимо метод `delDiscount()`, який виводитиме повідомлення про неможливість видалення даної властивості. Для виклику методу `delDiscount()`, при використанні виразу `del obj.discountPercentage`, встановимо його як значення аргументу `fdel` у функцію `property()` для керованого атрибуту `discountPercentage`.

```
class Product:
    def __init__(self, name, price,
                  discountPercentage=25):
```

```

        self.name = name
        self.price = price
        self.__discountPercentage = discountPercentage*100
            if 0<discountPercentage<1
            else discountPercentage

def getDiscount(self):
    return self.__discountPercentage

def setDiscount(self,value):
    if 0.1<=value<=0.75:
        self.__discountPercentage=value*100
    elif 10<=value<=75:
        self.__discountPercentage=value
    else:
        print(f"Discount value less than 10% or
            greater than 75% is not possible!")

def delDiscount(self):
    print("It is impossible to delete this
        property!")

def showInfo(self):
    print (f"name:{self.name}, price with discount:
        {self.price*(1-self.getDiscount()/100)}
        grn.")

def toUSD(self,usdExchRate):
    return self.price*(1-self.getDiscount())/
        usdExchRate

discountPercentage = property(
    fget=getDiscount,
    fset=setDiscount,
    fdel=delDiscount,
    doc="Product discount property."
)

```

```

item1=Product("Lipton Peach Iced", 42,
               "Super iced tea!", 30)
del item1.discountPercentage
help(Product.discountPercentage)

```

Результат:

```

It is impossible to delete this property!
Help on property:

    Product discount property.

```

Рисунок 76

Таким чином, використання механізму керованих атрибутів `property()` дозволяє легко вносити зміни у внутрішню реалізацію властивостей класу, не порушуючи стабільності загальнодоступного API-класу.

Крім застосування вбудованої функції `property()` для створення керованих атрибутів, в Python передбачений альтернативний спосіб — використання декоратора `@property`. Підхід з використанням декоратора для створення керованих атрибутів вимагає спочатку (першим) визначити метод — геттер. Перед його визначенням (рядком вище) потрібно використовувати декоратор `@property`. При цьому ім'я методу-геттера має співпадати з ім'ям створюваного керованого атрибуту. Того атрибута, для якого можливий загальний доступ при подальшому використанні класу.

Після створення методу-геттера потрібно створити метод-сеттер, ім'я якого має точно збігатися з ім'ям вище визначеного методу-геттера. А перед визначенням слід зазначити такой декоратор: `@propertyName.setter`.

Давайте перепишемо наш клас **Product** з використанням декораторів:

```
class Product:
    def __init__(self, name, price,
                  discountPercentage=25):
        self.name = name
        self.price = price
        self.__discountPercentage =
            discountPercentage*100
            if 0<discountPercentage<1
            else discountPercentage

    @property
    def discountPercentage(self):
        return self.__discountPercentage

    @discountPercentage.setter
    def discountPercentage(self, value):
        if 0.1<=value<=0.75:
            self.__discountPercentage=value*100
        elif 10<=value<=75:
            self.__discountPercentage=value
        else:
            print(f"Discount value less than 10% or
                  greater than 75% is not possible!")

    @discountPercentage.deleter
    def discountPercentage(self):
        print("It is impossible to delete this
              property!")

    def showInfo(self):
        print (f"name:{self.name}, price with discount:
              {self.price*(1-self.__discountPercentage/
              100)} grn.")

    def toUSD(self, usdExchRate):
```



```

        return self.price*(1-self.
            __discountPercentage)/usdExchRate
item1=Product("Lipton Peach Iced", 42, 30)
del item1.discountPercentage

```

Результат:

It is impossible to delete this property!

Рисунок 77

Як бачимо, при використанні такого способу нам не потрібно створювати методи з іменами типу наших попередніх `getDiscount()`, `setDiscount()`, `delDiscount()`, і далі реалізовувати їх «зв'язку» з відповідними параметрами функції `property()`. Тепер у нас є три методи з одним і тим же чітким та описовим ім'ям, яке збігається з ім'ям створюваного керованого атрибуту `discountPercentage`.

6.5. Дескриптори

У попередніх прикладах ми розглядали використання функції `property()` та декоратора `@property` для створення керованих атрибутів об'єкта (або, як ми їх називали, властивостей — `property`).

Керовані атрибути об'єкта — це властивості об'єкта, які мають «пов'язану» поведінку, тобто це такі властивості, при зверненні яких виконуються деякі алгоритми (реалізовані всередині методів класу). Як ми вже знаємо, у Python передбачено три варіанти звернення до якості:

- отримання значення властивості, використовуючи вираз `objName.propertyName`;

- зміна (встановлення) значення властивості: `objName.propertyName = value`;
- видалення атрибуту: `del objName.propertyName`.

Кожен із перелічених способів роботи з властивістю можна «пов'язати» з необхідним методом класу, тобто перевизначити поведінку, пов'язану з певним варіантом доступу. В Python таку «прив'язку» методів класу до властивості ми можемо реалізувати або за допомогою функції `property()`, або декоратором `@property`. Для того, щоб зрозуміти, як все це працює «з середини», нам потрібно познайомитися з поняттям дескриптора.

Саме дескриптори є основою таких механізмів і можливостей, як функція `property()`, декоратори `@staticmethod` (статичні методи) та `@classmethod` (методи класу), а також функції `super()`. Фактично, кожна з керованих властивостей є дескриптором. Давайте розберемося, чому це так.

Дескриптор — це об'єкт, для якого визначено хоча б один із методів `__get__()`, `__set__()` або `__delete__()`.

Коли цей об'єкт (фактично, посилання на нього) ми надаємо властивості класу, то в результаті отримуємо керований атрибут об'єкта (властивість з «пов'язаною поведінкою»). Весь цей процес реалізується через механізм **протоколу дескрипторів**.

В Python передбачено три методи, з яких і складається протокол дескриптора (це відповідає кількості способів звернення до атрибуту, переліченим вище):

- `__set__(self, obj, value)` — даний метод викликається при зміні значення властивості з допомогою виразу `objName.propertyName = value`;

- `__get__(self, obj, type=None)` — викликається при отриманні значення властивості, тобто при виразі `objName.propertyName`;
- `__delete__(self, object)` — викликається щоразу, коли використовується оператор `del` для видалення даної властивості.

Клас, в якому визначено лише метод `__get__`, називається **дескриптором без даних**, а якщо клас реалізує методи `__get__` і `__set__`, тоді це **дескриптор даних**.

Давайте створимо простий клас-дескриптор і на прикладі розберемося, що являють собою параметри в методах протоколу дескриптора.

```
class MyDescriptor1:
    def __init__(self, name=""):
        print("Descriptor's __init__ was started...")
        self.name = name

    def __get__(self, obj, objtype):
        print(f"Descriptor's __get__(instance={obj},
              objtype={objtype}) was started...")
        return "{} was processed".format(self.name)

    def __set__(self, obj, name):
        print(f"Descriptor's __set__(instance={obj},
              name={name}) was started...")
        if isinstance(name, str):
            self.name = name
        else:
            print("Name should be string")
```

Наш дескриптор `MyDescriptor1` — це дескриптор даних, який встановлює та повертає значення у звичайному

режимі (як от сеттери та гетери, розглянуті нами раніше). Також передбачено виведення повідомлення, що відображає процес запуску певного методу протоколу дескриптора та його параметрів.

Тепер створимо клас `User`, в якому буде керований атрибут об'єкта `userName`, поведінка якого «контролюється» дескриптором `MyDescriptor1`.

```
class User:
    userName = MyDescriptor1()

user1 = User()
user1.userName = "Jack"
print(user1.userName)
```

Результат:

```
Descriptor's __init__ was started...
Descriptor's __set__(instance=<__main__.User object
    at 0x10b763880>, name=Jack) was started...
Descriptor's __get__(instance=<__main__.User object
    at 0x10b763880>, objtype=<class '__main__.User'>)
    was started...
Jackjtfas processed
```

Рисунок 78

Отже, коли ми створили екземпляр-клас `User` (об'єкт `user1`) в процесі ініціалізації керованого атрибута об'єкта `userName`, був запущений конструктор дескриптора даних `MyDescriptor1`.

Далі під час встановлення нового значення для `userName` був запущений метод `__set__()` нашого дескриптора даних. Тут ми можемо бачити, що другий параметр методу `obj`

містить відповідний екземпляр класу `User` (посилання на об'єкт `user1`), а останній параметр `name` — значення, яке ми надаємо в `userName`.

І останнім у нас було запущено метод дескриптора `__get__()`, коли ми виводимо значення `userName` у консоль. Як і в методі `__set__()`, другий параметр методу `obj` містить посилання на об'єкт `user1`, а от третій параметр (`objtype`) — це клас, екземпляром якого є об'єкт `user1`.

Давайте спробуємо встановити в якості значення число для `userName`:

```
user1 = User()
user1.userName = "Jack"
print(user1.userName)
user1.userName = 111
print(user1.userName)
```

Результат:

```
Descriptor's __init__ was started...
Descriptor's __set__(instances__main__.User object
  at 0x10e63c8b0>, name=Jack) was started...
Descriptor's __get__(instances__main__.User object
  at 0x10e63c8b0>, objtype=<class '__main__.User'>)
  __was__started...
Jack was processed
Descriptor's __set__(instances__main__.User object at
  0x10e63c8b0>, name=111) was started...
Name should be string
Descriptor's __get__(instances__main__.User object at
  0x10e63c8b0>, objtype=<class '__main__.User'>)
  was started...
Jack was processed
```

Рисунок 79

Як бачимо, наш дескриптор коректно опрацював цю помилкову ситуацію.

Тепер припустимо, що нам потрібно додати до нашого класу `User` інформацію про вік користувача. Звичайно, нам також необхідно контролювати введення значень для цієї властивості (наприклад, нехай діапазон допустимих значень для віку користувача буде `[18,75]`). Для «керування» взаємодією з цією властивістю створимо другий дескриптор даних `MyDescriptor2` і внесемо відповідні зміни до класу `User`.

```
class MyDescriptor1:
    def __init__(self, name=""):
        print("Descriptor's __init__ was started...")
        self.name = name

    def __get__(self, obj, objtype):
        print(f"Descriptor's __get__ (instance={obj},
              objtype={objtype}) was started...")
        return "{} was processed".format(self.name)

    def __set__(self, obj, name):
        print(f"Descriptor's __set__ (instance={obj},
              name={name}) was started...")
        if isinstance(name, str):
            self.name = name
        else:
            print("Name should be string")

class MyDescriptor2:
    def __init__(self, age = 18):
        self.age = age

    def __set__(self, obj, age):
        if not 18 <= age <= 75:
            print('Valid age must be in [18, 75]')
```

```

        else:
            self.age = age
    def __get__(self, obj, objtype):
        return self.age
class User:
    userName = MyDescriptor1()
    userAge=MyDescriptor2()

user1 = User()
user1.userName = "Jack"
print("User name: {}".format(user1.userName))
print("User age: {}".format(user1.userAge))
user1.userAge = 4
print("User age: {}".format(user1.userAge))
user1.userAge = 100
print("User age: {}".format(user1.userAge))
user1.userAge = 25
print("User age: {}".format(user1.userAge))

```

Результат:

```

Descriptor's __init__ was started...
Descriptor's __set__ (instance=<__main__. User object
    at 0x1073868e0>, name=Jack) was started...
Descriptor's __get__ (instances__main__.User object
    at 0x1073868e0>, objtype=<class '__main__.User'>)
    was started...,
User name: Jack was processed
User age: 18
Valid age must be in [18, 75]
User age: 18
Valid age must be in [18, 75]
User age: 18
User age: 25

```

Рисунок 80

Як бачимо, обидві спроби встановити некоректне значення для віку користувача були зупинені дескриптором.

Однак, виникають два суттєві питання:

1. А чим, власне, використання дескрипторів краще, ніж використання функції `property()` або декоратора `@property`? Адже, використовуючи їх механізми, ми раніше контролювали коректність значень властивостей.
2. Чи можна «прив'язати» роботу конструктора класу (з контрольованими властивостями) до дескриптора? Наш клас `User` поки що немає конструктора взагалі, і це значно знижує зручність його використання.

Основна перевага дескрипторів у порівнянні з `property()` (або `@property`) — це керування роботою декількох властивостей класу, які мають схожу поведінку (або обмеження на значення).

Припустимо, що у класу `User` є два окремих поля для зберігання ім'я: `firstName`, `lastName`. Кожне з цих полів не має бути порожнім або менше певної довжини (3 символи для імені та 4 для прізвища). Реалізуймо цей клас, використовуючи декоратор `@property`:

```
class User:
    def __init__(self, firstName, lastName):
        self.__firstName = firstName
        self.__lastName = lastName

    @property
    def firstName(self):
        return self.__firstName
```



```

@firstName.setter
def firstName(self, value):
    if not isinstance(value, str):
        print('The first name must be a string!')
    elif len(value) == 0:
        print('The first name cannot be empty!')
    elif len(value) < 3:
        print('Too short value for the first name!')

    self.__firstName = value

@property
def lastName(self):
    return self.__lastName

@lastName.setter
def lastName(self, value):
    if not isinstance(value, str):
        print('The last name must be a string!')
    elif len(value) == 0:
        print('The last name cannot be empty!')
    elif len(value) < 4:
        print('Too short value for the last name!')
    else:
        self.__lastName = value

```

У цій реалізації «гетери» повертають значення властивостей `firstName` і `lastName`, а «сеттери» перевіряють їх нові значення перед наданням до властивостей.

```

user1=User("Joe", "Black")
print(f"User1: {user1.firstName} {user1.lastName}")

user1.firstName=""
user1.firstName="AB"

```

```
user1.lastName=""
user1.lastName="ABC"
```

Код виконує свої функції.

```
User1: Joe Black
The first name cannot be empty!
Too short value for the first name!
The last name cannot be empty!
Too short value for the last name!
```

Рисунок 81

Проте, ми можемо легко помітити надмірність у коді: не лише логіка, а й її реалізація у «сеттерів» властивостей, як і «геттерів», повністю збігається.

А тепер уявімо, що в нас є багато властивостей рядкового типу. Тоді рівень надмірності коду ще більше зросте.

Більш правильним рішенням буде створити дескриптор, який реалізує керування рядковими властивостями:

```
class StrDescriptor:
    def __init__(self, minLen, name=""):
        self.name=name
        self.minLen=minLen

    def __get__(self, obj, objtype):
        return self.name

    def __set__(self, obj, value):
        if not isinstance(value, str):
            print('The value must be a string!')
        elif len(value) == 0:
            print('The value cannot be empty!')
        elif len(value) < self.minLen:
            print('Too short value!')
```

```

else:
    self.name = value

```

І далі використовувати його при визначенні керованих атрибутів класу `User`.

```

class User:
    firstName=StrDescriptor(3)
    lastName=StrDescriptor(4)

    def __init__(self, firstName, lastName):
        self.firstName = firstName
        self.lastName = lastName

user1=User("Joe", "Black")
print(f"User1: {user1.firstName} {user1.lastName}")

user1.firstName=""
user1.firstName="AB"

user1.lastName=""
user1.lastName="ABC"

user2=User("AB", "ABC")

```

Результат:

```

User1: Joe Black
The value cannot be empty!
Too short value!
The value cannot be empty!
Too short value!
Too short value!
Too short value!

```

Рисунок 82

Як ми бачимо, за отриманим результатом, не тільки рядки коду, що містять явні звернення до властивостей `firstName` і `lastName`, обробляються відповідними дескрипторами — були «зупинені» ситуації з надання некоректних значень (порожній і занадто короткий рядки). Також дескриптори не допустили встановлення таких самих неправильних значень конструктором (`User("AB", "ABC")`). Це відбувається тому, що всередині тіла конструктора знаходяться вирази типу `objName.propertyName = value`, які викликають запуск методу `__set__()` протоколу дескриптора, який якраз і містить відповідні перевірки.

Таким чином, в ситуаціях, коли в класі є декілька керованих властивостей, логіка роботи яких (або перевірки їх значень) однакова, тоді використання дескрипторів допоможе зробити програму більш структурованою і дозволить уникнути дублювання коду.

7. Метакласи

7.1. Модель метакласів

Напевно, кожен із нас неодноразово чув питання: «А чи можуть програми самі генерувати код (створювати інші програми) замість програмістів?». І, звичайно, ми знаємо, що відповідь на це запитання ствердна. Більше того, існує окремий підхід — метапрограмування.

Метапрограмування — це вид програмування, який дозволяє нам розробляти код, що керує іншим кодом, у тому числі й генерувати його.

Фреймворки, «інтелектуальні» функції різних бібліотек, що спрощують роботу з кодом і його генерацію, активно використовують прийоми метапрограмування. І ми з вами також вже використовували одну з його форм — **декоратори**, які дозволяють нам розширити функціональність вже існуючої функції без прямої зміни коду в її тілі.

Крім декораторів, в Python передбачено ще одну форму метапрограмування — метакласи.

Потрібно сказати, що метакласи підтримуються не всіма об'єктно орієнтованими мовами програмування. Причому ті мови програмування, які забезпечують механізми метакласів, суттєво різняться за способом їх реалізації.

Метаклас — це спеціальний тип класу, екземплярами (об'єктами) якого є класи. Тому можна сказати, що метаклас визначає (задає) поведінку класу так само, як і клас задає поведінку об'єктів (своїх екземплярів).

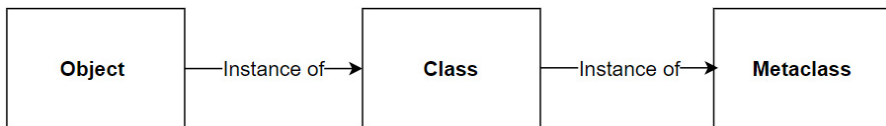


Рисунок 83

Ми знаємо: все в Python є об'єктом. І той факт, що клас є об'єктом, ще раз доводить істинність цього твердження. Для того, щоб краще зрозуміти особливості, процеси створення та роботи метакласів, давайте ще раз згадаємо основні аспекти звичайних класів, їх об'єктів та зв'язків між ними.

Припустимо, у нас є клас `MyClass1` (без властивостей і методів) і його екземпляр `myObj1`:

```
class MyClass1():
    pass

myObj1 = MyClass1()
print(myObj1)
```

Виведення об'єкта `myObj1` показує, що він дійсно є екземпляром класу `MyClass1`:

```
<__main__.MyClass1 object at 0x00000216BC4D81F0>
```

Рисунок 84

Також ми знаємо, що інформацію про тип даних можна отримати за допомогою вбудованої функції `type()`. Спробуємо використати її для визначення типу нашого об'єкта (адже клас — це тип даних):

```
print(type(myObj1))
```

Отриманий результат містить назву класу, який створив даний екземпляр, тобто тип даних змінної `myObj1` — `MyClass1`:

```
<class '__main__.MyClass1'>
```

Рисунок 85

Отже, згідно з логікою, представленою на рисунку вище, ми зможемо скористатися функцією `type()` для визначення типу нашого класу `MyClass1` (метакласу, екземпляром якого він є):

```
print(type(MyClass1))
```

або так:

```
print(type(type(myObj1)))
```

Результат досить несподіваний:

```
<class 'type'>
```

Рисунок 86

Ми побачили назву вбудованого метакласу (`type`), який створив об'єкт з ідентифікатором `MyClass1`, коли ми використовували ключове слово `class` для його оголошення. А що щодо метакласу для вбудованих типів (класів `str`, `int`, `list`)? Проекспериментуємо ще:

```
myNumber=10
myStr="Admin"
myList=[1,2,3]
```

```

print(type(myNumber))

print(type(type(myNumber)))

print(type(myStr))
print(type(type(myStr)))

print(type(myList))
print(type(type(myList)))

```

Результат:

```

<class 'int'>
<class 'type'>
<class 'str'>
<class 'type'>
<class 'list'>
<class 'type'>

```

Рисунок 87

Отримані результати показують, що `type` є метакласом для класів `int`, `str`, `list`, які є вбудованими класами в Python. Таким чином, `type` є метакласом за замовчуванням.

Вище ми вже використовували вбудовану функцію `type()` для визначення типу об'єкта, який ми передавали як аргумент. Однак, функціональність `type` набагато «ширша»: вона може також повертати новий тип класу (фактично метаклас). Для цього потрібно викликати її не з одним, а з трьома аргументами.

Загальний синтаксис:

```

type(ClassName, (ParentClassesTuple), propDict)

```


- `ClassName` — ім'я нового класу (рядок);
- `ParentClassesTuple` — кортеж, що містить імена базових класів, від яких успадковується створюваний клас `ClassName` (у разі відсутності успадкування — порожній `()`);
- `propDict` — словник, що містить імена атрибутів та методів створюваного класу.

Давайте розберемося на прикладі, як це відбувається:

```
MyClass = type('MyClass', (BaseClass), clsDict)
```

Об'єкт `MyClass` є класом, який є похідним від класу `BaseClass`, а його властивості та методи вказані у словнику `clsDict` (визначеному, наприклад, раніше).

Визначений нами раніше клас `MyClass1` можна створити, використовуючи `type()`. Давайте також створимо екземпляри об'єктів в обох випадках (для класу `MyClass1`, створеного з допомогою ключового слова `class`, і для класу `MyClass1`, отриманого в результаті роботи функції `type()`), і перевіримо їх типи:

```
class MyClass1():
    pass
myObj1 = MyClass1()
print(myObj1)
print(type(myObj1))
```

Результат:

```
<__main__.MyClass1 object at 0x0000016FB38BBB20>
<class '__main__.MyClass1'>
```

Рисунок 88

```
MyClass1=type("MyClass1", (), {})

myObj1 = MyClass1()
print(myObj1)
print(type(myObj1))
```

Результат:

```
<__main__.MyClass1 object at 0x0000025A65E9BB80>
<class '__main__.MyClass1'>
```

Рисунок 89

В обох випадках створені об'єкти є екземплярами класу `MyClass1`. Зверніть увагу на той факт, що `MyClass1` є змінною для зберігання посилань на клас та ім'ям класу («`MyClass1`»).

Якщо у класі потрібно визначити властивості та методи, то потрібно заповнити третій параметр — словник з ключами та значеннями.

Додаймо до нашого класу `MyClass1` метод. Для цього потрібно заздалегідь (до виклику функції `type()`) визначити функцію, яка реалізує необхідну логіку майбутнього методу класу. Синтаксис оголошення цієї функції повністю подібний до синтаксису оголошення методу класу — першим параметром цієї функції має бути стандартний параметр `self` (посилання на об'єкт).

```
def method1(self):
    print(self.prop1)

MyClass2=type("MyClass2", (), {"prop1":"Hello",
                                "method1":method1})
```

```
myObj2 = MyClass2()
print(myObj2.prop1)
myObj2.method1()
```

Результат:

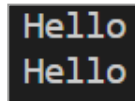


Рисунок 90

Як бачимо, додавання методу в клас практично нічим не відрізняється від додавання властивості. Після визначення потрібної функції ми просто призначили її як значення властивості — імені методу (у третьому параметрі — словнику під час виклику функції `type()`). Більше того, вже після створення класу ми можемо додавати до нього атрибути та методи, якщо це необхідно:

```
MyClass2.prop2=100

myObj3 = MyClass2()
print(myObj2.prop2) #100
```

Оскільки класи в Python є об'єктами, тоді можна динамічно створювати клас і змінювати його вміст. Проте, саме це (процес створення класу метакласом `type`) і відбувається під час виконання оператора `class`, який ми використовували раніше. У чому ж суть? У тому, що, розібравшись зараз в усіх деталях цього механізму, ми можемо його кастомізувати — створювати власні метакласи.

7.2. Метод-конструктор `__new__()`

Отже, ми вже знаємо як за допомогою функції `type()` динамічно створювати класи з потрібними особливостями структури та поведінки. Однак, у прикладах вище ми змінювали класи (додаючи до них властивості та методи) вже після його створення.

Тим не менш, існують ситуації, коли необхідно реалізувати автоматичну зміну класу під час його створення, або створення класу залежить від виконання якихось умов. Наприклад, нам потрібно створювати класи з обов'язковим атрибутом «`id`». І якщо у визначенні класу немає такого атрибуту, то автоматично додавати його. Або перед створенням класу необхідно перевірити, що кількість способів у його визначенні не перевищує певний поріг. І тільки класи, що задовольняють цю умову, можуть бути створені в програмі.

Для реалізації описаних ситуацій (автоматично, з допомогою програми, а не перегляду із наступними «ручними» корекціями коду) потрібно створити користувацький метаклас, похідний від метакласу `type`. І в цьому користувацькому метакласі перевизначити вже відомий нам «магічний» метод `__new__()`.

Далі нам необхідно налаштувати процес створення класу за потрібним нам шаблоном, особливості якого описані в метакласі. Для цього потрібно задати параметру `metaclass` (у визначенні класу) ім'я нашого користувача метакласу в якості значення.

```
class MyMetaClass1(type):  
    pass
```

```
class MyClass1(metaclass=MyMetaClass1):
    pass
```

Давайте перевіримо тип створеного метакласу `MyMetaClass1` та класу `MyClass1`:

```
print(type(MyMetaClass1))
print(type(MyClass1))
```

Результат:

```
<class 'type'>
<class '__main__.MyMetaClass1'>
```

Рисунок 91

Тепер перейдемо до деталей процесу створення користувацького метакласу. Базовий метаклас (метаклас за замовчуванням) `type` визначає «магічні» методи, які в нових користувацьких метакласах (похідних від `type`) можуть бути перевизначені для реалізації необхідних особливостей:

- `__new__()` — створює екземпляр класу, заснованого на даному метакласі. Перевизначення цього методу дозволяє керувати процесом створення класу. Наприклад, можна налаштувати параметр-словник для функції `type`, який задає властивості та методи створюваного класу. Або визначити (задати обмеження) на параметр-кортеж, який зберігає базові класи у тому випадку, коли створюваний клас успадковується

ся. Результат, що повертається методом `__new__()`, зазвичай є екземпляром класу (визначається як `cls`).

- `__init__()` — даний метод зазвичай викликається після створення об'єкта для його ініціалізації.
- `__call__()` — перевизначення даного методу дозволяє налаштувати поведінку процесу, що відбувається при виклику конструктора створюваного класу.

Будь-який користувальницький метаклас має визначити свій метод `__new__()` та/або метод `__init__()`, щоб повертати необхідний клас користувача, на створення (налаштування, зміну) якого він орієнтований. Більш традиційним вважається перевизначення методу `__new__()`.

Розглянемо з прикладу: перевизначимо у нашому метакласі `MyMetaClass1` метод `__new__()`, і додамо до нього вивід повідомлення, щоб побачити момент спрацьовування методу.

```
class MyMetaClass1(type):

    def __new__(cls, name, bases, dict):
        print("'Hello' from __new__()!")
        print("Type of the class being created ", cls)
        print("Name of the class being created:", name)
        print("Tuple of base classes: ", bases)
        print("Dictionary of attr.: ", dict)

        return type.__new__(cls, name, bases, dict)

class MyClass1(metaclass=MyMetaClass1):
    attr=100
```

Результат:

```
'Hello' from __new__()!
Type of the class being created <class '__main__.MyMetaClass1'>
Name of the class being created: MyClass1
Tuple of base classes: ()
Dictionary of attr.: {'__module__': '__main__', '__qualname__':
                      'MyClass1', 'attr': 100}
```

Рисунок 92

Метод `MyMetaClass1.__new__()` був викликаний в момент визначення класу `MyClass1` (з допомогою ключового слова `class`). У тілі нашого перевизначеного методу `__new__()` ми викликали метод базового метакласу `type.__new__()`, щоб створити наш користувацький метаклас `MyMetaClass1`.

Однак, цей рядок коду не зовсім коректний для ООП (ми знаємо, що методи базових класів у класах-нащадках традиційно викликаються з допомогою `super()`). Давайте внесемо необхідні корективи до нашого метакласу `MyMetaClass1`:

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        print("'Hello' from __new__()!")
        print("Type of the class being created ", cls)
        print("Name of the class being created:", name)
        print("Tuple of base classes: ", bases)
        print("Dictionary of attr.: ", dict)

        return super().__new__(cls, name, bases, dict)
```

Наразі наш метаклас `MyMetaClass1` не має особливої користі (крім демонстрації моменту виклику перевизначеного методу `__new__()`). Давайте додамо функціональності:

припустимо, що кожен клас, який визначається в коді, має містити властивість «`id`».

Насправді рішення дуже просте — ми додамо в метод `MyMetaClass1.__new__()` перевірку: чи є в словнику `dict` такий ключ:

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class '{name}'! ")
        else:
            print(f"Class '{name}' is creating...")
            return super().__new__(cls, name, bases, dict)

class MyClass1(metaclass=MyMetaClass1):
    attr=100

class MyClass2(metaclass=MyMetaClass1):
    attr=100
    id=1
```

Результат:

```
No 'id' attr. in the class 'MyClass1'!
Class 'MyClass2' is creating...
```

Рисунок 93

Додаймо ще вимогу про кількість методів: не більше трьох у кожному визначеному класі:

```
class MyMetaClass1(type):
    def __new__(cls, name, bases, dict):
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class '{name}'! ")
```



```

        else:
            nMethods = {key: value for key, value
                          in dict.items()
                          if callable(value)}

            if len(nMethods)>3:
                print(f"More than 3 methods in
                      the class '{name}'! ")
            else:
                print(f"Class '{name}' is creating...")
                return super().__new__(cls, name,
                                       bases, dict)

class MyClass1(metaclass=MyMetaClass1):
    attr=100

class MyClass2(metaclass=MyMetaClass1):
    attr=100
    id=1
    def method1(self):
        pass
    def method2(self):
        pass
    def method3(self):
        pass

class MyClass3(metaclass=MyMetaClass1):
    id=2
    def method1(self):
        pass
    def method2(self):
        pass
    def method3(self):
        pass
    def method4(self):
        pass

```

Результат:

```
No 'id' attr. in the class 'MyClass1'!
Class 'MyClass2' is creating...
More than 3 methods in the class 'MyClass3'!
```

Рисунок 94

Ми можемо також додавати потрібні атрибути до класу (наприклад, в разі їх відсутності у визначеному класі, як у прикладі з «[id](#)»):

```
class MyMetaClass2(type):
    def __new__(cls, name, bases, dict):
        resultCls=super().__new__(cls, name, bases, dict)
        if 'id' not in dict.keys():
            print(f"No 'id' attr. in the class '{name}'!
                  Let's add it.")
            resultCls.id=0
        return resultCls

class User(metaclass=MyMetaClass2):
    attr=100

obj=User()
print(obj.id)
```

Результат:

```
No 'id' attr. in the class 'User'! Let's add it.
0
```

Рисунок 95

А якщо нам потрібно додавати різні атрибути для різних класів? Для цього необхідно передати їх у метаклас з допомогою параметрів, використовуючи ****kwargs**:

```

class MyMetaClass3(type):
    def __new__(cls, name, bases, dict, **kwargs):
        resultCls=super().__new__(cls, name,
                                   bases,dict)

        if kwargs:
            for key, value in kwargs.items():
                setattr(resultCls, key, value)
        return resultCls

class User(metaclass=MyMetaClass3, firstName='Joe',
           age=15):
    attr=100

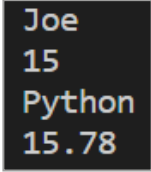
class Book(metaclass=MyMetaClass3, title='Python',
           price=15.78):
    attr=100

obj1=User()
print(obj1.firstName)
print(obj1.age)

obj2=Book()
print(obj2.title)
print(obj2.price)

```

Результат:



```

Joe
15
Python
15.78

```

Рисунок 96

На основі кожного з переданих, у визначенні, класів-аргументів (зі значеннями) в ці класи, в момент їх створення були додані відповідні атрибути.

Таким чином, основне призначення користувацьких метакласів, створених за допомогою «магічного» методу `__new__()` — це зміна (перетворення) або перевірка класу в момент його створення для того, щоб він задовольняв певний контекст.

7.3. Протоколи в Python

Досить часто при обговоренні мов з динамічною типізацією згадується термін «качина типізація» (*Duck Typing*).

Качина типізація (*Duck Typing*) — це концепція (підхід до системи типізації), який використовується в мовах програмування з динамічною типізацією (наприклад, Python, PHP, Javascript і т. д.).

При такому підході, певний тип даних (або клас) менш важливий, ніж властивості та методи, які у ньому визначені. Більше того, використовуючи концепцію *Duck Typing*, ми взагалі не контролюємо типи змінних (об'єктів). Натомість перевіряється наявність у об'єкта потрібного методу або властивості.

Назва концепції походить від фрази: «*If it looks like a duck and quacks like a duck, it's a duck*» («Якщо щось виглядає, як качка, і крякає, як качка, то це качка»). Тобто нас зовсім не цікавить (для вирішення наших завдань) внутрішній устрій та особливості даного виду птиці. Ми робимо висновок, що це качка (і ми будемо «поводитись» із цим птахом, як з качкою), ґрунтуючись на зовнішніх характеристиках та особливостях поведінки.

Розглянемо застосування качиної типізації в Python на простих прикладах:

```

number1=2
number2=5.7
print(number1+number2) #7.7
str1="Hello!"
str2="Python"
print(str1+str2) #Hello!Python

```

Оператор «+» для числових даних реалізує арифметичне додавання, а для рядкових — конкатенацію (об'єднання рядків). Інтерпретатор Python виконує перевірку типів (чи підтримується такий оператор, і що він має реалізовувати) під час виконання коду, на відміну від статичної типізації (таких як C++, Java), які виконують її під час компіляції.

Багаторазово використана нами функція `len()` обчислює довжину об'єкта залежно від його класу.

```

print(len("Student")) #7
print(len(["Python", "C#", "JavaScript"])) #3
print(len({"firstName": "Jane", "lastName": "Smith"})) #2

```

Для функції `len()` не важливий тип її аргументу, суттєвим є лише факт, що аргумент (об'єкт) може викликати метод `__len__()`. Таким чином, ми можемо визначити в тілі нашого класу метод `__len__()`, виклик функції `len()` для об'єктів цього класу стане можливим:

```

class MyClass:
    def __len__(self):
        return 1000

obj=MyClass()
print(len(obj)) #1000

```

Даний підхід забезпечує нам можливості поліморфізму, підвищує гнучкість коду, дозволяючи працювати з екземплярами різних класів (абсолютно не пов'язаних один з одним) за допомогою одного інтерфейсу (набору загальних властивостей та методів). Однак, поряд із позитивними факторами, такий підхід має й недолік: дуже складно (а іноді й неможливо) виявити помилку некоректного використання об'єкта.

Наприклад:

```
number=1000  
print(len(number))
```

Результат:

object of type 'int' has no len()

Рисунок 97

Іноді виявлення подібної помилки можливе лише на етапі тестування (тобто в момент виконання коду).

На щастя, в Python існує клас **Protocol**, з допомогою якого ми можемо як створювати інтерфейси, так і неявно перевіряти, чи задовольняють їм об'єкти.

Давайте спочатку розберемося з поняттям «протокол».

Традиційно під протоколом розуміється деякий прийнятий або встановлений набір (кодекс) процедур або правил поведінки у будь-якій групі, організації або ситуації.

В програмуванні **«протокол»** — це набір правил, що регулюють взаємодію сутностей (наприклад, обміну або передачі даних). Пристрої та програми обмінюються між собою даними лише за певними правилами — протоколами.

Наприклад, між співробітником та організацією, в якій він працює, укладається трудовий договір. Такий контракт фактично є протоколом, що регулює взаємодію (співробітництво) між роботодавцем та працівником. Співробітник повинен виконувати закріплені за ним трудові зобов'язання, а роботодавець гарантує виплачувати заробітну плату, забезпечувати соціальний пакет та належні умови праці. Якщо працівник не виконує своїх функціональних зобов'язань, тоді він може, наприклад, не отримати заробітну плату в повному обсязі, або його можуть навіть звільнити з посади. Так само і роботодавець — за невиконання якихось умов (пунктів) протоколу може бути оштрафований і т. д.

Саме за допомогою протоколів ми реалізуємо **інтерфейс**, який є механізмом (засобом) цієї взаємодії сутностей. Ми вже використали поняття «інтерфейс» при знайомстві з принципом інкапсуляції в ООП. Фактично інтерфейс є описом певної програмної структури (плану). І якщо, наприклад, клас відповідає цій структурі, то до його екземплярів можна застосувати певні властивості або методи. Про такий клас кажуть, що для нього **визначено** вказаний **інтерфейс**. Можна сказати, що з погляду реалізації, інтерфейс — це клас «без коду», який визначає, якою буде поведінка у його екземплярів, але без реалізації особливостей цієї поведінки. Але про це трохи згодом.

Навіщо нам потрібно розробляти інтерфейси? Створюючи інтерфейси (які називаються також протоколами користувача), ми забезпечуємо можливість перевіряти сумісність типів (екземплярів класів) на основі структури цих типів.

Даний підхід являє собою певну варіацію качиної типізації — «*compile time duck typing*», тобто аналіз сумісності типів відбувається до виконання програми. Таким чином, цей механізм дає нам можливість виявити помилки на ранніх етапах розробки (до тестування).

Для створення таких користувацьких інтерфейсів (класів інтерфейсів) нам потрібно створити похідний клас від класу `Protocol`. Клас `Protocol` був доданий до Python 3.8 і визначений у модулі `typing`.

Розглянемо необхідність використання протоколів з прикладу. Припустимо, що в нас є клас `Book` (книга), в якому визначено властивості: ціна та знижка:

```
class Book:
    def __init__(self, title, author, price, discount):
        self.title = title
        self.author = author
        self.price=price
        self.discount=discount
```

Також ми розробили функцію `calculateTotalPrice()`, яка приймає список об'єктів `Book` (наприклад, набір книг у «Кошику» покупця) та повертає загальну вартість покупки з урахуванням знижок.

```
def calculateTotalPrice(objs):
    sum=0
    for obj in objs:
        sum+=obj.price*(1-obj.discount/100)
    return sum

book1=Book("Python Crash Course","Eric Matthes", 150, 25)
```



```
book2=Book("JavaScript: The Good Parts",
           "Douglas Crockford", 178, 30)

print (calculateTotalPrice([book1,book2])) #237.1
```

Цілком ймовірно, що можливість, яка надається функцією `calculateTotalPrice()`, може бути корисною і для інших категорій товарів, реалізованих за допомогою інших класів.

Наприклад, у нас є клас `Product` (*Товар*), об'єкти якого також можуть «наповнювати» «Кошик» користувача, і нам може знадобитися виклик функції `calculateTotalPrice()` для підрахунку загальної вартості покупки. Однак, якщо в класі `Product` немає хоча б однієї з властивостей, що використовуються функцією `calculateTotalPrice()`, `price` або `discount`, то дану помилку несумісності ми, як і в прикладі з `len()`, зможемо виявити лише на етапі тестування.

Тому ми створимо клас-протокол `Item`, похідний від класу `Protocol`, який описуватиме потрібний функції `calculateTotalPrice()` інтерфейс: наявність у класі властивостей `price` та `discount`. Таким чином, наш власний клас-протокол `Item` міститиме два атрибути: `price` та `discount`.

```
from typing import Protocol

class Item(Protocol):
    price:float
    discount:int
```

Атрибути та методи, зазначені у класі-протоколі `Item` мають бути присутніми у всіх класах (`Book` та `Product` у прикладі), які відповідають даному протоколу.

Зверніть увагу, що після імені властивості у класі-протоколі слід вказати символ двокрапки (:) та тип даних значень цієї властивості. Наш (оголошений вище) клас **Book** відповідає протоколу **Item**.

Також нам потрібно внести зміни до нашої функції **calculateTotalPrice()**: вказати, що її параметр має бути списком об'єктів типу **Item**:

```
def calculateTotalPrice(objs:List[Item]):
    sum=0
    for obj in objs:
        sum+=obj.price*(1-obj.discount/100)
    return sum
```

Виклик функції для покупки, яка являє собою список книг, поверне коректний результат:

```
purchase1 = calculateTotalPrice([
    Book("JavaScript: The Good Parts",
        "Douglas Crockford", 178, 30),
    Book("Python Crash Course","Eric Matthes", 150, 25)
])

print(purchase1) #237.1
```

Тепер давайте створимо клас **Product**, який не відповідатиме протоколу **Item** (властивість, що містить інформацію про розмір знижки, називається «**disc**»).

```
class Product:
    def __init__(self, name, price, disc, producer):
        self.name = name
        self.price = price
```

```
self.disc=disc
self.producer=producer
```

Однак, якщо в нашій новій покупці ми замість однієї з книг помістимо продукт (екземпляр класу `Product`) і передамо цей список товарів до функції `calculateTotalPrice()`:

```
purchase2 = calculateTotalPrice([
    Product('Tea', 100, 15, 'Lipton'),
    Book("Python Crash Course", "Eric Matthes", 150, 25)
])

print(purchase2)
```

то отримаємо помилку:

```
'Product' object has no attribute 'discount'
```

Рисунок 98

Це пов'язано з тим, що до складу покупки `purchase2` увійшов екземпляр класу `Product` ("Tea"), який не підтримує протокол `Item` (відсутня властивість «`discount`»).

Причина виникнення помилки тільки на етапі виконання програми полягає в тому, що протоколи, в основному, застосовуються для моделювання статичної типізації в `Duck Typing`, а не використовуються явно під час виконання програми.

Тобто за допомогою протоколів ми створюємо опис (план), за яким розробники класів (для яких передбачається відповідність протоколу) повинні слідувати. У нашому прикладі, розробнику класу `Product`, знаючи, що клас повинен слідувати протоколу `Item`, потрібно

використовувати ім'я атрибута «**discount**», а не «**disc**», для зберігання розміру знижки.

Проте хотілося б мати можливість автоматизувати перевірку відповідності об'єкта протоколу перед його використанням функцією.

Нам відома функція `isinstance()`, з допомогою якої ми могли б перевірити, чи відповідає об'єкт `obj` протоколу `Item`. Однак ця функція за замовчуванням не працює з протоколами.

Для того, щоб забезпечити цю можливість, нам потрібно перед оголошенням класу-протоколу вказати декоратор `@runtime_checkable` (який попередньо треба імпортувати з бібліотеки `typing`):

```
from typing import Protocol, List, runtime_checkable

@runtime_checkable
class Item(Protocol):
    price:float
    discount:int
```

Змінимо логіку функції `calculateTotalPrice()`, щоб вона включала перевірку відповідності об'єкта `obj` протоколу `Item` з допомогою функції `isinstance()`:

```
def calculateTotalPrice(objs:List[Item]):
    sum=0
    for obj in objs:
        if isinstance(obj, Item):
            sum+=obj.price*(1-obj.discount/100)
        else:
            print("Incompatible type")
    return sum
```

```
purchase3 = calculateTotalPrice([
    Book("JavaScript: The Good Parts",
        "Douglas Crockford", 178, 30),
    Product('Tea', 100, 15, 'Lipton'),
    Book("Python Crash Course", "Eric Matthes", 150, 25)
])

print(purchase3) #237.1
```

Як бачимо, обидва товари-книги були враховані при підрахунку загальної вартості покупки, а товар-продукт — ні, тому що він не відповідає протоколу [Item](#).



Урок 9

Вступ до ООП

© STEP IT Academy, www.itstep.org

© Анна Егошина

Усі права на фото-, аудіо- і відеотвори, що охороняються авторським правом і фрагменти яких використані в матеріалі, належать їх законним власникам. Фрагменти творів використовуються в ілюстративних цілях в обсязі, виправданому поставленим завданням, у рамках учбового процесу і в учбових цілях, відповідно до законодавства про вільне використання твору без згоди його автора (або іншої особи, яка має авторське право на цей твір). Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає збитку нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора і правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними аналогами, що не охороняються авторським правом, і відповідають критеріям добросовісного використання і чесного використання.

Усі права захищені. Повне або часткове копіювання матеріалів заборонене. Узгодження використання творів або їх фрагментів здійснюється з авторами і правовласниками. Погоджене використання матеріалів можливе тільки якщо вказано джерело.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством.