Київський національний університет імені Тараса Шевченка

КРЕНЕВИЧ А.П.

PYTHON
У ПРИКЛАДАХ І ЗАДАЧАХ
Частина 2. Об’єктно-орієнтоване програмування

Навчальний посібник

КИЇВ 2020

УДК 004.438С(075.8); 519.682

Рецензенти:
доктор фіз.-мат. наук _____ (КНУ імені Тараса Шевченка)
кандидат техн. наук _____ (НТУУ «КПІ»)

Креневич А.П.

Python у прикладах і задачах. Частина 2. Об’єктно-орієнтоване програмування. Навчальний посібник – К.: ВПЦ "Київський Університет", 2020. – 152 с.

Посібник призначений для практичного опанування парадигми об’єктно-орієнтованого програмування із застосуванням мови Python. У ньому у систематизованому вигляді наводяться короткі теоретичні відомості, типові підходи проектування об’єктно-орієнтованих проектів та приклади розв'язання задач.

Посібник складено з урахуванням досвіду викладання програмування на механіко-математичному факультеті Київського національного університету імені Тараса Шевченка.

Для студентів молодших курсів університетів та викладачів, що проводять практичні заняття з програмування.

ЗМІСТ

Вступ

Цей посібник призначений для опанування студентами парадигми об’єктно-орієнтованого програмування. Теоретичний матеріал посібника пояснюється на великій кількості типових прикладів, більшість з яких супроводжуються детальними описами алгоритмів, а ключові моменти програм пояснені у коментарях.

Посібник складається з 6 параграфів, кожен з яких присвячений конкретній темі об’єктно-орієнтованого програмування. Більшість параграфів поділені на пункти, нумерація яких включає номер параграфу якому належить пункт. Нумерація всіх об'єктів у посібнику (означень, прикладів, рисунків тощо) складається з двох частин: номеру параграфу та порядкового номеру об'єкта у ньому. Для відображення фрагментів коду програм, інструкцій та ідентифікаторів, що використовуються при написанні програм, застосовується моноширинний шрифт з підсвічуванням (різними кольорами), аналогічним до того, яке використовується у сучасних інтегрованих середовищах розробки. Ці заходи мають значно полегшити процес сприйняття матеріалу.

Усі приклади наведені у посібнику опубліковані на GitHub сторінці автора, звідки при бажанні, читач може завантажити їх скориставшись посиланням https://github.com/krenevych/oop. Репозиторій за цим посиланням є проектом, орієнтованим на застосування інтегрованого середовища програмування PyCharm Community, що вільно розповсюджується. Структура репозиторія повністю відповідає структурі посібника. Всі програми розташовані у папці `source`, у якій міститься перелік папок, кожна з яких відповідає певному параграфу посібника. Так, наприклад, у папці `P_02` містяться програми, що відповідають § 2. Далі, кожен файл починається з префіксу, що містить номер лістингу. Отже, файл `L1_Pet.py`, що міститься у папці `source/P_02` відповідає лістингу 2.1 наведеному у підручнику. Сподіваємось така організація прикладів дозволить читачу без особливих труднощів знаходити програми наведені у посібнику.

Автор висловлює щиру подяку колегам кафедри математичної фізики за корисні поради, конструктивну критику та допомогу при створенні цього посібника.

Ваші відгуки і побажання щодо змісту підручника
надсилайте на електронну адресу автора:
krenevych@knu.ua

§1 ВСТУП ДО ООП. ОБ'ЄКТИ ТА КЛАСИ

Процедурний стиль програмування був домінуючим підходом під час розробки програмного забезпечення протягом десятиліть. Він все ще використовується сьогодні, оскільки добре працює для окремих типів проектів (наприклад, для задач обчислювальної математики). Проте цього підходу вже не вистачає для складних проектів, результати роботи яких оточують нас у сучасному комп'ютерному світі.

Дійсно, при процедурному підході можна виокремити два концептуально різні світи: світ даних і світ коду. Світ даних заповнений різними типами змінних, в той час як другий заселений кодом – функціями, що об'єднуються у модулі. Різні функції можуть використовувати дані. Реалізовуючи складний програмний проект потрібно постійно пам'ятати які дані якими функціями можуть опрацьовуватися, а відповідальність за правильний зв'язок дані-функція повністю покладається на програміста. Очевидно, що з ростом кількості коду у проекті збільшується і ймовірність помилок у ньому.

Іншим серйозним недоліком процедурного підходу є незахищеність даних. Дійсно, функції можуть застосовуватися до будь-яких даних доступних у програмі. Наприклад, якщо програма має доступ до банківських рахунків, то всі функції у програмі також матимуть доступ до цих даних. Уявіть який хаос почнеться, якщо будь-хто може безконтрольно перерахувати кошти будь-куди.

Тому з часом з'явився концептуально інший підхід – об'єктно-орієнтований – що подолав вищенаведені проблеми та спростив програмування, піднявши його на новий якісний рівень.

Об'єктно-орієнтований підхід передбачає зовсім інший, у порівнянні з процедурним програмуванням, спосіб мислення. Якщо при процедурному програмуванні дані і код розміщуються поруч, то при об'єктно-орієнтованому програмуванні пов'язані дані і код об'єднуються разом у єдину структуру – об'єкт. На відміну від функцій та модулів, об'єктно-орієнтоване програмування дозволяє не лише розділити програму на фрагменти, але і описати предмети реального світу у вигляді об'єктів, а також організувати зв'язки між цими об'єктами. Крім цього, правильно побудований об'єкт здатен захистити свої дані від несанкціонованого використання чи модифікації, адже при об'єктно-орієнтованому підході даними володіє об'єкт і він сам вирішує які дані та у якій ситуації він може віддавати іншим об'єктам.

Основна мета об'єктно-орієнтованого програмування – забезпечити повторне виконання існуючого програмного коду та спростити програмування піднявши його на новий рівень абстракції.

Означення 1.1. **Об'єктно-орієнтоване програмування** (у літературі часто використовується абревіатура ООП) – це парадигма[1] програмування, яка розглядає програму як множину "об'єктів", що взаємодіють між собою.

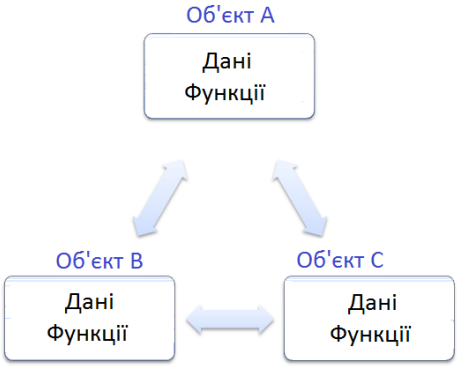


Рисунок 1.1. У ООП програма це множина "об'єктів", що взаємодіють між собою

1.1. Поняття про класи та об’єкти.

В основі ООП знаходяться такі поняття як: *клас* та *об’єкт* (або екземпляр класу). Перш ніж перейти до строгих означень, розглянемо такий приклад. Припустимо, ми проектуємо автомобіль. Ми знаємо, що автомобіль має складатися з двигуна, коліс, кузова, пристроїв освітлення тощо. Також ми знаємо, що автомобіль має їхати, гальмувати, змінювати напрямок руху, вмикати головне світло тощо. Крім цього ми вивчаємо як мають взаємодіяти всі агрегати автомобіля між собою. Вивчивши та проаналізувавши все вищенаведене, ми, описуємо всі необхідні запчастини з яких складається автомобіль, а також як ці запчастини мають взаємодіями між собою. Крім цього ми описуємо, яким чином кінцевий користувач взаємодіє з автомобілем для того, щоб ним керувати. У результаті виконаної роботи буде створений шаблон який у подальшому стане у нагоді для виробництва автомобілів. Цей шаблон у термінах об’єктно-орієнтованого програмування називається *класом*. Після того, як автомобіль спроектовано, на заводі, за цим шаблоном починається випуск автомобілів. Виготовлені таким чином автомобілі, на відміну від шаблона, який є певною абстракцією, є конкретними сутностями. У

[1] Парадигма програмування – це система ідей і понять, які визначають стиль написання комп'ютерних програм, а також спосіб мислення програміста.

об'єктно-орієнтованому програмуванні такі сутності називаються об'єктами або екземплярами класу. Якщо клас-шаблон формально описує складові автомобіля та інші його властивості, то кожен з виготовлених за цим шаблоном автомобілів, має конкретне значення цих властивостей. Наприклад, кожен автомобіль має свій унікальний номер кузова та двигуна, колір кузова, оббивку сидінь, тип кліматичної системи, поточні показники приладів тощо. Також, якщо клас описує шаблони поведінки, то об'єкт реалізує цю поведінку для себе (взагалі кажучи, не зачіпаючи інші екземпляри цього класу). Наприклад, для екземпляру класу автомобіль можна застосувати шаблон поведінки, «їхати», щоб привести його у рух. І цей шаблон поведінки буде застосований лише для конкретного автомобіля – інші об'єкти цього класу аж ніяк не зобов'язані почати рух.

Означення 1.2. Клас – це абстракція, що описує спільну поведінку та властивості подібних сутностей.

Означення 1.3. Об'єкт – це певна сутність, що має визначені властивості, стан та поведінку.

Фактично клас – це креслення за яким будують конкретний об'єкт. Створені об'єкти на основі цього класу-креслення є екземплярами класу. Клас описує властивості, що притаманні всім його об'єктам та визначає шаблони їхньої поведінки. Клас не має визначеного стану – стан визначено для екземплярів класу.

Щоб усвідомити різницю між класом та об'єктом можна навести такі аналогії: `Університет` є класом (до якого входять всі конкретні університети), а Київський національний університет імені Тараса Шевченка є об'єктом (екземпляром класу `Університет`). `Собака` є класом, а пес Тобі інспектора Лестрейда з циклу романів Артура Конан Дойля – об'єктом (екземпляром класу `Собака`). `Автомобіль` є класом, а Batmobile вигаданого героя коміксів Бетмена – об'єктом.

Атрибути та методи

Як вище було зазначено, головна ідея об'єктно-орієнтованого підходу полягає у тому, щоб об'єднати у одне ціле **дані**, тобто інформацію, що стосується об'єкту, і **функції**, що призначені для обробки цих даних. Всі дані об'єкта зберігаються у його атрибутах.

Означення 1.4. Атрибут (**поле, властивість**) – змінна пов'язана з класом або об'єктом.

Для прикладу, клас/об'єкт автомобіль має властивості:

- марка
- модель
- колір
- потужність двигуна
- максимальна швидкість

Стан об’єкта визначається поточними значеннями кожного з атрибутів. Отже, стан об’єкта автомобіль, з атрибутами зазначеними вище, визначається конкретними значеннями цих атрибутів:

- марка = Toyota
- модель = Corolla
- колір = білий
- потужність двигуна = 150 к.с.
- максимальна швидкість = 190 км/год

Об’єкти не існують ізольовано один від іншого. Вони взаємодіють з іншими об’єктами, обмінюючись даними. Саме ця взаємодія і характеризує поведінку об’єкта. Тобто поведінка об’єкта це його діяльність з точки зору оточуючого середовища. Формально поведінка об’єкту визначається у його методами

Означення 1.5. Метод – підпрограма, що використовується виключно разом із класом (метод класу) або екземпляром класу (метод екземпляру).

Технічно, призначення методів полягає в здійсненні певних дій над полями об’єкту або надання механізму доступу до полів, що містяться в об'єкті або класі. Таким чином методи визначають поведінку об’єкта або змінюють його стан.

Отже, поведінка для об’єкта автомобіль може бути такою:

- їхати
- гальмувати
- увімкнути світло
- поповнити пальне у бензобаку

Сукупно атрибути та методи називають **членами класу**.

Як і у звичайному процедурному програмування, доступ до полів та методів об’єкту здійснюється через їхні імена. Проте у ООП ці імена організовані спеціальним чином, щоб визначити приналежність поля або методу до конкретного об’єкта.

Абстракція та абстрагування

Створенню класу завжди передує процес абстрагування.

Означення 1.6. Абстрагування – спосіб виділити набір вагомих характеристик об'єкта та шаблонів поведінки, які чітко визначають його концептуальні межі, що відрізняють його від інших об'єктів.

Означення 1.7. Абстракція – набір характеристик об'єкта та шаблонів поведінки, що виникає в результаті абстрагування.

Для прикладу розглянемо процес абстрагування під час створення класу Трикутник, у контексті його геометричних властивостей. Очевидно, що у атрибутах цього класу повинна міститися інформацію про величини його сторін. Крім цього, необхідно описати перелік шаблонів поведінки, таких як відшукання площі та периметру цього трикутника, перевірки чи є трикутник прямокутним, тощо.

У іншому контексті, процес абстрагування породив би інші властивості та шаблони поведінки. Наприклад, якби нас цікавила візуалізація трикутника, то були б виділені властивості, що визначають його положення на екрані, розмір, колір, тощо.

Створення класу

Клас описується за допомогою ключового слова `class`. Синтаксис створення класу такий

```
class NewClass:
    class_body
```

де `NewClass` – ім'я створюваного класу, `class_body` – тіло класу – методи, класові поля тощо.

Зауважимо, що при описі тіла класу необхідно робити відступи, що вказують, що тіло класу є вкладеною інструкцією.

Як правило тіло класу у Python складається з опису методів.

```
class NewClass:
    def method1(self, arg1, ..., argN1):
        method1_body
    def method2(self, arg1, ..., argN2):
        method2_body
    ...
    def method_M(self, arg1, ..., argNM):
        method_M_body
```

Як ми вже знаємо, метод класу – це функція, що належить класу і викликатися вона буде не безпосередньо, а через екземпляр класу.

Відмінність методу від звичайної функції у Python полягає у тому, що під час опису, метод завжди першим параметром має обов'язковий параметр, що

є посиланням на екземпляр класу, з якого викликається метод. За домовленістю цей аргумент має ім'я **self**.

Методи можуть містити описи (оголошення) полів. Створення поля всередині класу відбувається аналогічно до створення звичайної змінної – потрібно вказати її ім'я та ініціалізувати її (наприклад, певним літералом). Проте, щоб відрізняти поля об'єкту від локальних змінних методу, усі поля починаються зі спеціального префіксу – посилання на цей же екземпляр класу (тобто через параметр **self**) після якого йде крапка:

```
self.a = e  # створення поля a зі значенням e
```

Конструктор та деструктор

Будь-який клас має два спеціальних методи (навіть якщо вони явно не визначені) – конструктор та деструктор.

Означення 1.8. Конструктор (від англ. constructor) – спеціальний метод класу, який автоматично викликається при створенні об'єкта.

В Python конструктором є метод класу з іменем **__init__**. Як і для інших методів класу, список формальних параметрів конструктора починається з формального аргументу **self**.

```
class NewClass:
    ...
    def __init__(self, par1, ..., parN):
        """ This method is constructor """
        ...
```

Конструктор неявно викликається під час створення екземпляра класу та призначений для ініціалізації його атрибутів. У випадку, якщо конструктор не був описаний у класі, інтерпретатор неявно його створює.

Означення 1.9. Деструктор (від англ. destructor) – спеціальний метод класу, який викликається автоматично при знищенні об'єкта і призначений для його де-ініціалізації.

Основне призначення деструктора полягає у звільнені ресурсів, наприклад, для звільнення пам'яті, що займали об'єкти, закритті файлів тощо. Деструктор об'єкта завжди неявно викликається під час його знищення. Він є прямою протилежністю до конструктора – життя будь-якого об'єкту починається з виклику конструктора та закінчується викликом деструктора.

В Python деструктором є метод класу з іменем **__del__**.

```
class NewClass:
    ...
    def __del__(self):
        """This method is destructor"""
        ...
```

Зауважимо, що список формальних параметрів деструктора завжди складається з єдиного формального аргументу **self**.

У більшості випадків деструктор не є обов'язковим для опису методом у Python.

Звернемо увагу читача, на той факт, що поля класу можна створювати у будь-якому методі класу. Проте, здебільшого будемо користуватися загальноприйнятою домовленістю, згідно з якою поля описуються (тобто створюються) у конструкторі класу.

Звернемо увагу, на те, що імена методів **__init__** та **__del__** (конструктора та деструктора відповідно) починаються та закінчуються подвійним символом нижнього підкреслення. Це вказує на те, що ці методи мають спеціальне призначення і не призначені для використання у ролі рядових методів класу.

У Python існує ціла низка спеціальних методів, які у своїх іменах мають два нижніх підкреслення на початку та в кінці імені, з якими ми познайомимося пізніше.

Створення та знищення об'єктів

Після створення класу, можна створити екземпляр (об'єкт) цього класу. Здійснюється це інструкцією наведеною нижче

```
obj = NewClass(arg1, ..., argN)
```

де `obj` – змінна, що буде містити посилання на створений екземпляр класу `NewClass`, аргументи `arg1, ..., argN` – фактичні аргументи, що відповідають списку формальних параметрів метода-конструктора (тобто метода **__init__**) за виключенням аргумента **self**.

Під час створення об'єкта, неявно викликається метод-конструктор:

```
obj.__init__(arg1, ..., argN)
```

Саме це і пояснює вищенаведену вимогу до списку фактичних аргументів `args`.

Деструктор неявно викликається під час знищення об'єкту, наприклад, під час завершення роботи програми або під час явного знищення об'єкту оператором **del**.

```
del obj
```

Проте робота метода-деструктора має певні особливості, Це пов'язано з тим, що інтерпретатор Python самостійно керує знищенням об'єктів – якщо настає такий момент, що на екземпляр класу не існує жодного посилання (жодна змінна у програмі не вказує на цей об'єкт), автоматично викликається деструктор для цього об'єкта. Крім цього, варто звернути увагу на той факт, що якщо у програмі на екземпляр класу існує хоча б одне посилання, то деструктор не буде викликаний, навіть якщо для нього явно був викликаний оператор **del**.

Тому (на відміну від багатьох інших мов програмування), у більшості випадків, контролювати цей процес зі сторони програміста не вимагається та й не має сенсу.

Приклад 1.1. Для прикладу Створимо клас Notifier, екземпляр якого буде при створенні та знищенні повідомляти про своє створення та знищення відповідно.

Для виконання поставленої задачі досить описати клас, що буде містити конструктор та деструктор і створити екземпляр цього класу.

Лістинг 1.1. Клас Notifier. Створення та знищення об'єктів

```
class Notifier:
    def __init__(self):
        print("Notifier: Екземпляр створено")

    def __del__(self):
        print("Notifier: Екземпляр знищено")

notifier1 = Notifier()   # список аргументів порожній
```

У результаті роботи вищезначеного коду у консоль буде виведене повідомлення:

```
Notifier: Екземпляр створено
Notifier: Екземпляр знищено
```

Тепер спробуємо скористатися оператором **del** для знищення об'єктів класу Notifier.

Лістинг 1.2. Особливості створення та знищення об’єктів

```
class Notifier:
    ...


notifier1 = Notifier() # Виклик конструктора - створення
                       # екземляру класу Notifier
notifier2 = notifier1  # Створення посилання на екземпляр класу

del notifier1 # Виклик деструктора не буде здійснено, бо є
              # ще одне посилання (notifier2) на цей об’єкт
del notifier2 # Здійснюється виклик деструктора
```

Результат роботи програми буде таким:

```
Notifier: Екземпляр створено
Notifier: Екземпляр знищено
```

Виклик методів та звернення до полів об’єкту

Ми вже вміємо викликати спеціальні методи класу – конструктор та деструктор. Поговоримо детальніше, яким чином можна викликати звичайні методи та звертатися до полів об’єкту. Доступ до атрибутів та методів класу здійснюється через посилання на екземпляр класу використовуючи «оператор крапку»:

```
obj.atribute
```

де `obj` – екземпляр класу, що має поле `atribute`.

```
obj.method(args)
```

де `obj` – екземпляр класу, що має метод `method`, `args` – список фактичних параметрів, що відповідають сигнатурі метода `method`. Причому, тут не потрібно вказувати першим параметром посилання на екземпляр класу – інтерпретатор здійснить це самостійно автоматично.

Приклад 1.2. Змінимо раніше описаний клас `Notifier`, додавши поле, що буде містити ім’я екземпляра класу, яке буде задаватися при створенні екземпляра у конструкторі.

Лістинг 1.3. Клас `Notifier`

```
class Notifier:
    def __init__(self, name):
        self.name = name  # створюється поле self.name
        print("Notifier: Екземпляр %s створено" % self.name)

    def __del__(self):
        print("Notifier: Екземпляр %s знищено" % self.name)

notifier1 = Notifier("Notifier1")
```

Звернемо увагу, що під час створення нового об'єкту `notifier`, ми вказали у конструкторі (виклик якого здійснюється неявно) фактичний параметр `"Notifier1"`, який передбачений його сигнатурою і буде співставленим з формальний аргументом `name`.

Отже, у результаті роботи вищезначеного коду у консоль буде виведено повідомлення

```
Notifier: Екземпляр Notifier1 створено
Notifier: Екземпляр Notifier1 знищено
```

Приклад 1.3. Створимо клас `Triangle`, що буде містити дані про трикутник. Опишемо методи обчислення периметра та площі цього трикутника.

Клас `Triangle`, буде містити три поля – довжини сторін трикутника – a, b, c. Опишемо метод **`perimeter`**, що визначає периметр трикутника та метод **`square`**, що обчислює площу зазначеного трикутника використовуючи формулу Герона.

Лістинг 1.4. Клас `Triangle`

```
class Triangle:
    def __init__(self, a, b, c):
        """ Конструктор трикутника
        :param a: перша сторона трикутника
        :param b: друга сторона трикутника
        :param c: третя сторона трикутника
        """
        # перевіримо чи можна створити такий трикутник
        assert a + b > c and a + c > b and c + b > a
        self.a = a  # поле a - перша сторона трикутника
        self.b = b  # поле b - друга сторона трикутника
        self.c = c  # поле c - третя сторона трикутника
```

```
    def perimeter(self):
        """ Обчислює периметр трикутника
        :return: периметр трикутника
        """
        # периметр це сума сторін трикутника
        return self.a + self.b + self.c

    def square(self):
        """ Обчислює площу трикутника за формулою Герона
        :return: площу трикутника
        """
        p = self.perimeter() / 2.0  # обчислимо півпериметр
        res = p * (p - self.a) * (p - self.b) * (p - self.c)
        return res ** 0.5


t = Triangle(3, 4, 5)
print("Площа заданого трикутника = %f" % t.square())
```

Ще раз звернемо увагу читача на той факт, що всередині класу (тобто під час його опису) доступ до атрибутів та методів здійснюється через посилання `self`:

```
        p = self.perimeter() / 2.0  # обчислимо півпериметр
```

у той час як поза межами класу (тобто для конкретного створеного екземпляру класу), доступ до членів класу здійснюється через ім'я створеного об'єкту:

```
t.square() # виклик методу square() об'єкту t
```

Копіювання об'єктів

Однією з класичних задач ООП, є така задача: побудувати на базі екземпляру класу точнісінько такий же екземпляр (тобто потрібно створити копію екземпляру класа).

Проблема у тому, що звичайне присвоєння для такої задачі є безсиле, оскільки присвоєння, як нам відомо – це фактично присвоєння посилань. Наприклад, якщо ми маємо клас `Triangle`, описаний у лістингу 1.4 і його екземпляр `t`, то присвоєння

```
t1 = t
```

створить посилання `t1` на раніше створений екземпляр класу, тобто посилання на той же екземпляр, що і `t`. У цьому легко переконатися, якщо змінити одне з полів створеного трикутника `t`

```
t.a = 5
```

і вивести це ж поле трикутника t1

```
print(t1.a)
```

Результатом буде

```
5
```

Проблему копіювання екземплярів класів, тобто створення нового екземпляру класу на базі вихідного, з точнісінько такими ж характеристиками, що і вихідний, вирішують функції **copy()** та **deepcopy()** з модуля copy. Їхнє використання демонструють приклади наведені у листингах 1.5 та 1.6 нижче

Лістинг 1.5. Копіювання об'єктів

```
from copy import copy

t = Triangle(3, 4, 5)

t1 = copy(t) # Створення копії об'єкта t

print("t:  ", t.a, t.b, t.c)
print("t1: ", t1.a, t1.b, t1.c)
print()

t1.a = 5
print("t:  ", t.a, t.b, t.c)
print("t1: ", t1.a, t1.b, t1.c)
```

Результатом роботи вищенаведеного коду буде виведення у консоль такого повідомлення:

```
t:   3 4 5
t1:  3 4 5

t:   3 4 5
t1:  5 4 5
```

При використанні функції **copy()** створюється поверхнева копія екземпляру, тоді як функція **deepcopy()** дозволяє створювати повні копії. Відмінність між поверхневим копіюванням і повним проявляється тоді, коли серед полів екземпляру є такі, що посилаються на дані змінюваних типів

(наприклад, списки, словники, екземпляри класів, тощо): функція **copy()** зробить у новому екземплярі посилання на дані цього поля, у той час як **deepcopy()** створить нове поле та скопіює всі дані.

Для прикладу розглянемо клас MyList, що описується таким чином

Лістинг 1.6. Різниця між **copy()** та **deepcopy()**.

```
from copy import copy, deepcopy

class MyList:
    def __init__(self):
        self.elements = []  # елементи колекції

    def print(self):
        """ Виведення елементів колекції """
        print(self.elements)
```

Створимо екземпляр класу MyList та додамо у нього кілька елементів та створимо копії функціями **copy()** та **deepcopy()**.

Лістинг 1.6. Продовження. Різниця між **copy()** та **deepcopy()**.

```
l = MyList()
l.elements.append(3)
l.elements.append(4)

l_copy = copy(l)
l_deepcopy = deepcopy(l)

print("Щойно після копіювання:")
l.print()
l_copy.print()
l_deepcopy.print()
```

Результатом роботи цього фрагменту коду буде:

```
Щойно після копіювання:
[3, 4]
[3, 4]
[3, 4]
```

Як бачимо всі три об'єкти ідентичні. Тепер змінимо перший елемент у об'єкту l_copy.

Лістинг 1.6. Продовження. Різниця між **copy()** та **deepcopy()**.

```
l_copy.elements[1] = 444
print("Після інструкції l_copy.elements[1] = 444 : ")
l.print()
l_copy.print()
l_deepcopy.print()
```

Результатом роботи цього фрагменту коду буде виведення на екран такого тексту

```
Після інструкції l_copy.elements[1] = 444 :
[3, 444]
[3, 444]
[3, 4]
```

Такий результат можна пояснити тим, що об'єкт `l_copy` був створений у наслідок копіювання об'єкта `l` функцією **copy()**, яка лише скопіювала посилання для поля `elements` з об'єкта `l`. Об'єкт `l_deepcopy` був створений у наслідок копіювання об'єкта `l` функцією **deepcopy()**, яка, на відміну від попередньої, створила для поля `elements` новий список, як копію списку `elements` об'єкта `l`.

Конструктор копіювання

Описаний вище підхід не завжди є прийнятним через цілий ряд причин (наприклад, швидкодія, необхідність підключати додаткові модулі). Тому в ООП замість нього широко застосовують механізм створення іншого екземпляру класу на базі вихідного за допомогою **конструктора копіювання**.

Означення 1.10. Конструктор копіювання – конструктор, список формальних параметрів якого складається з єдиного аргументу – посилання на інший екземпляр цього ж класу.

Класичний опис конструктора копіювання передбачає перевантаження конструктора – додатково до стандартного конструктора описується конструктор, аргументом якого є інший екземпляр цього класу.

Означення 1.11. Перевантаження функції (методу) – один із засобів реалізації поліморфізму, що полягає у можливості створення кількох реалізацій функції (методу) із одним і тим же іменем проте з різною сигнатурою (з різною кількістю параметрів або з різним типом параметрів)

На жаль Python не підтримує механізм перевантаження функцій і методів (не плутати з перевизначенням у нащадках класу). Відповідно, це призводить до неможливості створення конструктора копіювання у Python у класичному розумінні цього поняття. Натомість у Python, замість конструктора копіювання використовують конструктор з можливістю копіювання. Цей механізм реалізується за допомого типових параметрів та функції визначення приналежності об'єкту до класу. Функція

```
isinstance(x, cls)
```

повертає **True** , якщо екземпляр x належить до класу `cls`.

Приклад 1.4. Змінимо клас `Triangle`, описаний вище, визначивши у ньому конструктор з підтримкою копіювання.

Оскільки решта методів цього класу, залишиться без змін, то їх у цьому лістингу наводити не будемо.

Лістинг 1.7. Конструктор з підримкою копіювання.

```
class Triangle:
    def __init__(self, a, b=0, c=0):
        """ Конструктор трикутника з можливістю копіювання
        :param a: перша сторона або екземпляр класу Triangle
        :param b: друга сторона
        :param c: третя сторона
        """
        if isinstance(a, Triangle):  # a є екземпляром Triangle
            self.a = a.a  # поле a – копія поля a трикутника a
            self.b = a.b  # поле b - копія поля b трикутника a
            self.c = a.c  # поле c - копія поля c трикутника a
        else:
            # перевіримо чи можна створити такий трикутник
            assert a + b > c and a + c > b and c + b > a
            self.a = a  # поле a - перша сторона трикутника
            self.b = b  # поле b - друга сторона трикутника
            self.c = c  # поле c - третя сторона трикутника

    def perimeter(self):
        ...

    def square(self):
        ...


t1 = Triangle(3, 4, 5) # створення трикутника за сторонами
t2 = Triangle(t1)      # створення копії трикутника t1
```

```
t2.a = 5
print("Площа трикутника t1 = %f" % t1.square())
print("Площа трикутника t2 = %f" % t2.square())
```

Результатом вищенаведеного коду буде

```
Площа заданого трикутника = 6.000000
Площа заданого трикутника = 9.165151
```

1.2. Інкапсуляція

Інкапсуляція слугує надійним механізмом захисту даних об’єктів та класів від несанкціонованого доступу. Її використання зменшує ймовірність помилок через приведення екземпляра класу до непередбаченого стану.

Означення 1.12. Інкапсуляція – механізм, що полягає у приховуванні від зовнішнього користувача (прикладного програміста, іншого об'єкта тощо) деталей реалізації об'єкта, натомість надаючи механізм взаємодії з об'єктом.

Іншими словами, інкапсуляція – це механізм об’єднання коду і даних всередині об’єкта, а також їхній захист від непередбачених змін ззовні. Вона слугує передусім для того, щоб не давати можливості змінювати внутрішній стан об'єкта без його відома.

Простими словами, інкапсуляція полягає у забороні або обмежені доступу до частини атрибутів та методів класу, що призначені лише для внутрішнього використання всередині класу.

Для взаємодії об'єкта з іншими об'єктами надається набір "офіційних" (загальнодоступних) методів.

Означення 1.13. Інтерфейс класу – це набір загальнодоступних полів та методів класу.

Користувач може взаємодіяти з об’єктом класу лише через його інтерфейс.

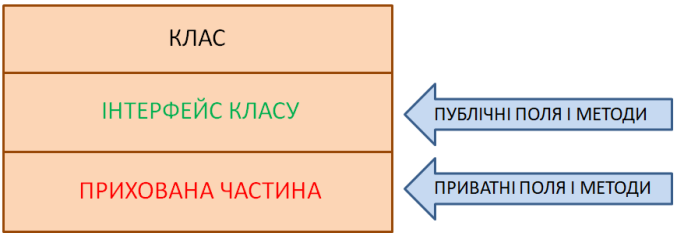


Рисунок 1.2. Інкапсуляція

Наприклад, для класу `Автомобіль`, поле, що містить інформацію про об'єм бензобаку повинно бути ініціалізованим під час створення екземпляру класу `Автомобіль` (тобто у конструкторі) і не може бути змінене ззовні, оскільки це відповідатиме втручанню у внутрішню конструкцію автомобіля. Аналогічно поле, що містить інформацію про кількість пального у бензобаку, може змінюватися лише під час заправки автомобіля на АЗС станції або в наслідок роботи двигуна автомобіля. Проте користувач у будь-який момент може переглянути рівень пального у бензобаку та поповнити кількість пального на АЗС – це є загальнодоступні методи.

Безпосередній доступ до двигуна автомобіля також є забороненим, натомість користувачу надаються інтерфейси (див. рисунок 1.3), за допомогою яких він може приводити автомобіль у рух (різні педалі, кнопки та важелі), переглядати його стан (різноманітні індикатори), тощо.

Рисунок 1.3. Інтерфейс автомобіля

У багатьох мовах програмування інкапсуляція реалізована на рівні синтаксису самої мови: всі атрибути і методи поділяються на приватні (внутрішні), публічні (загальнодоступні) захищені (protected).

Інкапсуляція у Python працює лише на рівні домовленості між програмістами які атрибути та методи є публічні, а які приватні: ідентифікатор атрибуту або методу, що має префікс нижнього підкреслювання (наприклад, `_spam`) вважається приватним.

Лістинг 1.8. Приватний та публічний методи.

```
class A:
    def _private(self):   # Приватний метод
        print("Хей, я приватний метод!")

    def public(self):     # Публічний метод
        print("Все ОК!")
```

```
a = A()       # Створення екземпляру класу
a._private()  # Виклик приватного методу
a.public()    # Виклик публічного методу
```

Результат роботи коду наведеного вище

```
Хей, я приватний метод!
Все OK!
```

Також для реалізації інкапсуляції (скоріше як механізму захисту від необережного використання) можна застосовувати інший механізм, що називається *корегуванням імен* (*name mangling*).

Будь-який ідентифікатор, що починається принаймні з двох і завершується не більше ніж одним знаком нижнього підкреслення (наприклад, `__spam`) заміняється новим ідентифікатором у якому у ролі префіксу додається ім'я класу з початковим символом нижнього підкреслення

Лістинг 1.9. Зовсім приватний метод.

```
class A:
    def __private(self):  # Зовсім приватний метод
        print("Хей, я ЗОВСІМ приватний метод!")

a = A()
a.__private()
```

У результаті виконання вищенаведеного коду буде породжена помилка, оскільки, як було сказано, інтерпретатор Python замінить ім'я методу `__private`, і, відповідно, об'єкт a не матиме такого методу.

```
    a.__private()
AttributeError: 'A' object has no attribute '__private'
```

Змінимо виклик цього методу через новий ідентифікатор, отриманий внаслідок застосування механізму коригування імен.

```
a._A__private()  # Виклик методу через модифіковане ім'я
```

Тоді наслідком виконання такого коду буде виклик методу `__private`

```
Хей, я ЗОВСІМ приватний метод!
```

Приклад 1.5. Вдосконалимо вищенаведений клас `Triangle` використовуючи інкапсуляцію.

Очевидно, що якщо до полів класу `Triangle`, що містять довжини сторін трикутника буде безпосередній доступ, то це може призвести до того, що користувач так змінить сторони трикутника, що той перестане існувати. Дійсно, якщо у трикутнику зі сторонами 3, 4, 5 замінити третю сторону на 8, то нерівність трикутника вже не виконуватиметься:

$$3 + 4 < 8$$

Отже, для коректної роботи програми нам необхідно захистити поля, що містять сторони трикутника, натомість описати інтерфесні методи, що будуть містити перевірку коректності.

Лістинг 1.10. Клас `Triangle` з підтримкою інкапсуляції.

```
class Triangle:
    def __init__(self, a, b=0, c=0):
        """ Конструктор трикутника з можливістю копіювання
        :param a: перша сторона трикутника або екземпляр класу
        :param b: друга сторона трикутника
        :param c: третя сторона трикутника
        """
        if isinstance(a, Triangle):  # гілка копіювання
            self._a = a._a # приватне поле _a –
                           # копія поля _a трикутника a
            self._b = a._b # приватне поле _b -
                           # копія поля _b трикутника a
            self._c = a._c # приватне поле _c -
                           # копія поля _c трикутника a
        else:
            # перевіримо чи можна створити такий трикутник
            assert self.isExist(a, b, c)
            self._a = a  # приватне поле _a
            self._b = b  # приватне поле _b
            self._c = c  # приватне поле _c

    def isExist(self, a, b, c):
        """ Перевіряє, чи можна створити
            тикутник з заданими сторонами
        :param a: перша сторона трикутника
        :param b: друга сторона трикутника
        :param c: третя сторона трикутника
        :return: True, якщо трикутник з сторонами a, b, c існує
        """
        return a + b > c and a + c > b and c + b > a
```

```
    def set_a(self, a):
        """ Встановити сторону a трикутника
        :param a: сторона трикутника
        """
        # перевіримо чи існуватиме такий трикутник
        assert self.isExist(a, self._b, self._c)
        self._a = a

    def set_b(self, b):
        """ Встановити сторону b трикутника
        :param b: сторона трикутника
        """
        # перевіримо чи існуватиме такий трикутник
        assert self.isExist(self._a, b, self._c)
        self._b = b

    def set_c(self, c):
        """ Встановити сторону c трикутника
        :param c: сторона трикутника
        """
        # перевіримо чи існуватиме такий трикутник
        assert self.isExist(self._a, self._b, c)
        self._c = c

    def get_a(self):
        """ Повертає сторону a трикутника
        :return: Значення сторони a
        """
        return self._a

    def get_b(self):
        """ Повертає сторону b трикутника
        :return: Значення сторони b
        """
        return self._b

    def get_c(self):
        """ Повертає сторону c трикутника
        :return: Значення сторони c
        """
        return self._c

    def perimeter(self):
        """ Обчислює периметр трикутника
        :return: периметр трикутника
        """
        # периметр це сума сторін трикутника
        return self._a + self._b + self._c
```

```
    def square(self):
        """ Обчислює площу трикутника за формулою Герона
        :return: площу трикутника
        """
        p = self.perimeter() / 2.0  # обчислимо півпериметр
        res = p * (p - self._a) * (p - self._b) * (p - self._c)
        return res ** 0.5

if __name__ == "__main__":
    t = Triangle(3, 4, 5)
    print("Площа заданого трикутника = %f" % t.square())

    t.set_c(2)
    print("Площа трикутника зі сторонами %f, %f, %f є %f"
          % (t.get_a(), t.get_b(), t.get_c(), t.square()))
```

Результат роботи вищенаведеного коду

```
Площа заданого трикутника = 6.0
Площа трикутника зі сторонами 3.0, 4.0, 2.0 є 2.904738
```

1.3. Статичні поля та методи

Статичні поля та методи класу – це спільні поля та методи для всіх екземплярів класу. Якщо статичне поле змінити для одного екземпляру класу, то воно зміниться для всіх інших також.

Наприклад, клас Автомобіль містить колеса. Кожне колесо, автомобіля належить конкретному автомобілю (екземпляру класу). Проте, як правило, для всіх автомобілів коліс 4. Тому, це поле (кількість коліс) можна вважати статичним.

Використання статичних полів та методів доступне без створення екземплярів класу – такі поля та методи належить не екземпляру класу, а самому класу. Дійсно, для того, щоб дізнатися скільки коліс має автомобіль, нам не потрібно його створювати – ми знаємо, що всі авто мають 4 колеса.

Статичні поля створюються (ініціалізують) всередині опису класу як звичайні змінні:

```
class Car:
    ...
    wheel_number = 4  # Статичне поле класу
    ...
```

Доступ до статичних полів як правило здійснюється через ім'я класу, до якого воно належить

```
print(Car.wheel_number)  # Звернення до статичного поля класу
```

Хоча допускається доступ і через екземпляр класу:

```
c = Car()
print(c.wheel_number)  # Звернення до статичного поля класу
                       # через екземпляр класу
```

Статичні поля у Python також називаються **класовими полями**.

Статичні методи описують як методи класу, без параметру `self` у списку аргументів. Перед описом методу вказують декоратор `@staticmethod`.

Лістинг 1.11. Клас `Car` зі статичними полями та методами.

```
class Car:

    wheel_number = 4  # Статичне поле

    @staticmethod
    def getWheelNumber():
        """ Статичний метод
        :return: значення статичного поля wheel_number
        """
        return Car.wheel_number

# Звернення до статичного поля через ім'я класу
print(Car.wheel_number)

c = Car()
# Звернення до статичного поля через екземпляр класу
print(c.wheel_number)

# Виклик статичного методу через ім'я класу
print(Car.getWheelNumber())

# Виклик статичного методу через екземпляр класу
print(c.getWheelNumber())
```

Повертаючись до прикладу наведеного у лістингу 1.10, можна звернути увагу, що метод `isExist(self,a,b,c)` може бути перетворений у статичний. Дійсно, перевірка існування трикутника за заданими трьома сторонами аж ніяк не залежить від стану поточного екземпляру класу. Нижче, у лістингу, наведено фрагмент модифікованого класу, що містить статичний метод `isExist(...)`.

Лістинг 1.12. Клас `Triangle` зі статичним методом.

```
class Triangle:
    ...
    @staticmethod
    def isExist(a, b, c):
        """ Перевіряє, чи можна створити
        тикутник з заданими сторонами
        :param a: перша сторона трикутника
        :param b: друга сторона трикутника
        :param c: третя сторона трикутника
        :return: True, якщо трикутник з сторонами a, b, c існує
        """
        return a + b > c and a + c > b and c + b > a
    ...

# Перевірка чи існує трикутник зі сторонами 4, 5, 6
Triangle.isExist(4, 5, 6)  # через ім'я класу

t = Triangle(3, 4, 5)
# Перевірка чи існує трикутник зі сторонами 4, 5, 6
t.isExist(4, 5, 6)         # через екземпляр класу
```

1.4. Графічна бібліотека turtle та її застосування

Графічна бібліотека `turtle`, що розглядається у цьому пункті хоча і не має прямого відношення до об'єктно-орієнтованого програмування, проте може стати у нагоді для його простішого опанування.

Цей модуль дозволяє зображувати прості фігури на екрані у графічному режимі. Фактично `turtle` – це графічний курсор, який можна пересувати по екрану віддаючи певні команди, наприклад, «рухатися вперед на 10 пікселів», «намалювати коло» тощо.

Модуль `turtle` містить багато різноманітних функцій. Основні його операції та можливості наведено у таблиці нижче. З повною документацією можна ознайомитися за посиланням docs.python.org/3/library/turtle.html

Таблиця 1.1. Методи бібліотеки turtle

Дія	Опис
`home()`	Перевести курсор у початкову позицію (центр вікна)
`reset()`	Очищає екран та переводить курсор у початкову позицію.
`bye()`	Завершує роботу turtle
`mainloop()`	Затримує графічне вікно на екрані.
`delay(pause)`	Визначити затримку у мілісекундах між

	окремими рухами курсора
`speed(speed)`	Встановлює швидкість курсора. Параметр speed має бути у діапазоні від 1 (повільно) до 10 (швидко) або 0 (миттєво).
`up()`	Підняти пензель (припинити малювання)
`down()`	Опустити пензель (почати малювання)
`forward(distance)`	Зміститися вперед на distance пікселів.
`backward(distance)`	Зміститися назад на distance пікселів.
`setpos(x, y)`	Встановити курсор у позицію (x, y)
`goto(x, y)`	Зміститися у позицію (x, y).
`left(angle)`	Поворот вліво на кут angle (у градусах)
`right(angle)`	Поворот вправо на кут angle (у градусах)
`setheading(angle)`	Встановити кут положення курсора angle.
`dot(size, color)`	Зображує точку діаметром size кольором color. Параметр color є необов'язковим.
`circle(radius)`	Зображує коло на екрані радіусом radius. Коло зображується, починаючи з його нижньої точки.
`pencolor(color)`	Встановити колір ліній.
`pencolor()`	Повертає поточний колір ліній.
`fillcolor(color)`	Встановити колір заповнення.
`bgcolor()`	Повертає колір фону

Приклад 1.6. Використовуючи бібліотеку `turtle` зобразимо на екрані квадрат зі стороною 50 пікселів.

Лістинг 1.13. Зображення квадрату.

```
from turtle import * # Підключаємо модуль turtle

reset()      # Ініціалізуємо режим малювання

# === Малюємо квадрат ===
down()       # Опускаємо перо
forward(50)  # Рухаємося вперед на 50 пікселів,
             # тобто зображуємо першу сторону квадрата
left(90)     # Поворот вліво на 90 градусів
forward(50)  # Зображуємо другу сторону квадрата
left(90)     # Поворот вліво на 90 градусів
forward(50)  # Зображуємо третю сторону квадрата
left(90)     # Поворот вліво на 90 градусів
forward(50)  # Зображуємо четверту сторону квадрата
up()         # Підіймаємо перо
# === Зображення квадрата завершено ===
```

```
mainloop()        # Затримуємо вікно на екрані
```

Приклад 1.7. Використовуючи бібліотеку `turtle` напишемо програму, що зображує коло на екрані. Проте цього разу застосуємо об'єктно-орієнтований підхід. Опишемо клас Коло, екземпляри якого будуть містити такі атрибути, як положення кола на екрані, його розмір, а також його колір зображення. Крім цього реалізуємо методи, що зображують коло на екрані, ховають його, та переміщують на задану позицію.

Лістинг 1.14. Клас Коло та його зображення на екрані.

```
import turtle as t # псевдонім для компактнішого запису

class Circle:
    """ Клас Коло """

    def __init__(self, x, y, r, color):
        """ Конструктор
        Ініціалізує положення кола, його радіус і колір
        :param x: координата x центру кола
        :param y: координата y центру кола
        :param r: радіус кола
        :param color: колір кола
        """
        self._x = x  # _x - координата x центру кола
        self._y = y  # _y - координата y центру кола
        self._r = r  # _r - радіус кола
        self._visible = False  # _visible - чи є коло
                               # видимим на екрані
        self._color = color    # _color - колір кола

    def _draw(self, color):
        """ Допоміжний метод, що зображує коло заданим кольором
        :param color: колір
        """
        t.pencolor(color)
        t.up()
        # малює починаючи знизу кола
        t.setpos(self._x, self._y - self._r)
        t.down()
        t.circle(self._r)
        t.up()

    def show(self):
        """ Зображує коло на екрані """
        if not self._visible:
```

```
            self._visible = True
            self._draw(self._color)

    def hide(self):
        """ Ховає коло (робить його невидимим на екрані) """
        if self._visible:
            self._visible = False
            # щоб сховати коло, потрібно його
            # зобразити кольором фону.
            self._draw(t.bgcolor())

    def move(self, dx, dy):
        """ Переміщує об'єкт
        :param dx: зміщення у пікселях по осі X
        :param dy: зміщення у пікселях по осі Y """
        vis = self._visible
        if vis:
            self.hide()
        self._x += dx
        self._y += dy
        if vis:
            self.show()

# Перевірка роботи класу
if __name__ == '__main__':
    t.home()
    t.delay(10)
    c = Circle(120, 120, 50, "blue")
    c.show()
    c.move(-30, -140)
    t.mainloop()
```

1.5. Вступ до UML

Для наочного графічного зображення класів у парадигмі об'єктно-орієнтованого програмування зручно використовувати уніфіковану мову візуального моделювання UML (Unified Modeling Language). Різні види діаграм, які підтримуються UML, і багатий набір можливостей зображення певних аспектів системи робить UML універсальним засобом опису як програмних, так і ділових систем. UML може бути застосовано на всіх етапах життєвого циклу аналізу бізнес-систем і розробки прикладних програм.

Діаграми дають можливість зобразити систему (як ділову, так і програмну) у такому вигляді, щоб її можна було легко перевести в програмний код.

Наразі нам стане у нагоді діаграма, що називається **діаграмою класів.**

Означення 1.14. Діаграма класів – це (engl. class diagram) статичне зображення структури моделі, яке відображає статичні (декларативні) елементи, такі як класи, типи даних, їх зміст та відношення.

Нижче наведено простий приклад діаграми класів

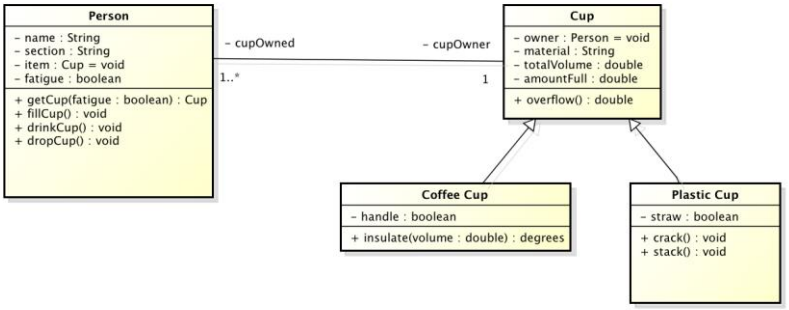


Рисунок 1.4. Діаграма класів

Діаграма класів є ключовим елементом в об'єктно-орієнтованому моделюванні. На діаграмі класи зображуються у прямокутниках, що містять три секції (див. рисунок 1.5):

- **Вгорі ім'я класу**. Ім'я класу вирівнюється по центру і пишеться напівжирним шрифтом. Імена класів починаються з великої літери.
- **Посередині** розташовуються **поля** (**атрибути) класу**. Вони вирівняні по лівому краю.
- **Нижня частина** містить **методи класу**. Вони також вирівняні по лівому краю.

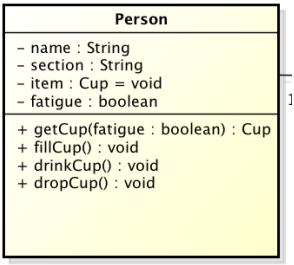


Рисунок 1.5. Клас на діаграмі класів UML

Залежно від ситуації, для спрощення діаграм, секції атрибутів і методів можуть не відображатися.

Атрибут на діаграмі визначається іменем і може записуватися разом з типом, видимістю, початковим значенням, тощо. Всі ці додаткові параметри, залежно від ситуації можуть опускатися (крім імені). Загальний синтаксис зображення атрибута має вигляд:

```
Видимість Ім'я : Тип [Множинність] = Початкове_значення
```

де `Ім'я` ім'я атрибуту, що визначається програмістом, `Тип` – його тип даних (наприклад, `int`, `float`), `Поч_значення` – початкове значення атрибуту. `Множинність` – діапазон індексів – вказує на те, якщо атрибут є індексованим набором. `Видимість` – тип доступу до атрибуту.

Як і атрибут, метод, визначається іменем, поруч з яким записуються видимість, список параметрів, тип даних, що повертає метод. Загальний синтаксис зображення методу такий:

```
Видимість Ім'я : (Список_параметрів) : Тип_результату
```

де `Ім'я` ім'я методу, що визначається програмістом, `Список_параметрів` – список вхідних формальний параметрів методу, `Видимість` – тип доступу до атрибуту, `Тип_ результату` – тип даних, що повертає метод.

Видимість атрибутів та методів

Для позначення типу видимості атрибутів та методів класу, перед іменем відповідного елемента вказують позначку з наведеного у таблиці нижче переліку

Таблиця 1.2. Позначення для типів доступу

Позначення	Назва	Опис
+	Публічний (Public)	Будь-який клієнт класу може використовувати атрибут/метод.
-	Приватний (Private)	Атрибут/метод може використовуватися виключно самим класом.
#	Захищений (Protected)	Будь-який нащадок класу може використовувати атрибут/метод.
~	Пакет (Package)	Будь-який клієнт класу, що оголошений у цьому ж пакеті, може використовувати цей атрибут

Якщо видимість атрибута/метода не вказана на діаграмі, то вважається, що атрибут/метод оголошений з публічним типом видимості.

Нагадаємо, що у Python видимість атрибутів/методів реалізується на рівні домовленості щодо синтаксису і є лише двох типів: приватний та публічний.

Відношення між класами

Класи та їхні екземпляри не існують у ізоляції один від одного – будь-яка програма це система взаємодіючих класів. При цьому одні класи можуть впливати на інші, класи можуть бути складовими частинами інших композитних класів і таке інше. Тоді, між такими взаємодіючими класами можуть встановлюватися різноманітні зв’язки.

Означення 1.15. Зв’язки (відношення або взаємозв’язки) – це тип логічних зв’язків між сутностями, що вказуються на діаграмах класів і об'єктів.

Найзагальніший тип зв’язків це

Означення 1.16. **Асоціація** – тип зв’язку, що відображає структурні відношення між екземплярами класів.

Асоціації забезпечують взаємозв’язки об’єктів, що належать до різних класів. Вони з’єднують у одне ціле всі елементи системи, завдяки чому виникає робоча система. Без асоціації система перетворилася б у набір ізольованих класів.

Іншими словами, асоціація означає, що об’єкти двох класів можуть посилатися один на одного. Для прикладу, розглянемо систему що складається з двох ключових класів Студент та Викладач. Екземпляр класу Студент навчається у екземпляра Викладач, у той час як Викладач навчає студентів – екземпляри класу Викладач. Тобто, студенти знають викладача (викладачів) у якого вони навчаються, у той час як викладач знає перелік студентів, котрих він навчає.

Під час створення моделі важливо вказати, скільки об'єктів одного класу пов'язано за допомогою асоціації з об’єктами другого класу і навпаки. Це число називається **потужністю** асоціації.

Потужність асоціації буває трьох типів:

- один-до-одного
- один-до-багатьох
- багато-до-багатьох

Асоціація на діаграмі класів позначається прямою або ламаною лінією, кінці якої з’єднують пов’язані класи. При цьому над лінією вказується потужність асоціації.

Розглянемо кілька прикладів, що пояснюють асоціацію між класами та їхню потужність.

Асоціація один-до-одного виникає коли кожному екземпляру одному класу відповідає рівно один екземпляр іншого класу. Прикладом такої асоціації може бути асоціація між класами Студент та Студентський квиток. Кожен студент має студентський квиток, у той час як студентський квиток виготовляється спеціально для одного студента.

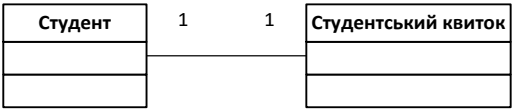


Рисунок 1.6. Асоціація один-до-одного

Асоціація один-до-багатьох, виникає коли екземпляру одного класу відповідає більше ніж один екземпляр іншого класу. Наприклад, на діаграмі нижче показана асоціація між класами Студент та Університет: кожен студент знає у якому (одному) університеті він навчається і навпаки, університет має список всіх своїх студентів (багатьох).

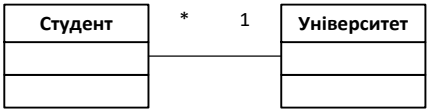


Рисунок 1.7. Асоціація один-до-багатьох

Наведена раніше асоціація між класами Студент та Викладач, матиме потужність багато-до-багатьох, адже кожен викладач має багато студентів, у той час як кожного студента навчають кілька викладачів.

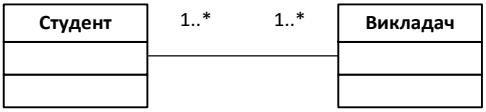


Рисунок 1.8. Асоціація багато-до-багатьох

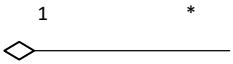
Асоціація визначає лише семантичний зв'язок. Вона не вказує на напрям і точну реалізацію відношення. Вона призначена для аналізу проблеми, коли необхідно лише ідентифікувати зв'язки та їхні потужності.

У загальному випадку асоціація позначає рівноправні відношення між класами. Наступні два типи зв'язків, **агрегація та композиція,** уточнюють асоціацію у випадку якщо один клас є частиною іншого. Такий тип асоціації, як правило, вказує на те, що один клас має атрибутами екземпляри іншого класу.

Означення 1.17. Агрегація – це різновид асоціації, що виникає при відношенні між цілим і його частинами, де час існування екземплярів агрегованого класу не пов'язаний з часом існування класу, що їх містить.

Агрегація зустрічається, коли один клас є колекцією або контейнером для інших. Причому, якщо контейнер буде знищений, то його вміст – ні. Наприклад, університет містить (тобто, агрегує) студентів та викладачів, що відповідно навчаються та викладають у ньому. Якщо університет ліквідують, викладачі та студенти продовжать існувати.

На UML діаграмах класів, агрегація позначається порожнім ромбом на блоці класу, і лінією, що йде від класу, що агрегується



Означення 1.18. Композиція – більш строгий різновид агрегації, що має жорстку залежність часу існування екземпляра класу контейнера та екземплярів класів, що містяться в ньому.

Як і агрегація, композиція зустрічається, коли один клас є колекцією або контейнером для інших. Проте, композиція – більш суворий варіант агрегації – якщо клас-контейнер буде знищений, то весь його вміст також буде знищено. Наприклад, колеса, кермо, мотор та інші вузли автомобіля є частиною автомобіля – при створенні автомобіля вони створюються разом з ним, а утилізація автомобіля у свою чергу призводить до знищення всіх його вузлів. А от водій з автомобілем пов'язаний агрегацією – екземпляр водій не створюється разом з автомобілем, а також, при утилізації автомобіля продовжує існувати.

Композицію часто називають агрегацією за значенням. Вона позначається подібним чином до агрегації, але з зафарбованим ромбом.

Нижче наведений приклад діаграми класів проекту, що складається з класів `Автомобіль`, `Водій` та `Колесо`.

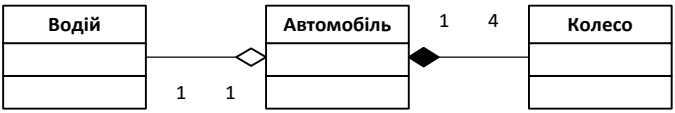


Рисунок 1.9. Діаграма класів

Наступний і останній у цьому параграфі тип зв'язків це залежність.

Означення 1.19. Залежність – це відношення, що показує, що зміни у одному класі (не залежному) можуть впливати на інший клас (залежний), що використовує його

Графічно на діаграмах класів, залежність зображується пунктирною лінією зі стрілкою, що направлена на незалежний клас. Найчастіше залежність вказує на те, що один клас використовує інший як аргумент у своєму методі. На рисунку зображеному нижче, клас Комп'ютер залежить від класу Програма (що є незалежним класом).

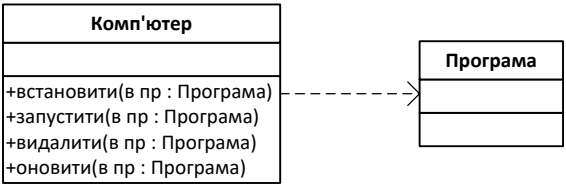


Рисунок 1.10. Залежність

Дійсно, у результаті встановлення програми, змінюється стан комп'ютера (наприклад у меню «Start» операційно системи Windows з'являється нова іконка). Крім цього програма, встановлена на комп'ютері, в результаті своєї діяльності, може змінювати його стан. Програма може оновитися, що також змінить стан комп'ютера в цілому.

Ми продовжимо знайомство з UML у наступному параграфі, після того, як познайомимося з такими концепціями об'єктно-орієнтованого програмування як наслідування і поліморфізм.

§2 НАСЛІДУВАННЯ ТА ПОЛІМОРФІЗМ

2.1. Наслідування

В загальних рисах ідея наслідування полягає у тому, що клас створюється не на «порожньому місці», а на основі вже створеного класу.

Означення 2.1. Наслідування – механізм ООП, що дозволяє класу успадковувати властивості та методи іншого класу.

З точки зору наслідування виділяються базовий (батьківський) клас та породжений (дочірній) клас: клас, що створюється є породженим класом, а клас на основі якого створюється дочірній – базовим класом.

Наслідування є досить гнучким механізмом, який дозволяє дочірньому класу не просто бути копією батьківського класу, але і доповнювати успадковані риси власними характеристиками: дочірній клас може мати додаткові атрибути і методи. Крім цього дочірній клас може змінити зміст успадкованих атрибутів та методів, замістивши їх.

Отже, наслідування дозволяє будувати спеціалізовані класи на основі базових, що дозволяє уникнути повторного написання коду. Внаслідок наслідування створюється **ієрархія класів**.

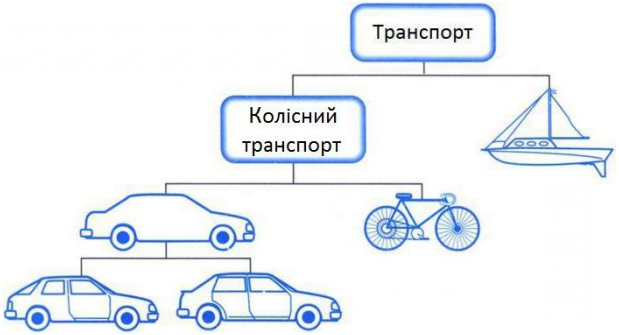


Рисунок 2.1. Ієрархія класів при наслідуванні

Концепція наслідування взята з реального життя – більшість класів, поділено на підкласи, що мають спільні властивості і поведінку успадковану від предків проте відрізняються від них новими атрибутами і методами. Наприклад, тварини діляться на земноводних, ссавців, комах тощо. Транспорт

поділяється на автомобільний, залізничний, водний, авіаційний – кожен з видів транспорту призначений для перевезення пасажирів чи вантажів, проте кожен з видів має свої особливості, що виділяють його з-поміж інших.

З точки зору ієрархії класів (крім відношення батьківський-дочірній) виділяють відношення **клас-предок** (суперклас) та **клас-нащадок** (підклас): клас, що породжений ланцюгом наслідувань, починаючи з деякого називається **нащадком** цього класу. Наприклад у ієрархії класів, зображених на рисунку 2.1 клас «Велосипед» є нащадком класу «Транспорт». При цьому клас «Транспорт» є предком для усіх класів зображених у ієрархії. Дочірній клас також називають безпосереднім нащадком, а батьківський клас – безпосереднім предком.

Для опису дочірнього класу на основі базового у Python використовується такий шаблон:

```
class NewClass(BaseClass):
    class_body
```

де `NewClass` – ім'я створюваного класу, `BaseClass` – ім'я базового класу на основі якого створюється клас `NewClass`, `class_body` – тіло класу – тут описуються та заміщуються методи, класові поля і т.д., тобто те, що відрізняє клас-нащадок від базового класу.

Приклад 2.1. Створимо клас `Pet` – клас тварина. Успадкуємо від нього класи `Dog`, `Cat` та `Parrot` у кожному з яких реалізуємо метод **`voice`**`()`, що виводить на екран повідомлення «тваринною мовою» відповідно до класу тварини.

Як правило, кожна домашня тварина має ім'я (кличку). Тому це поле можна створити у базовому класі `Pet`. Крім цього реалізуємо метод, що повертає ім'я тварини.

Лістинг 2.1. Базовий клас `Pet`.

```
class Pet:
    def __init__(self, name):
        """ Конструктор
        :param name: Кличка тварини
        """
        self._name = name  # приватне поле - кличка тварини

    def getName(self):
        """ Повертає кличку тварини
        :return: кличку тварини
        """
        return self._name
```

Кожен з підкласів Dog, Cat та Parrot буде розширяти батьківський клас Pet, додавши метод **voice**().

Лістинг 2.2. Нащадки класу Pet.

```
class Cat(Pet):
    """ Клас Cat - нащадок класу Pet """
    def voice(self):
        """ Метод "голос" """
        print("Cat", self._name, "says:" , "Miu, miu, miu!!!")

class Dog(Pet):
    """ Клас Dog - нащадок класу Pet """
    def voice(self):
        """ Метод "голос" """
        print("Dog", self._name, "says:" , "Gav, gav, gav!!!")

class Parrot(Pet):
    """ Клас Parrot - нащадок класу Pet """
    def voice(self):
        """ Метод "голос" """
        print("Parrot says:", self._name + " horoshy!")

# Створюємо домашніх тварин як екземпляри відповідних класів
my_cat = Cat("Kuzya")        # кличка кота - Кузя
my_dog = Dog("Barbos")       # кличка собаки - Барбос
my_parrot = Parrot("Flint")  # кличка папуги – Флінт

# Викликаємо метод voice для кожного об'єкту
my_cat.voice()
my_dog.voice()
my_parrot.voice()
```

Результатом роботи вищенаведеного коду буде повідомлення

```
Cat Kuzya says: Miu, miu, miu!!!
Dog Barbos says: Gav, gav, gav!!!
Parrot says: Flint horoshy!
```

Заміщення методів

В результаті наслідування, всі поля і методи з базового класу успадковуються нащадком у повному обсязі. При цьому під час наслідування існує можливість замістити методи успадковані нащадком. Фактично

заміщення методу полягає у тому, що для класу-нащадку визначається інша поведінка ніж та, що передбачена суперкласом.

Означення 2.2. Заміщення методу (англ. Method overriding) механізм мови програмування, що дозволяє підкласу надавати специфічну реалізацію методу, що вже реалізований в одному із суперкласів.

Для заміщення методу потрібно у класі-нащадку описати метод, з таким же іменем і списком формальних параметрів, що й у суперкласі.

Вибір версії методу (оригінальний чи заміщений), визначається типом екземпляру класу, з якого викликається цей метод: якщо виклик методу відбувається з об'єкту батьківського класу, то виконується метод батьківського класу, якщо ж викликає екземпляр підкласу, то виконуватиметься заміщена версія методу.

Механізм заміщення методів є основою поліморфізму у об'єктно-орієнтованому програмуванні. Детальному вивченню поліморфізму присвячений наступний пункт підручника.

Приклад 2.2. Для демонстрація механізму заміщення методів та їхні виклики, додамо у базовий клас `Pet`, що описаний вище у лістингу 2.1 метод **`voice`**`()`.

Лістинг 2.3. Базовий клас `Pet` з методом **`voice`**`()`.

```
class Pet:
    ...
    def voice(self):
        """ Метод "голос" """
        print("I'm a base class, I say nothing...")
```

Вважаючи, що клас `Cat` з лістингу 2.2 будується як нащадок цього класу, створимо екземпляр базового класу і екземпляр класу `Cat`. Для кожного з цих екземплярів викликатимемо метод **`voice`**`()`.

Лістинг 2.4. Базовий клас `Pet` з методом **`voice`**`()`.

```
base = Pet("No name")  # екземпляр базового класу
my_cat = Cat("Kuzya")  # кіт Кузя

base.voice()
my_cat.voice()
```

Результатом коду наведеного вище буде таке повідомлення:

```
I'm a base class, I say nothing...
Cat Kuzya says: Miu, miu, miu...
```

Особливості роботи з методом-конструктором

У прикладі 2.1 у жодному з класів нащадків не було явно описано метод-конструктор. Якщо у класах-нащадках конструктор (тобто метод **__init__**()) не описується явно (не заміщується), то, під час створення об'єкта, Python викликає конструктор базового класу. Отже, при створенні класу-нащадку, потрібно передати набір фактичних аргументів, що відповідають набору формальних параметрів конструктора базового класу.

Якщо ж задача передбачає розширити базовий клас не лише новими методами, але й новими полями, то швидше за все, постане задача замістити у підкласі метод-конструктор. У такому разі, першою інструкцією заміщеного конструктора має бути виклик батьківського конструктора: це необхідно, для того, щоб сконструювалися атрибути описані у базовому класі.

Приклад 2.3. Опишемо клас `Parrot`, що є нащадком класу `Pet`, у якому додамо поле, яке містить ім'я господаря папуги.

Лістинг 2.5. Клас `Parrot` з заміщеним конструктором

```
class Parrot(Pet):
    """ Клас Parrot - нащадок класу Pet """

    def __init__(self, name, masterName):
        """ Заміщений конструктор
        :param name: ім'я папуги
        :param masterName: ім'я господаря
        """
        Pet.__init__(self, name) # обо'язковий виклик
                                 # конструктора базового класу
        self._masterName = masterName  # ім'я господаря
    ...

# Створюємо папугу Капітана Флінта на ім'я Флінт
flint = Parrot("Flint", "Captain Flint")
flint.voice()
```

Звернемо увагу, що виклик конструктора базового класу відбувається з використанням імені базового класу, а у списку аргументів, першим параметром обов'язково вказується параметр **self** – посилання на екземпляр класу:

```
        BaseClass.__init__(self, *args)
```

де `BaseClass` – ім'я базового класу, `*args` – послідовність аргументів.

Наприклад, у вищенаведеному класі `Parrot` виклик конструктора базового класу `Pet` виглядає таким чином

```
        Pet.__init__(self, name)
```

Для того, щоб викликати однойменний метод з базового класу, наприклад, метод-конструктор, можна використовувати функцію **`super`**`()`. Її виклик, не вимагає вказання імені базового класу і посилання на екземпляр класу **`self`**, наприклад, у лістингу 2.5 замість вищенаведеного виклику конструктора можна використати такий

Лістинг 2.6. Клас `Parrot` з заміщеним конструктором

```
class Parrot(Pet):
    """ Клас Parrot - нащадок класу Pet """

    def __init__(self, name, masterName):
        super().__init__(name) # виклик конструктора базового
                               # класу через функцію super()
        self._masterName = masterName  # ім'я господаря
    ...
```

Виклик методів базового класу

В об'єктно-орієнтованому програмуванні часто виникає ситуація при якій потрібно у дочірньому класі викликати метод батьківського класу.

Якщо метод **не заміщений**, то виклик методу батьківського класу здійснюється звичайним чином:

```
        self.method(*args)
```

Якщо ж метод заміщений, то вищенаведений виклик буде здійснювати виклик заміщеного методу. Для того, щоб викликати однойменний метод з базового класу, використовують у ролі префіксу ім'я відповідного класу-предка (а список фактичних аргументів починається з параметра **`self`**) :

```
BaseClass.method(self, *args)
```

Інший спосіб передбачає використання функції **super**(), з якою ми вже познайомилися у попередньому пункті.

```
super().method(*args) # виклик методу базового класу
```

Фактично функція **super**() повертає екземпляр базового класу. Таким чином використання цієї функції дозволяє викликати методи безпосередньо з базового класу. Детальніше про функцію **super**() буде розказано у пункті 2.4.

Приклад 2.4. Опишемо клас Equation (Рівняння), що моделює лінійне алгебраїчне рівняння виду

$$bx + c = 0.$$

Опишемо у ньому метод solve(), що повертає усі розв'язки цього рівняння, залежно від його коефіцієнтів.

Лістинг 2.7. Клас Equation – лінійне алгебраїчне рівняння.

```
class Equation:
    """ Клас лінійне рівняння виду bx + c = 0 """

    def __init__(self, b, c):
        """ Конструктор
        :param b: коефіцієнт рівння при x
        :param c: вільний член рівння
        """
        self.b = b
        self.c = c

    def solve(self):
        """ Повертає розв'язки рівння у вигляді кортежу
        [] - порожній список, якщо рівнняння немає жодного
             розв'зку b == 0, c != 0,
        ['inf'] - список, що містить один рядковий літерал 'inf',
             якщо рівнняння має безліч розв'язків b == 0, c == 0,
        [x1] список з одного елементу, що є розв'язком рівняння
        :return: список розв'язків рівння """
        if self.b == 0 and self.c != 0:
            return []
        elif self.b == 0 and self.c == 0:
            return ['inf']
        else:
            return [-self.c / self.b]
```

Створимо тепер класи QuadraticEquation (Квадратне рівняння) та BiQuadraticEquation (Біквадратне рівняння), що є нащадками класу Equation та моделюють рівняння виду

$$ax^2 + bx + c = 0.$$

та

$$ax^4 + bx^2 + c = 0.$$

відповідно.

Звернемо увагу читача на той факт, що, якщо коефіцієнт a квадратного рівняння дорівнює нулю, то рівняння вироджується у лінійне і, тому у цьому випадку, при реалізації методу `solve()` у класі `QuadraticEquation` можна скористатися методом `solve()` з батьківського класу `Equation`.

Лістинг 2.8. Клас QuadraticEquation – квадратне рівняння.

```
class QuadraticEquation(Equation):
    """ Клас квадратне рівняння виду ax**2 + bx + c = 0 """

    def __init__(self, a, b, c):
        """ Конструктор
        :param a: коефіцієнт рівняння при степені 2
        :param b: коефіцієнт рівняння при x
        :param c: вільний член рівняння
        """
        super().__init__(b, c)
        self.a = a

    def d(self):
        """ Повертає дискримінант квадратного рівняння
        :return: значення дискримінанту """
        return self.b ** 2 - 4 * self.a * self.c

    def solve(self):
        if self.a == 0:   # квадратне рівняння вироджується у лінійне
            return super().solve() # використаємо успадкований метод
        else:
            d = self.d()  # обчислення дискримінанту
            if d < 0:     # розв'язків немає
                return []
            elif d == 0:  # один розв'язок
                return [- self.b / (2.0 * self.a)]
            else:         # два розв'язки
                sqr_d = d ** 0.5  # корінь з дискримінанту
                x1 = (- self.b - sqr_d) / (2.0 * self.a)
                x2 = (- self.b + sqr_d) / (2.0 * self.a)
                return [x1, x2]
```

Для реалізації класу Біквадратне рівняння достатньо описати лише метод `solve()`, що використовує метод `solve()` з батьківського класу. Дійсно, при заміні $y = x^2$ біквадратне рівняння перетворюється у квадратне. Тому для остаточного розв'язання вихідного рівняння досить, якщо це можливо, знайти квадратні корені зі списку отриманих розв'язків квадратного рівняння.

Лістинг 2.9. Клас BiquadraticEquation – біквадратне рівняння.

```
class BiquadraticEquation (QuadraticEquation):
    """  Клас біквадратне рівняння виду ax**4 + bx**2 + c = 0 """

    def solve(self):
        result = []
        solutions = super().solve()  # розв'язки квадратного рівняння
                                     # при заміні y = x**2
        for y in solutions:
            if y == "inf":
                result.append("inf")
            elif y == 0:
                result.append(0)
            elif y > 0:
                x1 = y ** 0.5
                x2 = -x1
                result.append(x1)
                result.append(x2)
        return result
```

Клас object

Починаючи з 3-ї версії Python всі класи мають спільного предка – суперклас `object`. Всі класи, що створюються розробниками, неявно створюються на базі цього класу, тобто автоматично будуть його нащадками, та успадковують всі його атрибути.

2.2. Поліморфізм та віртуальні методи

Наслідування та заміщення методів є основою поліморфізму у контексті об'єктно-орієнтованого програмування.

Означення 2.3. Поліморфізм – концепція програмування, відповідно до якої, може використовується спільний інтерфейс для обробки різних спеціалізованих типів.

У контексті об’єктно-орієнтованого програмування поліморфізм означає, що об'єкти двох або більше класів можуть реагувати по-різному на однакові команди. Іншими словами, поліморфізм це підхід, що дозволяє одне й те саме ім'я використовувати для розв'язання двох або більше схожих, але технічно різних задач.

Розглянемо такий приклад

```
print(2 + 2)
print("A" + "B")
```

Як бачимо у обох інструкціях використовується один і той самий оператор додавання. Проте, результат його роботи буде залежати від типів операндів, до яких він застосовується. Таким чином результатом роботи інструкції у першому рядку буде виведення на екран результату арифметичного додавання, у той час як результатом роботи інструкції другого рядка – конкатенація двох рядків

```
4
AB
```

З поліморфізмом ми вже зустрічалися у попередніх прикладах. Так, наприклад, метод **voice**() виводив різні повідомлення, залежно від того, з якого класу він викликався (Dog, Cat або Parrot). Метод **solve**() повертав розв’язки відповідного рівняння (лінійного, квадратного чи біквадратного).

У об'єктно-орієнтованому програмуванні найпоширенішим різновидом поліморфізму є здатність екземплярів підкласу грати роль об'єктів батьківського класу, завдяки чому екземпляри підкласу можна використовувати там, де використовуються екземпляри батьківського класу. Продемонструємо це на прикладі.

Приклад 2.5. Змінимо клас Pet, додавши туди поле _masterName – ім’я господаря. Крім цього опишемо у цьому класі метод **showInfo**(), що буде виводити на екран всю інформацію про тварину – її кличку, ім’я господаря та те як вона подає голос. Для останнього будемо викликати методи **getType**() та **voice**(), оголошені у класі Pet та заміщені у класі Dog – нащадку класу Pet.

Лістинг 2.10. Клас Pet.

```
class Pet:
    def __init__(self, name, masterName):
        """ Конструктор
        :param name: Кличка тварини
        """
        self._name = name  # приватне поле - кличка тварини
```

```
        self._masterName = masterName  # ім'я господаря

    def getType(self):
        """ Повертає тип тварини
        :return: рядок "Pet"
        """
        return "Pet"

    def getName(self):
        """ Повертає кличку тварини
        :return: кличку тварини
        """
        return self._name

    def voice(self):
        """ Метод "голос" """
        print("I'm a base class, I say nothing...")

    def showInfo(self):
        print(self.getType() + "'s name is " + self._name)
        print(self.getType()+ "'s master is "+self._masterName)
        print(self.getType() + " says: ", end="")
        self.voice()
```

Лістинг 2.11. Клас Dog, нащадок класу Pet.

```
class Dog(Pet):
    """ Клас Dog - нащадок класу Pet """

    def getType(self):
    """ Повертає тип тварини
    :return: рядок "Dog"
    """
        return "Dog"

    def voice(self):
        """ Заміщений метод "голос" """
        print("Bau-bau!")

# Створюємо собаку Тоббі інспектора Лестрейда
flint = Dog("Toby", "Inspector Lestrade")
flint.showInfo()
```

Результатом роботи програми буде таке повідомлення

```
Dog's name is Toby
Dog's master is Inspector Lestrade
```

```
Dog says: Bau-bau!
```

Як бачимо, метод **showInfo**(), що викликається з базового класу викликає методи **getType**() та **voice**() з дочірнього класу. Хоча можна було очікувати, що буде викликаний метод з базового класу. Це відбувається через те, що під час виклику метода (тобто під час виконання програми), Python перевіряє до якого типу належить екземпляр класу і в результаті цього викликає метод.

У нашому прикладі, екземпляр класу належав до типу Dog, що містить заміщені методи **getType**() та **voice**() – відповідно саме їх і викликав інтерпретатор.

У теорії програмування така властивість називається **віртуальністю** (або динамічним зв'язуванням об'єктів та методів), а методи (що динамічно зв'язуються з об'єктами) – відповідно **віртуальними**.

Означення 2.4. Віртуальний метод – метод класу, який може бути заміщений у класах-нащадках так, що конкретна реалізація методу для виклику буде визначатися під час виконання.

Отже, програмісту необов'язково знати точний тип об'єкта для роботи з ним через віртуальні методи: досить лише знати, що об'єкт належить класу або нащадку, у якому метод оголошено.

Віртуальні методи – один з найважливіших прийомів реалізації поліморфізму. Вони дозволяють створювати узагальнений код, який може працювати як з об'єктами базового класу, так і з об'єктами будь-якого з його нащадків.

У Python реалізоване динамічне зв'язування (оскільки програми інтерпретуються, а не компілюються) і, відповідно, **всі методи у Python є віртуальними**.

Розглянемо тепер інший досить показовий приклад застосування поліморфізму у об'єктно-орієнтованому програмуванні.

Приклад 2.6. У прикладі 1.7 ми розглянули застосування об'єктно-орієнтованого підходу до зображення кіл на екрані за допомогою графічної бібліотеки `turtle`. Очевидно, що такий підхід легко перенести на інші геометричні фігури (прямокутник, трикутник, тощо). Проте якщо уважно проаналізувати код реалізації цього прикладу наведений у лістингу 1.14 то легко помітити, що у реалізаціях відповідних класів буде багато спільного, тобто фактично буде відбуватися дублювання коду. Спробуємо реалізувати універсальний підхід для зображення різноманітних плоских фігур на екрані, так щоб мінімізувати дублювання коду. Як ми знаємо, щоб зобразити фігуру на екрані, потрібно знати її положення, колір лінії та правило за яким вона малюється. Причому лише останнє залежить від типу фігури. Це наштовхує на думку, що можна реалізувати ієрархію класів таким чином, щоб спільні методи

і властивості містилися у базовому класі, а те що їх розрізняє у відповідних нащадках.

Отже повертаючись до лістингу 1.14 можна стверджувати, що всі методи, крім метода **_draw**(), не залежать від типу фігури і можуть бути оголошені у базовому класі. А от метод **_draw**() буде реалізований у нащадках, оскільки він напряму залежить від типу фігури. Цей метод буде віртуальним методом. Тому, для цілісної структури базового класу, метод **_draw**() також оголосимо у базовому класі, проте з порожньою реалізацією.

Базовий клас назвемо `Figure`. Його реалізація, що майже тривіальним чином отримана з класу Circle лістингу 1.14, наведена нижче

Лістинг 2.12. Клас `Figure`.

```
from turtle import *

class Figure:
    """ Клас Фігура """

    def __init__(self, x, y, color):
        """ Конструктор
        :param x: координата x положення фігури
        :param y: координата y положення фігури
        :param color: колір фігури
        """
        self._x = x  # _x - координата x
        self._y = y  # _y - координата y
        self._visible = False  # _visible - чи є фігура
                               # видимою на екрані
        self._color = color    # _color - колір фігури

    def _draw(self, color):
        """ Допоміжний метод, що зображує фігуру
        Тут здійснюється лише декларація методу, а конкретна
        реалізація буде здійснюватися у конкретних нащадках
        :param color: колір
        """
        pass

    def show(self):
        """ Зображує фігуру на екрані """
        if not self._visible:
            self._visible = True
            self._draw(self._color)

    def hide(self):
        """ Ховає фігуру (робить її невидимою на екрані) """
```

```
        if self._visible:
            self._visible = False
            # щоб сховати фігуру, потрібно
            # зобразити її кольором фону.
            self._draw(bgcolor())

    def move(self, dx, dy):
        """ Переміщує об'єкт
        :param dx: зміщення у пікселях по осі X
        :param dy: зміщення у пікселях по осі Y
        """
        isVisible = self._visible
        if isVisible:
            self.hide()
        self._x += dx
        self._y += dy
        if isVisible:
            self.show()
```

Опишемо тепер конкретні реалізації цього класу у нащадках. Для прикладу опишемо два класи – `Circle` (Коло) і `Quadrate` (Квадрат). Як уже було зауважено ці класи будуть унаслідуватися від базового класу `Figure` і відрізнятися між собою лише конструктором та методом **`_draw`**`()`.

Лістинг 2.13. Клас `Circle`, нащадок класу `Figure`.

```
from turtle import *

class Circle (Figure):
    """ Клас Коло """
    def __init__(self, x, y, r, color):
        """ Конструктор
        Ініціалізує положення кола, його радіус і колір
        :param x: координата x центру кола
        :param y: координата y центру кола
        :param r: радіус кола
        :param color: колір кола
        """
        # Обов'язковий виклик конструктора базового класу
        super().__init__(x, y, color)
        self._r = r  # _r - радіус кола

    def _draw(self, color):
        """ Допоміжний метод, що зображує коло заданим кольором
        :param color: колір
        """
        pencolor(color)
```

```
        up()
        # малює починаючи знизу кола
        setpos(self._x, self._y - self._r)
        down()
        circle(self._r)
        up()

# Перевірка роботи класу
if __name__ == '__main__':
    home()
    delay(10)
    c = Circle(120, 120, 50, "blue")
    c.show()
    c.move(-30, -140)
    mainloop()
```

Лістинг 2.14. Клас `Quadrate`, нащадок класу `Figure`.

```
from turtle import *

class Quadrate(Figure):
    """ Клас Квадрат """

    def __init__(self, x, y, a, color):
        """ Конструктор
        Ініціалізує положення лівого нижнього кута квадрата,
        довжину його сторони і колір.
        :param x: координата x лівого нижнього кута квадрата
        :param y: координата y лівого нижнього кута квадрата
        :param a: довжина сторони квадрата
        :param color: колір квадрата
        """
        # Обов'язковий виклик конструктора базового класу
        super().__init__(x, y, color)
        self._a = a  # _a - довжина сторони квадрата

    def _draw(self, color):
        """ Допоміжний метод, що зображує квадрат
        :param color: колір
        """
        pencolor(color)
        up()
        # встановлюємо позицію лівого нижнього кута квадрата
        setpos(self._x, self._y)
        setheading(0)
        down()
        forward(self._a)  # перша сторона квадрата,
```

```
        left(90)
        forward(self._a)  # друга сторона квадрата
        left(90)
        forward(self._a)  # третя сторона квадрата
        left(90)
        forward(self._a)  # четверта сторона квадрата
        up()

# Перевірка роботи класу
if __name__ == '__main__':
    home()
    delay(30)
    q = Quadrate(0, 0, 150, "red")
    q.show()
    q.move(0, 140)
    q.hide()
    mainloop()
```

2.3. Наслідування на діаграмах класів

На UML діаграмах класів, наслідування позначається лінією, що з'єднує базовий та дочірній класи, і закінчується порожнім трикутником біля базового класу:

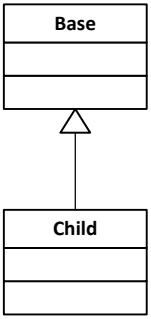


Рисунок 2.2. Наслідування на діаграмах класів

Зауважимо, що під час зображення наслідування у діаграмі класів у класах нащадках, як правило, не вказують атрибути і методи, що успадковані від базового класу. Виключенням може бути лише випадок, коли у класі-нащадку заміщується метод і це заміщення необхідно підкреслити, наприклад для зауваження його поліморфної сутності.

Як ми знаємо, у результаті операції наслідування, будується ієрархія класів. Нижче наведений приклад ієрархія класів домашніх тварин

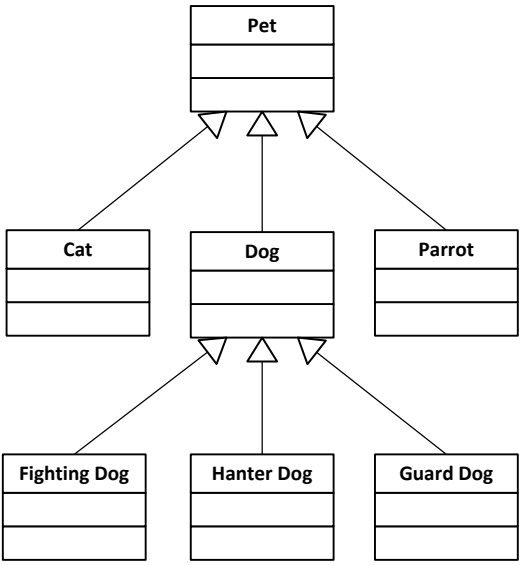


Рисунок 2.3. Приклад ієрархії класів домашніх тварин

2.4. Множинне наслідування

Іноді необхідно створити клас, що наслідує риси двох або більше класів. Наприклад, офіціант-співак повинен одночасно мати риси як офіціанта так і співака і, відповідно, здатен демонструвати двояку поведінку.

Означення 2.5. Наслідування від двох або більше базових класів називається **множинним наслідуванням**.

В результаті множинного наслідування, клас-нащадок успадковує всі поля і методи кількох базових класів. Множинне наслідування досить суперечливий механізм, тому у багатьох мовах програмування в чистому вигляді множинне наслідування заборонене.

У мові Python множинне наслідування дозволяється. Для того, щоб створити новий клас на основі кількох базових класів, використовується конструкція мови:

```
class NewClass(BaseClass1, BaseClass2):
    class_body
```

де NewClass – ім'я створюваного класу, BaseClass1, BaseClass2 – імена двох базових класів на основі яких створюється клас NewClass, class_body – тіло класу.

Базових класів може бути і більше двох – всі їхні імена перелічуються так само через кому.

Зауважимо, що порядок імен базових класів має значення – якщо під час виклику методу чи зверненні до поля, його не знайдено в успадкованому класі NewClass , то порядок відшукання методів у базових класах буде такий: спочатку буде відбуватися пошук у класі BaseClass1, а вже потім у класі BaseClass2.

Проблеми пов'язані з множинним наслідуванням

Основною проблемою, що може виникнути при множинному наслідуванні є неоднозначність вибору методів, якщо у базових класах є методи з однаковими іменами, а дочірній клас їх не заміщує їх. Пояснимо це на такому прикладі. Припустимо, що у нас є два класи Ксерокс та Сканер. У кожному з цих класів нехай визначено метод **copy**().Створимо клас БФУ, що є нащадком цих класів. Очевидно, що якщо під час опису цього класу метод **copy**() не буде заміщений, то екземпляр класу БФУ зіштовхнеться з проблемою: який метод взяти, з класу Ксерокс чи Сканер?

Інша проблема, що виникає при множинному наслідуванні – проблема дублювання даних при ромбовидному наслідуванні. Розглянемо такий приклад. Розглянемо таку ієрархію класів:

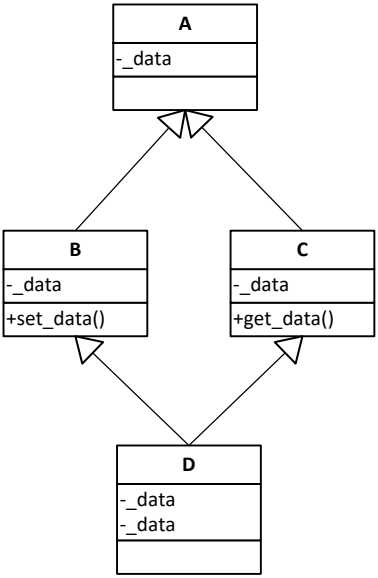


Рисунок 2.4. Ромбовидна ієрархія класів

Нехай у класі A визначена змінна _data. Як ми знаємо, при наслідуванні, клас-нащадок містить всі поля та методи базового класу. Отже кожен з класів B і C також містять екземпляр змінної _data. І у свою чергу, клас D буде містити всі поля і методи кожного з класів B і C тобто фактично екземпляр класу D буде містити два екземпляри класу A, а отже змінна _data буде входити до D двічі – один раз разом з екземпляром класу B, а інший – C!

Тепер подивимося, яким чином буде відбуватися роботи з цими даними. Нехай у класі B визначено метод **set_data**(), а у класі C – метод **get_data**(), які відповідно змінюють та повергають значення поля _data. Клас D отримує у спадок ці методи, проте, як ми вже зрозуміли, вони працюють з різними даними.

На щастя, на відміну від деяких інших мов програмування Python вміє розв'язувати обидві вищенаведені проблеми: першу, використанням порядку співставлення методів (Method Resolution Order або MRO), другу – перекриттям імен, фактично уникаючи дублювання.

Method Resolution Order

Method Resolution Order (MRO) визначає порядок, у якому Python шукає метод під час його виклику. Алгоритм, за яким будується порядок MRO базується на алгоритмі пошуку в глибину (проте не повторює його повністю).

Пошук методу починається з екземпляру класу для якого здійснюється виклик цього методу. Якщо у цьому класі методу не знайдено, здійснюється пошук у батьківському класі, що стоїть першим у переліку наслідуваних класів. Далі ця процедура здійснюється рекурсивно, доки або не буде знайдено шуканий метод, або не буде досягнуто класу, що не має предків. У випадку останнього відбувається повернення до дочірнього класу, звідки відбувається перехід до наступного у переліку наслідування класу. Як бачимо алгоритм на перший погляд здається складним, тому пояснимо його на такому прикладі.

Приклад 2.7. Нехай маємо ієрархію класів наведену нижче

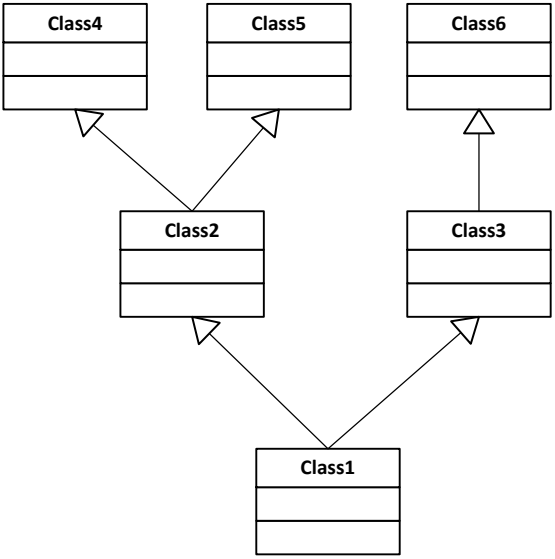


Рисунок 2.5. Ієрархія класів

Порядок пошуку методу, що викликається з екземпляру класу `Клас1` буде таким

```
Class1 -> Class2 -> Class4 -> Class5 -> Class3 -> Class6
```

На щастя, кожен клас у Python містить спеціальний атрибут із іменем `__mro__` який містить порядок класів для пошуку ідентифікаторів.

Створимо ієрархію класів зображену на рисунку 2.5 та виведемо на екран значення змінної `__mro__` для класу `Class1`.

Лістинг 2.15. Method Resolution Order.

```
class Class4: pass
class Class5: pass
class Class6: pass
class Class3(Class6): pass
class Class2(Class4, Class5): pass
class Class1(Class2, Class3): pass

print(Class1.__mro__)
```

Результатом роботи буде такий кортеж

```
(<class '__main__.Class1'>, <class '__main__.Class2'>, <class
'__main__.Class4'>, <class '__main__.Class5'>, <class
'__main__.Class3'>, <class '__main__.Class6'>, <class 'object'>)
```

Як бачимо перелік завершується класом `object`. Нагадаємо, що це спеціальний клас, що є базовим для усіх створюваних у програмі класів. Відповідно до алгоритму MRO, цей клас переглядається останнім (хоча це і суперечить алгоритму пошуку в ширину). Більше того, слід зауважити, що якщо кілька класів, мають спільного предка, те, всупереч алгоритму пошуку в глибину цей спільний предок розглядається алгоритмом MRO в останню чергу для усіх класів, що базуються на ньому. Продемонструємо це на ще одному прикладі.

Приклад 2.8. Нехай маємо ієрархію класів наведену на рисунку 2.6. Програма, що має таку ієрархію класів наведена у лістингу 2.16.

Лістинг 2.16. Method Resolution Order.

```
class Class7: pass
class Class8: pass
class Class9: pass
class Class6(Class7, Class8): pass
class Class4(Class6): pass
class Class5(Class6, Class9): pass
class Class2(Class4): pass
class Class3(Class5): pass
class Class1(Class2, Class3): pass

print(Class1.__mro__)
```

Результатом наведеної вище програми буде вивід послідовності пошуку методів, що матиме вигляд

```
(<class '__main__.Class1'>, <class '__main__.Class2'>, <class
'__main__.Class4'>, <class '__main__.Class3'>, <class
'__main__.Class5'>, <class '__main__.Class6'>, <class
'__main__.Class7'>, <class '__main__.Class8'>, <class
'__main__.Class9'>, <class 'object'>)
```

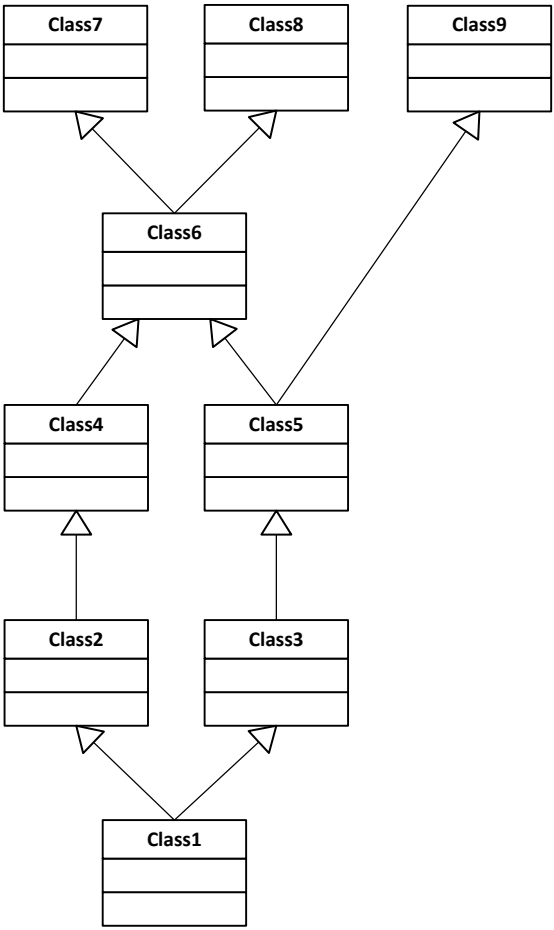


Рисунок 2.6. Ієрархія класів

Таким чином порядок обходу класів при пошуку відповідних методів буде таким:

```
Class1 -> Class2 ->Class4 -> Class3 -> Class5 -> Class6 -> Class7 ->
Class8 -> Class9 -> object
```

Звідки випливає, що пошук методу у класі `Class6` відбудеться лише після того, як буде здійснений пошук у всіх його нащадках, що є предками класу `Class1`. Хоча відповідно до алгоритму пошуку в глибину, цей клас має бути опрацьованим зразу після класу `Class4`.

Виклик методів базових класів

Раніше ми розповіли як відбувається звернення до методів батьківського класу при одинарному наслідування. Одним зі способів викликати однойменний метод з базового класу – скористатися функцію **`super`**`()`. Загалом, ця функція повертає найперший клас-предок відповідно до MRO, що містить метод який викликається. Наприклад, у випадку ієрархії класів зображених діаграмі класів на рисунку 2.7 при виклику методу `method()` з класу `D` використовуючи функцію **`super`**`()`

```
super().method()
```

буде викликано метод з класу `A`. Переконатися у цьому можна запустивши код зображений у лістингу 2.17.

Лістинг 2.17. Виклик метода з батьківського класу.

```
class A:
    def method(self):
        print("Called method from class A")

class B(A):
    pass

class C():
    def method(self):
        print("Called method from class C")

class D(B, C):
    def method(self):
        super().method()
```

```
d = D()
d.method()
```

Результатом роботи цієї програми буде повідомлення

```
Called method from class A
```

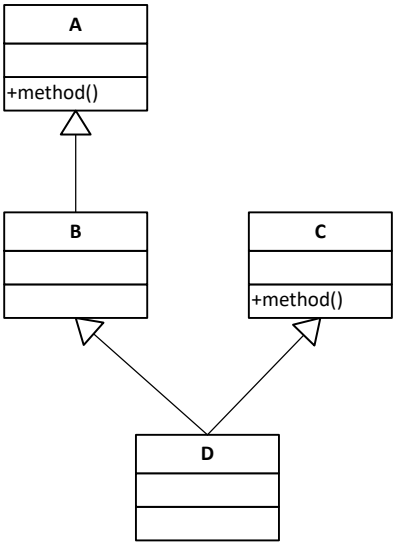


Рисунок 2.7. Ієрархія класів

Яким же чином, при необхідності, викликати метод `method()` з класу `C`? Для цього необхідно явно вказати клас-предок метод якого необхідно взяти, а у виклику методу першим параметром має бути посилання на поточний екземпляр класу `self`. Отже, якщо замінимо у лістингу 2.17. опис методу `method()` у класі `D` на такий

```
def method(self):
    C.method(self)  # виклик методу method() з класу C
```

отримаємо виклик методу `method()` з класу `C`:

```
Called method from class C
```

Найперше, де знадобиться описаний підхід – виклик конструкторів базових класів при множинному наслідуванні. Як ми пам'ятаємо конструктор базового

класу не викликається автоматично при заміщенні конструктора класу-нащадку – необхідно явно викликати конструктори усіх батьківських класів.

Лістинг 2.18. Виклик конструкторів з батьківських класів.

```
class A:
    def __init__(self, a):
        self.a = a

    def show(self):
        print("a =", self.a)

class B:
    def __init__(self, b):
        self.b = b

    def show(self):
        print("b =", self.b)

class C(A, B):
    def __init__(self, a, b, c):
        A.__init__(self, a)  # Виклик конструктора класу A
        B.__init__(self, b)  # Виклик конструктора класу B
        self.c = c

    def show(self):
        A.show(self)  # Виклик методу show класу A
        B.show(self)  # Виклик методу show класу B
        print("c =", self.c)

d = C(1, 2, 3)
d.show()
```

§3 СПЕЦІАЛЬНІ МЕТОДИ

3.1. Спеціальні поля та методи

У Python є група полів та методів, назва яких починається і закінчується подвійним підкресленням. З двома такими методами ми вже познайомилися – це конструктор та деструктор класу:

```
class Pet:
    def __init__(self, name):
        """ Конструктор
        ...
    def __del__(self):
        """ Деструктор
        ...
```

Такі методи та поля призначені для роботи з класами і екземплярами класів та дозволяють виконувати деякі, досить специфічні операції. Тому такі методи називають **спеціальними** або, на жаргоні програмістів, «**магічними**». Жаргонна назва пов'язана з тим, що виклик спеціальних методів, як правило, не здійснюється напряму, як виклик звичайних методів, а викликається вбудованими функціями або операторам. Наприклад, виклик конструктора здійснюється автоматично при створенні об'єкта, а деструктора – при його знищенні.

Як ми вже знаємо, всі класи Python неявно наслідуються від у суперкласу `object`, який розглядався у § 2. Саме у цьому класі оголошені та мають найпростіші реалізації усі спеціальні методи які будуть розглянуті у цьому параграфі.

Спеціальні поля

Крім спеціальних методів для класів та їхніх екземплярів автоматично створюються поля, що мають спочатку та в кінці по два символи нижнього підкреслення. Наприклад, раніше ми познайомилися з полем `__mro__` що містить послідовність класів при пошуку методів у класі та його нащадках.

У таблиці нижче наведено перелік деяких зі спеціальних полів.

Таблиця 3.1. Позначення для типів доступу

Спеціальне поле	Опис
`__bases__`	Містить список базових класів
`__dict__`	Містить словник атрибутів класу
`__doc__`	Містить текст документування класу
`__module__`	Містить ім'я модуля, у якому описано клас

`__mro__`	Містить ланцюг наслідування класу
`__name__`	Містить ім'я класу
`__qualname__`	Містить повне ім'я класу

Приклад 3.1. Розглянемо клас `Pet`, що описаний вище у лістингу 2.3 та його нащадок `Cat`, описаний у лістингу 2.2. Для класу та його екземпляру виведемо на екран значення кількох спеціальних полів

Лістинг 3.1.Спеціальні поля.

```
print("Ім'я класу     (__name__):  ", Cat.__name__)
print("Базовий клас   (__bases__): ", Cat.__bases__)
print("Атрибути класу (__dict__):  ", Cat.__dict__)
cat = Cat("Tom")    # Екземпляр класу
print("Атрибути екземпляру (__dict__): ", cat.__dict__)
```

Результатом роботи буде таке

```
Ім'я класу     (__name__):   Cat
Базовий клас   (__bases__):  (<class 'source.P_02.L3_Pet.Pet'>,)
Атрибути класу (__dict__):   {'__module__': '__main__', '__doc__': ' Клас
Cat - нащадок класу Pet ', 'voice': <function Cat.voice at 0x03C06B28>}
Атрибути екземпляру (__dict__):  {'_name': 'Tom'}
```

Як видно з результату роботи програми клас та його екземпляр мають різний перелік атрибутів. При цьому клас має перелік атрибутів, що не були явно оголошені у класі.

Спеціальні методи

Опис спеціальних методів нічим не відрізняється від опису звичайних методів. Єдиним виключенням є те, що програміст не може самовільно вигадувати імена для спеціальних методів – всі спеціальні методи мають чітко визначені імена на рівні мови програмування Python. А от виклик спеціальних методів, як уже було сказано раніше, має певну особливість – він здійснюється неявним чином інтерпретатором Python у певних ситуаціях, для яких призначені ці методи (наприклад інтерпретатор здійснює виклик метода `__init__()` в момент створення об'єкту).

Виклик спеціальних методів може здійснюватися і звичайним чином, аналогічно до виклику звичайних методів. Хоча інтерпретатор не забороняє таких дій, автор наполегливо рекомендує не застосовувати такий підхід у власних програмах – це може призвести до неочікуваного результату.

Метод __call__()

Якщо в класі перевизначено метод `__call__(self, args)`, то екземпляр класу можна викликати як функцію. Про такий об'єкт кажуть, що він є **функтором**.

Отже, якщо клас має метод `__call__(self, args)`, де `args` – перелік його формальних параметрів, то цей метод можна викликати з екземпляру `obj` таким чином

```
obj(args)
```

де `args` – список фактичних аргументів

Розглянемо приклад.

Приклад 3.2. Опишемо клас `Polynom`, що моделює роботу з поліномом. Поліном

$$P(x) = a_n x^n + \cdots + a_1 x + a_0$$

у класі будемо зберігати у вигляді словника, у якому зберігаються пари

```
{i : a_i}
```

де, ключами є степінь змінної x у відповідному доданку $a_i x^i$, а значенням –коефіцієнт a_i.

Лістинг 3.2. Використання магічного метода `__call__()`.

```
class Polynom:
    """ Клас для моделювання роботи з поліномами"""

    def __init__(self):
        """ Конструктор """
        self._coeffs = {} # словник коефіцієнтів

    def set(self, power, coef):
        """ Встановлює коефіцієнт при відповідному степені
        :param power: степінь
        :param coef:  коефіцієнт
        """
        self._coeffs[power] = coef

    def __call__(self, x):
        """ Магічний метод - обчислює значення полінома
        :param x: значення незалежної змінної
        :return: значення полінома у точці x
        """
        res = 0
```

```
        for i, a in self._coeffs.items():
            res += a * x ** i
        return res

p = Polynom()
# Задамо поліном p = x^2 + 2x + 1
p.set(0, 1)
p.set(1, 2)
p.set(2, 1)

x = float(input("x="))

f = p(x)  # виклик магічного метода __call__(self, x)

print("P(%f)=%f" % (x, f)) # Виведення результату
```

Як було зазначено вище, виклик метода `__call__(self, args)` можна здійснити аналогічно до звичайних методів. Тоді у програмі передостанній рядок матиме такий вигляд:

```
f = p.__call__(x)  # виклик спеціального метода як звичайного метода
```

Результатом роботи програми для введеного з клавіатури значення 2 буде таким:

```
x=2
p(2.000000)=9.000000
```

Метод визначення модуля

Якщо у класі визначено метод `__abs__(self)`, то для екземплярів класу можна застосовувати функцію **`abs`**`()`.

Приклад 3.3. Нехай `Vector2D` – клас, що моделює роботу з двовимірним вектором. Клас буде мати два поля – координати x та y вектора.

Визначимо у цьому класі спеціальний метод `__abs__(self)`, що повертає довжину цього вектора.

Лістинг 3.3. Клас `Vector2D`. Визначення довижини вектора.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x  # Координата x вектора
        self.y = y  # Координата y вектора
```

```
    def __abs__(self):
        """ Визначає довжину вектора
        :return: довжину вектора
        """
        return (self.x ** 2.0 + self.y ** 2.0) ** 0.5

if __name__ == "__main__":
    v = Vector2D(3, 4)
    print(abs(v))
```

Методи перетворення типів

У Python існує низка спеціальних методів, що призначені для перетворення об'єкту до базових типів.

Таблиця 3.2. Перетворення об'єктів до базових типів

Спеціальний метод	Опис
`__bool__(self)`	Метод для приведення екземпляру до логічного типу (тип `bool`). Викликається при використанні функції **`bool`**`()` або у випадках коли відбувається автоматичне приведення до логічного типу
`__complex__(self)`	Метод для приведення екземпляру до комплексного типу. Викликається при використанні функції **`complex`**`()`
`__float__(self)`	Метод для приведення екземпляру до дійсного типу. Викликається при використанні функції **`float`**`()`
`__int__(self)`	Метод для приведення екземпляру до цілого типу. Викликається при використанні функції **`int`**`()`
`__str__(self)`	Метод для приведення екземпляру до рядкового типу. Викликається при використанні функції **`str`**`()`

Найчастіше у класах оголошується метод `__str__(self)`, оскільки він дозволяє виводити на екран за допомогою функції **`print`**`()` інформацію про об'єкт. Наведемо такий приклад

Приклад 3.4. Опишемо клас `RectTriangle`, що моделює роботу з прямокутним трикутником. Опишемо метод `__str__(self)`, який буде повертати інформацію про трикутник.

Лістинг 3.4. Використання магічного метода `__str__()`.

```
class RectTriangle:
    """ Клас прямокутний трикутник """
```

```
    def __init__(self, a, b):
        self._a = a  # поле _a - перший катет
        self._b = b  # поле _b - другий катет

    def __str__(self):
        return ("Прямокутний трикутник з катетами %f та %f"
                % (self._a, self._b))

t = RectTriangle(3, 4)
print(t)  # Фактично тут замість t викликається метод t.__str__()
```

У результаті роботи програми на екрані з’явиться повідомлення

```
Прямокутний трикутник з катетами 3.000000 та 4.000000
```

Спеціальні методи для колекцій

У Python передбачено низку спеціальних методів, що дозволяють спросити синтаксис використання класів-колекцій, роблячи їх подібною до роботи зі списками чи словниками.

Таблиця 3.3. Спеціальні методи для колекцій

Спеціальний метод	Опис
`__len__(self)`	Повертає довжину об’єкта, тобто кількість елементів у контейнері. Використовується під час виклику функції `len()`
`__getitem__(self, key)`	Визначає поведінку при доступі до елемента за ключем, тобто коли використовується `self[key]`
`__setitem__(self, key, value)`	Визначає поведінку при присвоєнні значення елементу за ключем, тобто при `self[key]=value`
`__delitem__(self, key)`	Визначає поведінку при видаленні елемента за ключем, тобто при `del self[key]`
`__contains__(self, item)`	Призначений для перевірки приналежності елемента до колекції під час використання оператора `in`, тобто item `in self`
`__iter__(self)`	Повертає ітератор для колекції, наприклад, у випадку перебору елементів колекції оператором `for x in collection:` Вимагає опис методу `__next__(self)`.
`__next__(self)`	Реалізує ітеративний протокол послідовного доступу до елементів колекції. Виконується автоматично у циклі `for` та при використанні функції `next(self)`

`__reversed__(self)`	Визначає поведінку вбудованої функції **`reversed()`**.

Методи `__iter__(self)` та `__next__(self)` будуть детально розглянуті у цьому параграфі нижче у пункті «Ітератори та генератори». Розглянемо приклад, що демонструє роботу інших, наведених у цьому пункті спеціальних методів.

Приклад 3.5. Опишемо клас `Collection`, що є списком з захищеними від виключних ситуацій операціями додавання, видаленням елементів. При цьому, щоб спростити розуміння прикладу, розіб'ємо код на кілька частин. У першій частині лістингу наведемо опис методів додавання (зміни) елементів до колекції та метод перетворення колекції у рядок, для виведення її на екран.

Лістинг 3.5. Використання спеціальних методів для колекцій.

```
class Collection:
    """ Клас захищений список"""
    def __init__(self):
        self.__elements = []  # список елементів колекції

    def __setitem__(self, key, value):
        """ Встановлює значення елемента колекції
        Магічний метод - викликається при використанні
        оператора [] для присвоєння: self[key] = value
        :param key: індекс
        :param value: значення
        """
        try:
            # якщо ключ існує, змінюємо значення за ключем
            self.__elements[key] = value
        except IndexError:  # якщо ключа не існує
            #  додаємо елемент до колекції
            self.__elements.append(value)

    def __str__(self):
        """ Магічний метод перетворення об'єкту у рядок
        :return: Рядкове зображення об'єкта
        """
        return str(self.__elements)
```

Для демонстрації роботи класу, створимо його екземпляр, додамо кілька елементів у нього та виведемо колекцію на екран.

Лістинг 3.5. Продовження. Використання спеціальних методів для колекцій.

```
c = Collection()

print("Колекція :", c)
c[0] = 0  # c.__setitem__(0) - додаємо 0-й елемент
print("Колекція :", c)
c[1] = 1  # c.__setitem__(1) - додаємо 1-й елемент
print("Колекція :", c)
c[5] = 5  # c.__setitem__(5) - додаємо 2-й! елемент
print("Колекція :", c)
c[2] = 2  # c.__setitem__(2) - змінюємо 2-й елемент
print("Колекція :", c)
```

Результат роботи коду наведено нижче

```
Колекція : []
Колекція : [0]
Колекція : [0, 1]
Колекція : [0, 1, 5]
Колекція : [0, 1, 2]
```

Продовжимо опис класу і тепер оголосимо метод, що повертає з колекції елемент за заданим ключем та метод, що перевіряє чи входить заданий елемент у колекцію

Лістинг 3.5. Продовження. Використання спеціальних методів для колекцій.

```
class Collection:
    ...
    def __getitem__(self, key):
        """ Повертає значення елемента колекції
        Магічний метод - викликається при використанні
        оператора [] для читання: self[key]
        :param key: індекс
        :return: значення елемента колекції
        """
        try:
            return self.__elements[key]
        except IndexError:       # якщо ключа не існує
            return None          # None

    def __contains__(self, item):
        """ Перевіряє чи входить елемент item у колекцію
        Магічний метод - виклик здійснюється під час виклику
```

```
        оператора in:  item in self
        :param item: шуканий елемент
        :return: True, якщо item міститься у колекції
        """
        return item in self.__elements
```

Будемо вважати, що програма продовжує роботу з попереднього стану, тобто колекція містить елементи `[0, 1, 2]`.

Лістинг 3.5. Продовження. Використання спеціальних методів для колекцій.

```
# c = [0, 1, 2]
print(c[1])   # c.__getitem__(1)
print(c[100]) # c.__getitem__(100)
if 2 in c:    # c.__contains__(2):
    print("Елемент 2 входить до колекції!")
```

Результатом такого фрагменту коду буде

```
1
None
Елемент 2 входить до колекції!
```

На завершення демонстрації прикладу опишемо метод, що видаляє елемент з колекції та метод, що повертає кількість елементів у колекції.

Лістинг 3.5. Продовження. Використання спеціальних методів для колекцій.

```
class Collection:
    ...
    def __delitem__(self, key):
        """ Видаляє елемент з колекції
        Магічний метод - викликається при використанні
        оператора del: del self[key]
        :param key: індекс елемента, що видаляється
        """
        try:
            self.__elements.pop(key)
        except IndexError:
            pass

    def __len__(self):
        """ Повертає кількість елементів, у колекції
```

```
        Магічний метод, викликається під час виклику функції len:
        len(self)
        :return: кількість елементів у колекції
        """
        return len(self.__elements)
```

Як і при демонстрації роботи попереднього фрагменту класу, будемо вважати, що програма продовжує своє виконання з попереднього стану.

Лістинг 3.5. Продовження. Використання спеціальних методів для колекцій.

```
# c = [0, 1, 2]
print("Кількість елементів =", len(c))  # c. __len__()
del c[0]   # c.__delitem__(0) - видаляємо 0-й елемент
print("Колекція :", c)
print("Колекція :", c)
print("Кількість елементів =", len(c))  # c. __len__()
del c[100] # c.__delitem__(100)
print("Колекція :", c)
```

Результатом цього фрагменту коду буде

```
Кількість елементів = 3
Колекція : [1, 2]
Кількість елементів = 2
Колекція : [1, 2]
```

3.2. Перевантаження операторів

Перш ніж перейти до пояснення матеріалу цього пункту, розглянемо клас `Vector2D`, визначений у прикладі 3.3 (лістинг 3.3). У випадку, якщо потрібно додати два вектори

Лістинг 3.6. Додавання об’єктів класу `Vector2D`.

```
v1 = Vector2D(1, 3)
v2 = Vector2D(4, 2)
```

що є екземплярами цього класу, потрібно буде кожного разу додатково описати фрагмент коду на кшталт такого:

Лістинг 3.6. Продовження. Додавання об'єктів класу `Vector2D`.

```
v3 = Vector2D(v1.x + v2.x, v1.y + v2.y)
```

Чим складнішим буде структура класу, тим складнішим буде код такої операції. Це, у свою чергу буде знижувати його читабельність і, відповідно, збільшувати ймовірність помилки. Хотілося б, сховати реалізацію такої операції у реалізації самого класу, а її виклик здійснювати аналогічно до того, як це здійснюється у математиці, наприклад:

```
v3 = v1 + v2
```

Саме для цього у програмуванні використовується перевантаження операторів, яке дозволяє використовувати звичайні оператори такі як «+» чи «–» для екземплярів класів. Це у свою чергу спрощує написання нового та розуміння існуючого коду.

Щоб перевантажити оператор, потрібно в класі описати спеціальний (магічний) метод з відповідним іменем.

Під час виклику перевантаженого оператора, викликається відповідний метод для екземпляру класу, що є лівим операндом оператора.

Перевантаження бінарних арифметичних операторів

У таблиці нижче наведені доступні для перевантаження бінарні арифметичні операції та ім'я методу, що має бути описаний у класі для їхнього перевантаження. За домовленістю кожен з наведених нижче спеціальних методів повинен повертати результат, що часто є екземпляром деякого класу, при цьому сама операція не повинна впливати на жоден з екземплярів класів, що є лівим та правим її операндом.

Таблиця 3.4. Методи перевантаження арифметичних операторів

Метод	Оператор	Приклад виклику	Еквівалент виклику
`__add__(self, other)`	Додавання	`x + y`	`x.__add__(y)`
`__sub__(self, other)`	Віднімання	`x - y`	`x.__sub__(y)`
`__mul__(self, other)`	Множення	`x * y`	`x.__mul__(y)`
`__truediv__(self, other)`	Ділення	`x / y`	`x.__truediv__(y)`
`__floordiv__(self, other)`	Цілочислове ділення	`x // y`	`x.__floordiv__(y)`
`__mod__(self, other)`	Остача від ділення	`x / y`	`x.__mod__(y)`

`__divmod__(self, other)`	Частка та остача	`divmod(x,y)`	`x.__divmod__(y)`
`__pow__(self, other)`	Піднесення до степеня	`x ** y`	`x.__pow__(y)`

Приклад 3.6. Перевантажимо операції бінарні оператори додавання, віднімання та множення для вищеописаного класу `Vector2D`. Для цього необхідно у описі класу описати методи `__add__()`, `__sub__()`, `__mul__.()`. При цьому, результатом перших двох методів буде новий вектор, утворений по-членним додаванням/відніманням відповідних компонент векторів, а результатом останнього буде їхній скалярний добуток. Крім цього, для зручного виведення вектора на екран, опишемо спеціальний метод `__str__(self)`. Також, перепишемо спеціальний метод знаходження модуля, використовуючи перевантажений оператор множення.

Лістинг 3.7. Клас `Vector2D` з перевантаженими арифметичними операторами.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x  # Координата x вектора
        self.y = y  # Координата y вектора

    def __str__(self):
        return "(%f, %f)" % (self.x, self.y)

    def __add__(self, other):
        """ Оператор +
        :param other: Правий операнд
        :return: Результат операції  self + other
        """
        return Vector2D(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """ Оператор -
        :param other:  Правий операнд
        :return: Результат операції self - other
        """
        return Vector2D(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        """ Оператор *
        :param other: Правий операнд
        :return: Скалярний добуток векторів self і other
        """
        return self.x * other.x + self.y * other.y
```

```
    def __abs__(self):
        """ Визначає довжину вектора, використовуючи
         перевантажений оператор множення
        :return: довжину вектора
        """
        return (self * self) ** 0.5  # self.__mul__(self)

if __name__ == "__main__":
    v1 = Vector2D(1, 3)
    v2 = Vector2D(4, 2)

    v3 = v1 + v2  # v3 = v1.__add__(v2)
    v4 = v1 - v2  # v4 = v1.__sub__(v2)
    a = v1 * v2   # a = v1.__mul__(v2)

    print(v3)
    print(v4)
    print(a)
```

У результаті виконання вищенаведеного коду на екран буде виведено

```
(5.000000, 5.000000)
(-3.000000, 1.000000)
10
```

Складені оператори присвоєння

Наведені нижче спеціальні методи застосовуються для перевантаження бінарних арифметичних операцій, що впливають на самі об’єкти.

Таблиця 3.5. Перевантаження складених операторів присвоєння

Метод	Оператор	Приклад виклику	Еквівалент виклику
`__iadd__(self, other)`	`+=`	`x += y`	`x.__iadd__(y)`
`__isub__(self, other)`	`-=`	`x -= y`	`x.__isub__(y)`
`__imul__(self, other)`	`*=`	`x *= y`	`x.__imul__(y)`
`__itruediv__(self, other)`	`/=`	`x /= y`	`x.__itruediv__(y)`
`__ifloordiv__(self, other)`	`//=`	`x //= y`	`x.__ifloordiv__(y)`
`__imod__(self, other)`	`%=`	`x %= y`	`x.__imod__(y)`
`__ipow__(self, other)`	`**=`	`x **= y`	`x.__ipow__(y)`

За домовленістю кожен з наведених нижче спеціальних методів, що перевантажують оператор, змінює лівий операнд, тобто діє на екземпляр з якого викликається. Результатом операції має бути посилання на лівий операнд (після зміни), тобто `self`. Це необхідно для того, щоб були коректними операції на кшталт

```
z = x += y
```

Приклад 3.7. Доповнимо клас `Vector2D` методом, що перевантажує оператор +=.

Лістинг 3.8. Клас `Vector2D` з перевантаженим оператором +=

```
class Vector2D:
    ...
    def __iadd__(self, other):
        """ Оператор +=
        :param other: Правий операнд
        :return: self
        """
        self.x += other.x
        self.y += other.y
        return self

if __name__ == "__main__":
    v1 = Vector2D(1, 1)
    v2 = Vector2D(2, 3)
    v2 += v1    # v2.__iadd__(v1)
    print(v2)
```

Результатом виконання цього коду буде

```
(3.000000, 4.000000)
```

Оператори порівняння

Розглянемо спеціальні методи, що дозволяють перевантажити оператори порівняння.

Таблиця 3.6. Перевантаження операторів порівняння

Метод	Оператор	Приклад виклику	Еквівалент виклику
`__lt__(self, other)`	<	`x < y`	`x.__lt__(y)`
`__le__(self, other)`	=<	`x <= y`	`x.__le__(y)`

`__eq__(self, other)`	`==`	`x == y`	`x.__eq__(y)`
`__ne__(self, other)`	`!=`	`x != y`	`x.__ne__(y)`
`__gt__(self, other)`	`>`	`x > y`	`x.__gt__(y)`
`__ge__(self, other)`	`>=`	`x >= y`	`x.__ge__(y)`

За домовленістю, кожен з вищенаведених методів, що перевантажує відповідний оператор, має обов'язково повертати булеве значення `True` або `False`.

Приклад 3.8. Доповнимо вищенаведений клас `Vector2D` методами, що перевантажують оператори порівняння `==` та `!=`.

Лістинг 3.9. Клас `Vector2D` з перевантаженим оператором `==` та `!=`.

```
class Vector2D:
    ...
    def __eq__(self, other):
        """ Оператор ==
        :param other: Правий операнд
        :return: True, якщо self == other
        """
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        """ Оператор !=
        :param other: Правий операнд
        :return: True, якщо self != other
        """
        return not self.__eq__(other)

if __name__ == "__main__":
    v1 = Vector2D(1, 1)
    v2 = Vector2D(2, 3)

    print(v1 == v2)
    print(v1 != v2)
```

Унарні оператори

Таблиця 3.7. Перевантаження унарних операторів

Метод	Оператор	Приклад виклику	Еквівалент виклику
`__pos__(self)`	унарний плюс «+»	`+x`	`x.__pos__()`
`__neg__(self)`	унарний мінус «-»	`-x`	`x.__neg__()`

Приклад 3.9. Доповнимо клас `Vector2D` спеціальним методом, що перевантажує оператор унарний мінус.

Лістинг 3.10. Клас `Vector2D` з перевантаженим оператором `==` та `!=`.

```
class Vector2D:
    ...
    def __neg__(self):
        """ Оператор унарний мінус """
        return Vector2D(-self.x, -self.y)

if __name__ == "__main__":
    a = Vector2D(3, 4)
    print(-a)
```

Результатом вищенаведеної програми буде

```
(-3.000000, -4.000000)
```

3.3. Ітератори та генератори

Основним призначенням таких інструментів, як ітератори та генератори є спрощення обробки великих наборів даних, послідовностей, колекцій тощо.

Ітератори

Основне призначення ітераторів – це спрощення навігації по елементах колекцій. Вони забезпечують уніфікований інтерфейс для обходу елементів, не залежно від типу колекції.

Коли мова йде про загальну концепцію ітераторів, то розрізняють два типи шаблонів – ітератор і ітерований об’єкт, що завжди реалізуються у парі. Такий підхід забезпечує підтримку кількох активних паралельних обходів однієї колекції.

Означення 3.1.**Ітератором** називається інтерфейс (спеціальний об'єкт), що надає послідовний доступ до елементів колекції та навігацію по ним, не розкриваючи внутрішньої структури самої колекції. Колекція, що надає можливість послідовного доступу до своїх елементів ітератору, називається **ітерованим об'єктом**.

Іншими словами, ітератором є об'єкт, що наділений здатністю звертатися до елементів колекції, відстежуючи при цьому свою поточну позицію у ній.

Застосування концепції ітераторів дозволяє розділити відповідальність обробки елементів колекції між клієнтом (за допомогою ітератора) та колекцією (ітерованим об'єктом): клієнти отримують можливість універсальним чином працювати з різними колекціями, а колекції стають простішими, оскільки делегують перебір своїх елементів іншій сутності.

Найуживаніше місце використання ітераторів – цикл `for`. Коли відбувається перебір елементів деякої колекції `collection` циклом `for`

```
for element in collection:
    print(element)
```

то, фактично, при кожній ітерації циклу відбувається звернення до ітератора, що пов'язаний з колекцією `collection` (яка є ітерованим об'єктом), з вимогою надати черговий елемент (змінна `element` послідовно набуває значень, що повертає їй ітератор). Якщо елементів у колекції більше немає, то ітератор генерує виключення, що обробляється циклом `for` непомітно для користувача.

Раніше ми вже неявно користувалися ітераторами для базових колекцій Python, таких як списки, словники, рядки, тощо. У цьому пункті вивчимо спосіб створення ітераторів та механізми доступу до елементів колекцій користуючись ними.

Щоб створити клас-ітератор екземпляри якого будуть ітераторами для деякої колекції, потрібно

- передати екземпляру класу посилання на колекцію, для якої цей об'єкт буде ітератором (частіше за все як параметр конструктора);
- описати спеціальний метод `__next__()`, у якому визначити спосіб послідовного обходу елементів колекції.

Опис методу `__next__()` має одну особливість – він повинен породжувати виключення `StopIteration`, якщо у колекції не залишилося не опрацьованих елементів. Крім цього, будемо дотримуватися загальноприйнятої домовленості, що цей метод має повертати поточний елемент колекції. Спеціальний метод ітератора `__next__()` викликається неявно кожного разу під час виклику вбудованої функції `next()`, аргументом якої є об'єкт ітератор.

```
x = next(it)  # виклик методу it.__next__() для ітератора it
```

Для прикладу опишемо клас `Iterator`, що є ітератором для колекції, що агрегує список деяких елементів.

Лістинг 3.11. Клас `Iterator`

```
class Iterator:
    """ Клас Ітератор """

    def __init__(self, collection):
        """ Конструктор ітератора
        :param collection: посилання на колекцію
        """
        self._collection = collection
        self._cursor = 0  # поточна позиція ітератора у колекції

    def __next__(self):
        try:
            value = self._collection[self._cursor]
            self._cursor += 1
            return value
        except IndexError:
            raise StopIteration
```

У свою чергу, щоб перетворити клас-колекцію у ітерований об’єкт, у ній повинен бути описаний спеціальний метод `__iter__()`, що повертає екземпляр класу-ітератора. У такому разі кажуть, що клас підтримує ітераційний протокол. Спеціальний метод ітератора `__iter__()` викликається неявно кожного разу під час виклику вбудованої функції **`iter`**`()`, аргументом якої є екземпляр колекції.

```
it = iter(c)   # виклик спеціального методу c.__iter__(),
               # що повертає ітератор it.
```

Опишемо клас `Iterable`, що підтримує ітераційний протокол та є колекцією, що містить список елементів.

Лістинг 3.12. Клас `Iterable`

```
class Iterable:
    """ Клас Ітерований об'єкт """
```

```
    def __init__(self):
        self.__container = []  # Список елементів колекції
```

```
    def append(self, value):
        self.__container.append(value)

    def __getitem__(self, item):
        assert isinstance(item, int)
        assert item >= 0
        return self.__container[item]

    def __iter__(self):
        """ Спеціальний метод, що повертає ітератор для колекції
        :return: Ітератор колекції
        """
        return Iterator(self.__container)
```

Для демонстрації роботи цього класу створимо екземпляр класу `Iterable`, додамо у нього кілька довільних елементів та скористаємося ітераторами для їхнього перебору.

Лістинг 3.12. Продовження. Клас `Iterable`.

```
c = Iterable()
c.append(1)
c.append(2)
c.append(3)
c.append(4)

# Неявне створення ітератора для колекції та виклику функції next()
for i in c:
    print(i)
```

Цикл `for` використовує ітератор колекції, неявно викликаючи функції `iter()` та `next()`. Такий підхід є найбільш рекомендованим для роботи з колекціями, що підтримують ітераційний протокол. Результат виконання вищенаведеного коду наведено нижче

```
1
2
3
```

```
4
```

Як бачимо, цикл повністю перебрав усі елементи у колекції.

Тепер явно створимо об'єкт-ітератор для цієї ж колекції та здійснимо перебір її елементів за допомогою функції **`next()`**.

Лістинг 3.12. Продовження. Клас `Iterable`. Явне створення ітератора.

```
# Явне створення ітератора
it = iter(c)     # виклик спеціального методу __iter__()
print(next(it))  # явний виклик методу __next__()
print(next(it))  # явний виклик методу __next__()
print(next(it))  # явний виклик методу __next__()
print(next(it))  # явний виклик методу __next__()
```

Результат виконання цього коду буде повністю повторювати відповідний результат для циклу **`for`**. Проте, зауважимо, що при явному створенні ітератора та використанні функції **`next()`**, відповідальність за обробку виключення, що породжує ітератор коли у колекції перебрані усі елементи, повністю покладається на програміста. Для щойно наведеного коду, наступний виклик функції **`next()`** породив би виключення. Тому, рекомендується при явному використанні ітератора виклик функції **`next()`** розташовувати у блоці **`try...except`**. Наприклад, для того, щоб перебрати усі елементи колекції можна скористатися таким кодом, результат роботи якого еквівалентний вищенаведеному застосуванню циклу **`for`**.

Лістинг 3.13. Перебір усіх елементів колекції за допомогою ітератора

```
# Явне створення ітератора
it = iter(c)  # виклик спеціального методу __iter__()
while True:
    try:
        val = next(it) # Явно викликається функція next(it)
        print(val)
    except StopIteration: # якщо елементи скінчилися
        break             # завершуємо цикл
```

Зауважимо, що якщо клас містить колекцію (наприклад, список чи кортеж), то можна використати вбудований ітератор цієї колекції. Наприклад, попередній клас `Iterable` містив колекцію-список, що має вбудований ітератор. Отже, у класі `Iterable` можемо переписати метод `__iter__(self)`, який буде повертати ітератор списку.

Лістинг 3.14.

```
def __iter__(self):
    """ Спеціальний метод, що повертає ітератор для колекції
    :return: вбудований ітератор списку
    """
    return iter(self.__container)  # вбудований ітератор списку
```

Розглянута концепція, що побудована на взаємодії екземплярів класів Ітератор-Ітерований дозволяє не лише здійснити перебір усіх елементів колекції, але й здійснювати кілька одночасних обходів елементів однієї колекції, не заважаючи один одному. Це демонструє приклад наведений нижче

Лістинг 3.15. Два паралельних обходи однієї колекції.

```
# c = [1, 2, 3, 4]
# Створення першого ітератора для колекції c
it1 = iter(c)
# Створення другого ітератора для колекції c
it2 = iter(c)

print(next(it1))  # наступний елемент для ітератора it1
print(next(it2))  # наступний елемент для ітератора it2
print(next(it1))  # наступний елемент для ітератора it1
print(next(it2))  # наступний елемент для ітератора it2
```

Результат роботи цього коду

```
1
1
2
2
```

Генератори

Об’єкти генератори, подібні ітераторів, тільки на відміну від останніх вони повертають не елементи колекції, а щойно згенеровані об’єкти. При цьому генератор не будує всю послідовність одразу – він послідовно конструює її члени по-одному при кожному зверненні. Ця особливість дуже корисна, наприклад, при обробці великого масиву даних, оскільки не потрібно завантажувати (чи обчислювати) весь масив даних, що економить час та оперативну пам’ять.

У об'єкті-генераторі одночасно визначені і спеціальний метод `__next__()` і `__iter__()`, що робить його одночасно і ітератором і ітерованим. Отже, формально генератор є ітератором, проте концептуально між ними існує значна різниця: ітератор – це механізм по-елементного обходу даних (що містяться у колекції), а генератор дозволяє відкладено створювати результат під час кожної наступної ітерації (за допомогою описаного в генераторі алгоритму).

Приклад 3.10. Використовуючи ітераційний протокол опишемо генератор, що генерує послідовність факторіалів натуральних чисел, що не перевищують деякого заданого числа n.

Звернемо увагу читача та той факт, що оскільки генератор з формальної точки зору є одночасно і ітератором і ітерованим, що його спеціальний метод `__iter__()`, має повертати посилання на поточний екземпляр класу (тобто `self`).

Лістинг 3.16. Генератор для обчислення $n!$.

```
class FactorialGenerator:
    """ Клас генератор факторіалів натуральни чисел
        1, 1, 2, 6, 24, ...     """

    def __init__(self, n):
        """ Конструктор генератора
        :param n: Номер найбільшого елементу послідовності
        """
        self._n = n  # Номер найбільшого члена послідовності
        self._k = 0  # Номер поточного члена послідовності
        self._f = 1  # Поточний член послідовності

    def __iter__(self):
        """ Спеціальний метод, що повертає ітератор
        :return: посилання на себе """
        return self

    def __next__(self):
        if self._k == 0:
            self._k = 1
            return 1
        elif self._k <= self._n:
            self._f *= self._k
            self._k += 1
            return self._f
        else:
            raise StopIteration
```

Використання генератора, як і ітератора, може здійснюватися як за допомогою циклу **for**

Лістинг 3.16. Продовження. Використання генератора у циклі **for**.

```
for f in FactorialGenerator(5):
    print(f)
```

так і явним створенням генератора методом **iter**(), з подальшим отриманням елементів послідовності методом **next**()

Лістинг 3.16. Продовження. Явне використання генератора.

```
f = FactorialGenerator(5)
print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
```

Результат виконання обох фрагментів коду буде однаковим і наведений нижче

```
1
1
2
6
24
120
```

Функція-генератор

Створення та використання генераторів у Python це частий процес. Тому, для спрощення їхнього опису у Python є спеціальна конструкція, що називається

Означення 3.2. Функція-генератор – спеціальний об’єкт Python, що будує послідовність елементів деякої послідовності з по-елементним доступом до її членів.

Надалі, під терміном генератор, будемо розуміти саме функцію-генератор. Її опис подібний до синтаксису звичайної функції, за виключенням того, що для повернення результату (тобто поточного члена послідовності) замість оператора

```
return result
```

використовується оператор

```
yield result
```

Оператор **yield** (на відміну від **return**) не лише повертає деяке значення, але й запам'ятовує стан функції (місце завершення, значення всіх локальних змінних). Під час наступного виклику генератор-функції її виконання починається з наступного після **yield** оператора. Таким чином, виконання програми перемикається від програми до генератор-функції і назад.

Приклад 3.11. Опишемо генератор наведений у прикладі 3.10 використовуючи синтаксис функції-генератора

Лістинг 3.17. Функція-генератор для обчислення $n!$.

```
def FactorialGenerator(n):
    yield 1  # 0! = 1
    f = 1    # поточний член послідовності
    for k in range(1, n+1):
        f *= k
        yield f
```

Виклик та використання такого генератор дослівно повторює використання генератора описаного у лістингу 3.16.

Завершення роботи генератора неявно завершується породженням виключення `StopIteration`, що дозволяє використовувати генератор у циклі **for** без небезпеки зациклення програми.

Оскільки генератор не будує одразу всієї послідовності, то можна описувати генератори для визначення як завгодно великих членів необмежених послідовностей.

Приклад 3.12. Опишемо генератор що будує послідовність чисел Фібоначчі.

Лістинг 3.18. Генератор для обчислення чисел Фібоначчі.

```
def Fibonacci():
    F2 = F1 = 1
    yield F2
    yield F1
    while True: # нескінченний цикл
```

```
        F2, F1 = F1, F1 + F2
        yield F1
```

Оскільки генератор містить нескінченний цикл, то він буде обчислювати члени послідовності нескінченно. Тому необхідно передбачити у головній програмі завершення його роботи.

Лістинг 3.18. Продовження. Використання "нескінченного" генератора.

```
n = int(input("n = "))
i = 0
for f in Fibonacci():
    print(f)
    i += 1
    if i > n:
        break
```

3.4. Рекурентні співвідношення

Рекурентне співвідношення першого порядку

Нехай $\{a_n : n \geq 0\}$ деяка послідовність дійсних чисел.

Означення 3.3. Послідовність $\{a_n : n \geq 0\}$ називається заданою **рекурентним співвідношенням першого порядку**, якщо явно задано її перший член a_0, а кожен наступний член a_n цієї послідовності визначається деякою залежністю через її попередній член a_{n-1}, тобто

$$\begin{cases} a_0 = u \\ a_n = f(n, p, a_{n-1}),\ n \geq 1 \end{cases}$$

де u задане (початкове) числове значення, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування (зокрема у нашому випадку Python).

Приклад 3.13. Розглянемо послідовність $\{a_n = n! : n \geq 0\}$. Її можна задати рекурентним співвідношенням першого порядку. Дійсно, враховуючи означення факторіалу отримаємо

$$\begin{cases} a_0 = 1 \\ a_n = n a_{n-1},\ n \geq 1. \end{cases}$$

Маючи рекурентне співвідношення можна знайти який завгодно член послідовності. Наприклад, якщо потрібно знайти a_5, то використовуючи рекурентні формули, послідовно від першого члена отримаємо

$$\begin{gathered} a_0 = 1 \\ a_1 = 1 \cdot a_0 = 1 \cdot 1 = 1 \\ a_2 = 2 \cdot a_1 = 2 \cdot 1 = 2 \\ a_3 = 3 \cdot a_2 = 3 \cdot 2 = 6 \\ a_4 = 4 \cdot a_3 = 4 \cdot 6 = 24 \\ a_5 = 5 \cdot a_4 = 5 \cdot 24 = 120 \end{gathered}$$

З точки зору програмування, послідовність задана рекурентним співвідношенням значно зручніша, ніж задана у явному вигляді. Для обчислення членів послідовностей, заданих рекурентними співвідношеннями, використовують цикли.

Нехай послідовність a_n задана рекурентним співвідношенням

$$\begin{cases} a_0 = u \\ a_n = f(n, p, a_{n-1}),\ n \geq 1 \end{cases}$$

Тоді, після виконання коду

```
a = u
for n in range(1, N +1):
    a = f(n, p, a)
```

у змінній a буде міститися значення елемента a_N послідовності.

Вправа 3.1. Доведіть вищенаведене твердження використовуючи метод математичної індукції.

Приклад 3.14. Для введеного з клавіатури значення N обчислимо $N!$

Як було зазначено раніше послідовність $a_n = n!$ може бути задана рекурентним співвідношенням

$$\begin{cases} a_0 = 1 \\ a_n = na_{n-1},\ n \geq 1 \end{cases}$$

Тоді, згідно з вищенаведеним алгоритмом, отримаємо

Лістинг 3.19. Використання рекурентного співвідношення для знаходження $n!$

```
N = int(input("N = "))
a = 1                          # a = u
for n in range(1, N+1):
    a = n * a                  # a = f(n, p, a)
print ("%d! = %d" % (N, a))  # виводимо на екран результат
```

Результатом вищенаведеного коду для введеного з клавіатури числа 5 буде

```
N = 5
5! = 120
```

Приклад 3.15. Складемо програму для обчислення елементів послідовності, заданої у явному вигляді співвідношенням

$$a_n = \frac{x^n}{n!},\ n \geq 0.$$

Складемо рекурентне співвідношення для заданої послідовності. Легко бачити, що кожен член послідовності a_n є добутком чисел. Враховуючи це, обчислимо частку двох сусідніх членів послідовності. Для $n \geq 1$ отримаємо

$$\frac{a_n}{a_{n-1}} = \frac{x^n}{n!} \cdot \frac{(n-1)!}{x^{n-1}} = \frac{x}{n}.$$

Звідки випливає, що для $n \geq 1$

$$a_n = \frac{x}{n} a_{n-1}$$

Отже ми отримали для послідовності a_n рекурентну формулу, у якій кожен член послідовності для всіх $n \geq 1$ визначається через попередній член a_{n-1}. Щоб задати рекурентне співвідношення, залишилося задати перший член a_0. Для цього просто підставимо 0 у вихідну формулу

$$a_0 = \frac{x^0}{0!} = 1.$$

Отже остаточно отримаємо рекурентне співвідношення першого порядку

$$\begin{cases} a_0 = 1 \\ a_n = \frac{x}{n} a_{n-1},\ n \geq 1. \end{cases}$$

Оформимо алгоритм знаходження членів послідовності у вигляді підпрограми. Тоді, згідно з вищенаведеним алгоритмом, отримаємо програму

Лістинг 3.20. Використання рекурентних співвідношень.

```
def rec(N, x):
""" Знаходження елементів послідовності використовуючи
    рекурентні співвідношення
:param N: Номер члена послідовності, що необхідно знайти
```

```
:param x: Параметр
:return:  Знайдений член послідовності.
    a = 1                # початкове значення
    for n in range(1, N + 1):
        a = x / n * a  # можна так: a *= x / n
    return a  # Повертаємо знайдений член послідовності

a = rec(5, 1)    # Обчислюємо 5-й член послідовності
print("a =", a)  # виводимо на екран результат
```

Результат роботи вищенаведеної програми для параметрів `N = 5` і `x = 1` буде

```
a = 0.008333333333333333
```

Зауважимо, що нумерація членів послідовності інколи починається не з 0, а з деякого натурального числа m, тобто $\{a_n : n \geq \mathrm{m}\}$. Припустимо, що рекурентне співвідношення для цієї послідовності має вигляд

$$\begin{cases} a_m = u, \\ a_n = f(n, p, a_{k-1}), n \geq m + 1. \end{cases}$$

Тоді для того, щоб отримати a_N, необхідно замінити наведений вище алгоритм на такий

```
a = u
for n in range(m + 1, N + 1):
    a = f(n, p, a)
```

який, отриманий заміною стартового значення у інструкції **range** на значення `m + 1`.

Приклад 3.16. Складемо програму обчислення суми:

$$S_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Зазначимо, що задане співвідношення має сенс тільки для $n \geq 1$. Складемо рекурентне співвідношення. Помічаємо, що на відміну від попереднього прикладу, кожен член послідовності S_n є сумою елементів вигляду $1/k$, де k змінюється від 1 до n. Отже, для побудови рекурентного співвідношення знайдемо різницю двох сусідніх членів послідовності S_n. Для $n \geq 2$

$$S_n - S_{n-1} = 1/n$$

Підставляючи у вихідне співвідношення $n = 1$, отримаємо $S_1 = 1$. Отже, рекурентне співвідношення для послідовності S_n матиме вигляд:

$$\begin{cases} S_1 = 1 \\ S_n = S_{n-1} + \dfrac{1}{n},\ n \geq 2 \end{cases}$$

Аналогічно до попереднього прикладу, враховуючи, що нумерація членів послідовності починається з 1, а не з нуля, отримаємо програму.

Лістинг 3.21. Використання рекурентних співвідношень.

```
def reqS(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    S = 1
    for n in range(2, N + 1):
        S += 1 / n
    return S

S = reqS(10)    # Знайдемо 10-й член послідовності
print("S =", S) # Виводимо результат на екран
```

Результат вищенаведеного коду

```
S = 2.9289682539682538
```

Приклад 3.17. Створимо програму обчислення суми

$$S_n = \sum_{i=1}^{n} 2^{n-i} i^2,\ n \geq 1.$$

Спочатку складемо рекурентне співвідношення для заданої послідовності. Підставляючи $n = 1$, отримаємо $S_1 = 1$. Щоб отримати вираз для загального члена, розкриємо суму для $n \geq 2$

$$S_n = 2^n \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} + \frac{n^2}{2^n} \right) =$$

$$= 2 \cdot 2^{n-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} + \frac{n^2}{2^n} \right) =$$

$$2 \cdot 2^{n-1} \left(\frac{1^2}{2^1} + \frac{2^2}{2^2} + \ldots + \frac{(n-1)^2}{2^{n-1}} \right) + 2 \cdot 2^{n-1} \frac{n^2}{2^n} = 2 \cdot S_{n-1} + n^2$$

Отже, рекурентне співвідношення для буде мати вигляд

$$\begin{cases} S_1 = 1, \\ S_n = 2 \cdot S_{n-1} + n^2, \ n \geq 2. \end{cases}$$

і відповідно програма для знаходження 10-го члена послідовності

Лістинг 3.22. Використання рекурентних співвідношень.

```
def req(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    S = 1
    for n in range(2, N + 1):
        S = 2 * S + n ** 2
    return S

print("S(10) = ", req(10))
```

Рекурентні співвідношення старших порядків

Нехай $\{a_n : n \geq 0\}$ деяка послідовність дійсних чисел. m – деяке натуральне число більше за одиницю.

Тоді

Означення 3.4. Послідовність $\{a_n : n \geq 0\}$ називається заданою **рекурентним співвідношенням m-го порядку**, якщо

$$\begin{cases} a_0 = u, \ a_1 = v, \dots, a_{m-1} = w, \\ a_n = f(n, p, \ a_{n-1}, \dots, a_{n-m},), \ n \geq m \end{cases}$$

де $u, v, \dots, w$ – задані числові сталі, p – деякий сталий параметр або набір параметрів, f – функція, задана аналітично у вигляді арифметичного виразу, що складається з операцій доступних для виконання з допомогою мови програмування.

Найпоширенішим прикладом послідовності заданої рекурентним співвідношенням 2-го порядку є послідовність чисел Фібоначчі. Перші два члени цієї послідовності дорівнюють одиниці, а кожен наступний член є сумою двох попередніх

$$\begin{cases} F_0 = 1, \ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \ n \geq 2 \end{cases}$$

Як і у випадку рекурентного співвідношення першого порядку, маючи рекурентне співвідношення можна знайти який завгодно член послідовності.

$$F_0 = 1,\ F_1 = 1$$
$$F_2 = F_1 + F_0 = 1 + 1 = 2$$
$$F_3 = F_2 + F_1 = 2 + 1 = 3$$
$$F_4 = F_3 + F_2 = 3 + 2 = 5$$
$$F_5 = F_4 + F_3 = 5 + 3 = 8$$
$$F_6 = F_5 + F_4 = 5 + 3 = 13$$

Для обчислення елементів послідовності, заданої рекурентним співвідношенням вищого порядку, застосовується інший підхід ніж для співвідношень першого порядку.

Алгоритм наведемо на прикладі співвідношення 3-го порядку. Нехай послідовність a_n задана рекурентним співвідношенням

$$\begin{cases} a_0 = u,\ a_1 = v, a_2 = w, \\ a_n = f(n, p,\ a_{n-1},, a_{n-2}, a_{n-3}),\ n \geq 3 \end{cases} \tag{3.1}$$

Тоді, після виконання коду

```
a3 = u  # a3 - змінна для (n-3)-го члену послідовності
a2 = v  # a2 - змінна для (n-2)-го члену послідовності
a1 = w  # a1 - змінна для (n-1)-го члену послідовності
for n in range(3, N + 1):
    # Обчислення наступного члену
    a = f(n, p, a1, a2, a3)
    # Зміщення змінних для наступних ітерацій
    a3 = a2
    a2 = a3
    a1 = a
```

у змінних a і a1 буде міститися a_N, у змінній a2 – a_{N-1} , а в змінній a3 – a_{N-2}.

Вправа 3.2. Доведіть вищенаведене твердження використовуючи метод математичної індукції.

Звернемо увагу на той факт, що для обчислення членів послідовності заданої рекурентним співвідношенням першого порядку не потрібно жодних додаткових змінних – лише змінна у якій обчислюється поточний член послідовності. Для рекурентних співвідношень старших порядків, крім змінної, у якій обчислюється поточних член послідовності, необхідні ще додаткові змінні, кількість яких дорівнює порядку рекурентного співвідношення.

Приклад 3.18. Знайдемо N-й член послідовності Фібоначі.

Розв'язок. Як було зазначено раніше послідовність чисел Фібоначчі F_n може бути задана рекурентним співвідношенням

$$\begin{cases} F_0 = 1,\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2},\ n \geq 2 \end{cases}$$

Оскільки послідовність Фібоначчі задана рекурентним співвідношенням другого порядку, то для того, щоб запрограмувати обчислення її членів, необхідно три змінних. Модифікувавши наведений вище алгоритм для обчислення відповідного члена послідовності заданої рекурентним співвідношенням третього порядку на випадок рекурентного співвідношення другого порядку, отримаємо програму

Лістинг 3.23. Обчилення послідовності чисел Фібоначчі.

```
def fib(N):
    """ Знаходження елементів послідовності Фібоначчі
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    f2 = 1
    f1 = 1
    for n in range(2, N + 1):
        f = f1 + f2
        f2 = f1
        f1 = f
    return f1

print("fib(10) = ", fib(10))
```

Кількість додаткових змінних, що явно використовуються у алгоритмі, у програмі Python можна скоротити використовуючи операцію пакування-розпакування кортежів. Наприклад, після виконання такого коду для послідовності заданої рекурентним співвідношенням (3.1)

```
a3 = u  # a3 - змінна для (n-3)-го члену послідовності
a2 = v  # a2 - змінна для (n-2)-го члену послідовності
a1 = w  # a1 - змінна для (n-1)-го члену послідовності
for n in range(3, N + 1):
    # Обчислення наступного члену послідовності з
    # одночасним зсувом значень змінних
    a3, a2, a1 = a2, a1, f(n, p, a1, a2, a3)
```

у змінній a1 буде міститися a_N, у змінній a2 – a_{N-1} , а в змінній a3 – a_{N-2}.

Саме цей підхід рекомендується використовувати у програмах, оскільки він значно спрощує розуміння алгоритму і зменшує ймовірність помилки.

Змінимо програму для прикладу 3.18**,**

Лістинг 3.24. Обчилення послідовності чисел Фібоначчі.

```
def fib(N):
    """ Знаходження елементів послідовності Фібоначчі
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    f2 = 1
    f1 = 1
    for n in range(2, N + 1):
        f2, f1 = f1, f1 + f2
    return f1

print("fib(10) = ", fib(10))  # 10-й член послідовності Фібоначчі
```

Приклад 3.19. Скласти програму для обчислення визначника порядку n:

$$D_n = \begin{vmatrix} 5 & 3 & 0 & 0 & \dots & 0 & 0 \\ 2 & 5 & 3 & 0 & \dots & 0 & 0 \\ 0 & 2 & 5 & 3 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 2 & 5 \end{vmatrix}.$$

Легко обчислити, що

$$D_1 = 5;$$
$$D_2 = \begin{vmatrix} 5 & 3 \\ 2 & 5 \end{vmatrix} = 19.$$

Розкладаючи для всіх $n \geq 3$ визначник D_n по першому рядку отримаємо рекурентне співвідношення

$$D_n = 5D_{n-1} - 6D_{n-2}, n \geq 3.$$

Тоді, згідно з вищенаведеним алгоритмом, програма для знаходження N-го члена послідовності D_n буде мати вигляд

Лістинг 3.25. Обчилення послідовності чисел Фібоначчі.

```
def D(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    d2 = 5   # 1-й член послідовності
```

```
    d1 = 19  # 2-й член послідовності
    for n in range(3, N + 1):
        d2, d1 = d1, 5 * d1 - 6 * d2
    return d1

N = int(input("N = "))
print("D_%d = %d" % (N, D(N)))
```

Системи рекурентних співвідношень

Вищенаведена теорія рекурентних співвідношень легко узагальнюється на системи рекурентних співвідношень, якщо вважати, що послідовності у означеннях вище є векторними.

Розглянемо системи рекурентних співвідношень на прикладах

Приклад 3.20. Опишемо програму для обчислення N-го члена послідовності, що визначається сумою

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!}, \ , \ n \geq 0.$$

Розкриваючи суму побачимо, що для всіх $n \geq 1$

$$S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{n-1}}{(n-1)!} + \frac{x^n}{n!} = S_{n-1} + \frac{x^n}{n!}$$

Отже послідовність S_n визначається рекурентним співвідношенням

$$\begin{cases} S_0 = 1, \\ S_n = S_{n-1} + \dfrac{x^n}{n!}, \ n \geq 1 \end{cases}$$

Позначимо

$$a_n := \frac{x^n}{n!}, \ n \geq 0.$$

У прикладі 3.15 для цієї послідовності було отримано рекурентне співвідношення. Тоді для вихідної послідовності S_n система рекурентних співвідношень матиме вигляд

$$\begin{cases} S_0 = 1, \ a_0 = 1 \\ a_n = \dfrac{x}{n} a_{n-1}, \ n \geq 1, \\ S_n = S_{n-1} + a_n, \ n \geq 1. \end{cases}$$

Отже, програма для знаходження N-го члена послідовності S_n буде мати вигляд

Лістинг 3.26. Системи рекурентних співвідношень.

```
def rec(N, x):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :param x: Параметр
    :return:  Знайдений член послідовності.
    """
    a = 1
    S = 1
    for n in range(1, N+1):
        a = x / n * a
        S = S + a
    return S

N = int(input("N = "))
x = float(input("x = "))
S = rec(N, x)
print("S(%d, %f) = %f", (N, x, S))
```

Приклад 3.21. Опишемо програму для обчислення суми

$$S_n = \sum_{k=0}^{n} a^k b^{n-k}$$

Рекурентне співвідношення можемо побудувати двома способами.

Спосіб 1. Очевидно, що $S_0 = 1$. Розкриваючи суму і групуючи доданки аналогічно до прикладу 3.17, отримаємо

$$\begin{cases} S_0 = 1, \\ S_n = b \cdot S_{n-1} + a^n, \qquad n \geq 1. \end{cases}$$

Введемо позначення $x_n = a^n$. Запишемо для послідовності $\{x_n : n \geq 0\}$ рекурентне співвідношення:

$$\begin{cases} x_0 = 1, \\ x_n = a \cdot x_{n-1}, \qquad n \geq 1. \end{cases}$$

Таким чином, отримаємо систему рекурентних співвідношень

$$\begin{cases} S_1 = x_1 = 1, \\ x_n = a \cdot x_{n-1}, \qquad n \geq 1. \\ S_n = b \cdot S_{n-1} + x_n, \end{cases}$$

Спосіб 2. Легко бачити, що послідовність S_n можна зобразити у вигляді

$$S_n = \frac{a^{n-1} - b^{n-1}}{a - b}, \qquad n \geq 1$$

Тоді система рекурентних співвідношень буде мати вигляд

$$\begin{cases} x_1 = a, y_1 = b, \\ x_n = a \cdot x_{n-1}, \\ y_n = b \cdot y_{n-1}, \\ S_n = \dfrac{x_n - y_n}{a - b}, \end{cases} \qquad n \geq 1.$$

Програма для знаходження N-го члена послідовності S_n, заданого рекурентним співвідношенням, котре отримано першим способом, має вигляд:

Лістинг 3.27. Системи рекурентних співвідношень.

```
def rec(N, a, b):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :param a: Параметр
    :param b: Параметр
    :return:  Знайдений член послідовності.
    """
    S = x = 1
    for n in range(1, N + 1):
        x = a * x
        S = b * S + x

N = int(input("N = "))
a = float(input("a = "))
b = float(input("b = "))
print(rec(N, a, b))
```

Приклад 3.22. Обчислимо суму, задану рекурентним співвідношенням

$$S_n = \sum_{k=1}^{n} \frac{a_k}{2^k},$$

де $a_1 = a_2 = a_3 = 1, a_k = a_{k-1} + a_{k-3}, k \geq 4$.

Звернемо увагу на те, що послідовність a_k задана рекурентним співвідношенням третього порядку. Введемо допоміжну послідовність $b_k = 2^k, k \geq 0$, для якої рекурентне співвідношення буде мати вигляд $b_1 = 1, b_k = 2b_{k-1}, k \geq 1$.

Тоді, поєднуючи алгоритми для визначення відповідних членів послідовностей, заданих рекурентними співвідношеннями першого і третього порядків, отримаємо програму

Лістинг 3.28. Системи рекурентних співвідношень.

```
def rec(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """

    a1 = a2 = a3 = 1  # Одночасна ініціалізація кількох
                      # змінних одним значенням
    b = 1
    S = 1 / 2
    # обчислення перших трьох членів послідовності S
    for k in range(1, min(4, N + 1)):
        b = 2 * b
        S = S + 1 / b

    for n in range(4, N + 1):
        b = 2 * b
        a3, a2, a1 = a2, a1, a1 + a3
        S = S + a1 / b

    return S

N = int(input("N = "))
print(rec(N))
```

Приклад 3.23. Обчислити суму, задану рекурентним співвідношенням

$$S_n = \sum_{k=0}^{n} \frac{a_k}{1 + b_k},$$

де

$$\begin{cases} a_0 = 1, \\ a_k = a_{k-1} b_{k-1}, \end{cases} \quad \begin{cases} b_0 = 1, \\ b_k = a_{k-1} + b_{k-1}, \end{cases} \quad k \geq 1.$$

Послідовності $\{a_k\}$ і $\{b_k\}$ задані рекурентним співвідношеннями першого порядку, проте залежність перехресна. Використаємо по одній допоміжній змінній для кожної з послідовностей.

Тоді, програма для знаходження N-го члена послідовності S_n:

Лістинг 3.29. Системи рекурентних співвідношень.

```
def rec(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    S = 0.5
    a = b = 1
    for n in range(1, N + 1):
        a, b = a * b, a + b
        S = S + a / (1 + b)
    return S

N = int(input("N = "))
print(rec(N))
```

Приклад 3.24. Обчислити добуток, заданий рекурентним співвідношенням

$$P_n = \prod_{k=0}^{n} \frac{a_k}{3^k},$$

де $a_0 = \ a_1 = 1, a_2 = 3, a_k = a_{k-3} + \frac{a_{k-2}}{2^{k-1}}, k \geq 3.$

Розв'язок. Послідовність $\{a_k\}$ задана рекурентним співвідношенням третього порядку. Тоді добуток P_n обчислюється за допомогою рекурентного співвідношення

$$\begin{cases} P_2 = 1/9, \\ P_k = P_{k-1} \cdot a_k / z_k, \qquad k \geq 3, \end{cases}$$

де z_k – k-й степінь числа 3, визначений рекурентним співвідношенням

$$\begin{cases} z_2 = 9, \\ z_k = 3z_{k-1}, \qquad k \geq 3. \end{cases}$$

Передбачивши змінну `t` для обчислення членів послідовності $\{t_k = 2^{k-1} : k \geq 3\}$, отримаємо програму

Лістинг 3.30. Системи рекурентних співвідношень.

```
def rec(N):
    """ Знаходження елементів послідовності використовуючи
        рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
    :return:  Знайдений член послідовності.
    """
    P = 1.0 / 9.0
    z = 9
    t = 2
    a2 = a3 = 1
    a1 = 3
    for n in range(3, N + 1):
        z = z * 3
        t = t * 2
        a3, a2, a1 = a2, a1, a3 + a2 / t
        P = P * a1 / z
    return P

N = int(input("N = "))
print(rec(N))
```

Рекурентні співвідношення та генератори

Досі, у більшості випадків, ми для визначення шуканого елемента послідовності заданого рекурентним співвідношенням користувалися функціями. Проте, для обчислення членів послідовностей заданих рекурентними співвідношеннями зручніше користуватися генераторами, які ми розглянули у цьому параграфі. Це пов'язано з тим, що при обчисленні членів послідовності нам доступний не лише фінальний шуканий член послідовності, але й усі проміжні члени послідовності. У лістингу 3.18 наведений приклад генератора для обчислення членів послідовності Фібоначчі. Визначимо ще генератор ще для однієї послідовності.

Приклад 3.25. Опишемо генератор, що будує послідовність задану в умові прикладу 3.16, використовуючи отримане рекурентне співвідношення.

Лістинг 3.31. Використання генератора для побудови послідовності заданої рекурентним співвідношенням.

```
def gen(N):
    """ Генератор для побудови елементів послідовності
        використовуючи рекурентні співвідношення
    :param N: Номер члена послідовності, що необхідно знайти
```

```
    """
    S = 1 # повернення 1-го члена
    yield S
    for n in range(2, N + 1):
        S += 1 / n
        yield S  # повернення n-го члена

# Використання генератора
for el in gen(10):
    print(el)
```

Результатом роботи вищенаведеної програми, що використовує генератор буде

```
1
1.5
1.8333333333333333
2.083333333333333
2.283333333333333
2.4499999999999997
2.5928571428571425
2.7178571428571425
2.8289682539682537
2.9289682539682538
```

Відшукання членів послідовності, що задовольняють умову

Досі ми будували програми, що знаходять значення члену послідовності за його номером. Проте, часто постає задача, коли потрібно знайти найперший член послідовності, що задовольняє певну умову. У такому разі цикл **for** по діапазону значень заміняється циклом з умовою **while**. Умова у цьому циклі є запереченням до умови, яка визначає коли потрібно припинити обчислення членів послідовності.

Розглянемо приклади

Приклад 3.26. Для довільного натурального $N \geq 2$ знайти найменше число вигляду 3^k, де k – натуральне, таке, що $3^k \geq N$.

Розв'язок. Розглянемо послідовність $a_k = 3^k, k \geq 0$. Легко бачити, що її можна задати рекурентним співвідношенням першого порядку

$$\begin{cases} a_0 = 1, \\ a_k = 3a_{k-1}, \qquad k \geq 1. \end{cases}$$

Отже, враховуючи, що послідовність a_k строго зростаюча, щоб виконати завдання задачі необхідно обчислювати члени послідовності a_k в циклі використовуючи вищенаведене рекурентне співвідношення, доки не знайдемо перший такий, що

$a_k \geq N$. Відповідно умова у циклі, буде запереченням до $a_k \geq N$, тобто $a_k < N$. Далі очевидним чином маємо програму

Лістинг 3.32. Відшукання членів послідовності, що задовольняють умову.

```
N = int(input("N = "))

a = 1
while a < N:
    a = a * 3

print(a)
```

Зауважимо, що для розв'язання таких задач зручніше користуватися нескінченними генераторам. Опишемо генератор, що будує вищенаведену послідовність.

Перепишемо попередній приклад, з використанням генератора.

Лістинг 3.33. Використання генератора.

```
def gen():
    """ Нескінченний генератор """
    a = 1
    yield a
    while True:
        a = a * 3  # обчилюємо наступний член послідовності
        yield a    # повертаємо поточний член послідовності
```

При використанні генератора, у цьому випадку, цикл **for**, заміняється циклом з умовою **while**.

Лістинг 3.33. Продовження. Використання "нескінченного" генератора.

```
N = int(input("N = "))
elem = gen()       # створюємо генератор
a = next(elem)     # генеруємо перший член послідовності
while a < N:
    a = next(elem) # генеруємо черговий член послідовності
print(a)           # виводимо знайдений член на екран
```

Хоча цей спосіб є більш громіздким, проте такий код простіший для сприйняття.

Приклад 3.27. Послідовність задана рекурентним співвідношенням

$$\begin{cases} x_0 = 1,\ x_1 = 0,\ x_2 = 1, \\ x_n = x_{n-1} + 2x_{n-2} + x_{n-3},\ n \geq 3. \end{cases}$$

Створимо програму для знаходження найбільшого члена цієї послідовності разом з його номером, який не перевищує число a.

Опишемо спочатку нескінченний генератор, що повертає елементи послідовності. У ньому x1, x2, x3 – змінні, що використовуються згідно до алгоритму для обчислення членів послідовності заданої вищенаведеним рекурентним співвідношенням третього порядку. Генератор буде повертати згенерований член послідовності (що буде міститися у змінній x1), а також номер цього елемента.

Лістинг 3.34. Використання генератора.

```
def gen():
    """ Нескінченний генератор """
    x3 = 1
    yield x3, 0  # повернення 0-го члена разом з номером
    x2 = 0
    yield x2, 1  # повернення 1-го члена разом з номером
    x1 = 1
    n = 2        # номер поточного члену послідовності
    yield x1, n  # повернення 2-го члена разом з номером
    while True:  # нескінченний цикл
        n += 1
        x3, x2, x1 = x2, x1, x1 + 2 * x2 + x3
        yield x1, n
```

Запишемо кілька перших членів заданої послідовності

$$1, 0, 1, 2, 4, 9, 19, 41, 88$$

Звернемо увагу на те, що ця послідовність є зростаючою. Тоді, для того, щоб знайти найбільший член цієї послідовності, що не перевищує задане число a, необхідно обчислювати члени цієї послідовності, доки не знайдемо перший такий член, що буде більшим за задане число a. Отже, умовою продовження циклу буде умова an < a. При цьому, член послідовності, що вимагається умовою задачі – елемент послідовності, що передує знайденому на останній ітерації циклу, елементу послідовності. Наприклад, якщо $a = 30$, то перший член послідовності, що більший за число 30 є 41, а відповідно шуканий згідно з умовою задачі член послідовності – той який йому передує, тобто 19. Отже будемо запам'ятовувати попередній член послідовності, для виведення результату, що вимагається у задачі.

Таким чином, маємо програму

Лістинг 3.34. Продовження. Використання "нескінченного" генератора.

```
# Використання генератора
gen_an = gen()
an, n = next(gen_an)          # перший член послідовності
prev_an, prev_n = an, n       # перший член послідовності
while an < a:
    prev_an, prev_n = an, n  # запам'ятовуємо поточний член
                # послідовності та його номер для результату
    an, n = next(gen_an)      # генеруємо черговий член
                              # послідовності та його номер

print ("x(%d) = %d <= %d = a" % (prev_n, prev_an, a))
```

Наведемо результат виконання програми для числа $a = 30$.

```
a = 30
x(6) = 19 <= 30 = a
```

Наближені обчислення границь послідовностей

Одне з важливих призначень рекурентних співвідношень це апарат для наближених обчислень границь послідовностей та значень аналітичних функцій.

Нехай задано послідовність $\{y_n : n \geq 0\}$, така, що $y_n \to y, n \to \infty$.

Означення 3.5. Під наближеним з точністю ε значенням границі послідовності y_n будемо розуміти такий член y_N послідовності, що виконується співвідношення

$$|y_N - y_{N-1}| < \varepsilon$$

Вищенаведене означення не є строгим з точки зору чисельних методів. У загальному випадку математичний апарат наближеного обчислення границь послідовностей та значень алгебраїчних функцій перебуває поза межами цього підручника і вимагає від читача додаткових знань з теорії чисельних методів [17].

Отже, згідно з наведеним вище означенням, для того щоб знайти значення границі послідовності потрібно обчислювати елементи послідовності доки виконується умова

$$|y_N - y_{N-1}| \geq \varepsilon$$

Розглянемо застосування зазначеного підходу на прикладах.

Приклад 3.28. Складемо програму наближеного обчислення золотого перетину c, використовуючи:

а) границю

$$\Phi = \lim_{n\to\infty} \frac{F_n}{F_{n-1}}$$

де F_n – послідовність чисел Фібоначчі;

б) ланцюговий дріб

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1+\ddots}}.$$

Розв'язок. Золотий перетин це число

$$\Phi \approx 1{,}6180339887..$$

яке має багато унікальних властивостей, причому не лише математичних. Це число можна зустріти як у різноманітних сферах діяльності людини (наприклад, у мистецтві, архітектурі, культурі) так і у оточуючому нас світі, зокрема фізиці, біології тощо. Для того, щоб переконатися у тому, що наш алгоритм працює правильно, скористаємося однією з властивостей золотого перетину, а саме:

$$\Phi - 1 = \frac{1}{\Phi}.$$

а) Розглянемо послідовність

$$\Phi_n = \frac{F_n}{F_{n-1}}, n \geq 1.$$

Знайдемо рекурентне співвідношення для c_n. Очевидно, що $c_1 = 1$, далі для $n \geq 2$ отримаємо

$$\Phi_n = \frac{F_n}{F_{n-1}} = \frac{F_{n-1} + F_{n-2}}{F_{n-1}} = 1 + \frac{1}{\frac{F_{n-1}}{F_{n-2}}} = 1 + \frac{1}{\Phi_{n-1}}.$$

б) Розглянемо послідовність

$$\Phi_n = 1 + \frac{1}{1 + \frac{1}{1+\ddots + \frac{1}{1}}},$$

що містить $n-1$ риску дробу. Очевидно, що для цієї послідовності рекурентне співвідношення буде таким же, як у пункті а).

Напишемо програму, що знаходить наближене з точністю ε значення границі послідовності Φ_n. Використаємо змінну `current` для обчислення поточного члену послідовності Φ_n і змінну `prev`, у якій будемо запам'ятовувати попередній член Φ_{n-1} цієї послідовності.

Тоді програма має вигляд

Лістинг 3.35. Обчислення золотого перетину.

```
eps = 0.0000000001  # точність
prev = 0            # попередній член послідовності
current = 1         # поточний член послідовності

while abs(current - prev) >= eps:
    prev = current  # запам'ятовуємо поточний член послідовності
    current = 1 + 1 / current # обчислюємо наступний член

print("Φ =", current)  # Виводимо значення золотого перетину

# Перевірка результату згідно з властивостями
print("Φ - 1 =", current - 1)
print("1 / Φ =", 1 / current)
```

Наведемо результат виконання програми.

```
Φ = 1.618033988738303
Φ - 1 = 0.6180339887383031
1 / Φ = 0.6180339887543225
```

Приклад 3.29. За допомогою розкладу функції e^x в ряд Тейлора

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

обчислимо з точністю $\varepsilon > 0$ її значення для заданого значення x.

Розв'язок. Позначимо загальний член вищенаведеного ряду через a_n, а його часткову суму

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!},$$

Очевидно, що $S_n \to e^x$, $n \to \infty$.

У прикладі 3.20 було отримано, що послідовність S_n визначається системою рекурентних співвідношень

$$\begin{cases} S_0 = 1,\ a_0 = 1 \\ a_n = \dfrac{x}{n} a_{n-1},\ n \geq 1, \\ S_n = S_{n-1} + a_n,\ n \geq 1. \end{cases}$$

Відповідно до означення, наведеного вище і з огляду на рекурентне співвідношення, під наближеним значенням границі послідовності S_n будемо розуміти такий член S_N , що виконується співвідношення

$$|S_N - S_{N-1}| = |a_N| < \varepsilon$$

Отже, програма матиме вигляд

Лістинг 3.36. Обчислення експоненти дійсного числа.

```
eps = 0.0000000001  # точність

x = float(input("x = "))
a = 1
S = 1
n = 0
while abs(a) >= eps:
    n += 1
    a = x / n * a
    S = S + a

print("exp(%f) = %f" % (x, S))
```

Звернемо увагу на те, що на відміну від цикла **for**, цикл **while** не має вбудованого лічильника. Тому, оскільки нам необхідно враховувати у формулі номер члена послідовності, то ми задали змінну n, яка відіграє роль лічильника.

Наведемо результат виконання програми.

```
x = 1.0
exp(1.000000) = 2.718282
```

§4 СТВОРЕННЯ ВИКЛЮЧЕНЬ

Власні класи виключень використовують тоді, коли стандартних класів не вистачає для класифікації виключень. Тобто, якщо треба більш точно вказати, що джерело виключення – у створеній програмі або класі.

Щоб створити власне виключення, достатньо описати клас, який походить від стандартного класу `Exception` або його нащадків. У цьому класі треба реалізувати методи `__init__()` та `__str__()`. У `__init__()` визначають атрибути власного виключення, а у `__str__()`, – повідомлення, яке буде видаватись при виключенні.

Власні класи виключень також можуть мати свою ієрархію. Ієрархія доцільна у випадку, якщо, скажімо, деякий модуль (клас) може ініціювати різнотипні виключення, але потрібно показати, що всі вони походять саме з даного модуля (класу). Оскільки **`except`** реагує не тільки на вказане виключення, але й на його нащадків, деякі програми можуть обробляти тільки кореневе виключення модуля (класу), а інші – для більш точної класифікації – конкретні виключення, що є підкласами кореневого.

Приклад 4.1. Опишемо клас виключення для функції введення з клавіатури лише невід'ємних цілих чисел.

Лістинг 4.1. Опис нового класу виключення.

```
class InputPositiveIntException(Exception):

    def __init__(self, message, err_code, original, converted):
        """ Конструктор
        :param message: повідомлення
        :param err_code: код помилки
        :param original: введене з клавіатури значення
        :param converted: перетворене у ціле введене значення
        """
        super().__init__()
        self.message = message
        self.original_value = original
        self.converted_value = converted
        self.err_code = err_code

    def __str__(self) -> str:
        return str(self.message)
```

Тепер опишемо саму функцію введення з клавіатури лише невід'ємних цілих чисел. Ця функція буде використовувати вбудовану функцію **`input`**`()` та, залежно від її результату, генерувати виключення, якщо користувач уведе

некоректні дані.

Лістинг 4.1. Продовження. Функція введення невід'ємних цілих чисел.

```
def input_positive_int(*args, **kwargs):
    s = input(*args, **kwargs)
    try:
        i = int(s)
    except ValueError:
        raise InputPositiveIntException("Non integer", 1, s, None)
    if i < 0:
        raise InputPositiveIntException("Non positive", 2, s, i)
    return i
```

Використаємо описану вище функцію для введення послідовності цілих невід'ємних чисел. Введення будемо здійснювати доти доки користувачем не буде введено рядок "exit".

Лістинг 4.1. Продовження. Застосування описаної функції.

```
while True:
    try:
        a = input_positive_int("Enter integer ('exit' to finish) = ")
        print(a)
    except InputPositiveIntException as e:
        if e.err_code == 1 and e.original_value == "exit":
            break
        print(e)
```

Приклад 4.2. Опишемо клас `ProtectedDictInt`, що є словником, у якому ключами можуть бути лише цілі числа та заборонена операція зміни значення за ключем (крім випадку додавання нової пари ключ-значення). Оформимо цей клас як обгортку навколо вбудованого типу даних `dict`.

Для обробки заборонених ситуацій створимо клас виключення, що

- є нащадком класу `KeyError`;
- ініціюється, якщо здійснюється спроба використання ключа, що не є цілим числом;
- ініціюється якщо відбувається спроба змінити у словнику значення за заданим ключем;
- генерує різні повідомлення у кожній із заборонених ситуацій.

Спочатку опишемо клас виключення, який назвемо `ProtectedDictIntError`.

Лістинг 4.2. Опис нового класу виключення.

```
class ProtectedDictIntError(KeyError):

    # Сталі для зазначення коду помилки
    NON_INTEGER_KEY = 0
    MISSED_KEY = 1
    CHANGE_VALUE = 2

    def __init__(self, err_code, message):
        """ Конструктор
        :param err_code: код помилки
        :param message: повідомлення
        """
        super().__init__()
        self.err_code = err_code
        self.message = message

    def __str__(self) -> str:
        return "ProtectedDictIntError: " + str(self.message)
```

Як бачимо цей клас місить три сталих (статичних поля), для ідентифікації типу (за кодом) помилки. Вхідними параметрами конструктора буде код помилки та повідомлення, що містить інформацію про помилку.

Тепер опишемо клас `ProtectedDictInt`, у якому перевантажимо оператор `[]` для доступу до даних словника по ключу для читання та запису.

Лістинг 4.2. Продовження. Опис класу `ProtectedDictInt`.

```
class ProtectedDictInt:
    def __init__(self):
        self.__dict = {}  # Словник

    def __setitem__(self, key, value):
        """ Метод, що перевантажує оператор [] для запису
        :param key: Ключ
        :param value: Значення
        """
        if not isinstance(key, int):  # якщо ключ не ціле число
            raise ProtectedDictIntError(
            ProtectedDictIntError.NON_INTEGER_KEY, "NON_INTEGER_KEY")

        if key in self.__dict:  # Якщо ключ міститься у словнику,
                                # забороняємо зміну значення
            raise ProtectedDictIntError(
                  ProtectedDictIntError.CHANGE_VALUE, "CHANGE_VALUE")
        self.__dict[key] = value
```

```
    def __getitem__(self, key):
        """ Метод, що перевантажує оператор [] для читання
        :param key: Ключ
        :return: значення, що відподає ключу
        """
        if key not in self.__dict:  # ключ не міститься у словнику
            raise ProtectedDictIntError(
                        ProtectedDictIntError.MISSED_KEY, "MISSED_KEY")
        return self.__dict[key]

    def __str__(self):
        return str(self.__dict)
```

§5 АБСТРАКТНІ КЛАСИ

5.1. Абстрактні класи

Означення 5.1. Клас називається абстрактним, якщо створення його екземплярів немає сенсу і, відповідно, не передбачається.

Ідея абстрактного класу полягає у тому, що часто для роботи необхідний не повністю готовий клас, а певна «заготовка». Ця заготовка частково має деякий функціонал, проте використовувати її безпосередньо не можна, а потрібно доопрацювати. Наприклад, клас `Pet`, описаний у лістингу 2.3 містить метод `voice()`, виклик якого є коректним лише для конкретної тварини, але не для абстрактної домашньої тварини. Хоча метод `voice()` і містив реалізацію у класі `Pet`, проте, вона не мала жодного сенсу, оскільки передбачається, що цей метод буде заміщено у нащадках класу `Pet`. Як правило, такі методи у абстрактних класах містять порожню реалізацію і називаються абстрактними.

Означення 5.2. Абстрактний метод (також використовується термін чистий віртуальний метод) – метод класу, реалізація якого відсутня у класі.

Абстрактний метод необхідно обов'язково замістити у класах-нащадках. Фактично, єдине призначення абстрактного методу це декларація методу, тобто визначення його сигнатури (ім'я, список формальних параметрів, значення, що повертається).

Наприклад, у випадку нашого класу `Pet`, абстрактний метод `voice()` фактично декларує, що кожна домашня тварина може подавати голос, а вже, як саме, буде залежати від того, до якого конкретного класу-нащадка буде належати ця домашня тварина.

Отже, можемо виділити два суттєвих моменти при роботі з абстрактними класами

- Потрібно заборонити створення екземплярів такого класу.
- Абстрактний клас вимагає доопрацювання під конкретні умови його використання.

Доопрацювання класу полягає у тому, що він є базовим для конкретних класів. У кожному з цих конкретних класів необхідно реалізувати абстрактні методи – методи, які були задекларовані проте не могли бути конкретно описані у базовому класі. Так, наприклад, ми описували конкретні класи `Dog`, `Cat` та `Parrot` у кожному з яких реалізовували конкретний метод `voice()`.

У Python клас вважається абстрактним, якщо він містить хоча б один абстрактний метод. Створення абстрактного класу є досить складною задачею з архітектурного боку програмного продукту. Необхідність використовувати абстрактний клас для тієї чи іншої задачі може проявитися не зразу. Необхідно провести детальний аналіз задачі і набору класів, що дозволить прийняти рішення.

З технічної сторони, створення абстрактного класу задача не складніша за створення звичайного класу. Щоб створити абстрактний клас, необхідно описати клас на основі метакласу `ABCMeta` з модуля `abc`. Всі абстрактні методи позначаються за допомогою декоратора **`@abstractmethod`** (що також описаний у модулі `abc`) та містять порожню реалізацію.

```
from abc import ABCMeta, abstractmethod

class AbstractClass(metaclass=ABCMeta):

    @abstractmethod     # позначає абстрактний метод
    def method(self):   # абстрактний метод
        pass            # порожня реалізація
```

Виправимо клас `Pet`, щоб він став абстрактним.

Лістинг 5.1. Оспис абстрактного класу.

```
from abc import ABCMeta, abstractmethod

class Pet(metaclass = ABCMeta):
    def __init__(self, name):
        """ Конструктор
        :param name: Кличка тварини
        """
        self._name = name  # приватне поле - кличка тварини

    @abstractmethod
    def voice(self): # абстрактний метод
        pass         # порожня реалізація
```

Якщо спробувати створити екземпляр класу Pet,

```
pet = Pet("Pet")
```

то під час виконання програми отримаємо помилку

```
    pet = Pet("Pet")
TypeError: Can't instantiate abstract class Pet with abstract methods voice
```

Як бачимо інтерпретатор не дає можливості створити екземпляр абстрактного класу з абстрактним методом **voice**().

Проведемо інший експеримент. Створимо клас-нащадок на базі класу Pet, проте абстрактний метод **voice**() лишимо незаміщеним.

Лістинг 5.1. Продовження.

```
class Cat(Pet):
    pass

c = Cat("Кузя")
```

запустимо програму:

```
    c = Cat("Кузя")
TypeError: Can't instantiate abstract class Cat with abstract methods voice
```

Як бачимо, інтерпретатор знову не дозволив створити, екземпляр класу, у якому ми не змістили абстрактний метод. Отже, через механізм абстрактних класів здійснюється контроль, щоб ненароком не забути про реалізацію абстрактного методу у нащадках.

Розглянемо як приклад реалізацію одного з важливих шаблонів об'єктно-орієнтованого проектування – Відвідувача. Цей шаблон проектування застосовується якщо необхідно відділити певний алгоритм від об'єктів для яких цей алгоритм має бути виконаний. Він дозволяє додавати нові операції у існуючі структури без змін самих структур, що у свою чергу дозволяє уникнути процесу «засмічування» цими операціям самих класів.

Шаблон Відвідувач складається з кількох частин, що взаємодіють між собою:

- Відвідувач (`Visitor`) – абстрактний клас, що визначає дію над кожним класом конкретних елементів. Містить абстрактний метод `visit(obj : ConcreteElement)`.
- Конкретний відвідувач (`ConcreteVisitor`) – нащадок класу `Visitor` – заміщує абстрактний метод `visit(obj : ConcreteElement)` конкретною реалізацією.
- Елемент (`Element`) – абстрактний клас, що визначає метод `accept(visitor : Visitor)`, який отримує об'єкт відвідувача як аргумент.
- Конкретний елемент (`ConcreteElement`) – конкретний клас, нащадок класу Елемент, що заміщує метод `accept(visitor : Visitor)`, який отримує об'єкт відвідувача як аргумент.
- Клієнт (`Client`) – структура елементів, надає високорівневий інтерфейс, що дозволяє відвідувачу опрацьовувати елементи.

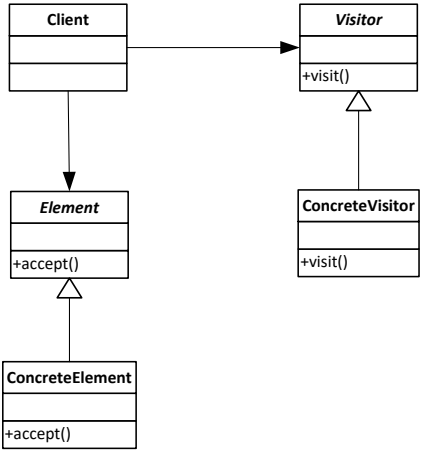


Рисунок 5.1. Діаграма класів шаблону Відвідувач

Зв'язок екземпляра класу Елемент і Відвідувача відбувається таким чином:

- Для елемента викликається метод `accept()` фактичним параметром якого є конкретний відвідувач (елемент приймає відвідувача)
- В свою чергу для цього відвідувача викликається метод `visit()` параметром якого є сам елемент.
- Відвідувач, у методі `visit()` здійснює перевірку до якого класу належить елемент і залежно від цього здійснює відповідний перелік дій.

В оточуючому нас житті існує безліч прикладів використання зв'язків, що є прототипом шаблону проектування Відвідувач. Наприклад, людину може відвідати лікар, що поправить її здоров'я, працівник бухгалтерії, що нарахує та видасть заробітну платню, а інший відвідувач грабіжник, цю платню може викрасти. Аналогічно домашнього улюбленця може відвідати ветеринар або тренувати дресирувальник. У цьому випадку людина або домашній улюбленець це приклади конкретних елементів, котрих відвідує лікар, ветеринар тощо. Навряд чи необхідно пояснювати, що результати взаємодії різних класів будуть призводити до різних результатів. Наприклад, лікар може вилікувати людину у той час як ветеринар цього зробити не може.

Приклад 5.1. Реалізуємо клас Pet, як абстрактний елемент та його нащадки Cat та Dog, як конкретні елементи, екземпляри яких можуть відвідувати відвідувачі.

Лістинг 5.2. Клас Pet та його нащадки Cat та Dog.

```
class Pet(metaclass=ABCMeta):
    """ Абстрактний клас Домашня тварина """

    def __init__(self, name) -> None:
        self.name = name       # ім'я домашньої тварини
        self.isHealthy = True  # чи здорова тварина
        self.isHungry = False  # чи голодна тварина
        self.isHappy = True    # чи щаслива тварина

    @abstractmethod
    def accept(self, visitor):
        """ Абстрактний метод, що використовується для реалізації
            шаблону проектування Відвідувач
        :param visitor: відмідувач
        """
        pass

    def __str__(self):
        return ("Name: " + self.name + ";\nHappy = " + str(self.isHappy) +
                ";\nHungry = " + str(self.isHungry) +
                ";\nHealthy = " + str(self.isHealthy))

    def treat(self):
        """ Лікувати тварину """
        self.isHealthy = True

    def hurt(self):
        """ Ображати тварину """
        self.isHealthy = False
        self.isHappy = False

class Cat(Pet):
    """ Клас Кіт """

    def __str__(self):
        return "== Cat ==\n" + super().__str__()

    def accept(self, visitor):
        visitor.visit(self)

    def feed(self):
        """ Годувати кота """
        self.isHungry = False
```

```
    def play(self):
        """ Пограти з котом """
        self.isHappy = True
        self.isHungry = True

class Dog(Pet):
    """ Клас Собака """

    def __init__(self, name) -> None:
        super().__init__(name)
        self.skills = 0

    def __str__(self):
        return ("== Dog ==\n" + super().__str__() +
                ";\nSkills = " + str(self.skills))

    def accept(self, visitor):
        visitor.visit(self)

    def getBone(self):
        """ Отримати кістку """
        self.isHungry = False

    def walk(self):
        """ Вигуляти собаку """
        self.isHappy = True
        self.isHungry = True

    def training(self):
        """ Тренувати собаку """
        self.skills += 10
        self.isHappy = False
```

Як бачимо, всі класи, базовий клас оголошує метод `accept()`, тоді як його нащадки, заміщують оголошений метод однаковою реалізацією. Цього вимагає класична реалізація шаблону Відвідувач. Проте, можемо зауважити, що, враховуючи особливості мови програмування Python, реалізацію цього методу можна було б здійснити у базовому класі.

Опишемо абстрактний клас `Visitor` та конкретні його реалізації – `Veterinarian` (Ветеринар, може лікувати тварину), `Scoundrel` (Негідник, ображає тварин), `DogTrainer` (Дресирувальник собак), `Master` (Господар, вміє годувати тварин) та `Child` (Дитина, грається з домашніми тваринами)

Лістинг 5.3. Клас `Visitor` та конкретні його реалізації.

```
class Visitor(metaclass=ABCMeta):
    """ Базовий клас Відвідувач """

    @abstractmethod
    def visit(self, pet):
        pass

class Veterinarian(Visitor):
    """ Клас лікар - вміє лікувати людей """

    def visit(self, pet):
        pet.treat()

    def __str__(self):
        return "Visitor = Veterinarian"

class Scoundrel(Visitor):
    """ Клас Негідник - ображає тварин """

    def visit(self, pet):
        pet.hurt()

    def __str__(self):
        return "Visitor = Scoundrel"

class DogTrainer(Visitor):
    """  Клас Дресирувальник домашніх тварин """

    def visit(self, pet):
        if isinstance(pet, Dog):
            pet.training()
        else:
            print("I can train only dogs")

    def __str__(self):
        return "Visitor = DogTrainer"

class Master(Visitor):
    """ Клас Господар - вміє годувати тварину """

    def visit(self, pet):
        if isinstance(pet, Cat):
            pet.feed()
```

```
        elif isinstance(pet, Dog):
            pet.getBone()

    def __str__(self):
        return "Visitor = Master"

class Child(Visitor):
    """ Клас Дитина - вміє розважати тварин """
    def visit(self, pet):
        if isinstance(pet, Cat):
            pet.play()
        elif isinstance(pet, Dog):
            pet.walk()

    def __str__(self):
        return "Visitor = Child"
```

Як бачимо з коду, кожен з відвідувачів аналізує екземпляр якого класу він відвідує та, залежно від цього, здійснює певні дії, з переліку який він вміє робити.

Нарешті, опишемо клієнта, що використовує обидва класи. У нашій реалізації клієнт створює дві домашні тварини, що є екземплярами наведених вище класів Dog та Cat та всі типи відвідувачів. Після цього для кожної з тварин застосовується операція `accept()` – прийняти тварину та виведення інформації про тварину.

Лістинг 5.4. Клієнт.

```
dog = Dog("Rex")
cat = Cat("Kuzya")

visitors = [Scoundrel(), Veterinarian(), DogTrainer(), Master(), Child()]

for visitor in visitors:
    print(visitor)
    cat.accept(visitor)
    print(cat)
    print()

for visitor in visitors:
    print(visitor)
    dog.accept(visitor)
    print(dog)
    print()
```

Вправа 5.1. Зобразіть діаграму класів, для наведеної вище реалізації шаблону Відвідувач.

Наведений вище шаблон Відвідувач можна реалізувати без застосування операції **isinstance**() визначення типу об'єкта якого буде відвідувати відвідувач. При реалізації цього шаблона ця операція слугувала механізмом який моделює механізм перевантаження методів, що є наявним у інших мовах програмування. Отже, щоб обійти застосування функції визначення типу, у класі відвідувачі можна оголосити методи відвідування орієнтовані під конкретного елемента.

Наведемо схематично реалізацію цього шаблона, а саме лише методи `visit()` та `accept()`, що будуть відрізнятися від наведеної вище реалізації.

Лістинг 5.5. Клас Pet та його нащадки Cat та Dog.

```
class Pet(metaclass=ABCMeta):
    ...
    @abstractmethod
    def accept(self, visitor):
        """ Абстрактний метод, що використовується для реалізації
        шаблону проектування Відвідувач
        :param visitor: відмідувач
        """
        pass
    ...

class Cat(Pet):
    """ Клас Кіт """
    ...
    def accept(self, visitor):
        visitor.visitCat(self)
    ...

class Dog(Pet):
    """ Клас Собака """
    ...
    def accept(self, visitor):
        visitor.visitDog(self)
    ...
```

Тепер наведемо класи, що стосуються Відвідувача. Наведемо реалізацію класу Відвідувач на прикладі класів Господар та Дитина, наведених вище.

Лістинг 5.6. Абстрактний клас Відвідувач

```
class Visitor(metaclass=ABCMeta):
    """ Базовий клас Відвідувач """
```

```
    @abstractmethod
    def visitCat(self, pet):
    """ Абстрактний метод відвідати Кота """
        pass

    @abstractmethod
    def visitDog(self, pet):
    """ Абстрактний метод відвідати Собаку """
        pass

class Master(Visitor):
    """ Клас Господар - вміє годувати тварину """

    def visitCat(self, pet):
        pet.feed()

    def visitDog(self, pet):
        pet.getBone()

class Child(Visitor):
    """ Клас Дитина - вміє вміє розважати тварин """

    def visitCat(self, pet):
        pet.play()

    def visitDog(self, pet):
        pet.walk()
```

Використання описаного таким чином шаблону повністю повторює наведену у лістингу 5.4 програму.

5.2. Інтерфейси

Абстрактні класи і інтерфейси – близькі за змістом поняття, хоча при детальнішому розгляді виявляється не так все і однозначно. Поруч із абстрактними класами, інтерфейси встановлюють взаємо обов'язки між елементами програмної системи.

Означення 5.3. Інтерфейс (також називають інтерфейсний клас) – абстрактний клас, що містить лише абстрактні методи.

Як наслідок, можемо зробити висновок, що інтерфейси не містять даних (тобто полів, статичний у тому ж числі), а всі їхні методи необхідно реалізувати у нащадках. Про клас-нащадок, кажуть, що він реалізує інтерфейс. Інтерфейси дозволяють використовувати множинне наслідування і в той же час вирішити проблему ромбовидного наслідування – оскільки даних немає, а всі методи абстрактні і

підлягають заміщенню, то відпадає проблема неоднозначності при виборі методу або зверненні до полів даних.

Питання коли використовувати інтерфейси, а коли абстрактні класи не однозначне, проте існує кілька характеристик, які однозначно кажуть, яку сутність використовувати.

Абстрактний клас використовують коли існує ряд характеристик, що об'єднують групу у єдину ієрархію. Наприклад, клас `Pet` – домашня тварина об'єднує групу (і є базою для) класів `Cat`, `Dog`. Всі нащадки мають спільні риси, які можна помістити у базовому класі, а у нащадках реалізувати лише відмінності.

Інтерфейс використовують коли потрібно відобразити лише зовнішню схожість поведінки об'єктів класів, для яких навіть не можна (або складно) побудувати ієрархію, які власне не знаходяться у якихось родинних зв'язках. Наприклад, метод `listen()` може бути визначений (по різному) для найрізноманітніших класів – `Human`, `Pet`, `MobilePhone`, `Microphone` тощо. Серед них не можливо виділити предка, оскільки їх всіх об'єднує лише властивість, що вони можуть слухати. Тоді можна оголосити інтефейс `Listener` (слухач), що буде мати абстрактний метод `listen()`, а кожен з класів перехованих вище буде реалізовувати цей інтерфейс по своєму. Отже, фактично основне призначення інтерфейсних класів – це гарантувати додаткові конкретно визначені можливості для екземплярів різнотипових класів!

Реалізація інтефейсів у Python здійснюється за допомогою модуля `zope.interfaces` який потрібно додатково встановлювати. Проте можна користуватися підходом абстрактний класів.

Інтерфейси на UML-діаграмах класів

На UML діаграмах класів, інтерфейси позначають подібно до класів. Перед назвою обов'язково вказується слово «interface», а також немає розділу для полів. Клас, який реалізує інтерфейс, з'єднується з інтерфейсом лінією зі стрілкою, як при наслідуванні, проте використовується пунктирна лінія

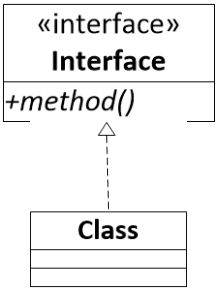


Рисунок 5.2. Інтерфейси на діаграмах класів

Приклад 5.2. Як відомо діагностувати можна найрізноманітніші об'єкти –

автомобілі, здоров'я людей чи тварин, техніку. Отже, можна описати інтерфейс `Diagnosable`, який буде містити (абстрактний) метод `diagnose()` – діагностування стану об'єкту. Опишемо базові класи `Car` та `Human`. На основі вищенаведених базових класів опишемо конкретні класи `DiagnosableCar` та `DiagnosableHuman` відповідно, кожен з яких, крім цього, буде реалізовувати інтерфейс `Diagnosable`. Отже, будь-який екземпляр класів `DiagnosableCar` та `DiagnosableHuman` можна буде діагностувати (причому ця діагностика буде залежати від того, до якого класу належить об'єкт). Таким чином отримаємо таку ієрархію класів

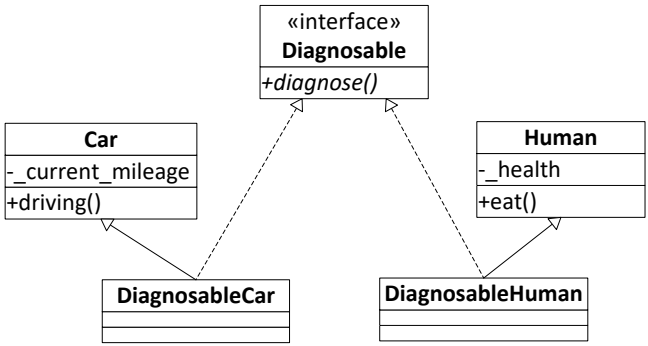


Рисунок 5.3. Діаграма класів

Запрограмуємо отриману ієрархію класів:

Лістинг 5.7. Інтерфейс «Діагностований»

```
class Diagnosable(metaclass=ABCMeta):

    @abstractmethod
    def diagnose(self):
        """ Абстрактний метод діагностувати """
        pass
```

Лістинг 5.8. Базовий клас Автомобіль

```
class Car:

    def __init__(self, resource=100000):
        """ Конструктор
        :param resource: ресурс автомобіля до капітального ремонту
```

```
        """
        self._resource = resource
        self._current_mileage = 0  # поточний пробіг автомобіля

    def driving(self):
        self._current_mileage += 15000
```

Лістинг 5.9. Базовий клас Людина

```
class Human:

    def __init__(self):
        """ Конструктор """
        self._health = 100  # Рівень здоров'я у відсотках

    def eat(self, food):
        if food == "junk food":          # шкідливий продукт
            self._health -= 20           # погіршує здоров'я
        elif food == "healthy food":     # корисний продукт
            self._health += 20           # покращує здоров'я
        self._health = self._health if self._health > 0 else 0
        self._health = self._health if self._health < 100 else 100
```

Тепер реалізуємо кожен з наведених вище інтерфейсів для кожного з класів Автомобіль та Людина.

Лістинг 5.10. Діагностований Автомобіль

```
class DiagnosableCar(Car, Diagnosable):

    def diagnose(self):
        if self._current_mileage >= self._resource:
            return "Your car requires major repairs!"
        rest = self._resource - self._current_mileage
        rest /= self._resource
        rest *= 100
        return "Rest {}% of resource".format(rest)
```

Лістинг 5.11. Діагностована Людина

```
class DiagnosableHuman(Human, Diagnosable):
```

```
    def diagnose(self):
        if self._health <= 0:
            return "A junk food has killed you!"
        elif self._health == 100:
            return "You have a great health!"
        else:
            return "Please, visit a doctor "
```

Лістинг 5.12. Використання класів, що реалізують інтерфейси.

```
c = DiagnosableCar()
h = DiagnosableHuman()

for i in range(5):
    c.driving()
    h.eat("junk food")

print(c.diagnose())
print(h.diagnose())
```

Для демонстрації ще одного прикладу застосування інтерфейсів, реалізуємо важливий шаблон об’єктно-орієнтованого проектування – Спостерігач (`Observer`). Цей шаблон реалізує механізм, який дозволяє об'єктам одного класу отримувати сповіщення про зміну стану об'єктів інших класів і тим самим спостерігати за ними. Клас, за яким спостерігають називаються Суб’єктом (`Subject`), а клас який спостерігає – Спостерігачем (`Observer`).

Для отримання повідомлень Спостерігач має підписатися (зареєструватися) на зміну стану Суб'єкта.

При реалізації шаблону Спостерігач зазвичай використовуються такі класи:

- `Subject` - абстрактний клас або інтерфейс, який визначає методи для додавання, видалення та сповіщення спостерігачів;
- `Observer` – інтерфейс, за допомогою якого спостерігач отримує сповіщення;
- `ConcreteSubject`, `ConcreteObserver` – конкретні реалізації наведених вище класів.

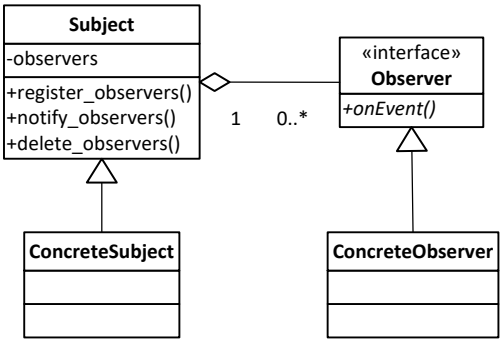


Рисунок 5.4. Діаграма класів шаблону Спостерігач

Поле `_observers` класу містить список підписаних спостерігачів. Методи `register_observers()` та `delete_observers()` реєструють та видаляють спостерігача, відповідно. Далі, екземпляр класу `Subject` (або `ConcreteSubject`, залежно від реалізації) при зміні свого стану викликає метод `notify_observers()`, у якому для кожного підписаного спостерігача викликається метод `onEvent()`. Таким чином відбувається оповіщення спостерігача про зміну стану об'єкту за яким відбувається спостереження.

Приклад 5.3. Реалізуємо шаблон Спостерігач у такому вигляді – об'єкти класу `FileReader` читають текстові файли та передають кожен прочитай рядок файлу усім підписаним спостерігачам.

Абстрактний клас `Subject` визначає методи для додавання та сповіщення спостерігачів.

Лістинг 5.13. Абстрактний клас `Subject`

```
class Subject(metaclass=ABCMeta):
    """ Абстрактний клас, що визначає методи для додавання та
        сповіщення спостерігачів """

    def __init__(self):
        self._observers = []  # список підписників-спостерігачів

    def register_observers(self, observer: Observer):
        """ Підписує спостерігача на отримання повідомлень
        :param observer: спостерігач  """
        self._observers.append(observer)
```

```
    def notify_observers(self, data):
        """ Інформує спостерігачів, надсилаючи кожному з них дані
        :param data: дані, що надсилаються спостерігачу.
        """
        for observer in self._observers:  # кожному підписаному спостерігачу
            observer.onDataReceive(data)  # надсилає дані
```

Клас `FileReader`, що є нащадком класу `Subject` описує методи читання з файлу та оповіщення спостерігачів.

Лістинг 5.13. Продовження. Клас `FileReader`.

```
class FileReader(Subject):
    """ Файловий читач - конкретна реалізація класу Subject.
        Його екземпляри читають заданий текстовий файл та надсилають
        підписаним спостерігачам по одному рядки цього файлу """

    def __init__(self, filename):
        super().__init__()
        self._filename = filename  # ім'я файлу

    def read(self):
        with open(self._filename) as f:
            for line in f:
                self.notify_observers(line.strip())
```

Інтерфейс `Observer` декларує єдиний абстрактний метод `onDataReceive()`, реалізація якого здійснюється у конкретних спостерігачах `FilePrinter` та `WordCounter` – перший з яких виводить усі отримані від суб'єкту рядки файлу на екран, другий підраховує кількість слів у текстовому файлі.

Лістинг 5.14. Інтерфейс `Observer`

```
class Observer(metaclass=ABCMeta):
    """ Інтефейс - Слухач """

    @abstractmethod
    def onDataReceive(self, line):
        pass
```

Лістинг 5.15. Реалізація інтерфейсу `Observer`

```
class FilePrinter(Observer):
    """ Реалізація інтерфейсу Observer
        Виводить отримані рядки на екран """

    def onDataReceive(self, line):
        print(line)

class WordCounter(Observer):
    """ Реалізація інтерфейсу Observer
        Підраховує кількість слів у текстовому файлі
        Тут під словом будемо розуміти послідовність літер, що відокремлена
        одним або кількома символами пропуску. """

    def onDataReceive(self, line):
        global words_number
        words = line.split()
        words_number += len(words)
```

Лістинг 5.15. Продовження. Головна програма.

```
filereader = FileReader("input.txt")

filereader.register_observers(FilePrinter())
filereader.register_observers(WordCounter())

words_number = 0
filereader.read()
print(words_number)
```

5.3. Класи домішки

Означення 5.4. Домішки (англ. mixin) – це класи, які додають до іншого класу визначену функціональність через механізм множинного наслідування.

Концепція домішків будується на ідеї чіткого розмежування властивостей і методів для сутностей – тобто даних і алгоритмів. Наприклад, припустимо, ми розглядаємо клас `Автомобіль`. Тоді для нього можемо використовувати метод `move()`. Аналогічно цей метод можемо використовувати для класів `Велосипед` або `Мотоцикл`. Тоді, якщо реалізація цього методу буде однаковою для всіх трьох класів, то, можемо виділити

цю поведінку в окремий клас-домішок, що дозволить уникнути дублювання коду, а всі три класи будуть мати цей домішок у ролі одного з базових класів.

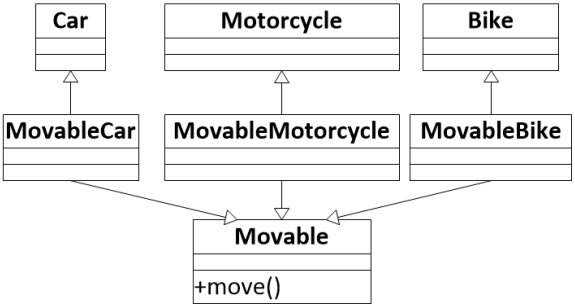


Рисунок 5.5.

Фактично домішки є різновидом множинного наслідуванні і синтаксично нічим від нього не відрізняються. Проте потрібно пам'ятати деяку їхню специфіку: домішки не призначені для породження самостійних екземплярів класів – вони слугують для того, щоб додати визначену функціональність будь-якому іншому класу.

Домішки є близькою концепцією до інтерфейсів, проте спід пам'ятати, що інтерфейс надає лише специфікацію поведінки, без її реалізації, у той час, як домішки містять і повну реалізацію. Крім того, як ми пам'ятаємо, інтерфейси використовуються для неспоріднених класів, серед яких інколи не можливо встановити ієрархію. Відповідно, домішки застосовуються для того, щоб розширити функціонал подібних класів, що (навіть якщо не можна побудувати ієрархію) мають багато спільних рис.

Розглянемо приклад. У попередніх параграфах ми використовували клас `Pet` та його нащадки `Cat`, `Dog` та `Parrot`. Будемо вважати, що кожен з цих класів, крім поля «кличка тварини», містить поля «кількість лап» та «кількість бліх».

Лістинг 5.16.

```
from abc import ABCMeta, abstractmethod

class Pet(metaclass = ABCMeta):
    def __init__(self, name, legs, fleas):
        """ Конструктор """
        self._name = name   # кличка тварини
        self._legs = legs   # кількість лап
        self._fleas = fleas # кількість бліх
```

Опишемо клас домішок `Logged`, що буде містити метод, що виводить повну інформацію про конкретну тварину.

Лістинг 5.17. Клас домішок.

```
class Logged:
    def log(self):
        print("========= Клас:", self.__class__.__name__)
        print("Кличка тварини:", self._name)
        print("Кількість лап: ", self._legs)
        print("Кількість бліх:", self._fleas)
```

Очевидно, що сам по собі цей клас не має сенсу, оскільки його метод `log()` намагається використати поля `self._name`, `self._legs` та `self._fleas` яких у ньому немає.

Застосуємо цей домішок для наших класів `Cat`, `Dog` та `Parrot`. Для цього опишемо класи `LoggedCat`, `LoggedDog` та `LoggedParrot`, що будуть містити у якості базових класів клас-домішок і відповідно клас `Cat`, `Dog` або `Parrot`.

Лістинг 5.18. Застосування домішок.

```
# Фактично додаємо до класів Cat, Dog, Parrot
# функціонал класу Loged
class LoggedCat(Logged, Cat):
    pass

class LoggedDog(Logged, Dog):
    pass

class LoggedParrot(Logged, Parrot):
    pass

c = LoggedCat("Кузя", 4, 2)
d = LoggedDog("Барбос", 4, 1)
p = LoggedParrot("Попка", 2, 0)

c.log()  # Викликаємо метод класу Loged
d.log()  # Викликаємо метод класу Loged
p.log()  # Викликаємо метод класу Loged
```

Результат виконання наведеного вище коду буде таким

```
========= Клас: LoggedCat
Кличка тварини: Кузя
Кількість лап:  4
Кількість бліх: 2
```

```
========= Клас: LoggedDog
Кличка тварини: Барбос
Кількість лап:  4
Кількість бліх: 1
========= Клас: LoggedParrot
Кличка тварини: Попка
Кількість лап:  2
Кількість бліх: 0
```

§6 МЕТАПРОГРАМУВАННЯ

Мова програмування Python допускає різні парадигми програмування (структурне, процедурне, модульне тощо), проте в її основі лежить об'єктно-орієнтований підхід. Фактично Python є типовим представником мов об'єктно-орієнтованого сімейства та базується на простій, проте дуже потужній об'єктній моделі. Будь-які дані у цій мові, навіть літерали базових типів даних, таких як числа або символи, є об'єктами. Об'єктами є представники вбудованих структур даних таких як списки, кортежі, словники, об'єктами є функції та методи описані програмістом, об'єктами є імпортовані модулі. Власне й самі класи у Python є об'єктами. При цьому програма Python може оперувати не лише з даними у тому сенсі, який ми вкладали у цей термін раніше. З точки зору мови програмування Python не існує жодної концептуальної різниці між числом, екземпляром класу (створеного програмістом), функцією, імпортованим модулем чи класом – програма Python сприймає всі ці об'єкти як дані. Саме така гнучка об'єктна модель забезпечує можливість програмам на мові Python (під час власного виконання) модифікувати себе чи інші програми, що є одним з видів метапрограмування.

6.1. Поняття про метапрограмування

Означення 6.1. Метапрограмування – це парадигма програмування, відповідно до якої будь-яка інформаційна система (тобто програма) є набором даних, стан та поведінку якого можна динамічно змінювати залежно від результатів взаємодії з іншими системами.

Іншими словами, метапрограмування – це побудова програм, які конструюють інші програми, у тому числі змінюють або доповнюють себе, під час виконання, залежно від значень даних, впливу інших об'єктів, дій користувача тощо. Фактично ці програми сприймають інші програми як дані, подібно до звичайних змінних, які можна не лише читати чи виконувати, але й створювати чи модифікувати.

Взагалі кажучи, термін метапрограмування це досить широке поняття, до якого належить, у тому ж числі, генерація вихідного коду зовнішніми засобами. Прикладом такого метапрограмування, є генерація мовою Python коду на JavaScript для виконання його у браузері користувача. У цьому параграфі ми зосередимося на тому розділі метапрограмування де мова йде про генерування або зміну програми написаної на мові програмування Python засобами Python.

Означення 6.2. Самопереробний код – програмний прийом, при якому програма створює або змінює частину свого програмного коду під час виконання.

За часом проведення модифікації метод поділяється на:

- модифікацію при ініціалізації – проводиться один раз, перед запуском змінюваного коду
- модифікацію на льоту – зміна стану програми під час виконання

Самопереробний код використовує такі концепції програмування, як інтроспекція та рефлексія, описані нижче.

Найпершим та найпростішим прикладом метапрограмування на Python, який ми розглянемо, є застосування функції `eval()`, параметром якої є рядковий літерал, що містить код на мові програмування Python.

Лістинг 6.1. Використання функції `eval()`.

```
x = 1
eval('print(x)')
y = 3
eval('print(x + y)')
```

Результатом роботи вищенаведеного коду буде виведення на екран такого

```
1
4
```

Загальний синтаксис функції `eval()` такий

```
eval(expression, globals=None, locals=None)
```

де

- `expression` – рядковий літерал, який Python інтерпретує як вираз.
- `globals` – необов'язковий параметр, що містить словник глобальних об'єктів Python (змінні, функції, класи), які можуть бути використані у виразі `expression`. Словник `globals` співставляє фрагменти виразу з об'єктами програми, а саме ключ – рядковий літерал, що міститься у виразі `expression`, значення – об'єкт у програмі.
- `locals` – необов'язковий параметр, що містить словник локальних об'єктів Python, які можуть бути використані у виразі `expression`.

```
print(eval('sqrt(25)')) # Буде породжена помилка, оскільки eval не знає
                        # нічого про глобальну функцію sqrt
```

А от такий фрагмент коду виконається коректно.

```
import math
print(eval('sqrt(25)', {"sqrt": math.sqrt}, {}))
```

6.2. Рефлексія та інтроспекція

Одними з основних механізмів метапрограмування є інтроспекція та рефлексія, перший з яких дозволяє аналізувати класи або їхні екземпляри, у той час як другий – модифікувати їхню структуру або поведінку.

Інтроспекція

Під час виконання програми часто постає необхідність володіти інформацією про дані якими оперує програма, причому, не лише про значення, але й зокрема, до якого типу вони належать, чи є вони екземплярами відповідного класу, яка їхня внутрішня структура тощо.

Означення 6.3. Інтроспекція (англ. type introspection) — можливість визначити тип і структуру об'єкта під час виконання програми.

У тому чи іншому вигляді, інтроспекція наявна у більшості сучасних мов програмування. Раніше ми вже зіштовхувалися з одним із способів використання інтроспекції, а саме коли використовувати функцію

```
isinstance(obj, Class)
```

Нагадаємо, ця функція повертає `True`, якщо об’єкт `obj` є екземпляром класу `Class`.

У мові програмування Python інтроспекція підтримується на рівні синтаксису і має значно ширші можливості, аніж просто визначати чи є об’єкт екземпляром певного класу. Наприклад, застосування інтроспекції дозволяє дізнатися ієрархію класів успадкованого об’єкту, якими атрибутами він володіє тощо.

Функція dir()

Найбільш загальним методом інтроспекції є використання функції `dir()`. Ця функція повертає перелік усіх атрибутів об’єкту.

Лістинг 6.2. Використання функції `dir()`.

```
class Pet:
    def __init__(self, name):
        self._name = name

    def getName(self):
        return self._name

p = Pet("Kuzya")
print(dir(p))
```

Результатом виконання наведеного коду буде

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_name',
'getName']
```

Як бачимо, функція **dir**() повертає список не лише членів класу, описаних програмістом, таких як поле `_name` та метод `getName`, але й перелік вбудованих (тобто успадкованих від базового класу `object`, що є предком для всіх породжених класів) атрибутів.

Спеціальні атрибути-поля

Атрибут `__class__` повертає посилання на клас, екземпляром якого є поточний об'єкт. В результаті виконання такого коду

```
print(p.__class__)
```

для екземпляру класу створеного у лістингу 6.2 буде

```
<class '__main__.Pet'>
```

Атрибут `__bases__` поверне список усіх безпосередніх батьків поточного класу.

Лістинг 6.3. Клас Cat, нащадок класу Pet.

```
class Cat(Pet):
    legs = 4
    ears = 2

print(Cat.__bases__)
```

Результатом буде посилання на батьківський клас Pet.

```
(<class 'source.P_06.L1.Pet'>,)
```

Звернемо увагу читача, що атрибут __bases__ застосовується не до екземпляру класу, а безпосередньо до самого класу.

Атрибут __dict__ містить словник усіх користувацьких атрибутів-полів об'єкта та їхні значення.

Лістинг 6.3. Продовження.

```
c = Cat("Kuzya")
print(Cat.__dict__)
print(c.__dict__)
```

Результатом буде

```
{'__module__': '__main__', 'legs': 4, 'ears': 2, '__doc__': None}
{'_name': 'Kuzya'}
```

Як бачимо, значення цього атрибута можна отримувати як для екземпляру класу, так і для самого класу. Як бачимо статичні поля legs та ears будуть належати до самого класу Cat, а не його екземпляру. Власне як і клас Cat не містить поля _name, що належить (разом з його конкретним значенням) конкретному екземпляру класу.

Визначення типу об'єкту та приналежність об'єкта до класу

Тип даних — характеристика, яку явно чи неявно надано об'єкту. Він визначає множину припустимих значень, формат їхнього збереження, розмір виділеної пам'яті та набір операцій, які можна робити над даними. У Python 3 усі класи є типами даних, оскільки автоматично наслідуються від базового класу object.

Зауваження. Для Python 2 це, взагалі кажучи не так – типами є лише ті класи, для яких явно вказано наслідування від класу `object`.

Для визначення типу об'єкту у Python використовується функція **`type`**`()` фактичним аргументом якої є посилання на екземпляр класу.

Лістинг 6.3. Продовження. Визначення типу функцією `type()`.

```
print(type(c))
```

Результатом буде

```
<class '__main__.Cat'>
```

Раніше ми розглянули спеціальний атрибут `__class__`, що повертає посилання на клас, до якого належить конкретний екземпляр. Враховуючи сказане на початку цього пункту, можна зробити висновок, що у Python 3 використання функції **`type`**`()` та звернення до атрибуту `__class__` екземпляру класу будуть призводити до одного й того ж результату.

Розглянемо тепер детальніше функцію **`isinstance`**`()`, яка дає відповідь на питання чи є поточний об'єкт екземпляром конкретного класу. На перший погляд може здатися, що використання інструкції

```
isinstance(obj, Class)
```

повністю еквівалентне такій інструкції

```
type(obj) == Class
```

проте це не так, оскільки функція **`type`**`()` повертає безпосередньо клас-еталон, у той час як **`isinstance`**`()` перевіряє реалізацію у об'єктів усього ланцюжка наслідування і якщо цей ланцюжок містить клас `Class`, то поверне **`True`**. Щоб переконатися у цьому розглянемо приклад застосування функцій **`type`**`()` та **`isinstance`**`()` до екземпляру класу `Cat` лістингу 6.3

Лістинг 6.3. Продовження. Методи `type()` та `isinstance()`.

```
print(isinstance(c, Pet))
print(type(c) == Pet)
```

Результатом буде

```
True
False
```

Такий результат пояснюється тим, що об'єкт c є не лише екземпляром класу Cat, а й екземпляром класу Pet через механізм наслідування, у той же час тип об'єкта є цілком конкретний – Cat. Це легко зрозуміти, якщо спроектувати механізми об'єктно-орієнтованого програмування на реальний світ – крім того що кіт є «Котом» (тобто екземпляром класу Cat), він є ще й «Домашнім улюбленцем» (тобто екземпляром класу Pet). Проте тип нашого улюбленця цілком конкретний – він «Кіт».

Перевірка наявності атрибутів

Для перевірки чи має клас або екземпляр певний атрибут (поле або метод), використовується функція **hasattr**(). Ця функція має два аргументи – перший екземпляр класу, що аналізується – другий рядковий літерал – ім'я атрибуту.

Перевіримо наявність атрибутів для екземпляру класу Cat описаного у лістингу 6.3.

Лістинг 6.4. Перевірка наявності атрибутів для об'єкту класу Cat.

```
c = Cat("Kuzya")

print(hasattr(c, "getName"))  # пошук методу getName
print(hasattr(c, "_name"))    # пошук поля _name
print(hasattr(c, "legs"))     # пошук статичного поля legs
print(hasattr(c, "unknown"))  # пошук атрибута unknown
```

Результатом буде

```
True
True
True
False
```

Функцію **hasattr**() можна застосовувати не лише до екземплярів класу, але й до самого класу. У цьому випадку вона буде давати відповідь на питання чи містить клас метод або статичне поле – тобто атрибут, що належить усьому класу, а не лише його екземпляру

Лістинг 6.5. Перевірка наявності атрибутів у класі Cat.

```
print(hasattr(Cat, "getName"))  # пошук методу getName
print(hasattr(Cat, "_name"))    # пошук поля _name
print(hasattr(Cat, "legs"))     # пошук статичного поля legs
```

Цілком очікувано, що результатом роботи цього коду буде

```
True
False
True
```

оскільки поле _name належить конкретному екземпляру класу – створюється у ньому і без нього не існує.

Рефлексія

Якщо інтроспекція дозволяє вивчити структуру об'єкту під час виконання програми, то рефлексія дозволяє комп'ютерній програмі під час її виконання маніпулювати цією структурою, наприклад, видаляти чи додавати нові поля чи методи до класу або конкретного екземпляру.

Означення 6.4. Рефлексія — це здатність комп'ютерної програми досліджувати і модифікувати власну структуру і поведінку під час виконання.

У Python рефлексія забезпечується трьома функціями:

- **getattr**() – дозволяє читати атрибут об'єкту або класу;
- **delattr**() – дозволяє видаляти атрибут у об'єкту або класу;
- **setattr**() – дозволяє встановлювати атрибут об'єкту або класу.

Кожна з наведених вище функцій **getattr**() та **delattr**(), як і функція **hasattr**(), має два аргументи – перший – екземпляр класу (або клас), що аналізується, другий рядковий літерал – ім'я атрибуту. Функція **setattr**(), має ще й третій параметр – посилання на атрибут, що встановлюється або додається.

Розглянемо детальніше кожну з цих функцій.

Функція **getattr**() повертає посилання на атрибут об'єкту або породжує виключення AttributeError, якщо такий атрибут відсутній у ньому.

Лістинг 6.6. Звернення до атрибутів екземпляру класу Cat.

```
c = Cat("Kuzya")

n = getattr(c, "_name")  # поле _name об'єкту c
print(n)

l = getattr(c, "legs")   # статичне поле legs об'єкту c
print(l)

method = getattr(c, "getName")          # метод getName об'єкту c
print(method())    # рефлексивний виклик методу getName об'єкту c
```

Результатом роботи наведеного вище коду буде такий вивід

```
Kuzya
4
Kuzya
```

Звернемо увагу читача на рефлексивний виклик методу **getName**(). Функція **getattr**() повертає не просто посилання на відповідний метод, а посилання на метод конкретного екземпляру. Якщо ж функцію **getattr**() застосовувати до класу, то вона поверне посилання на метод класу без конкретизації екземпляру. Щоб переконатися у цьому, розглянемо лістинг нижче

Лістинг 6.7. Звернення до атрибутів екземпляру класу Cat.

```
c = Cat("Kuzya")

method1 = getattr(Cat, "getName")  # метод getName класу Cat
method2 = getattr(c, "getName")    # метод getName об'єкту c

print(method1)
print(method2)
```

Результатом буде такий вивід

```
<function Pet.getName at 0x00F4D660>
<bound method Pet.getName of <source.P_06.L3.Cat object at 0x00F1F030>>
```

Як бачимо Python інтерпретує method1 як функцію, а method2 як метод, що належить конкретному об’єкту. Python інтерпретує методи отримані за допомогою рефлексії з класів як функції, першим аргументом яких є self – посилання на

екземпляр класу. Це означає, що ці методи можна застосовувати до конкретних об'єктів, передаючи першим параметром посилання на екземпляр. Наприклад, `method1` можна викликати для конкретного об'єкту таким чином

Лістинг 6.7. Продовження. Виклик методу класу для конкретного екземпляру класу.

```
print(method1(c))  # рефлексивний виклик методу getName об'єкту c
```

Функція **`delattr`**() видаляє атрибут у об'єкта (чи класу) або породжує виключення `AttributeError`, якщо атрибут відсутній у ньому чи видалення цього атрибута є неможливим.

Лістинг 6.8. Видалення атрибутів.

```
class A:

    static_property = 1  # статичне поле

    def __init__(self, prop):
        self.property = prop  # локальне поле

    def method(self):       # метод
        pass

a = A(1)
print(hasattr(a, "property"))
delattr(a, "property")
print(hasattr(a, "property"))
```

Результатом роботи програми буде вивід

```
True
False
```

Як бачимо, поле `property` було видалено з нашого об'єкту. Це поле `property` належало екземпляру класу. Постає питання, чи так само просто, можна видалити з об'єкту атрибути, що належать не конкретному екземпляру, а всьому об'єкту? Виявляється функція **`delattr`**() має ряд особливостей, основною з яких є те, що ця функція може видаляти лише ті атрибути, що безпосередньо належать екземпляру класу. Зокрема це означає, що якщо ми спробуємо у екземпляра класу видалити

метод або статичне поле, то інтерпретатор породить помилку (`AttributeError`). Таким чином, кожна з інструкцій наведених нижче породить помилку

Лістинг 6.8. Продовження.

```
delattr(a, "static_property")  #  породжується помилка AttributeError
delattr(a, "method")           #  породжується помилка AttributeError
```

оскільки, статичне поле `static_property` та метод `method` належать всьому класу, а не конкретному його екземпляру. Якщо ж застосуємо подібні інструкції до всього класу, вони відпрацюють коректно

Лістинг 6.8. Продовження.

```
delattr(A, "static_property") # видалення атрибуту static_property у класу A
delattr(A, "method")          # видалення атрибуту method у класу A

# перевірка наявності атрибутів у екземплярі класу A
print(hasattr(a, "static_property"))
print(hasattr(a, "method"))
```

Результатом буде вивід

```
False
False
```

який підтверджує видалення атрибутів.

Зауважимо, що аналогічна ситуація буде спостерігатися і при спробі видалити атрибут класу-предка. Розглянемо клас B, що є нащадком класу A, описаного у лістингу 6.8.

Лістинг 6.9. Видалення атрибуту батьківського класу.

```
class B(A):
    pass

delattr(B, "method")  # Спроба видалення атрибуту method у класу B
```

В результаті роботи наведеного вище коду буде породжена помилка

```
AttributeError: method
```

Вправа 6.1.. Особливості роботи функції не `delattr()` вичерпуються описаними результатами. Пропонуємо читачу самостійно дослідити роботу функції `delattr()` для заміщених методів при наслідуванні.

Розглянемо тепер функцію `setattr()`, призначення якої додати нові атрибути класу або його екземпляру.

Щоб додати нове поле у екземпляр класу, потрібно скористатися функцією `setattr()`, другим та третім параметром якої є ім'я атрибуту та його значення.

Лістинг 6.10. Додавання поля у об'єкт використовуючи рефлексію.

```
class A:
    pass

a = A()

setattr(a, "name", "World")  # додавання нового поля у екземпляр класу
print(a.name)                # використання щойно доданого поля
```

Результатом роботи цього коду буде вивід на екран

```
World
```

Зауважимо, що інструкція

```
setattr(a, "name", "World")  # додавання нового поля у екземпляр класу
```

наведена вище, еквівалентна стандартному механізму створення полів у екземплярах класів:

```
a.name = "World"
```

Необхідність у використанні функції `setattr()` постає, в першу чергу, саме під час метапрограмування, оскільки такий підхід дає можливість використовувати рядкові літерали для створення атрибутів у класах або їхніх екземплярах.

Щоб додати метод або статичне поле необхідно застосувати функцію `setattr()` не до конкретного екземпляру класу, а до всього класу в цілому. Продемонструємо це прикладі. Спочатку підготуємо метод, першим аргументом якого буде змінна (`self`), що у подальшому міститиме посилання на конкретний екземпляр класу

Лістинг 6.10. Продовження. Метод для додавання у клас

```
def prepeared_method(self):     # функція
    if hasattr(self, "name"):   # перевіряємо чи має клас атрибут name
        print("Hello, %s!" % self.name)
    else:
        print("Hello!")
```

Підготовлений таким чином метод можемо вставити у клас

Лістинг 6.10. Продовження. Додавання методу у клас та його використання

```
setattr(A, "method", prepeared_method)
a.method()  # Виклик методу доданого через рефлексію
```

Якщо наведеному вище коду передувала інструкція додавання поля name, з використанням рефлексії, то результатом роботи цього коду буде вивід.

```
Hello, World!
```

6.3. Декоратори

Одним з поширених застосувань метапрограмування у Python є створення декораторів. Декоратор це функція, що отримує об'єкт як параметр та додає йому певний функціонал або змінює його поведінку.

Означення 6.5. Декоратор – це функція, що дозволяє змінити поведінку об'єкта (отриманої у якості аргументу), не змінюючи самого об'єкту.

Декоратори для функцій

Як ми знаємо, функції це об'єкти Python. Відтак їх можна передавати у інші функції у ролі аргументів, а також функції можуть бути результатами роботи функцій. Цей факт використовується для створення спеціальних функцій, які називаються декораторами функцій. Як правило декоратори використовуються для того, щоб додати додаткові можливості функціям. Власне сенс застосування декоратора і полягає у тому, що вони можуть додавати однакову поведінку різним функціям. Наприклад, широко використовуються декоратори, що обчислюють час роботи функції, перевіряють коректність аргументів функцій, контролюють безпеку даних під час виконання функції тощо.

Щоб створити декоратор функції потрібно створити функцію, яка у якості аргументу буде отримувати функцію, модифікувати отриману функцію у тілі декоратора та повертати модифіковану функцію. Розглянемо такий приклад

Лістинг 6.11. Опис простого декоратора.

```
def decorator(function):

    # Створюємо нову функцію
    def decorated_function():
        print("Код, що буде виконано до виклику функції")
        function()  # Виклик функції, що декорується
        print("Код, що буде виконано після виклику функції")

    # Повертаємо декоровану функцію
    return decorated_function
```

Виникає запитання: як скористатися створеним декоратором? Визначимо функцію

```
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Задекоруємо її з допомогою декоратора `decorator`. Для цього необхідно змінній `seyHello` (що є посиланням на функцію у пам'яті) присвоїти результат виконання декоратора `decorator` над нею:

```
seyHello = decorator(seyHello)
```

Результатом виклику функції

```
seyHello()
```

буде

```
Код, що буде виконано до виклику функції
Усім вітання від функції seyHello!
Код, що буде виконано після виклику функції
```

Для спрощення розуміння коду, декорування функцій під час їхнього опису, можна здійснювати за допомогою оператора @. Наприклад, вищенаведені опис функції `seyHello` та її декорування

```
def seyHello():
    print("Усім вітання від функції seyHello!")

seyHello = decorator(seyHello)
```

можна замінити рівносильною їм синтаксичною конструкцією

```
@decorator
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Загальний опис декораторів

У попередньому прикладі декоратор `decorator` міг застосовуватися лише до функцій, з порожнім списком аргументів. Загальний шаблон декоратора, що може застосовуватися до функцій з будь-якою кількістю як позиційних так і ключових аргументів, має такий вигляд:

```
def decorator(function):
    # Створюємо нову функцію
    def decorated_function(*args, **kwargs):
        # Код, що буде виконано до виклику функції
        ...
        res = function(*args, **kwargs)  # Виклик функції
        # Код, що буде виконано після виклику функції
        ...
        return res
    # Повертаємо декоровану функцію
    return decorated_function
```

Приклад 6.1. Опишемо декоратор, що вимірює час виконання функції. Застосуємо його для перевірки часу обчислення чисел Фібоначчі з використанням нерекурсивного і рекурсивного варіантів.

Розв'язок. Опишемо спочатку декоратор. Тут буде використовуватися функція `clock()` з бібліотеки `time`, що повертає поточний системний час. Будемо зчитувати та запам'ятовувати час до початку виконання функції та після її виконання та виводити різницю на екран.

Лістинг 6.12. Декоратор для вимірювання часу виконання функції.

```
from time import clock # підключення функції clock
def benchmark(f):
```

```
    def _benchmark(*args, **kw):
        #вимірюємо час перед викликом функції
        current_time = clock()
        rez = f(*args, **kw)  #викликаємо f
        #обчилюємо різницю у часі
        dt = clock() - current_time
        print('Час виконання функції %1.5f сек' % dt)
        return rez
    return _benchmark
```

Опишемо функції `Fib1(n)` для нерекурсивного і `Fib2(n)` для рекурсивного варіантів обчислення n-го числа Фібоначчі, та задекоруємо їх за допомогою вищенаведеного декоратора `benchmark`. Оскільки декоратор діє на функцію, то для рекурсивного варіанту ми змушені використати допоміжну функцію `FibRecursive(n)`, яка буде реалізовувати рекурсію, а функція `Fib2` буде лише викликати функцію `FibRecursive(n)`.

Лістинг 6.13. Застосування декторатора.

```
@benchmark
def Fib1(n):
    F1 = F2 = 1
    for i in range(2, n + 1):
        F2, F1 = F1, F1 + F2
    return F1

def FibRecursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return FibRecursive(n - 1) + FibRecursive(n - 2)

@benchmark
def Fib2(n):
    return FibRecursive(n)
```

Виклик функцій нічим не відрізнятиметься від звичайного виклику недекорованих функцій:

```
N = 30
print(Fib1(N))
print(Fib2(N))
```

Проте, результатом роботи програми буде:

```
Час виконання функції 0.00001 сек
1346269
Час виконання функції 0.61396 сек
1346269
```

Вкладені декоратори

До функції може бути застосовано кілька декораторів одночасно. Їх вказують перед описом функції. Вказані декоратори застосовуються до функції послідовно, починаючи з найближчого до опису функції декоратора. Тому таке застосування декораторів називається їхнім вкладенням. Отже, якщо маємо код

```
@decorator1
@decorator2
def func():
    ...
```

то це означатиме, що до функції `func` спочатку застосовується декоратор `decorator2`, а вже потім до їхнього результату декоратор `decorator1`.

Декоратори для класів

Декоратор класу, це функція, що отримує клас як параметр, модифікує та повертає як результат модифікований клас:

```
def decorator(cls):
    """ Декоратор класу

    :param cls: Клас
    :return: Модифікований клас
    """

    # код модифікації класу
    ...
    return modified_cls
```

Застосування декоратора до класу може здійснюватися або стандартним чином через операцію присвоєння

```
class DecoratedClass:
    # опис класу

DecoratedClass = decorator(DecoratedClass)
```

або за допомогою оператора @ після якого йде ім'я декоратора, що застосовується до класу та, власне опис класу.

```
@decorator
class DecoratedClass:
    # опис класу
    ...
```

Для демонстрації створення та застосування декораторів класів, реалізуємо важливий шаблон об'єктно-орієнтованого проектування – Одинак (Singleton). Цей шаблон гарантує, що клас матиме тільки один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра.

Лістинг 6.14. Застосування декоратора.

```
def singleton(cls):
    """ Декоратор, що реалізує шабон проектування Одинак """

    instances = {}  # Словник класів, буде або порожнім,
                    # якщо ще не створено жодного екземпляру класу
                    # або міститиме рівно один екземпляр

    def getinstance():
        if cls not in instances:
            instances[cls] = cls()
        return instances[cls]

    return getinstance
```

Фактично описаний декоратор буде підміняти виклик механізму створення нового екземпляру класу викликом функції **getinstance**(), яка в свою чергу буде або створювати новий об'єкт, якщо екземплярів цього класу ще не було створено, або повертати раніше створений екземпляр класу.

Застосуємо декоратор до класу MyClass та переконаємося у тому, що створено лише один екземпляр цього класу.

Лістинг 6.14. Продовження. Застосування декоратора singleton до класу.

```
@singleton
class MyClass:
    ...
```

```
#  Спробуємо створити кілька екземплярів класу MyClass
obj1 = MyClass()
obj2 = MyClass()

#  Переконаємося, obj1 та obj2 це один і той же екземпляр класу
print(obj1 is obj2)
```

У цьому коді ми використали оператор **is**, що повертає істину у випадку, якщо лівий і правий операнди є одним і тим же об’єктом.

Список літератури та використані джерела

1. The Python Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/tutorial/index.html.
2. Навчальні матеріали: Python [Електронний ресурс] – Режим доступу до ресурсу: http://www.matfiz.univ.kiev.ua/pages/13.
3. Орлов С. А. Технологии разработки программного обеспечения. Разработка сложных программных систем [Текст] : учеб. пособие для вузов по направлению " Информатика и вычисл. техника" / Сергей Александрович Орлов. – СПб.: Питер, 2002. – 463 с.
4. Прохоренок Н. А. Python 3 и PyQt. Разработка приложений. / Николай Анатольевич Прохоренок. – СПб: БХВ-Петербург, 2012. – 704 с.
5. Васильев А. Н. Python на примерах. Практический курс по программированию / А. Н. Васильев. – СПб.: Наука и техника, 2016. – 432 с. – (Просто о сложном).
6. Python 3 для начинающих [Електронний ресурс] – Режим доступу до ресурсу: pythonworld.ru.
7. Креневич, А.П. С у задачах і прикладах : навчальний посібник із дисципліни "Інформатика та програмування" / А.П. Креневич, О.В. Обвінцев. – К. : Видавничо-поліграфічний центр "Київський університет", 2011. – 208 с.
8. Збірник задач з дисципліни "Інформатика і програмування" / Вакал Є.С., Личман В.В., Обвінцев О.В., Бублик В.В., Довгий Б.П., Попов В.В. -2-ге видання, виправлене та доповнене –К.: ВПЦ "Київський університет", 2006.– 94 с.
9. E-Olymp [Електронний ресурс] – Режим доступу до ресурсу: www.e-olymp.com.
10. Школа программиста [Електронний ресурс] – Режим доступу до ресурсу: http://acmp.ru/
11. Абрамов С.А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И. Задачи по программированию. –М.: Наука, 1988. – 224 с.
12. Златопольский Д.М. Сборник задач по программированию. – 2-е издание, переработанное и дополненное. – СПб.: БХВ-Петербург, 2007. –240 с.: ил.
13. Пильщиков В.Н. Сборник упражнений по языку Паскаль: Учебное пособие для вузов . –М.: Наука, 1989. –160 с.
14. Проскуряков И.В. Сборник задач по линейной алгебре. 11-е издание, стереотипное. – СПб.: Лань, 2008. –480 с.
15. Вирт Н. Систематическое программирование. Введение.–М.: Мир, 1977. –184 с.
16. Вирт Н. Алгоритмы + структуры данных = программы.–М.: Мир, 1985. –

406 с.

17. Калиткин Н.Н. Численные методы.– Наука, 1978. — 512 с.